Automatic Designs in Deep Neural Networks

by

Lanlan Liu

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2020

Lanlan Liu

llanlan@umich.edu

ORCID iD: 0000-0002-9776-5768

To the journey

ACKNOWLEDGMENTS

First I would like to give special thanks to my advisor, Professor Jia Deng. It has been an honor and privilege to be his student. I am grateful for his great support along the journey and amazing guidance that teaches me a lot in research and academic life.

I would also thank the committee members Professor Satinder Singh, Professor Jason Corso and Professor Honglak Lee. It is a great honor to have them on my dissertation committee.

I am also grateful to my awesome colleagues in our Vision and Learning lab: Weifeng Chen, Ankit Goyal, Darby Haller, Hei Law, Lahav Lipson, Alejandro Newell, Jonathan Stroud, Zachary Teed, Emily Walters, Mingzhe Wang, Dawei Yang and Kaiyu Yang. They are great companions and supporters along this long journey.

I would also like to thank the mentors in my industrial experiences: Thomas Pfister, Jia Li at Google; Yuting Zhang, Stefano Soatto, Pietro Perona at Amazon. Their guidance is valuable for my PhD journey and the coming industrial career. My amazing colleagues in Google and Amazon: Michael Mully, Lu Jiang, Donghoon Lee, Ming-Hsuan Yang, Wei Wei, Nam Vo, Shasha Li, Zezhou Chen, Chenxi Liu, Jiyang Gao, Ankan Bansal, Rui Zhang and many others, have also brought me great fun and interesting discussions.

Great thanks also go to my friends and family who give me their full support in both the happy and sour moments of the PhD journey: Yuhui Wang, Tianjie Wang, Ke Yu, Bowen Wei, Weier Wan, Yuan Xu, Xuke Zhai, Ruxin Zhang, Yuqi Wang, Chen Shao, Chuhang Zou, Lily Chen, Jia Li, Shang Zhang, Fangting Xia, Shike Mei, Fei Tian, Yin Xi and many others.

At last, my greatest thanks go to my parents and Donghoon, who support me with their full love and encouragement that gave me great power to get through all, even the darkest moments.

TABLE OF CONTENTS

Dedication . . . . **ii**

Acknowledgments . . . . **iii**

List of Figures . . . . **vii**

List of Tables . . . . **x**

Abstract . . . . **xi**

Chapter

1 Introduction **1**

1.1 Background: Design in Deep Neural Networks 1
1.2 Motivation 2
1.3 Contributions 4
1.4 Thesis Outline 5
1.4.1 Automated Dynamic Inference (Chapter 2) 5
1.4.2 Unified Loss Design (Chapter 3) 6
1.4.3 Synthetic Data Generation for Small-data Object Detection (Chapter 4) . 6
1.4.4 Architecture Search for GANs (Chapter 5) 7
1.5 Related Work 7
1.5.1 Automated Data Design 7
1.5.2 Automated Model Design 8
1.5.3 Automated Loss Design 9

2 Automated Dynamic Inference **10**

2.1 Introduction 10
2.2 Related work 12
2.3 Definition and Semantics of D^{2}NNs 14
2.3.1 D^{2}NN definition 14
2.3.2 D^{2}NN Semantics 15
2.4 D^{2}NN Learning 17

2.4.1 Learning a Single Control Node 17
2.4.2 Mini-Bags for Set-Based Metrics 18
2.5 Experiments 22
2.5.1 High-Low Capacity D^{2}NN 22
2.5.2 Cascade D^{2}NN 23
2.5.3 Chain D^{2}NN 24
2.5.4 Hierarchical D^{2}NN 25
2.5.5 Comparison with Dynamic Capacity Networks 26
2.5.6 Visualization of Examples in Different Paths 27
2.5.7 CIFAR-10 Results 27
2.6 Conclusion 28
2.7 Appendix 28
2.7.1 Implementation Details 28
2.7.2 ILSVRC-10 Semantic Hierarchy 29
2.7.3 Configurations 29

3 Unified Loss Design 35

3.1 Introduction 35
3.2 Related Work 39
3.2.1 Direct Loss Minimization 39
3.2.2 Surrogate Losses 41
3.3 UniLoss 42
3.3.1 Overview 42
3.3.2 Example: Multi-class Classification 44
3.3.3 Refactorization 45
3.3.4 Interpolation 46
3.4 Experimental Results 47
3.4.1 Tasks and Metrics 48
3.4.2 Average Precision for Unbalanced Binary Classification 49
3.4.3 PCKh for Single-Person Pose Estimation 50
3.4.4 Accuracy for Multi-class Classification 52
3.4.5 Discussion of Hyper-parameters 53
3.5 Conclusions 54
3.6 Appendix 55
3.6.1 Binary Classification 55
3.6.2 Pose Estimation 55
3.6.3 Classification 56

4 Synthetic Data Generation for Small-data Object Detection 58

4.1 Introduction 58
4.2 Related Work 61
4.2.1 Image-to-image Translation. 61
4.2.2 Object Insertion with GANs. 61
4.2.3 Integration of GANs and Classifiers. 61
4.2.4 Data Augmentation for Object Detection 62

4.3 DetectorGAN . . . 63
4.3.1 Model Architecture . . . 63
4.3.2 Train Generator with Detection Losses . . . 64
4.3.3 Overall Losses and Training . . . 67
4.4 Experiments . . . 69
4.4.1 Disease Localization . . . 71
4.4.2 Pedestrian Detection . . . 74
4.5 Conclusion . . . 75

5 Architecture Search for GANs . . . 76

5.1 Introduction . . . 76
5.2 Related Work . . . 78
5.2.1 Improving GANs . . . 78
5.2.2 Architecture Search with GANs . . . 79
5.2.3 Neural Architecture Search . . . 80
5.2.4 Hybrid Optimization . . . 80
5.3 Dynamically Grown GAN . . . 81
5.3.1 Dynamic Growing Overview . . . 81
5.3.2 Search Algorithm . . . 84
5.4 Experimental Results . . . 84
5.4.1 Datasets and Evaluation Metrics . . . 85
5.4.2 CIFAR-10 . . . 86
5.4.3 LSUN . . . 89
5.5 Analysis . . . 90
5.5.1 How does the capacity of the generator (G) and the discriminator (D) affect training? . . . 91
5.5.2 How does each action affect training? . . . 91
5.5.3 How does greedy pruning affect searching? . . . 93
5.5.4 Discussion on Efficiency . . . 94
5.6 Conclusions . . . 95
5.7 Appendix . . . 95
5.7.1 Additional Analysis . . . 95

6 Limitations and Future Work . . . 99

6.1 Integration of Multiple Automated Techniques . . . 99
6.2 Unified Loss . . . 100
6.3 Architecture Search . . . 100
6.4 Complete Automated System . . . 100

Bibliography . . . 102

LIST OF FIGURES

2.1 Two D^{2}NN examples. Input and output nodes are drawn as circles with the output nodes shaded. Function nodes are drawn as rectangles (regular nodes) or diamonds (control nodes). Dummy nodes are shaded. Data edges are drawn as solid arrows and control edges as dashed arrows. A data edge with a user defined default value is decorated with a circle. 11

2.2 The accuracy-cost or fscore-cost curves of various D^{2}NN architectures, as well as conventional DNN baselines consisting of only regular nodes. 20

2.3 Four different D^{2}NN architectures. 21

2.4 Distribution of examples going through different execution paths. Skipped nodes are in grey. The hyperparameter λ controls the trade-off between accuracy and efficiency. A bigger λ values accuracy more. *Left*: for the high-low capacity D^{2}NN. *Right*: for the hierarchical D^{2}NN. The X-axis is the number of nodes activated. 24

2.5 Examples with different paths in a high-low D^{2}NN (left) and a hierarchical D^{2}NN (right). 24

2.6 Accuracy-cost curve for a chain D^{2}NN on the CMNIST task compared to DCN. 25

2.7 The semantic class hierarchy of the ILSVRC-10 dataset. 29

3.1 Computation graphs for conventional losses and UniLoss. Top: (a) testing for conventional losses. The decoder and evaluator are usually non-differentiable. (b) training for conventional losses. To avoid the non-differentiability, conventional methods optimize a manually-designed differentiable loss function instead during training. Bottom: (a) refactorized testing in UniLoss. We refactorize the testing so that the non-differentiability exists only in Sign($\cdot$) and the multi-variant function. (b) training in UniLoss with the differentiable approximation of refactorized testing. $\sigma(\cdot)$ is the sigmoid function. We approximate the non-differentiable components in the refactorized testing pipeline with interpolation methods. 36

4.1 DetectorGAN generates object-inserted images as synthesized data to improve the detection performance. DetectorGAN integrates a detector into the generator-discriminator loop. 59

4.2 The illustration for Eqn. 4.3 – unrolling one forward-backward pass for training DET to bridge the link between G_X and L_{Det}^{real} (detection loss on real images). Detection loss on real images has no direct link to the generator G_X. Last step of training old DET (noted as DET' in the figure, refers to same DET module but in the previous training step) is unrolled as in the dotted rectangle. The red arrow represents the fact that there is a differentiable link between G_X and L_{Det}^{real} after the unrolling. 65

4.3 Example generated images from CycleGAN and DetectorGAN for NIH. The details are high-lighted in green boxes (added for visualization). Both methods generate synthetic images from clean images and bounding box masks. DetectorGAN generates nodule inserted images with better local and global quality. 69
4.4 Examples of generated synthetic images from PS-GAN and DetectorGAN. The details are high-lighted in green boxes (added for visualization). We see qualitative improvement for pedestrian task as well. 69
4.5 Comparison showing that adding synthetic images can help detect nodules in NIH Chest X-ray more accurately. Here, green boxes are ground truth and red are predictions. 70

5.1 Overview of the Dynamically Grown GAN. **Bottom**: The dynamic growing process. We alternate between growing the architecture and training the weights of the new architectures. Each growing step chooses among actions including growing the generator (G) with a certain convolution layer, or growing discriminator (D) with a certain convolution layer, or growing both G and D to a higher resolution. In each training step, the new architectures inherit the weights from the old architectures. **Top**: Examples of growing steps. 81
5.2 Examples of generated LSUN Images by ProgGAN and our DGGAN at 256×256 resolution. **Top**: ProgGAN, obtained from their paper. It is not stated whether it is randomly generated; **Bottom**: Our DGGAN, randomly generated. 88
5.3 Examples of generated CIFAR-10 Images by ProgGAN, AutoGAN and our DGGAN, from left to right. ProgGAN and our DGGAN are randomly generated. AutoGAN images are obtained from their paper, which is also randomly generated. 88
5.4 G-to-D number of parameters ratio and normalized FID for candidates on CIFAR-10. **Left**: all the candidates. **Right**: zoom-in to candidates with normalized FID smaller than 1.2. The red arrows show the growing process of the best performing GAN. . . . 92
5.5 How each action affects training, i.e. how much is the improvement of normalized FID over parent. Actions include "Upscale", which means increasing the resolution of both G and D by 2, "G, a, b", which means growing a layer in G with a as filter size and b as number of filters, "D, a, b", which means growing a layer in D with a as filter size and b as number of filters. **Top**: Percentage of candidates with positive improvement over parent after each action. **Bottom**: Average and standard deviation of the improvement for each action. 92
5.6 **Left**: How likely good/sub-optimal parent candidates result in good child candidates with different normalized FID thresholds. The good candidates given a threshold are defined as candidates with normalized FID larger than the threshold and the sub-optimal ones are lower than the threshold. Each threshold holds for both parent candidates and child candidates. **Right** Simulation with different hyper-parameters p and K. 93
5.7 **Left**: G-to-D number of parameters ratio and normalized FID for all the candidates in LSUN. **Right**: zoom-in to candidates with normalized FID smaller than 1.2. 96

