Development and Application of Hypernetworks for Discretization-Independent Surrogate Modeling of Physical Fields

by

James T. Duvall

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Aerospace Engineering and Scientific Computing)
in the University of Michigan
2024

James T. Duvall

jamesduv@umich.edu

ORCID iD: 0000-0001-6398-8819

Table of Contents

List of Tables

List of Figures

List of Appendices

List of Symbols

$\mathbf{A}$	Vandermonde / adjacency matrix
$\mathbf{B}$	bias matrix
$\mathbf{b}$	bias vector
$\mathbf{D}$	degree matrix
$\mathbf{H}$	Hessian matrix hidden-state matrix
$\mathbf{h}$	hidden-state vector
$\mathbf{I}_p$	Identity matrix, dimension $p \times p$
$\mathbf{K}$	filter matrix
$\mathbf{L}$	graph Laplacian matrix
$\mathbf{m}_0$ / $\mathbf{m}_{\text{post}}$	prior /posterior mean vectors
$\mathbf{p}(\cdot)$	polynomial feature-generating function
$\mathbf{Q}$	discretized HFM solution, matrix form
$\mathbf{q}$	vector of HFM solution variables
$\mathbf{q}_N$	discretized HFM solution, flattened vector form
$\mathbf{q}''$	heat flux vector
$\mathbf{u}$	velocity vector
$\mathbf{W}$	weight matrix
$\mathbf{w}$	vector of weights
$\mathbf{x}$	vector of spatial coordinates
$\mathbf{x}'$	augmented vector of spatial coordinates
$\mathbf{z}$	latent vector
a_{ij}	anisotropic stress tensor
C_D	drag coefficient
C_L	lift coefficient

Co_η	Cournat number, Kolmogorov scale
c_p	specific heat capacity at constant pressure
c_v	specific heat capacity at constant volume
E_0	stagnation energy
e	specific internal energy
F_D	drag force
h	enthalpy
I	turbulence intensity
k	turbulent kinetic energy
k_B	Boltzmann constant
$k(\cdot,\cdot)$	covariance function / kernel
ℓ_0	largest eddy scale
M	molar mass
$m(\cdot)$	mean function
$N(\cdot)$	neural network
N_A	Avogadro's number
$n_{(\cdot)}$	dimension of subscript quantity $(\cdot)$
p/p_0	pressure/stagnation pressure
p_R	stagnation pressure ratio
R	computer performance in flops
R_{sp}	specific gas constant
R_u	universal gas constant
Re	Reynolds number
S	Sutherland constant
S_{ij}	strain rate tensor
T/T_0	temperature/stagnation temperature
t	time
u	x-component of velocity
u'_i	fluctuating velocity component
v	y-component of velocity

w	z-component of velocity
$\mathbf{\Gamma}$	Noise matrix
$\boldsymbol{\epsilon}$	vector of errors
$\boldsymbol{\lambda}$	vector of eigenvalues
$\boldsymbol{\mu}$	vector of design variables
$\mathbf{\Phi}$	feature matrix
$\boldsymbol{\phi}(\cdot)$	feature generating function
$\mathbf{\Sigma}_0$ / $\mathbf{\Sigma}_{\text{post}}$	prior / posterior covariance matrix
α	learning rate / slant angle
δ_{ij}	Kronecker delta
γ	ratio of specific heats
η	adiabatic compression efficiency / Kolmogorov length scale
κ	thermal conductivity
λ	second coefficient of viscosity
ϕ	minimum distance function
$\Theta(\cdot)$	decoder
θ	set of trainable weights
μ	molecular viscosity
ρ	density
$\Phi(\cdot)$	encoder
$\phi(\cdot)$	signed distance function
Ω	physical domain
ω	specific turbulent dissipation rate
σ^2	variance
$\sigma.(\cdot)$	element-wise activation function
τ_{ij}	viscous stress tensor
ξ	streamwise computational coordinate
$\mathcal{B}$	boundary condition operator
$\mathcal{D}$	dataset
$\mathcal{E}$	graph edges

$\mathcal{G}$	graph
$\mathcal{GP}$	Gaussian process
$\mathcal{J}(\cdot)$	objective / loss function
$\mathcal{M}$	vector space, design variables
$\mathcal{Q}$	vector space, CFD solution
$\mathcal{R}$	PDE system of equations operator
$\mathcal{V}$	graph vertices
$\mathcal{X}$	vector space, spatial coordinates or data space
$\mathcal{X}'$	vector space, augmented spatial coordinates
$\mathcal{Z}$	vector space, latent representation
$\hat{\cdot}$	approximation of quantity $(\cdot)$
$\tilde{\cdot}$	non-dimensional quantity $(\cdot)$
$\mathbb{E}[\cdot]$	expectation of quantity $[\cdot]$
$\mathbb{R}$	real numbers

List of Acronyms

ANN	artificial neural network
ASO	aerodynamic shape optimization
BO	Bayesian optimization
CFD	computational fluid dynamics
CNN	convolutional neural network
DCNN	decoder convolutional neural network
DES	detached eddy simulation
DMD	dynamic mode decomposition
DNS	direct numerical simulation
DoE	design of experiments
DVH	design-variable hypernetwork
DV-MLP	design-variable multi-layer perceptron
EGO	efficient global optimization
FDM	finite-difference method
FEM	finite-element method
FFNN	feed-forward neural network
FiLM	feature-wise linear modulation
FIML	field inversion machine learning
FLOPS	floating point operations per second
FVM	finite-volume method
GAN	generative adversarial network
GCN	graph convolutional network
GNN	graph neural network
GPR	Gaussian process regression

GPU	graphics processing unit
HFM	high fidelity model
HPL	High Performance Linpack
LES	large-eddy simulation
LSTM	long short-term memory
MAE	mean-absolute error
MDF	minimum distance function
MDO	multi-disciplinary design optimization
MLP	multi-layer perceptron
MPGNN	message-passing graph neural network
MRL2E	mean-relative-L2 error
MSE	mean-squared error
NIDS	non-linear independent dual system
ODE	ordinary differential equation
PDE	partial differential equation
POD	proper orthogonal decomposition
QoI	quantity of interest
RANS	Reynolds-averaged Navier-Stokes
RMSE	root-mean-squared error
ROM	reduced-order model
RSM	response-surface method
SBO	surrogate-based optimization
SDF	signed distance function
SQP	sequential quadratic programming
SLSQP	sequential least-squares quadratic programming
TKE	turbulent kinetic energy
WR-LES	wall-resolved large eddy simulation
WM-LES	wall-modeled large eddy simulation

Abstract

Computational simulations have become central in many engineering and scientific disciplines, providing insight and understanding into the behavior of complex systems. High-fidelity models (HFMs) of physical phenomena are frequently expressed using partial differential equations which require expensive and complex numerical methods for solution. This limits their application in many-query scenarios such as design optimization, model-based control, and inverse problems. The complexity may be decreased through simplifying assumptions regarding the underlying physics and coarse-graining. Alternatively, data-driven approaches may be used, whereby the HFM solutions or scalar quantities of interest (QoIs) are approximated using a surrogate regression model.

Surrogate modeling techniques aim to decrease computational expense while retaining dominant solution features and characteristics. In the context of optimization, "surrogate modeling" typically corresponds to approximation of scalar QoIs which comprise terms in the objective function or constraints. The scalar QoIs are usually extracted from HFM solutions, often as integral quantities, and represent a distillation of the information present in the solution. Research into driving engineering design optimization with data-driven or deep-learning-based QoI surrogates has become widespread and seen many successes. However, QoI surrogates are limited in their portability, as a change in the objective function possibly requires retraining new surrogates. This may be remedied by approximating the HFM solution fields instead of the extracted QOIs as the solutions are objective agnostic. However, this is a more difficult task, not only due to the much greater problem dimensionality, but also due to practical considerations regarding the required varying mesh topologies across parameter or design-variable space.

This thesis is concerned with development of HFM surrogates or emulators which are *discretization independent*. The problem physics drives modeling choices regarding

numerical scheme and spatial/temporal discretization, and in many engineering-relevant scenarios the meshes must be adapted or changed among solution instances to effectively capture variation in all important solution-field features. These changes are often driven by alteration of the physical design or operating conditions as described by the problem parameters or design variables. Existing frameworks based on convolutional neural networks and snapshot-matrix decomposition often rely on lossy pixelization and data-preprocessing, limiting their effectiveness in realistic engineering scenarios. The interpolation represents a loss of important information in regions with large solution-field gradients where mesh points are tightly spaced, such as through fluid boundary layers and free-shear layers in the wake of aerodynamic bodies. The methods developed here, however, can learn from heterogeneous data sources by handling each point in every mesh separately, bypassing the need for constructing snapshot matrices of consistent dimension for each case.

Developed methods include decoder convolutional neural network (DCNN) models for situations where the computational domain has a regular Cartesian or block-Cartesian structure. DCNN models map directly between the design variables and full solution fields, providing richer information than QoI surrogates. This addresses information loss due to interpolation, but not variation in mesh topology among solution instances. Recently, coordinate-based multi-layer-perceptron networks have been found to be effective at representing 3D objects and scenes by regressing volumetric implicit fields, with applications in computer graphics. A key distinction is that coordinate-inputs are taken *pointwise* instead of as full-domain solution snapshots. These concepts are leveraged and adapted in the context of physical-field surrogate modeling, and allow for full discretization independence where each solution may have a unique and varying domain, boundary, mesh topology, and operating conditions.

Generalization across solution instances is achieved by conditioning the neural networks through a combination of *local* and *global* variables. Local conditioning relates to use of the signed-distance function or minimum-distance function as an additional network input to provide geometric information. Global conditioning utilizes the problem

design variables as the conditional input. Various methods of global conditioning are explored, including concatenation-based conditioning and the use of hypernetworks for full or partial network-weight conditioning. The methods are applied to predict solutions around complex, *parametrically-defined geometries* on *non-parametrically-defined meshes* with model predictions obtained many orders of magnitude faster than the full-order models. The incorporation of random Fourier features enhances prediction and generalization accuracy in some situations, as does incorporation of layer-normalization in the main network.

The HFM surrogates are applied to a variety of steady-state problems, including 2D external vehicle aerodynamics, jet-engine-compressor aerodynamics, the 2D poisson equation with a source term, and finally 3D Ahmed body aerodynamics. Additionally, the HFM surrogates are used to drive aerodynamic design optimization of jet-engine-compressor airfoils in subsonic and transonic regimes, with orders-of-magnitude reduction in online time to attain optimal designs as compared to CFD-driven optimization. In summary, this work develops and demonstrates HFM surrogates with promising potential as practical tools in industrial analysis and design.

Chapter 1

Introduction

The prevalence and importance of numerical simulation has grown in concert with and as a result of the exponential increase in computing power through the latter-half of the 20th century, as famously described by Moore's law [1]. His initial forecasting in 1965 posited a log-linear relation by which integrated circuit complexity would increase by a factor of two year-over-year until 1975, and actual industry developments matched this pace over that period. In 1975 the forecast was revised with a gentler slope corresponding to a doubling period of 18-24 months, and at this time the term "Moore's Law" was coined and entered the public lexicon [2], remaining relevant ever since. Engineers and researchers have continually developed computer components, architectures, and programming languages in order to put the transistor-dense chips to best use. Virtually every branch of science has utilized the enhanced computational resources, and thus these developments have gone hand-in-hand with more detailed and accurate models and methods; a natural coevolution.

Measuring the performance of computers is a complex task, as different applications have varying requirements and measures of success. One commonly used measure is a reporting of the **f**loating-**p**oint-**op**erations-per-**s**econd, or FLOPS, where a FLOP corresponds to a single mathematical operation such as addition or multiplication. The High Performance Linpack (HPL) benchmark [3] provides a standard problem for comparison, solving a dense linear system of equations, and the results are compiled in the TOP500 list [4]. Figure 1.1, taken from reference [5], plots the maximum performance on the HPL benchmark against year, showing the continual improvement and development through the teraflop and petaflop eras. The next milestone moves performance into the exascale, with the Frontier supercomputer at Oak Ridge National Laboratory the first to pierce this barrier while currently sitting atop the TOP500 list with a measured performance of 1.194 exaFLOPs R_{max}[1] [4].

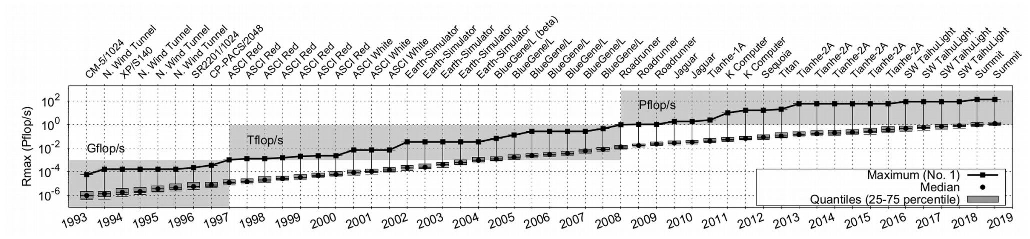


Figure 1.1: The HPL performance from 1993-2019, taken from Ref. [5], showing the development through the peta-FLOP era of supercomputing.

1.1 Interplay Between Fidelity and Cost in Engineering Simulation

Numerical simulations of complex physical systems are based upon mathematical descriptions representing governing equations or physical laws which are derived or discovered through empirical and theoretical scientific study. These models may be conceptual, phenomenological, mechanistic and/or probabilistic in nature and frequently do not have a closed-form solution, necessitating the need for numerical solutions. Engineering models typically involve the numerical solution of governing equations expressed using systems of partial differential equations (PDEs). In mechanical and aerospace engineering, computational fluid dynamics (CFD) is widely used to simulate external flows around machines or structures and internal flows through engineering devices. The governing equations for fluid mechanics generally take the form of the Navier-Stokes equations in the continuum regime, and the Boltzmann equations for the kinetic, non-continuum regime, with augmentations for additional physics to account for chemical reaction, non-equilibrium, and plasma-state scenarios. Within each regime, simplifying assumptions applied to the governing equations or solution procedure result in a hierarchy of models, where the

[1]R_{max} is a measured value using the HPL benchmark, as compared to R_{peak} which is theoretical.

simplifications are generally made to ease the computational burden or outright enable some problems to be simulated. This hierarchy is particularly notable for simulations of high-Reynolds-number fluid turbulence, characterized by chaotic multi-scale behavior.

While the main contributions of this thesis center on data-driven or deep-learning techniques for fast approximation of engineering-relevant PDE systems in general, most of the problems considered are in the domain of fluid mechanics and aerodynamics in particular. Thus, in what follows in this introductory chapter, notation surrounding high-fidelity simulation models is presented generally at first in Section 1.1.1 and then the governing equations of continuum-regime fluid motion are presented in Section 1.1.2. Following this an overview of the various fidelities of simulating fluid mechanics problems is given, highlighting the need for modeling simplification due to computational constraints arising from the multi-scale problem physics. Then background information and context surrounding optimization and data-driven methods is provided in Section 1.2 before the specific contributions are enumerated in Section 1.3 and an overall outline in Section 1.4.

1.1.1 High-Fidelity Models

Engineering models frequently have parametric dependence on some input conditions, typically describing the domain geometry, physical properties, and operation conditions. Collect all such parameters in vector $\boldsymbol{\mu} \in \mathbb{R}^{n_\mu}$, let $\mathbf{x} \in \mathbb{R}^{n_x}$ be the spatial coordinates, and $t \in \mathbb{R}$ be time. Then vector-valued solution-states at a point in physical space may be written as $\mathbf{q}(\mathbf{x}, t; \boldsymbol{\mu}) \in \mathbb{R}^{n_q}$, which are the unknowns to be determined. Frequently, governing-equation PDEs are derived from conservation laws which have spatio-temporal dependence, and considering a domain $\Omega(\boldsymbol{\mu})$ and boundary $\partial\Omega(\boldsymbol{\mu})$ which are functions of the parameters $\boldsymbol{\mu}$, then the PDE system may be written generally as

$$\frac{\partial \mathbf{q}(\mathbf{x}, t; \boldsymbol{\mu})}{\partial t} + \mathcal{R}\big(\mathbf{q}(\mathbf{x}, t; \boldsymbol{\mu}), \mathbf{x}, t, \ldots; \boldsymbol{\mu}\big) = 0, \; \mathbf{x} \in \Omega(\boldsymbol{\mu}), \; t \in [t_0, t_f], \tag{1.1}$$

where $\mathcal{R}$ is a (usually) non-linear PDE-term operator which generates the required expressions, and commonly includes spatial derivatives, divergences, and source terms. All other necessary outside information is collected in the ellipsis. Appropriate initial and boundary conditions must also be prescribed. The initial conditions may be written as

$$\mathbf{q}(\mathbf{x}, t_0; \boldsymbol{\mu}) = \mathbf{q}_0,\ \mathbf{x} \in \Omega(\boldsymbol{\mu}), \tag{1.2}$$

which fully defines the solution state over the domain. Consider a boundary-condition operator $\mathcal{B}$ which defines the boundary conditions in a similar manner to the governing equations, written as

$$\mathcal{B}(\mathbf{q}(\mathbf{x}, t; \boldsymbol{\mu}), \mathbf{x}, t, \dots; \boldsymbol{\mu}) = 0,\ \mathbf{x} \in \partial\Omega(\boldsymbol{\mu}),\ t \in [t_0, t_f]. \tag{1.3}$$

The boundary conditions may set the value of $\mathbf{q}$ directly (Dirichlet), the boundary-normal gradients $\nabla\mathbf{q} \cdot \hat{\mathbf{n}}$ (Neumann), or a linear combination of both (Robin), and may differ for each component of $\mathbf{q}$ or for varying positions on the boundary.

Equations 1.1-1.3 define the PDE-system of interest. Solutions are obtained numerically, usually by discretizing in space and time with scheme details dependent upon the problem physics, although mesh-free methods exist for some scenarios. Many engineering models treat the spatial discretization of $\mathcal{R}/\mathcal{B}$ separately from the time-integration scheme. Spatial discretization schemes include finite difference methods (FDMs), finite element methods (FEMs), and finite volume methods (FVMs), and entire fields of study, such as CFD, are devoted to the development of domain-specific algorithms. Applying the spatial discretization leads to the semi-discrete form of the governing equations, which have been transformed from a system of PDEs to a system of ODEs. Considering a spatial discretization which results in N nodes or cell centers holding the solution state, the semi-discrete form may be written as

$$\frac{d\mathbf{q}_N(t; \boldsymbol{\mu})}{dt} = f(\mathbf{q}_N(t; \boldsymbol{\mu})), \tag{1.4}$$

where $\mathbf{q}_N \in \mathbb{R}^{Nn_q \times 1}$ holds the n_q-dimensional solution state at all N spatial locations, and f represents the discretization of $\mathcal{R}/\mathcal{B}$, though rearranged. Note that in this thesis, column vectors are represented by lowercase bold letters while matrices, arrays, or tensors are represented by bold uppercase letters.[2] Thus $\mathbf{q}_N$ is a flattened representation of the spatially-discretized solution state. The semi-discrete initial conditions are written as

$$\mathbf{q}_N(t_0; \boldsymbol{\mu}) = \mathbf{q}_{N,0}. \tag{1.5}$$

Appropriate handling of boundary conditions depends greatly on the spatial discretization scheme selected and the boundary-condition type.

When time-varying problems are considered, the solution is marched forward in time using a time-integration scheme. Common methods include Runge-Kutta, linear-multistep methods, and predictor-corrector methods [6]. When a steady-state problem is considered, obviously no time integration is required but any non-linearity in f generally still requires iterative methods in order to obtain a solution. In either scenario $\mathbf{q}_N$ represents the high-fidelity-model (HFM) solution.

Numerical simulations are subject to many sources of error, with three important categories being modeling, discretization, and convergence errors. Modeling errors arise from differences between the true nature of the simulated phenomena and the mathematical model. Discretization errors account for the differences between exact discretized and mathematical-model solutions, while convergence errors are due to the difference between the exact discretized solution and that which is numerically obtained. Modeling errors are most fundamental and relate to overall state-of-knowledge, as development of a perfect model would require absolute understanding and mastery of the problem physics; a condition notably lacking for fluid turbulence. Assessment of modeling errors is difficult and requires comparison between simulation and experiment, but may be confounded by other sources of errors given that the mathematical models usually don't have closed-form solutions. Discretization and convergence errors are related to notions of consistency and

[2]An exception applies in discussion of the governing equations of fluid motion and RANS/LES models in Section 1.1.2, where index-notation conventions are followed and include representing tensors by their scalar entries, such as τ_{ij} for the viscous stress tensor.

stability. Consistency relates to the numerical scheme and its discretization, and for a scheme to be consistent the discretized form must tend to the underlying mathematical model as the space and time steps tend to zero, while stability requires that all numerical errors must remain bounded while obtaining an iterative solution [7].

Thus, the spatial (and temporal) discretization are inherently linked to the numerical scheme, and defining an appropriate mesh and scheme for a problem is incredibly important. Generally, FDMs require computational meshes with a regular Cartesian structure, or multiple mesh zones with such structure. Elliptic body-fitted meshes [8] allow for use of regular computational meshes on irregular domains or geometries but can struggle in regions with intricate geometric details. Whereas unstructured meshes can more naturally accommodate complex geometry and be adapted more easily in the presence of large solution gradients. FEM and FVM schemes can be applied using unstructured meshes and as a result these methods are more widely used in commercially available simulation software and remain at the forefront of advanced research topics. When a parametric set of HFM solutions is considered, as described by varying $\boldsymbol{\mu}$, each solution will require its own mesh that may be adapted to the manifestation of the problem physics according to the specifics of each design. This becomes problematic when attempting to apply data-driven methods to approximate HFM solutions, as many techniques require, at a minimum, consistent spatial discretization among all solution instances.

1.1.2 Navier-Stokes Equations and a Hierarchy of Models

The governing relations for continuum-regime fluid motion are derived by considering conservation of mass, momentum, and energy. A Eulerian perspective is considered, in which the fluid state is expressed by fields, as functions of space $\mathbf{x} = \begin{bmatrix} x & y & z \end{bmatrix}^T$ and time t. Several assumptions are made in this presentation:

1. Continuum regime, single species, non-reacting, gas-phase flow

2. Local thermodynamic equilibrium everywhere in the flow

3. Acceleration due to gravity $\mathbf{g}$ may be ignored

4. Newtonian fluid, linear relation between stress and strain

5. Heat is conducted via Fourier's law, $\mathbf{q}'' = -\kappa \nabla T$, with isotropic thermal conductivity κ

Assumption 1 may still be applied to non-reacting mixtures such as air since the main component species of Oxygen and Nitrogen are both diatomic molecules with similar properties, and thus thermodynamic properties of the mixture are computed and used. The unknowns are the velocity vector field $\mathbf{u}(\mathbf{x},t) = u(\mathbf{x},t)\hat{\mathbf{i}} + v(\mathbf{x},t)\hat{\mathbf{j}} + w(\mathbf{x},t)\hat{\mathbf{k}}$, the thermodynamic pressure $p(\mathbf{x},t)$, and the absolute temperature $T(\mathbf{x},t)$. Frequently the coordinates $(\mathbf{x},t)$ are left off to ease notation. Other included thermodynamic quantities are the density $\rho(\mathbf{x},t)$, internal energy $e(\mathbf{x},t)$ or enthalpy $h = e + p/\rho$, and transport properties of molecular viscosity $\mu(\mathbf{x},t)$ and thermal conductivity $\kappa(\mathbf{x},t)$. Given assumption 2, each of these may be considered a function of p and T via state relations expressed as empirical formulas or data collected in tables or charts, and in practice many depend much more strongly on temperature. For example the viscosity μ is frequently found using a power law or Sutherland's Law,

$$\frac{\mu}{\mu_0} \approx \left(\frac{T}{T_0}\right)^{3/2} \frac{T_0 + S}{T + S}, \tag{1.6}$$

where T_0 and μ_0 are reference states and S is a gas-specific Sutherland constant, all of which can be found from tables for a given gas. The thermal conductivity κ is also found using a power law or Sutherland's Law, identical in form to Equation 1.6 with κ replacing μ, along with different lookup tables. Frequently a perfect gas assumption is made, giving rise to the ideal gas law

$$p = \rho R_{sp} T, \tag{1.7}$$

where $R_{sp} = R_u/M$ is the specific gas constant, and

$$R_u = N_A k_B = \left(6.022 \times 10^{23} \frac{\#\text{ particles}}{\text{mol}}\right)\left(1.3806 \times 10^{-23} \frac{\text{J}}{\text{K}}\right) = 8.314 \frac{\text{J}}{\text{mol K}} \tag{1.8}$$

is the universal gas constant expressed in terms of Avogadro's number N_A and the Boltzmann constant k_B, where M is the molar mass or molecular weight of the gas species or mixture. Note that R_u, N_A, and k_B are not expressed with full precision in the expression above. Many other equivalent forms of the ideal gas law exist and are used in practice, scenario dependent. The perfect-gas assumption also means the specific heat capacities of the gas at constant pressure c_p and constant specific volume c_v are functions of temperature only, leading to $de = c_v dT$ and $dh = c_p dT$, and further $e = c_v T$, $h = c_p T$ are the specific internal energy and enthalpy when T is the absolute temperature. The total energy per unit mass is then given by $E = e + \frac{1}{2} u_i u_i$.

The conservation of mass, momentum, and energy may be expressed as

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x_i}(\rho u_i) = 0 \tag{1.9}$$

$$\frac{\partial}{\partial t}(\rho u_i) + \frac{\partial}{\partial x_j}(\rho u_i u_j) = \frac{\partial \tau_{ij}}{\partial x_j} - \frac{\partial p}{\partial x_i} \tag{1.10}$$

$$\frac{\partial}{\partial t}(\rho E) + \frac{\partial}{\partial x_j}[(\rho E + p)u_j] = \frac{\partial}{\partial x_j}(u_i \tau_{ij}) + \frac{\partial}{\partial x_j}\left(\kappa \tfrac{\partial T}{\partial x_j}\right) \tag{1.11}$$

respectively. Term τ_{ij} are the viscous stresses which may be written as

$$\tau_{ij} := \mu\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) + \delta_{ij}\lambda\frac{\partial u_k}{\partial x_k} \tag{1.12}$$

where δ_{ij} is the Kronecker delta

$$\delta_{ij} := \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j \end{cases} \tag{1.13}$$

and λ is the second coefficient of viscosity. Usually Stokes' hypothesis is used, $\lambda = -\frac{2}{3}\mu$, although this is an approximation and not true in general, especially for highly compressible flows. The viscous stresses are frequently written in terms of the strain-rate tensor,

$$S_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right), \tag{1.14}$$

and together with Stokes' hypothesis the viscous stresses become

$$\tau_{ij} = 2\mu S_{ij} - \delta_{ij}\frac{2}{3}\mu\frac{\partial u_k}{\partial x_k}. \tag{1.15}$$

Collectively Equations 1.9-1.11 are the governing equations of fluid motion. The conservation of momentum, expression 1.10, is vector valued and results in three separate relations, one for each coordinate direction, and are commonly known as the Navier-Stokes equations, but frequently this label is used in reference to the complete set of governing equations. Thus there are five expression with five unknowns, along with auxiliary expressions and equations of state for computing the thermodynamic properties. These governing equations are non-unique, and other formulations exist which are derived in terms of the entropy s for example. If chemical reactions are considered, then additional species conservation and chemical reaction expressions must be included.

Appropriate boundary conditions must be set, with common boundaries including solid walls, inlets and outlets, along with far-field or free-stream conditions, and the proper handling of boundary conditions depends greatly on the flow regime considered. Notably, in a wall bounded flow or flow past an object, the velocity and temperature are subject to the no-slip condition, $\mathbf{u} = \mathbf{u}_{\text{wall}}$ or $T = T_{\text{wall}}$. This gives rise to boundary layer behavior, where the gas state change rapidly in a thin region directly adjacent to the wall. Capturing this rapid variation is of primary concern in fully-resolved simulations, and gives rise to computational difficulties related to discretization and scaling.

The governing equations may be simplified when variation in density is negligible, known as the incompressible flow regime, and for external flows is loosely considered as $\text{Ma} < 0.3$, where Ma is the Mach number,

$$\text{Ma}(\mathbf{x}, \mathbf{t}) = \frac{|\mathbf{u}(\mathbf{x}, t)|}{a(x, t)}, \tag{1.16}$$

where $a \in \mathbb{R}_{>0}$ is the speed of sound. If the viscosity μ is also assumed constant, then the energy equation becomes decoupled and is no longer needed to determine $\mathbf{u}$ and p, with ρ prescribed. The energy equation may be used later to determine T if that information is also of interest. The continuity equation, Equation 1.9, reduces to $\nabla \cdot \mathbf{u} = 0$, providing simplification to the viscous stresses τ_{ij} and removing the need for Stokes' hypothesis. Considering the material derivative

$$\frac{D}{Dt} := \frac{\partial}{\partial t} + (\mathbf{u} \cdot \nabla), \tag{1.17}$$

the incompressible governing equations of continuity and momentum conservation are then given by

$$\nabla \cdot \mathbf{u} = 0 \quad \rightleftarrows \quad \frac{\partial u_i}{\partial x_i} = 0 \tag{1.18}$$

$$\rho \frac{D\mathbf{u}}{Dt} = \mu \nabla^2 \mathbf{u} - \nabla p \quad \rightleftarrows \quad \rho \frac{\partial u_i}{\partial t} + \rho u_j \frac{\partial u_i}{\partial x_j} = \frac{\partial}{\partial x_j}(2\mu S_{ij}) - \frac{\partial p}{\partial x_i} \tag{1.19}$$

respectively.

The Reynolds number is an important dimensionless similarity parameter in fluid mechanics. It describes the ratio of inertial and viscous forces in a flow and is given by

$$\mathrm{Re} = \frac{\rho |\mathbf{u}| L}{\mu}. \tag{1.20}$$

A single flow may be described by many Reynolds numbers depending on the length and velocity scales selected and the location within the flow. For example, consider an external incompressible flow around an airfoil with chord length c and free-stream conditions ρ_∞ and $\mathbf{u}_\infty$, then the Reynolds number for the configuration is given by $\mathrm{Re} = \frac{\rho_\infty |\mathbf{u}_\infty| c}{\mu}$. The Reynolds number also characterizes the multi-scale nature of the flow, given that the inertial forces are related to bulk fluid motion of length scale L, while the viscous forces are realized by much smaller diffusional or molecular length scales. Thus a flow with a larger Reynolds number displays greater multi-scale behavior, considered further below. Many excellent textbooks cover derivations and further considerations in greater detail, including those which were heavily referenced in preparing this section [9, 10, 11].

1.1.2.1 Direct Numerical Simulation

The highest fidelity of fluid simulation is known as direct numerical simulation (DNS), in which all relevant scales are fully resolved and Equations 1.9-1.11 or 1.18-1.19 are solved without further modeling simplification. The requirements this imposes on the spatial discretization and ultimately the overall computational cost is examined. The energy cascade exemplifies the multi-scale nature of turbulence, describing the process by which turbulent kinetic energy at the largest scales is transferred to successively smaller scales as eddies are broken up before finally being dissipated as heat. The Kolmogorov scales describe the length, velocity, and time scales of the smallest, dissipative eddies. These scales are intrinsically linked to the largest scales through the energy transfer rate from the large scales, which must ultimately equal the dissipation rate at the small scales.

Consider a flow characterized by a streamwise Reynolds number Re_L. The Kolmogorov length scale η is related to the size of the largest eddies ℓ_0 as $\eta/\ell_0 \sim \mathrm{Re}_L^{-3/4}$, meaning that as the Reynolds number increases, so does the difference between largest and smallest scales to be resolved [12]. The number of mesh points for boundary-layer DNS may be estimated as $N_{\mathrm{DNS}} \sim \mathrm{Re}_L^{37/14}$ [13]. Consider a motor vehicle of length 4 m traveling at highway speeds of 30 m/s. This corresponds to a Reynolds number of around 8×10^6. The number of required grid points for DNS is then estimated to be 1.8×10^{18}. Following ref. [14], the time in days to complete the simulation may be roughly estimated as

$$T_G = \frac{n_{\mathrm{ops}} \times n_{\mathrm{mesh}} \times n_{\mathrm{steps}}}{R \times 60 \times 60 \times 24}, \tag{1.21}$$

where n_{ops} are the number-of-operations per mesh-point per time-step, n_{mesh} is the number of mesh points, n_{steps} are the number of time steps, and R is the performance of the computer in FLOPS. To stably and fully resolve the finest scales, a Kolmogorov-scale Courant number may be considered. First, estimate the Kolmogorov scale as

$$\eta/\ell_0 \sim \mathrm{Re}^{-3/4} \rightarrow \eta \approx \ell_0 \mathrm{Re}^{-3/4} = 4 \times \left(8 \times 10^6\right)^{-3/4} = 2.66 \times 10^{-5}\ \mathrm{m}, \tag{1.22}$$

then consider turbulence intensity of 1%, reasonable though perhaps an overestimate for quiescent air as a vehicle approaches to give

$$I = \frac{u'}{|\mathbf{u}_\infty|} \rightarrow u' = I|\mathbf{u}_\infty| = (0.01)(30) = 0.3 \text{ m/s}. \tag{1.23}$$

Then a Kolmogorov-scale Courant number may be defined and set as $\text{Co}_\eta = \frac{u'\Delta t_\eta}{\eta} = 0.1$. Rearranging for the required time step size yields

$$\Delta t_\eta = \frac{\text{Co}_\eta \eta}{u'} = \frac{(0.1)(2.66 \times 10^-5)}{0.3} = 8.8 \times 10^{-6} \text{ s}. \tag{1.24}$$

Typically a computational domain around a vehicle will be sized as approximately $10 \times L$ [15] in the streamwise direction, so the number of time steps for a fluid particle traveling at the freestream velocity to traverse the domain is $n_{t,1} = \frac{10L/|\mathbf{u}_\infty|}{\Delta t_\eta} = \frac{(10\times4)/30}{8.8\times^{-6}} = 1.5 \times 10^5$. DNS simulations must be run until the solution becomes statistically stationary, and for petascale DNS of turbulent channel flow on the Mira supercomputer at Re = 5000 [16], statistical stationarity is achieved after an estimated 13 domain flow-throughs or roughly 6.5×10^5 time steps. Applying a similar multiplier of 10 yields $n_{\text{steps}} = 1.5 \times 10^6$ for the vehicle. Then, following [14], let $n_{\text{ops}} = 10^3$, and plugging into Equation 1.21 gives the estimated exascale simulation wall time of

$$T_{G,DNS} = \frac{(10^3)(1.8 \times 10^{18})(1.5 \times 10^6)}{10^{18} \times 60 \times 60 \times 24} = 30440.9 \text{ days} \approx 83.4 \text{ years}. \tag{1.25}$$

This may also be a conservative estimate by several orders of magnitude, as $n_{\text{ops}} = 1000$ was chosen to be representative for a pseudo-spectral method solved on a Cartesian mesh with a simple geometry, while a vehicle contains complex geometry requiring an unstructured mesh and thus a different method. Further, $R = 10^{18}$ represents exascale performance, with only a single supercomputer at Oak Ridge National Lab achieveing this peak performance, and it is unrealistic to expect peak performance at all or even most times. Even further, the free-stream turbulence intensity was used to estimate the Kolmogorov length scale, and while the intensity may be overestimated for the air in front of the vehicle, greater turbulence intensity will exist near the vehicle and in its wake, requiring an even finer spatial discretization and thus smaller time stepping requirements. Clearly DNS is out of the question for even a single vehicle design, let alone a trade study, and will remain so for some time. Therefore, modeling simplifications must be made in order to drastically reduce the simulation cost.

1.1.2.2 Large Eddy Simulation and Reynolds Averaged Navier Stokes

Large-eddy Simulation (LES) and Reynolds Averaged Navier Stokes (RANS) approaches lessen the computational burden of DNS by decomposing the state quantities by filtering or averaging, respectively.[3] This reduces the required number of mesh points, but both filtering and averaging bring about a closure problem whereby additional terms are introduced into the governing equations which must be modeled. Development of those models so that RANS/LES simulations remain physically consistent with the unmodified governing equations, via comparison with experiments or DNS, represent a central challenge and an active area of research. In general LES is more expensive and accurate than RANS, and only recently is LES becoming affordable for some industrial applications.

For RANS, Reynolds-averaging approaches are used to decompose state quantities q_i into mean $\langle q_i \rangle$ and fluctuating components q_i', that is

$$q_i(\mathbf{x}, t) = \langle q_i(\mathbf{x}, t) \rangle + q_i'(\mathbf{x}, t) \tag{1.26}$$

The most general form of the mean is an ensemble average,

$$\langle q_i(\mathbf{x}, t) \rangle_E := \lim_{N \to \infty} \frac{1}{N} \sum_{j=1}^{N} q_i^j(\mathbf{x}, t) \tag{1.27}$$

for N independent realizations which are identical other than for random perturbations of initial and boundary conditions. Often a time-average is used instead of the ensemble average, defined as

$$\langle q_i(\mathbf{x}) \rangle_T := \lim_{T \to \infty} \frac{1}{T} \int_t^{t+T} q_i(\mathbf{x}, t) dt, \tag{1.28}$$

[3]While filtering is a form of spatial averaging the distinction is maintained.

where of course T represents time not temperature in this context. Time averaging in this way is only applicable to flows which do not change with time, as t is lost from the mean coordinate as compared to ensemble averaging. Unsteady-RANS (URANS) approaches do not take the limit as $T \to \infty$, but instead define a time scale T which is large compared to the turbulent time scales T_1, but small compared to the time-scales by which the mean flow changes, T_2. Then the time-average may be redefined as

$$\langle q_i(\mathbf{x}, t) \rangle_T := \frac{1}{T} \int_t^{t+T} q_i(\mathbf{x}, t) dt, \quad T_1 \ll T \ll T_2, \tag{1.29}$$

and the solution may still be time-varying. However, this time-scale separation is not a valid assumption for all flows, or even for all locations within a flow, such as through a shear layer. When RANS equations are presented the averaging will be left general, using $\langle q_i \rangle$ without subscript, but many times the expressions are developed specifically using time-averaging [11].

Large-Eddy Simulation (LES) directly simulates the large-scale turbulent motions (the large eddies) while modeling the effects of smaller scales [17]. The decomposition of q_i has filtered $\overline{q}_i$ (resolved) and residual q_i^r components resulting,

$$q_i(\mathbf{x}, t) = \overline{q}_i(\mathbf{x}, t) + q_i^r(\mathbf{x}, t). \tag{1.30}$$

This is justified by deference to Kolmogorov's hypothesis of local isotropy which posits a universal or near-universal form for the small scales of turbulence. LES is more naturally suited for time-varying problems as unsteady effects are accounted for. Filtering is almost always applied via convolution, with spatial high-pass convolutional kernel G and spatial cutoff scale or filter width Δ, the filtering is defined by

$$\overline{q}_i(\mathbf{x}, t) := \frac{1}{\Delta} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} G\left(\frac{\mathbf{x} - \boldsymbol{\xi}}{\Delta}, t - t' \right) q_i(\boldsymbol{\xi}, t') dt' d^3\boldsymbol{\xi}. \tag{1.31}$$

Convolution is represented more compactly as

$$\overline{q}_i = G * q_i. \tag{1.32}$$

Conveniently, convolution in the spatial/temporal domain is multiplication in the wavenumber/frequency domain, as related by the Fourier transform, and thus many LES schemes are spectral in nature. Commonly used filters include the box, Gaussian, and sharp cutoff filters, among others [18].

The incompressible RANS equations may be expressed as

$$\frac{\partial \langle u_i \rangle}{\partial x_i} = 0 \tag{1.33}$$

$$\rho \frac{\partial \langle u_i \rangle}{\partial t} + \rho \langle u_j \rangle \frac{\partial \langle u_i \rangle}{\partial x_j} = \frac{\partial}{\partial x_j}\big(2\mu \langle S_{ij} \rangle\big) - \frac{\partial}{\partial x_j}\big(\rho \langle u_i^{'} u_j^{'} \rangle\big) - \frac{\partial \langle p \rangle}{\partial x_i}, \tag{1.34}$$

and the incompressible LES equations as

$$\frac{\partial \overline{u}_i}{\partial x_i} = 0 \tag{1.35}$$

$$\rho \frac{\partial \overline{u}_i}{\partial t} + \rho \overline{u}_j \frac{\partial \overline{u}_i}{\partial x_j} = \frac{\partial}{\partial x_j}\big(2\mu \overline{S}_{ij}\big) - \frac{\partial}{\partial x_j}\big(\rho \tau_{ij}^{\mathrm{r}}\big) - \frac{\partial \overline{p}_{\mathrm{m}}}{\partial x_i}, \tag{1.36}$$

where $\langle S_{ij} \rangle$ and $\overline{S}_{ij}$ are the averaged and filtered strain-rate tensors, defined analogously to Equation 1.14 with averaged/filtered velocity components.When compared to the incompressible governing equations of Equation 1.18-1.19, the only difference in structure are a single additional term on the right-hand side of each. These stress tensors arise from averaging/filtering the non-linear advection term of the Navier-stokes equations, $\rho u_j \frac{\partial u_i}{\partial x_j}$, which is re-expressed in conservation form as $\rho \frac{\partial (u_i u_j)}{\partial x_j}$ in the incompressible regime. The extra terms arise because the averaged/filtered products $\langle u_i u_j \rangle / \overline{u_i u_j}$ are not equal to the product of the averaged/filtered terms $\langle u_i \rangle \langle u_j \rangle / \overline{u}_i \overline{u}_j$. For RANS, the difference is known as the Reynolds-stress tensor,

$$\langle u_i^{'} u_j^{'} \rangle = \langle u_i u_j \rangle - \langle u_i \rangle \langle u_j \rangle, \tag{1.37}$$

and for LES, the difference defines the residual stress tensor,

$$\tau_{ij}^{\mathrm{R}} := \overline{u_i u_j} - \overline{u}_i \overline{u}_j. \tag{1.38}$$

While the actual stress tensors are given by $-\rho\langle u_i' u_j' \rangle$ or $-\rho\tau_{ij}^{\mathrm{R}}$, the terminology above persists as ρ is known and prescribed for incompressible flow simulations. The residual kinetic energy is defined as

$$k_{\mathrm{r}} := \tfrac{1}{2}\tau_{ii}^{\mathrm{R}}, \tag{1.39}$$

which is in turn used to define the anisotropic residual-stress tensor

$$\tau_{ij}^{\mathrm{r}} := \tau_{ij}^{\mathrm{R}} - \tfrac{2}{3}k_{\mathrm{r}}\delta_{ij} \tag{1.40}$$

and modified filtered pressure

$$\overline{p}_{\mathrm{m}} := \overline{p} + \tfrac{2}{3}\rho k_{\mathrm{r}} \tag{1.41}$$

which appear in Equation 1.36. Similarly, for RANS, the turbulent kinetic energy is defined as

$$k := \tfrac{1}{2}\langle u_i' u_i' \rangle, \tag{1.42}$$

and similar to Equation 1.40, a deviatoric anisotropic stress tensor is given by

$$a_{ij} := \langle u_i' u_j' \rangle - \tfrac{2}{3}k\delta_{ij}. \tag{1.43}$$

The Reynolds stresses of Equation 1.37 and the residual-stress tensor of Equation 1.38 must be modeled in order to close each system of equations. Eddy-viscosity models are widely used in both approaches and depend upon an appropriate eddy-viscosity hypothesis. While the unclosed terms result from the same non-linear component of the momentum equation, the physics which drives them is different. The Reynolds stresses are due to momentum transfer via turbulent fluctuations of the velocity field, while the residual-stresses are due to momentum transfer via the unresolved velocity field, representing a coupling between the resolved and unresolved scales. Widely known turbulence models for RANS include the algebraic mixing-length model, single-equation TKE models, and two-equation models such as k-ϵ [19], k-ω [20], and k-ω-SST [21] which blends the two: k-ω near walls and k-ϵ elsewhere. Two-equation models include transport equations for k and ϵ/ω containing scalar coefficients which are tuned using boundary-layer theory, experimental data, and/or expert intuition. When the compressible flow regime is considered, many additional cross-correlation terms which also require modeling arise due to fluctuations in temperature and density, on top of existing pressure and velocity field fluctuations. Favre-averaging or Favre-filtering is often used to mathematically simplify the resulting expressions, but the additional physics must still be accounted for. The compressible RANS and LES equations are considerably more complicated and their treatment is left to the literature [11, 22]. The modeling challenge is also greater in the compressible regime as there is less experimental data to anchor modeling choices.

At this time RANS approaches are most widely used in industry due to the reduced cost, where RANS is cheaper by a factor of up to 10^6 for Reynolds numbers around 10^6 [23]. LES variants include wall-resolved (WR-LES) and wall-modeled (WM-LES) approaches, with WR-LES being more desired and accurate. The number of required grid points for WR-LES scales approximately as $N_{\text{WR}-\text{LES}} \sim \text{Re}_L^{13/7}$ while WM-LES displays linear scaling $N_{\text{WM}-\text{LES}} \sim \text{Re}_L$ [13]. Applying the WR-LES estimator to the vehicle scenario considered for DNS results in 6.6×10^{12} required mesh points, a great improvement over DNS but still out of reach for practical application in industry. Some estimates place WR-LES becoming computationally tractable by around 2035 [23] to 2045 [24]. Hybrid RANS-LES schemes, such as detached-eddy simulation (DES) [25, 26], have also been developed and represent a middle-ground between the two. In DES the near-wall region is treated using RANS, while LES is used for the bulk flow, either using a zonal approach or a modified wall-distance to control switching between schemes. For flows where compressibility and/or unsteady effects are important, such as axial compressor design or aeroacoustics, the utility of LES is even greater and RANS simulations must be treated with greater uncertainty.

1.2 Data-driven Modeling and Optimization

High-fidelity numerical simulations are ubiquitous in engineering design and analysis but are often prohibitively expensive in design applications. Data-driven and machine-learning surrogate-modeling techniques offer an interesting alternative, particularly in situations where model accuracy may be acceptably traded for computational savings. However, many existing methods face limitations when confronted with unstructured and varying mesh topologies across the parameter or design-variable space. This confines such methods to problems that can be defined with a shared discretization, or requires additional lossy interpolation to map solutions onto consistent meshes. These limitations pose a significant challenge for problems involving multi-scale phenomena, commonly found in fluid and structural mechanics, where solutions frequently contain regions with large gradients and tightly-clustered mesh cells. Additionally, the domains may contain intricate and varying geometric features among solution instances. In such scenarios interpolating the solutions to a common and often Cartesian mesh results in unacceptable loss of critical information and fidelity.

Data-driven approaches may be considered in many engineering-relevant contexts. The list below enumerates a few broad, non-exclusive categories which will be expanded upon in the proceeding sections.

1. Solver augmentation: many high-fidelity models depend on modeled quantities or closures which have uncertainty in their form or numerical coefficients, resulting in mismatch between simulation and experimental measure. The experimental data is used to augment the uncertain quantities in an effort to reduce the error present. This includes Field-Inversion and Machine-Learning (FIML) techniques. Increasing solution speed is not the goal in this scenario, only increasing accuracy and/or interpretability.

2. Optimization and surrogate meta-modeling: in scenarios such as design optimization, scalar quantities of interest (QoIs) extracted from the solution field are used to compute the objectives and constraints. The QoIs are approximated directly,

without obtaining the solution field $\mathbf{q}_N$ or an approximation. Methods include Response Surface Methods (RSM), Guassian Process Regression (GPR), and use of artificial neural networks (ANN).

3. Solver acceleration: high-fidelity models are accelerated by modifying the solution procedure, either by replacing steps in an iterative scheme with an approximation, or by projecting the discretized solution $\mathbf{q}_N$ onto a reduced set of variables $\mathbf{q}_r \in \mathbb{R}^{r \times n_q}$, where $r << N$, and solving the governing equations, including nonlinear term f, on this reduced set; commonly referred to as intrusive Reduced Order Models (ROMs), and includes projection-based Galerkin and Petrov-Galerkin variants, along with deep-learning augmented POD-NN.

4. Solver replacement or emulation: the solution field $\mathbf{q}_N$ is approximated without direct reference to the high-fidelity model or its components. This includes non-intrusive ROMs, Dynamic Mode Decomposition (DMD), and Koopman Methods. Additionally, ANN-based methods built upon convolutional neural networks (CNN) graph neural networks (GNN), including Neural Operator variants, Deep Operator Networks, and physics-informed neural networks (PINNs). In some contexts this may also be referred to as surrogate modeling, with emphasis on modeling the *solution field* instead of QoIs, noting that the QoIs are instead extracted from the emulated field.

1.2.1 Solver Augmentation

As discussed in Section 1.1.2.2, RANS and LES formulations require turbulence models to close the system of equations due to the averaging/filtering operations. RANS turbulence models depend upon additional transport equations whose functional form is guided by expert intuition. They also contain empirical parameters which are calibrated to canonical flows or experimental data, and model performance suffers when applied to different scenarios. Recently, data-driven methods have been developed to improve existing models and to discover new, more accurate model forms using data from DNS, LES, and experiments.

Early works were within the regime of parameter estimation, and sought to tune turbulence model coefficients by regressing approximation models mapping between model coefficients and error metrics computed from experimental values, and then optimizing the approximation model to obtain new, improved coefficients [27, 28]. These techniques had limited portability and did not address model-form errors or uncertainties present in turbulence models. As a step beyond this, Bayesian uncertainty quantification methods were used to quantify RANS parameter and model-form uncertainties, where the turbulence model parameters are treated as random variables and model inadequacies are Gaussian random fields [29], as opposed to scalar measures such as root-mean-squared-error (RMSE) in the above parameter estimation approaches. Similarly, field-inversion machine-learning (FIML) techniques move beyond tuning existing coefficients and instead supplement existing closure models with additional spatial or spatiotemporal functions, either as additive or multiplicative modifications [30, 31, 32]. Correction fields are determined by solving inverse problems, usually using Bayesian approaches [33], and machine learning is used to regress functional forms in terms of local features accessible to the turbulence model [34]. The use of such features is an important step in attaining a generalizable and predictive augmented model, first recognized in the context of kernel regression to model local errors [35]. FIML has been applied in several contexts, including separated airfoil flows [36] and fuel-cell modeling [37].

1.2.2 Design Optimization and Surrogate Modeling

Design optimization routines seek to wholly or partially automate stages of the design process, and optimization underpins many methods for constructing data-driven models. Generally these tasks may be expressed as non-linear constrained optimization problems. Given design variables $\boldsymbol{\mu} = [\mu_1, \cdots, \mu_{n_\mu}]^T$, each with lower and upper bounds $\mu_{i,\min}/\mu_{i,\max}$, scalar objective function $\mathcal{J}(\boldsymbol{\mu}) \in \mathbb{R}$, n_g inequality constraints $g_j(\boldsymbol{\mu}) \in \mathbb{R}$, and n_h equality constraints $h_k(\boldsymbol{\mu}) \in \mathbb{R}$, the problem may be expressed as

$$\begin{aligned} \text{minimize} \quad & \mathcal{J}(\boldsymbol{\mu}) & \\ \text{by varying} \quad & \mu_i & \text{for } i = 1, \ldots, n_\mu \\ \text{subject to} \quad & g_j(\boldsymbol{\mu}) \leq 0 & \text{for } j = 1, \ldots, n_g \\ & h_k(\boldsymbol{\mu}) = 0 & \text{for } k = 1, \ldots, n_h \\ & \mu_{i,\min} \leq \mu_i \leq \mu_{i,\max} & \text{for } i = 1, \ldots, n_\mu. \end{aligned} \tag{1.44}$$

The above problem formulation does not include solution of relevant governing equations, but generally the objective and constraints are computed from parametric HFM solution fields $\mathbf{q}_N(\mathbf{x}, t; \boldsymbol{\mu})$. Equation 1.44 is solved using iterative numerical methods, using either gradient-based or gradient-free approaches, and more mathematical detail is provided in Section 2.2.

Sequential quadratic programming (SQP) methods are the most efficient and widely used gradient-based schemes to solve non-linear constrained optimization problems per Equation 1.44, and the sequential least-squares quadratic programming (SLSQP) algorithm [38] is available in many optimization software packages. These include `SciPy` [39] and `openMDAO` [40] python libraries, the latter providing a wrapper around the former, which are open source and free to use. SQP methods are analogs of Quasi-Newton methods for unconstrained optimization, and SLSQP uses very similar inverse-Hessian update expressions as the widely-used Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm [41, 42, 43, 44]. Other commonly used unconstrained gradient-based schemes include steepest descent and conjugate gradient methods, but do not include curvature information via the Hessian. Stochastic gradient descent and its variants are widely used in training neural networks.

For many engineering devices, varying model fidelities are acceptable at different stages in the design cycle, or for simulation of different sub-components or effects. When the design space is large then even lower-fidelity simulation models such as RANS may be unacceptably costly to fully explore the design space or to use directly in an optimization routine. This is especially true for multi-disciplinary design optimization (MDO), where the objectives and constraints may depend on segregated HFM solutions from multiple disciples, or from iterative or directly coupled multi-physics solvers. In any of these scenarios surrogate-based optimization (SBO) offers an affordable alternative, where surrogate models are regressed directly between the design variables $\boldsymbol{\mu}$ and the objective response $\mathcal{J}(\boldsymbol{\mu})$ or other QoI's, side-stepping HFM solution. Common methods for constructing surrogate models include least-squares regression, kriging or Gaussian process regression, and artificial neural networks. Later in Chapter 5, such surrogates are referred to as "QoI emulators" to help distinguish them from the flow-field emulators which are developed in this thesis. However, this is not common terminology, and generally when works refer to surrogates or SBO they are referencing models which predict scalar quantities.

Response surface methodology (RSM) was originally developed in the context of attaining optimal conditions in chemical experiments [45], where a second-order polynomial model is constructed to evaluate relationships between independent or explanatory variables and the experimental outcome or response. Ordinary least squares is used to obtain the unknown polynomial coefficients and the general approach may be applied in the context of design, where the design variables comprise the independent variables and the objective or QoIs are the scalar response, represented by $y \in \mathbb{R}$. The polynomial model takes the form

$$\hat{y}(\boldsymbol{\mu}) = w_0 + \sum_{i=1}^{n_\mu} w_i \mu_i + \sum_{i=1}^{n_\mu} w_{ii} \mu_i^2 + \sum_{i=1}^{n_\mu} \sum_{j>i}^{n_\mu} w_{ij} \mu_i \mu_j \tag{1.45}$$

where w are the unknown coefficients or weights to be determined. Given a dataset with M entries, $\mathcal{D} = \left\{ y_i | \boldsymbol{\mu}_i \right\}_{i=1}^{M}$, let $\mathbf{y} \in \mathbb{R}^M$ collect all $y \in \mathcal{D}$, let $\mathbf{w} \in \mathbb{R}^p$ collect all weights, construct Vandermonde matrix $\mathbf{A} \in \mathbb{R}^{M \times p}$, then the ordinary least-squares problem setup is

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \, \|\boldsymbol{\epsilon}\|_2^2 = \underset{\mathbf{w}}{\operatorname{argmin}} \, \|\mathbf{y} - \mathbf{A}\mathbf{w}\|_2^2 = \underset{\mathbf{w}}{\operatorname{argmin}} (\mathbf{y} - \mathbf{A}\mathbf{w})^T (\mathbf{y} - \mathbf{A}\mathbf{w}), \tag{1.46}$$

where $\boldsymbol{\epsilon} \in \mathbb{R}^M$ is the error. Typically there are more simulations than undetermined coefficients, that is $\mathbf{A}$ is taller than it is wide, $M > p$, and thus the problem is over-determined. In this scenario the solution to Equation 1.46 is well known, assuming $\text{rank}(\mathbf{A}) = p$, and is given by Equation 1.47.

$$\mathbf{w}^* = [\mathbf{A}^T\mathbf{A}]^{-1}\mathbf{A}^T\mathbf{y}. \tag{1.47}$$

Define polynomial-feature-generating function

$$\mathbf{p}(\boldsymbol{\mu}) := \mathbb{R}^{n_\mu} \to \mathbb{R}^p, \tag{1.48}$$

which applies the model form of Equation 1.45 (without coefficients) to a design variable vector $\boldsymbol{\mu}$, expressed as a column vector. This was used to construct the rows of the Vandermode matrix, and thus the model response prediction is written as

$$\hat{y}(\boldsymbol{\mu}; \mathbf{w}^*) = \mathbf{p}(\boldsymbol{\mu})^T\mathbf{w}^*. \tag{1.49}$$

The response surface may then be optimized to find an estimate for the optimal design, $\boldsymbol{\mu}^*$. RSMs are often applied sequentially, with more HFM solutions obtained spanning a region near the previous RSM optima. Quadratic response surfaces have been applied to design problems using FEM simulation data [46, 47] as well as RANS simulations for aerodynamic shape optimization [48, 49, 50].

Kriging or non-parametric Gaussian Process Regression (GPR) are also frequently used to construct response surfaces instead of least-squares polynomial models. Non-parametric GPR allows for predictions to be made directly in function space, without explicitly inferred the weights $\mathbf{w}$ of a linear model. Let matrix $\mathbf{M} \in \mathbb{R}^{n_\mu \times M}$ collect all input vectors $\boldsymbol{\mu} \in \mathcal{D}$. Given a covariance kernel $k(\boldsymbol{\mu}, \boldsymbol{\mu}') \in \mathbb{R}$ and mean function $m(\boldsymbol{\mu}) \in \mathbb{R}$, then matrix $k(\mathbf{M}, \mathbf{M}) \in \mathbb{R}^{M \times M}$ collects the covariance function over training points, where element (i, j) is $k(\boldsymbol{\mu}^{(i)}, \boldsymbol{\mu}^{(j)})$. Considering a design $\boldsymbol{\mu}_*$ where a response is sought, then column vector $k(\mathbf{M}, \boldsymbol{\mu}_*) \in \mathbb{R}^{M \times 1}$ collects the covariance function between the point of interest $\boldsymbol{\mu}_*$ and all training points in the dataset, as does row vector $k(\boldsymbol{\mu}_*, \mathbf{M}) \in \mathbb{R}^{1\times M}$, while $k(\boldsymbol{\mu}_*, \boldsymbol{\mu}_*)$ is a scalar. Further let $\boldsymbol{\Gamma} = \sigma^2 \mathbf{I}_M \in \mathbb{R}^{M\times M}$ be the i.i.d. Gaussian prior on the noise associated with the observations in dataset $\mathcal{D}$. The rules for conditional Gaussian distributions allow closed-form expressions for the posterior predictive distribution and variance functions for $\boldsymbol{\mu}_* \notin \mathcal{D}$, following Bayes' theorem.

$$\mathbb{E}[f(\boldsymbol{\mu}_*)|\mathbf{y}] = m(\boldsymbol{\mu}_*) + k(\boldsymbol{\mu}_*, \mathbf{M})\big(k(\mathbf{M},\mathbf{M}) + \boldsymbol{\Gamma}\big)^{-1}(\mathbf{y} - m(\mathbf{M})) \quad (1.50)$$

$$\mathrm{Var}(f(\boldsymbol{\mu}_*), f(\boldsymbol{\mu}_*)|\mathbf{y}) = k(\boldsymbol{\mu}_*, \boldsymbol{\mu}_*) - k(\boldsymbol{\mu}_*, \mathbf{M})\big(k(\mathbf{M},\mathbf{M}) + \boldsymbol{\Gamma}\big)^{-1} k(\mathbf{M}, \boldsymbol{\mu}_*) \quad (1.51)$$

Frequently the mean function is taken to be the zero function, and this simplifies the prediction, letting $f_* := \mathbb{E}[f(\boldsymbol{\mu}_*)|\mathbf{y}]$, then

$$f_* = k(\boldsymbol{\mu}_*, \mathbf{M})\big(k(\mathbf{M},\mathbf{M}) + \boldsymbol{\Gamma}\big)^{-1}\mathbf{y} \quad (1.52)$$

is the predicted function value. This may be interpreted as a linear combination of M kernel functions $k(\boldsymbol{\mu}_*, \mathbf{M})$, each centered on a training data point, with coefficients given by $\big(k(\mathbf{M},\mathbf{M}) + \boldsymbol{\Gamma}\big)^{-1}\mathbf{y}$. GPR is advantageous over RSM as it interpolates the training data exactly in the noise-free scenario, when $\boldsymbol{\Gamma} = \mathbf{I}$. That is, if $\boldsymbol{\mu}_* = \boldsymbol{\mu}_i \in \mathcal{D}$, then the product $k(\boldsymbol{\mu}_*, \mathbf{M})\big(k(\mathbf{M},\mathbf{M}) + \boldsymbol{\Gamma}\big)^{-1}$ is a row vector of zeros except for a one at position i, then subsequent inner product with $\mathbf{y}$ yields y_i, the exact value. Additional details and further background on parametric and non-parametric GPR are provided in Section 2.3.2.

Efficient global optimization (EGO) is a Bayesian SBO procedure designed for expensive, black-box objective functions, such as those dependent upon HFM solutions [51]. EGO utilizes GPR models to build response surfaces, and rather than minimizing the response surface directly, acquisition functions or figures of merit which account for uncertainty are optimized instead. A popular acquisition function is known as "expected improvement" which seeks to balance exploitation of points where the response is small and exploration of points where the uncertainty is high. Inclusion of this exploration term helps to free the optimizer from local minima, and accounts for the global nature of the algorithm. Other acquisition functions include probability of improvement, upper-confidence bound, and entropy search, among others which are designed with different trade-offs in mind [52]. Optimizing the acquisition function may be difficult, as they frequently contain local minima and regions with small gradients, and development of methods for their effective optimization is an active area of research [53]. Bayesian optimization is applied in the context of designing transonic compressor airfoils in Chapter 5, where flow-field emulators were used as a data source as compared against CFD.

There has been increasing interest in using artificial neural networks (ANNs) to regress surrogates, and for other tasks related to optimization. One advantage of ANNs as compared to RSM or GPR is that ANNs may be readily extended to predict a small vector of QoIs easily, whereas this is non-trivial for the other approaches. Co-kriging models do allow for this and one may naively predict vectors using vanilla GPR but this is effectively running several GPR models in parallel, without accounting for the covariance among the output quantities. This naive multi-output approach is demonstrated later in Section 2.3.2.1. ANNs have been used in SBO routines to predict lift, drag, and/or pitching moment coefficients in many scenarios [54, 55, 56].

Regressing surrogate models with deterministic computer simulation data has different recommendations and considerations as compared to experimental measures. In design-of-experiments (DoE) methods, experimental measurement uncertainty may be reduced by replicating an experiment many times. This is not necessary when selecting operating conditions from a HFM to train a meta-model, the recommendation is to instead sample $\boldsymbol{\mu}$ to span the design space without replication instead. Random sampling may be used, but this does not explicitly account for the span of the design space or the previously chosen designs. Latin hypercube sampling combats this and ensures that each input variable (element of $\boldsymbol{\mu}$) has all portions of its input distribution covered, and that the number of required samples does not increase with $\dim(\boldsymbol{\mu})$ [57].

1.2.3 Solver Acceleration and Intrusive Reduced Order Modeling

Numerical schemes used to solve the HFM are usually sparse, iterative methods, sometimes requiring many iterations to progress onto the next time step. There are frequently bottleneck steps within each iteration. A common example is the pressure Poisson equation resulting during application of the Semi-Implicit Method for Pressure-Linked Equations (SIMPLE) algorithm [58, 59, 60] for solution of the incompressible Navier-Stokes equations, knows as the pressure-correction step. Data-driven methods may be used to replace these bottleneck steps or to circumvent entire solver iterations. For example, CFD-Net is used to predict the full state, including an eddy-viscosity term for the Spalart Allmaras turbulence model, to accelerate 2D simulations of the incompressible RANS equations using the SIMPLE algorithm in OpenFOAM. After a warmup period, a CNN is used to predict the final solution state given the current intermediate values, and the CNN prediction is subsequently refined using the solver, providing a speedup by a factor of 1.9-7.4$\times$, once the network is trained.

As a classical dimensionality-reduction technique, proper orthogonal decomposition (POD) has been used to construct surrogate and reduced-order models [61, 62, 63, 64]. Despite many attractive properties, conventional POD implementations process discretized data, and require the use of a fixed topology mesh across *all parameter regimes*, fixing the number of degrees-of-freedom. This is restrictive in many engineering problems in which various solution features (e.g., relative motion of bodies, crack propagation, etc.) may emerge in different regions of parameter space. Further, data may be available from multiple sources with varying discretization and mesh topologies. Reduced order models (ROMs) seek to accelerate or supplement a high-fidelity model by utilizing information from previous observed solutions. ROM construction consists of two stages, an *offline* stage where solutions are generated and the model is formed, and an *online* stage, where the model is deployed in place of the full-order model. Note that the ROM may still make use of components of the high-fidelity solver.

ROMs are most commonly applied to time-varying problem, where an initial state is to be forward propagated in time. The ROM equations may be developed by considering the discretized form of the governing equations, as may be represented by the semi-discrete form

$$\begin{aligned} \frac{d\mathbf{q}_N(t)}{dt} &= \mathbf{f}(\mathbf{q}_N(t)) && (1.53) \\ \mathbf{q}_N(t_0) &= \mathbf{q}_{N,0} && (1.54) \end{aligned}$$

which may be viewed as an autonomous system. A single set of operating conditions $\boldsymbol{\mu}$ is considered and thus omitted from notation in this scenario. Let $N' = N \times n_q$ represent the full dimensionality of the state-space, where there are N mesh locations, and $\mathbf{q}_N \in \mathbb{R}^{N'}$. Projection-based ROMs seek a reduced set of variables $\mathbf{q}_r \in \mathbb{R}^k$ where $k << N'$ with which to evolve the system. This is achieved by projection and truncation in an appropriate basis, developed using data from high-fidelity solutions.

The ROM equations first approximate the full state as $\mathbf{q}_N(t) \approx \mathbf{\Phi}\mathbf{q}_r(t)$ or $\mathbf{q}_N(t) \approx \mathbf{q}_{N,0} + \mathbf{\Phi}\mathbf{q}_r(t)$ using an appropriate trial basis $\mathbf{\Phi} \in \mathbb{R}^{N' \times k}$, then given a test basis $\mathbf{\Psi} \in \mathbb{R}^{N' \times k}$ the ROM equation is given by

$$\frac{d\mathbf{q}_r}{dt} = \left[\mathbf{\Psi}^T\mathbf{\Phi}\right]^{-1}\mathbf{\Psi}^T\mathbf{f}(\mathbf{q}_{N,0} + \mathbf{\Phi}\mathbf{q}_r). \tag{1.55}$$

For Galerkin ROMs, the trial and test basis are set equal to another, $\mathbf{\Psi} = \mathbf{\Phi}$. When the columns of $\mathbf{\Phi}$ are orthonormal, then $\mathbf{\Phi}^T\mathbf{\Phi} = \mathbf{I}_k$ and the Galerkin ROM equation results as

$$\frac{d\mathbf{q}_r}{dt} = \mathbf{\Phi}^T\mathbf{f}(\mathbf{q}_{N,0} + \mathbf{\Phi}\mathbf{q}_r). \tag{1.56}$$

Petrov-Galerkin ROMs result when $\mathbf{\Psi} \neq \mathbf{\Phi}$ and the resulting projection is oblique instead of orthogonal. Proper orthogonal decomposition (POD) is frequently used to determine the basis $\mathbf{\Phi}$ for Galerkin ROMs. A POD expansion may be written as

$$\mathbf{q}_N(t) = \sum_{i=1}^{k} b_i(t)\boldsymbol{\phi}_i(\mathbf{x}), \tag{1.57}$$

where the spatial modes $\boldsymbol{\phi}_i(\mathbf{x})$ form a mutually orthogonal set and are weighted by scalar coefficients b_i. In the ROM equation, Equation 1.55, the entries of reduced representation $\mathbf{q}_r$ play the role of the coefficients.

Consider a data-snapshot matrix consisting of M solution snapshots at different times,

$$\mathbf{X} = \begin{bmatrix} | & | & \cdots & | \\ \mathbf{q}_N(t_0) & \mathbf{q}_N(t_1) & \ldots & \mathbf{q}_N(t_{M-1}) \\ | & | & \cdots & | \end{bmatrix} \in \mathbb{R}^{N' \times M}, \tag{1.58}$$

where the snapshots need-not be ordered. Note that the solution fields should be normalized or scaled, especially when $\mathbf{q}_N$ contains multiple output quantities, see Section 2.2.1. In some cases it may be beneficial to generate a separate POD basis for each flow quantity separately. Center the data snapshot matrix by subtracting the mean column $\langle \mathbf{q}_N \rangle$,

$$\mathbf{X}_m = \begin{bmatrix} \mathbf{q}_N(t_0) - \langle \mathbf{q}_N \rangle & \mathbf{q}_N(t_1) - \langle \mathbf{q}_N \rangle & \ldots & \mathbf{q}_N(t_{M-1}) - \langle \mathbf{q}_N \rangle \end{bmatrix} \in \mathbb{R}^{N' \times M}, \tag{1.59}$$

then perform a singular-value decomposition of $\mathbf{X}_m$,

$$\mathbf{X}_m = \mathbf{U}\Sigma\mathbf{V}^T = \begin{bmatrix} \mathbf{U}_1 & \mathbf{U}_2 \end{bmatrix} \begin{bmatrix} \mathbf{\Sigma}_1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{V}_1 & \mathbf{V}_2 \end{bmatrix}^T = \mathbf{U}_1\mathbf{\Sigma}_1\mathbf{V}_1^T. \tag{1.60}$$

Matrices $\mathbf{U} \in \mathbb{R}^{N' \times N'}$ and $\mathbf{V} \in \mathbb{R}^{M \times M}$ are orthonormal, and $\Sigma \in \mathbb{R}^{N' \times M}$ is a rectangular diagonal matrix whose entries are the singular values. $\mathbf{U}_1$ and $\mathbf{V}_1$ have r columns, and $\mathbf{\Sigma}_1 \in \mathbb{R}^{r \times r} = \text{diag}(\sigma_1, \sigma_2, ..., \sigma_r)$ where $\text{rank}(\mathbf{X}_m) = r$. The trial basis is selected as the first k columns of $\mathbf{U}$, known as the left singular vectors. That is, let

$$\mathbf{\Phi} = \mathbf{U}[:, : k] \tag{1.61}$$

using zero-indexing and python `numpy` slicing conventions. An appropriate value for k may be selected by examining reconstruction errors or by optimal hard-thresholding per reference [65]. The mean column must be added back wherever the basis is used.