5.8 How each action affects training, i.e. how much is the improvement of normalized FID over parent. **Top**: Percentage of candidates with positive improvement over parent after each action. **Bottom**: Average and standard deviation of the improvement for each action. 97
5.9 **Left**: How likely good/sub-optimal parent candidates result in good child candidates with different normalized FID thresholds. **Right:** Simulation with different hyper-parameters p and K. 97

LIST OF TABLES

3.1 PCKh of Stacked Hourglass with MSE and UniLoss on the MPII validation. 51
3.2 Accuracy of ResNet-20 with CE loss and UniLoss on the CIFAR-10 and CIFAR-100 test set. 52
3.3 Ablation study for mini-batch sizes on CIFAR-10. 53
3.4 Ablation study for number of anchors on the three tasks. 53

4.1 Nodule AP on expanded annotation setting on the test set with IoU = 0.1 for NIH. Baseline is using only real training data. We add 1000 synthetic images from CycleGAN and GAN-D for training. 67
4.2 Localization accuracy with different T_{IOU} on “old annotations” test set for NIH. The latter three uses RetinaNet as the model. 67
4.3 Localization accuracy with different T_{IOBB} on the “old annotations” test set on NIH. . 70
4.4 User study on the NIH Chest X-ray dataset. 70
4.5 Pedestrian detection AP trained with real data, synthetic data generated by PS-GAN, pix2pix and DetectorGAN. 74

5.1 Quantitative evaluation on CIFAR-10, compared with ProgGAN at each resolution. . . 86
5.2 Quantitative evaluation on CIFAR-10, compared to prior state-of-the-art works. Top: manually-designed GANs. Bottom: automatic GANs. Lower FID and higher Inception Score indicate better generation quality. 86
5.3 Quantitative evaluation on LSUN, compared with ProgGAN at each scale. 89
5.4 Quantitative evaluation on LSUN, compared to prior state-of-the-art works. SWD averages over multiple feature scales from 256 to 16. Lower FID and smaller SWD indicate better generation quality. All methods except COCO-GAN are non part-based GANs that learns to generate the whole images. COCO-GAN learns to generate parts of images instead. 89

ABSTRACT

To train a Deep Neural Network (DNN) that performs well for a task, many design steps are taken including data designs, model designs and loss designs. Despite that remarkable progress has been made in all these domains of designing DNNs, the unexplored design space of each component is still vast. That brings the research field of developing automated techniques to lift some heavy work from human researchers when exploring the design space. The automated designs can help human researchers to make massive or challenging design choices and reduce the expertise required from human researchers.

Much effort has been made towards automated designs of DNNs, including synthetic data generation, automated data augmentation, neural architecture search and so on. Despite the huge effort, the automation of DNN designs is still far from complete. This thesis contributes in two ways: identifying new problems in the DNN design pipeline that can be solved automatically, and proposing new solutions to problems that have been explored by automated designs.

The first part of this thesis presents two problems that were usually solved with manual designs but can benefit from automated designs. To tackle the problem of inefficient computation due to using a static DNN architecture for different inputs, some manual efforts have been made to use different networks for different inputs as needed, such as cascade models. We propose an automated dynamic inference framework that can cut this manual effort and automatically choose different architectures for different inputs during inference. To tackle the problem of designing differentiable loss functions for non-differentiable performance metrics, researchers usually design the loss manually for each individual task. We propose an unified loss framework that reduces the amount of manual design of losses in different tasks.

The second part of this thesis discusses developing new techniques in domains where the automated design has been shown effective. In the synthetic data generation domain, we propose a novel method to automatically generate synthetic data for small-data object detection. The synthetic data generated can amend the limited annotated real data of the small-data object detection tasks, such as rare disease detection. In the architecture search domain, we propose an architecture search method customized for generative adversarial networks (GANs). GANs are commonly known unstable to train where we propose this new method that can stabilize the training of GANs in the architecture search process.

CHAPTER 1

Introduction

1.1 Background: Design in Deep Neural Networks

With the big success of deep neural networks (DNNs) in vision, language, recommendation and many other Machine Learning (ML) fields, lots of efforts are put into designing deep neural network models that perform better, in terms of better accuracy, efficiency, and generalization ability. To train a DNN that performs well for a task, especially a supervised task, designing the DNN architecture is not the only step. It starts from data collection where we collect raw data such as images, videos or words from all kinds of sources, and human workers annotate the collected data with the desired labels. We also augment the data with flipping, re-scaling and more to improve the generalization ability. Next, we design the architecture of the DNN to train. We also design a loss function and choose the optimization method and hyper-parameters. Finally, when we deploy the trained model to certain applications, we may use the trained DNN with certain post-processing steps and evaluate it with the performance metrics that we care about.

The whole pipeline can be categorized into three types of designs: data design, model design and loss design. More specifically, the data design include data collection, data annotation, data augmentation etc. The model design include architecture design, network compression, etc. The loss design include differentiable loss function design, optimization method design, hyper-parameters exploration and so on.

In each line of work, enormous new works have been proposed in recent years. For example, in data design, data augmentation techniques (Xu et al., 2016; Perez and Wang, 2017) improve the generalization ability of DNNs significantly; hard-example mining techniques (Shrivastava et al., 2016; Loshchilov and Hutter, 2015) improve effective use of data in unbalanced data situations.

In model design, revolutionary architectures have been proposed, including Inception-Net (Szegedy et al., 2015), residual networks (He et al., 2016), dense networks (Huang et al., 2017) and so on. To address the issue that networks are sometimes too computationally expensive for edge devices, network compression (Han et al., 2015; Denton et al., 2014; Chen et al., 2014; Gupta et al., 2015) and conditional computation (Bengio et al., 2015, 2013; Shazeer et al., 2017) techniques are proposed.

In loss design, with a large amount of new tasks emerging, exploration for both using existing loss functions for newer tasks or developing new loss functions is conducted (Taylor et al., 2008; Henderson and Ferrari, 2016; Liu et al., 2016). In addition, new optimization methods (Reddi et al., 2018; Kingma and Ba, 2015; Loshchilov and Hutter, 2019; Ma and Yarats, 2019) and hyper-parameter tuning techniques (Bergstra et al., 2011; Bardenet et al., 2013) are proposed for faster and better optimization of the losses.

1.2 Motivation

Despite that remarkable progress has been made in all these domains of designing DNNs, the unexplored design space of each component is still vast. With the deep neural networks helping to lift some tiring and massive work from human workers, for example automatically detecting irregular events, classifying inappropriate entertainment content, a natural question arises: can we also develop automated techniques to lift some heavy work from human researchers when exploring the design space of the deep neural networks?

This is one of the biggest motivations behind this thesis. We expect automated design of deep neural networks can help lift the heavy work from human researchers in two aspects:

1. Automated design help human researchers to make massive or challenging design choices;
2. Automated design reduce the expertise required for human researchers.

We illustrate how exactly automated design can help in these two aspects below.

Massive or Challenging Design Choices In each component of the DNN training, there remains large unexplored design spaces. To explore them, massive work that shares certain similar sub-processes needs to be done. With those sub-processes automated and even smartly-oriented, the labor of human researchers can be freed.

For example, in architecture design, there is a big space to choose which layers to have and how to connect layers. Automated design can be used to explore these choices. Neural Architecture Search (NAS) (Zoph et al., 2018; Zoph and Le, 2017) explores to search over these choices to construct neural networks for various tasks, including image classification (Real et al., 2019; Liu et al., 2019b, 2018), segmentation (Liu et al., 2019a; Chen et al., 2018), and others (So et al., 2019; Wong et al., 2018).

In addition to architecture design, other design choices such as hyper-parameters, can also be explored automatically (Snoek et al., 2015; Domhan et al., 2015).

There are also design decisions where it is challenging to accomplish with only human labors. For example, training data is often obtained by many human annotators collecting and annotating data. However, sometimes obtaining training data is challenging, either due to the rarity of the data itself, or difficulties to obtain high quality data. One common example is to collect medical related images – some diseases are naturally rare to observe, and to annotate these images, well-trained medical specialists is needed. In such cases, methods such as the automatic annotation (Murthy et al., 2015), synthetic data generation (Liang et al., 2017; Gurumurthy et al., 2017; Li et al., 2017a; Yang and Deng, 2018) are used to help.

Human Expertise Automated design can also reduce the level of expert knowledge required. For example, many of the model design require human expertise, such as how to choose model capacity with different amounts of data and how to select thresholds for cascade models. With the automated neural architecture search and hyper-parameter tuning techniques, less expertise is needed to train a model. For example, sometimes a simple start button can do all the hyper-parameter tuning process. This further enables broader application of DNNs in various domains. A typical application case is the AutoML service in the cloud industry, where small business holders are able to train image classifiers for their own business needs.

In loss design, to optimize for non-differentiable performance metrics that we care about, researchers often need to propose differentiable surrogate losses for network training. However, designing surrogate losses can sometimes incur substantial manual effort, including a large amount of the trial and error and the hyper-parameter tuning. This effort is important because a poorly designed loss can be misaligned with the final performance metric and lead to ineffective training. Automated design of surrogate losses such as learning to learn (Li and Malik, 2017; Chen et al., 2017) helps to reduce the level of expert knowledge required.

1.3 Contributions

Recent works towards automated design of DNNs include synthetic data generation, automated annotation, automated data augmentation, neural architecture search, automated hyper-parameter tuning and so on. Despite the huge effort, the automation of DNN design is still far from complete. Towards automated design of DNNs, this thesis contributes in two ways: 1) identifying new problems in the DNN design pipeline that can be solved automatically, such as automated dynamic inference and unified loss, and 2) proposing new solutions to problems that have been explored by automated designs, such as synthetic data generation and architecture search.

We identify problems that were usually solved with manual design but can benefit from automated design.

- To tackle the problem of inefficient computation due to using a static DNN architecture for different inputs, some manual efforts have been made to use different networks for different inputs as needed, such as cascade models. We propose the first **automated dynamic inference** framework that can cut this manual effort and automatically choose different architectures for different inputs during inference.

- To tackle the problem of designing differentiable loss functions for non-differentiable performance metrics, manual trials are made for each task by researchers. We propose the first **unified loss** framework that reduces the amount of manual design of losses in different tasks.

We also make efforts towards developing new techniques in domains where the automated design has been shown effective.

- In **synthetic data generation** domain, we propose a novel method to automatically generate synthetic data for small-data object detection.

- In **architecture search** domain, we propose an architecture search method customized for Generative Adversarial Networks (GANs).

1.4 Thesis Outline

This thesis first summarizes our motivation and contributions in Chapter 1. Chapter 2-5 illustrate in detail about the particular problems in automated design that we are solving and the solutions we propose. Chapter 6 further discusses the limitations and future works.

1.4.1 Automated Dynamic Inference (Chapter 2)

In this chapter, we use automated design to address the problem of unnecessary computation in inference. During the inference of a DNN model, the same amount of computation is used for every input because the DNN is static. However, this is often not necessary because each input may require different amounts of computation to be classified correctly (some are difficult to classify but some are easy). Manual efforts such as cascade models are used to reduce unnecessary computation in the past. Instead of manually-designed cascades, we introduce Dynamic Deep Neural Networks (D^2NN), a new type of feed-forward deep neural network that allows automatic dynamic inference. That is, given an input, only a subset of D^2NN neurons are executed, and the particular subset is determined by the D^2NN itself automatically. By pruning unnecessary computation depending on input, D^2NNs provide a way to improve computational efficiency. With extensive experiments of various D^2NN architectures on image classification tasks, we demonstrate that D^2NNs are general and flexible, and can effectively improve computational efficiency during inference. This chapter is based on a joint work with Jia Deng (Liu and Deng, 2018).

1.4.2 Unified Loss Design (Chapter 3)

In this chapter, we use automated design to reduce the manual efforts needed to design loss functions. The target performance metric for a task is often non-differentiable with respect to the DNN parameters. We thus need to design trainable surrogate losses that are differentiable and properly approximating the performance metrics. Large amount of manual effort is needed to design task-specific surrogate losses. To reduce this manual effort, we introduce UniLoss, a unified framework to generate surrogate losses for training deep networks with gradient descent. Our key observation is that in many cases, evaluating a model with a performance metric on a batch of examples can be refactored into four steps: from input to real-valued scores, from scores to comparisons of pairs of scores, from comparisons to binary variables, and from binary variables to the final performance metric. Using this refactorization we generate differentiable approximations for each non-differentiable step through interpolation. Using UniLoss, we can optimize for different tasks and metrics using one unified framework, achieving comparable performance compared with manually-designed losses. This chapter is based on a joint work with Mingzhe Wang and Jia Deng (Liu et al., 2020a).

1.4.3 Synthetic Data Generation for Small-data Object Detection (Chapter 4)

In this chapter, we explore a new method in synthetic data generation, where we generate synthetic data to amend real data collected by human annotators. In particular, we explore object detection in the small data regime, where only a limited number of annotated bounding boxes are available due to data rarity and annotation expense. We address this problem by learning to generate new synthetic images with associated bounding boxes, and using these for training an object detector. We show that simply training previously proposed generative models does not yield satisfactory performance due to them optimizing for image realism rather than object detection accuracy. To this end we develop a new model with a novel unrolling mechanism that jointly optimizes the generative model and a detector such that the generated images improve the performance of the detector. We show this method outperforms the state of the art on two challenging datasets, disease detection and small data pedestrian detection. This chapter is based on a joint work with Michael Muelly, Jia Deng, Tomas Pfister and Li-Jia Li (Liu et al., 2019c).

1.4.4 Architecture Search for GANs (Chapter 5)

In this chapter, we explore a new strategy in architecture search domain. While architecture search methods have been successfully applied in many problems, little effective effort has been made for GANs, especially considering the difficulties and instabilities to train large-scale GANs. In light of the recent work introducing progressive network growing as a promising way to ease the training for large GANs, we propose a method to dynamically grow GANs during training, optimizing the network architecture and its parameters together with automation. The method embeds architecture search techniques as an interleaving step with gradient-based training to periodically seek the optimal architecture-growing strategy for the generator and discriminator. It enjoys the benefits of both eased training because of the progressive growing and improved performance because of the broader architecture design space. Experimental results demonstrate state-of-the-art of image generation. Observations in the search procedure also provide constructive insights into the GAN model design such as the generator-discriminator balance and convolutional layer choices. This chapter is based on a joint work with Yuting Zhang, Jia Deng and Stefano Soatto (Liu et al., 2020b).

1.5 Related Work

1.5.1 Automated Data Design

With the enormous progress in the data design, especially in constructing larger and larger datasets (Deng et al., 2009; Lin et al., 2014; Cordts et al., 2016), automated assistance to obtain and annotate data becomes essential. *Automatic annotation* techniques (Murthy et al., 2015) have been developed to reduce the workload to manually annotate data with human labors. *Synthetic data generation* is one of the techniques to automatically obtain clean and annotated data in many domains including 3D reconstruction, semantic segmentation, object detection and so on (Liang et al., 2017; Gurumurthy et al., 2017; Li et al., 2017a; Yang and Deng, 2018). As the first few to explore synthetic data generation for the small-data object detection task, we propose a GAN-based method that is fundamentally different from previous work—we construct a direct feedback link between the image generator and the object detector while previous work has no such feedback.

1.5.2 Automated Model Design

Neural Architecture Search (NAS) (Zoph et al., 2018; Zoph and Le, 2017) explores to search neural network architectures for various tasks, including image classification (Real et al., 2019; Liu et al., 2019b, 2018), segmentation (Liu et al., 2019a; Chen et al., 2018), image generation with GANs (Wang and Huan, 2019; Gong et al., 2019) and others (So et al., 2019; Wong et al., 2018). For architecture search of GANs, we propose a new method that combines progressive growing with architecture search. Our new method is thus able to search for high-resolution (256×256) GANs while previous work (Wang and Huan, 2019; Gong et al., 2019) can search for only lower resolution (up to 48×48) GANs.

Most of these works focus on the accuracy performance. To improve the efficiency performance of neural networks, *learned network compression* (Han et al., 2015; Denton et al., 2014; Chen et al., 2014; Gupta et al., 2015) is proposed to eliminate redundancy in data or computation in a way that is input-independent. Also, *conditional computation* (Bengio et al., 2015, 2013; Shazeer et al., 2017) techniques are proposed to perform input-dependent pruning of the network. Along the line of input-dependent computation saving, we propose a new framework with *automatic dynamic inference*: with a large computation graph, the inference of each image involves a dynamic part of the computation graph and which part is determined by the model itself.

1.5.3 Automated Loss Design

Automated design of surrogate losses of non-differentiable performance metrics is proposed to ease the design process and reduce the level of expert knowledge required. Reinforcement learning algorithms have been also used to optimize performance metrics for structured output problems, especially those that can be formulated as taking a sequence of actions (Ranzato et al., 2016; Liu et al., 2017b; Caicedo and Lazebnik, 2015; Yeung et al., 2016; Zhou et al., 2018). *Learning to learn* methods (Li and Malik, 2017; Chen et al., 2017) use neural networks to learn a loss function using output-performance data pairs. We provide a new framework to propose surrogate losses in a unified way that does not involve substantial task-specific designs. It does not involve learning parameters as learning to learn methods or formulating the task as a sequence of actions as reinforcement learning methods.

In addition, automatic hyper-parameter tuning techniques (Snoek et al., 2015; Domhan et al., 2015) are proposed for faster and better optimization of the losses.

CHAPTER 2

Automated Dynamic Inference [1]

2.1 Introduction

In this chapter, we address the problem of unnecessary computation in inference. There has been need for inference computational efficiency, in particular, by the need to deploy deep networks on mobile devices and data centers. Mobile devices are constrained by energy and power, limiting the amount of computation that can be executed. Data centers need energy efficiency to scale to higher throughput and to save operating cost.

However, during the inference of a traditional DNN model, the same amount of computation is used for every input because the DNN is static. This is often not necessary because each input may requires different amount of computation to be classified correctly. For example, to classify whether an image contains a car or not, some images with a front view of the car may be easy to classify with a small network but some images with only a part of the car visible may be difficult and need a larger network.

Manual efforts such as cascade models (Li et al., 2015; Sun et al., 2013) are used to reduce unnecessary computation in the past. Instead of manually-designed cascades, we introduce Dynamic Deep Neural Networks (D^{2}NN), a new type of feed-forward deep neural network that allows automatic dynamic inference. That is, given an input, only a subset of neurons are executed, and the particular subset is determined by the network itself based on the particular input. In other words, the amount of computation and computation sequence are dynamic based on input. This is dif-

[1]This chapter is based on a joint work with Jia Deng

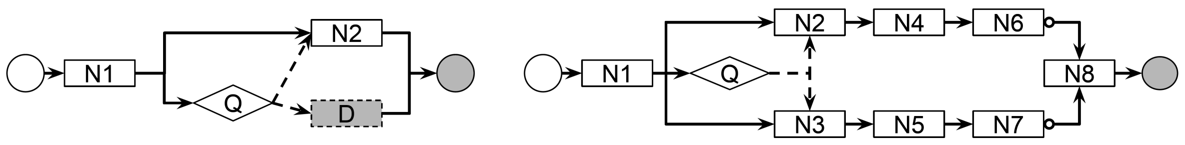


Figure 2.1: Two D^{2}NN examples. Input and output nodes are drawn as circles with the output nodes shaded. Function nodes are drawn as rectangles (regular nodes) or diamonds (control nodes). Dummy nodes are shaded. Data edges are drawn as solid arrows and control edges as dashed arrows. A data edge with a user defined default value is decorated with a circle.

ferent from standard feed-forward networks that always execute the same computation sequence regardless of input.

A D^{2}NN is a feed-forward deep neural network (directed acyclic graph of differentiable modules) augmented with one or more control modules. A control module is a sub-network whose output is a decision that controls whether other modules can execute. Fig. 2.1 (left) illustrates a simple D^{2}NN with one control module (Q) and two regular modules (N1, N2), where the controller Q outputs a binary decision on whether module N2 executes. For certain inputs, the controller may decide that N2 is unnecessary and instead execute a dummy node D to save on computation. As an example application, this D^{2}NN can be used for binary classification of images, where some images can be rapidly classified as negative after only a small amount of computation.

D^{2}NNs provide a way to improve computational efficiency by selective execution, pruning unnecessary computation depending on input. D^{2}NNs also make it possible to use a bigger network under a computation budget by executing only a subset of the neurons each time.

A D^{2}NN is trained end to end. That is, regular modules and control modules are jointly trained to optimize both accuracy and efficiency. We achieve such training by integrating backpropagation with reinforcement learning, necessitated by the non-differentiability of control modules.

Compared to prior work that optimizes computational efficiency in computer vision and machine learning, our work is distinctive in four aspects: (1) the decisions on selective execution are part of the network inference and are learned end to end together with the rest of the network, as opposed to hand-designed or separately learned (Li et al., 2015; Sun et al., 2013; Almahairi et al., 2016); (2) D^{2}NNs allow more flexible network architectures and execution sequences including parallel paths, as opposed to architectures with less variance (Denoyer and Gallinari, 2014; Shazeer et al., 2017); (3) our D^{2}NNs directly optimize arbitrary efficiency metric that is defined by the user, while previous work has no such flexibility because they improve efficiency indirectly through sparsity constraints (Bengio et al., 2015, 2013; Shazeer et al., 2017). (4) our method optimizes metrics such as the F-score that does not decompose over individual examples. This is an issue not addressed in prior work. We will elaborate on these differences in the Related Work section of this chapter.

We perform extensive experiments to validate our D^{2}NNs algorithms. We evaluate various D^{2}NN architectures on several tasks. They demonstrate that D^{2}NNs are general, flexible, and can effectively improve computational efficiency.