Even though the dimension of Equations 1.55 and 1.56 are $k \times 1$, they involve evaluating the non-linear term $\mathbf{f}$ for N' locations, and subsequently performing a matrix-vector product as $\boldsymbol{\Psi}^T\mathbf{f}$, which is $(k \times N') \times (N' \times 1)$. This still scales as N' and appears to defeat the purpose. However, methods such as the discrete empirical interpolation method (DEIM) [66, 67, 68], compressed sensing [69, 70], and gappy-POD [71, 72] may be used to actually reduce computational cost in ROMs, in many cases by approximating the non-linear function $\mathbf{f}$ with a reduced POD basis.

1.2.4 Solver Replacement or Emulation

Non-intrusive ROMs are different than those from the previous section in that they do not make direct use of the HFM solver or its components at all, beyond using data. One such method is known as POD-NN and was originally introduced in the context of steady-state PDE solutions with dependence on parameters $\boldsymbol{\mu}$ [73]. A POD basis is computed by first collecting M solution snapshots for different $\boldsymbol{\mu}$,

$$\mathbf{X} = \begin{bmatrix} | & | & \cdots & | \\ \mathbf{q}_N(\boldsymbol{\mu}_1) & \mathbf{q}_N(\boldsymbol{\mu}_2) & \cdots & \mathbf{q}_N(\boldsymbol{\mu}_M) \\ | & | & \cdots & | \end{bmatrix} \in \mathbb{R}^{N' \times M}, \tag{1.62}$$

subtracting the mean column $\langle \mathbf{q}_N \rangle$,

$$\mathbf{X}_m = \begin{bmatrix} \mathbf{q}_N(\boldsymbol{\mu}_1) - \langle \mathbf{q}_N \rangle & \mathbf{q}_N(\boldsymbol{\mu}_2) - \langle \mathbf{q}_N \rangle & \ldots & \mathbf{q}_N(\boldsymbol{\mu}_M) - \langle \mathbf{q}_N \rangle \end{bmatrix} \in \mathbb{R}^{N' \times M}, \tag{1.63}$$

and an SVD computed to obtain a reduced basis $\boldsymbol{\Phi}$ using k left singular vectors

$$\mathbf{X}_m = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T \rightarrow \boldsymbol{\Phi} = \mathbf{U}[:, : k]. \in \mathbb{R}^{N' \times k} \tag{1.64}$$

A mean-centered snapshot, a column of $\mathbf{X}_m$, may be rank-k reconstructed given the basis coefficients $\mathbf{a}(\boldsymbol{\mu}) \in \mathbb{R}^k$, computed for all snapshots in $\mathbf{X}_m$ as

$$\mathbf{X}_m \approx \mathbf{\Phi}\mathbf{A} \rightarrow \mathbf{A} := \mathbf{\Phi}^T \mathbf{X}_m = \begin{bmatrix} | & | & \dots & | \\ \mathbf{a}(\boldsymbol{\mu}_1) & \mathbf{a}(\boldsymbol{\mu}_1) & \dots & \mathbf{a}(\boldsymbol{\mu}_M) \\ | & | & \dots & | \end{bmatrix} \in \mathbb{R}^{k \times M}. \tag{1.65}$$

Alternatively the coefficients could be determined for the training set using the SVD, $\mathbf{A} = \mathbf{\Sigma}[:k, :k]\mathbf{V}[:, :k]^T$. Next a neural network is trained to map the design variables to the basis coefficients, $N(\boldsymbol{\mu}_i; \theta) = \mathbf{a}(\boldsymbol{\mu}_i)$, where N represents the ANN and θ is its set of trainable weights. Then when a prediction is needed for a new parameter set $\boldsymbol{\mu}_*$, the trained ANN is used to generate the basis coefficients, $N(\boldsymbol{\mu}_*; \theta) = \mathbf{a}(\boldsymbol{\mu}_*)$, which are in turn used with the basis $\mathbf{\Phi}$ to generate the approximate solution as

$$\mathbf{q}_N(\boldsymbol{\mu}_*) \approx \overline{\mathbf{q}}_N + \mathbf{\Phi}\mathbf{a}(\boldsymbol{\mu}_*) \tag{1.66}$$

POD-NN has also been applied to time-varying parametric problems using a two-step POD algorithm, where the time trajectories for each $\boldsymbol{\mu}$ are compressed in the first stage, and then a second stage performs POD on the compressed trajectories [74].

POD-based methods are powerful but have a few limitations and shortcomings. First, all HFM solutions must lie on meshes with the same topology and number of points. This restricts the types of problems which may be considered, as such meshes are not always appropriate for all regions of parameter space, especially in the presence of complex and variable geometry. Second, the methods assume that span($\mathbf{\Phi}$) fully encompasses the solution dynamics for all $(\boldsymbol{\mu}, t)$. This may be a poor assumption in convection dominated problems, where temporal predictions are sought outside the region spanned by the training set.

Convolutional neural network (CNN) based autoencoders have been used to construct solution-field surrogate models for both steady-state [75, 76, 77, 78, 79] and time-varying parametric problems [80, 81] by including an additional time-advance model such as an

LSTM or temporal-convolutional network. However, they place even greater restrictions on the discretization than POD-based methods, requiring inputs and outputs to be defined on regular Cartesian grids with consistent dimensions for all parameter regimes. Overcoming this restriction requires interpolation from the computational mesh to a Cartesian grid overlain on the problem domain, equivalent to pixelization. The interpolation results in a number of undesirable effects, including a reduced-fidelity representation of the domain geometry, and a loss of information in regions of tightly-clustered mesh points, such as within boundary layers, shocks, and wakes. The models may then be conceptualized as image-to-image mappings.

Another more problematic but surprisingly overlooked issue is that the memory requirements for 3D convolutions, commonly implemented on a single GPU, are not affordable for typical resolutions in realistic engineering problems. Considering mini-batch training, even storing the output of one single hidden layer (a 5-dimensional tensor), requires memory typically on the order of $O(10) - O(10^2)$ GB for a 3D Cartesian field with 40 million cells. As a result, most reported works using 3D CNNs for engineering problems are limited to below 5-6 million degrees of freedom [82, 83], and often still require lossy interpolation [84].

Graph neural networks have been developed to extend CNNs to problems defined on non-Euclidean domains, or with non-regular Cartesian structure. In the context of modeling physical simulations, the computational mesh may be treated as a graph, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of vertices representing points in the computational domain, and $\mathcal{E}$ is the set of edges defining the connections among the nodes corresponding to mesh connectivity. Graph neural networks may be classified as either spectral [85, 86, 87, 88] or spatial [89] approaches, although the two may be generalized by the message-passing graph neural network (MPGNN) [90]. MPGNNs have been used for body-force predictions of aerodynamic flows [91].

Additionally, MPGNNs are used as a sub-component for certain learning and prediction schemes, with a focus on PDEs in either a mesh-based [92, 93] or mesh-free scenario [94]. These methods operate in the computational domain and are used to advance a solution field from one time instance to the next, serving as a model for the high-fidelity simulator. The architectures consist of encoder-processor-decoder components, with MPGNNs used in the processor to compute interactions among computational nodes. In some instances a particle-based representation of the simulation is used [94], where message passing is used to capture non-local interactions between discrete particles in the simulation. While others adopt a more finite-volume-method inspired perspective in constructing the processor [93, 92], with message passing used to represent fluxes. Impressive results are seen with these methods, and they overcome many of the shortcomings of CNN-based approaches. Conditionally parameterized networks [92] also share some similarity with the proposed methods, in that neural network weights are treated as parametric functions, much like design-variable hypernetworks generate the weights and biases for the main network. However, the focus of those works is on simulating a particular problem instance forward in time, while here the focus is surrogate modeling of steady-state, parametrically-related cases.

Another class of relevant techniques capable of handling unstructured data includes operator-regression methods, such as those based on DeepONet [95, 96, 97], Neural Operator [98, 99], Fourier basis networks [100], and GMLS Nets [101]. DeepONet and Neural Operator methods have shown impressive results but generally seek to develop mappings between spatially-varying input functions appearing *explicitly* in the governing equations and the solution. The architecture of an unstacked DeepOnet corresponds to the use of a hypernetwork for just the final layer, except the DeepONet branch network also consumes input functions sampled across the domain as opposed to an embedding or known quantity such as $\boldsymbol{\mu}$, as is pursued here. This distinction may seem to split hairs, but DeepONet requires sensors to be fixed in location across all solutions which breaks the desired discretization independence, with no obvious way to place sensors when variable geometry is considered.

1.3 Objectives and Contributions

Data-driven methods may be applied to engineering-relevant scenarios in a variety of ways, notably by solver augmentation and acceleration, or through modeling of QoIs or HFM solution fields. The primary objective of this dissertation is to develop data-driven and ultimately deep-learning methods which are capable of handling unstructured simulation data in the presence of variable geometry and operating conditions without lossy interpolation of the prediction or ground truth data. Specifically, the objectives may be enumerated as follows:

1. To develop data-driven techniques for full-field surrogate modeling of PDE simulations, to allow for:
    (a) Mesh independence
    (b) Parametric variation in geometry and operating conditions
    (c) Indirectly coupled mesh and model size; allows for scaling to 3D domains with lessened memory limitations
2. To evaluate the effectiveness of the techniques on a variety of problems, with a focus on external aerodynamics, and including problems with industrial-scale complexity
3. To develop and explore methods for effective model construction and training
4. To demonstrate design-optimization routines driven by full-field surrogates

The resulting contributions are enumerated below.

1. Adapted coordinate-based neural networks for discretization-independent, full-field surrogate modeling of problems with complex, parametric geometry and operating conditions.
    (a) Developed methods include design-variable-embedded dense networks, along with one-shot full and partial hypernetwork models; inspired by recent advances in deep-learning for computer graphics rendering tasks and modal decomposition techniques.

2. Showed that hypernetwork-based predictions may be written in a form analogous to POD, without restriction on mesh size or topology.

3. Introduced and demonstrated decoder-convolutional-neural-networks (DCNN) for surrogate modeling of problems defined on single and multi-block coordinate transformed structured meshes; eliminates the need for lossy interpolation.

4. Trained DCNN and DVH flow-field emulators used in design optimization of subsonic and transonic compressor rotors; a problem of industrial scale complexity.

   (a) Dataset performance-space imbalance identified and mitigated through augmentation via simple repetition of cases, vastly improving model predictions for high-performing airfoils.

   (b) Emulators used in place of CFD in Bayesian design-optimization routine, providing orders-of-magnitude savings in online computational cost.

5. Developed batch-by-case training for hypernetwork-based models, providing increased training stability and an order-of-magnitude savings in training time when combined with mixed- or single-precision training.

6. Applied piecewise constant/exponentially-decaying learning rate schedules to model training, resulting in improved convergence rates.

7. Demonstrated scaling to three-dimensional, unstructured simulations of the Ahmed body for vehicle aerodynamics with parametric geometry.

1.4 Thesis Outline

The content of this thesis is arranged as follows. In Chapter 2, greater technical detail is provided on both classical regression techniques along with more recent relevant developments, many based on deep learning or ANN. The shortcomings of a few techniques in a vehicle aerodynamics scenario are presented as well, highlighting their inability to directly handle unstructured data and the consequences therein. Chapter 3 presents additional background along with the methods used to build fast surrogate models for HFM solutions, centered on coordinate-based networks capable of handling unstructured data, and include design-variable MLP (DV-MLP), design-variable hypernetworks (DVH), and nonlinear independent dual system (NIDS). Decoder convolutional neural networks (DCNN) are also developed for problems defined on structured meshes. Chapter 4 presents surrogate modeling applications of the methods developed in Chapter 3, where the goal is simply to regress accurate and generalizable surrogates, without use in a downstream task. Chapter 5 presents the culmination of a multi-year project centered on the emulation and design of energy-efficient compressor airfoils. This work was a partnership with Raytheon Technologies Research Center (RTRC) and a part of the MULTI-LEADER project, with funding through the ARPA-E DIFFERENTIATE program. This work was presented at SciTech 2023 [102] and was further developed into a journal article published in the AIAA journal [103]. Finally, conclusions, perspectives, and areas for future work are provided in Chapter 6, with additional details and figures given in the Appendix.

Chapter 2

Optimization and Relevant Machine-Learning-Based Regression and Surrogate Modeling Techniques

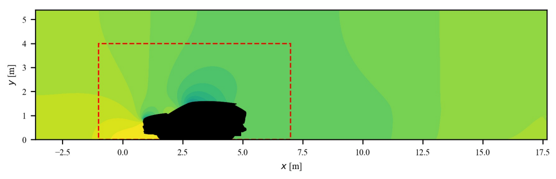
Many techniques for data approximation are in wide use, ranging from curve fitting or least-squares regression to complex neural network architectures. An overview of many methods is provided here and expands upon those presented in Chapter 1, with a greater emphasis on technical detail. The methods will be presented generally in some scenarios, but largely will center on the approximation of parametric fields or scalar QoI's which are extracted from said parametric fields. Following Section 1.1.1, the solution state at a point in space is written as $\mathbf{q}(\mathbf{x};\boldsymbol{\mu}) \in \mathbb{R}^{n_q}$, and let $\mathbf{q}_N(\mathbf{x};\boldsymbol{\mu}) \in \mathbb{R}^{n_{\text{mesh}}n_q}$ represent the parametric field produced by a HFM expressed as a flattened vector, where parameters or design variables are collected in $\boldsymbol{\mu} \in \mathbb{R}^{n_\mu}$ and $\mathbf{x} \in \mathbb{R}^{n_x}$ are the spatial coordinates for a single point in n_x-dimensional space. Steady-state solutions are considered so variable t is left out of notation, although t may be considered as an element of $\boldsymbol{\mu}$ to retain generality. Let $y(\boldsymbol{\mu}) := f(\mathbf{q}_N(\mathbf{x};\boldsymbol{\mu})) \in \mathbb{R}$ be a scalar quantity of interest which is extracted from the HFM solution via generic function f.

Before presentation of the various methods, a vehicle aerodynamics test problem is described with the goal of demonstrating the shortcomings of Gaussian Process Regression (GPR) and Proper Orthogonal Decomposition (POD) in that scenario. Then methods which are usually used to approximate scalar QoI's are presented next and include least squares regression, GPR, and multi-layer perceptron (MLP) neural networks. Following this, background on autoencoders, and many relevant ANN-based methods are presented.

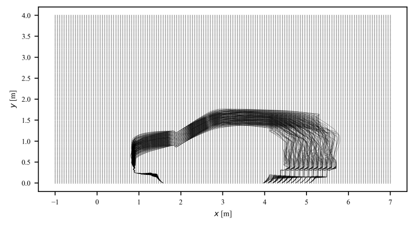
2.1 A Test Problem: 2D Vehicle Aerodynamics

A main point of emphasis in this thesis is the handling of solution data defined on unstructured meshes with varying dimension and topology. In order to understand and illustrate the shortcomings of some existing techniques, a vehicle-aerodynamics dataset consisting of solutions to the 2D steady incompressible RANS equations around realistic, parametric vehicle shapes on unstructured meshes is processed such that those methods may be applied. Greater detail on this dataset may be found in Section 4.3, where it is used in unprocessed form.

The methods considered rely on snapshot data matrices and require, at a minimum, that the number of mesh points is consistent among all snapshots. The vehicle solutions are processed by overlaying a Cartesian grid on the domain around the vehicle and interpolating each flow quantity from the computational mesh onto the Caresian grid. The flow solutions lie in the xy plane with freestream velocity vector pointing in the positive x direction, $\mathbf{u}_\infty = u_\infty \hat{\mathbf{i}}$. The domain limits for the Cartesian mesh are $x \in [-1, 7]$, $y \in [0, 4]$, shown in relation to the full domain in Figure 2.1a. Figure 2.1b shows a 150×150 Cartesian grid overlain on the region of interest along with all vehicle shapes from the dataset. This Cartesian grid resolution is used for most of the demonstrations, and was selected to be consistent with previous work from the lab group using convolutional autoencoders to predict airfoil flows using a similar processing scheme [76]. One major drawback of using the interpolated data is that an appreciable portion of the Cartesian points lie within the vehicle shape, where CFD solutions are not defined. Additionally, the points which lie inside each shape are not consistent among all cases.



(a) Full domain with region of interest.



(b) Region of interest with Cartesian grid for interpolation and all vehicle shapes.

Figure 2.1: Vehicle aerodynamics domain and processed Cartesian grid.

Another significant drawback of interpolation is loss of information in boundary layers and wakes, along with a reduced fidelity representation of the problem geometry. An interpolated vehicle pressure field is shown in its raw form in Figure 2.2, and the effect of the pixelation on the vehicle shape is apparent. Later, figures are generated using filled contours which further interpolate and smooth the appearance.

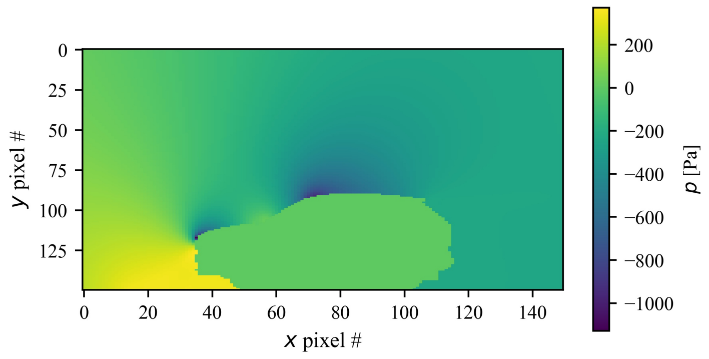


Figure 2.2: Cartesian grid interpolated pressure field in the region of interest, where the pixelation in the vehicle shape is apparent.

2.2 Optimization in Design and Model Construction

Conventional engineering design processes are iterative in nature, with repeated successive stages of design generation and performance evaluation using some metrics which grade the designs according to the specifications. In large organizations the design and analysis engineers may be different groups of people, particularly when physical prototypes must be produced and evaluated, potentially causing each iteration to consume large amounts of time and resources. Mathematical notions of optimality have been around from antiquity, becoming more common and precise since the invention and widespread adoption of Calculus, with examples including analytical means for finding and classifying function extrema, variational calculus and Lagrangian methods for identifying intrinsic coordinates, or through iterative root-finding procedures such as Newton-Rhapson iteration which forms the basis of Newton and Quasi-Newton methods in wide use today. More recently, within the last century, formal linear and non-linear optimization routines have been developed along with mathematically precise measures of optimality [104]. Applying these methods using computer simulation data from one or more disciplines concurrently is a very active area of research, with entire conferences dedicated to the topic. Further, optimization lies at the heart of nearly all data approximation or model construction techniques, from least squares to training of complex neural networks.

Gradient-based approaches are more mathematically formal and frequently make use of a first or second-order Taylor-series expansions of the objective function. A second-order multi-dimensional expansion about point $\boldsymbol{\mu}$ in direction $\mathbf{d} \in \mathbb{R}^{n_\mu}$ may be expressed as

$$\mathcal{J}(\boldsymbol{\mu} + \mathbf{d}) = \mathcal{J}(\boldsymbol{\mu}) + \nabla \mathcal{J}(\boldsymbol{\mu})^T \mathbf{d} + \tfrac{1}{2} \mathbf{d}^T \mathbf{H}(\boldsymbol{\mu}) \mathbf{d} + \mathcal{O}\big(\|\mathbf{d}\|_2^3\big), \tag{2.1}$$

where the gradient $\nabla \mathcal{J}(\boldsymbol{\mu}) \in \mathbb{R}^{n_\mu}$ is a column vector of partial derivatives

$$\nabla \mathcal{J}(\boldsymbol{\mu}) := \begin{bmatrix} \frac{\partial \mathcal{J}(\boldsymbol{\mu})}{\partial \mu_1} & \frac{\partial \mathcal{J}(\boldsymbol{\mu})}{\partial \mu_2} & \cdots & \frac{\partial \mathcal{J}(\boldsymbol{\mu})}{\partial \mu_{n_\mu}} \end{bmatrix}^T \tag{2.2}$$

and $\mathbf{H}(\boldsymbol{\mu}) \in \mathbb{R}^{n_\mu \times n_\mu}$ is the Hessian matrix of second-order partial derivatives, providing a measure of curvature. Adjoint-based solvers pair naturally with gradient-based schemes and provide the sensitivities $\nabla \mathcal{J}(\boldsymbol{\mu})$ along with the solution field $\mathbf{q}_N$ from which the objective is computed. Such solvers have been developed for many scenarios and governing equations, but represent a recent development and are not as widely used as other solution procedures. For non-adjoint solvers, computing derivatives of $\mathcal{J}(\boldsymbol{\mu})$ is not straight forward. The simplest approach is to use finite-differences, but this requires solving the HFM for each small perturbation of the design variables $\boldsymbol{\mu}$. This becomes impractical quickly as $\dim(\boldsymbol{\mu})$ increases, and the effect is compounded when considering the Hessian. Further, the appropriate size of the perturbations is not known a priori and numerical issues such as subtractive loss of precision may plague the result. Alternatively, the derivatives may be computed via the complex step method [105] or through algorithmic differentiation [106], also known as automatic differentiation, but these methods require access to and modification of the source code for the HFM solver.

The general optimization problem statement of Equation 1.44 also applies in the scenario of constructing or training an approximation model, and frequently corresponds to an unconstrained optimization problem with $n_h = n_g = 0$. In that context the objective is often referred to as a loss function, and may be written as $\mathcal{J}(\theta)$ or $\mathcal{J}(\mathbf{w})$, where θ is the set of all model weights or parameters, which are alternatively represented by vector $\mathbf{w}$. In scenarios such as least squares, the weights are defined as a vector so using $\mathbf{w}$ is natural, while for neural network models the weights are several matrices and vectors, represented more naturally by set θ. Neural networks in particular are frequently trained using first-order gradient based methods. Problem scaling is important both in design optimization and training of ANNs, so an overview is provided below. Following this, greater detail regarding unconstrained gradient-based optimization is provided, with a focus on first-order or non-Quasi-Newton schemes and other general considerations for training ANN.

2.2.1 Scaling and Normalization

The term $\nabla\mathcal{J}(\boldsymbol{\mu})^T\mathbf{d}$ from Equation 2.1 is known as a directional derivative, and quantifies the objective function rate-of-change projected onto vector $\mathbf{d}$, and in order to retain the correct units $\mathbf{d}$ should be scaled to have unit length,

$$\hat{\mathbf{d}} = \frac{\mathbf{d}}{\|\mathbf{d}\|_2} = \frac{\mathbf{d}}{\sqrt{\mathbf{d}^T\mathbf{d}}}. \tag{2.3}$$

The design variables $\boldsymbol{\mu}$ should also be normalized so that they have similar scale to another. When prescribed limits for each design variable are given, $\mu_{i,\min} \leq \mu_i \leq \mu_{i,\max}$ per Equation 1.44, then these limits may be used to apply min-max normalization to scale each entry of $\boldsymbol{\mu}$ so that it lies within $[0, 1]$. Let a normalized quantity be represented by $\breve{\cdot}$, then the formula for min-max normalization with prescribed limits is

$$\breve{\mu}_i = \frac{\mu_i - \mu_{i,\min}}{\mu_{i,\max} - \mu_{i,\min}}. \tag{2.4}$$

When constructing data-driven models, frequently all model inputs and outputs are normalized. At the time of model construction, only a training dataset $\mathcal{D}$ is on hand. For some inputs, like the design variables $\boldsymbol{\mu}$, prescribed limits may be on hand as above. For model outputs or target quantities the ranges may not be known for all unseen cases ahead of time, and instead statistics of the training dataset may be used. Let q_i be the ith entry of state vector $\mathbf{q}$, and collect all entries from the training dataset in vector $\mathbf{q}_{\mathcal{D},i}$. Min-max normalization is expressed similarly in this scenario as

$$\breve{q}_i = \frac{q_i - \min(\mathbf{q}_{\mathcal{D},i})}{\max(\mathbf{q}_{\mathcal{D},i}) - \min(\mathbf{q}_{\mathcal{D},i})}. \tag{2.5}$$

All training-set data will lie within $[0, 1]$, but it is possible data from unseen cases will lie slightly outside of this range if the values are above or below the training dataset limits. If normalization over a different range $[a, b]$ is desired, then a more general formula is

$$\breve{q}_i = (b - a)\frac{q_i - \min(\mathbf{q}_{\mathcal{D},i})}{\max(\mathbf{q}_{\mathcal{D},i}) - \min(\mathbf{q}_{\mathcal{D},i})} + a, \tag{2.6}$$

where a range of $[-1, 1]$ is also commonly used. Z-score normalization is also common in data-driven methods, and it transforms the data so that the normalized values have a mean of 0 and a standard deviation of 1. A z-score normalized quantity is computed as

$$\breve{q}_i = \frac{q_i - \mathrm{mean}(\mathbf{q}_{\mathcal{D},i})}{\mathrm{std}(\mathbf{q}_{\mathcal{D},i})}. \tag{2.7}$$

The optimizer should use normalized quantities, while the HFM solver usually uses fully-dimensional quantities. Thus the normalization should be reversed by rearranging the above expressions before use with the HFM solver.

2.2.2 Unconstrained Gradient-Based Optimization and Training ANN

Unconstrained optimization has straight-forward optimality conditions. Denoting the optimal design as $\boldsymbol{\mu}^*$, the conditions are

$$\begin{aligned} &\nabla \mathcal{J}(\boldsymbol{\mu}^*) = 0 \\ &\mathbf{H}(\boldsymbol{\mu}^*) \text{ is positive definite, } \mathbf{d}^T\mathbf{H}(\boldsymbol{\mu}^*)\mathbf{d} > 0 \text{ for all nonzero } \mathbf{d}. \end{aligned} \tag{2.8}$$

The former is the first-order optimality condition and depends only on the gradient information, without consideration of curvature via the Hessian $\mathbf{H}$.

Unconstrained optimization routines are typically grouped into one of two categories, either line-search or trust-region algorithms, and here line-search approaches are described further. Line-search methods follow a similar overall procedure with two important components; the selection of a search direction $\mathbf{d}$, and determination of a step size $\alpha \in \mathbb{R}_{>0}$ to take along that direction to decrease the objective. The design at iteration k is then updated according to

$$\boldsymbol{\mu}^{k+1} = \boldsymbol{\mu}^k + \alpha \mathbf{d}^k. \tag{2.9}$$

The procedure usually continues until the first-order optimality is satisfied to within some tolerance ε, often using $\|\nabla \mathcal{J}(\boldsymbol{\mu}^k)\|_\infty \leq \varepsilon$, or until a specified number of iterations have elapsed.

Gradient descent, also known as steepest descent, selects the search direction to be

$$\mathbf{d}^k = -\nabla \mathcal{J}(\boldsymbol{\mu}^k), \tag{2.10}$$

given that the gradient points in the direction of steepest ascent. As with consideration of a directional derivative, it is usually recommended to use a normalized search direction per Equation 2.3. Selecting $\mathbf{d} = -\nabla \mathcal{J}(\boldsymbol{\mu})$ seems like a sensible choice, but often results in a large number of iterations in the presence of high curvature in the objective; not surprising given that gradients are local and curvature effects were not considered.

One method to address the issues with steepest descent is to include a *momentum term* [107] in the update expression, such that the search direction at iteration k is a weighting of the current direction of steepest descent and the previous search direction, $k-1$. That is,

$$\mathbf{d}^k = -\nabla \mathcal{J}(\boldsymbol{\mu}^k) + \beta^{k-1} \mathbf{d}^{k-1}, \tag{2.11}$$

usually starting with $\mathbf{d}^0 = -\nabla \mathcal{J}(\boldsymbol{\mu}^0)$ and then updating per Equation 2.9 as usual. In the simplest case β may simply be a scalar specified ahead of time or left free as a hyperparameter, while other methods update β dynamically using a specific formula.

One such example is the conjugate gradient method [108], which was developed based upon linear conjugate gradient methods to solve linear systems of equations [109]. The update formula in that scenario is given by

$$\beta^k = \frac{\nabla \mathcal{J}(\boldsymbol{\mu}^k)^T \nabla \mathcal{J}(\boldsymbol{\mu}^k)}{\nabla \mathcal{J}(\boldsymbol{\mu}^{k-1})^T \nabla \mathcal{J}(\boldsymbol{\mu}^{k-1})}. \tag{2.12}$$

In the context of design, the step size α is usually determined using a line-search algorithm which are quite involved, with examples provided in references [110, 111].

Stochastic gradient descent (SGD) is a foundational algorithm for training ANN, and it differs from gradient descent only in that the gradients are estimated *stochastically* using mini-batches for each optimizer update. Momentum terms are a common feature of many gradient-based methods used for training ANN, and have been empirically shown to aid in optimization of such non-convex objective functions [107]. One such algorithm is Adam [112], which is used extensively in this thesis, and depends on estimated first and second moments m_t and v_t of the loss. It also depends upon scalar quantities $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$ with the default values given. The method uses update rules below, where β_1^k/β_2^k denotes exponentiation, not iteration.

$$\begin{aligned} m^k &= \beta_1 m^{k-1} + (1-\beta_1)\nabla \mathcal{J}(\theta^k) \\ v^k &= \beta_2 v^{k-1} + (1-\beta_2)(\mathcal{J}(\theta^k))^2 \\ \hat{m}^k &= m^k/(1-\beta_1^k) \\ \hat{v}^k &= v^k/(1-\beta_2^k) \\ \theta^{k+1} &= \theta^k - \alpha \hat{m}^k/(\sqrt{\hat{v}^k} + \epsilon) \end{aligned} \tag{2.13}$$

In training ANN, the learning rate α is usually specified ahead of time, either as a scalar or following a prescribed schedule such as an exponential decay. Other optimization schemes such as ADAGRAD [113], ADADELTA [114], and RMSprop [115] seek to dynamically adjust the learning rate per axis during training. Adam may be interpreted in this manner as well, and generally these methods are blended extensions of

SGD incorporating ideas from momentum and adaptive learning rates.

2.3 A Survey of Existing Regression Techniques

2.3.1 Regularized and Non-linear Least Squares

The least-squares procedure presented in Section 1.2.2 may be extended using *regularization* and *non-linear features*. Regularization is used to endow regression models with different properties by adding terms to the objective function, while non-linear features move the regression problem beyond polynomial curve fitting. If a response surface or other underlying function is highly complex or has other non-polynomial structure, such as periodicity, then a polynomial model may be insufficient. Other more complex feature transformations may be considered, written generally as

$$\boldsymbol{\phi}(\boldsymbol{\mu}) : \mathbb{R}^{n_\mu} \to \mathbb{R}^p, \tag{2.14}$$

where the entries of $\boldsymbol{\phi}(\boldsymbol{\mu})$ may contain arbitrary non-linear terms beyond monomials. The transformation may also represent a basis expansion, with Chebyshev and Hermite polynomial bases commonly used, for example. In this case the Vandermonde matrix $\mathbf{A}$, which is specific to polynomials, is replaced by general feature matrix $\boldsymbol{\Phi} \in \mathbb{R}^{M \times p}$, defined similarly as

$$\boldsymbol{\Phi} := \begin{bmatrix} - & \boldsymbol{\phi}(\boldsymbol{\mu}_1)^T & - \\ - & \boldsymbol{\phi}(\boldsymbol{\mu}_2)^T & - \\ \vdots & \vdots & \vdots \\ - & \boldsymbol{\phi}(\boldsymbol{\mu}_M)^T & - \end{bmatrix} \in \mathbb{R}^{M \times p}. \tag{2.15}$$

In all presentation of least squares, with and without regularization, the Vandermonde matrix $\mathbf{A}$ may be replaced with the general feature matrix $\boldsymbol{\Phi}$. Model predictions are then written as

$$\hat{y}(\boldsymbol{\mu}; \mathbf{w}^*) = \boldsymbol{\phi}(\boldsymbol{\mu})^T \mathbf{w}^*, \tag{2.16}$$

which generalizes Equation 1.49.

Ridge-regression [116] is used to avoid over-fitting, and it does so via regularization in the form of penalizing the L2-norm of the model weights $\mathbf{w}$,

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \|\mathbf{y} - \mathbf{\Phi}\mathbf{w}\|_2^2 + \lambda\|\mathbf{w}\|_2^2, \tag{2.17}$$

where λ is a user-specified weighting coefficient. This penalty helps to make the predictive model less sensitive to outliers in the training dataset $\mathcal{D}$, more common in scenarios with large measurement error, and is useful when the training data is correlated. Ridge regression has a closed-form solution, given by

$$\mathbf{w}^* = (\mathbf{\Phi}^T\mathbf{\Phi} + \lambda\mathbf{I}_p)^{-1}\mathbf{\Phi}^T\mathbf{y}, \tag{2.18}$$

where $\mathbf{I}_p$ is the $p \times p$ identity matrix. Ridge regression is a special case of Tikhonov regularization, given by

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \|\mathbf{y} - \mathbf{\Phi}\mathbf{w}\|_2 + \lambda\|\mathbf{D}\mathbf{w}\|_2^2, \tag{2.19}$$

where $\mathbf{D} \in \mathbb{R}^{p\times p}$ is a linear operator, such as a circulant matrix for smoothing or denoising via total variation regularization. The solution is given by

$$\mathbf{w}^* = (\mathbf{\Phi}^T\mathbf{\Phi} + \lambda\mathbf{D}^T\mathbf{D})^{-1}\mathbf{\Phi}^T\mathbf{y}, \tag{2.20}$$

where setting $\mathbf{D} = \mathbf{I}_p$ recovers ridge regression. Other variations include LASSO [117], which is intended to promote sparsity and parsimony by penalizing the L1-norm of $\mathbf{w}$,

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \|\mathbf{y} - \mathbf{\Phi}\mathbf{w}\|_2^2 + \lambda\|\mathbf{w}\|_1, \tag{2.21}$$

which aides in obtaining an interpretable model, particularly when a dictionary of non-linear terms is used. Elastic Net [118] combines features of ridge regression and LASSO, including both $L1$ and $L2$ penalties on coefficients $\mathbf{w}$, and is given by

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \|\mathbf{y} - \boldsymbol{\Phi}\mathbf{w}\|_2^2 + \lambda_1 \|\mathbf{w}\|_1 + \lambda_2 \|\mathbf{w}\|_2^2, \tag{2.22}$$

where two coefficients λ_1 and λ_2 control the tradeoff between penalties. No closed-form solution exists in general for LASSO and Elastic Net and thus must be solved numerically. Regardless of the method selected to regress the response surface, once the coefficients $\mathbf{w} = \mathbf{w}^*$ are on hand, then the QoI response for a given $\boldsymbol{\mu}$ are obtained using Equation 2.16.

Least squares is a vast and important topic, with many courses in linear algebra and data science partially or wholly devoted to its study. Greater background and description of these methods and their variants in the context of RSM, including those applied to physical experiments, may be found in the literature [119].

2.3.2 Gaussian Process Regression

Gaussian Process Regression (GPR) is closely related to kriging models, kernel regression, and kernel machines such as support vector machines (SVM). First, define a Gaussian Process.

Definition 2.3.1 (Gaussian Process) *A Gaussian process is a distribution over functions, such that for any n inputs, the resulting n function evaluations have a multivariate Gaussian distribution. A Gaussian process is fully specified by its mean and covariance functions*

$$m(\boldsymbol{\mu}) = \mathbb{E}[f(\boldsymbol{\mu})] \tag{2.23}$$

$$k(\boldsymbol{\mu}, \boldsymbol{\mu}') = \mathbb{E}[(f(\boldsymbol{\mu}) - m(\boldsymbol{\mu}))(f(\boldsymbol{\mu}') - m(\boldsymbol{\mu}'))], \tag{2.24}$$

and a Gaussian process is notated as

$$f(\boldsymbol{\mu}) \sim \mathcal{GP}(m(\boldsymbol{\mu}), k(\boldsymbol{\mu}, \boldsymbol{\mu}')). \tag{2.25}$$

GPR may be expressed either in parametric or non-parametric form. Bayes' theorem is central to both forms of GPR, and Bayesian linear inverse problems generally, and is expressed in context of parametric GPR in Equation 2.26.

$$\text{posterior} = \frac{\text{likelihood} \times \text{prior}}{\text{evidence}}, \quad p(\mathbf{w}|\mathbf{y}, \mathbf{M}) = \frac{p(\mathbf{y}|\mathbf{w}, \mathbf{M})p(\mathbf{w})}{p(\mathbf{y}|\mathbf{M})}. \tag{2.26}$$

Both parametric and non-parametric scenarios rely on Gaussian priors and the conditional properties of multivariate-Gaussian distributions, i.e. Gaussian Processes, to infer posterior distributions of either:

1. the weights for a linear model $\mathbf{w}$ (parametric), or
2. the predicted function values directly (non-parametric).

Beginning with parametric GPR, consider a linear model for the response, same form as Section 2.3.1 for a generalized feature transformation, written as

$$\begin{aligned} y(\boldsymbol{\mu}; \mathbf{w}) &= f(\boldsymbol{\mu}; \mathbf{w}) + \epsilon = \boldsymbol{\phi}(\boldsymbol{\mu})^T \mathbf{w} + \epsilon && (2.27) \\ & && (2.28) \end{aligned}$$

where noise $\boldsymbol{\epsilon}$ is i.i.d Gaussian, $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \boldsymbol{\Gamma})$, with $\boldsymbol{\Gamma} = \sigma_n^2 \mathbf{I}_M \in \mathbb{R}^{M \times M}$. Additionally, take a Gaussian prior on the weights $\mathbf{w} \sim \mathcal{N}(\mathbf{m}_0, \boldsymbol{\Sigma}_0)$, where $\mathbf{m}_0 \in \mathbb{R}^p$ is the prior mean and $\boldsymbol{\Sigma}_0 \in \mathbb{R}^{p \times p}$ is the prior covariance. Often times the prior mean is taken to be zero, $\mathbf{m}_0 = \mathbf{0}$. Random variable Y describing $\mathbf{y}$ is a linear combination of Gaussian random variables $\mathbf{w}$ and $\boldsymbol{\epsilon}$ and thus Y is also Gaussian,

$$p_Y(\mathbf{y}) \sim \mathcal{N}(\boldsymbol{\Phi}\mathbf{m}_0, \boldsymbol{\Phi}\boldsymbol{\Sigma}_0\boldsymbol{\Phi}^T + \boldsymbol{\Gamma}). \tag{2.29}$$

Further, the Gaussian priors induce a Gaussian-Process prior over predictive function $f(\boldsymbol{\mu}; \mathbf{w})$, the model without noise, and may be written as

$$f(\boldsymbol{\mu}; \mathbf{w}) \sim \mathcal{GP}(\boldsymbol{\phi}(\boldsymbol{\mu})^T \mathbf{m}_0, \boldsymbol{\phi}(\boldsymbol{\mu})^T \boldsymbol{\Sigma}_0 \boldsymbol{\phi}(\boldsymbol{\mu}')). \tag{2.30}$$

This is central to non-parametric GPR, discussed after parametric GPR.

Given Equation 2.29, the properties of multivariate Gaussian distributions may be used to write the joint distribution of $\mathbf{y}$ and $\mathbf{w}$, $p(\mathbf{y}, \mathbf{w})$, which is then marginalized to obtain the posterior distribution $p(\mathbf{w}|\mathbf{y})$. The joint distribution is expressed as

$$\begin{bmatrix} \mathbf{w} \\ \mathbf{y} \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mathbf{m}_0 \\ \mathbf{\Phi m}_0 \end{bmatrix}, \begin{bmatrix} \mathbf{\Sigma}_0 & \mathbf{\Sigma}_0 \mathbf{\Phi}^T \\ \mathbf{\Phi \Sigma}_0 & \mathbf{\Phi \Sigma}_0 \mathbf{\Phi}^T + \mathbf{\Gamma} \end{bmatrix} \right), \tag{2.31}$$

leading to posterior mean $\mathbf{m}_{\text{post}}$ and covariance $\mathbf{\Sigma}_{\text{post}}$ which are obtained from conditional properties of multivariate Gaussians.

$$\mathbf{m}_{\text{post}} = \mathbf{m}_0 + \mathbf{\Sigma}_0 \mathbf{\Phi}^T (\mathbf{\Phi \Sigma}_0 \mathbf{\Phi}^T + \mathbf{\Gamma})^{-1} (\mathbf{y} - \mathbf{\Phi m}_0) \tag{2.32}$$

$$\mathbf{\Sigma}_{\text{post}} = \mathbf{\Sigma}_0 - \mathbf{\Sigma}_0 \mathbf{\Phi}^T (\mathbf{\Phi \Sigma}_0 \mathbf{\Phi}^T + \mathbf{\Gamma})^{-1} \mathbf{\Phi \Sigma}_0 \tag{2.33}$$

That is, the posterior distribution of the weights is

$$p(\mathbf{w}|\mathbf{y}, \mathbf{M}) \sim \mathcal{N}(\mathbf{m}_{\text{post}}, \mathbf{\Sigma}_{\text{post}}) \tag{2.34}$$

and thus the updated predictive function over the training data is now distributed as

$$p_Y(\mathbf{y}) \sim \mathcal{N}(\mathbf{\Phi m}_{\text{post}}, \mathbf{\Phi \Sigma}_{\text{post}} \mathbf{\Phi}^T + \mathbf{\Gamma}). \tag{2.35}$$

Predictions on $\boldsymbol{\mu}_* \notin \mathcal{D}$ frequently exclude noise ϵ, and thus the predictive distribution over unseen data points is given as

$$p_Y(\hat{y}(\boldsymbol{\mu}_*)) \sim \mathcal{N}\big(\boldsymbol{\phi}(\boldsymbol{\mu}_*)^T \mathbf{m}_{\text{post}}, \boldsymbol{\phi}(\boldsymbol{\mu}_*)^T \mathbf{\Sigma}_{\text{post}} \boldsymbol{\phi}(\boldsymbol{\mu}_*)\big), \tag{2.36}$$

and usually the mean and covariance are taken as the prediction and uncertainty. That is, a model prediction is written as

$$\hat{y}(\boldsymbol{\mu}) = \boldsymbol{\phi}(\boldsymbol{\mu})^T \mathbf{m}_{\text{post}}, \tag{2.37}$$

and corresponds directly to the least-squares linear model, Equation 2.16, where $\mathbf{w}$ and $\mathbf{m}_{\text{post}}$ both represent the weights found by different methods. This may be extended to predictions at multiple points outside the training set simultaneously by constructing a feature matrix $\mathbf{\Phi}_*$.

Next non-parametric GPR is developed, where inference occurs *directly in predicted function space*, without explicitly referencing or determining model weights $\mathbf{w}$ / $\mathbf{m}_{\text{post}}$. Given Equation 2.30, the prior predictive function mean and covariance functions can be readily seen,

$$\begin{aligned} m(\boldsymbol{\mu}) &\Leftrightarrow \boldsymbol{\phi}(\boldsymbol{\mu})^T \mathbf{m}_0 && (2.38) \\ k(\boldsymbol{\mu}, \boldsymbol{\mu}') &\Leftrightarrow \boldsymbol{\phi}(\boldsymbol{\mu}) \mathbf{\Sigma}_0 \boldsymbol{\phi}(\boldsymbol{\mu}')^T. && (2.39) \end{aligned}$$

When a prediction is needed at $\boldsymbol{\mu}_*$, the joint distribution of the data and prediction $p(\mathbf{y}, f(\boldsymbol{\mu}_*))$ is written and the properties of Gaussians are again used to marginalize to obtain the mean and variance of the predictive distribution $p(f(\boldsymbol{\mu}_*)|\mathbf{y})$. The joint distribution is

$$\begin{bmatrix} \mathbf{y} \\ f(\boldsymbol{\mu}_*) \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} m(\mathbf{M}) \\ m(\boldsymbol{\mu}_*) \end{bmatrix}, \begin{bmatrix} k(\mathbf{M}, \mathbf{M}) + \mathbf{\Gamma} & k(\mathbf{M}, \boldsymbol{\mu}_*) \\ k(\boldsymbol{\mu}_*, \mathbf{M}) & k(\boldsymbol{\mu}_*, \boldsymbol{\mu}_*) \end{bmatrix} \right), \tag{2.40}$$

where matrix $\mathbf{M} \in \mathbb{R}^{n_\mu \times M}$ collects all input vectors $\boldsymbol{\mu} \in \mathcal{D}$. Matrix $k(\mathbf{M}, \mathbf{M}) \in \mathbb{R}^{M \times M}$ collects the covariance function over training points, where element i, j is $k(\boldsymbol{\mu}^{(i)}, \boldsymbol{\mu}^{(j)})$. Then column vector $k(\mathbf{M}, \boldsymbol{\mu}_*) \in \mathbb{R}^{M \times 1}$ collects the covariance function between the point of interest $\boldsymbol{\mu}_*$ and all training points in the dataset, as does row vector $k(\boldsymbol{\mu}_*, \mathbf{M}) \in \mathbb{R}^{1 \times M}$, while $k(\boldsymbol{\mu}_*, \boldsymbol{\mu}_*)$ is a scalar. Then applying the rules for conditional Gaussians to obtain the posterior predictive distribution and variance functions for $\boldsymbol{\mu}_* \notin \mathcal{D}$ yields the following.

$$\begin{aligned} \mathbb{E}[f(\boldsymbol{\mu}_*)|\mathbf{y}] &= m(\boldsymbol{\mu}_*) + k(\boldsymbol{\mu}_*, \mathbf{M})\big(k(\mathbf{M}, \mathbf{M}) + \mathbf{\Gamma}\big)^{-1}(\mathbf{y} - m(\mathbf{M})) && (2.41) \\ \text{Var}(f(\boldsymbol{\mu}_*), f(\boldsymbol{\mu}_*)|\mathbf{y}) &= k(\boldsymbol{\mu}_*, \boldsymbol{\mu}_*) - k(\boldsymbol{\mu}_*, \mathbf{M})\big(k(\mathbf{M}, \mathbf{M}) + \mathbf{\Gamma}\big)^{-1} k(\mathbf{M}, \boldsymbol{\mu}_*) && (2.42) \end{aligned}$$

Often times the mean function is taken to be the zero function, and this simplifies the prediction, letting $f_* := \mathbb{E}[f(\boldsymbol{\mu}_*)|\mathbf{y}]$, then

$$f_* = k(\boldsymbol{\mu}_*, \mathbf{M})\big(k(\mathbf{M}, \mathbf{M}) + \boldsymbol{\Gamma}\big)^{-1}\mathbf{y} \tag{2.43}$$

is the predicted function value. This may be interpreted as a linear combination of M kernel functions $k(\boldsymbol{\mu}_*, \mathbf{M})$, each centered on a training data point, with coefficients given by $\big(k(\mathbf{M}, \mathbf{M}) + \boldsymbol{\Gamma}\big)^{-1}\mathbf{y}$.

Global or local basis expansions may be used to construct features $\boldsymbol{\phi}(\boldsymbol{\mu})$, where common global basis functions are those mentioned previously, and include monomials as in the curve-fitting example, along with Chebyshev, Hermite, or Lagrange polynomials. The listed non-monomial bases are advantageous over monomials as they are orthogonal. Local basis functions are usually symmetric functions which are centered on the data points and decay towards zero far from the data points. This includes the commonly used squared-exponential kernel, written for scalar inputs as

$$k_{\mathrm{SE}}(\mu, \mu'; \gamma, \ell) = \gamma \exp\left[-\frac{(\mu - \mu')^2}{2\ell^2}\right], \tag{2.44}$$

where γ and ℓ are hyperparameters of the kernel. Multidimensional kernels may be constructed as products or summations of scalar kernels along each axis. Many other kernel functions are widely used, with The Kernel Cookbook providing a good overview [120], along with greater details on constructing kernels with different properties.

2.3.2.1 Application in Vehicle Aerodynamics

GPR is normally considered only in the context of predicting a scalar output quantity, and extending GPR to account for multi-dimensional output quantities while accounting for the covariance of those output quantities is non-trivial, and an active area of research [121]. In that scenario the kernel function becomes matrix valued instead of scalar valued. The alternative is to use separate GPR models for each output, and some existing python libraries such as `scikit-learn` offer a GPR implementation which will do so for you automatically. This method is pursued here, where the `scikit-learn` [122] GPR toolkit is used to regress snapshots of of a single flow quantity using the interpolated vehicle aerodynamics dataset, described in Section 2.1. Using that dataset, consider the ith flattened snapshot of a single flow quantity of the interpolated solution lying on a 150×150 Cartesian grid, $\mathbf{y}_i \in \mathbb{R}^{(22{,}500)}$. Construct a snapshot matrix with snapshots stored as rows,

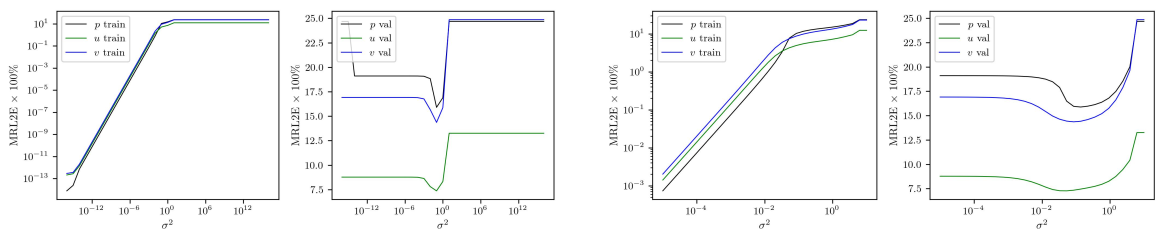
$$\mathbf{Y} = \begin{bmatrix} — \mathbf{y}_1^T — \\ — \mathbf{y}_2^T — \\ \vdots \\ — \mathbf{y}_M^T — \end{bmatrix} \in \mathbb{R}^{M\times 22{,}500}, \tag{2.45}$$

then Equation 2.29 becomes simply

$$\mathbf{f}_* = k(\boldsymbol{\mu}_*, \mathbf{M})\big(k(\mathbf{M},\mathbf{M}) + \boldsymbol{\Gamma}\big)^{-1}\mathbf{Y}. \tag{2.46}$$

In the `sklearn.gaussian_process.GaussianProcessRegressor` implementation, the term $\mathbf{C} = \big(k(\mathbf{M},\mathbf{M}) + \boldsymbol{\Gamma}\big)^{-1}\mathbf{Y} \in \mathbb{R}^{M\times 22{,}500}$ is computed first and stored, following Algorithm 2.1 of reference [123]. The lower Cholesky decomposition is computed, $\mathbf{L}\mathbf{L}^* = \big(k(\mathbf{M},\mathbf{M}) + \boldsymbol{\Gamma}\big)$, and then used to solve the system of equations $\mathbf{L}\mathbf{L}^*\mathbf{C} = \mathbf{Y}$ using `scipy.linalg.cho_solve` from the `SciPy` [39] python library. Then predictions are given as row vectors simply by $\mathbf{f}_* = k(\boldsymbol{\mu}_*, \mathbf{M})\mathbf{C}$.

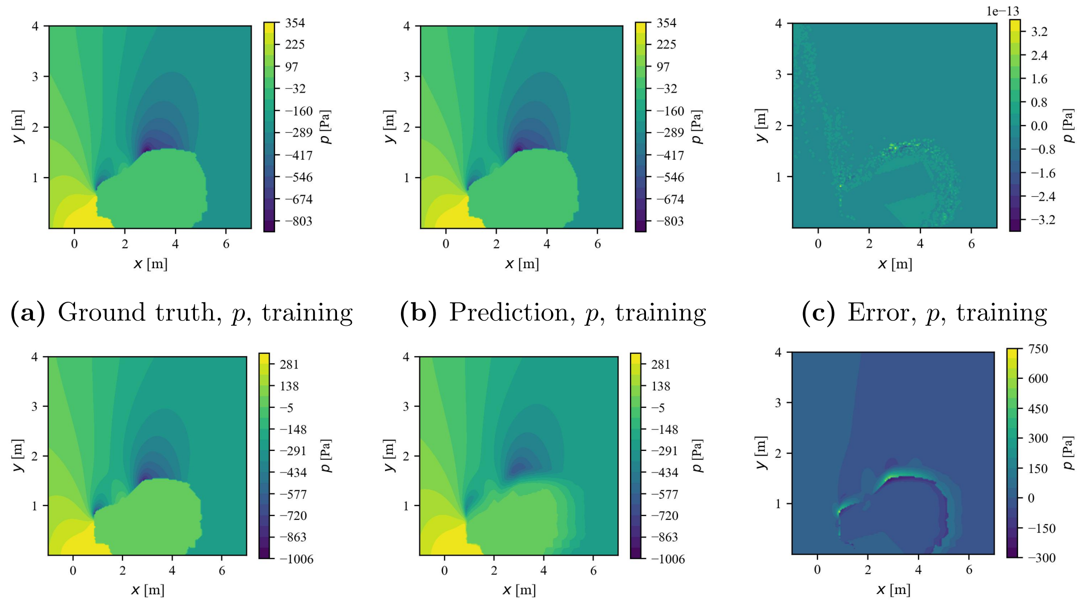
A vehicle speed of 90 kph was, with flow quantities of static pressure, x-velocity, and y-velocity, and a training validation split of 80/20 was applied to the 124 solutions giving $M = 99$ in the training set. The squared-exponential (radial basis function) kernel was used and the noise level was varied as $10^{-16} \leq \sigma_n^2 \leq 10^{16}$, where $\boldsymbol{\Gamma} = \sigma_n^2\mathbf{I}_M$, to understand the behavior on the validation set. The MRL2E against σ^2 over training and validation sets over this broad noise range is given in Figure 2.3a, and zoomed in (with a finer sweep) on validation minima in 2.3b.



(a) (b)

Figure 2.3: MRL2E versus data-noise level σ_n^2 (a): coarsely over a broad range and (b): more finely over a narrower range where validation-set minima are seen.

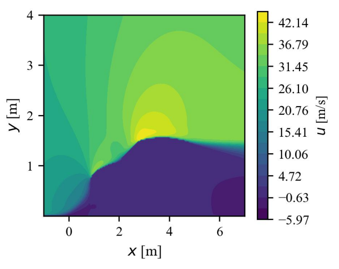
In the essentially noise-free regime, training-set performance is very good as would be expected since GPR returns exact values from the training set when $\sigma_n^2 = 0$. The training error increases with σ_n^2 until about $\sigma_n^2 = 1$, where it remains essentially flat after. The validation errors are considerably worse in all regimes, with slight minima seen in each curve between $10^{-2} \leq \sigma_n^2 \leq 10^{-1}$. Figures 2.4 and 2.5 shows pressure-field and x-velocity-field predictions for the training and validation sets in the very-low noise regime, with $\sigma_n^2 = 10^{-16}$, where the training set predictions match very well while the validation predictions do not, especially in regions near the vehicle surface. Note that the ground truth data is masked during pre-processing, while the prediction and error contours are not.



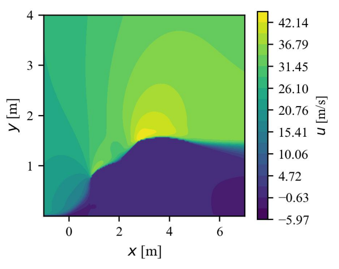
(a) Ground truth, p, training **(b)** Prediction, p, training **(c)** Error, p, training

(d) Ground truth, p, validation **(e)** Prediction, p, validation **(f)** Error, p, validation

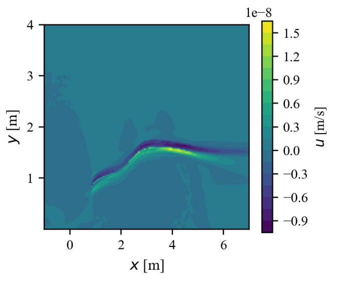
Figure 2.4: Example training-set (a-c) and validation-set (d-f) pressure-field predictions in the essentially noise-free regime, with $\sigma_n^2 = 10^{-16}$.



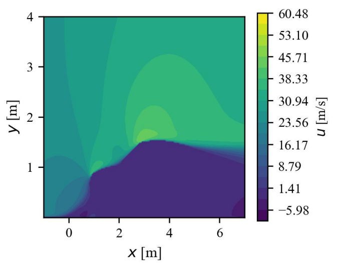
(a) Ground truth, u, training



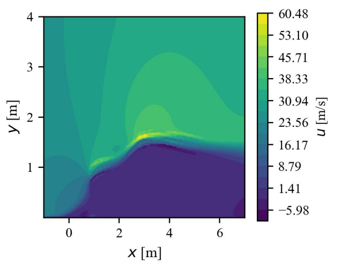
(b) Prediction, u, training



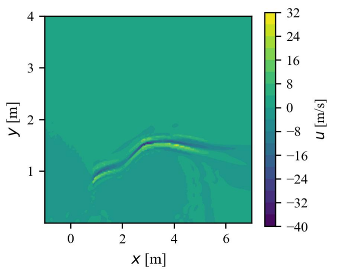
(c) Error, u, training



(d) Ground truth, u, validation



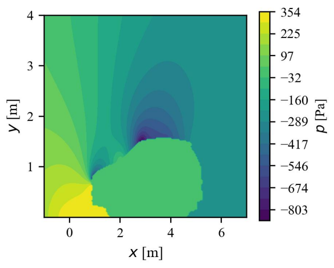
(e) Prediction, u, validation



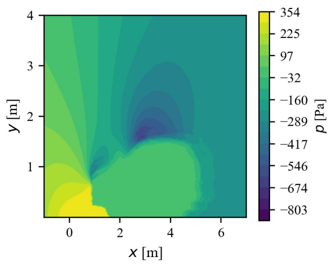
(f) Error, u, validation

Figure 2.5: Example training-set (a-c) and validation-set (d-f) x-velocity-field predictions in the essentially noise-free regime, with $\sigma_n^2 = 10^{-16}$.

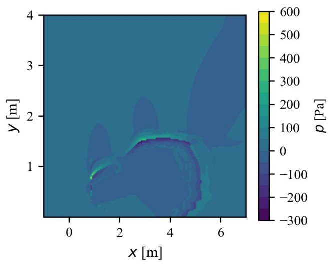
Figures 2.6 and 2.7 show training and validation predictions for the same vehicle shapes when the validation-best-MRL2E noise level was selected. The validation predictions are modestly improved, but the training predictions are vastly degraded and are of similar poor quality as the validation set. Note that each GPR model, for every noise level and flow variable, has its squared-exponential kernel length scale optimized per the log-marginal-likelihood during fitting.



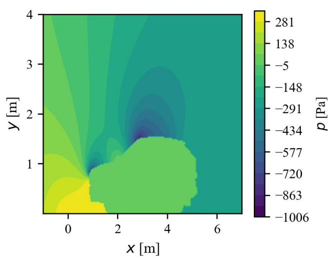
(a) Ground truth, p, training



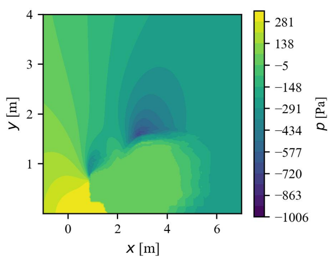
(b) Prediction, p, training



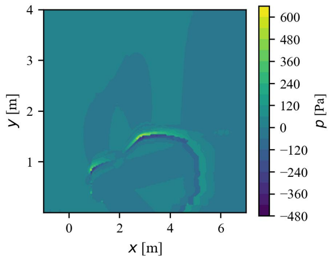
(c) Error, p, training



(d) Ground truth, p, validation

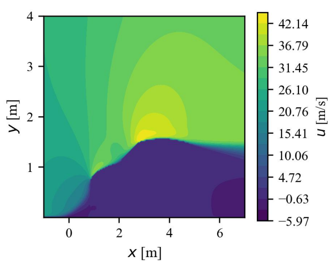


(e) Prediction, p, validation

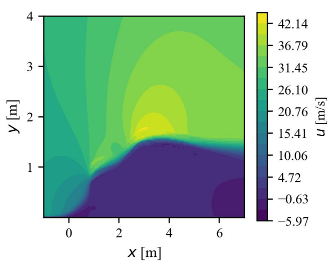


(f) Error, p, validation

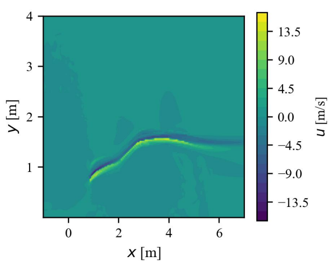
Figure 2.6: Example training-set (a-c) and validation-set (d-f) pressure field predictions when the MRL2E validation-best noise level was used.



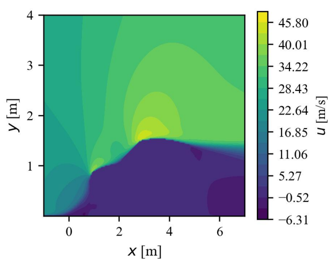
(a) Ground truth, u, training



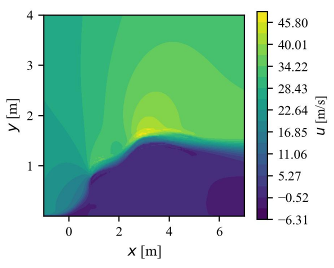
(b) Prediction, u, training



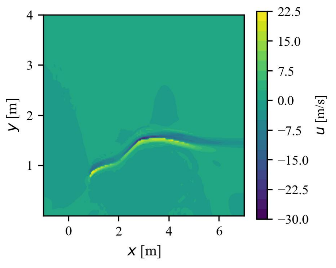
(c) Error, u, training



(d) Ground truth, u, validation



(e) Prediction, u, validation



(f) Error, u, validation

Figure 2.7: Example training-set (a-c) and validation-set (d-f) x-velocity field predictions when the MRL2E validation-best noise level was used.

Thus in this scenario, use of GPR without more involved treatment does not lead to a satisfactory, generalizable model. Using the signed-distance field as the input feature to explicitly give the vehicle shape does not lead to improved results. In the dataset's unprocessed, non-interpolated form, it cannot be processed by GPR as a snapshot since each solution has a varying number of mesh points and differing connectivity. So even if GPR performed well at predicting unseen snapshots, it would still be unsatisfactory due to the required interpolation.

2.3.3 Proper Orthogonal Decomposition

POD is a popular choice for selecting the trail/test basis used in intrusive projection-based reduced-order-modeling schemes, as presented in Section 1.2.3. POD-NN develops a similar POD basis for non-intrusive ROMs, see Section 1.2.4, where the basis coefficients are generated for an unseen parameter set using an ANN. As was mentioned previously, POD methods require consistent meshes across all cases considered in order to assemble the snapshot matrix. Thus these methods are not effective for handling data lying on unstructured meshes with varying dimension and topology, but they may still be evaluated in the context of the interpolated vehicle dataset of Section 2.1.

A single vehicle speed of 90 kph is considered and each flow quantity is treated separately, with 95 cases used in the training dataset to construct the snapshot matrices. Each output field is z-score normalized, per Section 2.2.1 before application of Equations 1.62-1.64. Before use with a method such as POD-NN, the basis $\mathbf{\Phi}$ may be evaluated in reconstructing the training and validation dataset, using Equations 1.65 and 1.66. A mean-centered data-snapshot matrix $\mathbf{X}_m$ may be assembled for each the training and validation groups, but only the training set used in computing the basis. Note that this is not a truly predictive task as the solutions are needed to compute the basis coefficients; in POD-NN a separate mapping between design variables and basis coefficients via ANN is needed. Thus evaluating the performance this way provides an upper bound on the possible predictive performance. The MRL2E for each flow quantity versus the number of retained POD modes is shown in Figure 2.8.

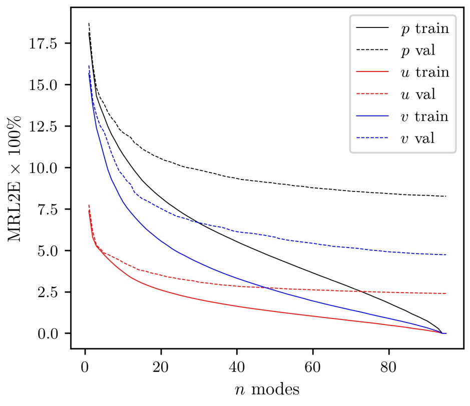
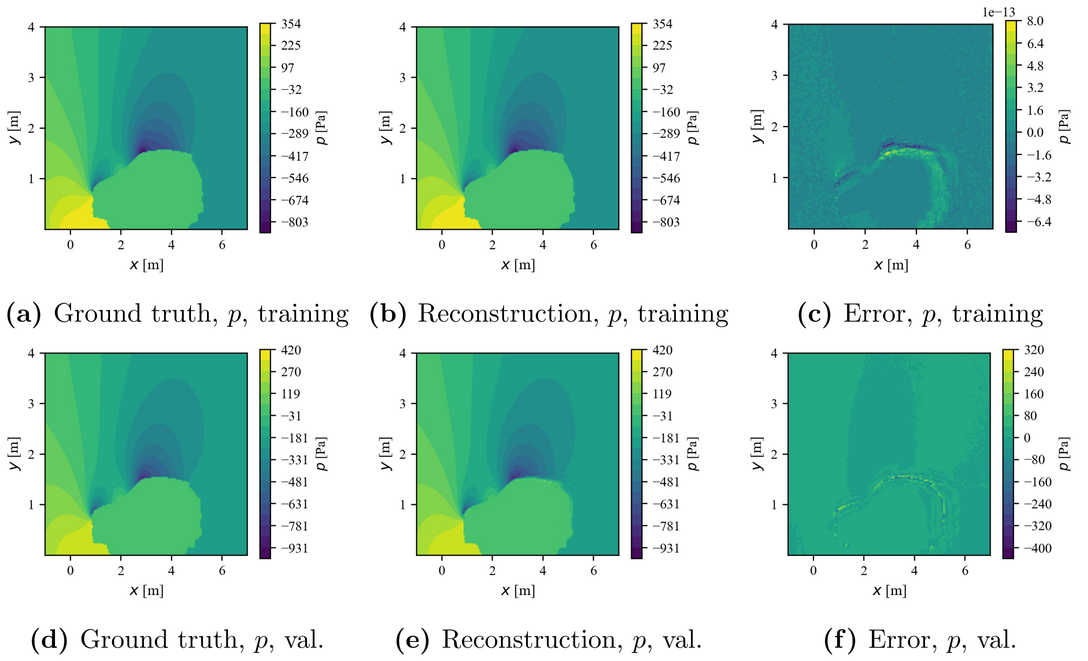


Figure 2.8: The variation in reconstruction MRL2E for each flow quantity of the interpolated vehicle dataset for the training and validation groups using a POD basis with a varying number of modes.

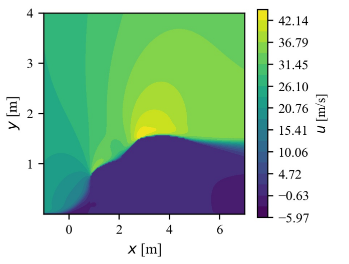
This shows that the error in reconstructing the training dataset does is fact go to zero as expected, but only when 94 or 95 modes are retained. The reconstruction errors for the validation group generally decrease as the number of retained POD modes is increased, but the validation reconstruction errors essentially flat-line after a certain point.



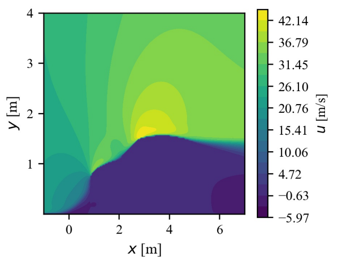
(a) Ground truth, p, training **(b)** Reconstruction, p, training **(c)** Error, p, training

(d) Ground truth, p, val. **(e)** Reconstruction, p, val. **(f)** Error, p, val.

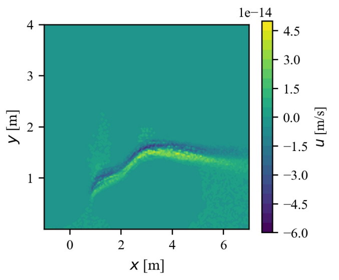
Figure 2.9: Example training-set (a-c) and validation-set (d-f) pressure-field reconstructions using a non-truncated POD basis with 95 modes.