Our main contribution is the D^{2}NN framework that allows a user to augment a static feed-forward network with control modules to achieve automated dynamic inference. We show that D^{2}NNs allow a wide variety of topologies while sharing a unified training algorithm. To our knowledge, D^{2}NN is the first single automated framework that can support various qualitatively different efficient network designs, including cascade designs and coarse-to-fine designs.

2.2 Related work

Input-dependent execution has been widely used in computer vision, from cascaded detectors (Viola and Jones, 2004; Felzenszwalb et al., 2010) to hierarchical classification (Deng et al., 2011; Bengio et al., 2010). The key difference of our work from prior work is that we *jointly* learn both visual features and control decisions *end to end*, whereas prior work either hand-designs features and control decisions (e.g. thresholding), or learns them separately.

In the context of deep networks, two lines of prior work have attempted to improve computational efficiency. One line of work tries to eliminate redundancy in data or computation in a way that is input-independent. The methods include pruning networks (Han et al., 2015; Wen et al., 2016; Alvarez and Salzmann, 2016), approximating layers with simpler functions (Denton et al., 2014; Zhang et al., 2016), and using number representations of limited precision (Chen et al., 2014;

Gupta et al., 2015). The other line of work exploits the fact that not all inputs require the same amount of computation, and explores input-dependent execution of DNNs. Our work belongs to the second line, and we will contrast our work mainly with them. In fact, our input-dependent D^{2}NN can be combined with input-independent methods to achieve even better efficiency.

Among methods leveraging input-dependent execution, some use pre-defined execution-control policies. For example, cascade methods (Li et al., 2015; Sun et al., 2013) rely on manually-selected thresholds to control execution; Dynamic Capacity Network (Almahairi et al., 2016) designs a way to directly calculate a saliency map for execution control. Our D^{2}NNs, instead, are fully learn-able; the execution-control policies of D^{2}NNs do not require manual design and are learned together with the rest of the network.

Our work is closely related to conditional computation methods (Bengio et al., 2015, 2013; Shazeer et al., 2017), which activate part of a network depending on input. They learn policies to encourage sparse neural activations (Bengio et al., 2015) or sparse expert networks (Shazeer et al., 2017). Our work differs from these methods in several ways. First, our control policies are learned to directly optimize arbitrary user-defined global performance metrics, whereas conditional computation methods have only learned policies that encourage sparsity. In addition, D^{2}NNs allow more flexible control topologies. For example, in (Bengio et al., 2015), a neuron (or block of neurons) is the unit controllee of their control policies; in (Shazeer et al., 2017), an expert is the unit controllee. Compared to their fixed types of controllees, our control modules can be added in any point of the network and control arbitrary subnetworks. Also, various policy parametrization can be used in the same D^{2}NN framework. We show a variety of parameterizations (as different controller networks) in our D^{2}NN examples, whereas previous conditional computation works have used some fixed formats: For example, control policies are parametrized as the sigmoid or softmax of an affine transformation of neurons or inputs (Bengio et al., 2015; Shazeer et al., 2017).

Our work is also related to attention models (Denil et al., 2012; Mnih et al., 2014; Gregor et al., 2015). Note that attention models can be categorized as *hard* attention (Mnih et al., 2014; Ba et al., 2014; Almahairi et al., 2016) versus *soft* (Gregor et al., 2015; Stollenga et al., 2014). Hard attention models only process the salient parts and discard others (e.g. processing only a subset of image subwindows); in contrast, soft attention models process all parts but up-weight the salient parts. Thus only hard attention models perform input-dependent execution as D^{2}NNs do. However, hard attention models differ from D^{2}NNs because hard attention models have typically involved only one attention module whereas D^{2}NNs can have multiple attention (controller) modules — conventional hard attention models are "single-threaded" whereas D^{2}NN can be "multi-threaded". In addition, prior work in hard attention models have not directly optimized for accuracy-efficiency trade-offs. It is also worth noting that many mixture-of-experts methods (Jacobs et al., 1991; Jordan and Jacobs, 1994; Eigen et al., 2013) also involve soft attention by soft gating experts: they process all experts but only up-weight useful experts, thus saving no computation.

D^{2}NNs also bear some similarity to Deep Sequential Neural Networks (DSNN) (Denoyer and Gallinari, 2014) in terms of input-dependent execution. However, it is important to note that although DSNNs' structures can in principle be used to optimize accuracy-efficiency trade-offs, DSNNs are not for the task of improving efficiency and have no learning method proposed to optimize efficiency. And the method to effectively optimize for efficiency-accuracy trade-off is non-trivial as is shown in the following sections. Also, DSNNs are single-threaded: it always activates exactly one path in the computation graph, whereas for D^{2}NNs it is possible to have multiple paths or even the entire graph activated.

2.3 Definition and Semantics of D^{2}NNs

Here we precisely define a D^{2}NN and describe its semantics, i.e. how a D^{2}NN performs inference.

2.3.1 D^{2}NN definition

A D^{2}NN is defined as a directed acyclic graph (DAG) without duplicated edges. Each node can be one of the three types: input nodes, output nodes, and function nodes. An input or output node represents an input or output of the network (e.g. a vector). A function node represents a (differentiable) function that maps a vector to another vector. Each edge can be one of the two types: data edges and control edges. A data edge represents a vector sent from one node to another, the same as in a conventional DNN. A control edge represents a control signal, a scalar, sent from one node to another. A data edge can optionally have a user-defined "default value", representing the output that will still be sent even if the function node does not execute.

For simplicity, we have a few restrictions on valid D^2NNs: (1) the outgoing edges from a node are either all data edges or all control edges (i.e. cannot be a mix of data edges and control edges); (2) if a node has an incoming control edge, it cannot have an outgoing control edge. Note that these two simplicity constraints do not in any way restrict the expressiveness of a D^2NN. For example, to achieve the effect of a node with a mix of outgoing data edges and control edges, we can just feed its data output to a new node with outgoing control edges and let the new node be an identity function.

We call a function node a *control node* if its outgoing edges are control edges. We call a function node a *regular node* if its outgoing edges are data edges. Note that it is possible for a function node to take no data input and output a constant value. We call such nodes "dummy" nodes. We will see that the "default values" and "dummy" nodes can significantly extend the flexibility of D^2NNs. Hereafter we may also call function nodes "subnetwork", or "modules" and will use these terms interchangeably. Fig. 2.1 illustrates simple D^2NNs with all kinds of nodes and edges.

2.3.2 D^2NN Semantics

Given a D^2NN, we perform inference by traversing the graph starting from the input nodes. Because a D^2NN is a DAG, we can execute each node in a topological order (the parents of a node are ordered before it; we take both data edges and control edges in consideration), same as conventional DNNs except that the control nodes can cause the computation of some nodes to be skipped.

After we execute a control node, it outputs a set of control scores, one for each of its outgoing control edges. The control edge with the highest score is “activated”, meaning that the node being controlled is allowed to execute. The rest of the control edges are not activated, and their controllees are not allowed to execute. For example, in Fig 2.1 (right), the node Q controls N2 and N3. Either N2 or N3 will execute depending on which has the higher control score.

Although the main idea of the inference (skipping nodes) seems simple, due to D^2NNs’ flexibility, the inference topology can be far more complicated. For example, in the case of a node with multiple incoming control edges (i.e. controlled by multiple controllers), it should execute if any of the control edges are activated. Also, when the execution of a node is skipped, its output will be either the default value or null. If the output is the default value, subsequent execution will continue as usual. If the output is null, any downstream nodes that depend on this output will in turn skip execution and have a null output unless a default value has been set. This “null” effect will propagate to the rest of the graph. Fig. 2.1 (right) shows a slightly more complicated example with default values: if N2 skips execution and outputs null, so will N4 and N6. But N8 will execute regardless because its input data edge has a default value. In our Experiments Section, we will demonstrate more sophisticated D^2NNs.

We can summarize the semantics of D^2NNs as follows: a D^2NN executes the same way as a conventional DNN except that there are control edges that can cause some nodes to be skipped. A control edge is active if and only if it has the highest score among all outgoing control edges from a node. A node is skipped if it has incoming control edges and none of them is active, or if one of its inputs is null. If a node is skipped, its output will be either null or a user-defined default value. A null will cause downstream nodes to be skipped whereas a default value will not.

A D^2NN can also be thought of as a program with conditional statements. Each data edge is equivalent to a variable that is initialized to either a default value or null. Executing a function node is equivalent to executing a command assigning the output of the function to the variable. A control edge is equivalent to a boolean variable initialized to False. A control node is equivalent to a “switch-case” statement that computes a score for each of the boolean variables and sets the one with the largest score to True. Checking the conditions to determine whether to execute a function is equivalent to enclosing the function with an "if-then" statement. A conventional DNN is a program with only function calls and variable assignments without any conditional statements, whereas a D^{2}NN introduces conditional statements with the conditions themselves generated by learnable functions.

2.4 D^{2}NN Learning

Due to the control nodes, a D^{2}NN cannot be trained the same way as a conventional DNN. The output of the network cannot be expressed as a differentiable function of all trainable parameters, especially those in the control nodes. As a result, backpropagation cannot be directly applied. The main difficulty lies in the control nodes, whose outputs are discretized into control decisions. This is similar to the situation with hard attention models (Mnih et al., 2014; Ba et al., 2014), which use reinforcement learning. Here we adopt the same general strategy.

2.4.1 Learning a Single Control Node

For simplicity of exposition we start with a special case where there is only one control node. We further assume that all parameters except those of this control node have been learned and fixed. That is, the goal is to learn the parameters of the control node to maximize a user-defined reward, which in our case is a combination of accuracy and efficiency. This results in a classical reinforcement learning setting: learning a control policy to take actions so as to maximize reward. We base our learning method on Q-learning (Mnih et al., 2013; Sutton and Barto). We let each outgoing control edge represent an action, and let the control node approximate the action-value (Q) function, which is the expected return of an action given the current state (the input to the control node).

It is worth noting that unlike many prior works that use deep reinforcement learning, a D^{2}NN is not recurrent. For each input to the network (e.g. an image), each control node only executes once. And the decisions of a control node completely depend on the current input. As a result, an action taken on one input has no effect on another input. That is, our reinforcement learning task consists of only one time step. Our one time-step reinforcement learning task can also be seen as a contextual bandit problem, where the context vector is the input to the control module, and the arms are the possible action outputs of the module. The one time-step setting simplifies our Q-learning objective to that of the following regression task:

$$L = (Q(\boldsymbol{s}, \boldsymbol{a}) - r)^2, \tag{2.1}$$

where r is a user-defined reward, $\boldsymbol{a}$ is an action, $\boldsymbol{s}$ is the input to control node, and Q is computed by the control node. As we can see, training a control node here is the same as training a network to predict the reward for each action under an L2 loss. We use mini-batch gradient descent; for each training example in a mini-batch, we pick the action with the largest Q, execute the rest of the network, observe a reward, and perform backpropagation using the L2 loss in Eqn. 2.1.