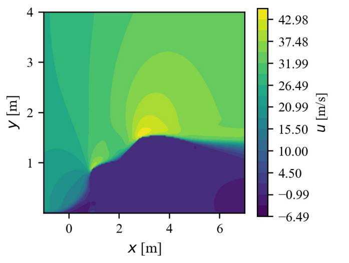
(a) Ground truth, u, training



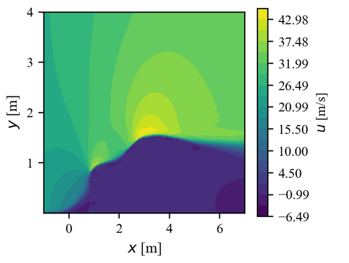
(b) Reconstruction, u, training



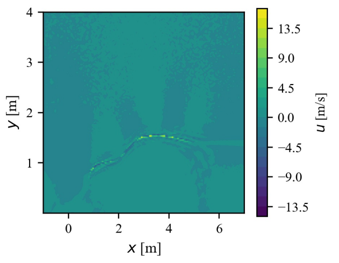
(c) Error, u, training



(d) Ground truth, u, val.



(e) Reconstruction, u, val.



(f) Error, u, val.

Figure 2.10: Example training-set (a-c) and validation-set (d-f) x-velocity-field reconstructions using a non-truncated POD basis with 95 modes.

It is re-emphasized that the validation-group errors are in *reconstruction* not prediction. A scheme such as POD-NN is required for a truly predictive scenario, see Section 1.2.4. Again, as with GPR, these results rely on snapshot matrices which require lossy interpolation of the ground-truth data.

2.3.4 Dense Neural Networks

The perceptron is the most fundamental building-block of ANN models and is a simplified mathematical model of a neuron [124] and in its most basic form is a binary classifier.The incoming signals are represented by a vector of inputs $\mathbf{x} \in \mathbb{R}^{n_x}$, and a weight is placed on each signal, collected in vector $\mathbf{w} \in \mathbb{R}^{n_x}$. The perceptron sums the weighted input signals, optionally adds a bias $b \in \mathbb{R}$, and then applies a non-linear activation function $\sigma()$ to generate the scalar output signal $y \in \mathbb{R}$ as

$$y = \sigma\big(\mathbf{w}^T\mathbf{x} + b\big). \tag{2.47}$$

The original perceptron used a heaviside step function, but other more commonly used activation functions include the logistic function (sigmoid), hyperbolic tangent, swish, and rectified linear unit (ReLU). A perceptron is a single-layer neural network, while multi-layer perceptron (MLP) models consist of several layers, each containing multiple perceptrons, also commonly referred to as nodes or neurons. Each node is densely connected to every node in the preceding and successive layers. The formula for a single dense layer with H nodes is given by

$$f_{\text{dense}}(\mathbf{h}; \mathbf{W}, \mathbf{b}) = \sigma.\big(\mathbf{W}\mathbf{h} + \mathbf{b}\big), \tag{2.48}$$

where matrix $\mathbf{W} \in \mathbb{R}^{H \times \dim(\mathbf{h})}$ collects the weights for each node as its rows, vector $\mathbf{b} \in \mathbb{R}^H$ holds the biases, and $\sigma.()$ represents element-wise application of the activation function. Given an output target $\mathbf{y} \in \mathbb{R}^{n_y}$, an MLP is a composite function which maps $f : \mathbb{R}^{n_x} \to \mathbb{R}^{n_y}$, where the input is sequentially processed into the output, and a general formula for propagating the hidden state through the ith layer is given by

$$\mathbf{h}^{(i)} = \sigma.\big(\mathbf{W}^{(i)}\mathbf{h}^{(i-1)} + \mathbf{b}^{(i)}\big). \tag{2.49}$$

This applies generally for $i \geq 1$ by considering the input vector to be the zeroth hidden state, $\mathbf{x} = \mathbf{h}^{(0)}$. Linear activation functions are commonly used for the output layer of predictive networks, while the softmax function is often used for classification.

All weight matrices and bias vectors may be collected in a set $\theta = \left\{\mathbf{W}^{(i)}, \mathbf{b}^{(i)}\right\}_{i=1}^{n_L+1}$ when there are n_L hidden layers.Let $\theta^{(i)} = \left\{\mathbf{W}^{(i)}, \mathbf{b}^{(i)}\right\}$ be the trainable weights for the ith layer. When the parentheses are not included in the superscript, θ^i, then this represents all network weights at the ith iteration of training. The elements of θ are trainable parameters, initialized randomly (though carefully) which are updated during training, usually using a variant of stochastic gradient descent, an unconstrained optimization algorithm. In predictive tasks the loss function is usually taken to be the mean-squared-error (MSE) loss. Given inputs $\mathbf{x}$, let the network prediction be written as $\hat{\mathbf{y}}(\mathbf{x}_i; \theta)$, or $\hat{\mathbf{y}}_i(\theta)$, then the training problem of finding optimal weights θ^* may be expressed as

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \frac{1}{M} \sum_{i=1}^{M} \|\hat{\mathbf{y}}_i(\theta) - \mathbf{y}_i)\|_2^2, \tag{2.50}$$

where the argument $\mathcal{J}(\theta) = \frac{1}{M}\sum_{i=1}^{M} \|\hat{\mathbf{y}}_i(\theta) - \mathbf{y}_i)\|_2^2$ is known as the loss function, analogous to the objective function in general optimization. Gradient descent is an iterative numerical method for solving non-linear optimization problems, where the weights are updated from one iteration to the next according to the formula

$$\theta^{j+1} = \theta^j - \alpha \nabla \mathcal{J}(\theta^j), \tag{2.51}$$

where α is the step size which is called the learning rate in deep-learning literature. The gradient $\nabla\mathcal{J}(\theta)$ has terms for each $\mathbf{W}^{(\mathbf{i})}$ and $\mathbf{b}^{(i)}$ and for a simple network such as an MLP those terms can be worked out analytically. The derivation is straight-forward although somewhat tedious, and a general rule may be found using the chain rule and backpropagating the derivatives through the network using previously computed terms [125]. Modern codes generally use automatic differentiation to compute the derivatives, using open-source libraries such as `tensorflow` [126], `PyTorch` [127], or JAX [128]. This becomes almost a necessity when more complex network architectures are considered. The "stochastic" part of stochastic gradient descent comes from splitting the dataset into minibatches and performing optimizer updates on the model weights after each minibatch, instead of processing the whole training dataset between updates as Equation 2.50 suggests. The minibatches are randomly generated and either shuffled or regenerated between dataset iterations, known as epochs.

2.3.5 Autoencoders

Autoencoders are a class of model characterized by a bottleneck structure for learning a meaningful, lower-dimensional latent representation of the data. These models have wide-ranging applications in supervised and unsupervised machine learning tasks, including clustering, classification, denoising, and generative modeling [129], and additionally serve as the basis for many surrogate modeling techniques. An autoencoder consists of two functions called the encoder and decoder. The encoder takes as input a data snapshot and produces a latent code vector which is of smaller dimension than the snapshot. The decoder in turn takes as input the latent vector and seeks to reproduce the input data. That is, given a data space $\mathcal{X} \subset \mathbb{R}^{n_x}$ and a latent space $\mathcal{Z} \subset \mathbb{R}^{n_z}$ where $n_x \geq n_z$, the encoder is written as

$$\Phi : \mathcal{X} \to \mathcal{Z}; \quad \Phi(\mathbf{x}) = \mathbf{z}, \tag{2.52}$$

and the decoder as

$$\Psi : \mathcal{Z} \to \mathcal{X}; \quad \Psi(\mathbf{z}) = \hat{\mathbf{x}} \approx \mathbf{x}. \tag{2.53}$$

The autoencoder output is then the composition of decoder and encoder, $\hat{\mathbf{x}} = \Psi \circ \Phi(\mathbf{x})$. The autoencoder problem for obtaining the encoder and decoder may then be defined as

$$\Phi, \Psi = \underset{\Phi,\Psi}{\operatorname{argmin}} \sum_{i=1}^{n} \Delta\big(\Psi \circ \Phi(\mathbf{x}_i), \mathbf{x}_i\big), \tag{2.54}$$

adapted from [130], where n data snapshots are present, and Δ is a dissimilarity function or reconstruction loss term. The L_p norms are possible dissimilarity functions, with the L_2 or L_2^2 norm being commonly used. In the described setting, the problem is auto-associative, meaning that the inputs and outputs are identical. In some cases the problem is non auto-associative, meaning that some external target $\mathbf{y} \neq \mathbf{x}$ is predicted by the network. In this case, define a third space $\mathcal{Y} \subset \mathbb{R}^{n_y}$ and redefine the decoder as

$$\Psi : \mathcal{Z} \to \mathcal{Y} : \quad \Psi(\mathbf{z}) = \hat{\mathbf{y}} \approx \mathbf{y}, \tag{2.55}$$

and the autoencoder problem as

$$\Phi, \Psi = \underset{\Phi,\Psi}{\operatorname{argmin}} \sum_{i=1}^{n} \Delta\big(\Psi \circ \Phi(\mathbf{x}_i), \mathbf{y}_i\big). \tag{2.56}$$

In this context the dimensions of input and output spaces are not required to be the same and this is commonly the case. This second class of autoencoders may be termed 'predictive' autoencoders. Typically autoencoders are composed of neural networks, but they were first proposed in the context of boolean networks [131], while linear autoencoders are analogous to principal components analysis [132]. Autoencoders may be constructed using any type of neural network for the encoder and decoder, but they are typically constructed as mirror images of another with dense and convolutional layers being most common. Predictive autoencoders are widely used in the context of surrogate modeling for scientific problems [76, 80, 78].

2.3.6 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a powerful ANN variant responsible for many state-of-the-art advances over the last 10-15 years, becoming particularly popular after winning the 2012 Imagenet competition [133]. CNNs were first conceived as the neocognitron with vision tasks in mind, with a structure inspired by the connection of neurons in a cat's visual cortex [134, 135]. Specifically, not every node is connected to every other node of the adjacent hidden layers as they are in MLPs, instead only nodes in a *local receptive field* are connected. A local receptive field is comprised of inputs which are "close" to one another, and defining "close" forces structure on the inputs and hidden states. For images, this is natural, as the connectedness or distance may be simply scaled differences in pixel coordinates, given the Cartesian structure. For a 2D image, the local receptive field is an area spanned by a filter matrix $\mathbf{K} \in \mathbb{R}^{K_1 \times K_2}$. Convolution may be thought of as a sliding dot product between input image $\mathbf{H} \in \mathbb{R}^{D_1 \times D_2}$ and flipped kernel matrix $\mathbf{K}$[1]. Note that the kernel matrix may not be larger than the input image. Convolution is represented with an asterisk or star, and in neural networks a non-linear activation is also applied, so a convolutional layer with a single kernel may be written as

$$f_{\text{conv}}(\mathbf{H}; \mathbf{K}, \mathbf{B}) = \sigma.\big(\mathbf{H} * \mathbf{K} + \mathbf{B}\big), \tag{2.57}$$

where $\mathbf{B}$ is a bias matrix with dimensions given later per Equation 2.61. The bias matrix is often a single repeated number, tiled to the appropriate dimensions. As with MLPs, an input image is processed sequentially into hidden states, with general forward-propagation

[1]The flipping is due to sign convention in defining convolution.

given by

$$\mathbf{H}^{(i)} = \sigma.\big(\mathbf{H}^{(i-1)} * \mathbf{K}^{(i)} + \mathbf{B}^{(i)}\big), \tag{2.58}$$

where $\mathbf{H}$ is the hidden state or feature map.

Zero-indexed two-dimensional convolution of feature map $\mathbf{H}$ with kernel $\mathbf{K}$ may be written as

$$\mathbf{Z}_{i,j} = \sum_{m=0}^{K_1-1} \sum_{n=0}^{K_2-1} \mathbf{H}_{i+m,j+n} \mathbf{K}_{m,n}. \tag{2.59}$$

CNNs parameterize the entries of the filter matrices, and when there are multiple input channels then the kernel is actually three-dimensional but the sliding movement is only in two dimensions. In that scenario, when there are C channels, the input image or hidden state is $\mathbf{H} \in \mathbb{R}^{D_1 \times D_2 \times C}$ and the kernel is $\mathbf{K} \in \mathbb{R}^{K_1 \times K_2 \times C}$, then the convolution has an additional summation across the channels, given as

$$\mathbf{Z}_{i,j} = \sum_{c=0}^{C-1} \sum_{m=0}^{K_1-1} \sum_{n=0}^{K_2-1} \mathbf{H}_{i+m,j+n,c} \mathbf{K}_{m,n,c}. \tag{2.60}$$

Further, if there are P channels in the next hidden state, then convolution is performed with P different kernels. This may be thought of as applying Equation 2.59 C times with C different two-dimensional kernels, and then aligning and adding to attain the final result for a single new channel. If there are P new channels in the next hidden state, then this is repeated P times with different kernels.

The size of the image changes after a convolution operation, and many times square images and kernels are used but that is not always the case, see the decoder-CNNs (DCNNs) used in Chapter 5. Strided convolution is also common, where the filter matrix moves more than one pixel at a time and is a form of sub-sampling. When strided convolutions are used, the stride length along each axis must be less than the kernel dimension or portions of the input will be skipped. It is also common to pad the images, often with zeros, to avoid or control changes in feature-map size, and pooling operations are also frequently applied, where the pixels in a local receptive field are down-sampled with average-pooling and max-pooling commonly used. Consider an input image or feature map for layer i of size $D_1^{(i-1)} \times D_2^{(i-1)}$, a kernel of size $K_1^{(i)} \times K_2^{(i)}$, strides of length

$S_1^{(i)} \times S_2^{(i)}$, and padding of size $P_1^{(i)} \times P_2^{(i)}$ added to the images, then the output image dimension along axis $j = 1, 2$ is given by

$$D_j^{(i)} = \text{floor}\left(\frac{D_j^{(i-1)} - K_j^{(i)} + 2P_j^{(i)}}{S_j^{(i)}} + 1\right). \tag{2.61}$$

The bias matrix of Equation 2.57 has the same dimensions as the resulting output. Generally convolutional layers reduce the image dimension, or include padding to keep it the same or control the reduction.

One-dimensional and three-dimensional CNNs are also commonly used, and Equation 2.59 is easily generalized for those scenarios as

$$\mathbf{z}_i = \sum_{m=0}^{K_1-1} \mathbf{h}_{i+m}\mathbf{k}_m \tag{2.62}$$

$$\mathbf{Z}_{i,j,k} = \sum_{m=0}^{K_1-1}\sum_{n=0}^{K_2-1}\sum_{o=0}^{K_3-1} \mathbf{H}_{i+m,j+n,k+o}\mathbf{K}_{m,n,o} \tag{2.63}$$

respectively, with an additional axis added and summation across channels included when necessary, similar to Equation 2.60. The entries of the kernels and bias matrices are the trainable parameters of a convolutional layer. Further, Equation 2.61 applies for each axis $j = 1, 2, 3$ in an analogous fashion. Using 2D convolutions as an example, if there are C_{in} channels in the incoming feature map and C_{out} channels in the output feature map, then there are C_{out} three-dimensional kernels, each with dimension $K_1 \times K_2 \times C_{\text{in}}$. Thus, the total number of trainable parameters in the layer is $C_{\text{out}} \times K_1 \times K_2 \times C_{\text{in}} + C_{\text{out}}$, where the final $+C_{\text{out}}$ comes from the tiled bias matrices with one trainable parameter per channel.

Note that to truly perform convolution the kernel matrices $\mathbf{K}$ must be flipped along each axis before applying the expression above. However, deep-learning frameworks such as `tensorflow` implement convolution as given above since the kernels are full of trainable parameters, and thus the flipping is unnecessary.

Transposed convolutional layers are also frequently used and act as in a nearly opposite way to convolution. While convolutional layers reduce dimensions through a sliding dot product between the input and kernel(s), transposed convolution *increases* dimension by broadcasting the kernel at each input location and summing the intermediate results. The feature map change in dimension for transposed convolution is given by

$$D_j^{(i)} = \text{floor}\bigg(\Big[D_j^{(i-1)} - 1\Big] \, S_j^{(i)} - 2P_j^{(i)} + K_j^{(i)}\bigg) \tag{2.64}$$

for $j = 1, 2, 3$. Note the relation to Equation 2.61, where $D_j^{(i)}$ and $D_j^{(i-1)}$ are swapped and the expression rearranged.

Transposed convolution is named as such due to its relation with the matrix transpose. Note that convolution may be expressed as a matrix-vector product by transforming the kernel $\mathbf{K}$ into a multiply-blocked Toeplitz-like matrix $\mathbf{K}_t$ and multiplying with a vectorized (flattened) input, where the number of blocks depends on the filter size and original dimension of the convolution. Consider a 2D convolution as an example, with $\mathbf{K} \in \mathbb{R}^{3\times 3}$ and $\mathbf{H}_{conv} \in \mathbb{R}^{4\times 4}$ with a stride of 1 and no padding. Then $\mathbf{K}_t \in \mathbb{R}^{4\times 16}$ is the block-Toeplitz kernel matrix, $\mathbf{h}_{f,conv} \in \mathbb{R}^{16}$ is the flattened input, and the flattened convolution output $\mathbf{z}_{f,conv} \in \mathbb{R}^4$ is given as

$$\mathbf{z}_{f,conv} = \mathbf{K}_t \mathbf{h}_{f,conv}, \tag{2.65}$$

which is then reshaped into $\mathbf{Z}_{conv} \in \mathbb{R}^{2\times 2}$. Then, suppose transposed convolution with the same kernel matrix is desired, again with strides of 1 and no padding. Then $\mathbf{H}_{trans} \in \mathbb{R}^{2\times 2}$ and $\mathbf{h}_{f,trans} \in \mathbb{R}^4$ is the flattened input to transposed convolution. Then $\mathbf{z}_{f,trans} \in \mathbb{R}^{16}$ is the flattened output which is computed by

$$\mathbf{z}_{f,trans} = \mathbf{K}_t^T \mathbf{h}_{f,trans}, \tag{2.66}$$

which may then be reshaped into $\mathbf{Z}_{trans} \in \mathbb{R}^{4\times 4}$.

Many networks use a combination of dense and convolutional layers, and a class of models where the input is processed sequentially into the output are known as *feed-forward* neural networks. The network function may be written as a composition of all layers, and for a network with L hidden layers with input vector $\mathbf{x}$ this is given by

$$N(\mathbf{x};\theta) = f^{(L+1)} \circ f^{(L)} \circ f^{(L-1)} \cdots f^{(2)} \circ f^{(1)}(\mathbf{x}) \tag{2.67}$$

where the $L+1$th layer is the output layer, and intermediate layers may be a combination of dense and convolutional layers, along with layers which simply reshape the hidden state. Examples of this include convolutional autoencoders and decoder-CNN (DCNN) models developed in this thesis.

2.3.7 Graph Neural Networks

Graphs are a flexible data structure which store a collection of objects and their attributes or data as nodes or vertices, and information about their relationships to another through edges. Many datasets are modeled naturally as graphs, with canonical examples being social networks, citation networks, knowledge graphs, and recommender systems. As discussed previously, CNNs have driven state-of-the-art performance in computer vision, image classification, and pattern recognition type problems, and recently researchers have worked to extend core concepts from CNNs to general unstructured graphs, with network designs termed graph neural networks (GNNs) or graph convolutional networks (GCNs). In the context of modeling physical simulations, the computational mesh may be treated as a graph, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of vertices representing points in the computational domain, and $\mathcal{E}$ is the set of edges defining the connections among the nodes corresponding to mesh connectivity. This is represented using the weighted adjacency matrix $\mathbf{A} \in \mathbb{R}^{N\times N}$ which encodes the graph connections and weights between all nodes. If v_i and v_j are connected, then they share an edge (v_i, v_j) with strength w_{ij}, which becomes an entry in the weighted adjacency matrix as $A_{ij} = w_{ij}$. If two nodes v_i and v_j are unconnected, then $A_{ij} = 0$. A diagonal degree matrix $\mathbf{D} \in \mathbb{R}^{N\times N}$ may also be defined, where $\mathbf{D}_{ii} = \sum_{j=1}^{N} \mathbf{A}_{ij}$.

Graph neural networks may be classified as either spectral [85, 86, 87, 88] or spatial [89, 136, 137] approaches, although the two may be generalized by the message-passing graph neural network (MPGNN) [90]. Extending the ideas of CNNs to graphs is not straightforward since the discrete convolution and pooling operations are only defined on regular, Cartesian grids or domains. Spectral graph theory and the analysis of signals on graphs is an emerging field which seeks to extend ideas and mathematics from traditional signal processing to the graph domain [138]. The spectral formulation of GCNs utilize the fact that convolution in the spatial domain corresponds to multiplication in the Fourier domain, which may be restated that convolutions are linear, time invariant operators which diagonalize in the Fourier basis [139]. Additionally, there is a correspondence between the definition of the Fourier transform in the Euclidean domain and that in the graph-spectral domain, and this correspondence is used to define spectral graph convolutions.

To demonstrate this correspondence first define the unnormalized graph Laplacian $\mathbf{L} = \mathbf{D} - \mathbf{A} \in \mathbb{R}^{N \times N}$. The graph Laplacian may also be normalized as $\tilde{\mathbf{L}} = \mathbf{I}_N - \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$.Then the Cartesian-space Laplace operator $\mathcal{L}_c$ and the graph Laplacian are given as

$$\begin{array}{ccc} \mathcal{L}_c := \nabla \cdot \nabla & & \mathbf{L} := \mathbf{D} - \mathbf{A} \\ & \leftrightarrow & \\ \mathcal{L}_c(f(\mathbf{x})) = \nabla \cdot \nabla f(\mathbf{x}) & & \mathbf{L}(\mathbf{X}) = (\mathbf{D} - \mathbf{A})\mathbf{f}, \end{array} \tag{2.68}$$

where $\mathbf{f} \in \mathbb{R}^N$ is a column vector representing the function or signal on the graph. The usual Fourier transform may be defined as an inner product of the signal with the complex conjugate of the eigenfunctions of the 1D Laplace operator, $\mathrm{e}^{-i\omega t}$ [138], with the forward and inverse Fourier transforms defined as

$$\hat{f}(\omega) := \langle f, \mathrm{e}^{i\omega t} \rangle = \int_{-\infty}^{+\infty} f(t)\mathrm{e}^{-i\omega t} dt \tag{2.69}$$

$$f(t) := \frac{1}{2\pi} \int_{-\infty}^{+\infty} \hat{f}(\omega)\mathrm{e}^{i\omega t} d\omega. \tag{2.70}$$

where $\langle \cdot \rangle$ is an inner product in this context. $\mathbf{L}$ is a real, symmetric, positive-semidefinite matrix which has a full set of eigenvectors and a corresponding eigenvalue diagonalization $\mathbf{L} = \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^{-1} = \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^T$ [88]. $\boldsymbol{\Lambda} = \mathrm{diag}(\lambda_0, ..., \lambda_{N-1}) = \mathrm{diag}(\boldsymbol{\lambda})$, where $\boldsymbol{\lambda} \in \mathbb{R}^N$ holds the eigenvalues, and $\mathbf{U} \in \mathbb{R}^{N \times N}$, where the columns of $\mathbf{U}$ are orthonormal eigenvectors in this case.By analogy, one may then define the forward/inverse graph Fourier transform of any function f on the nodes of $\mathcal{G}$ as an expansion in terms of the eigenvectors of $\mathbf{L}$, with index i this is given component-wise and vectorized

$$\hat{f}(\lambda_\ell) := \langle \mathbf{f}, \mathbf{u}_\ell \rangle = \sum_{\imath=0}^{N-1} f(i) u_\ell^*(i) \rightarrow \hat{\mathbf{f}} := \mathbf{U}^T \mathbf{f} \tag{2.71}$$

$$f(i) = \sum_{\ell=0}^{N-1} \hat{f}(\lambda_\ell) u_\ell(i) \rightarrow \mathbf{f} = \mathbf{U}\hat{\mathbf{f}} \tag{2.72}$$

And finally, convolution of signal f and filter h in the temporal domain is multiplication in the spectral domain, as related by the Fourier transform

$$h(t) = (f * g)(t) \rightarrow \hat{h}(\omega) = \hat{f}(\omega)\hat{g}(\omega). \tag{2.73}$$

This may be used to define convolution in the graph spectral domain between signal $\mathbf{f}$ and filter $\mathbf{g}$. Start by proposing that convolution is multiplication in the spectral domain, $\hat{\mathbf{h}}(\boldsymbol{\lambda}) = (\hat{\mathbf{g}}(\boldsymbol{\lambda})\mathbf{I})\hat{\mathbf{f}}(\boldsymbol{\lambda})$, then substitute Equation 2.71 for $\hat{\mathbf{f}}$ and take the inverse graph Fourier transform by left-multiplying with $\mathbf{U}$, and replacing $\hat{\mathbf{G}} = \hat{\mathbf{g}}(\boldsymbol{\lambda})\mathbf{I}$ to yield the definition

$$(\mathbf{f} *_\mathcal{G} \mathbf{g}) := \mathbf{U}\hat{\mathbf{G}}\mathbf{U}^T\mathbf{f}. \tag{2.74}$$

Vector $\hat{\mathbf{g}} \in \mathbb{R}^N$ holds the diagonal entries of the filter matrix and is its spectral representation. The filter may be viewed as a re-scaling the graph Laplacian eigenvalues.

The first spectral GCNs parameterized the filter $\hat{\mathbf{g}}(\theta)$, analogous to how CNNs parameterize the convolutional kernels, and wrap the result with a non-linear activation function [85]. For a layer with C_{in} input features, and C_{out} output features, then $\mathbf{H}^{(i)} = \begin{bmatrix} \mathbf{h}_1^{(i)} & \dots & \mathbf{h}_{C_{in}}^{(i)} \end{bmatrix} \in \mathbb{R}^{N \times C_{in}}$ and $\mathbf{H}^{(i+1)} = \begin{bmatrix} \mathbf{h}_1^{(i+1)} & \dots & \mathbf{h}_{C_{out}}^{(i+1)} \end{bmatrix} \in \mathbb{R}^{N \times C_{out}}$ are the ith and $(i+1)$th hidden states, with a forward propagation rule given by

$$\mathbf{h}_\ell^{(i+1)} = \sigma.\left(\textstyle\sum_{\ell'=1}^{C_{out}} \mathbf{U}_k \mathbf{G}(\theta)_{\ell,\ell'} \mathbf{U}_k^T \mathbf{h}_{\ell'}^{(i)} \right). \tag{2.75}$$

A truncated basis is frequently used, $\mathbf{U}_k \in \mathbb{R}^{N \times k}$, implying $\mathbf{G} \in \mathbb{R}^{k \times k}$ with k trainable parameters on the diagonal.

Other GCNs seek an approximation as even computing the eigendecomposition for large graphs may be prohibitively expensive, with each defined by the manner of approximation [86, 140, 88]. Despite operating on non-Euclidean data, spectral GCNs are limited to problems defined by the same graph, since the filter weights learned are with respect to the eigenbasis $\mathbf{U}$. Thus spectral GCNs are not dot discretization independent when applied to HFM solutions.

While spectral GCNs define graph convolution by analogy with the Euclidean Fourier transform and convolution, others take a more spatially-inspired approach, where nodes in the graph interact within a predefined neighborhood for each node v, $\mathcal{N}(v)$. This includes message-passing graph neural networks (MPGNN), which in fact even generalize the spectral GCN approaches given above [90]. In addition to node-defined quantities used in the spectral approaches, MPGNN also allow for edge features $\mathbf{e}_{vw}$ between nodes v and w. A forward-pass consists of message-passing and (optional) readout stages. The message-passing phase is iterative and depends upon message function M_t and vertex update function U_t as

$$\mathbf{m}_v^{t+1} = \sum_{w \in \mathcal{N}(v)} M_t(\mathbf{h}_v^t, \mathbf{h}_w^t, \mathbf{e}_{vw}) \tag{2.76}$$

$$\mathbf{h}_v^{t+1} = U_t(\mathbf{h}_v^t, \mathbf{m}_v^{t+1}). \tag{2.77}$$

Additional edge-feature hidden states may also be defined and updated analogously. The readout stage is used when a graph-level embedding is needed, $\hat{\mathbf{y}} \in \mathbb{R}^M$ for some scalar M, and may be written in terms of readout function R as

$$\hat{\mathbf{y}} = R\left(\Big[\mathbf{h}_v | v \in \mathcal{G}\Big].\right) \tag{2.78}$$

Functions M_t, U_t and R are all learned parametric, differentiable functions. GraphSAGE was proposed around the same time as MPGNNs and includes similar trainable aggregator functions [141] which fall under the generalized approach, but with an emphasis on allowing predictions for unseen nodes or subgraphs. MPGNNs have been used to emulate PDE solutions in both mesh-based [92, 93] and mesh-free scenarios [142, 94]. Within the context of modeling HFM solutions, whether or not a MPGNN can be applied in a discretization independent manner depends upon the details of the message passing and update functions.

2.3.8 Point cloud neural networks

Point cloud neural networks are useful in situations in which the data is available in the form of unstructured point clouds, such as the raw output of a LIDAR unit or other three-dimensional sensor. As such they are often seen in the context of autonomous vehicles or robotic vision. PointNet[143] and PointNet++[144] are architectures designed for point clouds and are used for scene recognition, classification, and segmentation tasks. The networks consume a point cloud corresponding to a 3D scan or mesh, and either offer an overall classification score for the scene, or a point-by-point segmentation score, where the goal is scene analysis. The network produces a global feature vector upon processing a point cloud, which is used in turn by a global classifier network, or a segmentation network. The segmentation network provides a pointwise score, and it is possible that a network with such an architecture may be used in a predictive setting instead. This is demonstrated with PointNet++ used to predict viscous, incompressible flows over 2D shapes lying on unstructured meshes [145].

2.3.9 Operator regression methods

Another class of relevant techniques capable of handling unstructured data in some instances includes operator-regression methods, such as those based on DeepONet [95, 96, 97], Neural Operator [98, 99], Fourier basis networks[100], and GMLS Nets [101]. These methods have also shown impressive results, and generally seek mappings between terms appearing *explicitly* in the governing equations and the state. Deep Operator Networks (DeepONet) are designed based on an operator representation theorem and can query any point in the output domain [95], and may be interpreted as a partial-hypernetwork model where the weights of only the output layer are generated via the branch network. DeepONet has been demonstrated for Darcy flow on complex domains, mapping from boundary conditions to full-domain pressure fields, but separate networks are trained for each different domain, rather than using a single network on all domains [146]. Additionally most Neural Operator (NO) methods [147] are capable of handling unstructured or varying meshes, excluding Fourier-NO (FNO) [100] which require inputs and outputs to lie on a Cartesian grid. Geo-FNO was developed recently to extend FNO to such problems [148], and FNO was extended previously by expanding/shrinking the input/output domains and interpolating to a Cartesian mesh and extrapolating the solution to a bounding box as required [146].

2.3.10 Solving PDEs with neural networks

Some techniques blur the line between *solving* PDEs and regressing approximate PDE solutions from data. Physics-informed neural networks [149, 150] (PINNs) are an example of this. PINNs may be seen as a modern extension of methods to solve ODEs/PDEs without data using neural networks, introduced in the late 90's [151, 152]. The general idea is to embed the governing ODE/PDE in the loss function, and to compute the required derivatives directly from the neural-network prediction. A system of ordinary or partial differential equations may be written generically as

$$\begin{aligned}\mathcal{R}(\mathbf{q}(\mathbf{x},t;\boldsymbol{\mu}),\ldots) &= 0, \quad \mathbf{x}\in\Omega && (2.79)\\ \mathcal{B}(\mathbf{q}(\mathbf{x},t;\boldsymbol{\mu}),\ldots) &= 0, \quad \mathbf{x}\in\partial\Omega && (2.80)\end{aligned}$$

where vector $\mathbf{q}(\mathbf{x},t)$ is the unknown system state, $\mathcal{R}$ is a PDE operator, $\mathcal{B}$ is a boundary condition operator, and $\boldsymbol{\mu}$ are the problem parameters. In the following discussion, limit to steady-state problems with 1-dimensional states for simplicity. Lagaris et. al introduced such a method for solving initial and boundary value problems on rectangular domains [151] and later introduced a modification allowing for irregular domains [152].

The method constructs a trial solution $\hat{q}$ consisting of two terms written as

$$\hat{q}(\mathbf{x};\theta) = A(\mathbf{x}) + F\big(\mathbf{x}, N(\mathbf{x};\theta)\big). \tag{2.81}$$

The first term $A(\mathbf{x})$ is constructed to satisfy the boundary conditions without trainable parameters, while the second term F uses a neural network $N(\mathbf{x};\theta)$ operating on the coordinates and is constructed such that it does not contribute on the boundary. The neural network weights and biases are represented by the set θ. To solve the differential equation, the approximation $\hat{q}$ is plugged into Equation 2.79 and used to compute a loss function, with an example total squared loss written as

$$\mathcal{L}(\theta) = \sum_i \mathcal{R}\big(\hat{q}(\mathbf{x}_i;\theta)\big)^2, \quad \mathbf{x}_i \in \Omega. \tag{2.82}$$

The solution is then obtained by solving an optimization problem for the network weights, written as

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}}\, \mathcal{L}(\theta). \tag{2.83}$$

This procedure was shown to work for simple ODEs/PDEs, but the method for constructing $A(\mathbf{x})$ to satisfy boundary conditions is a laborious process for higher dimensional states and problem domains more complex than simple rectangles. Additionally, their work predates the wide availability of automatic-differentiation tools so gradients with respect to the model parameters θ were computed explicitly, also a time intensive and difficult task. The modification presented in the follow on work of reference [152] for irregular boundaries uses a combination of feed forward network for the solution domain and radial basis function network to satisfy the the boundary conditions. This idea has seen a resurgence in interest with the introduction of Physics Informed Neural Networks (PINNs), which provide a slightly different approach to solving differential equations with or without the use of data [149]. In PINNs, the approximation of the unknown function $\hat{q}$ is given simply as a feed-forward neural network, without term $A(\mathbf{x})$. Violation of the boundary conditions is then weakly penalized during training, and additional data terms may also be included, with domain loss, boundary loss, data loss, and overall loss written as

$$\mathcal{L}_{\Omega}(\theta) = \frac{1}{n_{\Omega}} \sum_{i=1}^{n_{\Omega}} \mathcal{R}\big(\hat{q}(\mathbf{x}_i, \theta)\big)^2, \quad \mathbf{x}_i \in \Omega \tag{2.84}$$

$$\mathcal{L}_{\partial\Omega}(\theta) = \frac{1}{n_{\partial\Omega}} \sum_{i=1}^{n_{\partial\Omega}} \mathcal{B}\big(\hat{q}(\mathbf{x}_i, \theta)\big)^2, \quad \mathbf{x}_i \in \partial\Omega \tag{2.85}$$

$$\mathcal{L}_{d}(\theta) = \frac{1}{n_{d}} \sum_{i=1}^{n_{d}} \big(\hat{q}(\mathbf{x}_i, \theta) - q(\mathbf{x}_i)\big)^2 \tag{2.86}$$

$$\mathcal{L}(\theta) = \lambda_{\Omega}\mathcal{L}_{\Omega}(\theta) + \lambda_{\partial\Omega}\mathcal{L}_{\partial\Omega}(\theta) + \lambda_{d}\mathcal{L}_{d}(\theta). \tag{2.87}$$

Coefficients λ may be used to assign relative importance or contribution to the total loss. Solution of the differential equation is as before in Equation 2.83 and solved via unconstrained optimization. Computing derivative terms is facilitated by modern automatic-differentiation-based deep learning packages.

Chapter 3

Predictive Deep-Learning Models Without Interpolation of Ground-Truth Data

Many advances in machine learning have been driven by methods which approximate mappings involving high-dimensional spaces, with the cardinality of input or output spaces $\sim \mathcal{O}(10^2 - 10^5)$ [153, 154, 79, 133]. This includes matrix-decomposition-based methods [63, 155] and deep-learning autoencoder techniques [76, 80, 78] used in scientific applications. Computational limitations arise when applied to scientific or engineering simulations of scale, where the mesh-cell cardinality N may be in the tens-of-millions or even billions in super-computing settings [156], with the total degrees-of-freedom for a solution state an even larger multiple of N. This corresponds to mapping between high-dimensional snapshots in autoencoder-style models, and decomposing even larger snapshot matrices for projection-based ROMs, dynamic mode decomposition, or related Koopman methods [157]; scenarios which become easily limited by computational cost. Further, when several solutions at different conditions are considered, then the mesh topology and N may also vary, disallowing use of snapshot or decomposition based methods.

A recent line of research instead approximates infinite-dimensional (continuous) functions by mapping between lower-dimensional spaces using simple coordinate-based fully-connected multi-layer-perceptron (MLP) neural networks. That is, instead of mapping between snapshots over the full domain of the problem, for example with a non-autoassociative autoencoder convolutional neural network (CNN) [76], a mapping is regressed between inputs and outputs at each individual point in space. The resulting key distinction is that coordinate-inputs $\mathbf{x}$ are taken *pointwise*, for example as a single physical-coordinate-tuple (x, y, z), instead of entire-solution snapshots. A driving application is object and scene representation for rendering in computer graphics by which objects are represented by continuous implicit fields such as the signed-distance-function (SDF) zero-level-set [158, 159], SDF decision-boundary [160], or as a density/differential-opacity along a light ray [161] [1]. This concept is known by several names, including *coordinate-based networks*, *neural fields*, *neural implicits*, and *implicit neural representations* [162], the latter of which provides a framework which encompasses and generalizes the methods, extending them to other problem scenarios including recovery of typical supervised learning problems. Using a continuous representation is key in attaining discretization independence when applied to scientific simulation data. Every point in each mesh may be included separately, eliminating the need for lossy interpolation of solution data onto a common Cartesian mesh, as was needed in Section 2.1 for snapshot-based methods.

In this work coordinate-based MLPs are applied to the prediction of partial differential equation (PDE) solution fields defined on domains with complex and variable geometry. The predictive models must be able to concurrently handle unstructured data and variation in physical design and operating conditions, as described by design-variable vector $\boldsymbol{\mu}$, to provide the greatest utility in design optimization or other engineering tasks. The physical-coordinate inputs are augmented to include an evaluation of the signed-distance or minimum-distance function (MDF) to provide global information about the domain at each point; this may be viewed as a form of concatenation-based *local* conditioning as the SDF/MDF are functions of space over the relevant domains. This input is referred to as the augmented physical coordinates, and for a 3D problem is given as

$$\mathbf{x}' = \begin{bmatrix} x & y & z & \phi(x, y, z) \end{bmatrix}^T . \tag{3.1}$$

For the problems considered, the MDF and SDF are equivalent, as all mesh locations

[1]See Equation 3.5 for SDF definition.

are external to the relevant shapes. In addition to the MDF, the network predictions are *globally* conditioned upon the design variables $\boldsymbol{\mu}$. The distinction between global and local conditioning is that global-conditioning vectors do not vary with space and apply to all mesh points, while local conditioning vectors are spatial functions. Several different techniques for conditioning neural network predictions are used in the literature [163, 158, 164, 162] with concatenation-based conditioning and the use of hypernetworks [165] explored here. Most conditioning schemes involve learning an additional embedding to condition the networks upon, whereas here the design variables and SDF are used instead; both of which are known once a design is selected, simplifying the overall scheme.

The developed methods begin by first considering problems defined on coordinate-transformed meshes, where the computational domain has a regular, Cartesian structure. Common examples include body or domain-fitted meshes, as well as C-meshes and O-meshes commonly used in airfoil aerodynamics. The regular structure allows for convolutional architectures, variants of which drove many recent advances in deep learning, to be applied directly in the computational domain, eliminating the need for interpolation. This is significant, as none of the negative effects associated with interpolation of the ground-truth data are present. Developed methods include decoder-CNN (DCNN) based full-field surrogates which are covered in Section 3.1. These methods address the desire to handle the ground truth data without interpolation, but are not suited to handle variable mesh topologies as they are locked to and defined for the Cartesian computational meshes for a given problem.

Next, background relating to coordinate-based MLPs as described above are given in greater detail by examining implicit neural representations. Three methods towards generalization via conditioning are considered and compared. The first utilizes concatenation-based conditioning, where the design-variable-embedding-vector $\boldsymbol{\mu}$ is concatenated with the augmented physical coordinates, similar to autodecoder networks [158] but without concurrent learning of the embedding, referred to as design-variable MLP (DV-MLP). The second utilizes weight-embedding, where the weights of a main network are generated using a hypernetwork consuming the design variables, known as deign-variable hypernetworks (DVH). The third is a special case of DVH, called non-linear independent dual system (NIDS), where only weights and biases of the final layer are generated using a hypernetwork. NIDS predictions may be written in a format analogous to POD reconstruction, and further shares strong similarity with recently developed DeepONet methods.

3.1 Decoder Convolutional Neural Networks (DCNN)

As discussed in Section 2.3.5, autoencoders have a bottleneck structure and may be used for a variety of supervised and unsupervised learning or representation tasks. Non-autoassociative autoencoders have been used in the context of surrogate modeling. For example, an autoencoder CNN was used to predict turbulent airfoil flows using the signed distance field as input [76]: see Figure 3.1(a) for a network schematic. Global parameters $\boldsymbol{\mu} = \left[\text{Re AoA}\right]^T$ are concatenated with the encoder output and fed-forward through the decoder to generate an approximate solution, as shown in Equations 3.2-3.4.

$$\Phi(\mathbf{x}) = \tilde{\mathbf{z}} \tag{3.2}$$

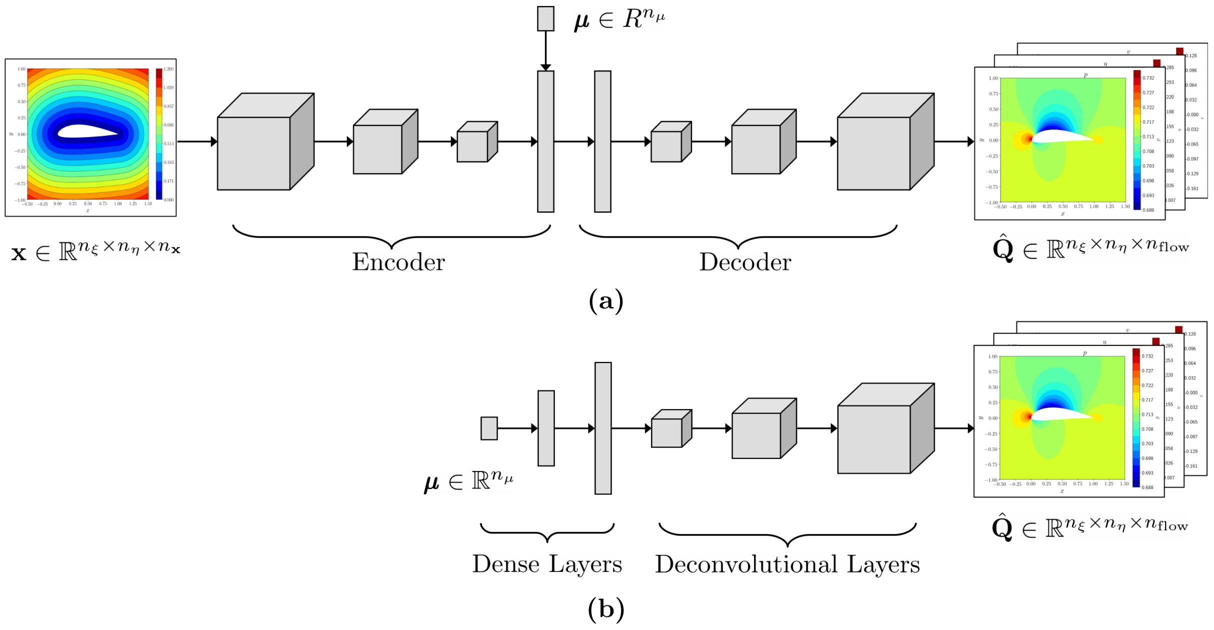
$$\mathbf{z} = \left[\boldsymbol{\mu}^T \;\; \tilde{\mathbf{z}}^T\right]^T \tag{3.3}$$

$$\Theta(\mathbf{z}) = \hat{\mathbf{Q}} \approx \mathbf{Q} \tag{3.4}$$

In other existing methods, an approximate solution for a new set of conditions or for a new time-step may be approximated using only the trained decoder and some form of mapping or interpolation in the latent space [80].

Here it is proposed that, for problems of interest in engineering design optimization, the design variables $\boldsymbol{\mu}$ may be used in lieu of a learned latent representation $\mathbf{z}$, given that the design variables define a unique design instance, assuming the problem definition is deterministic and one-to-one. This corresponds to eliminating the encoder and replacing the learned latent representation with the design variables, and this defines DCNN models.

The difference between autoencoder CNNs and DCNNs is visualized in Figure 3.1.



$\boldsymbol{\mu} \in R^{n_\mu}$
$\mathbf{x} \in \mathbb{R}^{n_\xi \times n_\eta \times n_{\mathbf{x}}}$
Encoder
Decoder
$\hat{\mathbf{Q}} \in \mathbb{R}^{n_\xi \times n_\eta \times n_{\text{flow}}}$
(a)
$\boldsymbol{\mu} \in \mathbb{R}^{n_\mu}$
Dense Layers
Deconvolutional Layers
$\hat{\mathbf{Q}} \in \mathbb{R}^{n_\xi \times n_\eta \times n_{\text{flow}}}$
(b)

Figure 3.1: Schematic comparison of (a) autoencoder CNN with latent-space injection, model as presented in [76], and (b) DCNN, with sequence of dense and transposed convolution layers.

Although the DCNN schematic shows a sequence of dense layers, numerical experiments showed the best results, in terms of mean squared error, were achieved with a single dense layer between the input and first decoder layer. Thus only a single dense layer is used for all DCNN predictions shown in later sections.

In Chapter 5, compressor airfoil RANS solutions are defined on multi-block meshes, where each zone has a Cartesian structure. In that scenario DCNN models may be applied by defining parallel decoder legs for each mesh zone, and in principle this method may be applied to multi-block meshes with an arbitrary number of zones, provided each block is Cartesian in structure. A notional schematic for a multi-block DCNN model for a subsonic compressor airfoil is shown in Figure 3.2.

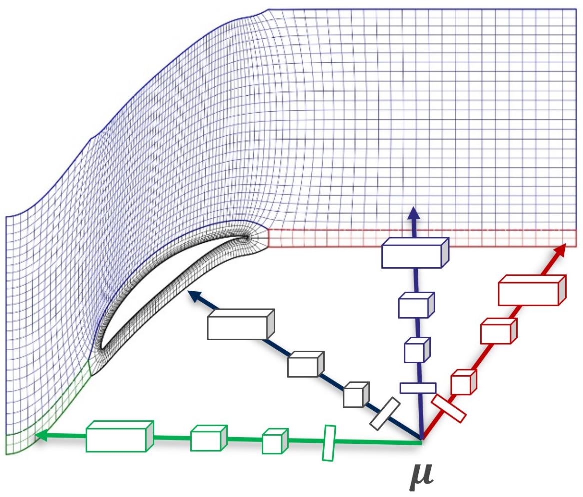


Figure 3.2: Notional schematic of a multi-block DCNN model with parallel decoder legs for each mesh zone.

Designing DCNN models, or convolutional autoencoder models in general, for computational meshes often involves unforeseen difficulties due to unusual or non-typical mesh dimensions. The difficulty arises from needing to account for non-integer values and the floor functions of Equations 2.61 and 2.64.

Beyond the present discussion, encoding and decoding are fundamental concepts and operations in machine learning. Convolutional classifiers correspond architecturally to the encoder of Figure 3.1, where a vector of class probabilities replaces the produced latent representation. Generative adversarial networks (GANs), a popular generative modeling method, may utilize convolutional decoders in image synthesis, analogous in architecture to DCNN models, although with a very different training scheme. These use cases correspond to more probabilistic scenarios and as a result use different training losses and training schemes, such as binary-cross-entropy loss for classification, and paired training of generator and discriminator networks in GANs via minimax game [166].

3.2 Discretization-Independent Methods

Prior to introducing the devised techniques, additional background is presented regarding deep-learning methods which directly influenced their development. This includes coordinate-based networks for scene representation which is generalized by implicit neural representations, along with concepts for conditioning network predictions which may be generalized by hypernetworks. The proposed methods draw directly from these to achieve full discretization independence.

3.2.1 Shape and Scene Representation via Coordinate-based Neural Networks

Given a set of points representing a surface $\mathcal{S} = \partial\mathcal{V}$ of an object $\mathcal{V} \in \mathbb{R}^m$ in m-dimensional physical space, the signed-distance function may be defined as

$$f_{\mathrm{sdf}}(\mathbf{x}; \mathcal{S}) \triangleq \begin{cases} \phi(\mathbf{x}, \mathcal{S}) & \mathbf{x} \notin \mathcal{V} \\ 0 & \mathbf{x} \in \mathcal{S} \\ -\phi(\mathbf{x}, \mathcal{S}) & \mathbf{x} \in \mathcal{V} \end{cases}, \tag{3.5}$$

where

$$\phi(\mathbf{x}, \mathcal{S}) \triangleq \inf_{\mathbf{y} \in \mathcal{S}} d(\mathbf{x}, \mathbf{y}) \tag{3.6}$$

is a minimum-distance function (MDF), and $d(\cdot, \cdot)$ is the Euclidean-distance function. Stated simply, the SDF is the minimum distance between the field point $\mathbf{x}$ and the surface $\mathcal{S}$ in consideration. It takes positive values for points outside the object ($\mathbf{x} \notin \mathcal{V}$), negative values for points inside ($\mathbf{x} \in \mathcal{V}$), and is identically zero on the surface.

Coordinate-based MLPs have been used to represent 3D objects for rendering tasks. The object's surface is implicitly represented within a volumetric field; including as the zero-level-set of a directly-regressed signed-distance field [158, 159] or decision boundary (interior/exterior) [167, 160], or as an emitted radiance and density/differential-opacity field [161]. Many of these methods include loss terms describing a rendering process, such that the entire image generation process contributes to the loss during training. This concept may be generalized by *implicit neural representations*, introduced along with sin-activation SIREN networks in by [168], from which defining Equations 3.7 and 3.8 are taken. In this setting, a function of interest Φ with input coordinates $\mathbf{x}$, written

$\Phi : \mathbf{x} \to \Phi(\mathbf{x})$, is defined by a set of m constraints $\mathcal{C}$,

$$\mathcal{C}_m(\mathbf{x}, \mathbf{a}(\mathbf{x}), \Phi, \nabla_{\mathbf{x}}\Phi, \nabla^2_{\mathbf{x}}\Phi, \dots) = 0,\ \mathbf{x} \in \Omega_m,\ m = 1, \dots, M \tag{3.7}$$

which optionally depend on the function values Φ, function gradients $\nabla_{\mathbf{x}}\Phi$, and additional quantities $\mathbf{a}(\mathbf{x})$ which are needed to compute the constraints. When a neural network N with parameters θ is used to approximate Φ, then this is referred to as an *implicit neural representation*. To train the neural-network approximation, a loss function with M terms is defined by penalizing deviation from the constraints,

$$\mathcal{J}(\theta) = \int_{\Omega} \sum_{m=1}^{M} \mathbb{1}_{\Omega_m}(\mathbf{x}) \parallel \mathcal{C}_m(\theta, \mathbf{x}, \mathbf{a}(\mathbf{x}),\ \dots) \| \, d\mathbf{x}, \tag{3.8}$$

where the indicator function $\mathbb{1}_{\Omega_m}$ activates over valid locations within the domain Ω_m. A key distinction between this and other methods is that coordinate-inputs $\mathbf{x}$ are taken *pointwise*, for example as a single physical-coordinate-tuple (x, y, z), instead of as entire-solution snapshots. Surprisingly-many problems may be cast in this form, including as-discussed surface representation and variations on classic deep-learning problems, such as classifying MNIST hand-written digits. The approaches proposed in this work may be viewed through this lens where the only constraint is that the predictions match the data, recovering basic supervised regression. However, this does not directly align with the main thrust of implicit neural representations, where an implicit function is regressed and additional constraints are imposed.

3.2.2 Conditioning Neural Networks

It is commonplace in machine learning to process an input in the context of another secondary input in a process known as conditioning. Several conditioning techniques will be introduced briefly, with the presentation influenced by and similar to that given in ref. [164]. Let $\mathbf{x} \in \mathbb{R}^{n_x}$ be the primary input and $\boldsymbol{\mu} \in \mathbb{R}^{n_\mu}$ be the secondary or conditional input. For all examples consider fully-connected MLP networks with a hidden dimension H, although the concepts may be readily applied to other network types.

Concatenation-based conditioning is a self descriptive term; the primary and conditional inputs are concatenated before being passed to the next layer. For example, if applied to network inputs (the zeroth hidden state), then $\mathbf{h}^{(0)}$ is given simply as

$$\mathbf{h}^{(0)} = \begin{bmatrix} \mathbf{x} \\ \boldsymbol{\mu} \end{bmatrix} \in \mathbb{R}^{n_x + n_\mu}. \tag{3.9}$$

The inclusion of the conditional input has the effect of adding $H \times n_\mu$ weights to the layer.

Another approach is known as conditional biasing, where a bias vector is generated from the conditional input, $f_{\text{bias}} : \boldsymbol{\mu} \to \mathbf{b}(\boldsymbol{\mu}) \in \mathbb{R}^H$, and added to a layers input. For example, consider conditional biasing of the first hidden state, written as

$$\mathbf{h}^{(1b)} = \mathbf{h}^{(1)}(\mathbf{x}) + \mathbf{b}(\boldsymbol{\mu}), \tag{3.10}$$

where the second layer now takes $\mathbf{h}^{(1b)}$ as its input. Conditional biasing is equivalent to concatenation-based conditioning when f_{bias} is linear and bias-less linear, dense layers are considered. To see this, consider $\mathbf{h}^{(0)}$ as given in Equation 3.9 for concatenation-based conditioning, and let $n_x = n_\mu = 2$, $H = 5$. Then the first-layer weight matrix may be written as

$$\mathbf{W}^{(1)} = \begin{bmatrix} | & | & | & | \\ \mathbf{w}_1 & \mathbf{w}_2 & \mathbf{w}_3 & \mathbf{w}_4 \\ | & | & | & | \end{bmatrix} \in \mathbb{R}^{5\times 4}, \tag{3.11}$$

where $\mathbf{w}_i \in \mathbb{R}^5$. Then the first layer may be written as

$$f^{(1)}_{\text{dense}} = \mathbf{W}^{(1)}\mathbf{h}^{(0)} = \begin{bmatrix} | & | & | & | \\ \mathbf{w}_1 & \mathbf{w}_2 & \mathbf{w}_3 & \mathbf{w}_4 \\ | & | & | & | \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \boldsymbol{\mu} \end{bmatrix} = x_1\mathbf{w}_1 + x_2\mathbf{w}_2 + \mu_1\mathbf{w}_3 + \mu_2\mathbf{w}_4 \tag{3.12}$$

$$= \mathbf{W}^{(1)}_{1,2}\mathbf{x} + \mathbf{W}^{(1)}_{3,4}\boldsymbol{\mu} \tag{3.13}$$

$$= \mathbf{h}^{(1)}(\mathbf{x}) + \mathbf{b}(\boldsymbol{\mu}) \tag{3.14}$$

where $\mathbf{W}^{(1)}_{1,2} = \begin{bmatrix}\mathbf{w}_1 & \mathbf{w}_2\end{bmatrix} \in \mathbb{R}^{5\times 2}$ and $\mathbf{W}^{(1)}_{3,4} = \begin{bmatrix}\mathbf{w}_3 & \mathbf{w}_4\end{bmatrix} \in \mathbb{R}^{5\times 2}$. Note the equivalence of Equations 3.14 and 3.10, with $f_{\text{bias}} = \mathbf{W}^{(1)}_{3,4}\boldsymbol{\mu}$.

Similar to conditional biasing is conditional scaling, where an elements-wise product is computed instead of addition. The conditional scaling vector is given by $f_{\text{scale}} : \boldsymbol{\mu} \rightarrow \mathbf{s}(\boldsymbol{\mu}) \in \mathbb{R}^H$. Then, as an example, the output of the first layer is modified according to

$$\mathbf{h}^{(1s)} = \mathbf{h}^{(1)} \odot \mathbf{s}(\boldsymbol{\mu}), \tag{3.15}$$

where $\odot$ is the Hadamard or element-wise product.

Feature-wise Linear Modulation (FiLM) layers [163], used in feature-wise transformations [164], include both conditional biasing and scaling terms, with a general expression given as

$$\text{FiLM}(\mathbf{x}) = \boldsymbol{\gamma}(\boldsymbol{\mu}) \odot \mathbf{x} + \boldsymbol{\beta}(\boldsymbol{\mu}). \tag{3.16}$$

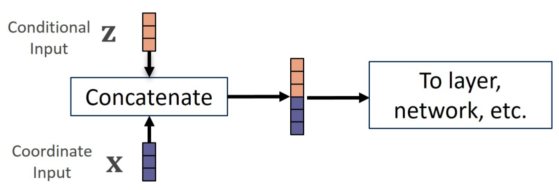
The vectors $\boldsymbol{\gamma}(\boldsymbol{\mu})$ and $\boldsymbol{\beta}(\boldsymbol{\mu})$ come from a FiLM generator, which is a separate function or neural network. The FiLM generator may be written as $f_{\text{FiLM}} : \boldsymbol{\mu} \rightarrow \boldsymbol{\gamma}(\boldsymbol{\mu}), \boldsymbol{\beta}(\boldsymbol{\mu}) \in \mathbb{R}^H$. A given main network may have FiLM layers or other conditional operations inserted between each existing hidden layer or blocks of layers. Each FiLM layer may have a separate FiLM generator network, or all required FiLM vectors may be produced from a single generator, with the same comments applying to conditional biasing and scaling.

3.2.2.1 Hypernetworks and Generalized Conditioning

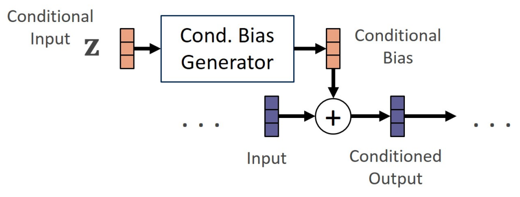
Hypernetworks constitute a metamodeling approach where one neural network is used to generate the weights of another main network [165], and are part of a broader class of proposed techniques where network weights are conditioned on model inputs or features [169, 170, 171]. Additionally, hypernetworks generalize all of the conditioning methods presented above by considering a hypernetwork which generates only a portion of the main-network weights and/or biases. This is easy to understand for conditional biasing, as f_{bias} may be interpreted as a hypernetwork which generates only the bias vectors. Concatenation-based conditioning is generalized with the caveat that the bias-only hypernetwork should use linear activations, but a non-linear hypernetwork may be more expressive. Conditional scaling is a linear map and as such may be expressed as a matrix-vector multiplication instead, where the matrix is the weight-matrix output by the hypernetwork. That is, given a scaling vector $\boldsymbol{\gamma}(\boldsymbol{\mu})$, then

$$\boldsymbol{\gamma}(\boldsymbol{\mu}) \odot \mathbf{x} = \mathbf{W}(\boldsymbol{\mu})\mathbf{x} \tag{3.17}$$

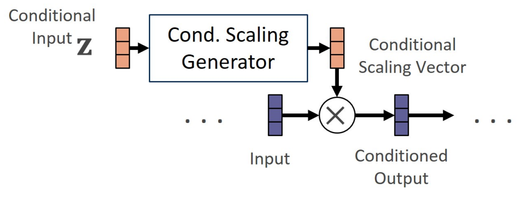
for some $\mathbf{W}(\boldsymbol{\mu})$ which may be viewed as expansion in an appropriate, learned basis. FiLM layers are comprised of conditional scaling and biasing so it follows immediately that they are generalized by hypernetworks. However, there are differences regarding the number of trainable parameters in a FiLM generator and equivalent hypernetwork, generally with the hypernetwork containing more parameters. Consider a main-network layer which has inputs and outputs of dimension H. Then the FiLM-generator-output-space dimension is $2H$, while for a hypernetwork it is $H^2 + H$. If a single-layer, biasless, dense FiLM generator or hypernetwork is used, then the number of trainable parameters in each is $2Hn_\mu$ and $(H^2+H)n_\mu$ respectively. Schematic representations of the methods are shown in Figure 3.3.



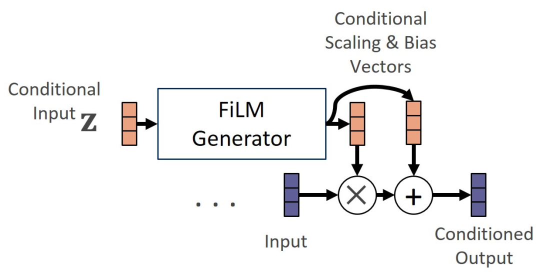
(a) Concatenation-based conditioning



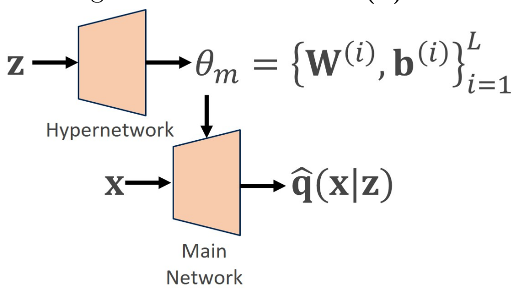
(b) Conditional biasing



(c) Conditional scaling



(d) Feature-wise linear modulation


z
$\theta_m = \{\mathbf{W}^{(i)}, \mathbf{b}^{(i)}\}_{i=1}^{L}$
Hypernetwork
x
$\hat{\mathbf{q}}(\mathbf{x}|\mathbf{z})$
Main Network

(e) Hypernetwork

Figure 3.3: Schematic representations of the discussed methods for conditioning neural fields. Conditional scaling and FiLM use pointwise multiplication.

Hypernetworks were originally applied to convolutional and recurrent neural networks for image- and natural-language-processing tasks, with the goal of reducing the number of trainable parameters while maintaining or improving model accuracy. In such models, the weights of the main network are generated on a layer-by-layer basis, where the hypernetwork consumes a layer-embedding vector and outputs the weights for that layer. The use-cases for hypernetworks have largely been the domain of computer science, but lately have been applied to scientific machine learning in some instances. Pan et al. [172] leveraged hypernetworks to learn latent representation from turbulence on arbitrary meshes in a scheme similar to design-variable hypernetworks developed here. HyperPINNs applies hypernetworks to Physics Informed Neural Networks (PINNs) [149] for parametric PDE solutions of 1D-viscous Burgers and the Lorenz system, with improved accuracy seen over baseline PINN models despite a smaller main network [173].

3.2.3 Problem Setup

Denote the solution snapshot for a single instance j of the FOM as

$$\mathcal{D}_j \triangleq \left\{ \left\{ \mathbf{q}_i \mid \mathbf{x}_i \right\}_{i=1}^{n_j}, \ \boldsymbol{\mu}_j \right\}, \tag{3.18}$$

where the solution output-input pairs are defined at n_j spatial (mesh) locations. Considering a dataset

$$D \triangleq \left\{ \mathcal{D}_1, \ \mathcal{D}_2, \ \dots, \mathcal{D}_{n_D} \right\} \tag{3.19}$$

containing n_D snapshots, a distinctive feature of this approach is that each snapshot may correspond to a solution domain with different spatial extent and discretization, with varying number and location of mesh points. Models are sought which can approximate the solution snapshots stored in D, without interpolation of ground-truth data or prediction. In other words, given the generative factors or design variables for a problem $\boldsymbol{\mu} \in \mathcal{M} \subset \mathbb{R}^{n_\mu}$, predict the system state $\mathbf{q}$ at any location $\mathbf{x} \in \Omega(\boldsymbol{\mu})$. Denote the input space as $\mathbf{x} \in \mathcal{X} \subset \mathbb{R}^{n_x}$, and the output space as $\mathbf{q} \in \mathcal{Q} \subset \mathbb{R}^{n_q}$.

The desired model should generalize across solution instances, and thus approximate the mapping $f : \mathcal{M} \times \mathcal{X} \to \mathcal{Q}$, without direct knowledge of the system state. Note that this is in contrast to initial-value problems, where the initial state to be integrated forward in time is necessary. The problems considered may contain complex and variable geometry, the input space is augmented to include an additional minimum-distance-function coordinate, defined as

$$\mathbf{x}' \triangleq \left[\mathbf{x}^T, \quad \phi(\mathbf{x}; \boldsymbol{\mu}) \right]^T. \tag{3.20}$$

In the scenarios considered all mesh-distances are positive, so use of MDF and SDF are equivalent. This defines an augmented input space, $\mathcal{X}' \subset \mathbb{R}^{n_x+1}$, where $\mathbf{x}' \in \mathcal{X}'$, and in turn the desired mapping is

$$f : \mathcal{M} \times \mathcal{X}' \to \mathcal{Q}. \tag{3.21}$$

The model approximation is then written as $f(\mathbf{x}';\boldsymbol{\mu}) \approx \hat{\mathbf{q}}(\mathbf{x}';\boldsymbol{\mu})$ or just $\hat{\mathbf{q}}$ in compact notation. Each design variable $\boldsymbol{\mu}$ defines a spatial solution field over a domain, thus it is natural to condition the models upon the design variables.

3.2.4 Method 1: Design-variable MLP (DV-MLP)

DV-MLP uses concatenation-based conditioning to account for different solution instances, where the augmented coordinates $\mathbf{x}'$ are concatenated with the design variables $\boldsymbol{\mu}$. This is a from of global conditioning as the same $\boldsymbol{\mu}$ is used for all spatial locations for a given solution field. The main network (denoted as N_m with weights θ_m) and resulting prediction are written as

$$\hat{\mathbf{q}}(\mathbf{x}';\boldsymbol{\mu}) = N_m(\mathbf{x}',\boldsymbol{\mu};\theta_m). \tag{3.22}$$

Simple fully-connected layers are used, where the hidden state of each layer has the same dimension or number of nodes, and no skip or recurrent connections are used.

3.2.5 Method 2: Design-Variable Hypernetworks (DVH)

DVH generates the weights and biases of the main network θ_m using a one-shot hypernetwork which takes as input the design variables $\boldsymbol{\mu}$. Ref. [159] noted an autodecoder trained to represent many shapes had marginally-worse performance representing the fine-grain details of the objects as compared to over-fitting a network on each case separately. In an effort to avoid this loss in fine-grain detail, and to avoid training a separate model for each case, a hypernetwork is used to generate main network weights θ_m for each solution instance. The hypernetwork is written as

$$\theta_m(\boldsymbol{\mu}) = N_h(\boldsymbol{\mu};\theta_h), \tag{3.23}$$

and the main-network prediction written as

$$\hat{\mathbf{q}}(\mathbf{x}';\theta_m(\boldsymbol{\mu})) = N_m(\mathbf{x}';\theta_m(\boldsymbol{\mu})), \tag{3.24}$$

where θ_h and θ_m are the weights and biases of the hypernetwork and main network. In the following experiments, simple MLPs are used for both the main network and hypernetwork and are referred to as one-shot dense hypernetworks; all of the weights and biases contained in θ_m are generated at once as one large vector which is sliced and reshaped as required. The training loss depends on the main-network predictions, but only the hypernetwork weights θ_h are adjusted during training; the loss gradients are back-propagated through to the hypernetwork.

3.2.5.1 Network-size Scaling Considerations

One-shot dense hypernetworks have an important scaling consideration relating the number of trainable hypernetwork parameters in θ_h to the size of the main network θ_m. Consider a main network with L_m hidden layers each with a hidden dimension H, and an analogous hypernetwork with hidden dimension H except for the final hidden layer which has dimension H_L. Since all of the main network weights are generated at once, the output layer of the hypernetwork has roughly H_L times as many weights as the main network. That is,

$$\dim(\theta_h) \propto \dim(\theta_m) H_L \propto L_m H^2 H_L, \tag{3.25}$$

showing that the total number of hypernetwork weights is linear in main-network depth L_m, quadratic in man-network hidden-dimension H, and linear in hypernet-final-hidden-dimension H_L. This is intuitive given the use of dense layers throughout, but Equation 3.25 is developed in more detail in the Appendix Section A. Once a main network architecture is chosen with L_m and H selected, then H_L is the remaining term to be selected which drives the number of weights, which of course must be managed for the given training hardware and problem at hand, suggesting selecting $H_L < H$. This leads to an encoder-like interpretation for the hypernetwork, where the final hidden state is an H_L-dimensional embedding for a linear neural-network generator; the hypernetwork output layer. The interpretation exists regardless, even if $H_L \geq H$, but the bottleneck structure brings it out, and is reminiscent of autoencoder-style models.

The scaling relationship of Equation 3.25 also highlights a difference between this and many other hypernetwork models. Frequently the goal is to reduce the number of trainable parameters in a given main network while retaining predictive accuracy, while here the differences in accuracy and generalization are studied between vector-embedded and weight-embedded coordinate-based networks. In most of the following experiments H_L is chosen to be simply $H_L = H = 50$, across main and hypernetworks, meaning that the size of the DVH model is roughly 50 times larger than the DV-MLP model. This is taken into account by comparing the time and resources needed to train each type of model while also assessing predictive accuracy. Further, the training methods which follow in Section 3.2.5.2 are found to have a large impact on the required training time. Differing hypernetwork architectures which reduce the number of trainable parameters are possible but are outside the scope of this work, with a specific example given in Chapter 6.

3.2.5.2 Training Considerations

Equation 3.25 implies that the computational complexity of training DVH models may follow a similar scaling relative to training DV-MLP models. This is investigated by considering two approaches for training DVH models, with differences relating to details of model evaluation. Consider a minibatch j consisting of N spatial locations drawn from each of M solution instances. Represent the minibatch as a tuple of tensors, $(\mathbf{M}_j, \mathbf{X}_j, \mathbf{Q}_j)$, where tensor $\mathbf{M}_j$ holds hypernetwork inputs, $\mathbf{X}_j$ holds main network inputs, and $\mathbf{Q}_j$ holds the target solution variables. The tensor shapes vary between the following methods which are described below and represented in Figure 3.4.

- **Method 1: Fully-Mixed Batches** Mini-batches are created which consist of points from different cases, with both the hypernetwork and main network forward-propagated for each data point, meaning hypernetwork input vector $\boldsymbol{\mu}$ is tiled across the mesh. In this scenario all tensors have two axes, $\dim(\mathbf{M}_j) = (M \times N) \times n_\mu$, $\dim(\mathbf{X}) = (M \times N) \times n_{x'}$, and $\dim(\mathbf{Q}) = (M \times N) \times n_q$ as shown in Figure 3.4a. Let $\mathcal{C}_M/\mathcal{C}_H$ be the cost for each the main and hypernetwork, then forward-propagating the minibatch is proportional to

$$\mathcal{C}_{FM} \propto (M \times N)\mathcal{C}_H + (M \times N)\mathcal{C}_M \tag{3.26}$$

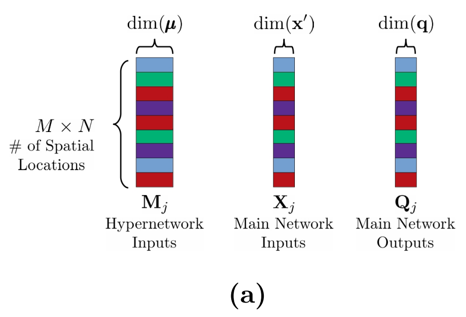
The data is *fully mixed* by shuffling over all locations and solution instances used in building the minibatch. The batch size then corresponds to the number of spatial locations where predictions are sought.

- **Method 2: Batch-by-Case** This method takes advantage of the fact that design variables $\boldsymbol{\mu}$ apply to an entire solution instance and that the hypernetwork is a neural-network generator. A single forward pass of the hypernetwork is combined with multiple forward passes of the main network for a given number of spatial locations, all coming from the same solution instance, defined by $\boldsymbol{\mu}$. In this scenario the design-variable tensor has two axes, $\dim(\mathbf{M}_j) = M \times n_\mu$, while the other tensors have three; $\dim(\mathbf{X}) = M \times N \times n_{x'}$, and $\dim(\mathbf{Q}) = M \times N \times n_q$, as represented in 3.4b. The complexity of forward-propagating the minibatch scales as

$$\mathcal{C}_{BC} \propto M \times \mathcal{C}_H + (M \times N)\mathcal{C}_M. \tag{3.27}$$

  The batch size is then the number of cases per mini-batch, where the same number of spatial locations N are evaluated for each case in each minibatch.

Comparing first terms between Equations 3.26 and 3.27 shows the batch-by-case first-term complexity is reduced by a factor of $N \times \mathcal{C}_H$ due to the many fewer expensive hypernetwork calls. The actual computational complexity will be measured through profiling in later sections.



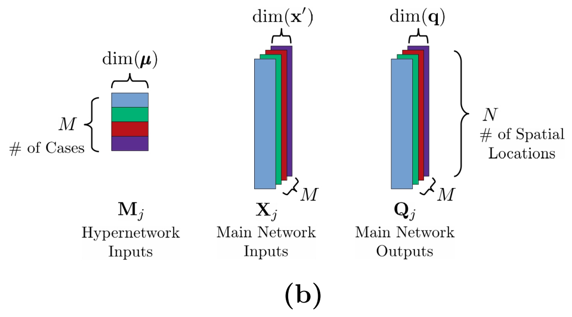


Figure 3.4: Illustrating the difference between (a) fully-mixed batches and (b) batch-by-case training mini-batches, by considering the shape and dimension of the training arrays for a single batch j. Colors correspond to data from a given case.

3.2.6 Method 3: Non-linear Independent Dual System (NIDS)

NIDS may be conceptualized as a design-variable hypernetwork which generates only the weights and biases of the final output layer of the main network. In this context, the hypernetwork is referred to as the parameter network. The spatial network, which is the main network *except* the output layer, and its output vector are defined as

$$N_x(\mathbf{x}'; \theta_x) \triangleq \mathbf{h}_x \in \mathcal{H} \subset \mathbb{R}^{n_h}. \tag{3.28}$$

The parameter network and its output are defined as

$$N_\mu(\boldsymbol{\mu}; \theta_\mu) \triangleq \mathbf{w}_\mu \in \mathcal{W} \in \mathbb{R}^{(n_q \times n_h)+n_q} = \mathbf{W}_\mu, \mathbf{b}_\mu. \tag{3.29}$$

Given this, the overall prediction is written as

$$\hat{\mathbf{q}}(\mathbf{x}', \boldsymbol{\mu}; \theta_x, \theta_\mu) \triangleq \mathbf{W}_\mu \mathbf{h}_x + \mathbf{b}_\mu. \tag{3.30}$$

In equation 3.29, the flattened output $\mathbf{w}_\mu$ is split and reshaped appropriately to form $\mathbf{W}_\mu$ and $\mathbf{b}_\mu$. Thus, the parameter network generates the weights and biases for the linear output layer of the main network, and the spatial network provides the final hidden state. Alternatively, $\mathbf{W}_\mu$ may be viewed as a basis matrix dependent on the problem parameters, while final hidden state $\mathbf{h}_x$ are the coordinates to that basis for a specific location in space $\mathbf{x}$. $\mathbf{W}_\mu$ is reused for every spatial location where a prediction is desired, while a new $\mathbf{h}_x$ is required. Thus, a prediction for a given case requires one forward-pass of the parameter network and as many forward-passes of the spatial network as there are spatial locations, much like DV-Hnet. Figure 3.5 shows a schematic diagram which emphasizes reuse of $\mathbf{w}_\mu$.

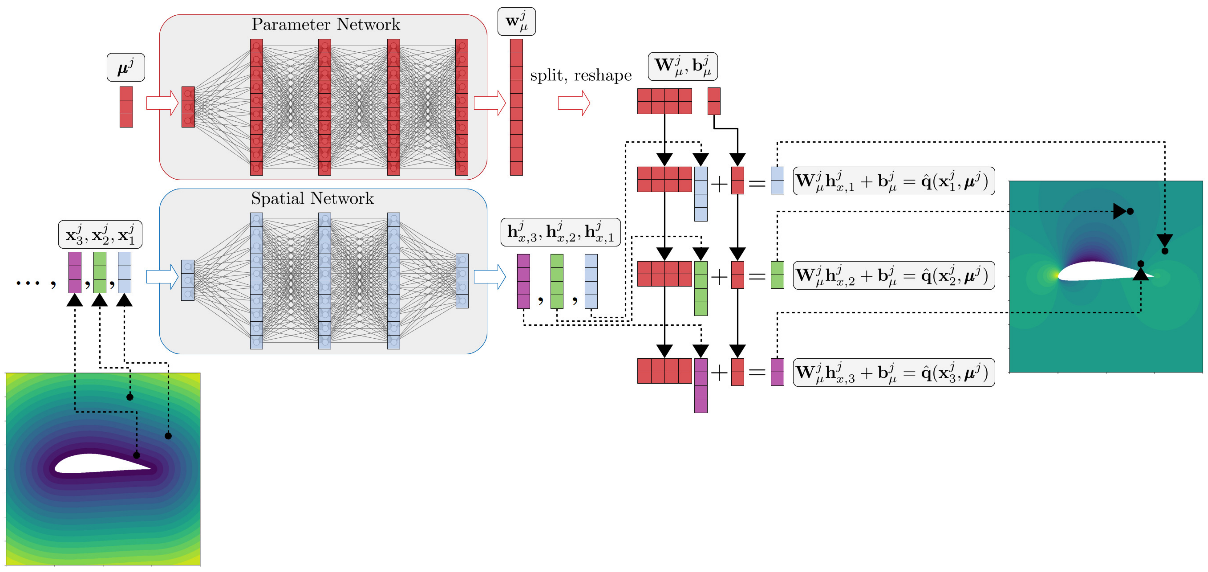

Parameter Network
$\boldsymbol{\mu}^j$
$\mathbf{w}_\mu^j$
split, reshape
$\mathbf{W}_\mu^j, \mathbf{b}_\mu^j$
Spatial Network
$\dots, \mathbf{x}_3^j, \mathbf{x}_2^j, \mathbf{x}_1^j$
$\mathbf{h}_{x,3}^j, \mathbf{h}_{x,2}^j, \mathbf{h}_{x,1}^j$
$\mathbf{W}_\mu^j \mathbf{h}_{x,1}^j + \mathbf{b}_\mu^j = \hat{\mathbf{q}}(\mathbf{x}_1^j, \boldsymbol{\mu}^j)$
$\mathbf{W}_\mu^j \mathbf{h}_{x,2}^j + \mathbf{b}_\mu^j = \hat{\mathbf{q}}(\mathbf{x}_2^j, \boldsymbol{\mu}^j)$
$\mathbf{W}_\mu^j \mathbf{h}_{x,3}^j + \mathbf{b}_\mu^j = \hat{\mathbf{q}}(\mathbf{x}_3^j, \boldsymbol{\mu}^j)$

Figure 3.5: Schematic diagram of a NIDS network emphasizing reuse of parameter network outputs. Ref. [174] was used in making this figure.

Functionally, the mapping of the spatial network is

$$N_x \quad : \quad \mathcal{X}' \to \mathcal{H}, \tag{3.31}$$

the mapping of the parameter network is

$$N_\mu \quad : \quad \mathcal{M} \to \mathcal{W}, \tag{3.32}$$

and that of the linear output layer may be written as

$$N_o \quad : \quad \mathcal{W} \times \mathcal{H} \to \mathcal{Q}. \tag{3.33}$$

Note that the triple $(\mathcal{W}, \mathcal{H}, N_o)$ defines a dual system over a subset of the real numbers $\mathbb{R}$ when the system state $\mathbf{q}$ is one-dimensional. When the system state is n-dimensional, then

n dual systems are induced of the form $(\mathcal{W}_i', \mathcal{H}, N_{o,i})$, with bilinear map $N_{o,i}$ described as

$$N_{o,i} \; : \; \mathcal{W}_i' \times \mathcal{H} \to \mathcal{Q}_i', \tag{3.34}$$

which corresponds to the inner product of each row of $\mathbf{W}_\mu$ with $\mathbf{h}_x$, and addition of the bias term. In this case, $\mathcal{W}_i' \subset \mathbb{R}^{n_h+1}$ and $\mathcal{Q}_i' \subset \mathbb{R}$, corresponding to the ith dimension of the system state. The maps are bilinear in $\mathbf{w}_\mu$ and $\mathbf{h}_x$, not in $\mathbf{x}$ and $\boldsymbol{\mu}$ due to the non-linear nature of neural networks.

From this we can see that the name 'non-linear independent dual system' is an apt description of how the model output layer operates. That is, the tripe describing the output layer $(\mathcal{W}, \mathcal{H}, N_o)$ is a dual system or induces multiple dual systems. The vectors from each space $\mathbf{w}_\mu \in \mathcal{W}$ and $\mathbf{h}_x \in \mathcal{H}$ serve as inputs to the dual system bilinear map N_o, and are generated non-linearly and independently by the neural networks.

3.2.6.1 Comparison of Proposed Methods

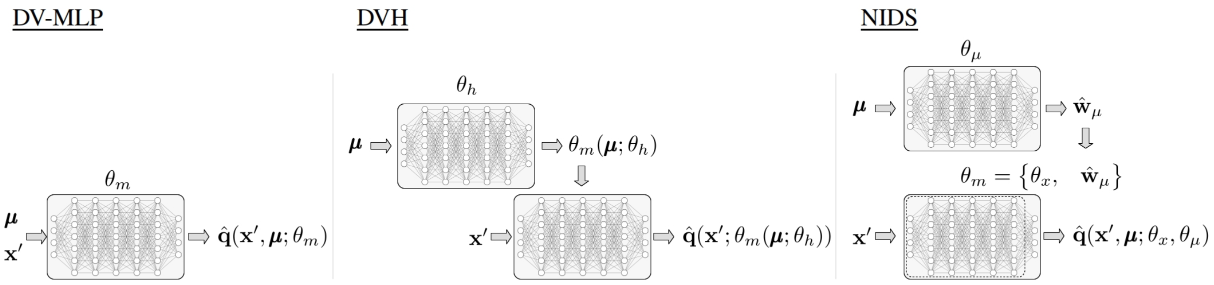

DV-MLP
θ_m
$\boldsymbol{\mu}$
$\mathbf{x}'$
$\hat{\mathbf{q}}(\mathbf{x}', \boldsymbol{\mu}; \theta_m)$
DVH
θ_h
$\boldsymbol{\mu}$
$\theta_m(\boldsymbol{\mu}; \theta_h)$
$\mathbf{x}'$
$\hat{\mathbf{q}}(\mathbf{x}'; \theta_m(\boldsymbol{\mu}; \theta_h))$
NIDS
θ_μ
$\boldsymbol{\mu}$
$\hat{\mathbf{w}}_\mu$
$\theta_m = \{\theta_x, \quad \hat{\mathbf{w}}_\mu\}$
$\mathbf{x}'$
$\hat{\mathbf{q}}(\mathbf{x}', \boldsymbol{\mu}; \theta_x, \theta_\mu)$

Figure 3.6: Schematic diagram comparing the different proposed methods. Ref. [174] was used in making this graphic.

A diagram comparing the various methods is shown in Figure 3.6, introducing DV-MLP, DVH and NIDS models from left to right. DV-MLP is a simple dense network, with weights and biases collected in θ_m, set above the main-network graphic to indicate the dependence. DVH introduces the design-variable hypernetwork and its trainable parameters θ_h in addition to the main network, with correspondence to the DV-MLP main network. The hypernetwork generates the weights for the main network as a function of the design variables $\theta_m(\boldsymbol{\mu})$. In the NIDS approach, the parameter network generates the vector $\hat{\mathbf{w}}_\mu$, and the figure highlights how the main network weights and biases θ_m may be considered as the spatial network weights and biases θ_x in addition to vector $\hat{\mathbf{w}}_\mu$. Additionally, the dashed line in the main network corresponds to the NIDS spatial network, highlighting the slight difference between the two in the context of the other models.

3.3 Modal Interpretation of NIDS Predictions

Proper orthogonal decomposition (POD) and dynamic mode decomposition (DMD) are popular techniques originally developed for analysis of flow fields and turbulence [175, 176], and are now frequently used for model order reduction, prediction, and control [64, 177, 62]. Here POD reconstruction is examined, and it is shown how NIDS predictions may be interpreted in a similar manner.

Consider M solution snapshots of a system with a single state variable defined on a D-dimensional common mesh with N nodes, where each solution corresponds to a unique set of design variables $\boldsymbol{\mu}$. The solution at a spatial location i is written as $q(\mathbf{x}_i, \boldsymbol{\mu}^j) \in \mathbb{R}$, and a snapshot of the state (FOM solution) at all locations may be written as $\mathbf{q}_N^j \in \mathbb{R}^N$ or $\mathbf{q}_N$. Additionally, collect mesh spatial-coordinates in matrix $\mathbf{X} \in \mathbb{R}^{N \times D}$ and all solution snapshots in data matrix $\mathbf{Q} \in \mathbf{R}^{N \times M}$. Follow Sections 1.2.3 or 1.2.4 (noting the change in notation regarding $\mathbf{Q}/\mathbf{X}$) to get the truncated, rank r POD basis, $\boldsymbol{\Phi} = \mathbf{U}[:, 1 : r] \in \mathbb{R}^{N \times r}$, where $\mathbf{U}$ contains the left singular vectors of mean-centered snapshot matrix $\mathbf{Q}$. A snapshot from the collection may be reconstructed using basis $\boldsymbol{\Phi}$ by first computing the basis coefficients $\mathbf{a}$, following Equations 1.65. Additionally, each POD mode is a function of the mesh coordinates in $\mathbf{X}$ and the basis coefficients are dependent on the parameters $\boldsymbol{\mu}$ associated with snapshot $\mathbf{q}_N$. Then the POD projection may be written as

$$\hat{\mathbf{q}}_N(\mathbf{X}, \boldsymbol{\mu})_{\text{POD}} = \boldsymbol{\Phi}\mathbf{a} = \sum_{i=1}^{r} a_i(\boldsymbol{\mu})\boldsymbol{\phi}_i(\mathbf{X}), \tag{3.35}$$

where addition of the mean-column is ignored, as it may be added to both NIDS and POD predictions/reconstructions.

Next consider the approximation of a snapshot $\mathbf{q}_N$ with a NIDS network, where a prediction for location i is

$$\hat{q}(\mathbf{x}_i, \boldsymbol{\mu}) = \mathbf{W}_\mu(\boldsymbol{\mu})\mathbf{h}_x(\mathbf{x}_i) + b_\mu, \tag{3.36}$$

where $\mathbf{x}_i$ is the ith row of $\mathbf{X}$. The full snapshot is attained by stacking the NIDS predictions at all locations. The system state is 1D, so $\mathbf{W}_\mu \in \mathbb{R}^{1\times n_h}$ is a row vector. Re-express this as a column vector as $\mathbf{w}(\boldsymbol{\mu}) = \mathbf{W}_\mu^T$, noting that $\mathbf{w}(\boldsymbol{\mu}) \neq \mathbf{w}_\mu$ which is the parameter network output per Equation 3.29. Collect all spatial-network evaluations in $\mathbf{H}_x$, defined as

$$\mathbf{H}_x := \begin{bmatrix} | & | & & | \\ \mathbf{h}_x(\mathbf{x_1}) & \mathbf{h}_x(\mathbf{x_2}) & \cdots & \mathbf{h}_x(\mathbf{x_m}) \\ | & | & & | \end{bmatrix} \in \mathbb{R}^{n_h\times N} \tag{3.37}$$

Then let $\hat{\mathbf{q}}_N(\mathbf{X}, \boldsymbol{\mu})_{\text{NIDS}} \in \mathbb{R}^N$ collect all NIDS predictions and let $\mathbf{b}_N(\boldsymbol{\mu}) = \vec{\mathbf{1}}_N \times b_\mu \in \mathbb{R}^N$ tile the bias over the mesh. Further, let $\mathbf{h}_{x,i} \in \mathbb{R}^N$ be the ith row of $\mathbf{H}_x$; the ith entry of spatial-network output $\mathbf{h}_x$ collected over all mesh locations. Then the NIDS predictions may be written as

$$\hat{\mathbf{q}}_N(\mathbf{X}, \boldsymbol{\mu})_{\text{NIDS}} = \mathbf{H}_x^T\mathbf{w}(\boldsymbol{\mu}) + \mathbf{b}_N(\boldsymbol{\mu}) = \sum_{i=1}^{n_h} w_i(\boldsymbol{\mu})\mathbf{h}_{x,i}(\mathbf{X}) + \mathbf{b}_N(\boldsymbol{\mu}). \tag{3.38}$$

Equation 3.38 for the NIDS prediction has a very similar structure as Equation 3.35 for the POD reconstruction, except for the additional bias term and the fact that the NIDS network may also consume an additional MDF/SDF coordinate in $\mathbf{x}$. However $\mathbf{X}$ may be defined to include this additional coordinate without affecting the POD analysis. Thus $\mathbf{h}_{x,i}$ may also be interpreted as a spatial mode and plotted/analyzed in a similar way as a POD mode. A key difference is that the POD modes were obtained via decomposition of a snapshot matrix, requiring a consistent discretization, while NIDS operates pointwise and the modes are learned during training.

Equation 3.38 inverts the role of the spatial and parameter networks as previously described. Previously the parameter network was conceptualized as providing a basis which applies globally for a solution instance, while the spatial network provides coordinates relative to that basis to generate the prediction at a spatial location. Whereas with the above interpretation, the coordinates provided by the spatial network are collected for all locations and interpreted as basis functions, with the parameter network now providing the coordinates for the global modes. The POD modes have additional structure beyond orthonormality since they are obtained via SVD, and thus the modes have an inherent order defined by the singular values. And more critically, it may be shown that that the POD reconstruction of data matrix $\mathbf{Q}$ is the optimal rank r reconstruction in a Frobenius norm sense. NIDS modes lack these structures and this guarantee, but indeed there are possibilities to impose such structure by including appropriate penalizing terms in the loss function during training.

3.4 Piecewise Learning-Rate Schedule

A piecewise learning-rate schedule was devised and used for many later-reported results, consisting of periods with constant and exponentially-decaying learning-rates. This was devised to combat large variation around training-loss plateaus seen in some instances, and to stabilize DVH M1 training dynamics which showed large training-loss fluctuations. To define the learning rate at optimizer-step i, written as $\alpha^{(i)}$, denote the number of optimizer steps with constant learning rate as s_c, the decay rate r, and decay steps as s_d, then

$$\alpha^{(i)} = \begin{cases} \alpha_0 & \text{if } i < s_c \\ \alpha_0 \times r^{(i-s_c)/s_d} & \text{if } i \geq s_c \end{cases} \tag{3.39}$$

defines the learning rate schedule. The interpretation is that the learning rate will decrease by a factor of r every s_d steps when the exponential term is active. To convert between optimizer steps and dataset iterations or epochs, use

$$s = n_{\text{epochs}} \times n_{\text{updates per epoch}}.$$

In mini-batch stochastic gradient descent (SGD) and single-example true SGD, the noise added to the training process due to the stochastic sampling of the dataset-loss-gradient is beneficial in reaching a better local minima, but also prevents the network from reaching the minima exactly [178]. It has been shown that the variance of the fluctuations around local minima are proportional to the learning rate [179], and thus to reduce the size of the fluctuations the learning rate may be decreased, or alternatively the mini-batch size increased. Intuitively, when the training loss plateaus and large fluctuations are seen with small mini-batch sizes, then this means that the "general" or common part of the underlying function has been learned as the mini-batch gradients point in differing directions from another. The optimizer then overshoots the corrections in each direction resulting in the training-loss oscillations. Thus reducing the learning rate when this is observed should decrease the overshooting and increase convergence.

It is generally accepted in the deep-learning community that large initial learning rates aid in generalization [180], at the cost of longer overall training times as compared to small initial learning rates. The strategy of using a large initial learning rate followed by an annealing period where the learning rate decreases is commonly used. For example, in training highly-influential CNNs for image recognition, the learning rate was decreased manually by a factor of 10 when a validation plateau was observed, with this repeated up to three times [133, 181, 182]. The piecewise learning-rate schedule chosen follows this line of thinking, and instead of using a stair-step decrease in learning rate a smoother, exponential decrease is applied such that the learning rate decreases by an order of magnitude over a given number of training epochs. Exponential learning-rate decay is a specific and widely used form of learning-rate annealing [183], although it is more frequently used from the beginning of the training process.

Chapter 4

Surrogate Modeling Applications

The proposed methods of Chapter 3 are applied to a variety of problems in this Chapter, with an additional compressor-rotor application covered in greater detail in Chapter 5. Not all methods are applied to every application, given that this is not generally possible since DCNN models have discretization dependence. Considered problems include a 2D Poisson problem designed to test discretization independence and translational invariance is presented in Section 4.2. Next, the 2D incompressible RANS equations are considered in the context of a vehicle aerodynamics problem in Section 4.3. And finally, the 3D incompressible RANS equations are applied to the Ahmed Body, a simplified vehicle shape, in Section 4.5. But first, some implementation details are provided in Section 4.1.

4.1 Model Implementation and Training Details

All models are implemented via Python 3.7 classes using Tensorflow v2.6 [126] and are trained using Nvidia RTX A6000 48 GPUs. The DV-MLP implenentation uses off-the-shelf Tensorflow-Keras sequential models, constructed using the Keras functional API. DVH and NIDS models subclass Tensorflow-Keras Models (`tf.keras.Model`), overwriting the `call` method as required, depending on the batching method used per Section 3.2.5.2. The DVH implementation uses a Tensorflow-Keras sequential model for the hypernetwork, required during class instantiation. Tensorflow-Keras models are useful as they provide high-level abstraction and contain easy-to-use functions for common tasks, such as training models and saving/loading network weights. All model weights are initialized using the Glorot-uniform weight-initialization scheme [249], and trained using calls to `tf.keras.Model.fit()` or custom training loops. Adam optimizer [140] with default settings and a mean squared error loss is used for all numerical experiments, with the piecewise learning rate schedule of Section 3.4 called out when used. Swish activation function is used for all hidden layers, and all output layers use a linear activation function. All inputs and outputs are min-max normalized using the statistics of the training dataset, see Section 2.2.1. Mixed or single precision is used in training all models, and model checkpoints are used to save the model weights corresponding to the final, best training, and best validation losses obtained as training progresses.

The error metrics of root-mean-squared-error (RMSE) and mean-absolute-error (MAE) are computed by averaging across all n_j mesh points in each case, and then averaging across all n_c cases. The root-mean-squared-error (RMSE) for the kth component of the state is then defined as

$$\text{RMSE}_k \triangleq \frac{1}{n_c}\sum_{j=1}^{n_c}\sqrt{\frac{1}{n_j}\sum_{m=1}^{n_j}\Big(\hat{\mathbf{q}}_k(\mathbf{x}_m^j,\boldsymbol{\mu}^j;\theta)-\mathbf{q}_k(\mathbf{x}_m^j,\boldsymbol{\mu}^j)\Big)^2}. \tag{4.1}$$

The mean-absolute-error (MAE) is computed analogously as

$$\text{MAE}_k \triangleq \frac{1}{n_c}\sum_{j=1}^{n_c}\frac{1}{n_j}\sum_{m=1}^{n_j}|\hat{\mathbf{q}}_k(\mathbf{x}_m^j,\boldsymbol{\mu}^j;\theta)-\mathbf{q}_k(\mathbf{x}_m^j,\boldsymbol{\mu}^j)|. \tag{4.2}$$

Both RMSE and MAE have units consistent with the predicted quantities, making them more intuitive than MSE alone. They provide similar measures, though the RMSE penalizes larger errors more than the MAE.

The mean-relative-L2-error (MRL2E) is also reported. To ease notation, gather all predictions for case j in matrix $\hat{\mathbf{Q}}^j \in \mathbb{R}^{n_j\times n_q}$, and all ground-truth in matrix $\mathbf{Q}^j \in \mathbb{R}^{n_j\times n_q}$. Then $\hat{\mathbf{Q}}^j[:,k]$ is the full snapshot for the kth component of the predicted state, for the jth case. Then the MRL2E for the kth state component may be expressed as

$$\text{MRL2E}_k \triangleq \frac{1}{n_c}\sum_{j=1}^{n_c}\frac{\|\hat{\mathbf{Q}}^j[:,k]-\mathbf{Q}^j[:,k]\|_2}{\|\mathbf{Q}^j[:,k]\|_2}\Big(\times 100\%\Big). \tag{4.3}$$

The final multiplication by 100% is placed in parentheses as this operation is not always performed. The distinction is made clear in context by reporting the value with or without the percentage symbol.

4.2 2D Poisson Equation

The Poisson equation with a source term is solved on a unit square two-dimensional domain with a randomly sized and oriented shape within the domain acting as another internal boundary. While the geometry of the domain is parametric, the meshes are not, with each mesh having different spatial extent, number of points, and topology. The square domain and its boundary (without the internal shape) are written as

$$B = \Big\{x, y \mid x, y \in [0,1]\Big\}, \tag{4.4}$$

and

$$\partial B = \Big\{x, y \mid x = 0, \text{ or } x = 1, \text{ or } y = 0, \text{ or } y = 1\Big\}, \tag{4.5}$$

respectively. Since each shape is different, the problem domains are also different. Let ∂S_j be the set of points defining the jth internal-shape boundary. Then let S_j represent the set of points which are either on the jth shape surface or enclosed by it. With this, the domain for the jth problem is

$$\Omega_j = B \cap \left(S_j - \partial S_j\right)^C, \tag{4.6}$$

with overall boundary given by

$$\partial \Omega_j = \partial B \cup \partial S_j. \tag{4.7}$$

The governing Poisson equation and source term are defined as

$$\nabla^2 q(x,y) = f(x,y), \quad x,y \in \Omega_j \tag{4.8}$$

$$f(x,y) = \begin{cases} +500 & x < 0.5 \\ -500 & x \geq 0.5 \end{cases} \tag{4.9}$$

with Dirichlet boundary conditions specified for all boundaries as

$$q(x,y) = 100, \quad x,y \in \partial\Omega_j. \tag{4.10}$$

Eight classes of shapes are considered, consisting of circles and polygons with 3-9 sides. 1000 instances of randomly scaled, located, and rotated shapes of each class are defined. Each shape is specified by its center point, the radius of its circumcircle (which is also the distance between the center point and each vertex), and a rotation angle measured positive counterclockwise from the x-axis. Example meshes and solution fields are shown in Figure 4.1. The ranges for each design variable are given in Table 4.1 and these entries are used to populate the design-variable vectors. The largest mesh contains 2677 points, while the smallest contains just 1341.

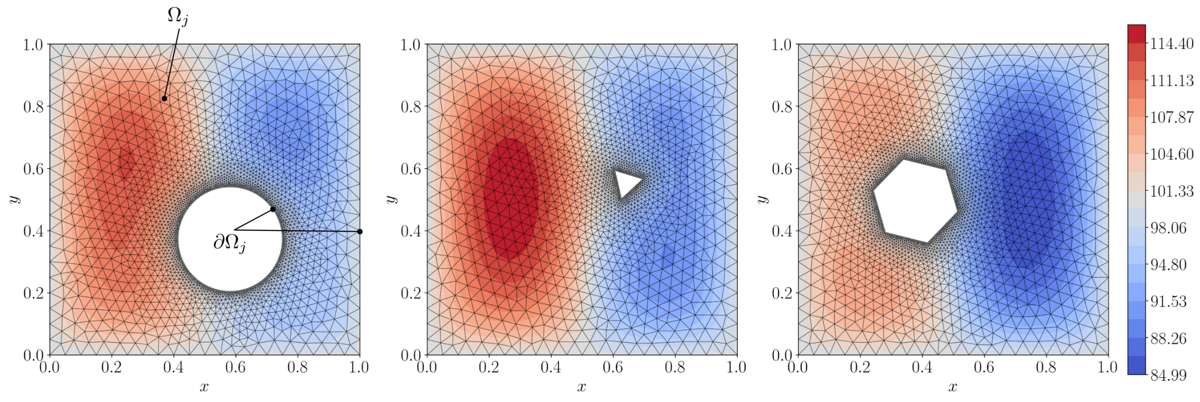


Figure 4.1: Example solution fields and meshes for three randomly generated shapes.

Table 4.1: Description of design variables for the 2D Poisson problem.

Symbol	Description	Range	Units
x_{cen}	x-coordinate of shape center point	$[0.3, 0.7]$	-
y_{cen}	y-coordinate of shape center point	$[0.3, 0.7]$	-
r	Shape radius	$[0.05, 0.2]$	-
γ	Shape rotation angle	$[-\pi, \pi)$	radians

Triangular meshes are generated for each random shape using the Gmsh Python API [184], though the surface mesh of each shape are fully specified programatically such that the distance between adjacent surface nodes, Δr, is approximately 5×10^{-3}. For polygons, the coordinates of the vertices are given as initial mesh points. Then the line connecting adjacent vertices is divided into n equal segments, where n is selected as the smallest integer such that the distance between mesh points is less than or equal to 5×10^{-3}. For circles, the number of points is chosen by rearranging the (approximate) arc length formula $\Delta s = r\Delta\gamma$ to compute the required $\Delta\gamma$ while setting $\Delta s = 5 \times 10^{-3}$. Then the number of points is chosen as $n = \left(\frac{2\pi}{\Delta\gamma}\right)$. The mesh for the rest of each domain Ω^j is generated using the Gmsh python API with mesh spacing on ∂B set to 5×10^{-2}. The meshes are saved to .vtk format and the governing equations solved using the finite element method using SfePY [185].

Two problem scenarios concerning the design-variable vector μ are considered. First is a scenario where $\boldsymbol{\mu}$ is 'incomplete,' with the class of the shape (circle, triangle, etc..) not explicitly given.

$$\boldsymbol{\mu} = \begin{bmatrix} x_{\text{cen}}, & y_{\text{cen}}, & r, & \gamma \end{bmatrix}^T \in \mathbb{R}^4 \tag{4.11}$$

This information enters the main network indirectly through the MDF coordinate present in $\mathbf{x}'$, defined the same as in Equation 4.21. The second scenario includes a one-hot-encoded label vector $\mathbf{c} \in \mathbb{R}^8$ in $\boldsymbol{\mu}$,

$$\boldsymbol{\mu} = \begin{bmatrix} x_{\text{cen}}, & y_{\text{cen}}, & r, & \gamma, & \mathbf{c}^T \end{bmatrix}^T \in \mathbb{R}^{12}. \tag{4.12}$$

One-hot-encoding is a commonly used feature-engineering tool for converting categorical labels to numerical representations, common in classification machine learning tasks. One-hot-encoded vectors are made by first enumerating the categories and giving each an integer label, starting with zero for zero-indexed languages such as python. If there are n categories, then each one-hot vector is first initialized as an n-dimensional zero vector, with a 1 added at the integer label index. The one-hot labels used are shown in Table 4.2, making the pattern clear.

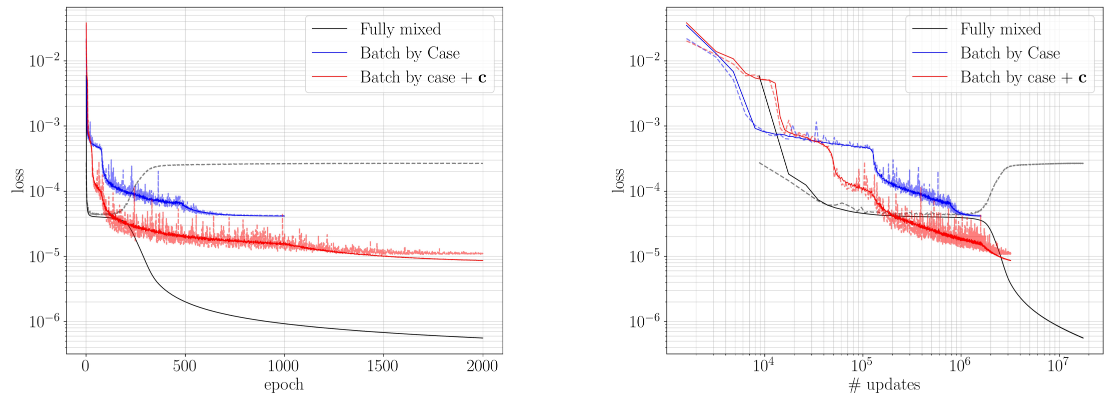
Table 4.2: One-hot encoded vectors

Shape Class	Integer Label	$\mathbf{c}$	Shape Class	Integer Label	$\mathbf{c}$
Circle	0	$\begin{bmatrix}1, & 0, & 0, & 0, & 0, & 0, & 0, & 0\end{bmatrix}^T$	Hexagon	4	$\begin{bmatrix}0, & 0, & 0, & 0, & 1, & 0, & 0, & 0\end{bmatrix}^T$
Triangle	1	$\begin{bmatrix}0, & 1, & 0, & 0, & 0, & 0, & 0, & 0\end{bmatrix}^T$	Septagon	5	$\begin{bmatrix}0, & 0, & 0, & 0, & 0, & 1, & 0, & 0\end{bmatrix}^T$
Square	2	$\begin{bmatrix}0, & 0, & 1, & 0, & 0, & 0, & 0, & 0\end{bmatrix}^T$	Octagon	6	$\begin{bmatrix}0, & 0, & 0, & 0, & 0, & 0, & 1, & 0\end{bmatrix}^T$
Pentagon	3	$\begin{bmatrix}0, & 0, & 1, & 1, & 0, & 0, & 0, & 0\end{bmatrix}^T$	Nonagon	7	$\begin{bmatrix}0, & 0, & 0, & 0, & 0, & 0, & 0, & 1\end{bmatrix}^T$

4.2.1 DVH: Effect of Training Method and Class Label

First the effects of training methods of Section 3.2.5.2 and the inclusion of class label vector $\mathbf{c}$ as model input are explored. A baseline result using fully-mixed training without the class label is first obtained. 80% of the 8000 available solutions are used as the training set, while the remaining 20% are withheld in the validation set, with an equal number of training cases used from each shape class. Adam optimizer was used with a learning rate of 1×10^{-4} and default settings otherwise. A batch size of 1500 points is used, resulting in 8765 batches/gradient updates per epoch. Then models with identical architecture were trained with and without the class-label vector $\mathbf{c}$ using the more efficient batch-by-case training method. Additionally, the piecewise learning rate schedule of Equation 3.39 was used. In this section no points from each training example were dropped out to make mini-batches of equal size. Instead, a custom training loop was implemented which allowed for varying mesh size. The models trained with the efficient batch-by-case routine also utilized a training fraction of just 0.2, where 200 solutions from each shape class were used to construct the training dataset.

Figure 4.2 shows the training loss for each scenario on the same set of axes, against both the number of epochs and optimizer updates. First, when fully-mixed training is used the loss rapidly decays initially before reaching a plateau, and then subsequently the training and validation losses diverge and over-fitting occurs. When batch-by-case training is then applied, the gross over-fitting is not seen while the best-validation-losses are observed to compare similarly to fully-mixed training before validation-loss divergence. Then finally, when class label **c** is included the gross over-fitting is again avoided and the overall performance is improved.



(a) Loss vs. epochs. **(b)** Loss vs. optimizer updates.

Figure 4.2: DVH training curves for the Poisson problem, showing the effect of training method and inclusion of class-label vector **c**. Training losses are solid lines while validation losses are dashed and transparent.

The training and validation loss metrics using the validation-best weights are summarized in Table 4.3. This shows that indeed the fully-mixed and batch-by-case compare similarly. The larger RMSE for fully-mixed while simultaneously having smaller MAE indicates that the fully-mixed predictions are subject to infrequent, larger errors, given that the RMSE penalizes larger deviations more than MAE. As expected from the training curves, the best overall results are achieved using batch-by-case with the class-label **c**. Generally, the results are very good across all methods, with mean errors less than 1% across the board, and the best errors less than 0.01%.

Table 4.3: Summary of training and validation error metrics for DVH models trained with various options on the Poisson problem.

Network Type	RMSE (train / val)	MAE (train / val)	MRL2E (train / val)
Fully Mixed	$2.08 \cdot 10^{-1}$ / $2.19 \cdot 10^{-1}$	$1.26 \cdot 10^{-1}$ / $1.32 \cdot 10^{-1}$	0.17% / 0.18%
Batch by Case	$1.88 \cdot 10^{-1}$ / $1.87 \cdot 10^{-1}$	$1.34 \cdot 10^{-1}$ / $1.34 \cdot 10^{-1}$	0.18/% / 0.19%
Batch by Case + **c**	$\mathbf{8.03 \cdot 10^{-2}}$ / $\mathbf{8.60 \cdot 10^{-2}}$	$\mathbf{5.60 \cdot 10^{-2}}$ / $\mathbf{6.00 \cdot 10^{-2}}$	**0.080%** / **0.086%**

The errors are seen to correlate strongly with the shape class, with predictive performance on triangles being worse than the other shape classes. This may be seen by plotting the MRL2E versus shape class as shown in Figure 4.3. This shows the similar performance between fully-mixed training and batch-by-case, where batch-by-case takes roughly an order of magnitude less time to train: see Table 4.7 which is for a different application but the trends hold here. All models in the present section were trained using mixed precision. Figure 4.3 also highlights how including the class-label vector greatly improves the predictions. Without the class label, the errors generally decrease with an increasing number of sides, with the exception of circles where the error increases relative to nonagons, with this trend not holding exactly for the fully-mixed model. However, when the class label is included the errors do indeed decrease as the number of sides are increased, where the circles have the smallest errors.

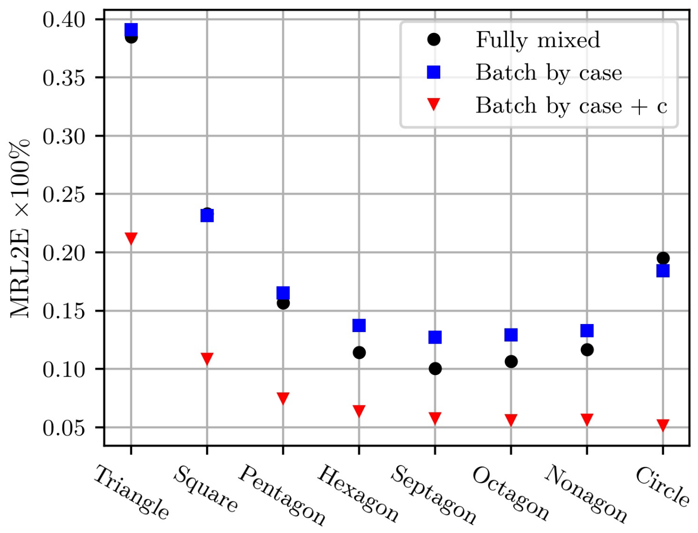


Figure 4.3: Mean-relative-L2-error (MRL2E) versus shape class for the three scenarios considered.

Next, to examine how well the network predictions achieve translational, rotational, and scale invariance, the joint distribution of RMSE versus each (non-label) generative factor are visualized using normalized 2D histograms in Figure 4.4, for the DVH model including the class label. The Pearson correlation coefficient is reported on each subplot. These plots show that the RMSE is very weakly correlated with the x and y locations of the center point, along with the rotation angle. This implies that the predictions have translational invariance as well as rotational invariance. There is greater correlation between RMSE and the shape radius however. This suggests weaker scale invariance in the predictions, but this may not be surprising given that the problem is not inherently scale-invariant, since the size of the domain is fixed while the shapes inside change size.

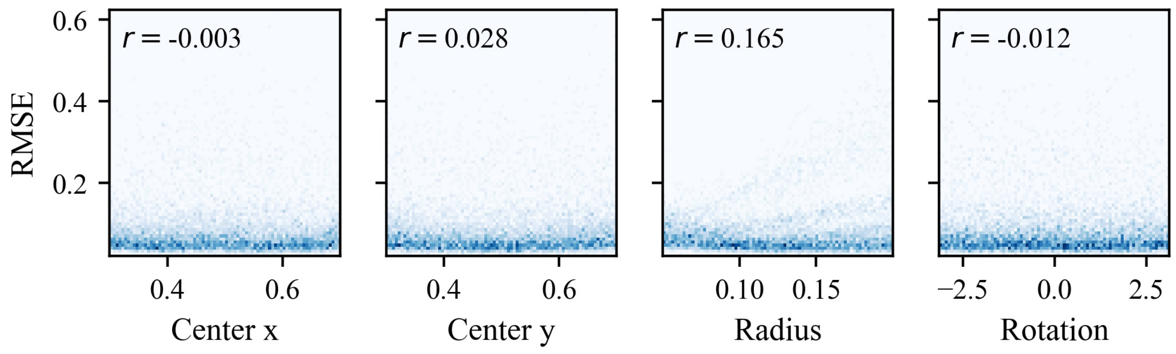
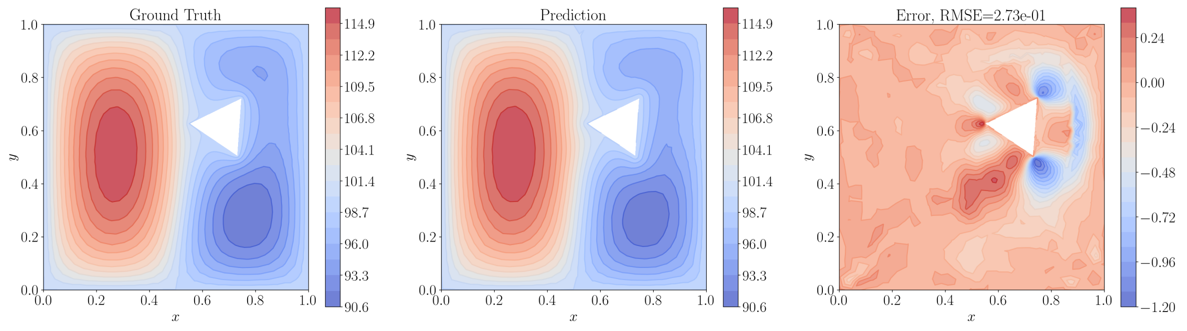
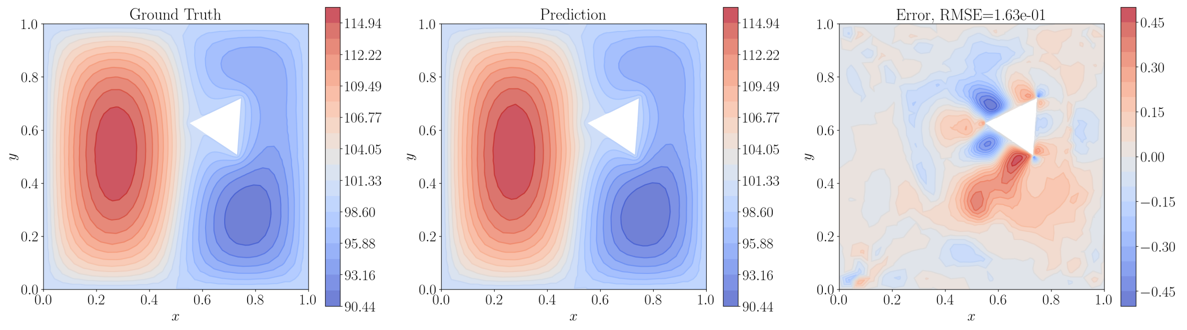


Figure 4.4: Histograms relating the RMSE to each generative factor, where the Pearson correlation coefficient is shown on each plot.

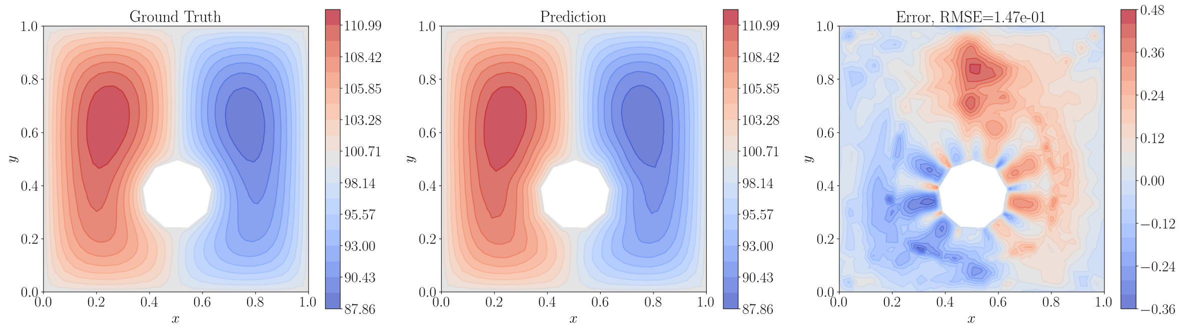


(a) Without class label **c**.

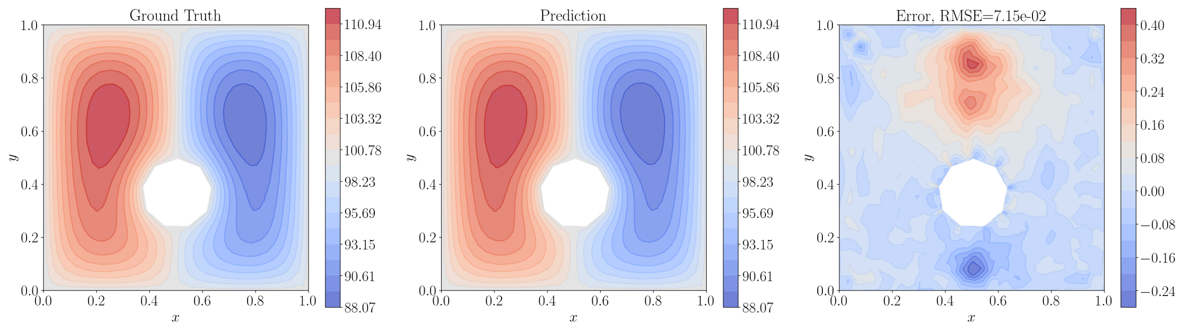


(b) With class label **c**.

Figure 4.5: DVH predictions on an unseen triangle, both without and with class-label vector **c** used as input. Note that the figures are generated independently of another, thus the differences in colorbar limits and location of contour lines in the ground truth and predictions.



(a) Without class label **c**.

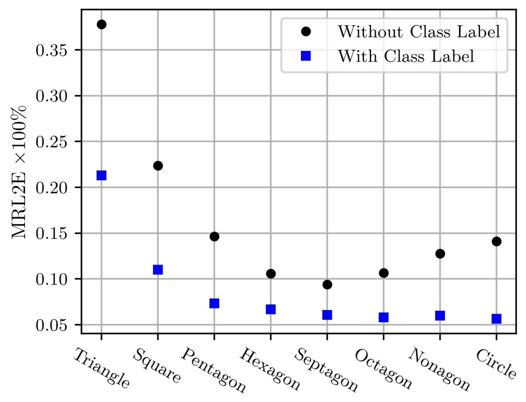


(b) With class label **c**.

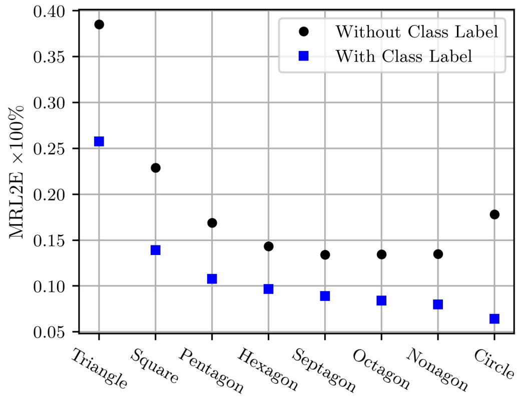
Figure 4.6: DVH predictions on an unseen nonagon, both without and with class-label vector **c** used as input. Note that the figures are generated independently of another, thus the differences in colorbar limits and location of contour lines in the ground truth and predictions.

4.2.2 Comparing Model Performance

Inclusion of the class label had a similar effect on DV-MLP and NIDS model predictions as was seen for DVH, with model errors being reduced as shown in Figure 4.7a and 4.7b for DV-MLP and NIDS respectively.



(a) DV-MLP



(b) NIDS

Figure 4.7: Comparing MRL2E by shape class with and without inclusion of the class-label vector as part of the design variables for (a): DV-MLP and (b): NIDS.

Model performance using the class labels is compared in Figure 4.8 and is summarized in Table 4.4. These show that DV-MLP and DVH perform very similarly but with DVH having slightly better error metrics across the board.

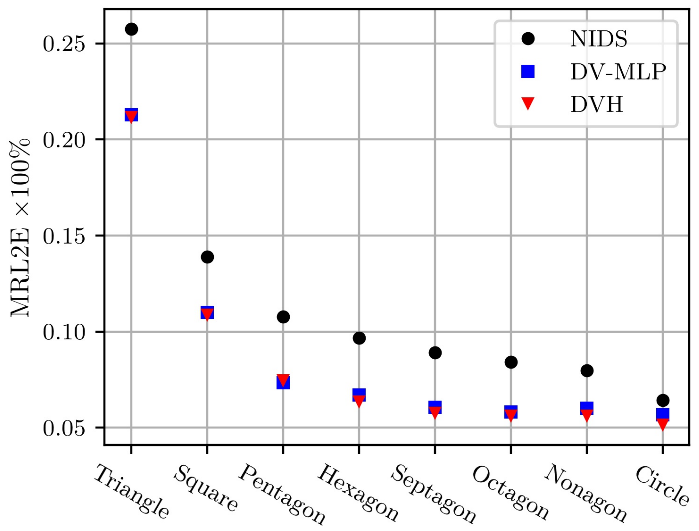


Figure 4.8: Mean-relative-L2-error (MRL2E) versus shape class for the three model types.

Table 4.4: Summary of training and validation error metrics on the Poisson problem for DV-MLP, NIDS, and DVH models with similar main and hypernetworks.

Network Type	RMSE (train / val)	MAE (train / val)	MRL2E (train / val)
DV-MLP	$8.29 \cdot 10^{-2}$ / $8.86 \cdot 10^{-2}$	$5.87 \cdot 10^{-2}$ / $6.25 \cdot 10^{-2}$	0.083% / 0.088%
NIDS	$1.10 \cdot 10^{-1}$ / $1.16 \cdot 10^{-1}$	$7.83 \cdot 10^{-2}$ / $8.33 \cdot 10^{-2}$	0.11/% / 0.12%
DVH	$\mathbf{8.03 \cdot 10^{-2}}$ / $\mathbf{8.60 \cdot 10^{-2}}$	$\mathbf{5.60 \cdot 10^{-2}}$ / $\mathbf{6.00 \cdot 10^{-2}}$	**0.080%** / **0.086%**

4.3 2D Incompressible RANS

4.3.1 Numerical Experiments I: Vehicle Aerodynamics

The Reynolds Averaged Navier Stokes (RANS) equations are derived by ensemble averaging the Navier Stokes equations and substituting the Reynolds-decomposed state variables. This decomposition separates the state variables into mean and fluctuating components $q = \langle q \rangle + q^{'}$, where q is a generic state variable, $\langle q \rangle$ is the mean, and $q^{'}$ is the fluctuating component. In the incompressible limit, the steady RANS equations may be written as

$$\begin{aligned} \nabla \cdot \langle \mathbf{u} \rangle &= 0 && (4.13) \\ \rho \langle u \rangle_j \frac{\partial \langle u \rangle_i}{\partial x_j} &= \frac{\partial}{\partial x_i}\left[\mu\left(\frac{\partial \langle u \rangle_i}{\partial x_j} + \frac{\partial \langle u \rangle_j}{\partial x_i}\right) - \langle p \rangle \delta_{ij} - \rho \langle u_i^{'} u_j^{'} \rangle\right], && (4.14) \end{aligned}$$

where velocity vector $\mathbf{u} = u\hat{\mathbf{i}} + v\hat{\mathbf{j}} + w\hat{\mathbf{k}}$. Non-dimensional flow quantities and inputs are used, denoted by a tilde $\tilde{\cdot}$. The Reynolds number Re is an important non-dimensional similarity parameter describing the relative importance of inertial and viscous forces in a flow, and since it has no dimensional counterpart the tilde is omitted.

$$\mathrm{Re} = \frac{\rho |\mathbf{u}| L}{\mu} = \frac{|\mathbf{u}| L}{\nu} \tag{4.15}$$

Free-stream flow conditions are used to define the Reynolds number, using the vehicle length L_v as the length scale. The freestream dynamic pressure is written as $q_\infty =$

$\frac{1}{2}\rho|\mathbf{u}_\infty|^2$, then the non-dimensional form of other relevant quantities are given as

$$\tilde{x} = \frac{x}{L_v} \tag{4.16}$$

$$\tilde{y} = \frac{y}{L_v} \tag{4.17}$$

$$\tilde{\phi} = \frac{\phi}{L_v} \tag{4.18}$$

$$\tilde{p} = \frac{p - p_\infty}{q_\infty} \tag{4.19}$$

$$\tilde{u}_i = \frac{u_i}{|\mathbf{u}_\infty|}. \tag{4.20}$$

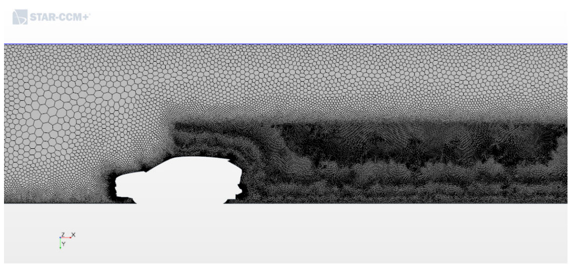
The non-dimensional pressure of Equation 4.19 is equivalent to the pressure coefficient c_p, and for incompressible flows the expression is further simplified by using the gauge pressure; taking $p_\infty = 0$. This is allowed because only derivatives of pressure enter into the incompressible momentum equation, Equation 4.14. If a compressible flow were considered this equivalency would not hold, as pressure enters directly into the compressible energy equation.

External vehicle aerodynamics are considered, with parametric vehicle shapes lying on unstructured, non-parametric meshes. The incompressible RANS equations were solved using Star CCM+ with the k-ϵ turbulence model. The dataset - generated by General Motors, Inc. - consists of 2D slices along the vehicle centerline for 124 unique vehicle shapes at speeds of 90 and 130 kilometers-per-hour (kph). The simulations utilize unstructured polyhedral meshes of varying size, with an example mesh shown in Figure 4.9a. Each vehicle shape is parameterized by 8 geometric parameters as summarized in Table 4.5, with all 124 shapes overlain on one set of axes in Figure 4.9b. The vehicle designs were selected using Latin hypercube sampling, and random subsets of the dataset are chosen for training and validation for each problem.

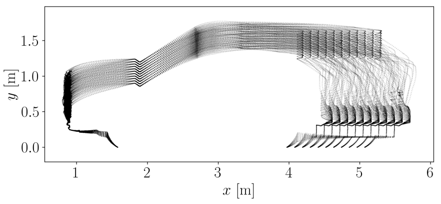
Table 4.5: Description of entries in geometric design-variable-vector $\boldsymbol{\mu}_{\text{geo}}$ for the 2D vehicle aerodynamics dataset.

Design Variable	**Units**	**Range**	**Design Variable**	**Units**	**Range**
Backlight Angle	Degrees	$[25, 57]$	Windshield Angle	Degrees	$[57, 63]$
Face Lip Angle	Degrees	$[0, 5]$	Hood Front Angle	Degrees	$[10, 20]$
Angle of Approach	Degrees	$[15, 25]$	Angle of Departure	Degrees	$[15, 25]$
Vehicle Length	mm	$[3800, 4900]$	Floor to Roof Height	mm	$[1448, 1788]$

The CFD meshes vary in size and topology, with the number of cells ranging from 108,748 to 115,751. In all sections, a spatial batch size of 54,000 points is used, resulting in two minibatches per case for a total of 108,000 mesh points used for each case. Mesh points are randomly dropped from each solution as required to make the minibatches, corresponding to 0.7%-6.7% of the points being left out for each training case. Reported training-error metrics include all mesh points, even if they were dropped from the training set, with further discussion of this in Section 4.3.3.



(a)



(b)

Figure 4.9: Pertaining the training dataset, (a) an example unstructured CFD mesh and (b) a composite image of all 124 vehicle shapes overlain on the same axes

The spatial input quantities are

$$\mathbf{x}' = [x,\ y,\ \phi(x, y)]^T \in \mathbb{R}^3, \tag{4.21}$$

and the predicted state is

$$\mathbf{q} = [p(x, y),\ u(x, y),\ v(x, y)]^T \in \mathbb{R}^3. \tag{4.22}$$

The design-variable vectors differ slightly depending on the number of speeds considered, and are given as

$$\boldsymbol{\mu} = \begin{cases} \boldsymbol{\mu}_{\text{geo}} \in \mathbb{R}^8 & n_{\text{speeds}} = 1, \\ [\text{Re},\ \boldsymbol{\mu}_{\text{geo}}^T]^T \in \mathbb{R}^9 & n_{\text{speeds}} > 1. \end{cases} \tag{4.23}$$

All spatial and predicted state quantities in $\mathbf{x}'$ and $\mathbf{q}$ are non-dimensionalized according to Equations 4.16-4.20. Non-dimensional quantities and dimensional analysis are widely used in fluid mechanics, leading to satisfying similarity solutions and increased interpretability. Following non-dimensionalization, all input and output vectors are min-max normalized component-wise using training-set stats before use with the neural networks. Details on this normalization are given in Section 2.2.1 . All reported errors are in the fully-dimensional units of the state, with the exception of training curves which correspond to fully-normalized quantities. Without normalization, the loss and loss-gradients may be biased towards the state quantities with the largest unit-scale. Normalization eases this problem and ensures that smaller output quantities are not ignored during training.

4.3.2 Model Architecture and Training Options

In this section 5 hidden-layer networks are used, both main and hypernetworks, each with a hidden dimension of 50 for all layers. The number of trainable weights for Section 4.3.1 are given in Table 4.6. As noted in Section 3.2.5.1, the DVH model has roughly $H_L = 50$ times as many trainable weights as the corresponding DV-MLP model, and the computational consequences of this are explored and quantified here through profiling. Three model-method combinations are considered, and include DV-MLP with its sole fully-mixed training mode, DVH method 1 (M1) fully-mixed batches, and DVH method 2 (M2) batch-by-case. See Section 3.2.5.2 for greater detail. The average step time, average compute time, and maximum memory usage during training among the models and methods are reported in Table 4.7, where the spatial batch size is 54,000 points for DV-MLP and DVH M1, and a corresponding case batch-size of 1 for DVH M2. All profiled results were obtained using `tensorboard` callbacks. Three different

Table 4.6: Number of trainable parameters for DV-MLP and DVH models for all baseline results of Section 4.3.1, where $H = H_L = 50$, $L_m = L_H = 5$.

Method	# Trainable Weights	
	Single Speed	Multiple Speeds
DV-MLP	10,953	11,003
DVH	548,853	548,903

levels of precision are compared for each method, including double-precision (float64), single-precision (float32), and mixed-precision (combination of float32 and float16). The reported values are averaged over the first 100 minibatches for 100 epochs of training, and the Δt step standard deviation is given in parentheses.

Table 4.7: Comparing training profiles across the models and training methods.

Type	Training Method	Precision	Δt step [ms]	Δt compute [ms]	Max. Mem. [GB]
DV-MLP	-	float64	18.1 (0.2)	17.2	1.20
DV-MLP	-	float32	3.7 (0.1)	2.9	0.62
*DV-MLP	-	mixed	3.6 (0.5)	1.9	0.54
DVH	M1 Fully-mixed	float64	490.0 (1.6)	488.9	9.70
DVH	M1 Fully-mixed	float32	55.6 (0.2)	54.4	4.85
DVH	M1 Fully-mixed	mixed	40.5 (0.8)	37.5	2.66
DVH	M2 Batch-by-case	float64	20.1 (0.4)	18.7	0.72
*DVH	M2 Batch-by-case	float32	4.4 (0.1)	3.3	0.36
*DVH	M2 Batch-by-case	mixed	4.3 (0.4)	2.3	0.28

There are several important takeaways from these profiled results. First, the precision has a large impact on the step-times and memory requirements, which decrease massively between float64 and float32 in all instances. Subsequent smaller improvements are seen moving between float32 and mixed precision. Next, M2 batch-by-case training decreases the step-time and memory requirements by roughly an order of magnitude as compared to M1 fully-mixed for a given precision. Further, the DVH M2 step times are comparable to the DV-MLP step times and consume less memory despite the much greater parameter count. Two average times are given, Δt step which includes all operations in a single optimizer update, and Δt compute which includes only GPU operations per update. Rows marked with an asterisk (*) in Table 4.7 may benefit from improvements in the data pipeline, as the overhead associated with those operations becomes appreciable as the Δt step times decrease.

When comparing the accuracy of the resulting fully-trained models, it was found that using single precision generally improved the mean-relative-L2 error (MRL2E) by roughly 1-8 percentage points as compared to mixed precision. The use of double precision negligibly improved the predictive performance beyond that seen with single precision, and in some cases single precision performed best overall. Thus, given the greatly improved step times from double to single precision, and the smaller improvements afforded by using mixed precision, all models in later sections are trained using single precision unless otherwise stated.

4.3.3 Single Vehicle Speed

Training information is given in Table 4.8 where s_t is the total number of training steps, and the values shown in parentheses are the corresponding number of epochs. Note that DVH M1 uses $\alpha_0 = 5 \times 10^{-5}$ instead of $\alpha_0 = 1 \times 10^{-3}$ as reported in the table due to large instabilities which frequently resulted in training-loss divergence with higher learning rates.

Table 4.8: Dataset and training options, where s_t is the total number of optimizer steps, and values in parentheses are dataset-iterations or epochs.

Dataset Options		**Learning Rate Schedule**	
vehicle speed	90 kph	α_0	1×10^{-3}
# training cases	99	r	0.1
# validation cases	25	s_t	1,386,000 (7000)
spatial batch size	54,000	s_c	198,000 (1000)
case batch size	1	s_d	594,000 (3000)

DV-MLP and DVH models are trained using Adam optimizer [112] with default options and single precision. In the case of DVH, both training methods described in Section 3.2.5.2 were employed, where DVH M1/M2 correspond to methods 1 and 2 respectively. Table 4.9 presents error metrics, revealing that DVH M2 performs best across the board, with both DVH methods having very similar and lesser errors as compared to DV-MLP.

DVH M2 slightly outperforms DVH M1 while requiring significantly less training time, approximately 4.4 ms per optimizer step versus 40.5 ms as shown in Table 4.7.

Table 4.9: Summary of training and validation error metrics at a vehicle speed of 90 kph.

$\hat{\mathbf{q}}_i$	**Network Type**	**RMSE (train / val)**	**MRL2E (train / val)**
p [Pa]	DV-MLP	11.5 / 12.6	4.75% / 5.12%
	DVH M1	8.7 / 11.5	3.60% / 4.61%
	DVH M2	**8.1** / **9.9**	**3.34%** / **3.99%**
u [m/s]	DV-MLP	0.66 / 0.74	2.85% / 3.18%
	DVH M1	0.59 / 0.68	2.56% / 2.92%
	DVH M2	**0.56** / **0.60**	**2.41%** / **2.59%**
v [m/s]	DV-MLP	0.41 / 0.49	9.94% / 11.9%
	DVH M1	0.34 / 0.47	8.28% / 11.3%
	DVH M2	**0.27** / **0.38**	**6.72%** / **9.22%**

Each model's ability to infill or predict a seen case at unseen spatial locations may be evaluated since training cases had points randomly dropped out or truncated during minibatch construction. Error metrics were calculated separately for the truncated and retained locations, with the case-wise mean-absolute error (MAE) for truncated locations plotted against that for retained locations in Figure 4.10 for DV-MLP and DVH M2. The dashed line is the identity line $y(x) = x$, meaning points which lie above the dashed line correspond to truncated-point errors which are larger than those for the retained points, and the opposite for points below the line. The points are generally located near the line, some above and some below, indicating that the performance is very similar between the retained and truncated groups. This is quantified in Table 4.10 where the mean error metrics for retained and truncated points are given separately, while the training errors reported in Table 4.9 include all points for the instance. This shows that the error metrics are nearly identical between the two groups for all models, demonstrating that the models are effective in this in-filling scenario.

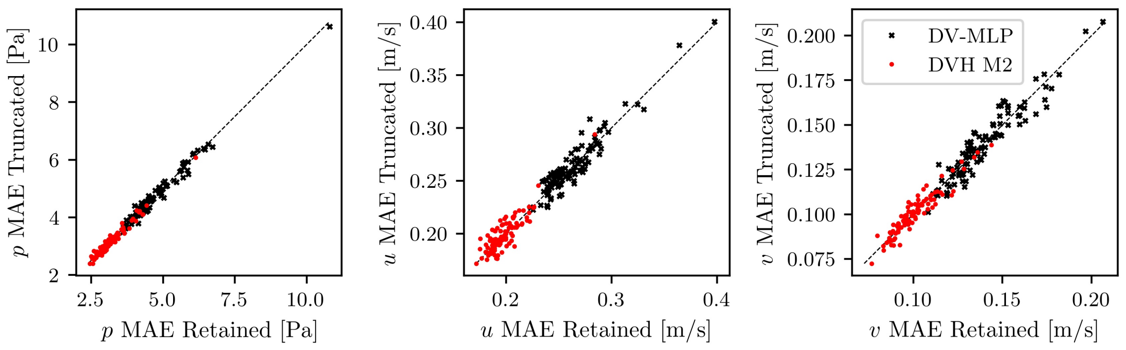
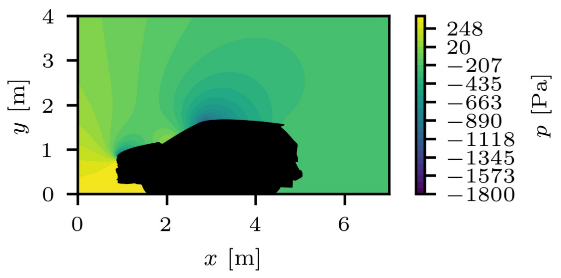


Figure 4.10: Mean absolute error over points truncated from training cases versus those retained and used in training for DV-MLP and DVH M2 predictions for each field variable.

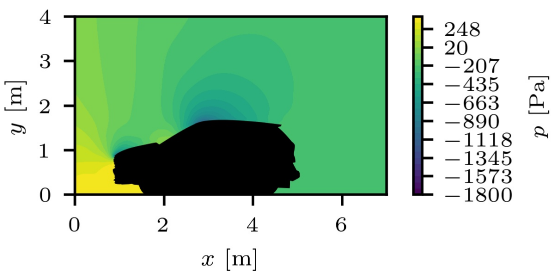
Table 4.10: Comparing training-case error metrics between points with are retained and actually used in training versus those which are truncated.