During training we also perform ϵ-greedy exploration — instead of always choosing the action with the best Q value, we choose a random action with probability ϵ. The hyper-parameter ϵ is initialized to 1 and decreases over time. The reward r is user defined. Since our goal is to optimize the trade-off between accuracy and efficiency, in our experiments we define the reward as a combination of an accuracy metric A (for example, F-score) and an efficiency metric E (for example, the inverse of the number of multiplications), that is, $\lambda A + (1 - \lambda)E$ where λ balances the trade-off.

2.4.2 Mini-Bags for Set-Based Metrics

Our training algorithm so far has defined the state as a single training example, i.e., the control node takes actions and observes rewards on each training example independent of others. This setup, however, introduces a difficulty for optimizing for accuracy metrics that cannot be decomposed over individual examples.

Consider *precision* in the context of binary classification. Given predictions on a set of examples and the ground truth, precision is defined as the proportion of true positives among the predicted positives. Although precision can be defined on a single example, precision on a set of examples does not generally equal the average of the precisions of individual examples. In other words, precision as a metric does not decompose over individual examples and can only be computed using a set of examples *jointly*. This is different from decomposable metrics such as *error rate*, which can be computed as the average of the error rates of individual examples. If we use precision as our accuracy metric, it is not clear how to define a reward independently for each example such that maximizing this reward independently for each example would optimize the overall precision. In general, for many metrics, including precision and F-score, we cannot compute them on individual examples and average the results. Instead, we must compute them using a set of examples as a whole. We call such metrics “set-based metrics”. Our learning setup so far is ill-equipped for such metrics because a reward is defined on each example independently.

To address this issue we generalize the definition of a state from a single input to a set of inputs. We define such a set of inputs as a *mini-bag*. With a mini-bag of images, any set-based metric can be computed and can be used to directly define a reward. Note that a mini-bag is different from a mini-batch which is commonly used for batch updates in gradient decent methods. Actually in our training, we calculate gradients using a mini-batch of mini-bags. Now, an action on a mini-bag $\mathbf{s} = (s_1, \ldots, s_m)$ is now a joint action $\mathbf{a} = (a_1, \ldots, a_m)$ consisting of individual actions a_i on example s_i. Let $\mathbf{Q}(\mathbf{s}, \mathbf{a})$ be the *joint* action-value function on the mini-bag $\mathbf{s}$ and the joint action $\mathbf{a}$. We constrain the parametric form of $\mathbf{Q}$ to decompose over individual examples:

$$\mathbf{Q} = \sum_{i=1}^{m} Q(s_i, a_i), \tag{2.2}$$

where $Q(s_i, a_i)$ is a score given by the control node when choosing the action a_i for example s_i. We then define our new learning objective on a mini-bag of size m as

$$L = (r - \mathbf{Q}(\mathbf{s}, \mathbf{a}))^2 = (r - \sum_{i=1}^{m} Q(s_i, a_i))^2, \tag{2.3}$$

where r is the reward observed by choosing the joint action $\mathbf{a}$ on mini-bag $\mathbf{s}$. That is, the control

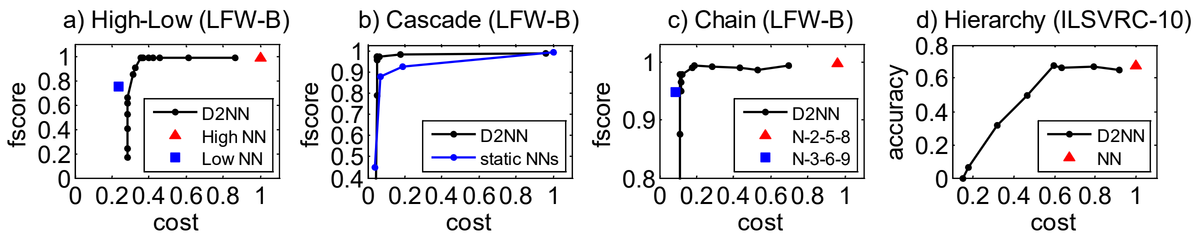


Figure 2.2: The accuracy-cost or fscore-cost curves of various D^{2}NN architectures, as well as conventional DNN baselines consisting of only regular nodes.

node predicts an action-value for each example such that their sum approximates the reward defined on the whole mini-bag.

It is worth noting that the decomposition of $\mathbf{Q}$ into sums (Eqn. 2.2) enjoys a nice property: the best joint action $\mathbf{a}^*$ under the joint action-value $\mathbf{Q}(\mathbf{s}, \mathbf{a})$ is simply the concatenation of the best actions for individual examples because maximizing

$$\mathbf{a}^* = \arg\max_{\mathbf{a}}(\mathbf{Q}(\mathbf{s}, \mathbf{a})) = \arg\max_{\mathbf{a}}(\sum_{i=1}^{m} Q(s_i, a_i)) \tag{2.4}$$

is equivalent to maximizing the individual summands:

$$a_i^* = \arg\max_{a_i} Q(s_i, a_i), i = 1, 2...m. \tag{2.5}$$

That is, during test time we still perform inference on each example independently.

Another implication of the mini-bag formulation is:

$$\frac{\partial L}{\partial x_i} = 2(r - \sum_{j=1}^{m} Q(s_j, a_j))\frac{\partial Q(s_i, a_i)}{\partial x_i}, \tag{2.6}$$

where x_i is the output of any internal neuron for example i in the mini-bag. This shows that there is no change to the implementation of backpropagation except that we scale the gradient using the difference between the mini-bag Q-value $\mathbf{Q}$ and reward r.

Joint Training of All Nodes We have described how to train a single control node. We now describe how to extend this strategy to all nodes including additional control nodes as well as

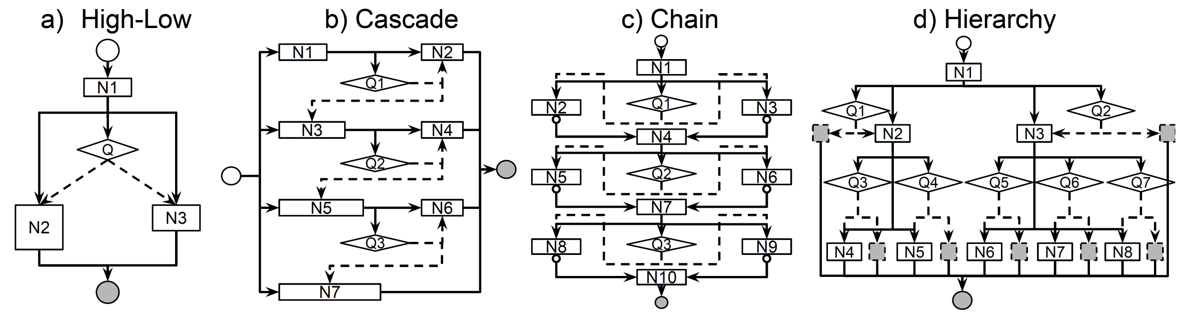


Figure 2.3: Four different D^{2}NN architectures.

regular nodes. If a D^{2}NN has multiple control nodes, we simply train them together. For each mini-bag, we perform backpropagation for multiple losses together. Specifically, we perform inference using the current parameters, observe a reward for the whole network, and then use the same reward (which is a result of the actions of all control nodes) to backpropagate for each control node.

For regular nodes, we can place losses on them the same as on conventional DNNs. And we perform backpropagation on these losses together with the control nodes. The implementation of backpropagation is the same as conventional DNNs except that each training example have a different network topology (execution sequence). And if a node is skipped for a particular training example, then the node does not have a gradient from the example.

It is worth noting that our D^{2}NN framework allows arbitrary losses to be used for regular nodes. For example, for classification we can use the cross-entropy loss on a regular node. One important detail is that the losses on regular nodes need to be properly weighted against the losses on the control nodes; otherwise the regular losses may dominate, rendering the control nodes ineffective. One way to eliminate this issue is to use Q-learning losses on regular nodes as well, i.e. treating the outputs of a regular node as action-values. For example, instead of using the cross-entropy loss on the classification scores, we treat the classification scores as action-values—an estimated reward of each classification decision. This way Q-learning is applied to all nodes in a unified way and no additional hyperparameters are needed to balance different kinds of losses. In our experiments unless otherwise noted we adopt this unified approach.

2.5 Experiments

We here demonstrate four D^{2}NN structures motivated by different demands of efficient network design to show its flexibility and effectiveness, and compare D^{2}NNs' ability to optimize efficiency-accuracy trade-offs with prior work.

We implement the D^{2}NN framework in Torch. Torch provides functions to specify the subnetwork architecture inside a function node. Our framework handles the high-level communication and loss propagation.

2.5.1 High-Low Capacity D^{2}NN

Our first experiment is with a simple D^{2}NN architecture that we call "high-low capacity D^{2}NN". It is motivated by that we can save computation by choosing a low-capacity subnetwork for easy examples. It consists of a single control nodes (Q) and three regular nodes (N1-N3) as in Fig. 2.3a). The control node Q chooses between a high-capacity N2 and a low-capacity N3; the N3 has fewer neurons and uses less computation. The control node itself has orders of magnitude fewer computation than regular nodes (this is true for all D^{2}NNs demonstrated).

We test this hypothesis using a binary classification task in which the network classifies an input image as face or non-face. We use the Labeled Faces in the Wild (Huang et al., 2007; Learned-Miller, 2014) dataset. Specifically, we use the 13k ground truth face crops (112$\times$112 pixels) as positive examples and randomly sampled 130k background crops (with an intersection over union less than 0.3) as negative examples. We hold out 11k images for validation and 22k for testing. We refer to this dataset as LFW-B and use it as a testbed to validate the effectiveness of our new D^{2}NN framework.

To evaluate performace we measure accuracy using the F1 score, a better metric than percentage of correct predictions for an unbalanced dataset. We measure computational cost using the number of multiplications following prior work (Almahairi et al., 2016; Shazeer et al., 2017) and for reproductivity. Specifically, we use the number of multiplications (control nodes included), normalized by a conventional DNN consisting of N1 and N2, that is, the high-capacity execution path. Note that our D^{2}NNs also allow to use other efficiency measurement such as run-time, latency.

During training we define the Q-learning reward as a linear combination of accuracy A and efficiency E (negative cost): $r = \lambda A + (1-\lambda)E$ where $\lambda \in [0,1]$. We train instances of high-low capacity D^{2}NNs using different λ's. As λ increases, the learned D^{2}NN trades off efficiency for accuracy. Fig. 2.2a) plots the accuracy-cost curve on the test set; it also plots the accuracy and efficiency achieved by a conventional DNN with only the high capacity path N1+N2 (High NN) and a conventional DNN with only the low capacity path N1+N3 (Low NN).

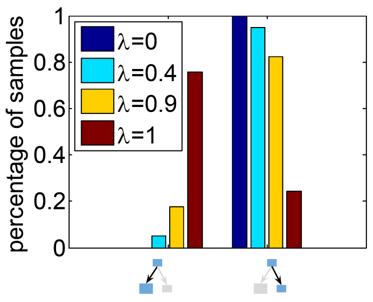
As we can see, the D^{2}NN achieves a trade-off curve close to the upperbound: there are points on the curve that are as fast as the low-capacity node and as accurate as the high-capacity node. Fig. 2.4(left) plots the distribution of examples going through different execution paths. It shows that as λ increases, accuracy becomes more important and more examples go through the high-capacity node. These results suggest that our learning algorithm is effective for networks with a single control node.