$\hat{\mathbf{q}}_i$	Network Type	RMSE (Retained / Truncated)	MRL2E (Retained / Truncated)
p [Pa]	DV-MLP	11.5 / 11.3	4.75% / 4.66%
	DVH M1	8.7 / 8.5	3.60% / 3.49%
	DVH M2	**8.1 / 8.0**	**3.34% / 3.27%**
u [m/s]	DV-MLP	0.66 / 0.66	2.85% / 2.86%
	DVH M1	0.59 / 0.60	2.55% / 2.58%
	DVH M2	**0.56 / 0.56**	**2.41% / 2.42%**
v [m/s]	DV-MLP	0.41 / 0.40	9.94% / 9.88%
	DVH M1	0.34 / 0.34	8.27% / 8.20%
	DVH M2	**0.27 / 0.28**	**6.72% / 6.76%**

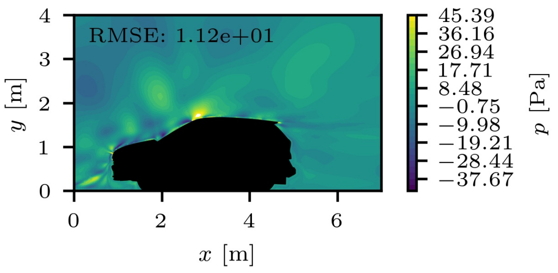
Pressure field predictions near an unseen vehicle are shown in Figure 4.11 for DV-MLP and DVH M2, with predictions for all field variables over the full domain shown in the Appendix, Section B.1. The error contour colorbars are limited to $\pm 4\times$RMSE centered on the average case error in order to see more fine-grain detail. Infrequent, comparably-large errors obscure these details in many instances by skewing the colorbar, and points where the error is outside of this band are left white. The pressure field predictions match the ground truth well, with the main features captured including the high-pressure in front, the low-pressure due to flow acceleration on the roof, and the slowly-recovering pressure in the wake seen in the full-domain plots. Some distortion of the contour lines is present in the predictions. The largest errors are seen near the vehicle surface, near the ground in front, and at locations along the free-shear layer and in the wake.



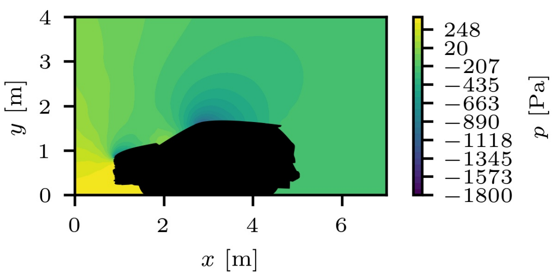
(a) Ground truth pressure field.



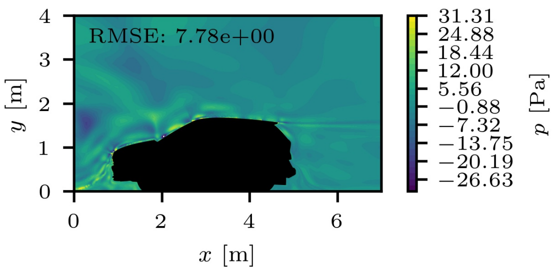
(b) DV-MLP prediction, validation group.



(c) DV-MLP error field.



(d) DVH M2 prediction, validation group.



(e) DVH error field.

Figure 4.11: Validation-group instance (a) ground-truth pressure field, (b) DV-MLP prediction, (c) DV-MLP error, (d) DVH prediction, and (e) DV-MLP error. Error colorbars are limited to $\pm 4 \times RMSE$ centered on the average error for the instance.

4.3.4 Multiple Speeds and Generalization: Low-Data Regime

The goal in situations such as surrogate-based design optimization is to obtain an accurate and generalizable surrogate using the least amount of training data and resources. The effect of the training dataset size on the generalization and convergence properties is explored by varying the number of available training cases. Two vehicle speeds of 90 and 130 kph are used, giving a total of 248 available solution instances. DV-MLP and DVH are compared while the number of training cases is varied from 5 to 199 instances, corresponding to training fractions ranging from 0.02 to 0.8. Training method 2 batch-by-case is used to train DVH models given that similar or better accuracy may be obtained as compared to fully-mixed training in a fraction of the time, as demonstrated for a single vehicle speed in Section 4.3.3. The dataset and training options are given in Table 4.11. The number of epochs remains fixed, resulting in varying learning rate schedule entries as the number of training cases is adjusted. Given a training fraction f_{train}, the number of training cases is chosen to be $\text{ceil}(f_{\text{train}} n_{\text{cases}})$, then the number of steps is given as

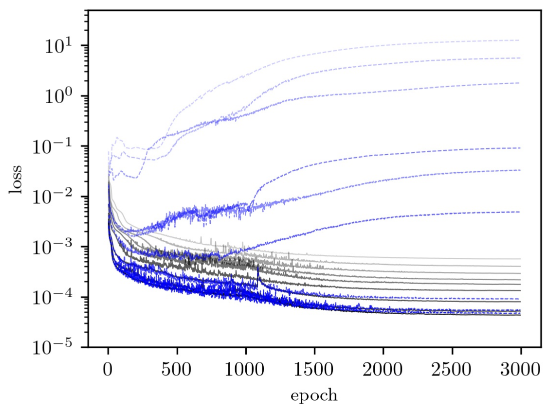
$$s = n_{\text{epochs}} \times \text{ceil}(f_{\text{train}} n_{\text{cases}}) \times n_{\text{batches per case}}. \tag{4.24}$$

The number of epochs are given in Table 4.11 in parentheses and $n_{\text{batches per case}} = 2$.

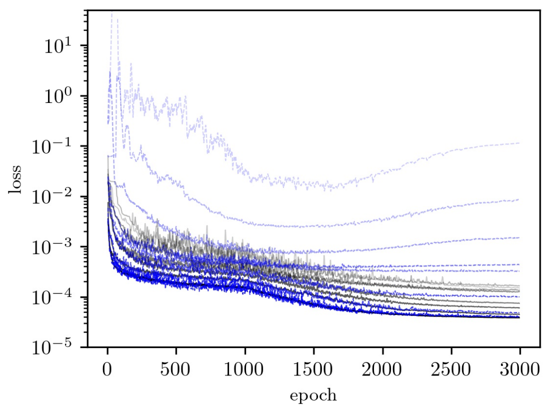
Table 4.11: Dataset and training options, where s_t is the total number of optimizer steps, and values in parentheses are dataset-iterations or epochs.

Dataset Options		**Learning Rate Schedule**	
vehicle speeds	90, 130 kph	α_0	1×10^{-3}
spatial batch size	54,000	r	0.1
# cases total	248	s_t	Equation 4.24 (3000)
case batch size	1	s_c	Equation 4.24 (1000)
-	-	s_d	Equation 4.24 (1000)

In the low-data regime, it is evident that neither model demonstrates strong generalization capabilities, leading to a substantial disparity between training and validation losses. The effect is more pronounced for DV-MLP, as illustrated by the training curves in Figure 4.12. In the figure the solid lines represent the training loss, the dashed lines the validation loss, and darker lines correspond to a greater number of training cases. It is worth noting that the training loss reaching a plateau at a higher value with less data can be attributed, in part, to the fact that the number of epochs remains consistent rather than the number of optimizer updates. This should be kept in mind while interpreting the results.



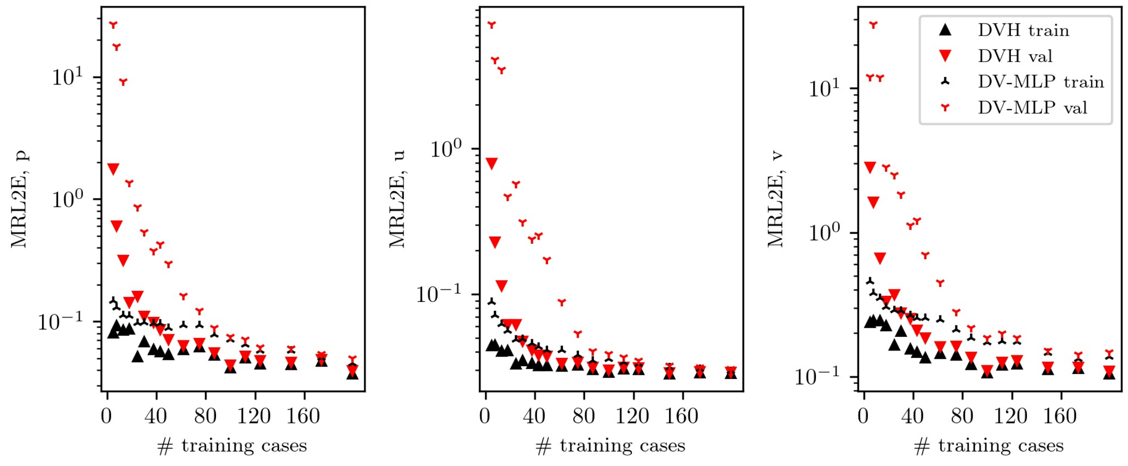
(a) DV-MLP



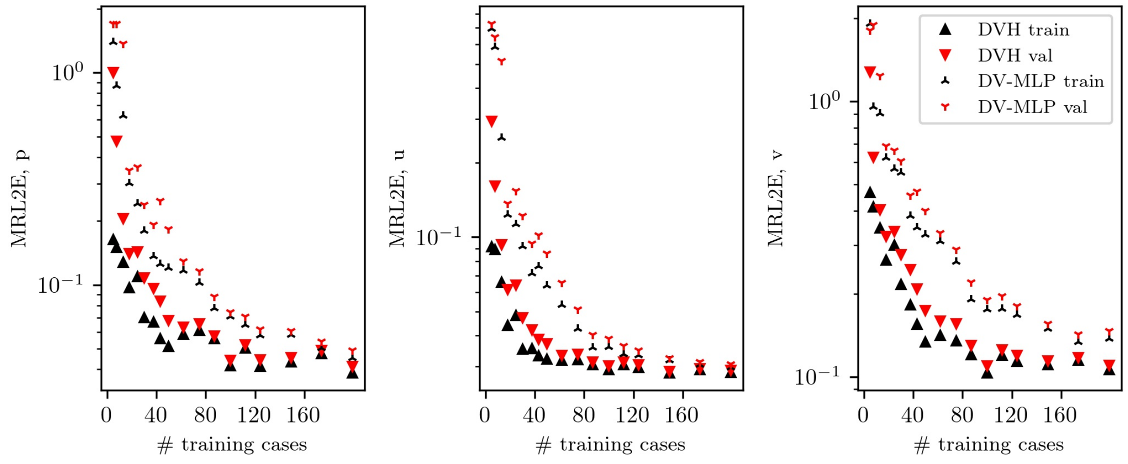
(b) DVH

Figure 4.12: Training (solid) and validation (dashed) losses during training as the amount of training data is varied from 5 to 199 cases, where darker lines correspond to more data, for (a) DV-MLP and (b) DVH. The curves have been smoothed using a moving average with a window length of 3 epochs for DV-MLP and 5 epochs for DVH.

During training the best weights based on the validation loss are saved along with the final weights. Figure 4.13a shows the MRL2E versus the number of training cases using the final weights for each flow quantity. In the very low-data regime a large gap between training and validation losses is seen for both models, with DV-MLP exhibiting poorer performance. As the number of cases is increased this gap is closed more quickly for DVH than DV-MLP. When 199 cases are used the models perform similarly, with a slight advantage observed for DVH. Similar plots are generated using the validation best weights, as is shown in Figure 4.13b. In this scenario there is a smaller gap between the training and validation losses for each model. However, a persistent gap between DVH and DV-MLP remains across all training fractions considered, most notable for the y velocity v.



(a)



(b)

Figure 4.13: Comparing trends in predictive error using mean-relative-L2-error (MRL2E), with (a) the final weights, and (b) the best weights per validation loss seen during training. The y-axis is not multiplied by 100%, therefore 10^{-1} corresponds to 10% mean error in the state variable.

Dimensional error metrics computed using the validation-best weights with 199 training instances, corresponding to the rightmost point in Figure 4.13b, are reported for both DV-MLP and DVH in Table 4.12. This shows that DVH performs best across the board, as expected. The RMSEs with two vehicle speeds are larger than those reported for a single speed in Table 4.9, and this is due in part to the 130 kph solutions having higher pressures and velocities than at 90 kph. To dig into this, the nondimensional and dimensional error metrics are broken out by vehicle speed instead of training group in for DVH in Table 4.13. This reveals that the non-dimensional errors compare similarly for each vehicle speed, but with 130 kph errors being slightly larger. The dimensional pressure errors for 90 kph lie between the training and validation errors for DVH M2 at a single speed, while the velocity component errors are slightly larger.

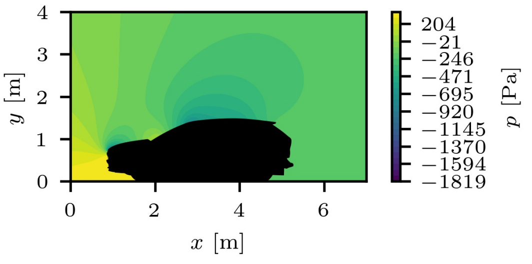
Table 4.12: Summary of training and validation error metrics for vehicle speeds of 90 and 130 kph with a training fraction of 0.80.

$\hat{\mathbf{q}}_i$	**Network Type**	**RMSE (train / val)**	**MRL2E (train / val)**
p [Pa]	DV-MLP	16.4 / 18.3	4.37% / 4.69%
	DVH M2	**12.9 / 14.8**	**3.43% / 3.72%**
u [m/s]	DV-MLP	0.78 / 0.83	2.75% / 2.87%
	DVH M2	**0.61 / 0.65**	**2.14% / 2.23%**
v [m/s]	DV-MLP	0.59 / 0.63	11.7% / 12.2%
	DVH M2	**0.46 / 0.49**	**9.06% / 9.40%**

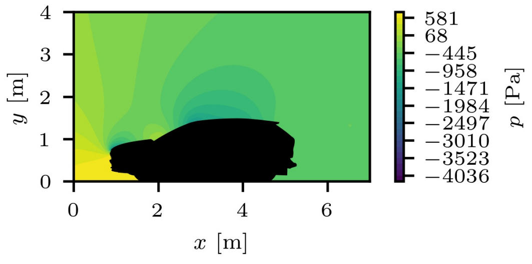
Table 4.13: Comparing DVH non-dimensional and dimensional error metrics computed for each vehicle speed separately with a training fraction of 0.80.

$\hat{\mathbf{q}}_i$	**Error Type**	**RMSE**		**MRL2E**	
		90 kph	130 kph	90 kph	130 kph
p [-]/[Pa]	Nondimensional	$2.14 \cdot 10^{-2}$	$2.30 \cdot 10^{-2}$	3.37%	3.61%
	Dimensional	8.2	18.4	3.37%	3.61%
u [-]/[m/s]	Nondimensional	$1.99 \cdot 10^{-2}$	$2.02 \cdot 10^{-2}$	2.15%	2.18%
	Dimensional	0.50	0.73	2.15%	2.18%
v [-]/[m/s]	Nondimensional	$1.48 \cdot 10^{-2}$	$1.54 \cdot 10^{-2}$	9.04%	9.22%
	Dimensional	0.37	0.56	9.04%	9.22%

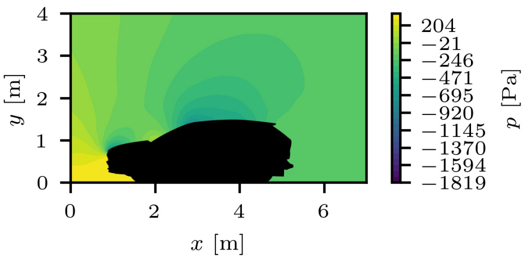
DVH pressure field predictions for a single vehicle shape at both 90 and 130 kph are shown in Figure 4.14, where the 90 kph case is from the training set while the 130 kph case is from the validation set. Good agreement between ground truth and prediction is seen at both speeds. Additional plots for velocities u and v are shown in the Appendix Section B.1.2 with similar good agreement.



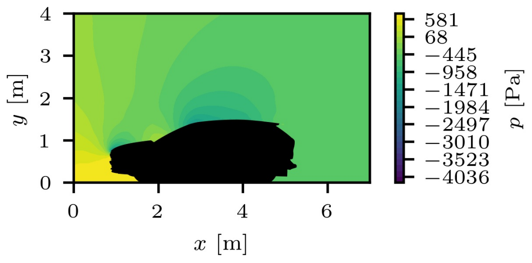
(a) 90 kph, ground truth.



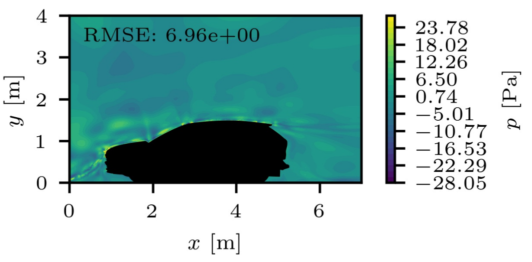
(b) 130 kph, ground truth.



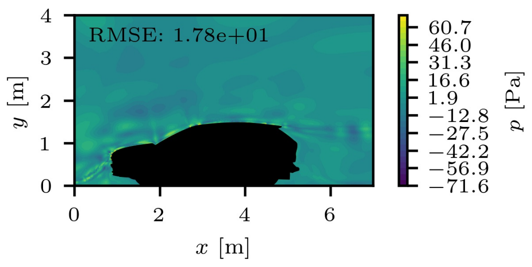
(c) 90 kph, DVH prediction, training set.



(d) 130 kph, DVH prediction, validation set.



(e) 90 kph DVH error, training set.



(f) 130 kph DVH error, validation set.

Figure 4.14: Pressure field ground truth, DVH prediction, and errors at 90 and 130 kph for the same vehicle shape.

4.4 Numerical Experiments II: Effect of Random Fourier Features

Spectral bias is a noted difficulty in training neural networks, where low-frequency signal content is learned more quickly and readily than high-frequency content. Consequently, the networks exhibit a bias towards low-frequency signal content [186]. This issue may be addressed though use of random Fourier features [187] or positional encoding techniques

[161]. With these methods the input coordinates are processed by sinusoidal terms of varying frequency before being fed into the MLP. Given network inputs $\mathbf{x}$, a Fourier-feature mapping is written as

$$\gamma(\mathbf{x}) = \Big[a_1 \cos(2\pi \mathbf{b}_1^T \mathbf{x}), a_1 \sin(2\pi \mathbf{b}_1^T \mathbf{x}), \ldots, a_m \cos(2\pi \mathbf{b}_m^T \mathbf{x}), a_m \sin(2\pi \mathbf{b}_m^T \mathbf{x})\Big]^T, \quad (4.25)$$

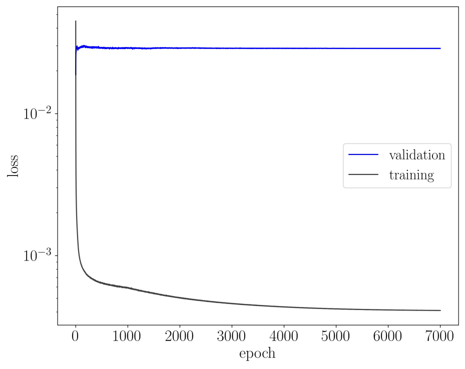
where coefficients a_i, frequency-vectors $\mathbf{b}_i$, and their number m are parameters of the mapping. It has been shown that the simple strategy of setting $a_i = 1$ and drawing each $\mathbf{b}_i$ randomly from a sampling distribution is an effective strategy in practice, with the width σ of the sampling distribution having much greater importance than the distribution shape [187]. This width has a major impact on the convergence and generalization properties of the network, with underfitting observed for σ "too small" and overfitting observed for σ "too large." Positional-encoding strategies are generally a special case of Fourier features [188] where the frequencies are specified as a geometric progression and applied along each input dimension x_i separately [161, 189]. This may be written as

$$\gamma(x_i) = \Big[\cos(2^0 \pi x_i), \sin(2^0 \pi x_i), \ldots, \cos(2^{m-1} \pi x_i), \sin(2^{m-1} \pi x_i)\Big], \quad (4.26)$$

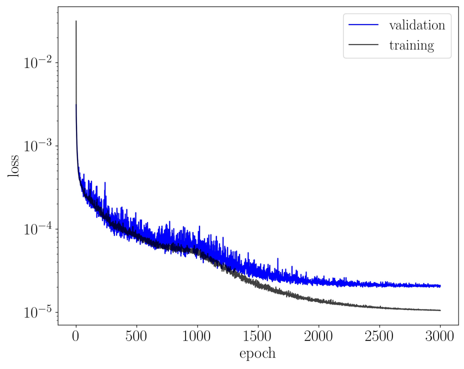
where m may be set independently along each input axis. Another similar alternative includes the specific weight-initialization of sin-activated SIREN networks [168]. A downside to random Fourier features is that the width σ and the number of features m must generally be found by trial and error or hyperparameter tuning. In all experiments a zero-mean truncated isotropic Gaussian sampling distribution is used, $\mathbf{b}_i \sim \mathcal{N}(\mathbf{0}, \sigma \mathbf{I})$, with $\sigma = 3$ for all results here. Samples greater than 2σ from the mean are discarded. The number of features is selected to be $m = 256$, the same number of features chosen in Ref. [187], despite the smaller hidden dimension of $H = 50$ used in experiments here.

4.4.1 Model Architectures and Training Options

The options and architectures of Section 4.3.2 are used again here but with the insertion of a random Fourier feature layer after the input layer of the main network. Note that the Fourier layer does not contain any trainable parameters, the sampled $\mathbf{b}_i$ are fixed.Applying this to DVH is straightforward and the same set of random Fourier features are used for all main networks. However, Fourier features are not used in the hypernetwork. For DV-MLP, Fourier features are applied only to spatial inputs $\mathbf{x}'$. The resulting output $\gamma(\mathbf{x}')$ is then concatenated with $\boldsymbol{\mu}$ and passed into the MLP. Naively applying Fourier features to all DV-MLP inputs $\mathbf{x}'$ and $\boldsymbol{\mu}$ results in poor convergence and generalization, as shown in Figure 4.15a. When applied to only the spatial inputs $\mathbf{x}'$ the models converge readily as shown in 4.15b. The network architecture and dataset are identical between the two models other than the difference in which inputs are processed by Fourier features. Thus, only the spatial input quantities $\mathbf{x}'$ are passed through the random Fourier layer in the following experiments.



(a)



(b)

Figure 4.15: Training curves for DV-MLP models, where the Fourier features are: (a) applied to all inputs $\mathbf{x}'$ and $\boldsymbol{\mu}$, and (b) applied to only spatial inputs $\mathbf{x}'$.

Insertion of a random Fourier layer with $m = 256$ results in models with more weights as compared to a model without a random Fourier layer. This is because $\dim(\gamma(\mathbf{x}')) = 2m \gg \dim(\mathbf{x}')$, leading to many more parameters in the first layer of the MLP. Models are profiled with the addition of the Fourier-feature layer as shown in Table 4.14, with the number of trainable parameters given in Table 4.15 as compared against Table 4.6. Double precision is not considered here due to the much greater training time. Additionally only DVH training method 2 batch-by-case is considered due to the lessened training time and resources.

Table 4.14: Profiling DVH models with varying hypernetwork-final-hidden-dimension H_L using training method 2, batch-by-case, with mixed precision.

Type	Precision	Δt step [ms]	Δt compute [ms]	Max. Mem. [GB]
DV-MLP FF	single	5.0 (0.1)	4.2	0.85
DV-MLP FF	mixed	4.1 (0.2)	2.7	0.71
DVH M2 FF	single	5.4 (0.1)	4.5	0.55
DVH M2 FF	mixed	5.1 (0.5)	3.1	0.41

Table 4.15: Number of trainable parameters for DV-MLP and DVH models for all Fourier-feature results of Section 4.4, where $H = H_L = 50$, $L_m = L_H = 5$.

Method	# Trainable Weights	
	Single Speed	Multiple Speeds
DV-MLP	36,403	36,453
DVH	1,846,803	1,846,853

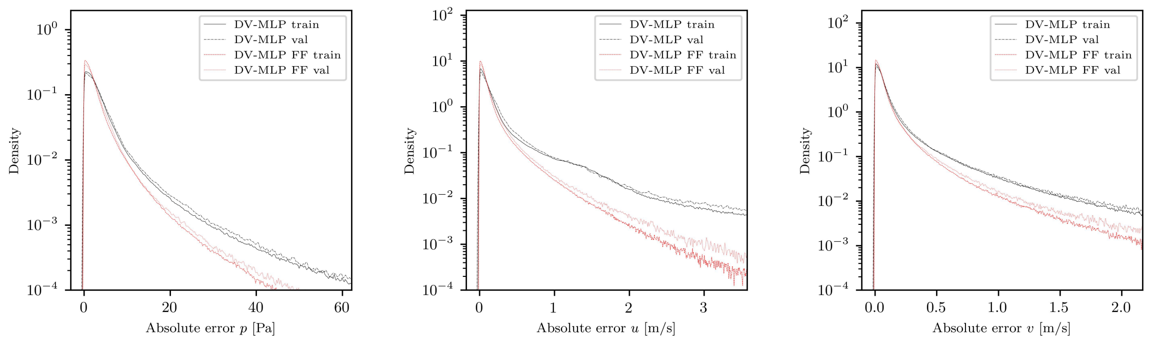
4.4.2 Single Vehicle Speed

DV-MLP and DVH using random Fourier features are first trained on a single vehicle speed using the same dataset and training options as given in Table 4.8. Error metrics are reported in Table 4.16, with the corresponding percentage improvement in RMSE compared to the models without Fourier features indicated in parentheses. With the use of Fourier features, DVH and DV-MLP exhibit similar performance and DV-MLP is now best for several entries. For DV-MLP this is a rather large improvement as the errors are roughly halved for a 35% increase in training step time (from 3.7 ms to 5.0 ms) when using single precision. DVH shows large but less significant improvements, with the training step time increased by 23% (from 4.4 to 5.4 ms) when using single precision.

Table 4.16: Summary of training and validation error metrics at a vehicle speed of 90 kph for models using a Fourier-feature layer.

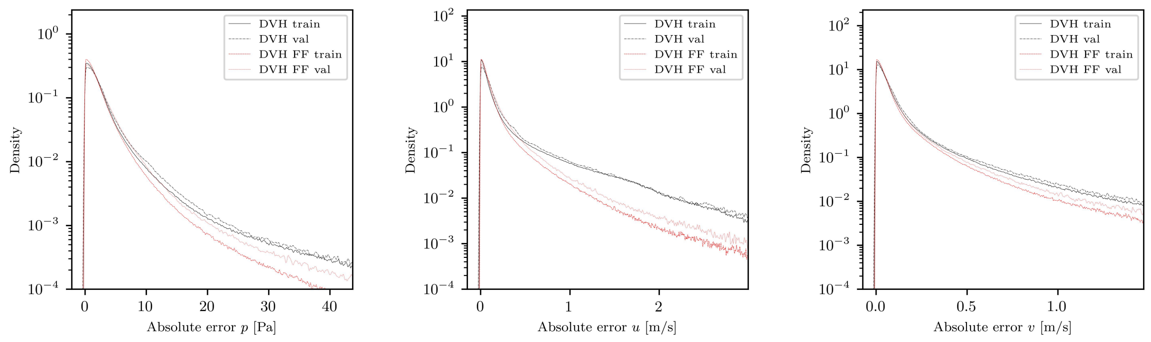
$\hat{\mathbf{q}}_i$	**Network Type**	**RMSE**		**MRL2E**	
		Train	Val	Train	Val
p [Pa]	DV-MLP FF	5.55 (51.7%)	7.37 (41.3%)	2.31%	2.96%
	DVH M2 FF	**4.53** (44.1%)	**7.26** (26.6%)	**1.88%**	**2.89%**
u [m/s]	DV-MLP FF	0.243 (63.3%)	**0.30** (59.5%)	1.05%	**1.29%**
	DVH M2 FF	**0.236** (57.8%)	0.32 (47.3%)	**1.02%**	1.37%
v [m/s]	DV-MLP FF	**0.20** (49.6%)	**0.27** (44.6%)	**5.04%**	**6.58%**
	DVH M2 FF	0.22 (21.1%)	0.31 (19.5%)	5.32%	7.40%

The pointwise absolute error probability distributions with and without Fourier features are visualized using kernel density estimates, computed using the `FFTKDE` function of the python library KDEpy [190] using the Silverman method for kernel bandwidth selection. Other implementations were found to be very slow by comparison due to the large number of points: ∼11.1 million training and ∼2.8 million validation mesh points per vehicle speed. These are shown in Figure 4.16 for DV-MLP and Figure 4.17 for DVH. In all instances the absolute error distributions are narrowed when using Fourier features, meaning smaller errors are more prevalent.



(a) Pressure, single speed **(b)** x-velocity **(c)** y-velocity.

Figure 4.16: DV-MLP single speed, pointwise absolute error probability distributions with and without random Fourier features, computed using Gaussian kernel density estimates.



(a) Pressure, single speed **(b)** x-velocity **(c)** y-velocity.

Figure 4.17: DVH single speed, pointwise absolute error probability distributions with and without random Fourier features, computed using Gaussian kernel density estimates.

The drag coefficient is commonly used in assessing aerodynamic designs and is given by

$$C_D = \frac{F_D}{\frac{1}{2}\rho_\infty u_\infty^2 A}, \tag{4.27}$$

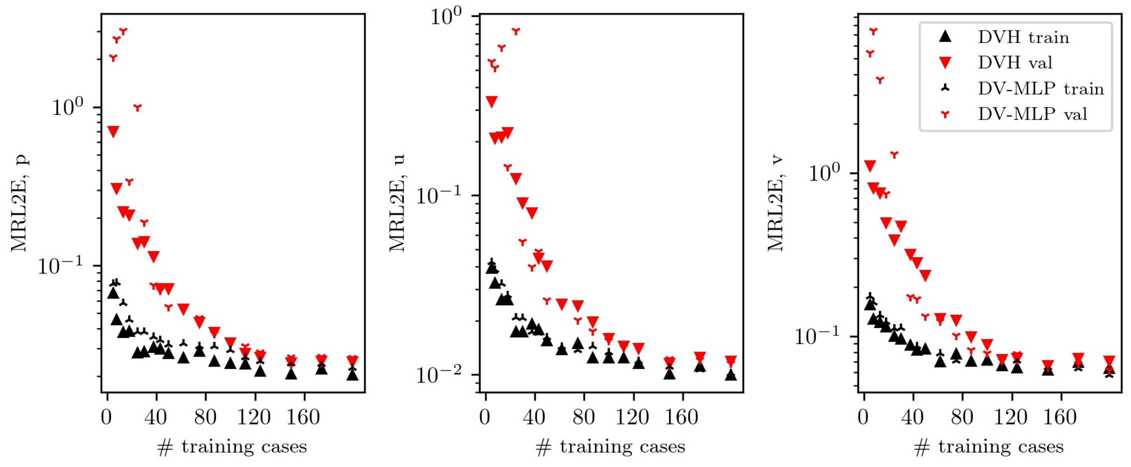
where F_D is the drag force, subscript ∞ corresponds to freestream conditions, and A is the frontal area. The MRL2E in computing the pressure drag coefficient is shown in Table 4.17 for both the Fourier models of this section and the non-Fourier models of Section 4.3.3. DVH M2 FF shows the smallest overall percent error over both the training and validation groups. In general the Fourier models perform better than the non-Fourier models, with the lone exception of the DVH M2 training group errors.

Table 4.17: MRL2E (equivalent to mean-absolute-percent error) in predicting the pressure drag coefficient over the training and validation groups for non-Fourier and Fourier-based models for a single vehicle speed of 90 kph.

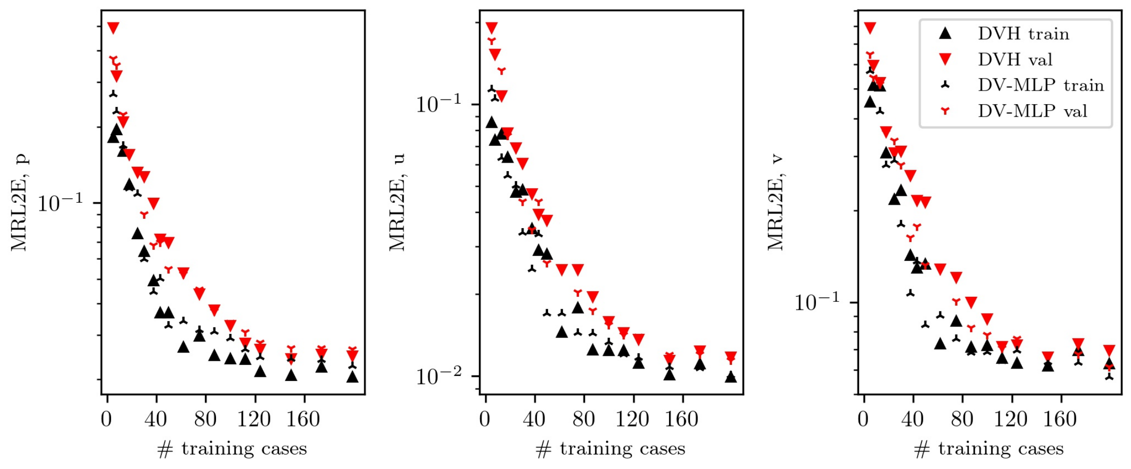
Network Type	**Train**	**Val**
DV-MLP	2.66%	2.45%
DVH M1	2.28%	2.72%
DVH M2	1.26%	2.39%
DV-MLP FF	1.43%	1.50%
DVH M2 FF	**0.67%**	**1.30%**

4.4.3 Multiple Speeds and Generalization: Low-Data Regime

The impact of Fourier features on generalization in the low-data regime with multiple vehicle speeds is studied here in an analogous fashion to Section 4.3.4. Dataset and training options of Table 4.11 apply. Figure 4.18ashows the training and validation MRL2E for each output quantity as a the number of training instances is varied. As before, in the very-low data regime there is a large gap between training and validation losses, with DV-MLP showing a greater disparity. However, as the number of training instances is increased to around 40 the difference between DV-MLP and DVH is greatly reduced, and thereafter their performance is very similar to one another. Figure 4.18b shows the corresponding plots using the best weights per validation loss. Without Fourier features there was a persistent gap between DVH and DV-MLP, but when Fourier features are used this gap is more or less eliminated.



(a)



(b)

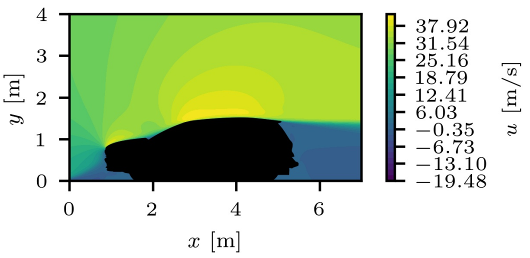
Figure 4.18: Comparing trends in predictive error using mean-relative-L2-error (MRL2E), with (a) the final weights, and (b) the best weights per validation loss seen during training. The y-axis is not multiplied by 100%, therefore 10^{-1} corresponds to 10% mean error in the state variable.

Dimensional error metrics computed using the validation-best weights with 199 training instances, corresponding to the rightmost point in Figure 4.18b, are reported for both DV-MLP and DVH in Table 4.18. The percentage improvement in RMSE as compared to a model without Fourier features is given in parentheses. As with a single vehicle speed, substantial improvements are seen for both DV-MLP and DVH, with the effect larger for DV-MLP. Similarly to the single-speed scenario, DV-MLP now performs best for several entries as well.

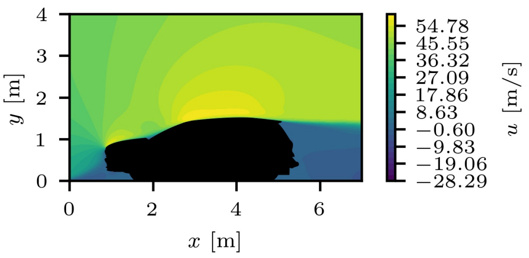
Table 4.18: Summary of training and validation error metrics for vehicle speeds of 90 and 130 kph with a training fraction of 0.80, including use of Fourier features. The percentage improvement when using Fourier features is given in parentheses.

$\hat{\mathbf{q}}_i$	**Network Type**	**RMSE**		**MRL2E**	
		Train	Val	Train	Val
p [Pa]	DV-MLP FF	8.4 (48.7%)	10.2 (44.5%)	2.26%	2.63%
	DVH M2 FF	**7.6** (40.9%)	**9.7** (34.4%)	**2.05%**	**2.48%**
u [m/s]	DV-MLP FF	0.29 (63.2%)	**0.33** (59.7%)	1.01%	**1.14%**
	DVH M2 FF	**0.28** (53.4%)	0.34 (47.3%)	**0.99%**	1.17%
v [m/s]	DV-MLP FF	**0.29** (51.2%)	**0.31** (49.9%)	**5.70%**	**6.13%**
	DVH M2 FF	0.32 (30.7%)	0.36 (26.5%)	6.28%	6.95%

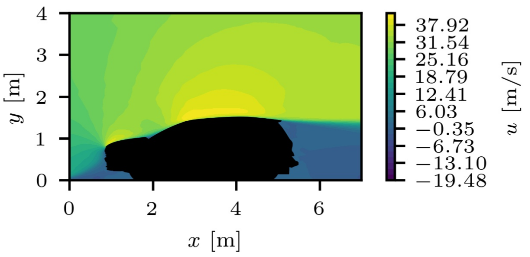
DVH predictions and errors of the x-velocity field at both speeds are shown in Figure 4.19, where neither speed is included in the training dataset. The predicted fields match the ground truth well and capture the dominant flow features including the small recirculating regions in front of the vehicle, acceleration and flow turning over the roof, and a decaying free-shear layer in the wake. Similar plots for the pressure and y-velocity predictions are given in the Appendix, Section B.2.



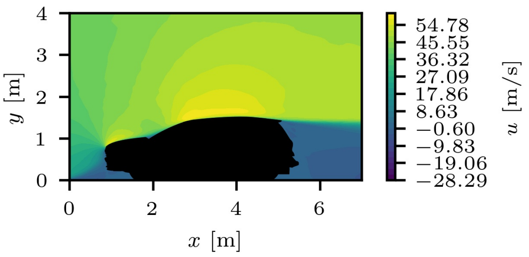
(a) 90 kph, ground truth.



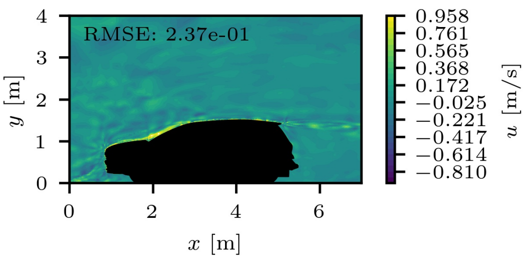
(b) 130 kph, ground truth.



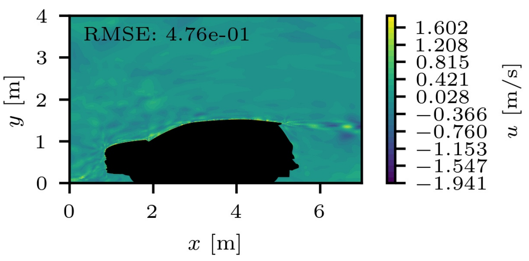
(c) 90 kph, DVH prediction, validation set.



(d) 130 kph, DVH prediction, validation set.



(e) 90 kph DVH error, validation set.



(f) 130 kph DVH error, validation set.

Figure 4.19: x-velocity ground truth, DVH prediction, and errors at 90 and 130 kph for the same vehicle shape, where neither instance was included in the training set.

For further comparison, vertical line probes are placed near the vehicle, with one in front, one through the vehicle's highest point, and two in the wake. The probe in front of the vehicle is offset by 1 m, while those in the wake are offset by 1 m and 3 m. The ground truth and DVH predictions with and without Fourier features are interpolated from mesh points to the line probe locations using the `griddata` function from the `scipy.interpolate` library, with the results shown in Figure 4.20. Generally, the line probe predictions match the ground truth well, though some oscillation is present in the predictions. This is most prevalent for the pressure probes in the vehicle wake, though the effect is more pronounced due to the x-axis limits.

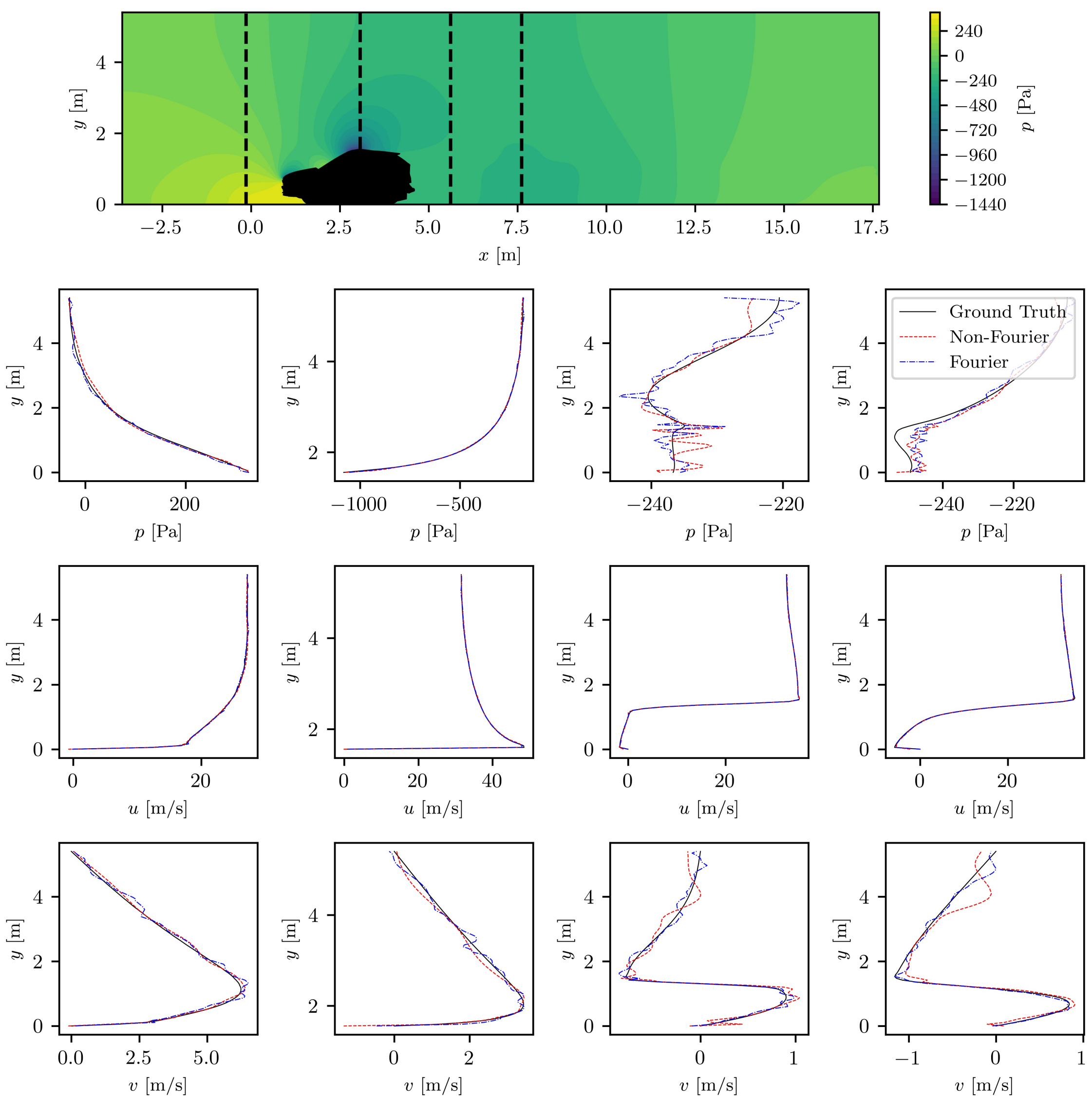
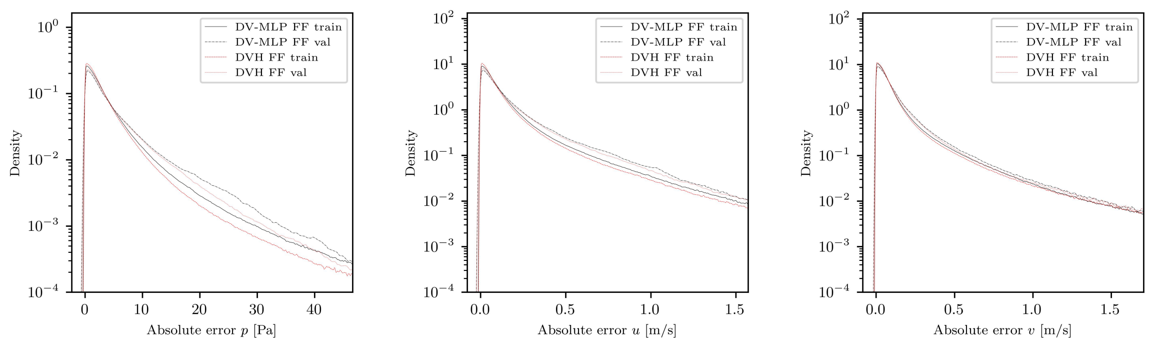


Figure 4.20: Line probes comparing baseline and Fourier feature DVH predictions for an unseen case at 90 kph.

Pointwise absolute error probability distributions for DV-MLP and DVH using Fourier features are visualized using kernel density estimates, again computed using the FFTKDE function of the python library KDEpy [190] using the Silverman method [191] for kernel bandwidth selection. The distributions have similar shape for both network types, showing a peak and gradual trailing off as the errors increase.



(a) Pressure **(b)** x-velocity **(c)** y-velocity.

Figure 4.21: Comparing DV-MLP and DVH pointwise absolute error probability distributions using random Fourier features, computed using Gaussian kernel density estimates.

The MRL2E in predicting the pressure drag coefficient using the Fourier models of this section and the non-Fourier models of Section 4.3.4 are shown in Table 4.19. In general the Fourier models slightly outperform the non-Fourier models, with the lone exception being the validation set using DVH without Fourier features.

Table 4.19: MRL2E (equivalent to mean-absolute-percent error) in predicting the pressure drag coefficient over the training and validation groups for non-Fourier and Fourier-based models for multiple vehicle speeds of 90 and 130 kph.

Network Type	**Train**	**Val**
DV-MLP	2.65%	3.30%
DVH	1.20%	**1.54%**
DV-MLP FF	1.90%	2.29%
DVH FF	**1.05%**	1.58%

Although the predicted fields agree well visually with the ground truth solutions, and that Fourier features improved the predictions, errors in the predicted fields may or may not be small enough depending on details of the desired use case. If the goal is to use such a model for near-real-time flow-field visualization to draw qualitative comparisons, then this level of accuracy may be sufficient. However, if the goal were to use the models to drive surrogate-based optimization to improve an existing design then this accuracy may not be sufficient. For example, if one were after improvements on the order of 1%, then the errors would likely be too large. It was not discussed previously, but there are shortcomings in the vehicle dataset which may be preventing better performance. Notably, the data was provided as a point cloud, without connectivity information between cells. This includes the points which define the vehicle surfaces, which were provided out of order. The vehicle shapes were sorted using a nearest-neighbor approach, but the fine details in the grille area leads to several points being left unconnected from the rest of the vehicle, which are dropped from the shape. This sorting is adequate to aid in creation of the contour plots, with the sorted shape provided as a mask. However, the uncertainty in ordering of the surface points has a negative and uncontrolled effect on computation of the signed distance function, and in turn on model convergence and predictive capability.

4.5 Ahmed Body: 3D Vehicle Aerodynamics

The Ahmed body is a simplified vehicle shape devised in the mid-80's to study flow around passenger vehicles in order to investigate the main sources of efficiency-robbing drag [192]. The body consists of a rounded fore-body, a middle section of constant rectangular cross-section, and a blunt or slanted rear-end, with the effect of wheel-rotation ignored by placing the body on legs or suspending it from strings. The model is reconfigurable so that the slant angle at the end may be changed from 0 to 40 degrees, and wind-tunnel studies showed that most of the drag comes from pressure drag, generated largely at the rear end. The model dimensions and details on the reconfigurable rear end are shown in Figure 4.22.

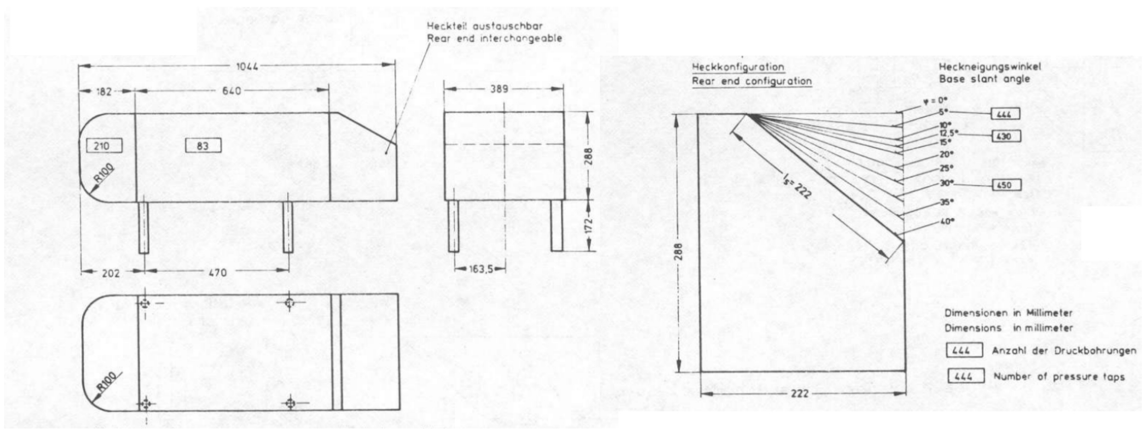
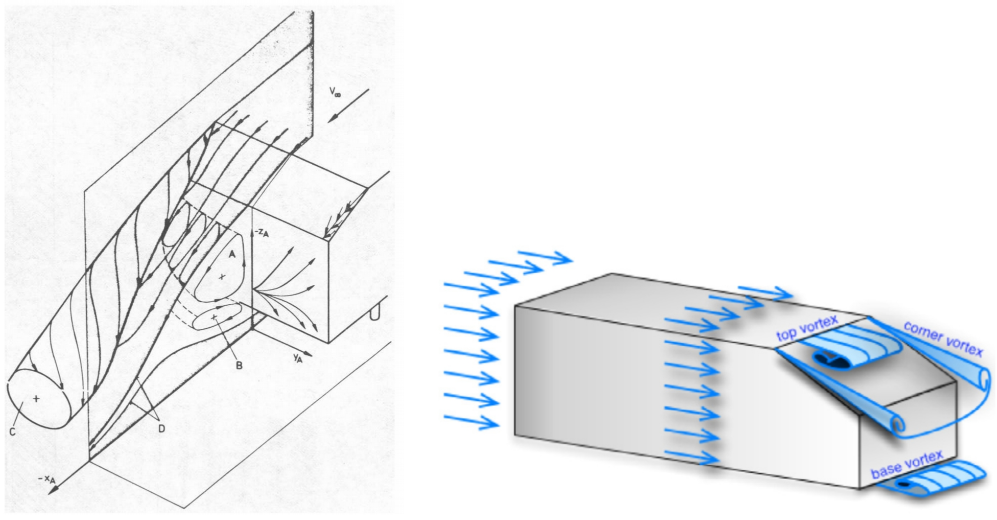


Figure 4.22: Geometry and dimensions of the Ahmed body in millimeters, Figures taken from reference [192].

Sometimes this general shape is referred to as the square-back body, while the exact geometric details may differ the overall shape is very similar [193]. This has proven to be an impactful concept, inspiring many follow on studies including experiments and computations [194, 195, 196]. Detailed wake measurements [192, 197] helped make the Ahmed body a popular simulation benchmark and test-case [198].

The flow field at the rear end changes greatly depending upon the slant angle. Some of the main features of the flow are shown conceptually in Figure 4.23 and include a pair of counter-rotating"horeseshoe" vortices emanating from the top corners.



(a) **(b)**

Figure 4.23: Conceptual representations of prominent flow-field vortices at the rear end of the Ahmed body, where (a) is taken from [192], and (b) is taken from [196].

The strength of the corner vortices varies with slant angle and they interact with the other wake vortices in a complex manner. For slant angles less than 15°the flow remains fully attached along the slant and two vortices remain attached to the lower, vertical flat surface. At 25°the flow begins to separate at the top of the slant, but the corner vortices are strong enough to cause flow reattachment farther along the slant, resulting in a separation bubble. The size of this bubble grows with increasing slant angle while the corner vortices weaken until a slant angle of 35°where the flow becomes fully separated. While the separation bubble grows the vortices at the rear end begin to separate with the top vortex becoming more prominent and displaced upward. The movement of these rear-end vortices with slant angle is shown in Figure 4.24, taken from reference [199], which are time averaged streamlines $\langle\Psi\rangle$ obtained using particle image velocimetry (PIV) along the vehicle centerline. As these complex interactions occur, the drag coefficient increases greatly with slant angle until the flow becomes fully-separated, where it then decreases substantially. The drag coefficient versus slant angle is shown in Figure 4.26a.

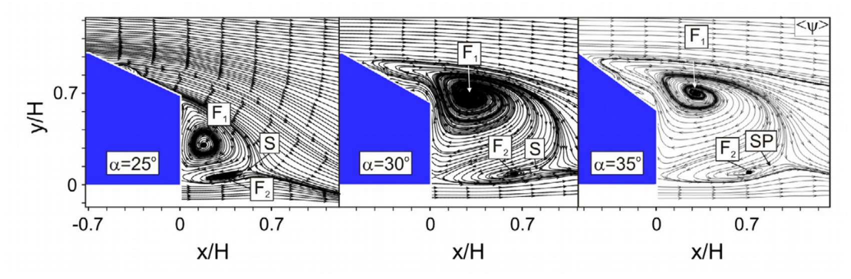


Figure 4.24: Time-averaged streamlines at the Ahmed body rear-end obtained using PIV, taken from [199].

4.5.1 Generation of CFD Solutions

The Ahmed-body flow field was simulated by solving the steady, 3D incompressible RANS equations using the k-ω SST turbulence model [200], using the OpenFOAM application `simpleFoam`, an implementation of the SIMPLE algorithm [58, 59, 60]. The rear slant angle was varied from 0 to 45 degrees in 5 degree increments. Solutions were obtained at speeds of 40 m/s and 60 m/s, corresponding to highway-relevant Reynolds numbers based on vehicle length of 2.78×10^6 and 4.18×10^6 respectively, with 60 m/s matching the original experiments [192]. A centerline symmetry plane was used to reduce computational costs, and the legs were removed from the geometry to ease meshing. This may also be justified since the aerodynamic forces from the legs are tared out in experimental works.

A public repository has been created containing python and bash scripts which were used to generate the solutions at 60 m/s, located at `https://github.com/jamesduv/ahmedBodyParametric_Public`. Light modifications are made to run at 40 m/s. The repository depends upon two environment variables, `$AHMED_REPO_PUB` which points to the repository and `$AHMED_SLANT_PATH` where each case directory will be placed. The workflow descriptions will reference files from this repository. Included in the repository is a directory `case_setup` which includes the usual OpenFOAM directories `0` for initial and boundary conditions, `constant` for the turbulence and transport properties, and `system` containing files:

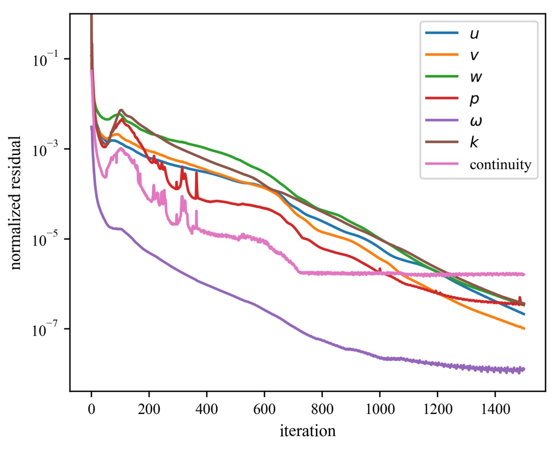
- `controlDict`: high-level settings for running `simpleFoam`, such as start and stop times and write intervals
- `controlDict.postProcess`: settings for post-processing the solutions, to compute forces, force coefficients, y^+ values, and wall shear stresses
- `createPatchDict`: creates mesh patches from the named .stl regions of the geometry file
- `decomposeParDict`: mesh decomposition settings for running in parallel
- `fvSchemes`: schemes settings (gradients, divergences, wall distance, etc..)
- `fvSolution`: solver and algorithm settings, including tolerances and relaxation factors
- `meshDict`: mesh settings, including refinement and boundary layer options.

An additional directory `slurm` contains anonymized job submission files. Use script `copy_case_setup.sh` with slant-angle argument to copy the case files to the case directory, located at `$AHMED_SLANT_PATH\slant_angle_$ANGLE.00` for integer slant angle `$ANGLE`. This is the `case_path` for a given simulation.

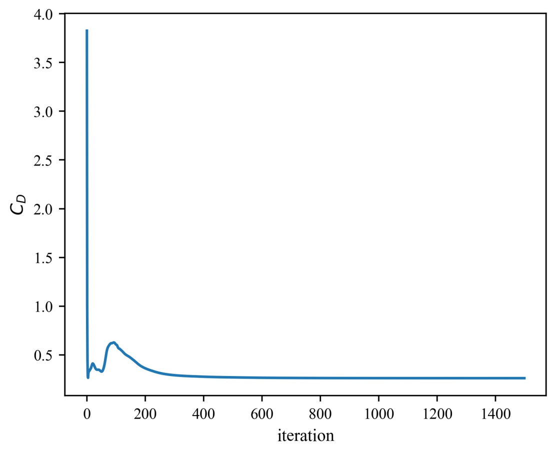
The body and domain geometry were generated as .stl files using the Gmsh [184] python API, see `generate_case_geometry_nolegs.py` which places the generated files in `case_path\geometry`. Meshes were generated using the OpenFOAM-included version of cfMesh which takes a single .stl file of the fluid domain as input. Thus the individual .stl files generated by Gmsh were merged and the patches named appropriately for later reference using `modify_stl_patch_merge.sh`. Meshes were generated using `generate_case_mesh.sh` or `case_path\slurm\run_mesh.sh`. The mesh is placed in `case_path\constant\polyMesh`. The meshes are unstructured with polyhedral elements, and the number of cells per mesh varies between 9.44 and 9.53 million. The resulting solutions have $y^+ < 5$, deemed acceptable for the purposes here.

The simulations were run in parallel with MPI using 4 nodes and 120 cores on the Great Lakes Slurm HPC Cluster at the University of Michigan. To set up the parallel computation a mesh domain decomposition is performed, using `case_path\slurm\run_decomp.sh`. The solution fields were initialized with potential flow solutions using `potentialFoam` by running `case_path\slurm\run_potentialFoam_parallel.sh`. Initializing with the potential flow solution increased solver stability and convergence and had low computational overhead. Then the solutions were generated using `simpleFoam` by running `case_path\slurm \run_simpleFoam_parallel.sh`. Per the slurm submission this creates a log file located at `case_path\*solve.log`. The residuals and drag coefficient for each iteration may be gathered and placed in .txt files located at `case_path\residuals` using scripts `gather_cd.sh` and `gather_residuals.sh`. Plots may then be generated using `plot_cd.py` and `plot_residuals.py`.

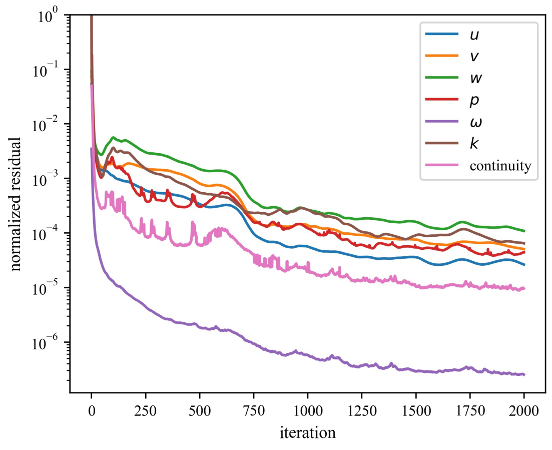
RANS simulations of the Ahmed body with varying slant angle are known to have difficulty fully capturing behavior of the flow over the rear slant for slant angles between roughly 20 and 40 degrees. From around 20-30 degrees the flow separates at the beginning of the slant before reattaching, and this continues until around 35 degrees where the flow fully separates. This can be observed in the residuals, where for slant angles less than that all residuals decrease by at least 10^{-5} relative to the uniform, pre-potential flow initial conditions. Note that OpenFOAM scales residuals so that if the initial field is uniform then the residual is 1, while the curves for some quantities start with a value lower than 1 due to the use of potential-flow initial conditions. All cases were run for an initial 1500 iterations, and within the transitory regime residuals are less convergent so the solutions were run for an additional 500 iterations. This behavior is shown in Figure 4.25, where the convergence of the residuals and drag coefficient are examined for slant angles of 0 and 30 degrees for a vehicle speed of 60 m/s.



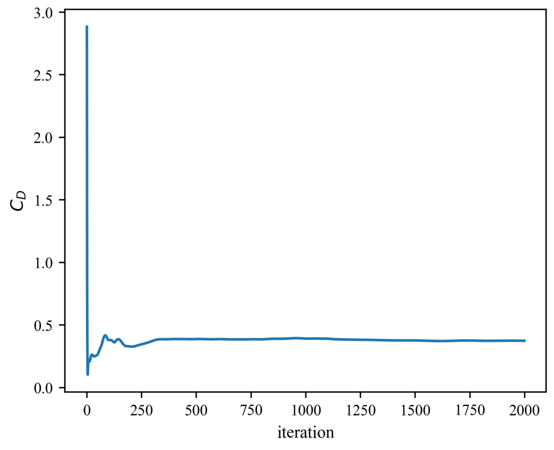
(a) Normalized residuals versus iteration, slant angle 0 degrees.



(b) Drag coefficient versus iteration, slant angle 0 degrees.



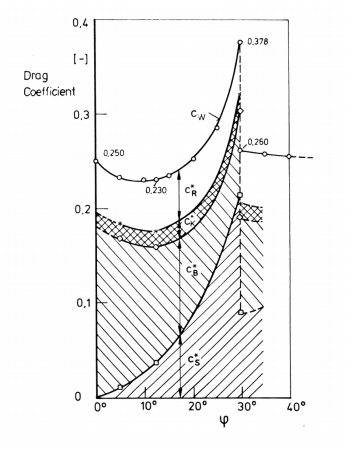
(c) Normalized residuals versus iteration, slant angle 30 degrees.



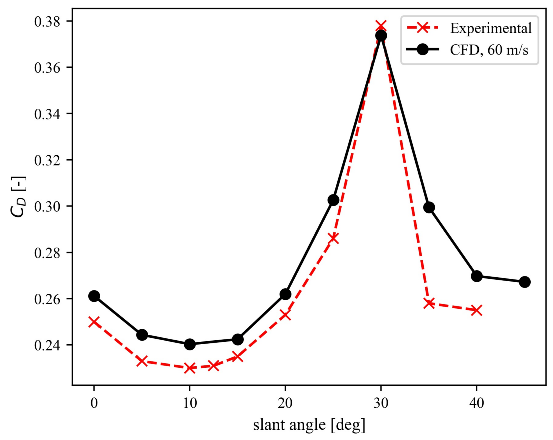
(d) Drag coefficient versus iteration, slant angle 30 degrees.

Figure 4.25: Comparing residual and drag coefficient convergence for slant angles of 0 and 30 degrees at a speed of 60 m/s. Residuals are normalized relative to uniform initial conditions.

The drag coefficient versus slant angle is reported in the original Ahmed body paper [192], but they are not fully tabulated with only a few values reported in a plot, shown in Figure 4.26a. The known values were used to extract values from the other slant angles by plotting horizontal lines and using the pixel values to linearly interpolate from known values. These extracted values are shown compared against those resulting from the CFD simulations in Figure 4.26b.



(a)



(b)

Figure 4.26: (a): Experimental C_D versus slant angle from ref. [192], not all numerical values provided in the figure or elsewhere in the paper. (b): Comparing the experimental values extracted from the image with the CFD results here, both at a speed of 60 m/s.

This shows that generally the drag coefficients are over-predicted by the CFD solutions, but the overall trends are largely captured. An exception is the 35 degree slant angle where the drag coefficient is grossly over-predicted. There are several reasons which may explain this and the general discrepancy between the simulated and experimental values. First, RANS models are known to be deficient in accurately predicting separation. The separation point in RANS simulations is often later than in experiments as the turbulent kinetic energy of the boundary layer is over-estimated, keeping the flow attached for too long. Secondly, RANS is time averaged, and the differences in drag coefficient may be a result of unsteady effects. And third, a symmetry plane about the centerline is used while in reality there would be an element of side-to-side vortex shedding which won't be captured, even if unsteady effects were accounted for at all. Despite these non-idealities, the dataset is used as-is and all errors are reported against the CFD simulations as the ground truth. This should be kept in mind when interpreting later results.

4.5.2 Data Processing and Preparation

OpenFoam is a cell-centered code, so the training dataset uses those values as opposed to nodal quantities. The wall-distance functions present within OpenFoam are used to compute the MDF, used as model input. A small utility was written to compute this distance from only the Ahmed body patch instead of all boundary patches and it is found with instructions for compilation in the repository in directory `ahmedPatchDist`. Non-dimensional inputs and flow variables are considered, computed analogously to Equations 4.16-4.20. The augmented spatial coordinates are defined as

$$\mathbf{x}' = \begin{bmatrix} \tilde{x} & \tilde{y} & \tilde{z} & \tilde{\phi}(x, y, z) \end{bmatrix}^T \in \mathbb{R}^4, \tag{4.28}$$

and the predicted state is

$$\mathbf{q} = \begin{bmatrix} \tilde{p} & \tilde{u} & \tilde{v} & \tilde{w} & k & \omega & \nu_t \end{bmatrix}^T \in \mathbb{R}^7, \tag{4.29}$$

where k is the turbulent kinetic energy, ω is the specific dissipation, and ν_t is the eddy viscosity. These final three quantities are not non-dimensionalized, but all inputs and outputs are min-max normalized on a signal by signal basis. Note that the quantities in $\mathbf{x}'$ lose the impact of non-dimensionalization after min-max normalization since all simulations have the same vehicle length, but non-dimensionalization was performed regardless. The design-variable vector consists of just the rear slant angle α, given as

$$\boldsymbol{\mu} = \alpha \in \mathbb{R}^1. \tag{4.30}$$

4.5.3 Numerical Experiments: Single Vehicle Speed

All results of this section utilize a main network with 8 hidden layers and a hidden dimension of 100. DVH models use one-shot dense hypernetworks consisting of 5 hidden layers with 50 nodes per layer. A training/validation split of 80/20 was used, corresponding to just 2 cases in the validation set, which were randomly selected to be the 10 and 40 degree cases. Counter to what was seen in the 2D vehicle aerodynamics, the use of random Fourier features did not improve performance over baseline results, and in fact were worse across the board for all values of σ used. Instead, it was found that using Layer Normalization layers [201] between each hidden layer in DV-MLP models improved network predictions instead. Layer Normalization layers were added to the main network of DVH models but were found to be difficult to train. A summary of training and validation errors is given in table 4.20.

Table 4.20: Summary of training and validation error metrics at a vehicle speed of 60 m/s.

$\hat{\mathbf{q}}_i$	**Network Type**	**RMSE (train / val)**	**MRL2E (train / val)**
p [Pa]	DV-MLP	2.45e+01 / 2.27e+01	3.54% / 3.35%
	DV-MLP-LN	1.83e+01 / 1.91e+01	2.64% / 2.82%
	DVH	2.64e+01 / 2.63e+01	3.82% / 3.88%
u [m/s]	DV-MLP	1.31e+00 / 1.38e+00	2.47% / 2.63%
	DV-MLP-LN	8.31e-01 / 9.79e-01	1.57% / 1.86%
	DVH	1.46e+00 / 1.48e+00	2.75% / 2.80%
v [m/s]	DV-MLP	4.03e-01 / 4.34e-01	5.39% / 5.98%
	DV-MLP-LN	3.03e-01 / 3.90e-01	4.05% / 5.38%
	DVH	4.25e-01 / 4.67e-01	5.68% / 6.43%
w [m/s]	DV-MLP	7.94e-01 / 7.19e-01	10.33% / 9.79%
	DV-MLP-LN	6.03e-01 / 5.91e-01	7.84% / 8.05%
	DVH	8.02e-01 / 7.52e-01	10.43% / 10.24%
k [J/kg]	DV-MLP	2.05e+00 / 2.11e+00	9.38% / 11.11%
	DV-MLP-LN	1.52e+00 / 1.92e+00	6.89% / 10.10%
	DVH	2.03e+00 / 2.51e+00	9.30% / 13.27%
ω [1/s]	DV-MLP	1.92e+05 / 2.03e+05	12.51% / 13.28%
	DV-MLP-LN	1.64e+05 / 1.80e+05	10.71% / 11.77%
	DVH	2.04e+05 / 2.15e+05	13.34% / 14.10%
ν_t [m^2/s]	DV-MLP	4.78e-04 / 1.25e-03	3.23% / 8.08%
	DV-MLP-LN	3.55e-04 / 1.23e-03	2.40% / 8.01%
	DVH	4.61e-04 / 1.30e-03	3.11% / 8.50%

The pressure drag forces and coefficients were computed using Paraview batch scripting after writing the predictions in VTK format, with the method verified using the ground-truth data and comparing to the values reported by OpenFOAM. The errors in predicting the pressure drag coefficient are given in Table 4.21, where DV-MLP-LN performs best overall, with the validation errors actually smaller than the training errors. Plots of the predicted and CFD-ground-truth pressure drag coefficient are shown in Figure 4.27 for DV-MLP-LN.

Table 4.21: MRL2E (equivalent to mean-absolute-percent error) in predicting the pressure drag coefficient over the training and validation groups.

Network Type	**Train**	**Val**	**All**
DV-MLP	**1.60%**	3.61%	2.00%
DV-MLP-LN	1.75%	**1.69%**	**1.74%**
DVH	2.70%	5.05%	3.17%

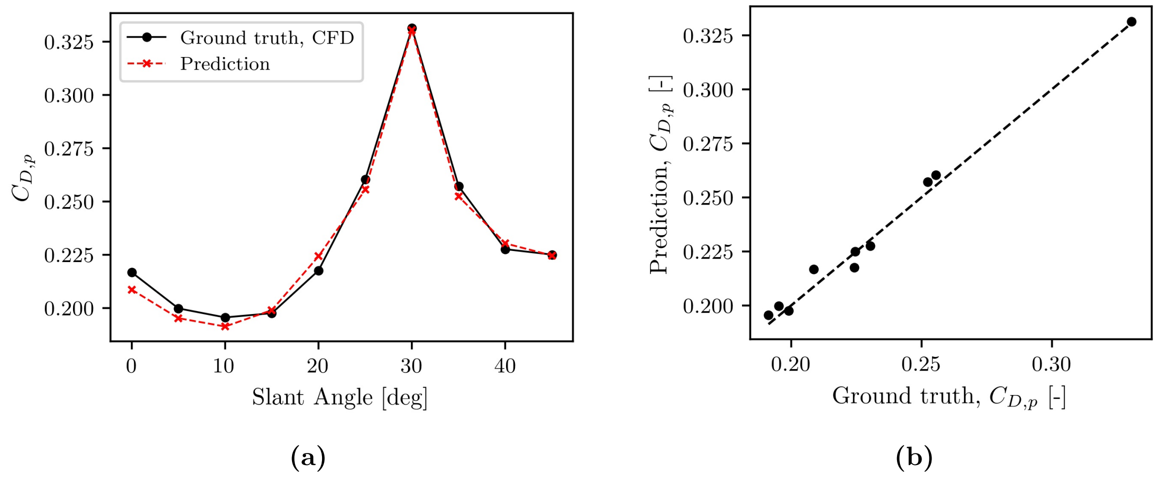


Figure 4.27: DV-MLP-LN pressure drag coefficient predictions, (a): versus slant angle and (b): predicted versus ground truth, where the CFD solution is considered the ground truth in this scenario. The overall trend is well captured, but with small errors for each drag coefficient.

Overall this is a positive result, given that the developed methods were able to scale to 3D without memory or other computational limitations. However, similar comments regarding the accuracy as mentioned at the end of Section 4.4.3. Namely, the achieved accuracy may be acceptable for a flow-field visualization scenario, but may not be adequate for driving optimization if the desired improvements are on the order of a few percent, given that the errors are also of a few percent.

Chapter 5

Design of Axial Compressor-Rotor Sections

5.1 Introduction

Aerodynamic shape optimization (ASO) routines typically use computational fluid dynamics (CFD) as part of an iterative algorithm to accelerate and automate the design process. Given some performance metric(s) comprising the objective and constraints along with a baseline design, the algorithms repeatedly vary the shape of the aerodynamic body, solve the appropriate governing equations, and compute, store, and track the resulting objective. The computational bottleneck tends to be the CFD solution, where hundreds to thousands of solutions may be required during run-time, often at prohibitively high cost. Surrogate-based optimization (SBO) reduces this expense by constructing cheaper approximations or meta-models to replace the costly CFD solution during convergence. Integral quantities of interest (QoIs) for the design are computed from the spatial or spatiotemporal field produced by the higher-fidelity CFD model, whereas the lower-fidelity surrogate models usually seek a map from the design variables to the QoIs, bypassing prediction of the complex flow field. Established surrogate-building methods include polynomial and radial basis function regression models, obtained via least squares, and kriging or equivalent Gaussian process regression models [202, 203]. The present work considers the development and application of deep-learning based flow-field emulators for use in surrogate-based ASO routines, applied to the design of axial compressor rotors.

ASO routines depend on at least three critical components:

1. the geometric parameterization to effectively represent shapes

2. the aerodynamic analysis to compute performance

3. the optimization routine to drive the process forward

In the context of airfoils, a variety of parameterization techniques have been developed and compared in terms of shape reconstruction [204] and impact during optimization [205]. An ideal parameterization would provide complete coverage of the design space with a small number of interpretable design variables. The expense of the aerodynamic analysis via CFD solution varies greatly depending on the problem dimensionality, the complexity of the domain and geometry, and the form of the governing equations considered. For some problems, lower-fidelity analysis via inviscid or potential flow solutions may be sufficient, and surrogates may not be required. For the axial compressors considered here, compressibility and viscous effects are important thus higher-fidelity modeling via the compressible RANS equations is needed. A wide variety of optimization algorithms are used and may be classified as either gradient-based or gradient-free approaches, depending upon how the next shape or shapes to be evaluated are determined. Gradient-based methods have been shown to scale to larger design spaces more effectively and converge more quickly than gradient-free methods [206], but are more susceptible to becoming trapped in local-minima and may fail to fully explore the design space [207]. Some gradient-free methods are more effective in exploration and avoiding such minima-trapping, but at the cost of a greater number of high-fidelity CFD evaluations.

Recent research incorporates machine learning into ASO algorithms in a variety of ways, and as outlined by Li et al. in ref. [208], the applications relate to the three critical ASO-routine components enumerated above via design of effective, compact geometric design spaces and generators [209, 210, 211], fast aerodynamic analysis [209, 55], and efficient design of optimization algorithms [212, 213]. Deep-learning in particular, the use of artificial neural networks (ANN), is widely applied in these tasks, and drives the recent surge in research.

A key difference between the present work and most existing works pertains to the mapping of the surrogate model. A notional ASO problem may be written generally as

$$\begin{aligned} \text{minimize} \quad & f(\boldsymbol{\mu}) \\ \text{by varying} \quad & \boldsymbol{\mu} = [\mu_1, \cdots, \mu_{n_\mu}]^T \\ \text{subject to} \quad & g(\boldsymbol{\mu}) \leq 0 \\ & h(\boldsymbol{\mu}) = 0. \end{aligned} \tag{5.1}$$

Machine-learning surrogates may replace the objective $f(\boldsymbol{\mu})$, usually mapping from the design variables $\boldsymbol{\mu} \subset \mathcal{M} \in \mathbb{R}^{n_\mu}$ directly to the objective $f(\boldsymbol{\mu}) \in \mathbb{R}$, or other quantity of interest such as a constraint, bypassing the prediction of the complex flow field. These surrogate may be referred to as "QoI emulators," since they approximate the mapping between the design variables and the QoIs directly, and may be notated as

$$\hat{f}_{\text{QoI}}(\boldsymbol{\mu}) : \mathcal{M} \rightarrow \mathbb{R}. \tag{5.2}$$

Whereas here *CFD emulators* are sought, where the entire CFD solution of all or a portion of the flow quantities are predicted, from which the objective and constraints are computed. Let the discretized CFD solution be written as $\mathbf{Q}(\mathbf{x}; \boldsymbol{\mu}) \subset \mathcal{Q} \in \mathbb{R}^{n_{\text{mesh}} \times n_q}$, where $\mathbf{x} \subset \Omega(\boldsymbol{\mu}) \in \mathbb{R}^d$ are the coordinates in d-dimensional physical space, defined over domain $\Omega(\boldsymbol{\mu})$. The CFD emulator approximation may be written as $N(\boldsymbol{\mu}; \theta) = \hat{\mathbf{Q}}(\boldsymbol{\mu})$, where N is a neural network with its set of trainable weights θ. The objective approximation is then $\hat{f}_N(\boldsymbol{\mu}) = f\big(\hat{\mathbf{Q}}(\boldsymbol{\mu})\big)$.

The proposed CFD emulators generate flow-field approximations in different ways, either operating directly in the computational domain and ignoring $\mathbf{x} \subset \Omega(\boldsymbol{\mu})$,

$$N_{\text{DCNN}} : \mathcal{M} \rightarrow \mathcal{Q}; \quad N_{\text{DCNN}}(\boldsymbol{\mu}; \theta) = \hat{\mathbf{Q}}, \tag{5.3}$$

or by operating in the physical domain and including additional spatial inputs $\mathbf{x}'$

$$N_{\text{DVH}} : \mathcal{M} \times \mathcal{X}' \rightarrow \mathcal{Q}; \quad N_{\text{DVH}}(\boldsymbol{\mu}, \mathbf{x}'; \theta) = \hat{\mathbf{Q}}. \tag{5.4}$$

In this case $\mathbf{x}' \triangleq \left[\mathbf{x}^T, \phi(\mathbf{x}; \boldsymbol{\mu})\right]^T$, where $\phi(\cdot)$ is the signed distance function. Approximating the CFD solution is more difficult as it is of higher dimension, $n_{\text{mesh}} \times n_q$, whereas QoI emulators predicts a scalar objective or small vector of QoIs.

The content of this paper is as follows: First the compressor-rotor ASO problem of interest is introduced, along with information regarding the CFD solutions and Bayesian optimization routine. Next the ANN models are introduced, starting with Decoder Convolutional Neural Network (DCNN) models, which are placed in context with widely-used autoencoder-type models. Then Design-Variable Hypernetworks (DVH) are introduced, and the inherent advantages and disadvantages of each are discussed. Results emulating subsonic compressor-rotor flows are then presented, where both models perform well. Next, a challenging transonic compressor-rotor problem is explored, starting with geometric design variables only. The design space is then expanded to include variation in flow conditions via changing rotor speed. Finally, a single DVH model is used in place of CFD to drive shape optimization at varying rotor speed.

5.2 Methodology

5.2.1 ASO Problem Statement and RANS Solutions

ASO may be used in the aerodynamic design of axial compressor components, using solely CFD or with surrogates. The objective of each compressor stage is to increase the total pressure from inlet to outlet, and this of course requires mechanical work which should be minimized. Thus the efficiency is a critical design objective for a compressor stage, and the adiabatic compressor efficiency may be written as

$$\eta = \frac{\left(\frac{p_{02}}{p_{01}}\right)^{(\gamma-1)/\gamma} - 1}{\frac{T_{02}}{T_{01}} - 1}, \tag{5.5}$$

where all quantities are mass-flow averaged, station 1 is the inlet, and station 2 is the outlet. The stagnation pressure ratio, $p_R \triangleq \frac{p_{02}}{p_{01}}$ is clearly of prime importance, and is often a design target during stage or overall engine sizing and detailed design. The ASO problem of interest is to improve the efficiency of a baseline design $\boldsymbol{\mu}_0$ while at least maintaining baseline p_R, and may be written as

$$\begin{aligned} \text{maximize} \quad & \eta(\boldsymbol{\mu}) \\ \text{by varying} \quad & \boldsymbol{\mu} = [\mu_1, \cdots, \mu_{n_\mu}]^T \\ \text{subject to} \quad & p_R(\boldsymbol{\mu}) \geq p_R(\boldsymbol{\mu}_0). \end{aligned} \tag{5.6}$$

This general ASO problem is applied to 2D rotor sections and is solved using surrogate-based optimization. The surrogates require CFD solutions, which are generated using RTRC in-house CFD solver UTCFD which solves the 3D compressible RANS equations using the k-ω turbulence model. A 2D circumferential slice of the 3D solution is considered, with an example downsampled mesh for a subsonic condition shown in Figure 5.1. Multi-block or multi-zone meshes are used, consisting of 4 blocks, differentiated by color. Subsonic and transonic meshes have similar structure, but varying topology in terms of the size and extent of each sub-domain. Each mesh zone involves a coordinate transformation between non-uniformly spaced physical coordinates x/y and uniformly-spaced computational coordinates ξ/η, with a body-fitted mesh in the zone nearest the airfoil. This unit spacing in computational coordinates and consistent mesh dimensions allows direct application of convolutional layers, as discussed in Section 5.2.2.1.

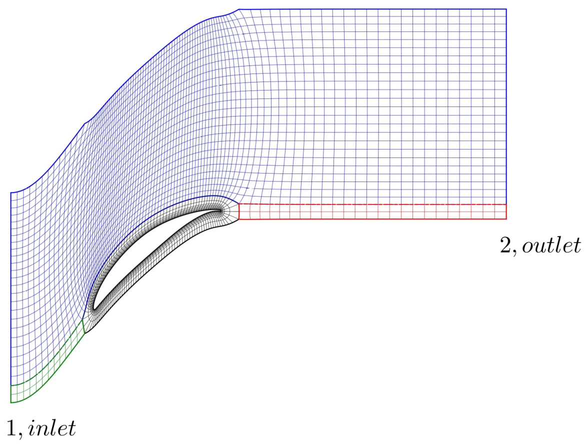


Figure 5.1: Example multi-block mesh for a subsonic rotor airfoil, where the inlet and outlet are labeled.

Surrogate-based optimization is employed, using CFD or CFD emulators in a constrained Bayesian optimization (BO) technique [214] with flexible surrogate structure, which is built upon authors' prior work [215, 216, 217]. This technique addresses the lack of data generated from expensive CFD solvers by starting with a very low sample-to-variable ratio DoE and iteratively learning an adaptive surrogate with flexible structure (e.g. Gaussian process, neural net etc.) for efficiency maximization via adaptive sampling. Gaussian processes are used in this work, trained with data either from CFD or from a CFD-emulator. In Bayesian sampling, the acquisition function controls the trade-off between exploration and exploitation. To handle the pressure ratio inequality constraint in Equation 5.6 in the BO setting, a constrained version [218] of the lower confidence bound acquisition function is used. In the comparisons in later sections, CFD-based BO is compared against CFD-emulator based BO, where the predictions from the CFD-emulator are not used directly but instead train separate QoI emulators in the BO routine.

Figure 5.2 illustrates the performance space (efficiency and pressure ratio relative to the baseline, i.e. 70% span section of NASA rotor 37, shown as the black squares) and CFD-based single-objective optimization result per Equation 5.6 at distinct rotational speeds. These CFD-based optimization results will be used to benchmark the ability of the emulators to drive meaningful optimization.

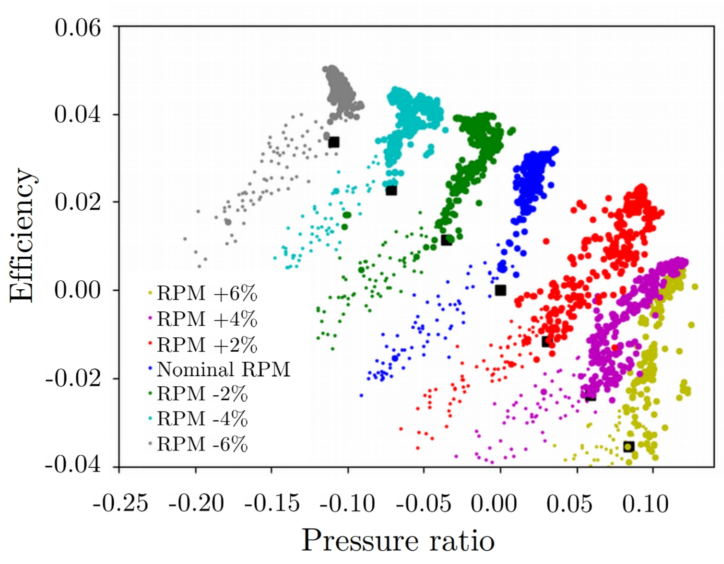


Figure 5.2: CFD-driven optimization in performance space at varying rotor speed, where the baseline NASA rotor 37 at 70% span corresponds to the black squares.

5.2.2 ANN Flow Emulators

Neural networks are a popular class of machine learning model responsible for many recent advances in image and natural language processing [219] with increasing application in science and engineering [220]. In feed-forward neural networks (FFNN), information moves in just one direction through the model, from inputs to outputs, through a series of layers without cycles. Architectures containing cycles are known as recurrent networks which are often used for sequential or temporal problems. Each hidden layer receives an input, applies a non-linear operation via activation function, and passes the output to the next layer. Often times in predictive tasks a linear output layer is used. Entire models may be expressed as differentiable composite functions, and each layer has a set of trainable parameters which are determined during training, usually using a variant of stochastic gradient descent. A wide variety of network layers and models have been developed, with dense and convolutional layers used to construct the models here. FFNNs consisting of only dense layers are also known as multi-layer perceptron (MLP), while convolutional neural networks (CNN) often contain dense and convolutional layers.

Early research showed the promise of utilizing neural networks in aerodynamic design [221, 222, 223, 224], including the design of turbomachinery airfoils [225, 226]. This early work often involved using neural networks to build response surfaces and as such most may be categorized as QoI emulators, although some predict the spatial variation in airfoil-surface pressure coefficient. Applications may be further classified according to the mapping the neural network approximates; either the "forward" map from geometry/geometric parameters to performance metrics, or the "reverse" map, from specified performance metrics, usually airfoil surface pressure coefficient distribution, to the shape or shape parameters for the corresponding design. ANNs using the reverse map are often used in inverse design routines. For example, FFNNs have been used in reverse mode with panel-method aerodynamics to predict the Bezier-PARSEC parameters [227], and with panel-method, inviscid potential, or Euler solvers to predict Bezier coefficients [228]. A CNN was used in reverse mode to map from an image of the desired pressure coefficient distribution to the shape parameters of the corresponding airfoil [229]. Self-organizing maps with MLPs were used to map from performance QoIs and operating conditions to PARSEC shape parameters for airfoils and wings [230].

Neural networks operating as QoI emulators in forward mode are more commonly used in SBO routines. CNN-based models predicting lift and drag coefficients were successfully integrated into a Bayesian optimization routine for subsonic airfoils, achieving the same designs as CFD with less than two orders of magnitude online runtime [231]. Gradient-enhanced MLPs were used for airfoil optimization in the subsonic and transonic regimes with a single model, where inclusion of gradient information decreased the number of training epochs and improved predictive performance [56]. MLPs have been used to predict lift and drag coefficients for UAV applications given wing shape and loading parameters, using the vortex lattice method for aerodynamics and a genetic algorithm for optimization [55]. Multi-fidelity ASO was performed using a novel ANN architecture, with sub-networks to model the linear and non-linear components of the correlation between low and high fidelity models, with lift and drag coefficients of interest [232]. Response surface methods and radial basis networks have been used jointly to map between design variables and QoIs to optimize a supersonic turbine, using a combination of meanline 1D codes and 3D CFD [233]. In many other instances QoI emulators are developed on their own or as a sub-component of a larger problem, without integration into optimization, with applications including airfoil performance [234, 235], transonic buffeting [236], and aeroelastic modeling [237]. More thorough overviews of machine learning in ASO may be found in the literature [208, 238].

The proposed methods use the flow solutions directly, without relying on interpolation. CNN-based models require Cartesian-structured data, while flow solutions lie on irregular meshes. Typically to use a CNN to predict a flow field, one must interpolate from the computational domain to a Cartesian grid overlain on the problem domain [76, 80, 78, 239] or restrict the problem to the laminar regime so a reduced-fidelity solver may be used [75]. The interpolation has several drawbacks, most critical is the fluid domain loss of information where mesh points are more tightly clustered (boundary layers, wakes), as well as a lower-fidelity, pixelated representation of the problem domain and geometry. Additionally, if an aerodynamic body is present, points lying within the component are wasted since they will be masked out of the final prediction. These issues are exemplified in Figure 5.3, most notable through the boundary layer.

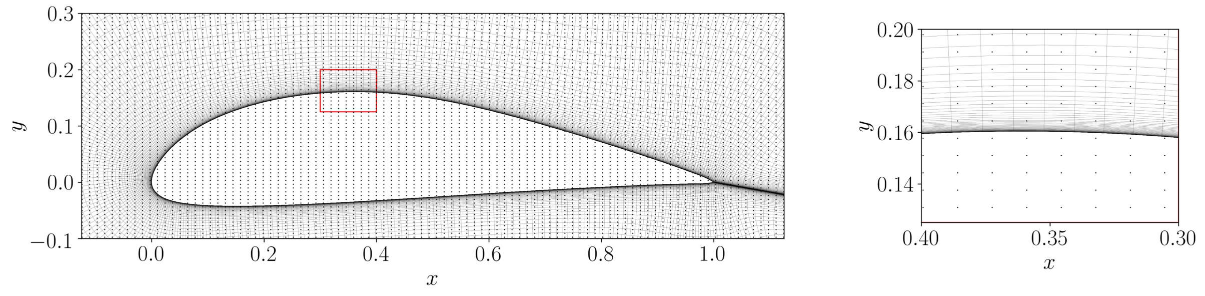


Figure 5.3: Cartesian grid overlain on computational mesh, highlighting the loss of information associated with interpolation to such a grid.

However, if there is regular structure to the mesh then convolutional neural network architectures may be applied directly in the computational domain without interpolation, avoiding the issues given above. The multi-block meshes under consideration here meet this requirement, as discussed in Section 5.2.1.

Several recent works utilize this computational domain CNN concept, though most are not linked to optimization. A U-net, an autoencoder with skip connections across the latent space, has been used in the computational domain, mapping mesh metrics or signed-distance-functions representations to airfoil flow-fields [240] for 2D and 3D aerodynamic problems. A fully-convolutional computational-domain CNN was coupled with a steady state, incompressible RANS CFD solver by mapping from partially-converged to fully-converged flow quantities, providing a speedup of 1.9-7.4X [241]. A decoder CNN with residual connections, similar to DCNN models of Section 5.2.2.1, were used to reconstruct micro-fluidic heat transfer flows, using either design parameters or experimentally-measured values as network inputs [242].

5.2.2.1 Autoencoders and Decoder Convolutional Neural Networks (DCNN)

Autoencoders have a bottleneck structure and may be used for a variety of supervised and unsupervised learning or representation tasks. Autoencoders seek mappings to and from a data space $\mathcal{X} \in \mathbb{R}^{n_\mathrm{x}}$ and a latent space $\mathcal{Z} \in \mathbb{R}^{n_\mathrm{z}}$, where $n_\mathrm{z} \ll n_\mathrm{x}$. The latent representation, a vector $\mathbf{z} \in \mathcal{Z}$, is then a reduced-dimensional embedding of the data. Typically autoencoders consist of an encoder $\Phi(\mathbf{x})$ which maps from the data space to the bottleneck latent space

$$\Phi : \mathcal{X} \to \mathcal{Z}; \quad \Phi(\mathbf{x}) = \mathbf{z}, \tag{5.7}$$

and a decoder $\Theta(\mathbf{z})$ which inverts the mapping to approximate the input data,

$$\Theta : \mathcal{Z} \to \mathcal{X}; \quad \Theta(\mathbf{z}) = \hat{\mathbf{x}} \approx \mathbf{x}. \tag{5.8}$$

Typically, the encoder and decoder are parameterized using neural networks, and trained using a variant of stochastic gradient descent. Convolutional autoencoders have been used in a predictive setting by taking an external target $\mathbf{Q}$ different from $\mathbf{x}$ and redefining the decoder as

$$\Theta : \mathcal{Z} \to \mathcal{Q}; \quad \Theta(\mathbf{z}) = \hat{\mathbf{Q}} \approx \mathbf{Q}. \tag{5.9}$$

As an example, an autoencoder CNN was used to predict turbulent airfoil flows using the signed distance field as input [76] (see Figure 5.4a). Global parameters $\boldsymbol{\mu} = \begin{bmatrix}\mathrm{Re} \ \mathrm{AoA}\end{bmatrix}^T$ are concatenated with the encoder output and fed-forward through the decoder to generate an approximate solution, as shown in Equations 5.10-5.12.

$$\begin{aligned} \Phi(\mathbf{x}) &= \tilde{\mathbf{z}} && (5.10) \\ \mathbf{z} &= \begin{bmatrix}\boldsymbol{\mu}^T & \tilde{\mathbf{z}}^T\end{bmatrix}^T && (5.11) \\ \Theta(\mathbf{z}) &= \hat{\mathbf{Q}} \approx \mathbf{Q} && (5.12) \end{aligned}$$

In other methods, a solution for a new set of conditions may be approximated using only the trained decoder and some form of mapping or interpolation in the latent space [80].

In a predictive setting, the dimensionality of input and output spaces do not have to agree.Additionally, the dimensions of the grids in input and output spaces do not have to agree. Computational experiments were performed using dataset 1 where the input space

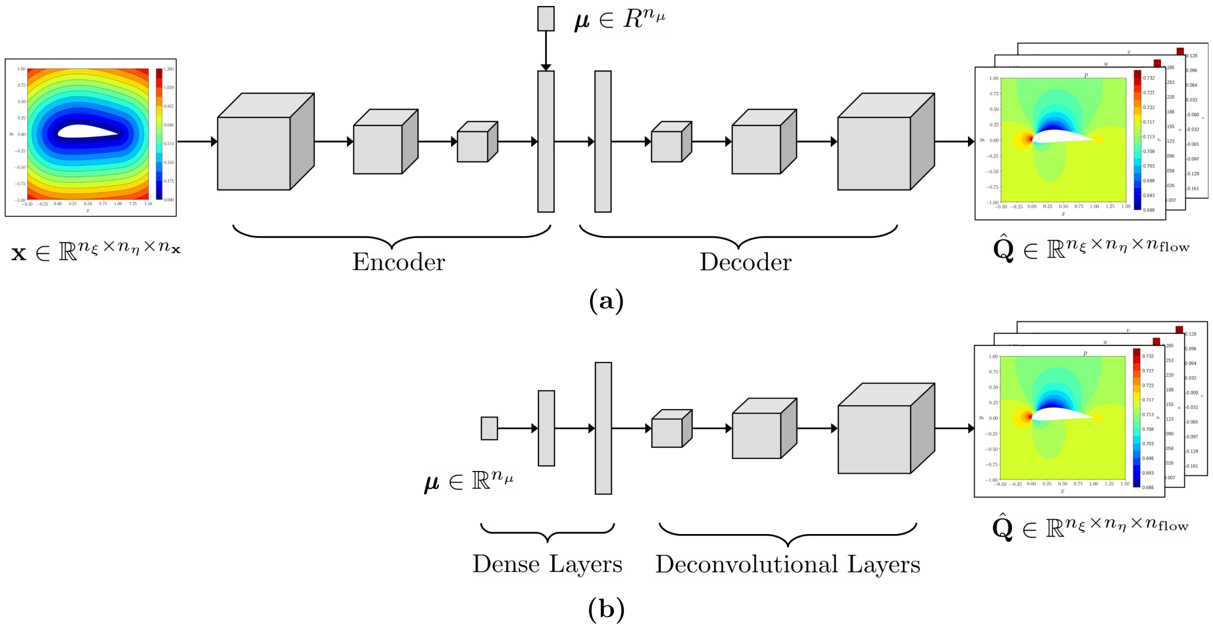

$\boldsymbol{\mu} \in R^{n_\mu}$
$\mathbf{x} \in \mathbb{R}^{n_\xi \times n_\eta \times n_\mathbf{x}}$
Encoder
Decoder
$\hat{\mathbf{Q}} \in \mathbb{R}^{n_\xi \times n_\eta \times n_\text{flow}}$
(a)
$\boldsymbol{\mu} \in \mathbb{R}^{n_\mu}$
Dense Layers
Deconvolutional Layers
$\hat{\mathbf{Q}} \in \mathbb{R}^{n_\xi \times n_\eta \times n_\text{flow}}$
(b)

Figure 5.4: Schematic comparison of (a) autoencoder CNN with latent-space injection, model as presented in [76], and (b) DCNN, with sequence of dense and transposed convolution layers.

was down-sampled, resulting in a non-symmetric autoencoder CNN mapping from integer mesh coordinates η/ξ to flow field predictions. It was found that these networks performed better in terms of validation MSE when the inputs were downsampled, suggesting that a smaller encoder is better for this application. Following this line of reasoning leads to DCNN-based models, consisting of only the decoder half of the network. In this setting, $\mathbf{z} = \boldsymbol{\mu}$, the prior information is used as network input, fed directly to the decoder. We propose that, for the types of problems of interest in optimization, the design variables for the problem $\boldsymbol{\mu}$ may be used in lieu of a learned latent representation $\mathbf{z}$, given that the design variables define a unique design instance, assuming the generative model is deterministic and one-to-one. This corresponds to eliminating the encoder as the design variables replace the learned latent representation, and this defines DCNN models.

The difference between autoencoder CNNs and DCNNs is visualized in Figure 5.4. Although the DCNN schematic shows a sequence of dense layers, numerical experiments showed the best results, in terms of mean squared error, were achieved with a single dense, linear layer between the input and first decoder layer. Thus only a single dense layer is used for all DCNN predictions shown in later sections.

Beyond the present discussion, encoding and decoding are fundamental concepts and operations in machine learning. Convolutional classifiers correspond architecturally to the encoder of Figure 5.4, where a vector of class probabilities replaces the produced latent representation. Generative adversarial networks (GANs), a popular generative modeling method, may utilize convolutional decoders in image synthesis, analogous in architecture to DCNN models, although with a very different training scheme. These use cases correspond to more probabilistic scenarios and as a result use different training losses and training schemes, such as binary-cross-entropy loss for classification, and paired training of generator and discriminator networks in GANs via minimax game [166].

5.2.2.2 Design Variable Hypernetworks (DVH)

Some recent research in computer graphics and image processing has moved away from autoencoder-type models with grid-sampled data to continuous representations using MLPs [168]. For example, if the goal is to represent an $m \times n$ image with c channels $\mathbf{Y} \in \mathbb{R}^{m \times n \times c}$ using a neural network, then the input would be the coordinates for a single pixel $\mathbf{x}_{ij} \in \mathbb{R}^2$ and the output would be the channel values $\mathbf{y}_{ij} \in \mathbb{R}^c$ of that pixel. This is in contrast to the image-to-image mappings of autoencoders. In rendering tasks 3D objects have been represented by learning the volumetric signed-distance field around the object in a pointwise, mesh-agnostic sense, with the object implicitly defined by the zero level set of the field. To represent multiple shapes an embedding vector $\mathbf{z}$ may be defined for each unique image and used as additional input, concatenated with each $\mathbf{x}_{ij}$ [158]. However, it appears that doing so comes at the cost of reduced accuracy for any single case, as compared to a network $N(\cdot, \theta_m)$ without input vector $\mathbf{z}$, overfit to a single case [159]. Following this line of thinking, the main network weights θ_m could then be used as an embedding for each instance, and this leads to Design Variable Hypernetworks (DVH). With DVH, the main network weights for a case θ_m^j are generated using a neural network $N(\boldsymbol{\mu}_j; \theta_h)$ known as a hypernetwork.

Hypernetworks [165] are a meta-modeling approach to deep learning, where one network generates the weights of another. This means the output space of the hypernetwork is that of *flow-field generating neural networks* instead of flow-fields directly. In this ap-

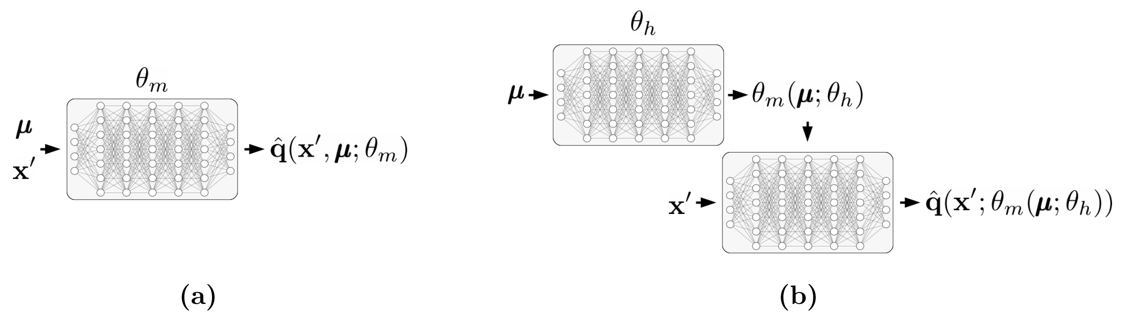

θ_m
$\boldsymbol{\mu}$
$\mathbf{x}'$
$\hat{\mathbf{q}}(\mathbf{x}', \boldsymbol{\mu}; \theta_m)$
(a)
θ_h
$\boldsymbol{\mu}$
$\theta_m(\boldsymbol{\mu}; \theta_h)$
$\mathbf{x}'$
$\hat{\mathbf{q}}(\mathbf{x}'; \theta_m(\boldsymbol{\mu}; \theta_h))$
(b)

Figure 5.5: Comparison of (a) coordinate-MLP with embedding vector $\boldsymbol{\mu}$, and (b) design-variable hypernetwork, where the main network weights and biases θ_m are generated using a hypernetwork.

proach, a weight-generating hypernetwork is used along with a flow-field-predicting main network, where the training loss is computed according to the flow-field predictions, but the trainable parameters exist solely in the hypernetwork. DVH uses a one-shot dense hypernetwork, meaning that all of the main network weights and biases are generated in one forward pass of the hypernetwork as a long vector. The vector is then split and reshaped appropriately to form the weight matrices and bias vectors of the main network. DVH may be considered as an instance of Neural Implicit Flow [172], as applied to surrogate modeling.

DCNN models generate snapshots in the computational domain while DVH models provide pointwise predictions in physical space. First, a main network architecture is defined, with a coordinate-based MLP with embedding vector $\mathbf{z} = \boldsymbol{\mu}$ shown in Figure 5.5a. Then the trainable weights and biases θ_m are generated as a function of the design variables $\boldsymbol{\mu}$ using a hypernetwork, as shown in Figure 5.5b.

5.2.2.3 Implementation and Considerations Between DCNN and DVH

All models are implemented using the Tensorflow python library [126]. The models are trained using Adam optimizer [112] using a mean-squared-error (MSE) loss function. All inputs and outputs are min-max normalized component-wise so they lie approximately between 0 and 1. Batch-normalization layers are used after each layer in DCNN models, except the output layer. The DCNN models use the same number of filters in each hidden convolutional layer across the model. Swish activation function is used for all hidden layers, and all models use a linear output layer.

Two methods of training DVH models are considered, and are summarized in Table 5.1. The first method, known as fully-mixed batching, evaluates the hypernetwork and main network for each spatial location. The solution data from all cases is flattened and fully mixed, with the design variables tiled across the mesh. The batch size corresponds to the number of physical locations where a prediction is needed, spanning many designs. An optimizer step is taken after a specified number of locations are evaluated. A second, more efficient method, known as batch-by-case, evaluates the hypernetwork for a given design once and makes predictions for all spatial locations corresponding to that design. The batch axis corresponds to the number of cases in this scenario, and when batch-by-case training is used is always set to 1 in this work. Fully-mixed batching was developed first, and preliminary explorations showed that fully mixing the data increased the stochatsticity of the training process, allowing the optimizer to break free from local minima. Batch-by-case was implemented later as computational costs became a concern, and was found to be roughly an order of magnitude faster than fully-mixed batching, with no ill effects on training accuracy or convergence rate. The differences in the approaches can be understood by examining the dimension and shape of the training arrays, as given in Table 5.1 below.

Table 5.1: Comparison of DVH training methods by examining the shape of the training arrays, meaning of batch axis, and compared training time for the transonic problem in Section 5.4.4.

	Fully-Mixed Batches		**Batch-by-Case**	
	$\dim(\mathbf{M}_{\text{train}}) = \left[(n_{\text{train}} \times n_{\text{mesh}}) \times \dim(\boldsymbol{\mu})\right]$	(2D)	$\dim(\mathbf{M}_{\text{train}}) = \left[n_{\text{train}} \times \dim(\boldsymbol{\mu})\right]$	(2D)
Training Arrays	$\dim(\mathbf{X}'_{\text{train}}) = \left[(n_{\text{train}} \times n_{\text{mesh}}) \times \dim(\mathbf{x}')\right]$	(2D)	$\dim(\mathbf{X}'_{\text{train}}) = \left[n_{\text{train}} \times n_{\text{mesh}} \times \dim(\mathbf{x}')\right]$	(3D)
	$\dim(\mathbf{Q}_{\text{train}}) = \left[(n_{\text{train}} \times n_{\text{mesh}}) \times \dim(\mathbf{q})\right]$	(2D)	$\dim(\mathbf{Q}_{\text{train}}) = \left[n_{\text{train}} \times n_{\text{mesh}} \times \dim(\mathbf{q})\right]$	(3D)
Batch Axis	# spatial locations, any case		# complete cases, all locations, always 1	
Training Time	51.3 seconds/epoch		4.01 seconds/epoch	

DCNN models, or more generally CNN-based models, are more widely used and understood than hypernetworks. CNNs have driven many advances in the state-of-the-art in deep learning, particularly in image recognition, processing, and synthesis tasks. However, being designed for Cartesian data is an inherent limitation and disadvantage for CFD emulation, as it requires that all meshes must be topologically identical with regular Cartesian structure. That is, a DCNN model trained on the subsonic dataset could not make a prediction for a transonic case since the meshes are not identical in structure, assuming the parameterization $\boldsymbol{\mu}$ accounted for the differences in flow condition and shape. Further, CNN-based models do not scale well to 3D problems and can quickly run into memory limitations, particularly for intermediate hidden states which are 5D tensors when batching is considered during training. DVH does not have either of these limitations since it provides pointwise predictions in space, allowing the meshes to vary in size and topology among training examples. Fundamentally, the size of the model and the mesh are linked for DCNN, while for DVH they are indirectly coupled, in that a DVH model must have sufficient size to represent the complexity of the fields defined on the mesh while not being directly linked to the degrees-of-freedom. DVH does however require spatial information, implying knowledge of and access to the parametric geometry model, mesh generator, and signed distance-function calculator (Eikonal equation solver or other approximation). DVH scales naturally to 3D or higher dimensions since it is not linked to snapshots; instead the input/output spaces/vectors simply increase dimension by one. Additionally, defining a valid CNN-based model is a non-trivial task as it depends on the dimension of the input and output spaces. On the other hand, design of DVH models is straight-forward, particularly when a dense, one-shot hypernetwork is used, where just the network width and depth for the main network and hypernetwork are specified.

5.3 Subsonic Compressor Airfoil Emulation and Optimization

The subsonic airfoils are parameterized by 4 geometric parameters, $\boldsymbol{\mu} \in \mathbb{R}^4$, corresponding to 4 offset points along the camber distribution. A dataset of solutions corresponding to 1000 unique airfoil geometries, selected via latin hyper-cube sampling, was generated using the CFD solver as described in Section 5.2.1. All subsonic solutions lie on similar meshes with the same dimension and topology.

Parallel DCNN models sharing an input layer are used in each mesh block and trained concurrently, with the model architecture specified in Table 5.2. The kernel size and strides are specified in equivalent-encoder order; starting from the output space and working towards the DCNN input dense layers. The dimensions of each mesh block required rectangular kernels to be used instead of square kernels, and the kernel size and strides are set separately from each other. Indeed, we emphasize that finding a valid design with odd mesh dimensions is a non-trivial task.

Table 5.2: Summary of subdomain DCNNs.

Block	**Output Dimension**	**Kernel size** (Equiv. encoder)	**Strides** (Equiv. encoder)
1	513×57	$[2,2] \to [2,2] \to [4,2] \to [4,4] \to [2,2]$	$[1,1] \to [2,2] \to [4,2] \to [4,2] \to [2,1]$
2	393×97	$[2,2] \to [7,2] \to [2,3] \to [2,2] \to [2,2]$	$[1,1] \to [7,2] \to [2,3] \to [2,2] \to [2,2]$
3	49×9	$[2,2] \to [2,3] \to [4,4]$	$[1,1] \to [2,1] \to [4,1]$
4	97×9	$[2,2] \to [2,3] \to [2,2] \to [2,2]$	$[1,1] \to [2,1] \to [2,1] \to [2,1]$

Training curves for DCNN and DVH models are shown in Figure 5.6, where just 160 cases are used to train the models and three flow quantities are predicted: $\mathbf{q} = [p\ u\ v]^T$. The DCNN model uses the architecture of Table 5.2 with 100 filters per layer.The DVH main network and hypernetwork each consist of 5 hidden layers with 50 hidden nodes per layer, and the DVH model was trained using fully-mixed batching with a batch size of 40,000 points. The DCNN model was trained with a batch size of 8 cases, and both used a constant learning rate of 1×10^{-4}.

A comparison of error metrics for the resulting DCNN and DVH models is given in Table 5.3. The min-max normalized outputs are re-dimensionalized before errors are computed. The root-mean squared error (RMSE) and mean absolute error (MAE) are intuitive since they have the units of the output quantity while providing slightly different measures of the error, where RMSE penalizes large errors more so than MAE. The mean relative L2 error (MRL2E) may be multipled by 100 and loosely interpreted as an average percentage error. The best performing model for each is highlighted in bold in Table 5.3, and this shows that DCNN and DVH models perform similarly to one another. Interestingly, for the flow quantity u, the best RMSE belongs to the DCNN, while the best MAE belongs to DVH, indicating that DVH has a higher frequency of larger errors than DCNN, but on average the errors are smaller in magnitude, with a similar comment

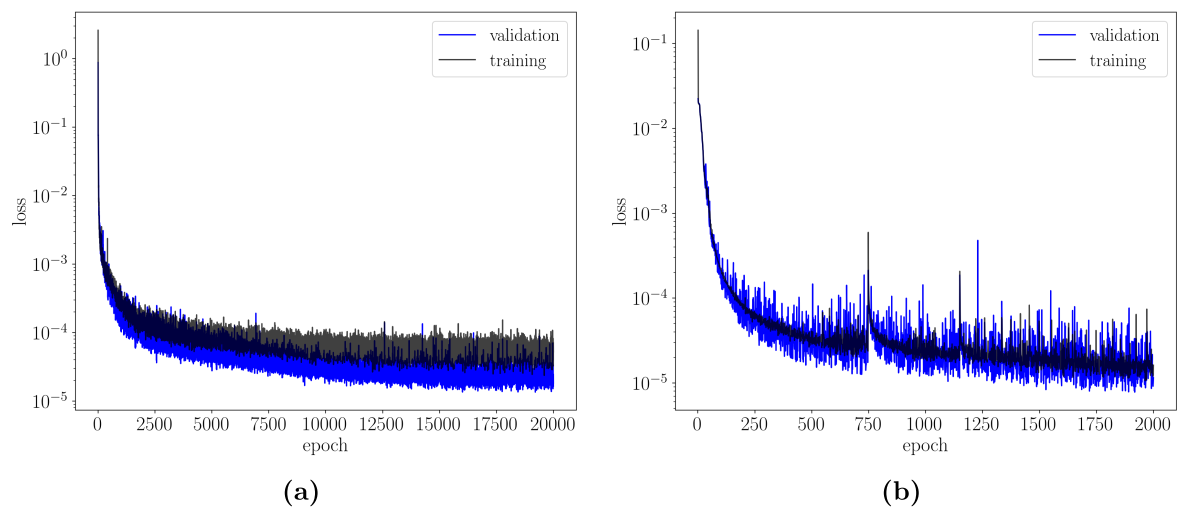


Figure 5.6: Training curves for (a) DCNN and (b) DVH emulators, corresponding to min-max normalized quantities.

for the validation group for flow quantity p. These errors demonstrate that both models generalize well and predict accurately, with the training and testing errors of very similar size and MRL2E ¡ 1% for all quantities. This also shows that both model types are data-efficient, in that only 160 of 1000 cases are used in training, while the validation group consists of the remaining 840 designs.

Table 5.3: Summary of dimensional training and validation error metrics RMSE, MAE, and MRL2E for the subsonic airfoil dataset.

$\hat{\mathbf{q}}_k$	Network Type	RMSE (train / val)	MAE (train / val)	MRL2E (train/val)
p [lbf/ft^2]	DCNN	$\mathbf{6.30 \cdot 10^{-2}}$ / $\mathbf{6.98 \cdot 10^{-2}}$	$\mathbf{3.95 \cdot 10^{-2}}$ / $4.42 \cdot 10^{-2}$	$\mathbf{2.98 \cdot 10^{-5}}$ / $\mathbf{3.30 \cdot 10^{-5}}$
	DVH	$7.63 \cdot 10^{-2}$ / $8.05 \cdot 10^{-2}$	$4.00 \cdot 10^{-2}$ / $\mathbf{4.24 \cdot 10^{-2}}$	$3.58 \cdot 10^{-5}$ / $3.78 \cdot 10^{-5}$
u [ft/s]	DCNN	**0.32** / **0.40**	0.22 / 0.25	$\mathbf{4.98 \cdot 10^{-3}}$ / $\mathbf{6.15 \cdot 10^{-3}}$
	DVH	0.46 / 0.52	**0.21** / **0.23**	$6.78 \cdot 10^{-3}$ / $7.40 \cdot 10^{-3}$
v [ft/s]	DCNN	0.27 / 0.29	0.21 / 0.22	$6.16 \cdot 10^{-3}$ / $\mathbf{6.79 \cdot 10^{-3}}$
	DVH	**0.25** / **0.27**	**0.13** / **0.14**	$\mathbf{5.56 \cdot 10^{-3}}$ / $6.81 \cdot 10^{-3}$

Representative pressure and x-velocity field predictions for DCNN and DVH models on an unseen airfoil are shown in Figures 5.7 and 5.8. The ground-truth and predicted fields agree well, and the dominant flow structures are well captured. The suction-side low-pressure pressure region is accurately captured, with only small deviations present in the errors, which cluster near the trailing edge for p in both model types. The x-velocity is also well represented, but larger errors are seen throughout the wake. It is interesting to note that the DCNN error contours have a slightly pixelated quality, while DVH does not, likely an artifact of convolutional models versus dense models.

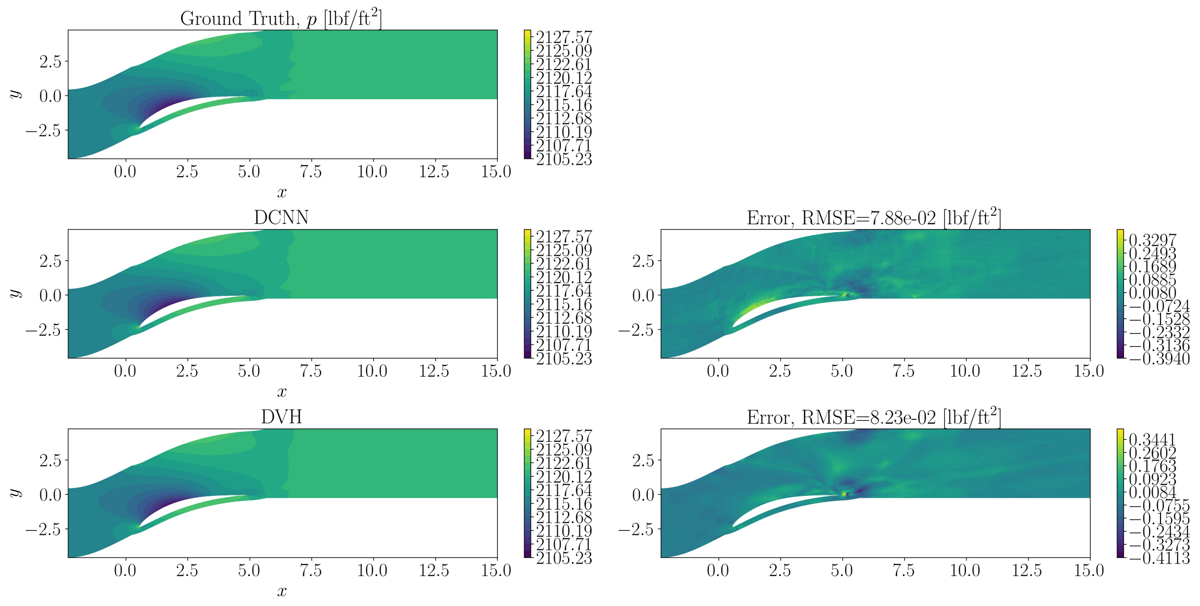


Figure 5.7: Comparing ground truth and predicted pressure fields for DCNN and DVH models on an unseen airfoil, for a subsonic condition. Error colorbars are limited to $\pm 5\times$ RMSE.

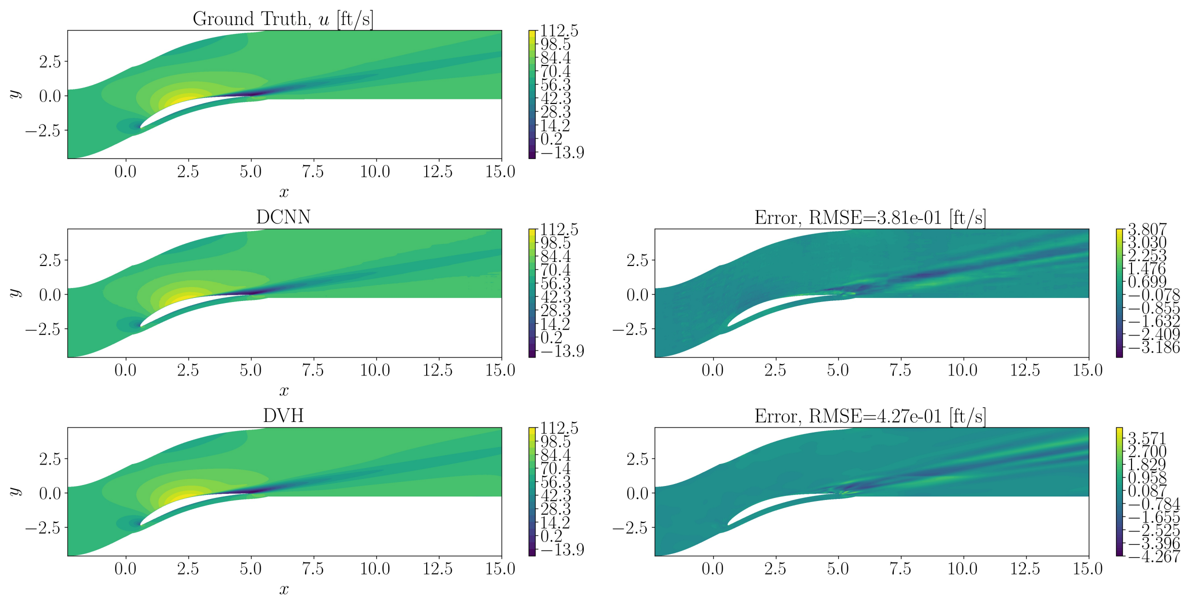


Figure 5.8: Comparing ground truth and predicted x-velocity fields for DCNN and DVH models on an unseen airfoil, for a subsonic condition. Error colorbars are limited to $\pm 10\times$ RMSE.

The pressure forces acting on the airfoil surfaces were extracted from the predictions for each model. The corresponding lift and drag coefficients for the DCNN and DVH models are shown in Figures 5.9 and 5.10 respectively, where a perfect prediction would lie on the dashed line. The drag coefficient predictions match closely for both models, but a small offset for the lift coefficient is seen for DCNN, with the predicted values smaller than the ground truth.

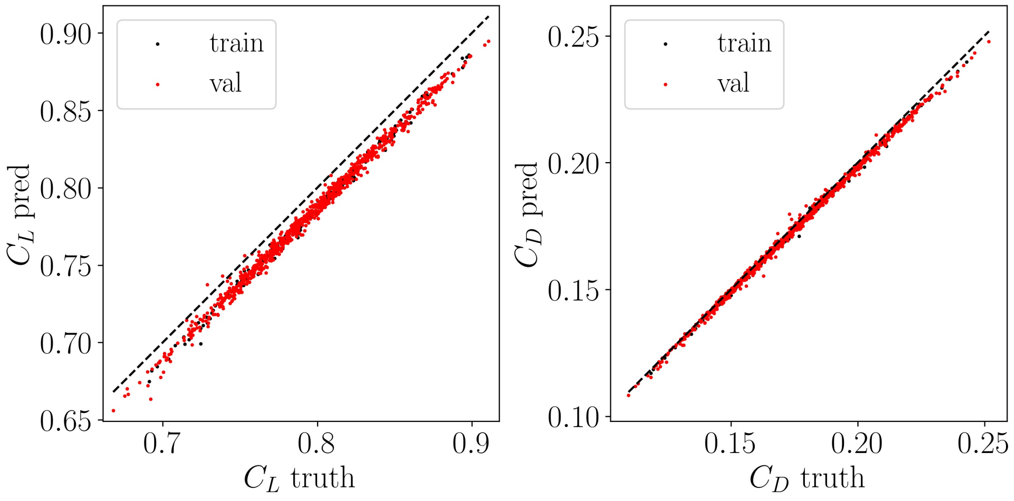


Figure 5.9: The predicted vs. ground truth pressure lift and drag coefficients for the DCNN model.

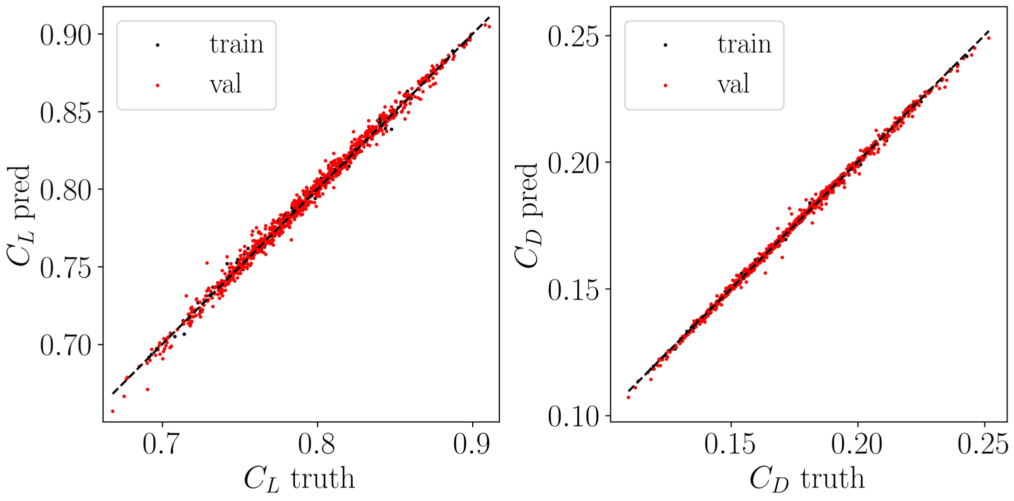


Figure 5.10: The predicted vs. ground truth pressure lift and drag coefficients for the DVH model.

The offset in predicted lift coefficient for DCNN is due to over-prediction of the suction side (top) pressure, towards the leading edge. This leads to an under-predicted pressure coefficient in this region, and consequently less lift overall when integrated over the surface. An example pressure coefficient distribution on the airfoil surface is shown in Figure 5.11, where the predictions of both models match the overall trend, but the suction-side pressure coefficient offset can be seen for DCNN. The average error in predicted pressure coefficient against airfoil surface computational coordinate ξ may be computed across all cases, and is shown in Figure 5.12. $\xi = 0$ at the trailing edge and walks along the airfoil surface in the clockwise direction, with the leading edge marked with a vertical dashed line. The suction side of the airfoil is to the right of the dashed line, and the over-prediction of c_p by the DCNN model is apparent. From this it also seen that the DVH predictions are subject to larger, high-frequency errors near the leading and trailing edges.

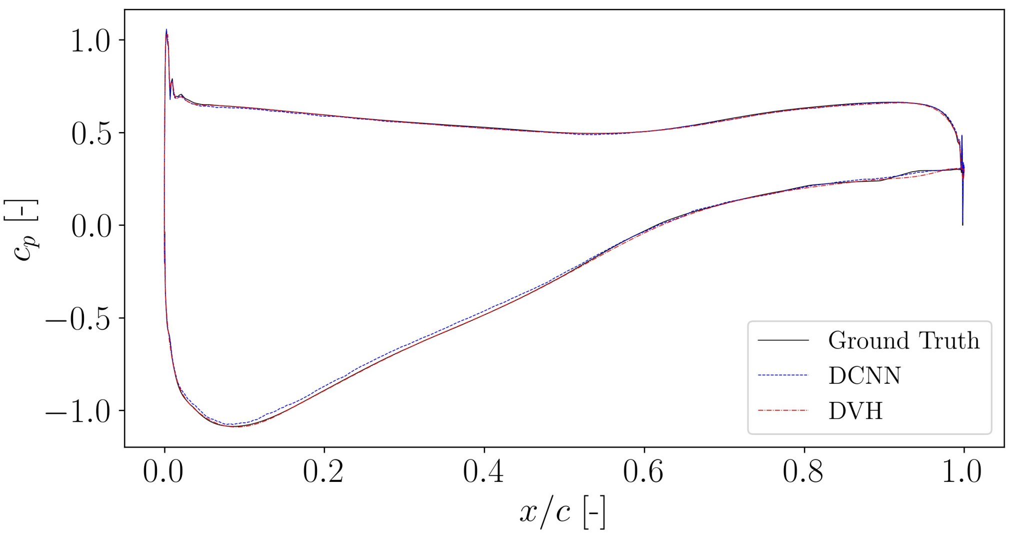


Figure 5.11: The predicted and ground truth pressure coefficient distributions for an unseen airfoil.

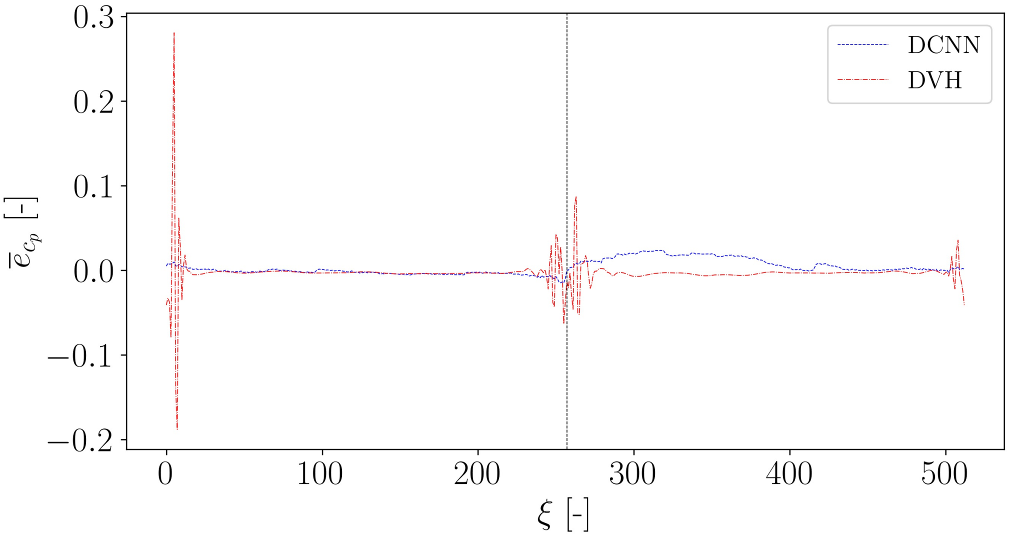


Figure 5.12: The average distribution in pressure coefficient prediction again surface computational domain coordinate ξ, which is 0 at the trailing edge and walks around the airfoil surface clockwise, with the leading edge marked by a vertical dash line in the plot.

5.3.1 Proof of Concept: Driving Design Optimization

A trained DCNN model from above was incorporated into an airfoil optimization workflow to assess the ability of the DCNN model to drive an aerodynamic optimization. A multi-objective optimization strategy was employed with the formulation of two objective functions from the emulated airfoil pressure distribution: (i) maximize airfoil loading, and (ii) minimize airfoil diffusion in the aft suction side of the airfoil. A differential evolution algorithm [243] is used and extended to multi-objective problems using a Pareto-based approach [244], using a similar nondominated sorting and ranking selection procedure as in NSGA-II [245]. The DCNN-driven optimization history and Pareto front are presented in the objective space in Figure 5.13. An optimum candidate is selected along the Pareto front, providing maximum airfoil loading while maintaining the diffusion level of the baseline airfoil. Figure 5.14 compares the loading of the baseline and optimized airfoils, as predicted by the DCNN model on the left and as validated by CFD on the right. This result highlights the ability of the DCNN flow emulator to drive an optimization towards increased level of loading not seen in the training data. While the DCNN model exhibits some discontinuities in the prediction of the loading distribution, relative differences in overall shape of the loading distributions appear well validated against CFD.

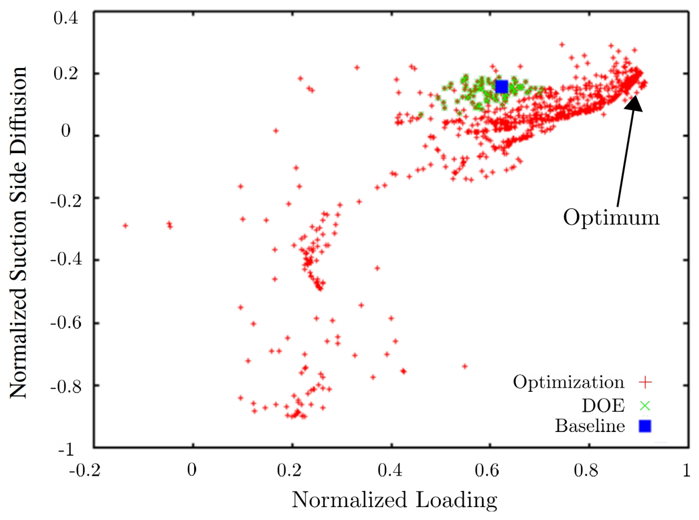


Figure 5.13: Objective space displaying baseline and outcome of DCNN-driven optimization

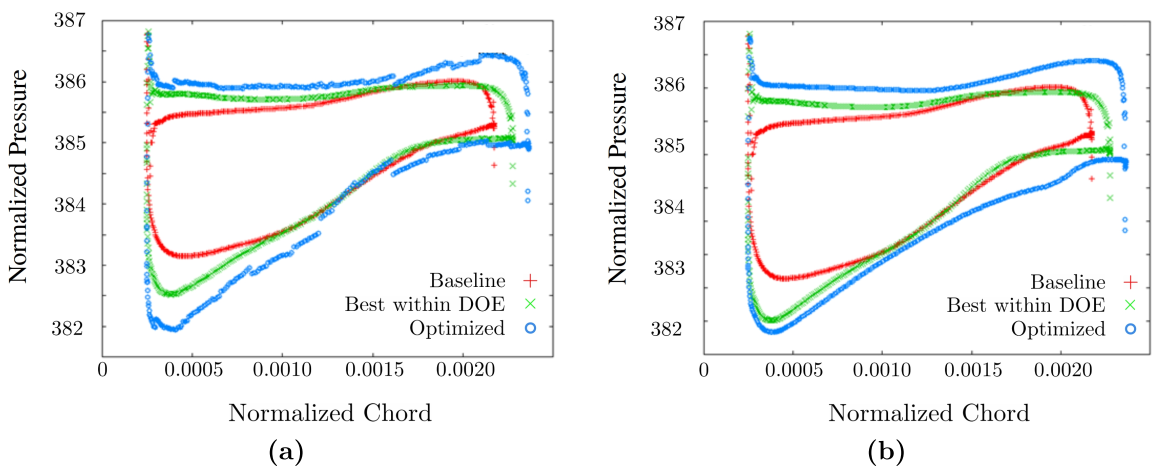


Figure 5.14: Airfoil loading as predicted by (a) DCNN surrogate model; and (b) as validated by CFD.

5.4 Transonic Compressor Airfoil Emulation and Optimization

5.4.1 Geometric Parameterization(s) and Datasets

The baseline design of the transonic airfoils considered corresponds to the NASA rotor 37 at 70% span. CFD solutions were generated under two scenarios and parameterizations, and all details related to the CFD solver from Section 5.2.1 apply. The first scenario is nominal rotor speed with $\boldsymbol{\mu} \in \mathbb{R}^7$, corresponding to 3 offset points along the thickness distribution and 4 offset points along the camber distribution. The results for this are given in Section 5.4.3. The next scenario is an expansion of the first, where the rotor speed is allowed to vary (and is an additional parameter), and an additional camber distribution offset point to control the metal inlet angle is added. This is presented in Section 5.4.4, whereg $\boldsymbol{\mu} \in \mathbb{R}^9$, and the rotor speed was allowed to vary $\pm 6\%$ from nominal. The motivation of this second parameterization is to assess the ability of the emulators to infer flow-fields for varying airfoil shapes *and* operating conditions, allowing shape optimization to be performed at different conditions using a single trained emulator.

5.4.2 Emulator Model Architectures

In both Sections 5.4.3 and Section 5.4.4 DCNN emulators with the architecture in Table 5.4 were used. The DVH models consist of a main network with 5 hidden layers with 100 nodes per layer, while the hypernetwork has 5 hidden layers with 50 nodes per layer.

Table 5.4: Multi-block DCNN architecture for the transonic airfoils.

Block	**Output Dimension**	**Kernel size** (Equiv. encoder)	**Strides** (Equiv. encoder)
1	357×41	$[6,2] \to [4,2] \to [2,2] \to [2,2] \to [2,2]$	$[1,1] \to [4,2] \to [2,2] \to [2,2] \to [2,1]$
2	265×47	$[2,2] \to [4,2] \to [3,3] \to [4,3]$	$[1,1] \to [4,2] \to [3,2] \to [2,2]$
3	66×17	$[3,2] \to [4,2] \to [2,2] \to [2,2]$	$[1,1] \to [4,2] \to [2,2] \to [2,1]$
4	38×17	$[3,2] \to [3,2] \to [2,2] \to [2,2]$	$[1,1] \to [3,2] \to [2,2] \to [1,1]$

5.4.3 Problem 1: Geometric Design Variables Only

A design-of-experiments was performed to select 1000 unique airfoil shapes in addition to the baseline design via latin hyper-cube sampling and flow solutions generated. Only converged solutions with CFD residual $< 10^{-6}$ were retained, corresponding to 702/1001 of the requested shapes. Two groups or clusters are observed in the dataset when plotting the QoIs of stagnation pressure ratio and adiabatic efficiency versus the design variables, with the desireable airfoils being massively under-represented. Figure 5.15 shows a pair-plot of the geometric design variables against the QoIs, where group 1 are the desireable airfoils. Just 36 group 1 instances are present, with 666 group 2 instances. This imbalance caused preliminary models to be biased towards the lower-performing group 2 airfoils, and the predictions on group 1 airfoils were unacceptably inaccurate for both DCNN and DVH models.

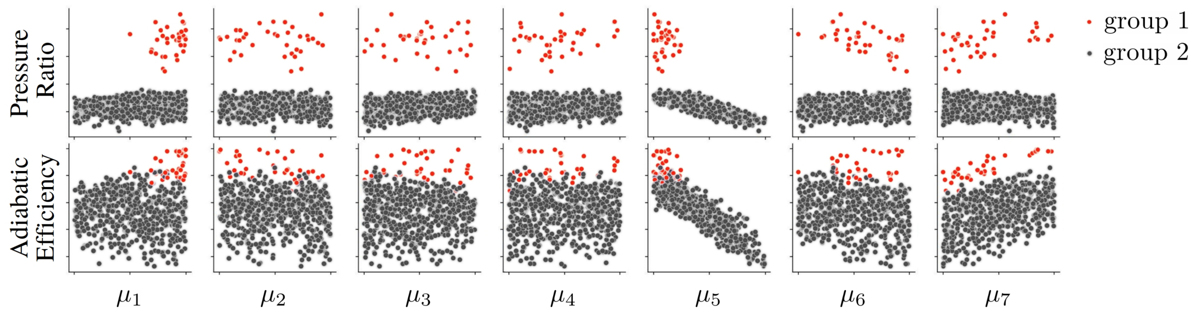


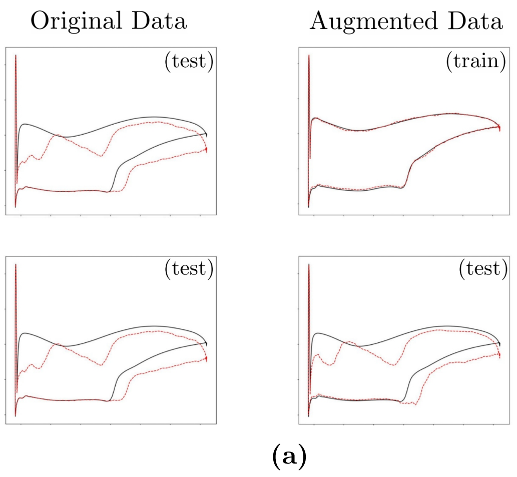
Figure 5.15: Pairplot of pressure ratio and adiabatic efficiency against the individual geometric design variables μ_i, with the higher-performing group 1 airfoils in red. Axis labels removed for proprietary reasons.