With inference efficiency improved, we also observe that for training, a D^{2}NN typically takes 2-4 times more iterations to converge than a DNN, depending on particular model capacities, configurations and trade-offs.

2.5.2 Cascade D^{2}NN

We next experiment with a more sophisticated design that we call a "cascade D^{2}NN" (Fig. 2.3b). It is inspired by the standard cascade design commonly used in computer vision. The intuition is that many negative examples may be rejected early using simple features. The cascade D^{2}NN consists of seven regular nodes (N1-N7) and three control nodes (Q1-Q3). N1-N7 form 4 cascade stages (i.e. 4 conventional DNNs, from small to large) of the cascade: N1+N2, N3+N4, N5+N6, N7. Each control node decides whether to execute the next cascade stage or not.

We evaluate the network on the same LFW-B face classification task using the same evaluation protocol as in the high-low capacity D^{2}NN. Fig. 2.2b) plots the accuracy-cost tradeoff curve for the



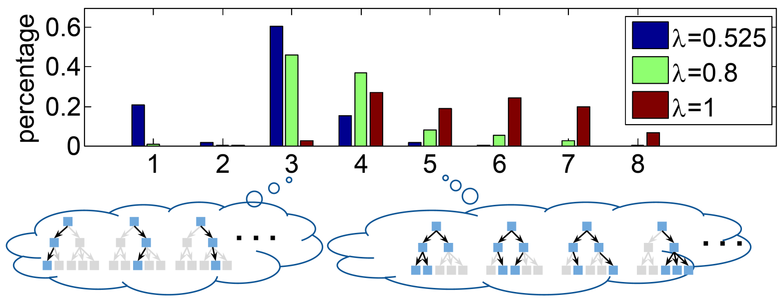


Figure 2.4: Distribution of examples going through different execution paths. Skipped nodes are in grey. The hyperparameter λ controls the trade-off between accuracy and efficiency. A bigger λ values accuracy more. *Left*: for the high-low capacity D^{2}NN. *Right*: for the hierarchical D^{2}NN. The X-axis is the number of nodes activated.

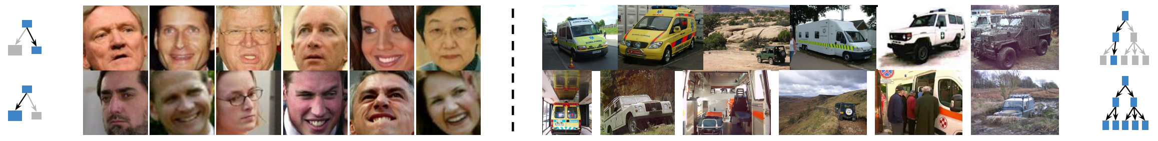


Figure 2.5: Examples with different paths in a high-low D^{2}NN (left) and a hierarchical D^{2}NN (right).

D^{2}NN. Also included are the accuracy-cost curve (“static NNs”) achieved by the four conventional DNNs as baselines, each trained with a cross-entropy loss. We can see that the cascade D^{2}NN can achieve a close to optimal trade-off, reducing computation significantly with negligible loss of accuracy. In addition, we can see that our D^{2}NN curve outperforms the trade-off curve achieved by varying the design and capacity of static conventional networks. This result demonstrates that our algorithm is successful for jointly training multiple control nodes.

For a cascade, wall time of inference is often an important consideration. Thus we also measure the inference wall time (excluding data loading with 5 runs) in this Cascade D^{2}NN. We find that a 82% wall-time cost corresponds to a 53% number-of-multiplication cost; and a 95% corresponds to a 70%. Defining reward directly using wall time can further reduce the gap.

2.5.3 Chain D^{2}NN

Our third design is a “Chain D^{2}NN” (Fig. 2.3c). The network is shaped as a chain, where each link consists of a control node selecting between two (or more) regular nodes. In other words, we perform a sequence of vector-to-vector transforms; for each transform we choose between several

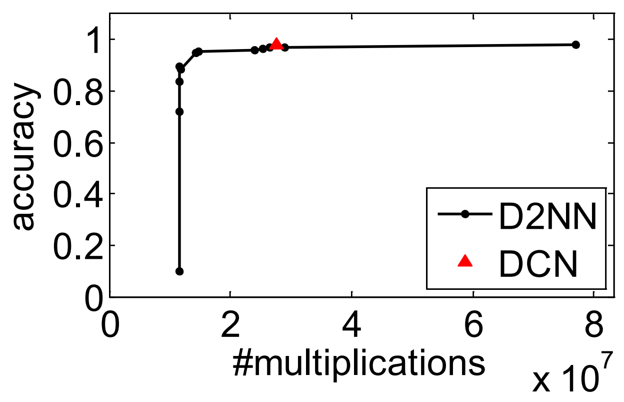


Figure 2.6: Accuracy-cost curve for a chain D^{2}NN on the CMNIST task compared to DCN (Almahairi et al., 2016).

subnetworks. One scenario that we can use this D^{2}NN is that the configuration of a conventional DNN (e.g. number of layers, filter sizes) cannot be fully decided. Also, it can simulate shortcuts between any two layers by using an identity function as one of the transforms. This chain D^{2}NN is qualitatively different from other D^{2}NNs with a tree-shaped *data graph* because it allows two divergent *data paths* to merge again. That is, the number of possible execution paths can be exponential to the number of nodes.

In Fig. 2.3c), the first link is that Q1 chooses between a low-capacity N2 and a high-capacity N3. If one of them is chosen, the other will output a default value zero. The node N4 adds the outputs of N2 and N3 together. Fig. 2.2c) plots the accuracy-cost curve on the LFW-B task. The two baselines are: a conventional DNN with the lowest capacity path (N1-N2-N5-N8-N10), and a conventional DNN with the highest capacity path (N1-N3-N6-N9-N10). The cost is measured as the number of multiplications, normalized by the cost of the high-capacity baseline.

Fig. 2.2c) shows that the chain D^{2}NN achieves a trade-off curve close to optimal and can speed up computation significantly with little accuracy loss. This shows that our learning algorithm is effective for a D^{2}NN whose data graph is a general DAG instead of a tree.

2.5.4 Hierarchical D^{2}NN

In this experiment we design a D^{2}NN for hierarchical multiclass classification. The idea is to first classify images to coarse categories and then to fine categories. This idea has been explored by numerous prior works (Liu et al., 2013; Bengio et al., 2010; Deng et al., 2011), but here we show that the same idea can be implemented via a D^{2}NN trained end to end.

We use ILSVRC-10, a subset of the ILSVRC-65 (Deng et al., 2012). In ILSVRC-10, 10 classes are organized into a 3-layer hierarchy: 2 superclasses, 5 coarse classes and 10 leaf classes. Each class has 500 training images, 50 validation images, and 150 test images. As in Fig. 2.3d), the hierarchy in this D^{2}NN mirrors the semantic hierarchy in ILSVRC-10. An image first goes through the root N1. Then Q1 decides whether to descend the left branch (N2 and its children), and Q2 decides whether to descend the right branch (N3 and its children). The leaf nodes N4-N8 are each responsible for classifying two fine-grained leaf classes. It is important to note that an input image can go down parallel paths in the hierarchy, e.g. descending both the left branch and the right branch, because Q1 and Q2 make separate decisions. This "multi-threading" allows the network to avoid committing to a single path prematurely if an input image is ambiguous.

Fig. 2.2d) plots the accuracy-cost curve of our hierarchical D^{2}NN. The accuracy is measured as the proportion of correctly classified test examples. The cost is measured as the number of multiplications, normalized by the cost of a conventional DNN consisting only of the regular nodes (denoted as NN in the figure). We can see that the hierarchical D^{2}NN can match the accuracy of the full network with about half of the computational cost.

Fig. 2.4(right) plots for the hierarchical D^{2}NN the distribution of examples going through execution sequences with different numbers of nodes activated. Due to the parallelism of D^{2}NN, there can be many different execution sequences. We also see that as λ increases, accuracy is given more weight and more nodes are activated.

2.5.5 Comparison with Dynamic Capacity Networks

In this experiment we empirically compare our approach to closely related prior work. Here we compare D^{2}NNs with Dynamic Capacity Networks (DCN) (Almahairi et al., 2016), for which efficency measurement is the absolute number of multiplications. Given an image, a DCN applies an additional high capacity subnetwork to a set of image patches, selected using a hand-designed saliency based policy. The idea is that more intensive processing is only necessary for certain image regions.

To compare, we evaluate with the same multiclass classification task on the Cluttered MNIST (Mnih et al., 2014), which consists of MNIST digits randomly placed on a background cluttered with fragments of other digits. We train a chain D^{2}NN of length 4 , which implements the same idea of choosing a high-capacity alternative subnetwork for certain inputs. Fig. 2.6 plots the accuracy-cost curve of our D^{2}NN as well as the accuracy-cost point achieved by the DCN in (Almahairi et al., 2016)—an accuracy of 0.9861 and and a cost of 2.77×10^7. The closest point on our curve is an slightly lower accuracy of 0.9698 but slightly better efficiency (a cost of 2.66×10^7). Note that although our accuracy of 0.9698 is lower, it compares favorably to those of other state-of-the-art methods such as DRAW (Gregor et al., 2015): 0.9664 and RAM (Mnih et al., 2014): 0.9189.

2.5.6 Visualization of Examples in Different Paths

In Fig. 2.5 (left), we show face examples in the high-low D^{2}NN for λ=0.4. Examples in low-capacity path are generally easier (e.g. more frontal) than examples in high-capacity path. In Fig. 2.5 (right), we show car examples in the hierarchical D^{2}NN with 1) a single path executed and 2) the full graph executed (for λ=1). They match our intuition that examples with a single path executed should be easier (e.g. less occlusion) to classify than examples with the full graph executed.

2.5.7 CIFAR-10 Results

We train a Cascade D^{2}NN on CIFAR-10 where the corresponded DNN baseline is the ResNet-110. We initialize this D^{2}NN with pre-trained ResNet-110 weights, apply cross-entropy losses on regular nodes, and tune the mixed-loss weight as explained in Sec. 4. We see a 30% reduction of cost with a 2% loss (relative) on accuracy, and a 62% reduction of cost with a 7% loss (relative) on accuracy. The D^{2}NN's ability to improve efficiency relies on the assumption that not all inputs require the same amount of computation. In CIFAR-10, all images are low resolution (32×32), and it is likely that few images are significantly easier to classify than others. As a result, the efficiency improvement is modest compared to other datasets.

2.6 Conclusion

In this chapter, we have introduced Dynamic Deep Neural Networks (D^{2}NN), a new type of feed-forward deep neural networks that allow automated dynamic inference. Extensive experiments have demonstrated that D^{2}NNs are flexible and effective.

2.7 Appendix

2.7.1 Implementation Details

We implement the D^{2}NN framework in Torch. Torch already provides implementations of conventional neural network modules (nodes). So a user can specify the subnetwork architecture inside a control node or a regular node using existing Torch functionalities. Our framework then handles the communication between the user-defined nodes in the forward and backward pass.

To handle parallel paths, default-valued nodes and nodes with multiple data parents, we need to keep track of an example's execution status (which nodes are activated by this example) and output status (which nodes have output for this example). An example's output status is different from its execution status if some nodes are not activated but have default values. For runtime efficiency, we implement the tracking of examples at the mini-batch level. That is, we perform forward and backward passes for a mini-batch of examples as a regular DNN does. Each mini-batch consists of several mini-bags of images.

We describe the implementation of D^{2}NN learning procedure as two steps. First, the preprocessing step: When a user-defined D^{2}NN model is fed into our framework, we first perform a breadth-first search to get the DAG orders of nodes while performing structure error checks, contructing data and control relationships between nodes and calculating the cost (number of multiplications) of each node.

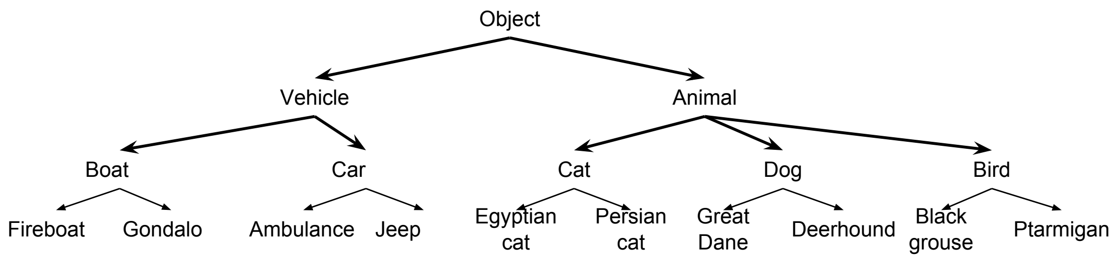


Figure 2.7: The semantic class hierarchy of the ILSVRC-10 dataset.

After the preprocessing, the training step is similar to a regular DNN: a forward pass and a backward pass. All nodes are visited according to a topological ordering in a forward pass and the reverse ordering in a backward pass.

For each function node, the forward pass has three steps: fetch inputs, forward inside the node, and send data or control signals to children nodes. When dealing with multiple data inputs and multiple control signals, the D^2NN will filter examples with more than one null inputs or all negative control signals. When a default value has been set for a node, all examples have to send out data. If the node is not activated for a particular example, the output will take the default value. A backward pass has similar logic: fetch gradients from children, perform the backward pass inside and send out gradients to parents. It is worth noting that when a default value is used in a node, the gradients can be blocked by this node because it is not actually executed.

2.7.2 ILSVRC-10 Semantic Hierarchy

The ILSVRC-10 dataset is a subset of the ILSVRC-65 dataset (Deng et al., 2012). In our ILSVRC-10, there are 10 classes organized into a 3-layer hierarchy: 2 superclasses, 5 coarse classes and 10 leaf classes as in Fig 2.7. Each class has 500 training images, 50 validation images, and 150 test images.

2.7.3 Configurations

High-Low Capacity D^2NN The high-low capacity D^2NN consists of a single control node (Q) and three regular nodes (N1,N2,N3) as illustrated in Fig. 2.3a).

- Node N1: a convolutional layer with a 3×3 filter size, 8 filters and a stride of 2, followed by a 3×3 max-pooling layer with a stride of 2.
- Node N2: a convolutional layer with a 3×3 filter size and 16 filters, followed by a 3×3 max-pooling layer with a stride of 2. The output is reshaped and fed into a fully connected layer with 512 neurons followed by another fully connected layer with the 2-class output.
- Node N3: three 3×3 max-pooling layers, each with a stride of 2, followed by two fully connected layers with 32 neurons and the 2-class output.
- Node Q1: a convolutional layer with a 3×3 filter size and 2 filters, followed by a 3×3 max-pooling layer with a stride of 2. The output is reshaped and fed into a fully connected layer with 128 neurons followed by another fully connected layer with the 2-action output.

Cascade D^{2}NN The cascade D^{2}NN consists of a sequence of four regular nodes (N1 to N7) and three control nodes (Q1-Q3) as in Fig. 2.3b).

- Node N1: a convolutional layer with a 3×3 filter size, 2 filters and a stride of 2, followed by a 3×3 max-pooling layer with a stride of 2.
- Node N2: three 3×3 max-pooling layers with strides of 2. The output is reshaped and fed into a fully connected layer with the 2-class output.
- Node N3: two convolutional layers with both 3×3 filter sizes and 2, 8 filters respectively, each followed by a 3×3 max-pooling layer with a stride of 2.
- Node N4: two 3×3 max-pooling layers with strides of 2. The output is reshaped and fed into a fully connected layer with the 2-class output.
- Node N5: two convolutional layers with both 3×3 filter sizes and 4, 16 filters respectively, each followed by a 3×3 max-pooling layer with a stride of 2.

- Node N6: two 3×3 max-pooling layers with strides of 2. The output is reshaped and fed into a fully connected layer with the 2-class output.
- Node N7: five convolutional layers with all 3×3 filter sizes and 2, 8, 32, 32, 64 filters repectively, each followed by a 3×3 max-pooling layer with a stride of 2 except for the third and fifth layer. The output is reshaped and fed into a fully connected layer with 512 neurons followed by another fully connected layer with the 2-class output.
- Node Q1, Q2, Q3: the input is reshaped and fed into a fully connected layer with the 2-action output.

Chain D^{2}NN The Chain D^{2}NN is shaped as a chain, where each link consists of a control node selecting between two regular nodes. In the experiments of LFW-B dataset, we use a 3-stage Chain D^{2}NN as in Fig. 2.3c).

- Node N1: a convolutional layer with a 3×3 filter size, 2 filters and a stride of 2, followed by a 3×3 max-pooling layer with a stride of 2.
- Node N2: a convolutional layer with a 1×1 filter size and 16 filters.
- Node N3: a convolutional layer with a 3×3 filter size and 16 filters.
- Node N4: a 3×3 max-pooling layer with a stride of 2.
- Node N5: a convolutional layer with a 1×1 filter size and 32 filters.
- Node N6: two convolutional layers with both 3×3 filter sizes and 32, 32 filters repectively.
- Node N7: a 3×3 max-pooling layer with a stride of 2.
- Node N8: a convolutional layer with a 1×1 filter size and 32 filters followed by a 3×3 max-pooling layer with a stride of 2. The output is reshaped and fed into a fully connected layer with 256 neurons.

- Node N9: a convolutional layer with a 3×3 filter size and 64 filters. The output is reshaped and fed into a fully connected layer with 256 neurons.
- Node N10: a fully connected layer with the 2-class output.
- Node Q1: a convolutional layer with a 3×3 filter size and 8 filters with a 3×3 max-pooling layer with a stride of 2 before and a 3×3 max-pooling layer with a stride of 2 after. The output is reshaped and fed into two fully connected layers with 64 neurons and the 2-action output respectively.
- Node Q2: a 3×3 max-pooling layer with a stride of 2 followed by a convolutional layer with a 3×3 filter size and 4 filters. The output is reshaped and fed into two fully connected layers with 64 neurons and the 2-action output respectively.
- Node Q3: a convolutional layer with a 3×3 filter size and 2 filters. The output is reshaped and fed into two fully connected layers with 64 neurons and the 2-action output respectively.

Hierarchical D^{2}NN Fig. 2.3d) illustrates the design of our hierarchical D^{2}NN.

- Node N1: a convolutional layer with a 11×11 filter size, 64 filters, a stride of 4 and a 2×2 padding, followed by a 3×3 max-pooling layer with a stride of 2.
- Node N2 and N3: a convolutional layer with a 5×5 filter size, 96 filters and a 2×2 padding.
- Node N4 N8: a 3×3 max-pooling layer with a stride of 2 followed by three convolutional layers with 3×3 filter sizes and 160, 128, 128 filters respectively. The output is fed into a 3×3 max-pooling layer with a stride of 2 and three fully connected layers with 2048 neurons, 2048 neurons and the 2 fine-class output respectively.
- Node Q1 and Q2: two convolutional layers with 5×5, 3×3 filter sizes and 16, 32 filters respectively (the former has a 2×2 padding), each followed by a 3×3 max-pooling layer with a stride of 2. The output is reshaped and fed into three fully connected layers with 1024 neurons, 1024 neurons and the 2-action output respectively.

- Node Q3 Q7: two convolutional layers with 5×5, 3×3 filter sizes and 16, 32 filters respectively (the former has a 2×2 padding), each followed by a 3×3 max-pooling layer with a stride of 2. The output is reshaped and fed into three fully connected layers with 1024 neurons, 1024 neurons and the 2-action output respectively.

Comparison with Dynamic Capacity Networks We train a chain D^{2}NN of length 4 similar to Fig. 2.3c).

- Node N1: a convolutional layer with a 3×3 filter size and 24 filters.
- Node N3: a convolutional layer with a 3×3 filter size and 24 filters.
- Node N4: a 2×2 max-pooling layer with a stride of 2.
- Node N6: a convolutional layer with a 3×3 filter size and 24 filters.
- Node N7: an identity layer which directly uses inputs as outputs.
- Node N9: a convolutional layer with a 3×3 filter size and 24 filters.
- Node N10: a 2×2 max-pooling layer with a stride of 2.
- Node N12: a convolutional layer with a 3×3 filter size and 24 filters.
- Node N2, N5, N8, N11: an identity layer.
- Node N13: a convolutional layer with a 4×4 filter size, 96 filters, a stride of 2 and no padding, followed by a 11×11 max-pooling layer. The output is reshaped and fed into a fully connected layer with the 10-class output.
- Node Q1: a convolutional layer with a 3×3 filter size and 8 filters with two 2×2 max-pooling layers with strides of 2 before and one 2×2 max-pooling layer with a stride of 2 after. The output is reshaped and fed into two fully connected layers with 256 neurons and the 2-action output respectively.

- Node Q2: a convolutional layer with a 3×3 filter size and 8 filters with a 2×2 max-pooling layer with a stride of 2 before and a 2×2 max-pooling layer with a stride of 2 after. The output is reshaped and fed into two fully connected layers with 256 neurons and the 2-action output respectively.
- Node Q3: a convolutional layer with a 3×3 filter size and 8 filters with a 2×2 max-pooling layer with a stride of 2 before and a 2×2 max-pooling layer with a stride of 2 after. The output is reshaped and fed into two fully connected layers with 256 neurons and the 2-action output respectively.
- Node Q4: a convolutional layer with a 3×3 filter size and 8 filters, followed by a 2×2 max-pooling layer with a stride of 2. The output is reshaped and fed into two fully connected layers with 256 neurons and the 2-action output respectively.

For all 5 D^2NNs, all convolutional layers use 1×1 padding and each is followed by a ReLU layer unless specified individually. Each fully connected layer except the output layers is followed by a ReLU layer.

CHAPTER 3

Unified Loss Design [1]

3.1 Introduction

In this chapter, we make efforts towards designing loss functions with reduced manual efforts. Many supervised learning tasks involve designing and optimizing a loss function that is often different from the actual performance metric for evaluating models. For example, cross-entropy is a popular loss function for training a multi-class classifier, but when it comes to comparing models on a test set, classification error is used instead.

Why not optimize the performance metric directly? Because many metrics or output decoders are non-differentiable and cannot be optimized with gradient-based methods such as stochastic gradient descent. Non-differentiability occurs when the output of the task is discrete (e.g. class labels), or when the output is continuous but the performance metric has discrete operations (e.g. percentage of real-valued predictions within a certain range of the ground truth).