To address this issue, data augmentation was pursued. Traditional methods, such as translation, rotation, and scaling used in image-processing tasks [246], do not apply to this problem setting. Instead, the dataset imbalance was addressed by creating subsets which have an approximately equal number of group 1 and group 2 training instances by simply repeating the group 1 instances, similar to oversampling augmentations but without generation of synthetic cases [1]. 30 of the group 1 instances were repeated 4 times, and combined with 130 randomly selected group 2 instances. 6 group 1 cases are held out for testing.

With this parameterization, all models predict a 5-dimensional fluid state, $\mathbf{q} = \left[\rho\ p\ u\ v\ w\right]^T$, allowing computation of all necessary quantities through use of the ideal gas law and isentropic relations. For DCNN models, using the augmented dataset lead to improvements in group 1 cases which were explicitly added to the training group. However, some large errors persist in the testing cases. For DVH, more efficient batch-by-case training (see section 5.2.2.3) is used in combination with the augmented dataset, leading to massively improved predictions in an order of magnitude less time (relative to fully-mixed batching), for all training and testing cases. The training times reported in Table 5.1 correspond to this dataset.

Figure 5.16 shows a comparison of predicted and ground truth surface-pressure distributions for both model types, with and without use of the augmented dataset and efficient DVH training. For each model, the left column corresponds to the original imbalanced

[1]If synthetic cases could be generated, then the CFD emulator would not be needed and that generator used instead.



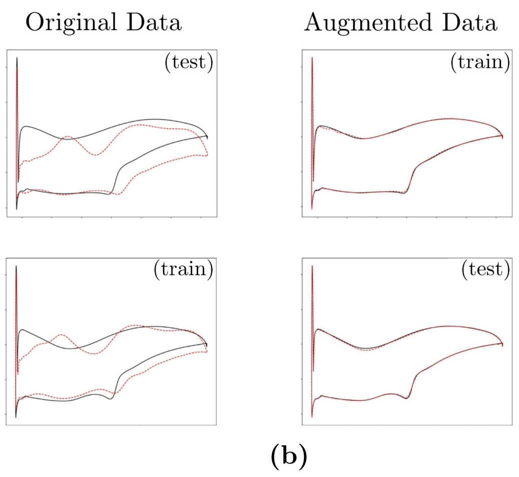
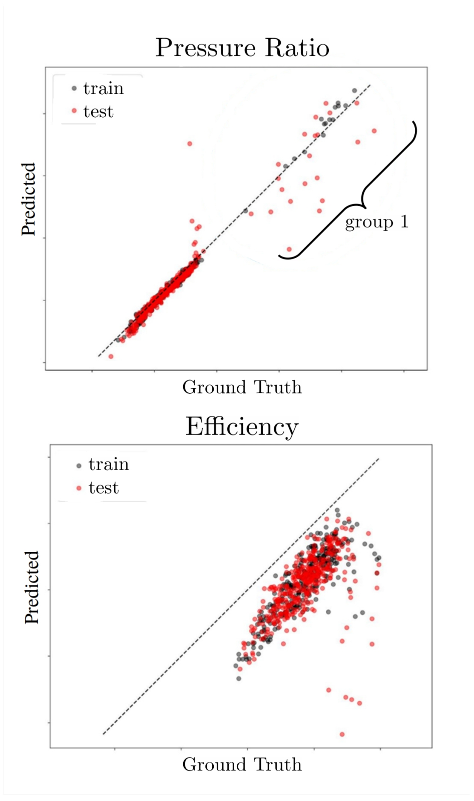


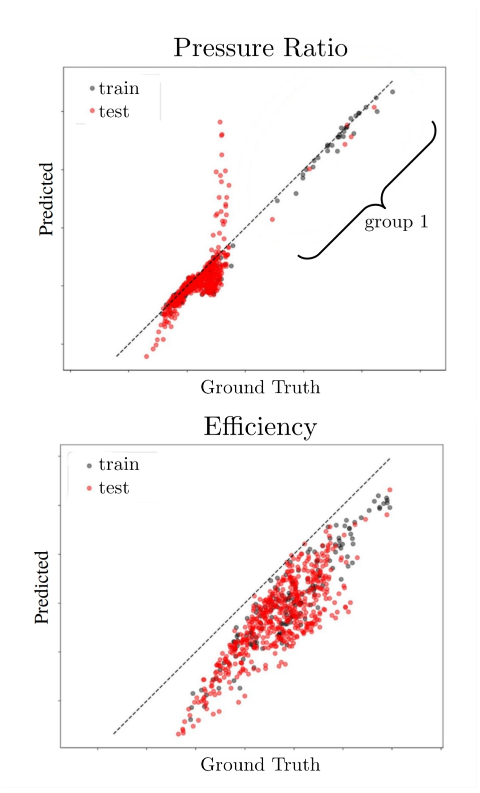
Figure 5.16: A comparison of the effect of training with the augmented dataset on group 1 airfoil surface pressure predictions for (a) DCNN and (b) DVH models, where the ground-truth is solid black and the predictions are dashed red. For each emulator, each row corresponds to the same airfoil, with a label marking it as part of the training or testing set. Axis labels removed for proprietary reasons.

dataset, while the right column corresponds to the augmented dataset. Each row corresponds to the same airfoil shape (for each model) with a note indicating whether the solution is in the training or testing set. For DCNN, the top row of Figure 5.16a shows that the prediction is greatly improved when the case is added to the training set. The bottom row shows that large errors persist for some cases which remain in the testing group. For DVH, the top row of Figure 5.16b again shows that the prediction is greatly improved when the case is added to the training set. However, unlike DCNN, it was found that the group 1 testing cases were all well predicted when augmented data was used. This is exemplified in the bottom row of 5.16b, where the prediction appears greatly improved with use of augmented data and efficient training, despite the specific airfoil moving from the training to testing group. Since the predictions on group 1 airfoils are of greater importance for shape optimization, only the DVH models are pursued for use in such a routine.

The effect of using the augmented data is explored further by plotting the predicted vs. ground truth QoIs for the DVH predictions with and without the augmentation in Figure 5.17. Observing the top row for pressure ratio shows that the effect of the augmented data is to trade accuracy on group 2 cases for increased accuracy on group 1 cases. The data augmentation itself can be seen by observing the number of higher-performing, group 1 cases in the training set (black markers). Without the augmentation, the group 2 cases cluster along the dashed line, while the group 1 cases are scattered. With the augmented data, the group 2 predictions suffer while the group 1 predictions now cluster more tightly along the dashed line. An increased number of group 2 airfoils are over-predicted in this scenario, but this is deemed a worthwhile trade for increased accuracy on the more desirable, higher-performing group 1 airfoils. Although this does open the possibility of discovering a false minima due to over-predicted performance, but this was not problematic here.



(a)



(b)

Figure 5.17: Predicted QoI's, (a) without the data augmentation and (b) with the data augmentation. Using data augmentation hurts group 2 predictions while improving those for desirable group 1 airfoils. Axis labels removed for proprietary reasons.

The overall ASO problem as stated in Equation 5.6 is solved using the Bayesian optimization routine described in Section 5.2.1, using UTCFD RANS solutions and the DVH flow emulator above. The routine utilizes CFD-residual information, which was supplied for the demonstration here, but later a separate CFD-residual XGBoost QoI emulator was trained offline using the training DoE. A comparison of emulator-driven and CFD-driven optimization is shown in Figure 5.18, where emulator-driven optimization reaches the same optimal design as CFD in fewer iterations. The use of residual information was key in achieving success with the emulator, as the residual information may be interpreted as a measure of feasibility, and thus steers the overall optimization process.

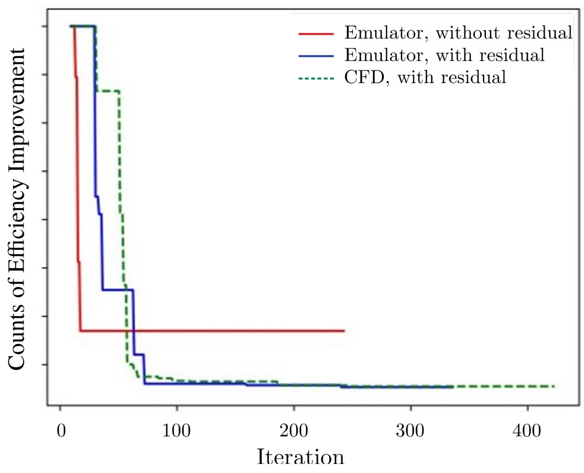


Figure 5.18: Comparing CFD-driven (green) and emulator-driven shape optimization with (blue) and without (red) CFD residual information. The iteration count excludes initial DOE of 10 cases. Axis labels removed for proprietary reasons.

5.4.4 Problem 2: Geometric Design Variables and Rotor Speed

A larger training DoE of 10,000 cases was generated, and inclusion of rotor speed allows for a much wider variety of resulting flows. No obvious clusters were seen in QoI versus design-variable space. A greater number of flow quantities is predicted, $\mathbf{q} = \left[\rho \;\, p \;\, u \;\, v \;\, w \;\, k \;\, \omega \;\, E_0\rho\right]^T$. This allows for an alternate method for computing the QoIs, but is not explored here further. DCNN and DVH models were regressed on the dataset, with DVH models converging more quickly and readily using subsets of the overall DoE. Observing the training curves showed that as the training loss plateaued the loss bounced around with large spikes, indicating repeated overshooting caused by too large a step size or learning rate. Thus a piecewise learning rate schedule was used, where the models were trained with a constant learning rate of 1×10^{-4} for 2000 epochs, followed by a period of exponentially decaying learning rate for 2000 epochs, where the learning rate decreased by an order of magnitude every 1000 epochs. The resulting training curve is shown in Figure 5.19, where 800 randomly chosen cases are used for training.

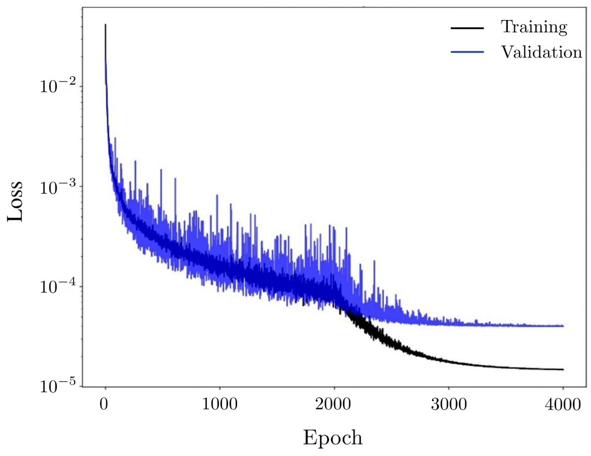


Figure 5.19: Training curve for DVH model with a piecewise learning rate schedule, consisting of two regions, with constant learning rate followed by exponentially-decaying learning rate, starting at epoch 2000.

The predicted flow fields agree well with the ground truth data, with surface-pressure predictions around unseen airfoils at varying rotor speed shown in Figure 5.20. The top-left is similar to group 2 airfoils in the original dataset, top-center is similar to group 1, while the others show vastly different behavior. Example full flow-field predictions are shown in 5.21 for an unseen airfoil. The overall detailed structure of the field is well captured, with the largest errors seen near the shock waves.

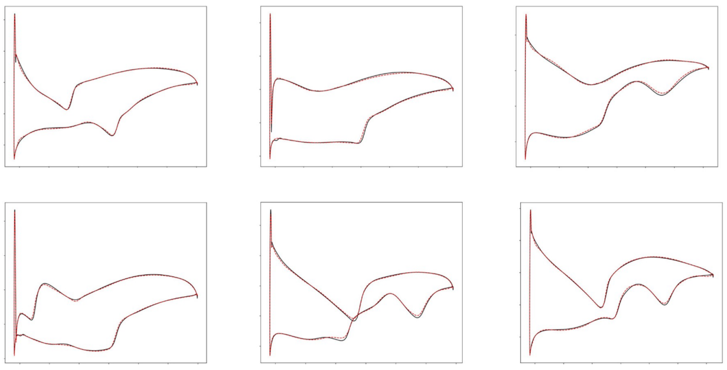


Figure 5.20: Ground truth (black) and predicted (red) surface-pressure distributions around several unseen airfoils. A wide variety of behavior is seen, and is captured well by the DVH emulator. Axis labels removed for proprietary reasons.

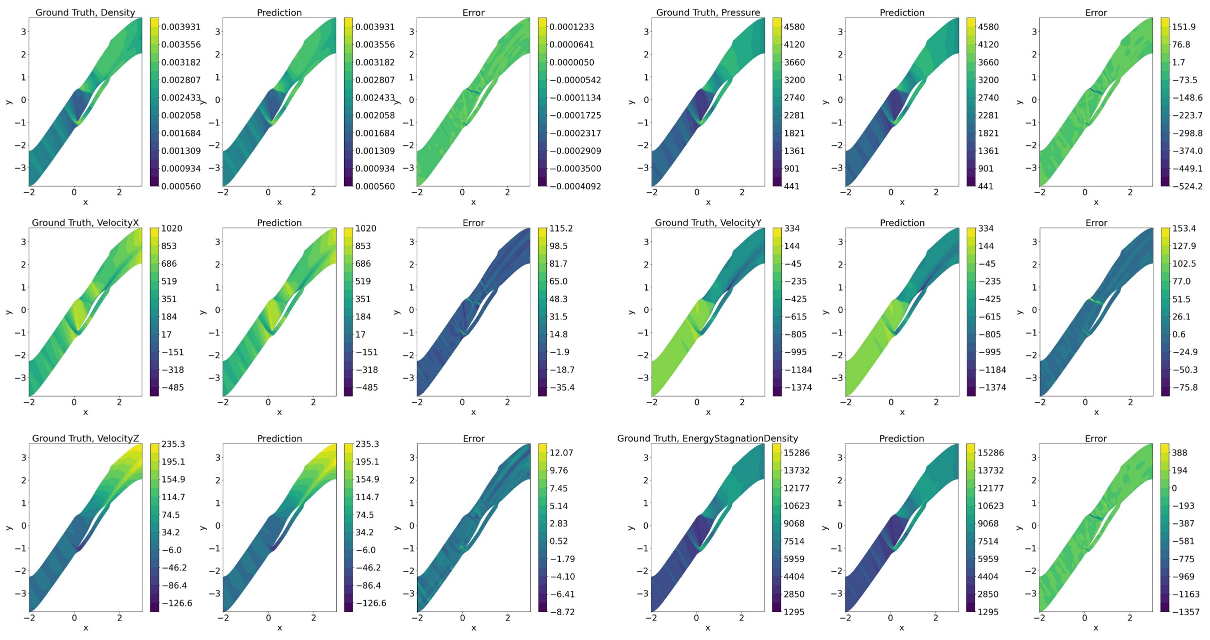


Figure 5.21: Contours plots showing the ground truth, prediction, and error for some of the DVH predicted flow quantities for an unseen case.

Additionally the QoIs computed from the emulator predictions show good agreement, as shown in Figure 5.22 across all 10,000 samples, where a perfect prediction would lie exactly on the dashed line. The color represents rotor speed, and most visible markers are testing cases.

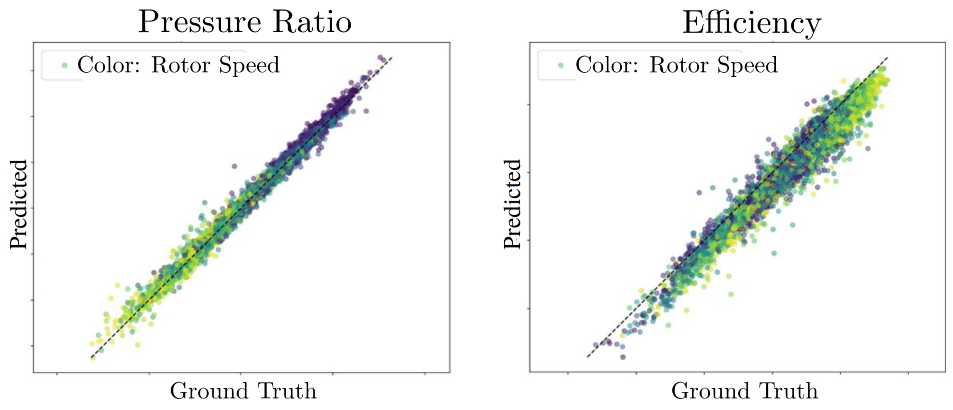


Figure 5.22: The predicted vs. ground truth objectives, colored by rotor speed. Axis labels removed for proprietary reasons.

The DVH emulator was then used to drive shape optimization at different rotor speeds; under nominal conditions and at ±4% rotation rate. The Bayesian optimization routine described in Section 5.2.1 is used again here, where CFD and CFD-emulator are used separately to train Gaussian process QoI emulators, with the emulator-driven routine using an XGBoost residual model. The relative improvement for emulator-driven and CFD-driven optimization is shown in the objective-history curves of Figure 5.23, where 5.23a and 5.23b use different scaling of the x-axis. Compared to CFD-based Bayesian Optimization (showing statistical average over 20 runs), the emulation-driven optimization (showing one run) is observed capable to identify high-performing designs for the three rotor rotational speeds considered. In Figure 5.23a, both history traces are plotted against the raw number of QoI-emulator evaluations, showing that the emulator-driven routines generally converge in a fewer number of evaluations. The reason for this is not immediately clear and warrants further investigation. Whereas Figure 5.23b shows the same objective history now plotted against time, expressed in terms of equivalent CFD evaluations after the initial training DoE. In this view the emulator-driven optimization converges and plateaus almost immediately, representing a large decrease in online time to find the optimal design. A performance-space view comparing CFD-driven and emulator-driven histories is shown in Figure 5.24. The difference in convergence speed between CFD-driven and emulator-driven routines manifests itself as decreased dot density using the emulator.

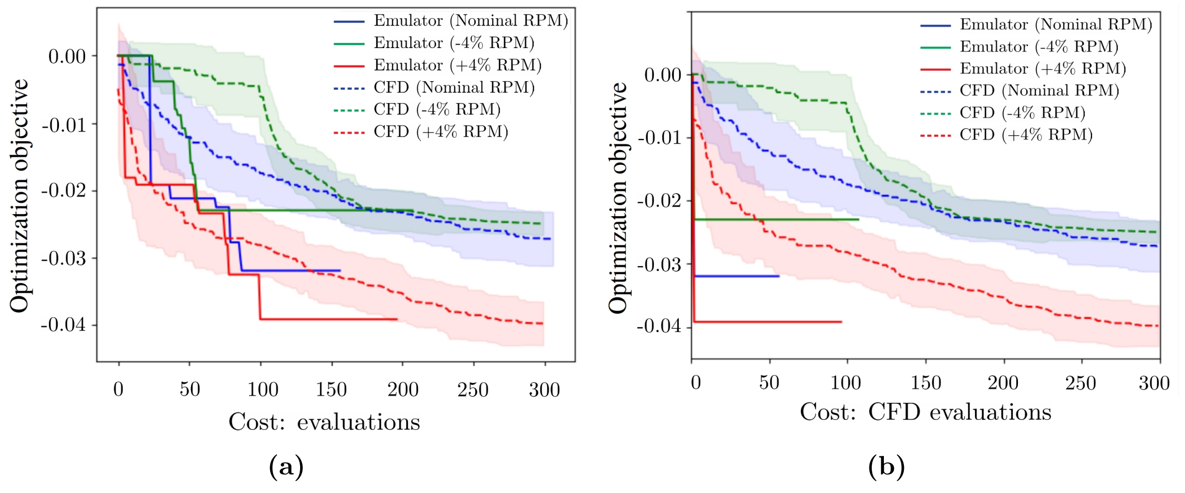


Figure 5.23: Comparing CFD-driven and emulator-driven optimization objective histories for the design of a transonic airfoil at varying rotor speed, where the objective is plotted against (a) the number of evaluations or iterations, and (b) the number of CFD evaluations.

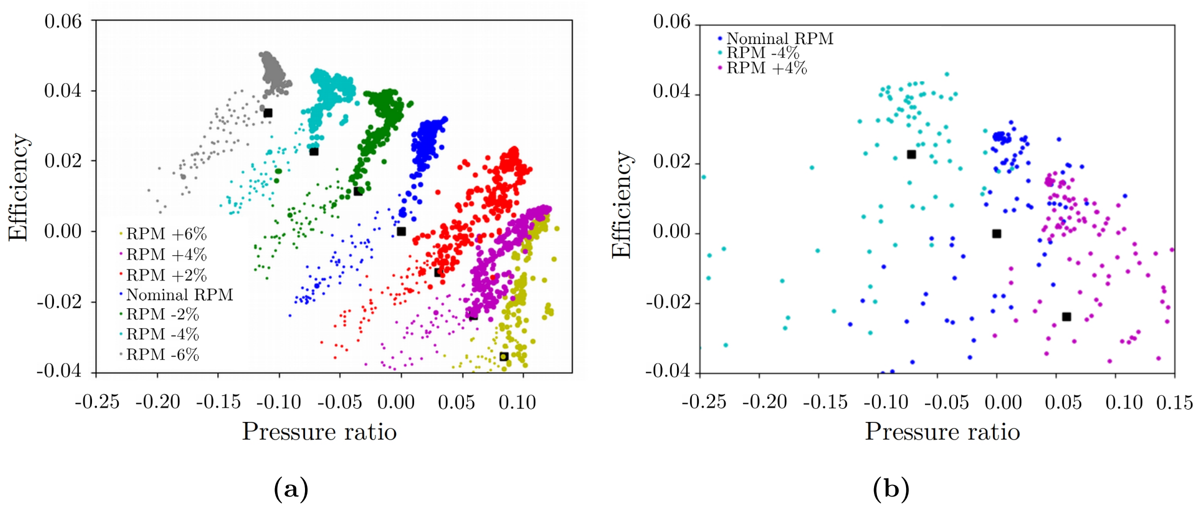


Figure 5.24: Performance-space view of (a) CFD-driven surrogate assisted optimization and (b) CFD-emulator-driven surrogate assisted optimization, where the black squares correspond to the baseline design.

While Figure 5.23 highlights the online saving afforded by the CFD emulators, it is also important to consider the offline costs and an overall break-even point as compared to CFD-driven optimization when evaluating the use of such models. For either method, the overall cost includes the generation of a training dataset, the cost to train the model, either the initial QoI surrogate or the CFD emulator, and then the online costs per iteration. This may be written generally as

$$\mathcal{C}_{\text{overall}} = \mathcal{C}_{\text{solve}} n_{\text{train}} + \mathcal{C}_{\text{train}} + \mathcal{C}_{\text{iteration}} n_{\text{iterations}}, \tag{5.13}$$

where $\mathcal{C}_{\text{solve}}$ is the cost for a single CFD solution. Usually n_{train} is much smaller when considering CFD-driven optimization as compared to that required to train a CFD emulator; a small DoE is used for CFD-driven optimization. Further, $\mathcal{C}_{\text{train}}$ is much smaller for CFD-driven optimization as it only involves training the QoI surrogate, whereas emulator-driven optimization also includes the cost of neural-network training on GPU. Thus the offline costs for emulator-driven optimization are generally much larger than that for CFD-driven. This is offset by the lesser online costs, where CFD-driven optimization requires a CFD-solve for each iteration while emulator-driven optimization does not involve generation of CFD solutions once the model is trained. [2] Thus determination of an

[2]Other than a CFD-based verification of the optimal design.

overall break-even point must weigh the greater cost of attaining a CFD emulator against the greater cost per iteration using CFD-driven optimization. A CFD emulator which is more general and which may be reused to solve for a different operating condition (or design problem) without additional training is more likely to have an overall lower cost than one which can only be used for a single optimization.

5.5 Summary

Simulation-based Aerodynamic Shape Optimization plays an increasingly important role in the design of aircraft and engine components, but are limited by the time required to compute high fidelity CFD solutions. Deep-learning-based *CFD emulators*, which predict full RANS CFD solutions instead of QoI's, were developed and demonstrated for subsonic and transonic compressor-rotor sections. Critically, the methods do not require use of interpolated data, either by generating snapshots directly in the computational domain (DCNN), or by operating continuously in the physical domain (DVH). While 2D sections are presently considered, DVH scales naturally to 3D problems while 3D CNNs quickly become memory limited.

Although both methods were effective in the prediction of subsonic flows (Section 5.3), DVH models were found to perform better than DCNN models under transonic conditions (Section 5.4). Dataset augmentation via simple repetition of under-represented cases allowed DVH models to generalize reasonably well, trading increased accuracy on higher-performing cases for worsened predictions on lower-performing designs, while DCNN models still struggled (comparatively) in prediction of the higher-performing, underrepresented cases. When varying flow conditions via altered rotor speed were considered in addition to geometric shape parameters, DVH models converged and generalize more readily than DCNN models, especially when trained using a piecewise learning-rate schedule. In that scenario the wide variation in the resulting complex flows were well captured, and as a result the extracted QoIs of pressure ratio and efficiency were also well predicted. Further, DVH emulators were used in place of CFD in a surrogate-based

Bayesian optimization routine, where the emulator-based surrogates reached similarly-performing designs in a reduced number of model evaluations. This corresponds to an acceleration of the online stages of design optimization, but does not consider the additional off-line computational cost of training the emulators. However, we note that once the emulator is trained once off-line, it can be used on-line to conduct many different design optimizations - e.g. with different objective functions, constraints and tradeoffs.

Chapter 6

Conclusions

The computational cost of numerically solving high-fidelity models (HFMs) limits their application in engineering-relevant many-query scenarios such as design optimization, with each iteration requiring a solution for the current design. Data-driven approaches offer an appealing alternative in these scenarios, but many methods face practical shortcomings related to the processing of solution data. In particular, powerful non-intrusive methods based on Proper Orthogonal Decomposition (POD), Gaussian Process Regression (GPR), and Convolutional Neural Networks (CNNs) use discrete snapshot matrices, requiring all data points to lie on the same meshes with consistent dimension and topology; an unrealistic requirement in the presence of variable, complex geometry and operating conditions. In order to apply those methods, a Cartesian mesh may be overlain on the problem domain and the solution data interpolated to that mesh, as was demonstrated in Sections 2.3.2 and 2.3.3. However, this has a number of undesirable effects, including a reduced fidelity, pixelated representation of the geometry and a loss of information in regions with large solution gradients, sometimes an extreme effect. Another way around this is to instead approximate scalar quantities of interest (QoIs) which are extracted from the HFM solutions. These QoI surrogate are generally easier to create, in part to the reduced problem dimensionality, but have limited portability when different design problems or objectives are considered.

Optimization lies at the heart of iterative design and surrogate-model regression. Gradient-based and gradient-free constrained optimization routines are applied directly to design optimization, with HFM solutions used to compute the objectives and constraints.

Classical linear and non-linear least-squares techniques pose model construction as the solution to an optimization problem, often-times with a closed-form result. Several POD-based methods utilize the singular-value decomposition (SVD) to obtain the rank-r trial and test bases, and it may be shown that the SVD provides optimal rank-r reconstruction of the training-data snapshot matrices in a Frobenius-norm sense. GPR lies at the heart of Bayesian optimization routines, where an acquisition function, usually dependent upon the GPR mean-function and covariance kernel, is optimized at each iteration to select the next design. Further, training of advanced deep-learning models is usually carried out using a form of stochastic gradient descent (SGD), an unconstrained optimization routine. While POD-based methods in particular have been used in the development of reduced-order-models (ROMs) which provide full-solution fields, they are limited by the need to construct a snapshot matrix to hold the data for all training instances. This is also true for naive multi-output GPR and many of the most popular deep learning techniques which are based on autoencoders and/or convolutional architectures. Many of these methods are effective for data reconstruction and prediction, but the processing schemes required to place the solution data on consistent meshes severely limit their effectiveness for application in problems with industrial-scale complexity. Further, POD-based methods also face limitations due to the implicit assumption that the training snapshot matrix spans all relevant dynamics, which is generally not true in practice, especially for convection-dominated problems.

6.1 Summary of Contributions

The past few years have witnessed significant activity in the use of neural networks to develop surrogate representations of physical fields (e.g. [75, 76, 80]) on a given discretized mesh, which still involve the shortcomings given above. Whereas this thesis seeks the development of surrogate models without these interpolation-related shortcomings by treating the solutions as *continuous fields*, allowing learning and prediction on meshes with arbitrary discretization, topology, and geometry.

6.1.1 Methods

As a stepping-stone in this direction, DCNN models were developed to map directly between the design variables and the solution fields in Section 3.1. DCNNs were conceived in the context of convolutional autoencoders and are applied directly in the computational domain to avoid lossy interpolation. They posit that the design variables $\boldsymbol{\mu}$ may be used directly as input to the decoder instead of a learned latent representation. This means that the encoder is no longer needed, resulting in a simpler model with roughly half the parameters of a corresponding autoencoder. DCNN models use transposed convolutional layers, and as such are limited to problems which have a Cartesian or block-Cartesian structure in the computational domain.

Full discretization independence is achieved through use of coordinate-based neural networks which map between low dimensional spaces instead of snapshots, inspired by a line of research involving scene and object representation for rendering tasks [158, 159, 168, 161, 162]. The key distinction is that network inputs and outputs are taken *point-wise*, for example as a physical coordinate tuple, resulting in a continuous representation of the fields. Accounting for variation in physical design and operating conditions is achieved through a combination of *local* and *global* conditioning. First, an evaluation of the signed or minimum distance function is included as a main-network input along with the physical coordinates. This represents a form of local concatenation-based conditioning as the SDF/MDF are functions of space and generally vary along with the physical coordinates. All spatially varying quantities are collected in the augmented spatial coordinates, denoted by $\mathbf{x}'$. The developed methods are distinguished by the form of global conditioning used, with the design variables used as the global-conditional input as they are not functions of space and apply to entire solution instances. Design-variable MLP (DV-MLP) utilizes input-layer concatenation-based conditioning, where the design variables are concatenated with the augmented physical coordinates and used as network inputs; see Section 3.2.4. Whereas design-variable hypernetworks (DVH) of Section 3.2.5 condition all of the main-network weights upon the design variables by generating them with a hypernetwork. When only the weights of the output layer are generated via hypernetwork, then the resulting model predictions may be placed in a form analogous to POD reconstruction, with that method known as non-linear independent dual system (NIDS); see Section 3.2.6. DVH model predictions may also be loosely interpreted in this manner. In practice it was observed that DV-MLP and DVH performed better than NIDS and as such were more widely applied to the given problems.

The DVH models considered use a one-shot hypernetwork, whereby all weights are generated at once via one forward pass of the hypernetwork. A side effect of this is that a DVH model using dense layers in the hypernetwork has many more parameters for a given main network than a similar DV-MLP model. However, the development and utilization of batch-by-case training for DVH models significantly reduced the computational cost associated with training; see Sections 3.2.5.1-3.2.5.2. This approach led to a substantial decrease in step times and memory consumption, approximately by an order of magnitude, compared to fully-mixed training when considering a backend precision policy, although improvements also depends upon the spatial batch size. This enabled DVH models with significantly more parameters to be trained in a comparable amount of time as DV-MLP models. Further, a piecewise learning-rate schedule with periods of constant and exponentially-decaying learning rates was devised and implemented which improved model convergence, see Section 3.4.

6.1.2 Numerical Experiments

The models were first evaluated on a 2D Poisson problem defined with varying shapes embedded in a square domain with a spatially-distributed source term in Section 4.2. Each solution instance has a different mesh with unique cardinality and topology. The shapes location within the domain was varied to asses translational effects, while the shapes size was varied to asses scaling effects. It was found that inclusion of a class-label vector in addition to scaling and locating information in the design variables greatly enhanced predictive accuracy for all models. Additionally, errors were observed to vary with the shape class or number of sides, generally decreasing as the number of sides increased. All models performed well with training- and validation-group MRL2Es less than 0.1%, but with DVH and DV-MLP performing even better with MRL2Es less than 0.01%.

The effectiveness of DVH and DV-MLP was also evaluated in the context of a challenging 2D vehicle aerodynamics problem, utilizing realistic vehicle shapes with solutions lying on unstructured meshes of variable topology. Baseline results indicated that DVH consistently outperformed DV-MLP by a few percentage points, while also demonstrating superior convergence and generalization properties in the low-data regime. By incorporating a random-Fourier-features layer to process the spatially-varying inputs $\mathbf{x}'$, the RMSEs were significantly reduced by roughly 20-60% with greater improvements seen for DV-MLP as compared to DVH. In this scenario DV-MLP results were improved so substantially that the performance gap between DVH and DV-MLP was essentially closed, including the convergence and generalization properties in the low-data regime. Both DVH and DV-MLP exhibited strong generalization capabilities when multiple vehicle speeds were considered, with minimal disparities between training and validation errors. The pressure fields errors were near 2%, x-velocity errors around 1%, and y-velocity errors around 6-7%. The pressure-drag coefficients were predicted within 2% by DVH models, for both single and multiple speeds. The main features of the flow fields were well-captured, with the largest errors often clustering in regions close to the vehicle, in the fine details of the grill, and along the free-shear layer of the wake. Line probes showed some oscillation in the network predictions compared to the ground truth, consistent with discrepancies seen in the contour levels. Several researchers have noted the effectiveness of positional encoding techniques in a variety of problem scenarios [161, 189, 247]. The success of the techniques may be explained using Neural Tangent Kernel theory [248], where the use of random Fourier features results in a stationary, shift-invariant kernel with a tunable bandwidth controlled by sampled frequency vectors $\mathbf{b}_i$ [187]. However, this result is in the context of dense, nearly-uniform sampling of input coordinates without an additional signed-distance input, and further study is warranted in the current scenario with unstructured, non-uniform meshes.

DVH and DV-MLP models were also applied to a 3D Ahmed body problem, with geometric variation in the rear slant angle. Under this scenario it was observed that DV-MLP performed best, particularly when Layer Normalization layers were added between each layer. The pressure drag coefficients were predicted to within 1.74% by DV-MLP-LN in this scenario. DVH models did not perform well with the addition of Layer Normalization in the main network. However, Layer Normalization includes scaling and biasing vectors as trainable parameters, and these parameters were not included in the hypernetwork output, and instead were left as part of the main network for all cases. It is possible including those vectors in the hypernetwork output may improve the performance, but this was not attempted. The use of Fourier features did not improve the performance in this scenario, and was worse for all values of m and σ attempted for all model types. This is unfortunate, as hyperparameter tuning or parameter sweeps with this larger dataset will require non-trivial computational resources and time.

DCNN and DVH models were also applied to emulation of 2D subsonic and transonic compressor-airfoil flows lying on multi-block meshes with a Cartesian structure in the computational domain in Chapter 5 . In the subsonic regime DCNN and DVH performed similarly but with DVH having greater accuracy in the predicted lift and drag coefficients. A DCNN model was used to drive aerodynamic shape optimization in the subsonic regime as a proof-of-concept. Although both methods were effective in the prediction of subsonic flows (Section 5.3), DVH models were found to perform better than DCNN models under transonic conditions (Section 5.4). Dataset augmentation via simple repetition of under-represented cases allowed DVH models to generalize reasonably well, trading increased accuracy on higher-performing cases for worsened predictions on lower-performing designs, while DCNN models still struggled (comparatively) in prediction of the higher-performing, underrepresented cases. When varying flow conditions via altered rotor speed were considered in addition to geometric shape parameters, DVH models converged and generalize more readily than DCNN models, especially when trained using a piecewise learning-rate schedule. In that scenario the wide variation in the resulting complex flows were well captured, and as a result the extracted QoIs of pressure ratio and efficiency were also well predicted. Further, DVH emulators were used in place of CFD in a surrogate-based Bayesian optimization routine, where the emulator-based surrogates reached similarly-performing designs in a reduced number of model evaluations. This corresponds to an acceleration of the online stages of design optimization, but does not consider the additional off-line computational cost of training the emulators. However, we note that once the emulator is trained once off-line, it can be used on-line to conduct many different design optimizations - e.g. with different objective functions, constraints and tradeoffs.

The results suggest that both DVH and DV-MLP models can be accurate and effective alternatives to CNN-based methods for surrogate modeling of PDE solution fields over complex geometries and arbitrary mesh topologies. It is re-emphasized that CNNs typically require a fixed grid topology and have a large memory footprint for 3D problems since the entire grid is an input. In contrast, coordinate-based networks take pointwise information and design variables as inputs, allowing model size and memory requirements to be indirectly decoupled to the solution field degrees-of-freedom via the field complexity. This is critical for 3D applications and simulations of realistic configurations where solutions may contains tens-of-millions to billions of mesh cells.

The use of the design-variables $\boldsymbol{\mu}$ as model input is an advantage in terms of ease and simplicity, but is also a drawback as it limits model portability and training-data sources. The use of computational methods such as CFD and FEA have become commonplace in industrial settings over the last few decades, and as a result many companies may have large repositories of previously-performed simulations. Learning from heterogeneous data sources opens the possibility of utilizing these repositories to train data-driven models or generalized emulators which predict solution fields. However, the dependence upon the design variables $\boldsymbol{\mu}$ as model input complicates this, as it is very unlikely that all previous simulations utilize a consistent parameterization. In order for repositories of heterogeneous simulation data to be most useful then additional steps must be made to encode the problems into a generalized representation to use in place of $\boldsymbol{\mu}$. Thus an area of future work involves devising and implementing methods which encode solutions in a generalized manner, and linking them to predictive coordinate-based models. It is possible that point cloud neural networks, graph neural networks, and attention-based networks may be useful in such a scheme.

6.2 Future Work

Future work in this area should include further experiments with 3D problems and differing PDE solutions. While the Ahmed body results obtained here are promising, it is a simpler geometry and configuration than that seen in industrial automotive aerodynamics. Given the positive results with 2D subsonic and transonic compressor airfoils seen here, extending the models to predict 3D flows over full compressor blades is a natural next step. Further, this thesis tackled only steady-state problems, and many important engineering considerations involve unsteadiness and time-dependence. The simplest approach to handling such problems would be to simply include t as an element of $\boldsymbol{\mu}$ with no further modifications to the model's architecture, similar to what is done in neural implicit flow [172] which achieved excellent performance regressing spatio-temporal data. However, the problems neural implicit flow was tested on did not include geometric design variables in the parameter set, and also did not include the SDF as an additional input coordinate. It is possible that geometric and temporal variations may need to be treated separately, potentially using alternate conditioning schemes instead of or in addition to networks or subnetworks which utilize a hypernetwork; see Section 3.2.2.

The use of alternate hypernetwork architectures beyond one-shot dense networks is another area of potential future research. This is important due to the scaling considerations and limitations of dense hypernetworks described in Section 3.2.5. All results in this thesis use a one-shot dense hypernetwork, some experiments were performed using one-shot convolutional decoder hypernetworks. Note that the main network is still an MLP in this scenario, and to ease design of the hypernetwork consider a main network with n_L hidden layers each with dimension H which is a power of 2. The decoder hypernetworks are similar in structure to multiblock DCNN models with parallel decoder legs, with a single dense layer between the input layer and the first transposed convolutional layer, with the dimension(s) of the first transposed convolutional layer selected to be D_{min} along each axis, where D_{min} is also a smaller power of 2. The resulting decoder hypernetwork then has 3 zones;

- Zone 1 for the first hidden-layer weights which uses 1D transposed convolutions. The output space has dimension H with $n_{x'}+1$ channels. This is sliced and reshaped to give weight matrix $\mathbf{W}^{(1)} \in \mathbb{R}^{H \times n_{x'}}$ and bias vector $\mathbf{b}^{(1)} \in \mathbb{R}^{H}$.
- Zone 2 for all other hidden-layer weights uses 2D transposed convolutions. The output space has dimension $H \times (H+1)$ with $n_L - 1$ channels. This is sliced and reshaped to give weight matrices and bias vectors $\mathbf{W}^{(i)} \in \mathbb{R}^{H \times H}$, $\mathbf{b}^{(i)} \in \mathbb{R}^{H}$, for $i = 2, 3, \ldots, n_L$.
- Zone 3 for output layer weights uses 1D transposed convolutions. The output space has dimension $H+1$ with n_q channels. This is sliced and reshaped to give weight matrix $\mathbf{W}^{(n_L+1)} \in \mathbb{R}^{n_q \times H}$ and bias vector $\mathbf{b}^{(n_L+1)} \in \mathbb{R}^{n_q}$.

The hidden dimension of the hypernetwork intermediate layers progresses by powers of 2, by selecting $P = 0$, $S = 2$, $K = 2$ in Equation 2.64, which simplifies $D_j^{(i)} = 2D_j^{(i-1)}$. To attain dimension $H+1$ along needed axes, let $P = 0$, $S = 1$, $K = 2$, giving $D_j^{(i)} = D_j^{(i-1)} + 1$. And finally, to let an axis not change dimension (as is needed in zone 2), select $P = 0$, $S = 1$, $K = 1$ along that axis. For example, consider a main network with $H = 32$ and $n_L = 4$ hidden layers. Further let $D_{\text{min}}^{(1)} = 8$ and let F be the number of filters for a given layer. Then a decoder convolutional hypernetwork will have a structure as shown in Figure 6.1.

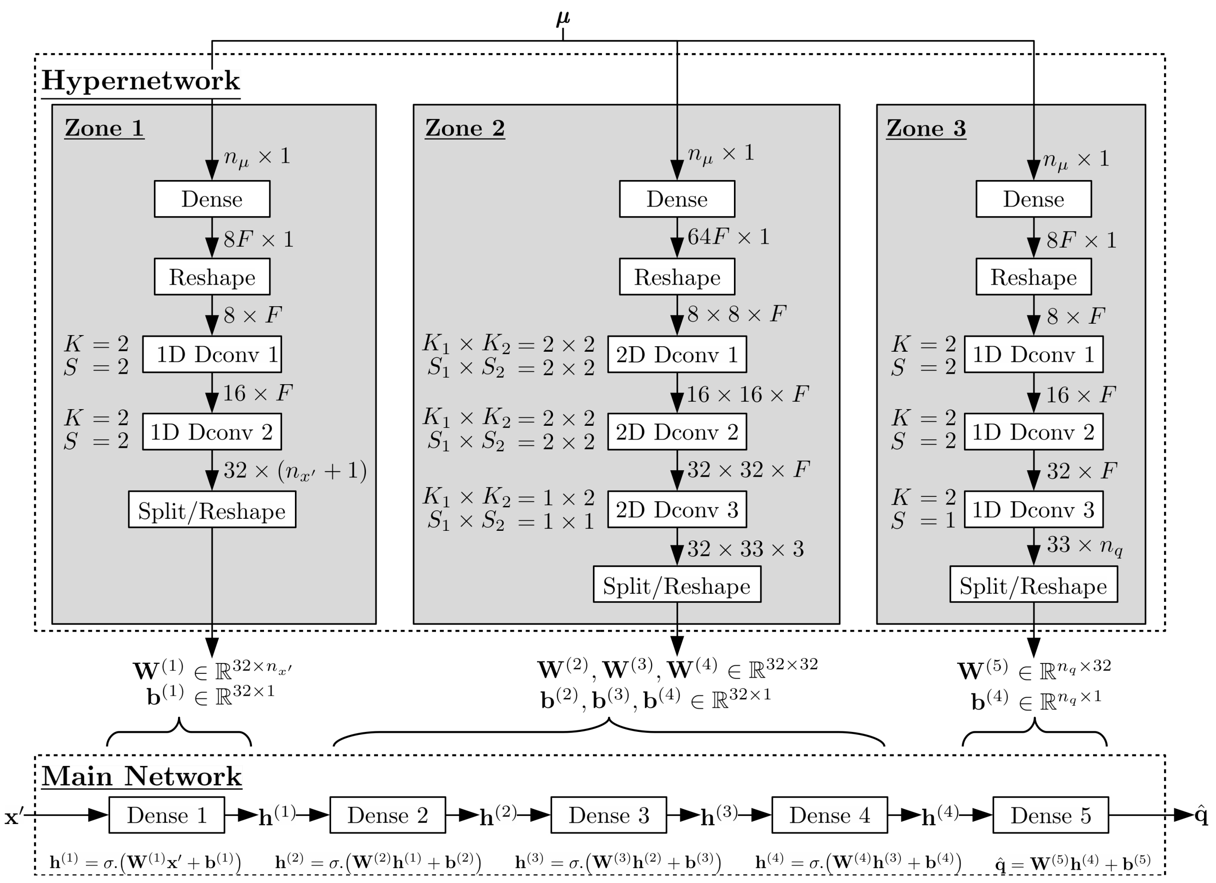

$\boldsymbol{\mu}$
Hypernetwork
Zone 1
$n_\mu \times 1$
Dense
$8F \times 1$
Reshape
$8 \times F$
$K = 2$, $S = 2$
1D Dconv 1
$16 \times F$
$K = 2$, $S = 2$
1D Dconv 2
$32 \times (n_{x'} + 1)$
Split/Reshape
Zone 2
$n_\mu \times 1$
Dense
$64F \times 1$
Reshape
$8 \times 8 \times F$
$K_1 \times K_2 = 2 \times 2$, $S_1 \times S_2 = 2 \times 2$
2D Dconv 1
$16 \times 16 \times F$
$K_1 \times K_2 = 2 \times 2$, $S_1 \times S_2 = 2 \times 2$
2D Dconv 2
$32 \times 32 \times F$
$K_1 \times K_2 = 1 \times 2$, $S_1 \times S_2 = 1 \times 1$
2D Dconv 3
$32 \times 33 \times 3$
Split/Reshape
Zone 3
$n_\mu \times 1$
Dense
$8F \times 1$
Reshape
$8 \times F$
$K = 2$, $S = 2$
1D Dconv 1
$16 \times F$
$K = 2$, $S = 2$
1D Dconv 2
$32 \times F$
$K = 2$, $S = 1$
1D Dconv 3
$33 \times n_q$
Split/Reshape
$\mathbf{W}^{(1)} \in \mathbb{R}^{32 \times n_{x'}}$
$\mathbf{b}^{(1)} \in \mathbb{R}^{32 \times 1}$
$\mathbf{W}^{(2)}, \mathbf{W}^{(3)}, \mathbf{W}^{(4)} \in \mathbb{R}^{32 \times 32}$
$\mathbf{b}^{(2)}, \mathbf{b}^{(3)}, \mathbf{b}^{(4)} \in \mathbb{R}^{32 \times 1}$
$\mathbf{W}^{(5)} \in \mathbb{R}^{n_q \times 32}$
$\mathbf{b}^{(4)} \in \mathbb{R}^{n_q \times 1}$
Main Network
$\mathbf{x}'$
Dense 1
$\mathbf{h}^{(1)}$
Dense 2
$\mathbf{h}^{(2)}$
Dense 3
$\mathbf{h}^{(3)}$
Dense 4
$\mathbf{h}^{(4)}$
Dense 5
$\hat{\mathbf{q}}$
$\mathbf{h}^{(1)} = \sigma.(\mathbf{W}^{(1)}\mathbf{x}' + \mathbf{b}^{(1)})$
$\mathbf{h}^{(2)} = \sigma.(\mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)})$
$\mathbf{h}^{(3)} = \sigma.(\mathbf{W}^{(3)}\mathbf{h}^{(2)} + \mathbf{b}^{(3)})$
$\mathbf{h}^{(4)} = \sigma.(\mathbf{W}^{(4)}\mathbf{h}^{(3)} + \mathbf{b}^{(4)})$
$\hat{\mathbf{q}} = \mathbf{W}^{(5)}\mathbf{h}^{(4)} + \mathbf{b}^{(5)}$

Figure 6.1: Detailed schematic for a one-shot decoder convolutional hypernetwork, where the main network hidden dimension is the same for all layers with a value of 32, a power of 2. This results in a 3 zone hypernetwork, with the kernel and stride dimensions shown. Note that padding is excluded in all zones, $P = 0$, and F is the number of filter maps per layer.

Note that the training considerations and methods of Section 3.2.5.2 also apply to decoder hypernetworks. However, the scaling in number of trainable parameters per Equation 3.25 does not apply to convolutional decoder hypernetworks, and it is actually possible to design a hypernetwork which has fewer weights than the main network. Once a main-network design is specified via n_L and H, then the hypernetwork design depends only on the choice of $D_{\min}$ and F. Define p_2 such that $H = 2^{p_2}$ and p_1 such that $D_{\min}^{(1)} = 2^{p_1}$, then the total number of trainable parameters in the hypernetwork has a closed-form expression, given by

$$\dim(\theta_h)_{\text{zone1}} = D_{\min} F(n_\mu + 1) + (p_2 - p_1 - 1)(2F^2 + F) + (n_{x'} + 1)(2F + 1) \tag{6.1}$$

$$\dim(\theta_h)_{\text{zone2}} = D_{\min}^2 F(n_\mu + 1) + (p_2 - p_1)(4F^2 + F) + (n_L - 1)(2F + 1) \tag{6.2}$$

$$\dim(\theta_h)_{\text{zone3}} = D_{\min} F(n_\mu + 1) + (p_2 - p_1)(2F^2 + F) + n_q(2F + 1) \tag{6.3}$$

$$\dim(\theta_h) = \dim(\theta_h)_{\text{zone1}} + \dim(\theta_h)_{\text{zone2}} + \dim(\theta_h)_{\text{zone3}}. \tag{6.4}$$

Letting $n_L = 5$, $H = 64$, $D_{\min} = 4$, with $n_{x'} = 3$, $n_q = 3$, and $n_\mu = 9$, which corresponds to a vehicle aerodynamics problem in Section 4.3.4, then $\dim(\theta_h)$ may be plotted versus F as shown in Figure, where the red dashed line is the number of main network weights generated by the hypernetwork. Thus choosing F less than approximately 20 results in a model with fewer parameters than even DV-MLP. This is a great reduction in the number of trainable parameters as compared to dense one-shot hypernetworks, which scale according to Equation 3.25.

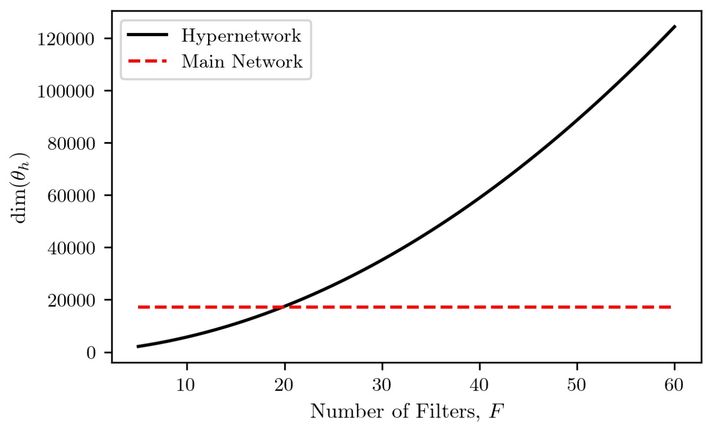


Figure 6.2: Plotting the number of hypernetwork trainable weights against the number of filters F, for a main-network with $n_L = 5$ hidden layers with $H = 64$, and input/output dimensions corresponding to a vehicle aerodynamics problem of Section 4.3.4. This shows that using a decoder convolutional hypernetwork may reduce the number of weights even compared to the main network.

A small handful of experiments were performed using decoder-convolutional DVH (DC-DVH) models on the vehicle aerodynamics problem of Section 4.3. A similar main network to the above experiments was selected, with a hidden dimension of $H = 64$ and $n_L = 5$ hidden layers. A minimal convolutional dimension of $D_{\text{min}} = 4$ was selected, and a few networks trained using different values of F, with the number of trainable parameters given in Table and the resulting errors shown in Table 6.2.

Table 6.1: Summary of training and validation error metrics for vehicle speeds of 90 and 130 kph with a training fraction of 0.80.

# Filters	**# Trainable Weights**, $\dim(\theta_h)$
20	17,471
40	58,931
100	327,311
130	542,501

In Chapter 5 it was observed that supplying residual information, from CFD or from an additional QoI emulator, was required to obtain good results in optimization. Exploration of methods for incorporating both convergence and spatial discretization errors into the model predictions may provide similar utility for the models themselves. A simple idea for handling convergence errors would be to include the solution residual as an element of $\boldsymbol{\mu}$. This may make sense during regression where the residual is known, but

Table 6.2: Summary of training and validation error metrics for vehicle speeds of 90 and 130 kph with a training fraction of 0.80.

$\hat{\mathbf{q}}_i$	F	RMSE (train / val)	MRL2E (train / val)
p [Pa]	20	29.9 / 32.4	7.95% / 8.22%
	40	16.9 / 19.3	4.51% / 4.98%
	100	17.1 / 19.1	4.54% / 4.83%
	130	16.9 / 19.1	4.50% / 4.88%
u [m/s]	20	0.97 / 1.03	3.42% / 3.58%
	40	0.82 / 0.88	2.90% / 3.05%
	100	0.92 / 0.96	3.26% / 3.33%
	130	0.97 / 1.07	3.44% / 3.71%
v [m/s]	20	1.10 / 1.15	21.9% / 8.22%
	40	0.67 / 0.73	13.3% / 14.2%
	100	0.75 / 0.79	14.8% / 15.3%
	130	0.68 / 0.72	13.4% / 14.1%

during inference this may not make sense. Handling of spatial discretization errors, for example training a single model with data from meshes of variable coarseness, may also be handled in a similar manner, with a metric describing the mesh coarseness used as an element of $\boldsymbol{\mu}$. This scenario may make more sense in inference as the mesh coarseness is known before a prediction is generated. Another area of future work involves uncertainty quantification. Given that DVH models are neural network generators, it may be possible to use DVH in a Bayesian neural network scheme to provide uncertainty information.

Appendix A

DVH Network Scaling: Further Details

Considering a main network with L_m hidden layers, each with hidden dimension H, the total number of weights in the main network is

$$\dim(\theta_m) = (Hn_{x'} + H) + (L_m - 1)(H^2 + H) + (n_q H + n_q), \tag{A.1}$$

where the first term corresponds to the first hidden layer, the second term to all remaining hidden layers, and the final term the output layer. With low-dimensional input and output spaces, typically $n_{x'},\ n_\mu,\ n_q << H$, thus quadratic H^2 terms dominate, leading to simplifying approximations

$$\dim(\theta_m) \approx (L_m - 1)H^2 + \mathcal{O}(H) \approx L_m H^2 + \mathcal{O}(H). \tag{A.2}$$

Next consider a hypernetwork with L_h hidden layers, where the first $L_h - 1$ layers also have a hidden dimension of H, while the final hidden layer has dimension H_L. The total number of weights in the hypernetwork is

$$\dim(\theta_h) = (Hn_\mu + H) + (L_h - 2)(H^2 + H) + (H_L H + H_L) + (\dim(\theta_m)H_L + \dim(\theta_m)), \tag{A.3}$$

where the terms are again in forward-propagation order. Retaining quadratic terms leads to the approximation

$$\dim(\theta_h) \approx (L_h - 2)H^2 + H_L H + \dim(\theta_m)(H_L + 1) + \mathcal{O}(H). \tag{A.4}$$

The third term in this expression typically dominates, corresponding to the hypernetwork output layer, revealing the important scaling consideration: a one-shot dense hypernetwork model will have roughly H_L times as many trainable weights as a similar DV-MLP model. Writing this proportionality, and substituting Equation A.2 gives the final result,

$$\dim(\theta_h) \propto \dim(\theta_m) H_L \propto L_m H^2 H_L, \tag{A.5}$$

corresponding to Equation 3.25 of the main text.

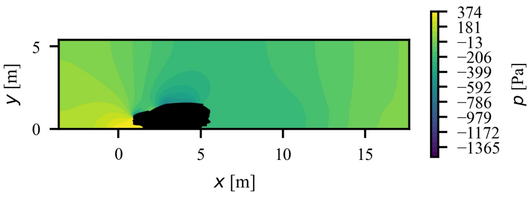
Appendix B

Vehicle Aerodynamics Dataset

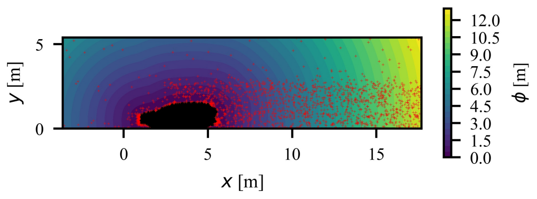
B.1 Baseline Results: Additional Figures

B.1.1 Single Vehicle Speed

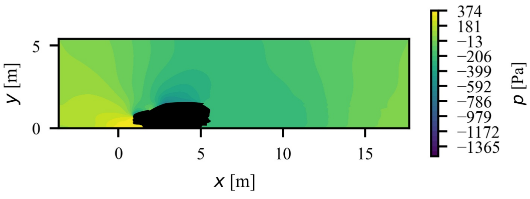
Full domain predictions for an unseen vehicle shape corresponding to Figure 4.11 are shown below.



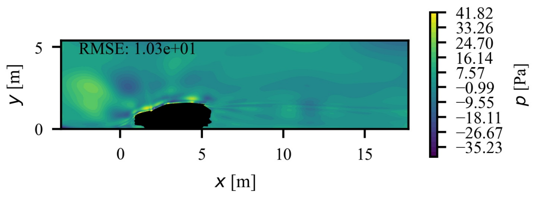
(a) Ground truth pressure field.



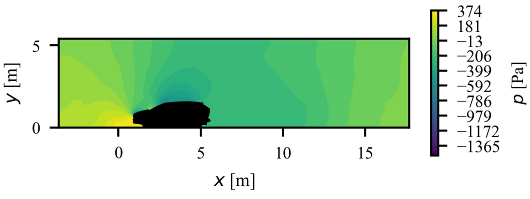
(b) SDF field with truncated points marked.



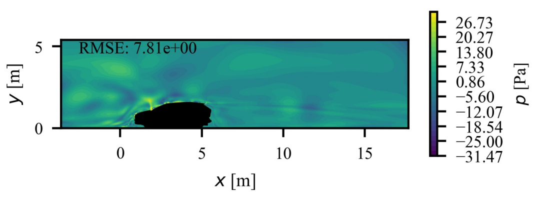
(c) DV-MLP predicted pressure field.



(d) DV-MLP error.

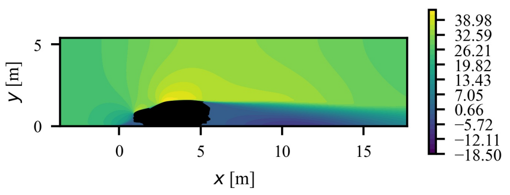


(e) DVH predicted pressure field.

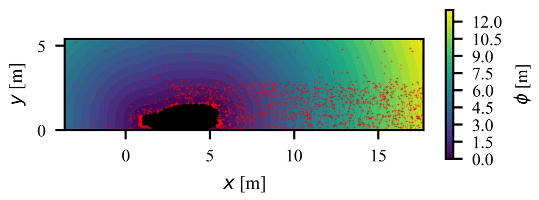


(f) DVH error.

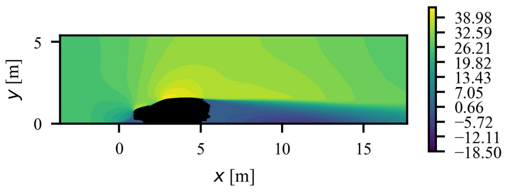
Figure B.1: Unseen vehicle pressure field predictions and errors.



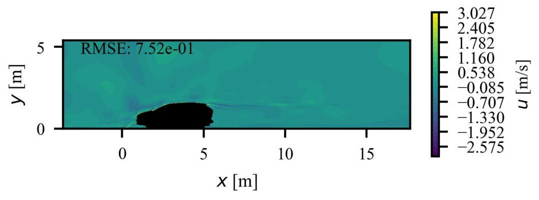
(a) Ground truth x-velocity field.



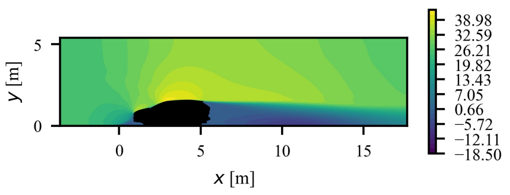
(b) SDF field with truncated points marked.



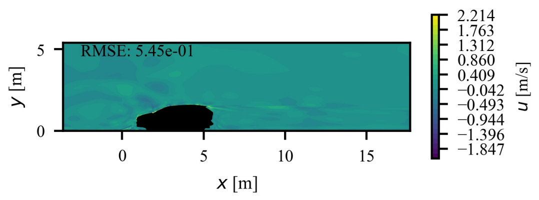
(c) DV-MLP predicted x-velocity field.



(d) DV-MLP error.

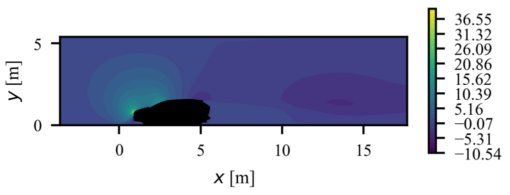


(e) DVH predicted x-velocity field.

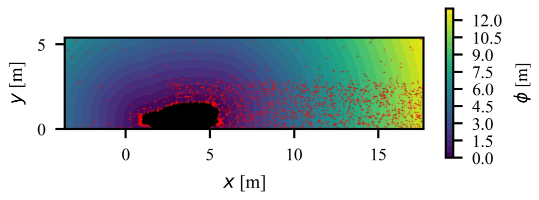


(f) DVH error.

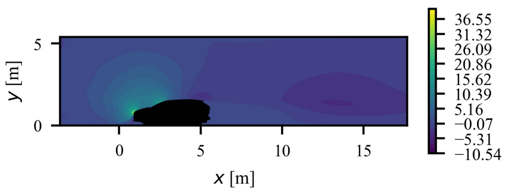
Figure B.2: Unseen vehicle x-velocity predictions and errors.



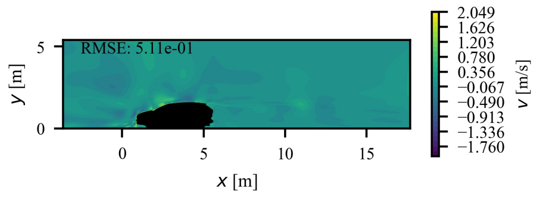
(a) Ground truth y-velocity field.



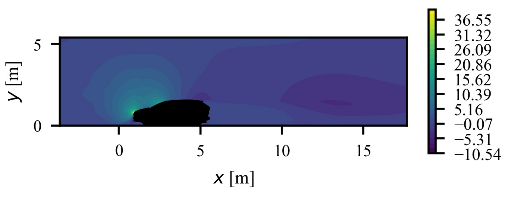
(b) SDF field with truncated points marked.



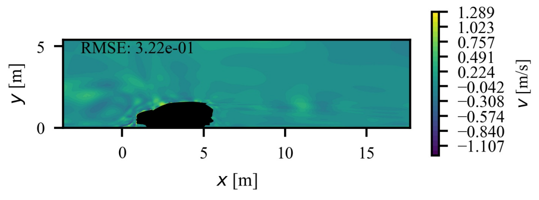
(c) DV-MLP predicted y-velocity field.



(d) DV-MLP error.



(e) DVH predicted y-velocity field.

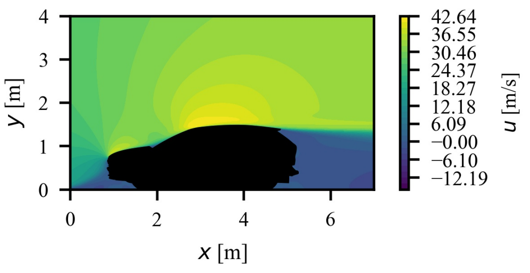


(f) DVH error.

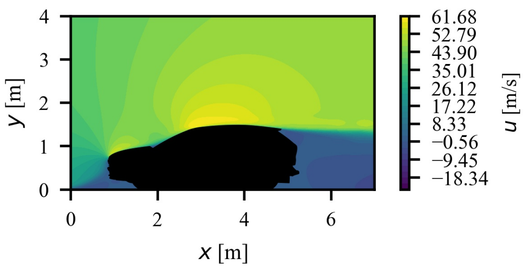
Figure B.3: Unseen vehicle y-velocity predictions and errors.

B.1.2 Multiple Vehicle Speeds

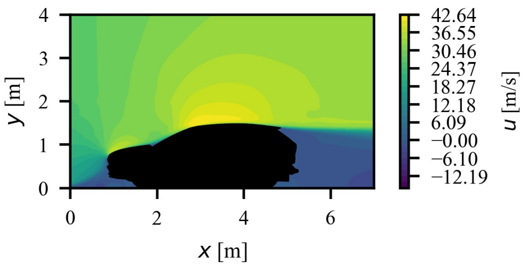
Additional plots of ground truth and predicted x-velocity and y-velocity fields for a single vehicle shape at both speeds of 90 and 130 kph, corresponding to the same vehicle shape as Figure 4.14. As with the pressure field, the velocity component predictions closely match the ground truth at both speeds, with the largest errors seen near the vehicle surface and in the free-shear layer of the wake.



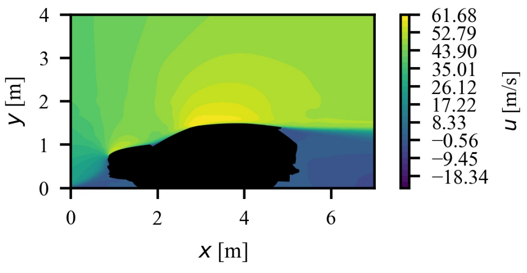
(a) 90 kph, ground truth.



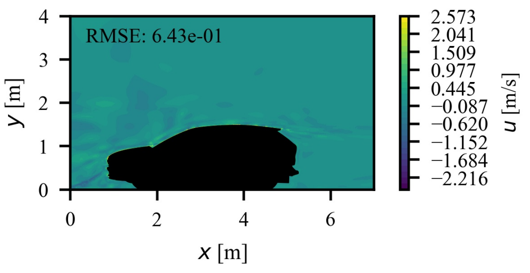
(b) 130 kph, ground truth.



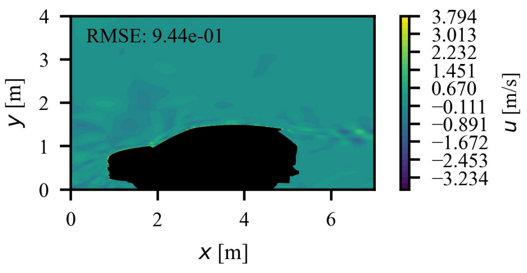
(c) 90 kph, DVH prediction, training set.



(d) 130 kph, DVH prediction, validation set.

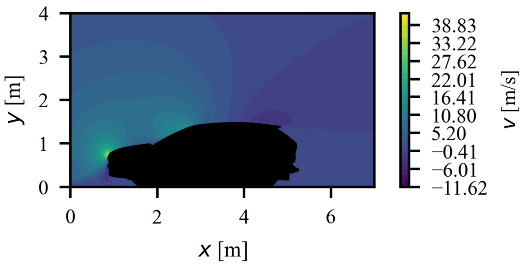


(e) 90 kph DVH error, training set.

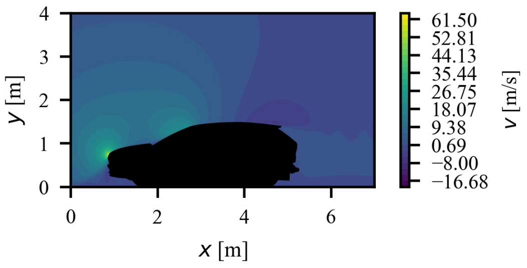


(f) 130 kph DVH error, validation set.

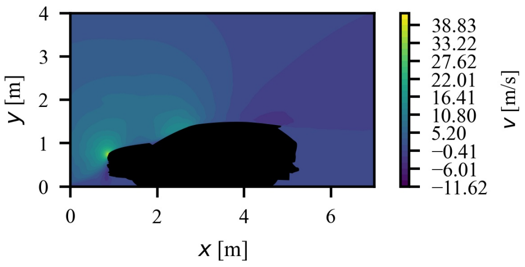
Figure B.4: x-velocity field ground truth, DVH prediction, and errors at 90 and 130 kph for the same vehicle shape.



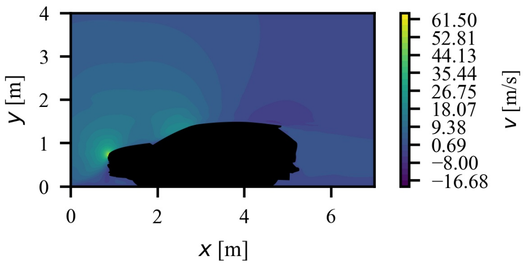
(a) 90 kph, ground truth.



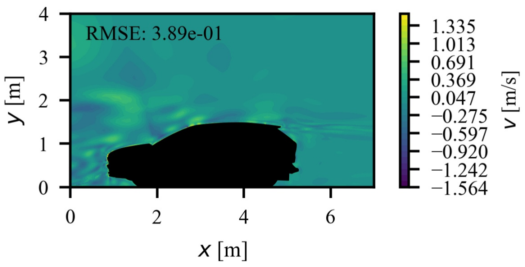
(b) 130 kph, ground truth.



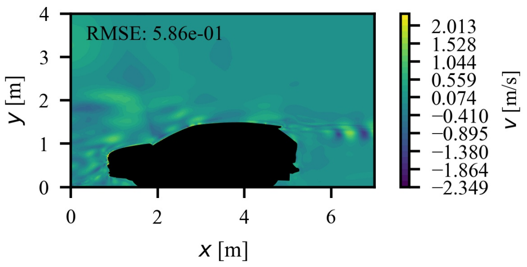
(c) 90 kph, DVH prediction, training set.



(d) 130 kph, DVH prediction, validation set.



(e) 90 kph DVH error, training set.

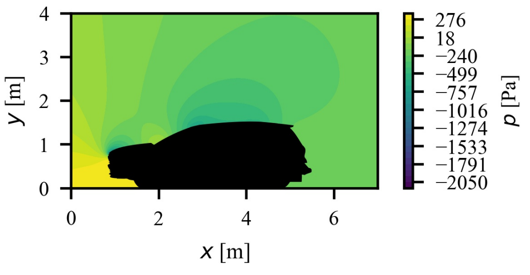


(f) 130 kph DVH error, validation set.

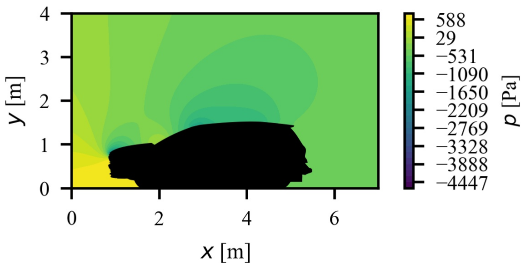
Figure B.5: y-velocity field ground truth, DVH prediction, and errors at 90 and 130 kph for the same vehicle shape.

B.2 Fourier Features: Additional Figures

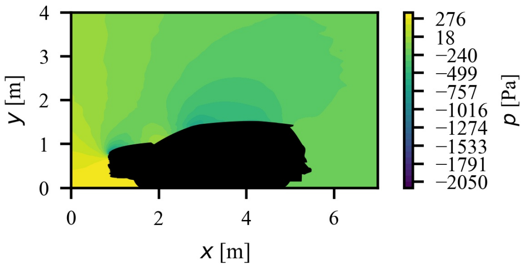
Additional figures showing DVH pressure field and y-velocity field predictions at speeds of 90 and 130 kph, where neither instance was included in the training set. These correspond to the same case as shown in Figure 4.19.



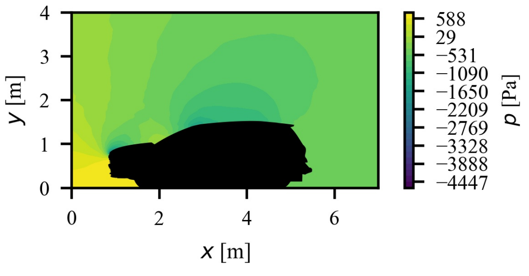
(a) 90 kph, ground truth.



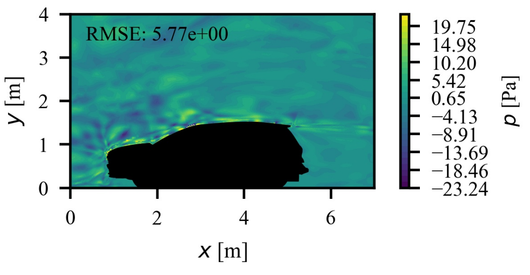
(b) 130 kph, ground truth.



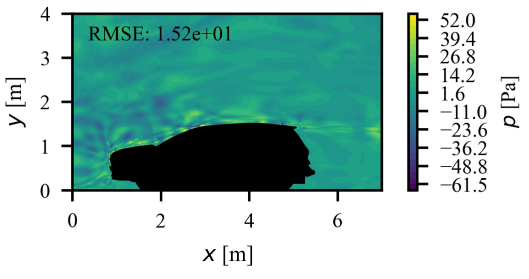
(c) 90 kph, DVH prediction, validation set.



(d) 130 kph, DVH prediction, validation set.

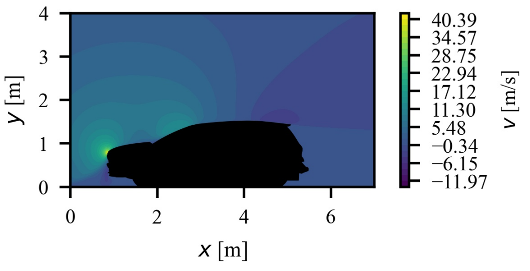


(e) 90 kph DVH error, validation set.

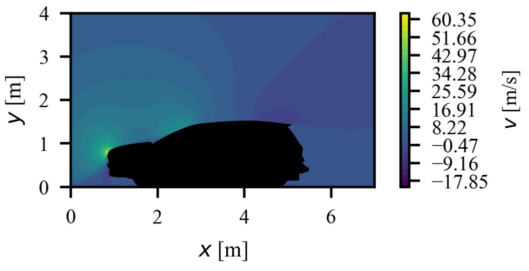


(f) 130 kph DVH error, validation set.

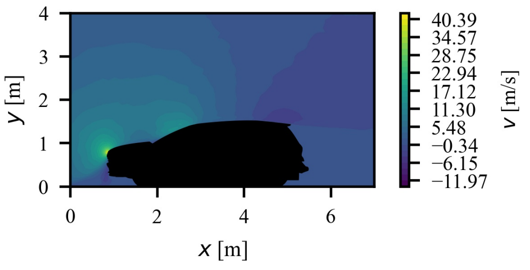
Figure B.6: Pressure field ground truth, DVH prediction, and errors at 90 and 130 kph for the same vehicle shape, where neither instance was included in the training set.



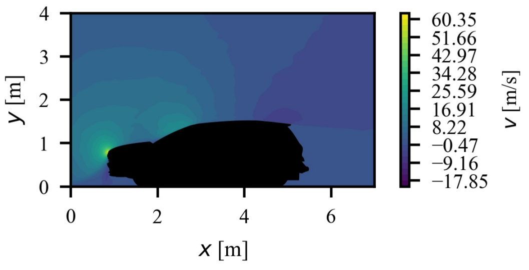
(a) 90 kph, ground truth.



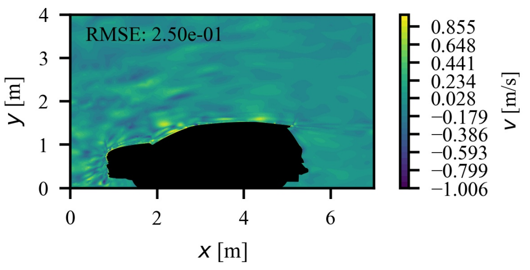
(b) 130 kph, ground truth.



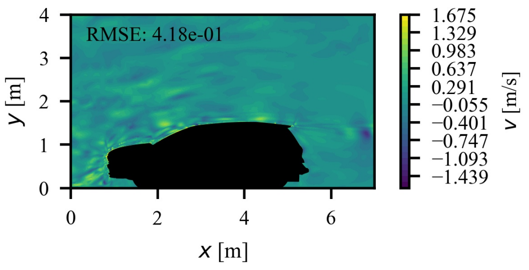
(c) 90 kph, DVH prediction, validation set.



(d) 130 kph, DVH prediction, validation set.



(e) 90 kph DVH error, validation set.



(f) 130 kph DVH error, validation set.

Figure B.7: y-velocity field ground truth, DVH prediction, and errors at 90 and 130 kph for the same vehicle shape, where neither instance was included in the training set.

Bibliography

[1] Gordon E. Moore. *Cramming more components onto integrated circuits*. 1965.

[2] Robert R Schaller. "Moore's law: past, present and future". In: *IEEE Spectrum* 34.6 (1997), pp. 52–59. DOI: 10.1109/6.591665.

[3] Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. "The LINPACK benchmark: past, present and future". In: *Concurrency and Computation: practice and experience* 15.9 (2003), pp. 803–820. DOI: 10.1002/cpe.728.

[4] URL: https://www.top500.org/lists/top500/2023/06/.

[5] Awais Khan, Hyogi Sim, Sudharshan S Vazhkudai, Ali R Butt, and Youngjae Kim. "An Analysis of System Balance and Architectural Trends Based on Top500 Supercomputers". In: *The International Conference on High Performance Computing in Asia-Pacific Region*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 11–22. ISBN: 9781450388429. DOI: 10.1145/3432261.3432263.

[6] Tom Pulliam and David Zingg. *Fundamental Algorithms in Computational Fluid Dynamics*. Jan. 2014. ISBN: 978-3-319-05052-2. DOI: 10.1007/978-3-319-05053-9.

[7] Charles Hirsch. *Numerical computation of internal and external flows: The fundamentals of computational fluid dynamics*. Elsevier, 2007.

[8] Joe F Thompson, Frank C Thames, and C Wayne Mastin. "Automatic numerical generation of body-fitted curvilinear coordinate system for field containing any number of arbitrary two-dimensional bodies". In: *Journal of Computational Physics* 15.3 (1974), pp. 299–319. DOI: 10.1016/0021-9991(74)90114-4.

[9] Frank M. White. *Viscous Fluid Flow*. 3rd ed. Mcgraw-Hill, 2005.

[10] H. Schlichting and K. Gersten. *Boundary–Layer Theory.* 8th ed. Springer–Verlag, 2000.

[11] David C. Wilcox. *Turbulence modeling for CFD.* Vol. 3. DCW Industries La Canada, CA, 2006.

[12] Stephen B. Pope. “The scales of turbulent motion”. In: *Turbulent Flows.* Cambridge University Press, 2000, 182–263. DOI: 10.1017/CBO9781316179475.008.

[13] Haecheon Choi and Parviz Moin. “Grid-point requirements for large eddy simulation: Chapman’s estimates revisited”. In: *Physics of fluids* 24.1 (2012). DOI: 10.1063/1.3676783.

[14] Stephen B. Pope. “Direct numerical simulation”. In: *Turbulent Flows.* Cambridge University Press, 2000, 344–357. DOI: 10.1017/CBO9781316179475.011.

[15] Marco Lanfrit. *Best practice guidelines for handling Automotive External Aerodynamics with FLUENT.* 2005.

[16] Myoungkyu Lee, Nicholas Malaya, and Robert D Moser. “Petascale direct numerical simulation of turbulent channel flow on up to 786k cores”. In: *SC ’13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* 2013, pp. 1–11. DOI: 10.1145/2503210.2503298.

[17] Joseph Smagorinsky. “General circulation experiments with the primitive equations: I. The basic experiment”. In: *Monthly weather review* 91.3 (1963), pp. 99–164. DOI: 10.1175/1520-0493(1963)091<0099:GCEWTP>2.3.CO;2.

[18] Stephen B. Pope. “Large-eddy simulation”. In: *Turbulent Flows.* Cambridge University Press, 2000, 558–640. DOI: 10.1017/CBO9781316179475.015.

[19] W Peter Jones and Brian Edward Launder. “The prediction of laminarization with a two-equation model of turbulence”. In: *International Journal of Heat and Mass Transfer* 15.2 (1972), pp. 301–314. DOI: 10.1016/0017-9310(72)90076-2.

[20] David C Wilcox. “Formulation of the k-w Turbulence Model Revisited”. In: *AIAA journal* 46.11 (2008), pp. 2823–2838. DOI: 10.2514/1.36541.

[21] Florian R Menter, Martin Kuntz, Robin Langtry, et al. "Ten years of industrial experience with the SST turbulence model". In: *Turbulence, heat and mass transfer* 4.1 (2003), pp. 625–632.

[22] P. Sagaut E. Garnier N. Adams. *Large Eddy Simulation for Compressible Flows.* Springer, 2009. DOI: 10.1007/978-90-481-2819-8.

[23] Nicolas Gourdain, Frédéric Sicot, Florent Duchaine, and Laurent Gicquel. "Large eddy simulation of flows in industrial compressors: a path from 2015 to 2035". In: *Philosophical Transactions of the Royal society A: Mathematical, Physical and engineering sciences* 372.2022 (2014), p. 20130323. DOI: 10.1098/rsta.2013.0323.

[24] Philippe R Spalart. "Strategies for turbulence modelling and simulations". In: *International Journal of Heat and Fluid Flow* 21.3 (2000), pp. 252–263. DOI: 10.1016/S0142-727X(00)00007-2.

[25] Philippe R Spalart. "Comments on the Feasibility of LES for Wings and on the Hybrid RANS/LES Approach". In: *Proceedings of the First AFOSR International Conference on DNS/LES, 1997.* 1997, pp. 137–147. ISBN: 1570743657.

[26] Philippe R Spalart. "Detached-Eddy Simulation". In: *Annual Review of Fluid Mechanics* 41 (2009), pp. 181–202. DOI: 10.1146/annurev.fluid.010908.165130.

[27] S Yarlanki, Bipin Rajendran, and H Hamann. "Estimation of turbulence closure coefficients for data centers using machine learning algorithms". In: *13th Inter-Society Conference on Thermal and Thermomechanical Phenomena in Electronic Systems.* IEEE. 2012, pp. 38–42. DOI: 10.1109/ITHERM.2012.6231411.

[28] Zhengqi Gu, Xin Song, Yejie Jiang, and Xu Gong. *Optimization of the Realizable k-ε Turbulence Model Especially for the Simulation of Road Vehicle.* Tech. rep. SAE Technical Paper, 2012. DOI: 10.4271/2012-01-0778.

[29] Todd A Oliver and Robert D Moser. “Bayesian uncertainty quantification applied to RANS turbulence models”. In: *Journal of Physics: Conference Series*. Vol. 318. 4. IOP Publishing. 2011, p. 042032. DOI: 10.1088/1742-6596/318/4/042032.

[30] Eric J Parish and Karthik Duraisamy. “A paradigm for data-driven predictive modeling using field inversion and machine learning”. In: *Journal of Computational Physics* 305 (2016), pp. 758–774. DOI: 10.1016/j.jcp.2015.11.012.

[31] Jonathan R Holland, James D Baeder, and Karthikeyan Duraisamy. “Field Inversion and Machine Learning With Embedded Neural Networks: Physics-Consistent Neural Network Training”. In: *AIAA Aviation 2019 Forum*. 2019, p. 3200. DOI: 10.2514/6.2019-3200.

[32] Vishal Srivastava and Karthik Duraisamy. “Generalizable physics-constrained modeling using learning and inference assisted by feature-space engineering”. In: *Physical Review Fluids* 6.12 (2021), p. 124602. DOI: 10.1103/PhysRevFluids.6.124602.

[33] Eric Parish and Karthikeyan Duraisamy. “Quantification of Turbulence Modeling Uncertainties Using Full Field Inversion”. In: *22nd AIAA Computational Fluid Dynamics Conference*. 2015, p. 2459. DOI: 10.2514/6.2015-2459.

[34] Ze Jia Zhang and Karthikeyan Duraisamy. “Machine Learning Methods for Data-Driven Turbulence Modeling”. In: *22nd AIAA Computational Fluid Dynamics Conference*. 2015, p. 2460. DOI: 10.2514/6.2015-2460.

[35] Brendan Tracey, Karthik Duraisamy, and Juan Alonso. “Application of Supervised Learning to Quantify Uncertainties in Turbulence and Combustion Modeling”. In: *51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition*. 2013. DOI: 10.2514/6.2013-259.

[36] Anand Pratap Singh, Shivaji Medida, and Karthik Duraisamy. “Machine-Learning-Augmented Predictive Modeling of Turbulent Separated Flows over Airfoils”. In: *AIAA Journal* 55.7 (2017), pp. 2215–2227. DOI: 10.2514/1.J055595.

[37] Vishal Srivastava, Valentin Sulzer, Peyman Mohtat, Jason B Siegel, and Karthik Duraisamy. "A non-intrusive approach for physics-constrained learning with application to fuel cell modeling". In: *Computational Mechanics* (2023), pp. 1–20. DOI: 10.1007/s00466-023-02342-7.

[38] Dieter Kraft. "A software package for sequential quadratic programming". In: *Forschungsbericht- Deutsche Forschungs- und Versuchsanstalt fur Luft- und Raumfahrt* (1988).

[39] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[40] Justin S. Gray, John T. Hwang, Joaquim R. R. A. Martins, Kenneth T. Moore, and Bret A. Naylor. "OpenMDAO: An open-source framework for multidisciplinary design, analysis, and optimization". In: *Structural and Multidisciplinary Optimization* 59.4 (2019), pp. 1075–1104. DOI: 10.1007/s00158-019-02211-z.

[41] Charles George Broyden. "The Convergence of a Class of Double-rank Minimization Algorithms 1. General Considerations". In: *IMA Journal of Applied Mathematics* 6.1 (1970), pp. 76–90. DOI: 10.1093/imamat/6.1.76.

[42] Roger Fletcher. "A new approach to variable metric algorithms". In: *The Computer Journal* 13.3 (1970), pp. 317–322. DOI: 10.1093/comjnl/13.3.317.

[43] Donald Goldfarb. "A family of variable-metric methods derived by variational means". In: *Mathematics of computation* 24.109 (1970), pp. 23–26. DOI: 10.2307/2004873.

[44] David F Shanno. "Conditioning of quasi-Newton methods for function minimization". In: *Mathematics of computation* 24.111 (1970), pp. 647–656. DOI: 10.2307/2004840.

[45] G. E. P. Box and K. B. Wilson. "On the Experimental Attainment of Optimum Conditions". In: *Journal of the Royal Statistical Society: Series B (Methodological)* 13.1 (1951), pp. 1–38. DOI: https://doi.org/10.1111/j.2517-6161.1951.tb00067.x. eprint: https://rss.onlinelibrary.wiley.com/doi/pdf/10.1111/j.2517-6161.1951.tb00067.x. URL: https://rss.onlinelibrary.wiley.com/doi/abs/10.1111/j.2517-6161.1951.tb00067.x.

[46] JT Li, ZJ Liu, MA Jabbar, and XK Gao. "Design optimization for cogging torque minimization using response surface methodology". In: *IEEE Transactions on Magnetics* 40.2 (2004), pp. 1176–1179. DOI: 10.1109/TMAG.2004.824809.

[47] Hasan Kurtaran, Azim Eskandarian, D Marzougui, and NE Bedewi. "Crashworthiness design optimization using successive response surface approximations". In: *Computational Mechanics* 29 (2002), pp. 409–421. DOI: 10.1007/s00466-002-0351-x.

[48] Jaekwon Ahn, Hyoung-Jin Kim, Dong-Ho Lee, and Oh-Hyun Rho. "Response Surface Method for Airfoil Design in Transonic Flow". In: *Journal of Aircraft* 38.2 (2001), pp. 231–238. DOI: 10.2514/2.2780.

[49] JT Xiong, ZD Qiao, and ZH Han. "Aerodynamic shape optimization of transonic airfoil and wing using response surface methodology". In: *25th Congress of the International Council of the Aeronautical Sciences*. 2006.

[50] Jafar Nejadali. "Shape optimization of regenerative flow compressor with aero-foil type blades using response surface methodology coupled with CFD". In: *Structural*

and Multidisciplinary Optimization 64.4 (2021), pp. 2653–2667. DOI: 10.1007/s00158-021-03020-z.

[51] Donald R Jones, Matthias Schonlau, and William J Welch. “Efficient Global Optimization of Expensive Black-Box Functions”. In: *Journal of Global optimization* 13 (1998), pp. 455–492. DOI: 10.1023/A:1008306431147.

[52] Jiabin Hu, Yuze Jiang, Jiayu Li, and Tianyue Yuan. “Alternative Acquisition Functions of Bayesian Optimization in terms of Noisy Observation”. In: *Proceedings of the 2021 European Symposium on Software Engineering*. 2021, pp. 112–119. DOI: 10.1145/3501774.3501791.

[53] James Wilson, Frank Hutter, and Marc Deisenroth. “Maximizing acquisition functions for Bayesian optimization”. In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. Vol. 31. Red Hook, NY, USA: Curran Associates Inc., 2018.

[54] Ney Rafael Secco and Bento Silva de Mattos. “Artificial neural networks to predict aerodynamic coefficients of transport airplanes”. In: *Aircraft Engineering and Aerospace Technology* 89.2 (2017), pp. 211–230. DOI: 10.1108/AEAT-05-2014-0069.

[55] Abdelwahid Boutemedjet, Marija Samardžić, Lamine Rebhi, Zoran Rajić, and Takieddine Mouada. “UAV aerodynamic design involving genetic algorithm and artificial neural network for wing preliminary computation”. In: *Aerospace Science and Technology* 84 (2019), pp. 464–483. DOI: 10.1016/j.ast.2018.09.043.

[56] Mohamed Amine Bouhlel, Sicheng He, and Joaquim RRA Martins. “Scalable gradient–enhanced artificial neural networks for airfoil shape design in the subsonic and transonic regimes”. In: *Structural and Multidisciplinary Optimization* 61 (2020), pp. 1363–1376. DOI: 10.1007/s00158-020-02488-5.

[57] Michael D McKay, Richard J Beckman, and William J Conover. “A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output

from a Computer Code". In: *Technometrics* 21.2 (1979), pp. 239–245. DOI: 10.2307/1268522.

[58] SV Patankar and DB Spalding. "A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows". In: *International Journal of Heat and Mass Transfer* 15.10 (1972), pp. 1787–1806. DOI: 10.1016/0017-9310(72)90054-3.

[59] Jeffrey P Van Doormaal and George D Raithby. "Enhancements of the SIMPLE Method for Predicting Incompressible Fluid Flows". In: *Numerical Heat Transfer* 7.2 (1984), pp. 147–163. DOI: 10.1080/01495728408961817.

[60] F Moukalled and M Darwish. "A unified formulation of the segregated class of algorithms for fluid flow at all speeds". In: *Numerical Heat Transfer, Part B: Fundamentals* 37.1 (2000), pp. 103–139. DOI: 10.1080/104077900275576.

[61] Valentina Dolci and Renzo Arina. "Proper Orthogonal Decomposition as Surrogate Model for Aerodynamic Optimization". In: *International Journal of Aerospace Engineering* 2016 (2016). DOI: 10.1155/2016/8092824.

[62] Filippo Salmoiraghi, Angela Scardigli, Haysam Telib, and Gianluigi Rozza. "Free-form deformation, mesh morphing and reduced-order methods: enablers for efficient aerodynamic shape optimisation". In: *International Journal of Computational Fluid Dynamics* 32.4–5 (2018), pp. 233–247. DOI: 10.1080/10618562.2018.1514115.

[63] Karen Willcox and Jaime Peraire. "Balanced Model Reduction via the Proper Orthogonal Decomposition". In: *AIAA Journal* 40.11 (2002), pp. 2323–2330. DOI: 10.2514/2.1570.

[64] Peter Benner, Serkan Gugercin, and Karen Willcox. "A Survey of Projection-Based Model Reduction Methods for Parametric Dynamical Systems". In: *SIAM Review* 57.4 (2015), pp. 483–531. DOI: 10.1137/130932715.

[65] Matan Gavish and David L Donoho. "The Optimal Hard Threshold for Singular Values is $4/\sqrt{3}$". In: *IEEE Transactions on Information Theory* 60.8 (2014), pp. 5040–5053. DOI: 10.1109/TIT.2014.2323359.

[66] Maxime Barrault, Yvon Maday, Ngoc Cuong Nguyen, and Anthony T Patera. "An 'empirical interpolation' method: application to efficient reduced-basis discretization of partial differential equations". In: *Comptes Rendus Mathematique* 339.9 (2004), pp. 667–672. DOI: 10.1016/j.crma.2004.08.006.

[67] Saifon Chaturantabut and Danny C Sorensen. "Nonlinear Model Reduction via Discrete Empirical Interpolation". In: *SIAM Journal on Scientific Computing* 32.5 (2010), pp. 2737–2764. DOI: 10.1137/090766498.

[68] Zlatko Drmac and Serkan Gugercin. "A New Selection Operator for the Discrete Empirical Interpolation Method—Improved A Priori Error Bound and Extensions". In: *SIAM Journal on Scientific Computing* 38.2 (2016), A631–A648. DOI: 10.1137/15M1019271.

[69] J Nathan Kutz, Syuzanna Sargsyan, and Steven L Brunton. "Leveraging Sparsity and Compressive Sensing for Reduced Order Modeling". In: *Model Reduction of Parametrized Systems*. Springer International Publishing, 2017, pp. 301–315. ISBN: 978-3-319-58786-8. DOI: 10.1007/978-3-319-58786-8_19.

[70] Krithika Manohar, Bingni W Brunton, J Nathan Kutz, and Steven L Brunton. "Data-Driven Sparse Sensor Placement for Reconstruction: Demonstrating the Benefits of Exploiting Known Patterns". In: *IEEE Control Systems Magazine* 38.3 (2018), pp. 63–86. DOI: 10.1109/MCS.2018.2810460.

[71] Richard Everson and Lawrence Sirovich. "Karhunen–Loève procedure for gappy data". In: *Journal of the Optical Society of America A* 12.8 (1995), pp. 1657–1664. DOI: 10.1364/JOSAA.12.001657.

[72] Patricia Astrid, Siep Weiland, Karen Willcox, and Ton Backx. "Missing Point Estimation in Models Described by Proper Orthogonal Decomposition". In: *IEEE*

Transactions on Automatic Control 53.10 (2008), pp. 2237–2251. DOI: 10.1109/TAC.2008.2006102.

[73] Jan S Hesthaven and Stefano Ubbiali. “Non-intrusive reduced order modeling of nonlinear problems using neural networks”. In: *Journal of Computational Physics* 363 (2018), pp. 55–78. DOI: 10.1016/j.jcp.2018.02.037.

[74] Qian Wang, Jan S Hesthaven, and Deep Ray. “Non-intrusive reduced order modeling of unsteady flows using artificial neural networks with application to a combustion problem”. In: *Journal of Computational Physics* 384 (2019), pp. 289–307. DOI: 10.1016/j.jcp.2019.01.031.

[75] Xiaoxiao Guo, Wei Li, and Francesco Iorio. “Convolutional Neural Networks for Steady Flow Approximation”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* 2016, pp. 481–490. DOI: 10.1145/2939672.2939738.

[76] Saakaar Bhatnagar, Yaser Afshar, Shaowu Pan, Karthik Duraisamy, and Shailendra Kaushik. “Prediction of aerodynamic flow fields using convolutional neural networks”. In: *Computational Mechanics* 64.2 (2019), pp. 525–545. DOI: 10.1007/s00466-019-01740-0.

[77] Kaustubh Tangsali, Vinayak R Krishnamurthy, and Zohaib Hasnain. “Generalizability of Convolutional Encoder–Decoder Networks for Aerodynamic Flow-Field Prediction Across Geometric and Physical-Fluidic Variations”. In: *Journal of Mechanical Design* 143.5 (2021). DOI: 10.1115/1.4048221.

[78] Nils Thuerey, Konstantin Weißenow, Lukas Prantl, and Xiangyu Hu. “Deep Learning Methods for Reynolds-Averaged Navier–Stokes Simulations of Airfoil Flows”. In: *AIAA Journal* 58.1 (2020), pp. 25–36. DOI: 10.2514/1.J058291.

[79] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. “”U-Net: Convolutional Networks for Biomedical Image Segmentation”. In: *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015.* Springer. 2015, pp. 234–241. ISBN: 978-3-319-24574-4. DOI: 10.1007/978-3-319-24574-4_28.

[80] Jiayang Xu and Karthik Duraisamy. "Multi-level convolutional autoencoder networks for parametric prediction of spatio-temporal dynamics". In: *Computer Methods in Applied Mechanics and Engineering* 372 (2020), p. 113379. ISSN: 0045-7825. DOI: 10.1016/j.cma.2020.113379.

[81] Kazuto Hasegawa, Kai Fukami, Takaaki Murata, and Koji Fukagata. "Machine-learning-based reduced-order modeling for unsteady flows around bluff bodies of various shapes". In: *Theoretical and Computational Fluid Dynamics* 34.4 (2020), pp. 367–383. DOI: 10.1007/s00162-020-00528-w.

[82] Arvind Mohan, Don Daniel, Michael Chertkov, and Daniel Livescu. "Compressed Convolutional LSTM: An Efficient Deep Learning framework to Model High Fidelity 3D Turbulence". In: *arXiv preprint arXiv:1903.00033* (2019). URL: https://api.semanticscholar.org/CorpusID:119353217.

[83] Javier E Santos, Duo Xu, Honggeun Jo, Christopher J Landry, Maša Prodanović, and Michael J Pyrcz. "PoreFlow-Net: A 3D convolutional neural network to predict fluid flow through porous media". In: *Advances in Water Resources* 138 (2020), p. 103539. ISSN: 0309-1708. DOI: 10.1016/j.advwatres.2020.103539.

[84] Feng Gao, Zhuang Zhang, Chenyang Jia, Yin Zhu, Chunli Zhou, and Jingtao Wang. "Simulation and prediction of three-dimensional rotating flows based on convolutional neural networks". In: *Physics of Fluids* 34.9 (2022), p. 095116. DOI: 10.1063/5.0113030.

[85] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. *Spectral Networks and Locally Connected Networks on Graphs.* 2014. DOI: 10.48550/arXiv.1312.6203. arXiv: 1312.6203 [cs.LG].

[86] Mikael Henaff, Joan Bruna, and Yann LeCun. *Deep Convolutional Networks on Graph-Structured Data.* 2015. DOI: https://doi.org/10.48550/arXiv.1506.05163. arXiv: 1506.05163 [cs.LG].

[87] Max Welling and Thomas N Kipf. “Semi-supervised classification with graph convolutional networks”. In: *J. International Conference on Learning Representations (ICLR 2017)*. 2016.

[88] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. “Convolutional neural networks on graphs with fast localized spectral filtering”. In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*. Vol. 29. Red Hook, NY, USA: Curran Associates Inc., 2016, 3844–3852. ISBN: 9781510838819.

[89] David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P Adams. “Convolutional networks on graphs for learning molecular fingerprints”. In: *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*. Vol. 28. NIPS’15. MIT Press, 2015, 2224–2232.

[90] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. “Neural message passing for quantum chemistry”. In: *Proceedings of the 34th International Conference on Machine Learning - Volume 70*. Sydney, NSW, Australia: JMLR.org, 2017, pp. 1263–1272.

[91] Francis Ogoke, Kazem Meidani, Amirreza Hashemi, and Amir Barati Farimani. “Graph convolutional networks applied to unstructured flow field data”. In: *Machine Learning: Science and Technology* 2 (2020). URL: `https://api.semanticscholar.org/CorpusID:227306077`.

[92] Jiayang Xu, Aniruddhe Pradhan, and Karthikeyan Duraisamy. “Conditionally Parameterized, Discretization-Aware Neural Networks for Mesh-Based Modeling of Physical Systems”. In: *Advances in Neural Information Processing Systems*. Vol. 34. 2021, pp. 1634–1645.

[93] Tobias Pfaff, Meire Fortunato, Alvaro Sanchez-Gonzalez, and Peter Battaglia. “Learning Mesh-Based Simulation with Graph Networks”. In: *International Con-*

ference on Learning Representations. 2021. URL: https://openreview.net/forum?id=roNqYL0_XP.

[94] Alvaro Sanchez-Gonzalez, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, and Peter Battaglia. “Learning to simulate complex physics with graph networks”. In: *International Conference on Machine Learning.* PMLR. 2020, pp. 8459–8468.

[95] Lu Lu, Pengzhan Jin, Guofei Pang, Zhongqiang Zhang, and George Em Karniadakis. “Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators”. In: *Nature Machine Intelligence* 3.3 (2021), pp. 218–229. DOI: 10.1038/s42256-021-00302-5.

[96] Sifan Wang, Hanwen Wang, and Paris Perdikaris. “Learning the solution operator of parametric partial differential equations with physics-informed DeepONets”. In: *Science Advances* 7.40 (2021), eabi8605.

[97] Shengze Cai, Zhicheng Wang, Lu Lu, Tamer A Zaki, and George Em Karniadakis. “DeepM&Mnet: Inferring the electroconvection multiphysics fields based on operator approximation by neural networks”. In: *Journal of Computational Physics* 436 (2021), p. 110296. DOI: 10.1016/j.jcp.2021.110296.

[98] Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. “Neural operator: Graph kernel network for partial differential equations”. In: *arXiv preprint arXiv:2003.03485* (2020).

[99] Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Andrew Stuart, Kaushik Bhattacharya, and Anima Anandkumar. “Multipole graph neural operator for parametric partial differential equations”. In: *Proceedings of the 34th International Conference on Neural Information Processing Systems.* Vol. 33. 2020, pp. 6755–6766.

[100] Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. “Fourier neural operator for

parametric partial differential equations”. In: *arXiv preprint arXiv:2010.08895* (2020).

[101] Nathaniel Trask, Ravi G. Patel, Ben J. Gross, and Paul J. Atzberger. *GMLS-Nets: A framework for learning from unstructured data.* 2019. arXiv: 1909.05371 [cs.LG].

[102] James Duvall, Michael Joly, Karthikeyan Duraisamy, and Soumalya Sarkar. “Flowfield Emulation and Shape Optimization of Compressor Airfoils using Design-Variable Hypernetworks”. In: *AIAA SCITECH 2023 Forum.* DOI: 10.2514/6.2023-1678. eprint: https://arc.aiaa.org/doi/pdf/10.2514/6.2023-1678. URL: https://arc.aiaa.org/doi/abs/10.2514/6.2023-1678.

[103] James Duvall, Michael Joly, Karthik Duraisamy, and Soumalya Sarkar. “Design-Variable Hypernetworks for Flowfield Emulation and Shape Optimization of Compressor Airfoils”. In: *AIAA Journal* (2023), pp. 1–17. DOI: 10.2514/1.J063156.

[104] Tinne Hoff Kjeldsen. “A Contextualized Historical Analysis of the Kuhn–Tucker Theorem in Nonlinear Programming: The Impact of World War II”. In: *Historia Mathematica* 27.4 (2000), pp. 331–361. ISSN: 0315-0860. DOI: https://doi.org/10.1006/hmat.2000.2289. URL: https://www.sciencedirect.com/science/article/pii/S0315086000922894.

[105] Joaquim RRA Martins, Peter Sturdza, and Juan J Alonso. “The complex-step derivative approximation”. In: *ACM Transactions on Mathematical Software (TOMS)* 29.3 (2003), pp. 245–262. ISSN: 0098-3500. DOI: 10.1145/838250.838251.

[106] Charles C Margossian. “A review of automatic differentiation and its efficient implementation”. In: *WIREs Data Mining and Knowledge Discovery* 9.4 (2019), e1305. DOI: https://doi.org/10.1002/widm.1305.

[107] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. “On the importance of initialization and momentum in deep learning”. In: *Proceedings of the 30th International Conference on Machine Learning.* Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research 3. Atlanta,

Georgia, USA: PMLR, 2013, pp. 1139–1147. URL: https://proceedings.mlr.press/v28/sutskever13.html.

[108] Reeves Fletcher and Colin M Reeves. "Function minimization by conjugate gradients". In: *The Computer Journal* 7.2 (1964), pp. 149–154. DOI: 10.1093/comjnl/7.2.149.

[109] Magnus R Hestenes and Eduard Stiefel. "Methods of conjugate gradients for solving linear systems". In: *Journal of research of the National Bureau of Standards* 49.6 (1952), pp. 409–436. URL: https://api.semanticscholar.org/CorpusID:2207234.

[110] Joaquim R. R. A. Martins and Andrew Ning. *Engineering Design Optimization.* Cambridge University Press, 2021. DOI: 10.1017/9781108980647.

[111] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization.* 2e. New York, NY, USA: Springer, 2006.

[112] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014). DOI: 10.48550/arXiv.1412.6980.

[113] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *Journal of Machine Learning Research* 12.61 (2011), pp. 2121–2159. URL: http://jmlr.org/papers/v12/duchi11a.html.

[114] Matthew D. Zeiler. *ADADELTA: An Adaptive Learning Rate Method.* 2012. DOI: 10.48550/arXiv.1212.5701. arXiv: 1212.5701 [cs.LG].

[115] Geoffrey Hinton. *Coursear Neural Networks for Machine Learning, Lecture 6.* https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf. 2018.

[116] Arthur E Hoerl and Robert W Kennard. "Ridge Regression: Applications to Nonorthogonal Problems". In: *Technometrics* 12.1 (1970), pp. 69–82. DOI: 10.1080/00401706.1970.10488635.

[117] Robert Tibshirani. “Regression Shrinkage and Selection Via the Lasso”. In: *Journal of the Royal Statistical Society: Series B (Methodological)* 58.1 (1996), pp. 267–288. DOI: 10.1111/j.2517-6161.1996.tb02080.x.

[118] Hui Zou and Trevor Hastie. “Regularization and Variable Selection Via the Elastic Net”. In: *Journal of the Royal Statistical Society Series B: Statistical Methodology* 67.2 (Mar. 2005), pp. 301–320. ISSN: 1369-7412. DOI: 10.1111/j.1467-9868.2005.00503.x. eprint: https://academic.oup.com/jrsssb/article-pdf/67/2/301/49795094/jrsssb_67_2_301.pdf. URL: https://doi.org/10.1111/j.1467-9868.2005.00503.x.

[119] Raymond H Myers, Douglas C Montgomery, G Geoffrey Vining, Connie M Borror, and Scott M Kowalski. “Response Surface Methodology: A Retrospective and Literature Survey”. In: *Journal of Quality Technology* 36.1 (2004), pp. 53–77. DOI: 10.1080/00224065.2004.11980252.

[120] David Duvenaud. *The Kernel Cookbook: Advice on Covariance functions.* URL: https://www.cs.toronto.edu/~duvenaud/cookbook/.

[121] Haitao Liu, Jianfei Cai, and Yew-Soon Ong. “Remarks on multi-output Gaussian process regression”. In: *Knowledge-Based Systems* 144 (2018), pp. 102–121. ISSN: 0950-7051. DOI: 10.1016/j.knosys.2017.12.034.

[122] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12.85 (2011), pp. 2825–2830. URL: http://jmlr.org/papers/v12/pedregosa11a.html.

[123] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian processes for machine learning.* Adaptive computation and machine learning. MIT Press, 2006, pp. I–XVIII, 1–248. ISBN: 026218253X.

[124] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological Review* 65.6 (1958), p. 386. DOI: 10.1037/h0042519.

[125] Shun-ichi Amari. "Backpropagation and stochastic gradient descent method". In: *Neurocomputing* 5.4-5 (1993), pp. 185–196. DOI: 10.1016/0925-2312(93)90006-O.

[126] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[127] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[128] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. *JAX: composable transformations of Python+NumPy programs*. Version 0.3.13. 2018. URL: http://github.com/google/jax.

[129] Dor Bank, Noam Koenigstein, and Raja" Giryes. "Autoencoders". In: *Machine Learning for Data Science Handbook: Data Mining and Knowledge Discovery Handbook*. Ed. by Lior Rokach, Oded Maimon, and Erez Shmueli. Springer International Publishing, 2023, pp. 353–374. ISBN: 978-3-031-24628-9. DOI: 10.1007/978-3-031-24628-9_16.

[130] Pierre Baldi. "Autoencoders, Unsupervised Learning, and Deep Architectures". In: *Proceedings of ICML Workshop on Unsupervised and Transfer Learning*. Ed. by Isabelle Guyon, Gideon Dror, Vincent Lemaire, Graham Taylor, and Daniel Silver. Vol. 27. Proceedings of Machine Learning Research. JMLR Workshop and Conference Proceedings. 2012, pp. 37–49. URL: https://proceedings.mlr.press/v27/baldi12a.html.

[131] David E. Rumelhart and James L. McClelland. "Learning Internal Representations by Error Propagation". In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations*. The MIT Press, 1986, pp. 318–362. ISBN: 9780262291408. DOI: 10.7551/mitpress/5236.001.0001.

[132] Elad Plaut. "From principal subspaces to principal components with linear autoencoders". In: *arXiv preprint arXiv:1804.10253* (2018). DOI: 10.48550/arXiv.1804.10253.

[133] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems* 25 (2012). URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[134] Kunihiko Fukushima. "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". In: *Biological cybernetics* 36.4 (1980), pp. 193–202. DOI: 10.1007/BF00344251.

[135] David H Hubel and Torsten N Wiesel. "Receptive fields of single neurones in the cat's striate cortex". In: *The Journal of Physiology* 148.3 (1959), p. 574. DOI: 10.1113/jphysiol.1959.sp006308.

[136] James Atwood and Don Towsley. "Diffusion-convolutional neural networks". In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*. Vol. 29. Curran Associates Inc., 2016, 2001–2009. ISBN: 9781510838819.

[137] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. "Learning convolutional neural networks for graphs". In: *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*. ICML'16. JMLR.org, 2016, pp. 2014–2023.

[138] David I Shuman, Sunil K. Narang, Pascal Frossard, Antonio Ortega, and Pierre Vandergheynst. "The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains". In: *IEEE Signal Processing Magazine* 30.3 (2013), pp. 83–98. DOI: `10.1109/MSP.2012.2235192`.

[139] Stephane Mallat. *A Wavelet Tour of Signal Processing*. 2009.

[140] Thomas N. Kipf and Max Welling. "Semi-Supervised Classification with Graph Convolutional Networks". In: *International Conference on Learning Representations*. 2017. URL: `https://openreview.net/forum?id=SJU4ayYgl`.

[141] Will Hamilton, Zhitao Ying, and Jure Leskovec. "Inductive representation learning on large graphs". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. Vol. 30. Curran Associates Inc., 2017, 1025–1035. ISBN: 9781510860964.

[142] Peter Battaglia, Razvan Pascanu, Matthew Lai, Danilo Jimenez Rezende, et al. "Interaction networks for learning about objects, relations and physics". In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*. Vol. 29. Curran Associates Inc., 2016, 4509–4517. ISBN: 9781510838819.

[143] R. Qi Charles, Hao Su, Mo Kaichun, and Leonidas J. Guibas. "PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 77–85. DOI: `10.1109/CVPR.2017.16`.

[144] Charles Ruizhongtai Qi, Li Yi, Hao Su, and Leonidas J Guibas. "Pointnet++: Deep hierarchical feature learning on point sets in a metric space". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems.* Vol. 30. Curran Associates Inc., 2017, 5105—5114. ISBN: 9781510860964.

[145] Ali Kashefi, Davis Rempe, and Leonidas J Guibas. "A point-cloud deep learning framework for prediction of fluid flow fields on irregular geometries". In: *Physics of Fluids* 33.2 (2021), p. 027104. ISSN: 1070-6631. DOI: `10.1063/5.0033376`.

[146] Lu Lu, Xuhui Meng, Shengze Cai, Zhiping Mao, Somdatta Goswami, Zhongqiang Zhang, and George Em Karniadakis. "A comprehensive and fair comparison of two neural operators (with practical extensions) based on fair data". In: *Computer Methods in Applied Mechanics and Engineering* 393 (2022), p. 114778. DOI: `10.1016/j.cma.2022.114778`.

[147] Nikola B Kovachki, Zongyi Li, Burigede Liu, Kamyar Azizzadenesheli, Kaushik Bhattacharya, Andrew M Stuart, and Anima Anandkumar. "Neural Operator: Learning Maps Between Function Spaces With Applications to PDEs." In: *Journal of Machine Learning Research* 24.89 (2023), pp. 1–97. URL: `http://jmlr.org/papers/v24/21-1524.html`.

[148] Zongyi Li, Daniel Zhengyu Huang, Burigede Liu, and Anima Anandkumar. "Fourier neural operator with learned deformations for pdes on general geometries". In: *arXiv preprint arXiv:2207.05209* (2022). DOI: `10.48550/arXiv.2207.05209`.

[149] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations". In: *Journal of Computational physics* 378 (2019), pp. 686–707. DOI: `https://doi.org/10.1016/j.jcp.2018.10.045`.

[150] Luning Sun, Han Gao, Shaowu Pan, and Jian-Xun Wang. "Surrogate modeling for fluid flows based on physics-constrained deep learning without simulation

data". In: *Computer Methods in Applied Mechanics and Engineering* 361 (2020), p. 112732. ISSN: 0045-7825. DOI: 10.1016/j.cma.2019.112732.

[151] Isaac E Lagaris, Aristidis Likas, and Dimitrios I Fotiadis. "Artificial neural networks for solving ordinary and partial differential equations". In: *IEEE transactions on neural networks* 9.5 (1998), pp. 987–1000. DOI: 10.1109/72.712178.

[152] Isaac E Lagaris, Aristidis C Likas, and Dimitris G Papageorgiou. "Neural-network methods for boundary value problems with irregular boundaries". In: *IEEE Transactions on Neural Networks* 11.5 (2000), pp. 1041–1049. DOI: 10.1109/72.870037.

[153] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[154] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. "Extracting and composing robust features with denoising autoencoders". In: *Proceedings of the 25th international conference on Machine learning*. 2008, pp. 1096–1103.

[155] J Nathan Kutz, Steven L Brunton, Bingni W Brunton, and Joshua L Proctor. *Dynamic mode decomposition: data-driven modeling of complex systems*. SIAM, 2016.

[156] Carlos Pérez Arroyo, Jérôme Dombard, Florent Duchaine, Laurent Gicquel, Benjamin Martin, Nicolas Odier, and Gabriel Staffelbach. "Towards the large-eddy simulation of a full engine: Integration of a 360 azimuthal degrees fan, compressor and combustion chamber. Part I: Methodology and initialisation". In: *Journal of the Global Power and Propulsion Society* May (2021), pp. 1–16. DOI: 10.33737/jgpps/133115.

[157] Peter J Schmid. "Dynamic Mode Decomposition and Its Variants". In: *Annual Review of Fluid Mechanics* 54 (2022), pp. 225–254. DOI: 10.1146/annurev-fluid-030121-015835.

[158] Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. "DeepSDF: Learning Continuous Signed Distance Functions for Shape Representation". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).* 2019, pp. 165–174. DOI: `10.1109/CVPR.2019.00025`.

[159] Thomas Davies, Derek Nowrouzezahrai, and Alec Jacobson. "On the Effectiveness of Weight-Encoded Neural Implicit 3D Shapes". In: *arXiv preprint arXiv:2009.09808* (2020). DOI: `10.48550/arXiv.2009.09808`.

[160] Lars Mescheder, Michael Oechsle, Michael Niemeyer, Sebastian Nowozin, and Andreas Geiger. "Occupancy networks: Learning 3d reconstruction in function space". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition.* 2019, pp. 4460–4470. URL: `https://openaccess.thecvf.com/content_CVPR_2019/papers/Mescheder_Occupancy_Networks_Learning_3D_Reconstruction_in_Function_Space_CVPR_2019_paper.pdf`.

[161] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. "NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis". In: *Communications of the ACM* 65.1 (2021), pp. 99–106. DOI: `10.1145/3503250`.

[162] Yiheng Xie, Towaki Takikawa, Shunsuke Saito, Or Litany, Shiqin Yan, Numair Khan, Federico Tombari, James Tompkin, Vincent sitzmann, and Srinath Sridhar. "Neural Fields in Visual Computing and Beyond". In: vol. 41. 2. 2022, pp. 641–676. DOI: `https://doi.org/10.1111/cgf.14505`.

[163] Ethan Perez, Florian Strub, Harm de Vries, Vincent Dumoulin, and Aaron Courville. "FiLM: Visual Reasoning with a General Conditioning Layer". In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence.* AAAI'18/IAAI'18/EAAI'18. New Orleans, Louisiana, USA: AAAI Press, 2018. ISBN: 978-1-57735-800-8.

[164] Vincent Dumoulin, Ethan Perez, Nathan Schucher, Florian Strub, Harm de Vries, Aaron Courville, and Yoshua Bengio. "Feature-wise transformations". In: *Distill* (2018). https://distill.pub/2018/feature-wise-transformations. DOI: 10.23915/distill.00011.

[165] David Ha, Andrew Dai, and Quoc V Le. "Hypernetworks". In: *arXiv preprint arXiv:1609.09106* (2016). DOI: 10.48550/arXiv.1609.09106.

[166] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. "Generative Adversarial Nets". In: *Advances in Neural Information Processing Systems.* Ed. by Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger. Vol. 27. Curran Associates, Inc., 2014. URL: https://proceedings.neurips.cc/paper/2014/file/5ca3e9b122f61f8f06494c97b1afccf3-Paper.pdf.

[167] Zhiqin Chen and Hao Zhang. "Learning Implicit Fields for Generative Shape Modeling". In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).* 2019, pp. 5932–5941. DOI: 10.1109/CVPR.2019.00609.

[168] Vincent Sitzmann, Julien Martel, Alexander Bergman, David Lindell, and Gordon Wetzstein. "Implicit Neural Representations with Periodic Activation Functions". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 7462–7473. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/53c04118df112c13a8c34b38343b9c10-Paper.pdf.

[169] Max Jaderberg, Karen Simonyan, Andrew Zisserman, and koray kavukcuoglu. "Spatial Transformer Networks". In: *Advances in Neural Information Processing Systems.* Ed. by C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett. Vol. 28. Curran Associates, Inc., 2015. URL: https://proceedings.neurips.cc/paper_files/paper/2015/file/33ceb07bf4eeb3da587e268d663aba1a-Paper.pdf.

[170] Xu Jia, Bert De Brabandere, Tinne Tuytelaars, and Luc V Gool. "Dynamic filter networks". In: *Proceedings of the 30th International Conference on Neural Infor-*

mation Processing Systems. Vol. 29. NIPS'16. Barcelona, Spain: Curran Associates Inc., 2016, 667–675. ISBN: 9781510838819.

[171] Luca Bertinetto, João F Henriques, Jack Valmadre, Philip Torr, and Andrea Vedaldi. "Learning feed-forward one-shot learners". In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*. NIPS'16. Barcelona, Spain: Curran Associates Inc., 2016, 523–531. ISBN: 9781510838819.

[172] Shaowu Pan, Steven L Brunton, and J Nathan Kutz. "Neural Implicit Flow: a mesh-agnostic dimensionality reduction paradigm of spatio-temporal data". In: *Journal of Machine Learning Research* 24.41 (2023), pp. 1–60. URL: `http://jmlr.org/papers/v24/22-0365.html`.

[173] Filipe de Avila Belbute-Peres, Yi-fan Chen, and Fei Sha. "HyperPINN: Learning parameterized differential equations with physics-informed hypernetworks". In: *The Symbiosis of Deep Learning and Differential Equations*. Vol. abs/2111.01008. 2021. URL: `https://api.semanticscholar.org/CorpusID:240354714`.

[174] Alexander LeNail. "NN-SVG: Publication-Ready Neural Network Architecture Schematics". In: *Journal of Open Source Software* 4.33 (2019), p. 747. DOI: `10.21105/joss.00747`. URL: `https://doi.org/10.21105/joss.00747`.

[175] John Leask Lumley. "The structure of inhomogeneous turbulent flows". In: *Atmospheric turbulence and radio wave propagation* (1967), pp. 166–178.

[176] Peter J Schmid. "Dynamic mode decomposition of numerical and experimental data". In: *Journal of Fluid Mechanics* 656 (2010), pp. 5–28. DOI: `10.1017/S0022112010001217`.

[177] Joshua L Proctor, Steven L Brunton, and J Nathan Kutz. "Dynamic Mode Decomposition with Control". In: *SIAM Journal on Applied Dynamical Systems* 15.1 (2016), pp. 142–161. DOI: `10.1137/15M1013857`.

[178] Yann LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. "Efficient backprop". In: *Neural networks: Tricks of the trade*. Springer, 2002, pp. 9–50.

[179] Noboru Murata, Klaus-Robert Müller, Andreas Ziehe, and Shun-ichi Amari. "Adaptive on-line learning in changing environments". In: *Advances in Neural Information Processing Systems* 9 (1996). Ed. by M.C. Mozer, M. Jordan, and T. Petsche. URL: https://proceedings.neurips.cc/paper_files/paper/1996/file/0e095e054ee94774d6a496099eb1cf6a-Paper.pdf.

[180] Yuanzhi Li, Colin Wei, and Tengyu Ma. "Towards explaining the regularization effect of initial large learning rate in training neural networks". In: vol. 32. 2019. URL: https://proceedings.neurips.cc/paper/2019/file/bce9abf229ffd7e570818476ee5d7dde-Paper.pdf.

[181] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *arXiv preprint arXiv:1409.1556* (2014). DOI: https://doi.org/10.48550/arXiv.1409.1556.

[182] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[183] Kensuke Nakamura, Bilel Derbel, Kyoung-Jae Won, and Byung-Woo Hong. "Learning-Rate Annealing Methods for Deep Neural Networks". In: *Electronics* 10.16 (2021). DOI: 10.3390/electronics10162029.

[184] Christophe Geuzaine and Jean-François Remacle. "Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities". In: *International journal for numerical methods in engineering* 79.11 (2009), pp. 1309–1331. DOI: 10.1002/nme.2579.

[185] Robert Cimrman, Vladimír Lukeš, and Eduard Rohan. "Multiscale finite element calculations in Python using SfePy". In: *Advances in Computational Mathematics* (2019). ISSN: 1572-9044. DOI: 10.1007/s10444-019-09666-0. URL: https://doi.org/10.1007/s10444-019-09666-0.

[186] Nasim Rahaman, Aristide Baratin, Devansh Arpit, Felix Draxler, Min Lin, Fred Hamprecht, Yoshua Bengio, and Aaron Courville. "On the spectral bias of neu-

ral networks". In: *International Conference on Machine Learning*. PMLR. 2019, pp. 5301–5310. URL: http://proceedings.mlr.press/v97/rahaman19a/rahaman19a.pdf.

[187] Matthew Tancik, Pratul Srinivasan, Ben Mildenhall, Sara Fridovich-Keil, Nithin Raghavan, Utkarsh Singhal, Ravi Ramamoorthi, Jonathan Barron, and Ren Ng. "Fourier Features Let Networks Learn High Frequency Functions in Low Dimensional Domains". In: *Proceedings of the 34th International Conference on Neural Information Processing Systems*. NIPS'20. Vancouver, BC, Canada: Curran Associates Inc., 2020. ISBN: 9781713829546.

[188] Ali Rahimi and Benjamin Recht. "Random Features for Large-Scale Kernel Machines". In: 20 (2007). Ed. by J. Platt, D. Koller, Y. Singer, and S. Roweis. URL: https://proceedings.neurips.cc/paper_files/paper/2007/file/013a006f03dbc5392effeb8f18fda755-Paper.pdf.

[189] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. "Attention is All you Need". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[190] Tommy Odland. *tommyod/KDEpy: Kernel Density Estimation in Python*. Version v0.9.10. Dec. 2018. DOI: 10.5281/zenodo.2392268.

[191] B. W. Silverman. *Density Estimation for Statistics and Data Analysis*. London: Chapman & Hall, 1986.

[192] Syed R Ahmed, G Ramm, and Gunter Faltin. "Some Salient Features Of The Time-Averaged Ground Vehicle Wake". In: *SAE Transactions* (1984), pp. 473–503. URL: http://www.jstor.org/stable/44434262.

[193] R Verzicco, M Fatica, G Iaccarino, P Moin, and B Khalighi. "Large eddy simulation of a road vehicle with drag-reduction devices". In: *AIAA journal* 40.12 (2002), pp. 2447–2455. DOI: 10.2514/2.1613.

[194] Ilhan Bayraktar, Drew Landman, and Oktay Baysal. "Experimental and Computational Investigation of Ahmed Body for Ground Vehicle Aerodynamics". In: *SAE Transactions* (2001), pp. 321–331. ISSN: 0096736X, 25771531. URL: http://www.jstor.org/stable/44687433.

[195] RK Strachan, K Knowles, and NJ Lawson. *A CFD and experimental study of an Ahmed reference model.* Tech. rep. SAE Technical Paper, 2004. DOI: 10.4271/2004-01-0442.

[196] Neil Ashton, A West, S Lardeau, and Alistair Revell. "Assessment of RANS and DES methods for realistic automotive models". In: *Computers & fluids* 128 (2016), pp. 1–15. DOI: 10.1016/j.compfluid.2016.01.008.

[197] Hermann Lienhart and Stefan Becker. "Flow and turbulence structure in the wake of a simplified car model". In: *SAE Transactions* 112 (2003), pp. 785–796. URL: http://www.jstor.org/stable/44745451.

[198] Walter Meile, Günter Brenn, Aaron Reppenhagen, Bernhard Lechner, and Anton Fuchs. "Experiments and numerical simulations on the aerodynamics of the Ahmed body". In: *CFD Letters* 3.1 (2011), pp. 32–39.

[199] Tural Tunay, Besir Sahin, and Veli Ozbolat. "Effects of rear slant angles on the flow characteristics of Ahmed body". In: *Experimental Thermal and Fluid Science* 57 (2014), pp. 165–176. DOI: 10.1016/j.expthermflusci.2014.04.016.

[200] Florian R Menter. "Two-equation eddy-viscosity turbulence models for engineering applications". In: *AIAA journal* 32.8 (1994), pp. 1598–1605. DOI: 10.2514/3.12149.

[201] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. "Layer Normalization". In: (2016). DOI: 10.48550/arXiv.1607.06450. arXiv: 1607.06450 [stat.ML].

[202] Timothy Simpson, Farrokh Mistree, John Korte, and Timothy Mauery. "Comparison of response surface and kriging models for multidisciplinary design optimization". In: *7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*. 1998. DOI: 10.2514/6.1998-4755.

[203] Nestor V Queipo, Raphael T Haftka, Wei Shyy, Tushar Goel, Rajkumar Vaidyanathan, and P Kevin Tucker. "Surrogate-based analysis and optimization". In: *Progress in Aerospace Sciences* 41.1 (2005), pp. 1–28. DOI: 10.1016/j.paerosci.2005.02.001.

[204] Dominic A Masters, Nigel J Taylor, TCS Rendall, Christian B Allen, and Daniel J Poole. "Geometric Comparison of Aerofoil Shape Parameterization Methods". In: *AIAA journal* 55.5 (2017), pp. 1575–1589. DOI: 10.2514/1.J054943.

[205] Dominic A Masters, Daniel J Poole, Nigel J Taylor, Thomas Rendall, and Christian B Allen. "Impact of Shape Parameterisation on Aerodynamic Optimisation of Benchmark Problem". In: *54th AIAA Aerospace Sciences Meeting*. 2016. DOI: 10.2514/6.2016-1544.

[206] Zhoujie Lyu, Zelu Xu, and JRRA Martins. "Benchmarking Optimization Algorithms for Wing Aerodynamic Design Optimization". In: *Proceedings of the 8th International Conference on Computational Fluid Dynamics, Chengdu, Sichuan, China*. Vol. 11. ICCFD8-2014-0203. 2014, p. 585.

[207] Shigeru Obayashi and Takanori Tsukahara. "Comparison of optimization algorithms for aerodynamic shape design". In: *AIAA journal* 35.8 (1997), pp. 1413–1415. DOI: 10.2514/2.251.

[208] Jichao Li, Xiaosong Du, and Joaquim R.R.A. Martins. "Machine learning in aerodynamic shape optimization". In: *Progress in Aerospace Sciences* 134 (2022), p. 100849. DOI: 10.1016/j.paerosci.2022.100849.

[209] Jichao Li, Mohamed Amine Bouhlel, and Joaquim R. R. A. Martins. "Data-based approach for fast airfoil analysis and optimization". In: *AIAA Journal* 57.2 (2019), pp. 581–596. DOI: 10.2514/1.J057129.

[210] Wei Chen, Kevin Chiu, and Mark Fuge. “Aerodynamic Design Optimization and Shape Exploration using Generative Adversarial Networks”. In: *AIAA Scitech 2019 Forum*. 2019. DOI: 10.2514/6.2019-2351.

[211] Jichao Li, Mengqi Zhang, Joaquim R. R. A. Martins, and Chang Shu. “Efficient Aerodynamic Shape Optimization with Deep-Learning-Based Geometric Filtering”. In: *AIAA Journal* 58.10 (2020), pp. 4243–4259. DOI: 10.2514/1.J059254.

[212] Alejandro González Pérez, Christian B. Allen, and Daniel J. Poole. “GSA-SOM: A metaheuristic optimisation algorithm guided by machine learning and application to aerodynamic design”. In: *AIAA AVIATION 2021 FORUM*. 2021. DOI: 10.2514/6.2021-2563.

[213] Jonathan Viquerat, Jean Rabault, Alexander Kuhnle, Hassan Ghraieb, Aurélien Larcher, and Elie Hachem. “Direct shape optimization through deep reinforcement learning”. In: *Journal of Computational Physics* 428 (2021), p. 110080. DOI: 10.1016/j.jcp.2020.110080.

[214] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. “Practical Bayesian Optimization of Machine Learning Algorithms”. In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper/2012/file/05311655a15b75fab86956663e1819cd-Paper.pdf.

[215] Soumalya Sarkar, Sudeepta Mondal, Michael Joly, Matthew E Lynch, Shaunak D Bopardikar, Ranadip Acharya, and Paris Perdikaris. “Multifidelity and Multiscale Bayesian Framework for High-Dimensional Engineering Design and Calibration”. In: *Journal of Mechanical Design* 141.12 (2019). DOI: 10.1115/1.4044598.

[216] Michael Joly, Soumalya Sarkar, and Dhagash Mehta. “Machine Learning Enabled Adaptive Optimization of a Transonic Compressor Rotor With Precompression”. In: *Journal of Turbomachinery* 141.5 (2019). DOI: 10.1115/1.4041808.

[217] Matthew E Lynch, Soumalya Sarkar, and Kurt Maute. "Machine Learning to Aid Tuning of Numerical Parameters in Topology Optimization". In: *Journal of Mechanical Design* 141.11 (2019). DOI: 10.1115/1.4044228.

[218] Michael A. Gelbart, Jasper Snoek, and Ryan P. Adams. "Bayesian Optimization with Unknown Constraints". In: UAI'14. Quebec City, Quebec, Canada: AUAI Press, Arlington, Virginia, USA, 2014, 250–259. ISBN: 9780974903910.

[219] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *Nature* 521.7553 (2015), pp. 436–444. DOI: 10.1038/nature14539.

[220] Steven L. Brunton, Bernd R. Noack, and Petros Koumoutsakos. "Machine Learning for Fluid Mechanics". In: *Annual Review of Fluid Mechanics* 52.1 (2020), pp. 477–508. DOI: 10.1146/annurev-fluid-010719-060214.

[221] II William Lamarsh, Joanne Walsh, and James Rogers. "Aerodynamic performance optimization of a rotor blade using a neural network as the analysis". In: *4th Symposium on Multidisciplinary Analysis and Optimization.* 1992. DOI: 10.2514/6.1992-4837.

[222] S Huang, L Miller, and J Steck. "An exploratory application of neural networks to airfoil design". In: *32nd Aerospace Sciences Meeting and Exhibit.* 1994. DOI: 10.2514/6.1994-501.

[223] Magnus Norgaard, Charles C Jorgensen, and James C Ross. *Neural network prediction of new aircraft design coefficients.* Tech. rep. NASA-TM-112197. 1997.

[224] Roxana M Greenman. *Two-dimensional high-lift aerodynamic optimization using neural networks.* NASA-TM-1998-112233. Stanford University, 1998.

[225] Man Rai, Nateri Madavan, and Frank Huber. "Improving the unsteady aerodynamic performance of transonic turbines using neural networks". In: *38th Aerospace Sciences Meeting and Exhibit.* 2000. DOI: 10.2514/6.2000-169.

[226] Man Mohan Rai and Nateri K Madavan. "Application of Artificial Neural Networks to the Design of Turbomachinery Airfoils". In: *Journal of Propulsion and Power* 17 (2001), pp. 176–183. DOI: 10.2514/2.5725.

[227] Athar Kharal and Ayman Saleem. “Neural networks based airfoil generation for a given Cp using Bezier–PARSEC parameterization”. In: *Aerospace Science and Technology* 23.1 (2012), pp. 330–344. DOI: 10.1016/j.ast.2011.08.010.

[228] Abdurrahman Hacioglu. “Fast Evolutionary Algorithm for Airfoil Design via Neural Network”. In: *AIAA journal* 45.9 (2007), pp. 2196–2203. DOI: 10.2514/1.24484.

[229] Vinothkumar Sekar, Mengqi Zhang, Chang Shu, and Boo Cheong Khoo. “Inverse Design of Airfoil Using a Deep Convolutional Neural Network”. In: *AIAA Journal* 57.3 (2019), pp. 993–1003. DOI: 10.2514/1.J057894.

[230] Gang Sun, Yanjie Sun, and Shuyue Wang. “Artificial neural network based inverse design: Airfoils and wings”. In: *Aerospace Science and Technology* 42 (2015), pp. 415–428. DOI: 10.1016/j.ast.2015.01.030.

[231] Ruo-Lin Liu, Yue Hua, Zhi-Fu Zhou, Yubai Li, Wei-Tao Wu, and Nadine Aubry. “Prediction and optimization of airfoil aerodynamic performance using deep neural network coupled Bayesian method”. In: *Physics of Fluids* 34.11 (2022), p. 117116. DOI: 10.1063/5.0122595.

[232] Xinshuai Zhang, Fangfang Xie, Tingwei Ji, Zaoxu Zhu, and Yao Zheng. “Multifidelity deep neural network surrogate model for aerodynamic shape optimization”. In: *Computer Methods in Applied Mechanics and Engineering* 373 (2021), p. 113485. DOI: 10.1016/j.cma.2020.113485.

[233] Nilay Papila, Wei Shyy, Lisa Griffin, and Daniel Dorney. “Shape optimization of supersonic turbines using response surface and neural network methods”. In: *39th Aerospace Sciences Meeting and Exhibit*. 2016. DOI: 10.2514/6.2001-1065.

[234] Manas Khurana, Hadi Winarto, and Arvind Sinha. “Application of Swarm Approach and Artificial Neural Networks for Airfoil Shape Optimization”. In: *12th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*. 2008. DOI: 10.2514/6.2008-5954.

[235] Yao Zhang, Woong Je Sung, and Dimitri N Mavris. "Application of Convolutional Neural Network to Predict Airfoil Lift Coefficient". In: *2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference.* 2018. DOI: 10.2514/6.2018-1903.

[236] Rebecca Zahn, Maximilian Winter, Moritz Zieher, and Christian Breitsamter. "Application of a long short-term memory neural network for modeling transonic buffet aerodynamics". In: *Aerospace Science and Technology* 113 (2021), p. 106652. DOI: 10.1016/j.ast.2021.106652.

[237] AJ Torregrosa, LM García-Cuevas, P Quintero, and A Cremades. "On the application of artificial neural network for the development of a nonlinear aeroelastic model". In: *Aerospace Science and Technology* 115 (2021), p. 106845. DOI: 10.1016/j.ast.2021.106845.

[238] Gang Sun and Shuyue Wang. "A review of the artificial neural network surrogate modeling in aerodynamic design". In: *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering* 233.16 (2019), pp. 5863–5872. DOI: 10.1177/0954410019864485.

[239] Moritz Krügener, Jose Felix Zapata Usandivaras, Michaël Bauerheim, and Annafederica Urbano. "Coaxial-Injector Surrogate Modeling Based on Reynolds-Averaged Navier–Stokes Simulations Using Deep Learning". In: *Journal of Propulsion and Power* 38.5 (2022), pp. 783–798. DOI: 10.2514/1.B38696.

[240] Jiawei Hu and Weiwei Zhang. "Flow field modeling of airfoil based on convolutional neural networks from transform domain perspective". In: *Aerospace Science and Technology* 136 (2023), p. 108198. DOI: 10.1016/j.ast.2023.108198.

[241] Octavi Obiols-Sales, Abhinav Vishnu, Nicholas Malaya, and Aparna Chandramowliswharan. "CFDNet: A Deep Learning-Based Accelerator for Fluid Simulations". In: *Proceedings of the 34th ACM International Conference on Supercomputing.* 2020. DOI: 10.1145/3392717.3392772.

[242] Tianyuan Liu, Yunzhu Li, Qi Jing, Yonghui Xie, and Di Zhang. “Supervised learning method for the physical field reconstruction in a nanofluid heat transfer problem”. In: *International Journal of Heat and Mass Transfer* 165 (2021), p. 120684. DOI: 10.1016/j.ijheatmasstransfer.2020.120684.

[243] Rainer Storn and Kenneth Price. “Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces”. In: *Journal of global optimization* 11.4 (1997), pp. 341–359. DOI: 10.1023/A:1008202821328.

[244] Nateri K Madavan. “Multiobjective optimization using a Pareto differential evolution approach”. In: *Proceedings of the 2002 Congress on Evolutionary Computation. CEC’02 (Cat. No.02TH8600).* Vol. 2. 2002, 1145–1150 vol.2. DOI: 10.1109/CEC.2002.1004404.

[245] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. “A fast and elitist multiobjective genetic algorithm: NSGA-II”. In: *IEEE Transactions on Evolutionary Computation* 6.2 (2002), pp. 182–197. DOI: 10.1109/4235.996017.

[246] Connor Shorten and Taghi M Khoshgoftaar. “A survey on Image Data Augmentation for Deep Learning”. In: *Journal of Big Data* 6.1 (2019), pp. 1–48. DOI: 10.1186/s40537-019-0197-0.

[247] Ellen D. Zhong, Tristan Bepler, Joseph H. Davis, and Bonnie Berger. “Reconstructing continuous distributions of 3D protein structure from cryo-EM images”. In: *International Conference on Learning Representations.* 2020. URL: https://openreview.net/forum?id=SJxUjlBtwB.

[248] Arthur Jacot, Franck Gabriel, and Clément Hongler. “Neural tangent kernel: Convergence and generalization in neural networks”. In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems.* Vol. 31. NIPS’18. Red Hook, NY, USA: Curran Associates Inc., 2018.

[249] Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks”. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics.* Proceedings of Machine Learning

Research. PMLR, 2010, pp. 249–256. URL: https://proceedings.mlr.press/v9/glorot10a.html.