To address this issue, designing a differentiable loss that serves as a surrogate to the original metric is standard practice. For standard tasks with simple output and metrics, there exist well-studied surrogate losses. For example, cross-entropy or hinge loss for classification, both of which have proven to work well in practice.

However, designing surrogate losses can sometimes incur substantial manual effort, including a large amount of trial and error and hyper-parameter tuning. For example, a standard evaluation of single-person human pose estimation—predicting the 2D locations of a set of body joints for

[1]This chapter is based on a joint work with Mingzhe Wang and Jia Deng

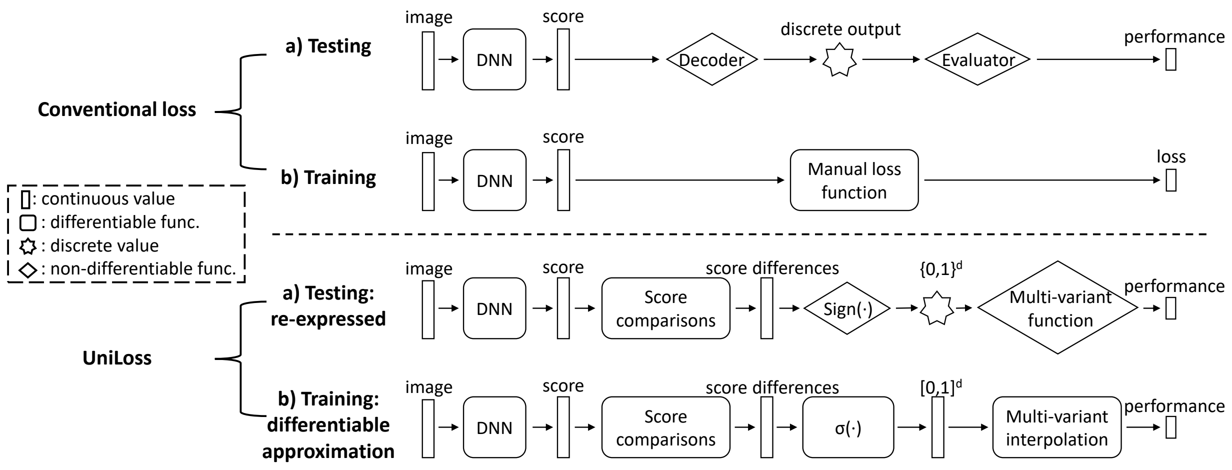


Figure 3.1: Computation graphs for conventional losses and UniLoss. Top: (a) testing for conventional losses. The decoder and evaluator are usually non-differentiable. (b) training for conventional losses. To avoid the non-differentiability, conventional methods optimize a manually-designed differentiable loss function instead during training. Bottom: (a) refactorized testing in UniLoss. We refactorize the testing so that the non-differentiability exists only in Sign(·) and the multi-variant function. (b) training in UniLoss with the differentiable approximation of refactorized testing. $\sigma(\cdot)$ is the sigmoid function. We approximate the non-differentiable components in the refactorized testing pipeline with interpolation methods.

a single person in an image—involves computing the percentage of predicted body joints that are within a given radius of the ground truth. This performance metric is non-differentiable. Existing work instead trains a deep network to predict a heatmap for each type of body joints, minimizing the difference between the predicted heatmap and a “ground truth” heatmap consisting of a Gaussian bump at the ground truth location (Tompson et al., 2014; Newell et al., 2016). The decision for what error function to use for comparing heatmaps and how to design the “ground truth” heatmaps are manually tuned to optimize performance.

This manual effort in conventional losses is tedious but necessary, because a poorly designed loss can be misaligned with the final performance metric and lead to ineffective training. As we show in the experiment section, without carefully-tuned loss hyper-parameters, conventional manual losses can work poorly.

In this chapter, we seek to reduce the efforts of manual design of surrogate losses by introducing a unified surrogate loss framework applicable to a wide range of tasks. We provide a unified framework to mechanically generate a surrogate loss given a performance metric in the context of deep learning. This means that we only need to specify the performance metric (e.g. classification error) and the inference algorithm—the network architecture, a "decoder" that converts the network output (e.g. continuous scores) to the final output (e.g. discrete class labels), and an "evaluator" that converts the labels to final metric—and the rest is taken care of as part of the training algorithm.

We introduce UniLoss (Fig. 3.1), a unified framework to generate surrogate losses for training deep networks with gradient descent. We maintain the basic algorithmic structure of mini-batch gradient descent: for each mini-batch, we perform inference on all examples, compute a loss using the results and the ground truths, and generate gradients using the loss to update the network parameters. Our novelty is that we generate all the surrogate losses in a unified framework for various tasks instead of manually design it for each task.

The key insight behind UniLoss is that for many tasks and performance metrics, evaluating a deep network on a set of training examples—pass the examples through the network, the output decoder, and the evaluator to the performance metric—can be refactorized into a sequence of four transformations: the training examples are first transformed to a set of real scores, then to some real numbers representing comparisons (through subtractions) of certain pairs of the real-valued scores, then to a set of binary variables, and finally to a single real number. Note that the four transforms do not necessarily correspond to running the network inference, the decoder, and the evaluator.

Take multi-class classification as an example, the training examples are first transformed to a set of scores (one per class per example), and then to pairwise comparisons (subtractions) between the scores for each example (i.e. the argmax operation), and then to a set of binary values, and finally to a classification accuracy.

The final performance metric is non-differentiable with respect to network weights because the decoder and the evaluator are non-differentiable. But this refactorization allows us to generate a differentiable approximation of each non-differentiable transformation through interpolation.

Specifically, the transformation from comparisons to binary variables is nondifferentiable, we can approximate it by using the sigmoid function to interpolate the sign function. And the transformation from binary variables to final metric may be nondifferentiable, we can approximate it by multivariate interpolation

The proposed UniLoss framework is general and can be applied to various tasks and performance metrics. More specifically, when the nondifferentiability only occurs as step functions (i.e. comparisons), the metric can always be refactorized as our formulation. Many tasks and metrics satisfy this requirement. For example, classification scenarios such as accuracy in image classification, precision and recall in object detection; ranking scenarios such as average precision in binary classification, area under curve in image retrieval; pixel-wise prediction scenarios such as mean IOU in segmentation, PCKh in pose estimation. However, UniLoss cannot be applied to evaluation metrics that have learnable parameters. Because our UniLoss does not involve learning parameters in the loss function. Learnable parameters in the evaluation metric (Lita et al., 2005) is however rare in practice.

To validate its effectiveness, we perform experiments on three representative tasks from three different scenarios. We show that UniLoss performs well on a classic classification setting, multi-class classification, compared with the well-established conventional losses. We also demonstrate UniLoss's ability in a ranking scenario that evolves ranking multiple images in an evaluation set: average precision (area under the precision-recall curve) in unbalanced binary classification. In addition, we experiment with pose estimation where the output is structured as pixel-wise predictions.

Our main contributions in this work are:

- We present a new perspective of finding surrogate losses: evaluation can be refactorized as a sequence of four transformations, where each nondifferentiable transformation can be tackled individually.

- We propose a new method: a unified framework to generate losses for various tasks without tedious task-specific manual design.

- We validate the new perspective and the new method on three tasks and four datasets, achieving comparable performance with conventional losses.

3.2 Related Work

3.2.1 Direct Loss Minimization

The line of direct loss minimization works is related to UniLoss because we share a similar idea of finding a good approximation of the performance metric. There have been many efforts to directly minimize specific classes of tasks and metrics.

For example, Taylor et al. (2008) optimized ranking metrics such as Normalized Discounted Cumulative Gain by smoothing them with an assumed probabilistic distribution of documents. Henderson and Ferrari (2016) directly optimized mean average precision in object detection by computing "pseudo partial derivatives" for various continuous variables. Nguyen and Sanner (2013) explored to optimize the 0-1 loss in binary classification by search-based methods including branch and bound search, combinatorial search, and also coordinate descent on the relaxations of 0-1 losses. Liu et al. (2016) proposed to improve the conventional cross-entropy loss by multiplying a preset constant with the angle in the inner product of the `softmax` function to encourage large margins between classes. Fu et al. (2018) proposed an end-to-end optimization approach for speech enhancement by directly optimizing short-time objective intelligibility (STOI) which is a differentiable performance metric.

In addition to the large algorithmic differences, these works also differ from ours in that they are tightly coupled with specific tasks and applications.

Hazan et al. (2010) and Song et al. (2016) proved that under mild conditions, optimizing a max-margin structured-output loss is asymptotically equivalent to directly optimizing the performance metrics. Specifically, assume a model in the form of a differentiable scoring function $S(x, y; w) : \mathcal{X} \times \mathcal{Y} \rightarrow \mathbf{R}$ that evaluates the compatibility of output y with input x. During inference, they predict the y_w with the best score:

$$y_w = \operatorname*{argmax}_y S(x, y; w). \tag{3.1}$$

During training, in addition to this regular inference, they also perform the *loss-augmented inference* (Tsochantaridis et al., 2005; Hazan et al., 2010):

$$y^\dagger = \operatorname*{argmax}_y S(x, y; w) + \epsilon\xi(y_w, y), \tag{3.2}$$

where ξ is the final performance metric (in terms of error), and ϵ is a small time-varying weight. Song et al. (2016) generalized this result from linear scoring functions to arbitrary scoring functions, and developed an efficient loss-augmented inference algorithm to directly optimize average precision in ranking tasks.

While above max-margin losses can ideally work with many different performance metrics ξ, its main limitation in practical use is that it can be highly nontrivial to design an efficient algorithm the loss-augmented inference, as it often requires some clever factorization of the performance metric ξ over the components of the structured output y. In fact, for many metrics the loss-augmented inference is NP-hard and one must resort to designing approximate algorithms, which further increases the difficulty of practice use.

In contrast, our method does not demand the same level of human ingenuity. The main human effort involves refactoring the inference code and evaluation code to a particular format, which may be further eliminated by automatic code analysis. There is no need to design a new inference algorithm over discrete outputs and analyze its efficiency. The difficulty of designing loss-augmented inference algorithms for each individual task makes it impractical to compare fairly with max-margin methods on diverse tasks, because it is unclear how to design the inference algorithms.

Recently, some prior works propose to directly optimize the performance metric by learning a parametric surrogate loss (Huang et al., 2019; Wu et al., 2018; Santos et al., 2017; Grabocka et al., 2019). During training, the model is updated to minimize the current surrogate loss while the parametric surrogate loss is also updated to align with the performance metric.

Compared to these methods, UniLoss does not involve any learnable parameters in the loss. As a result, UniLoss can be applied universally across different settings without any training, and the parametric surrogate loss has to be trained separately for different tasks and datasets.

Reinforcement learning algorithms have been used to optimize performance metrics for structured output problems, especially those that can be formulated as taking a sequence of actions (Ranzato et al., 2016; Liu et al., 2017b; Caicedo and Lazebnik, 2015; Yeung et al., 2016; Zhou et al., 2018). For example, Liu et al. (2017b) use policy gradients (Sutton et al., 2000) to optimize metrics for image captioning.

We differ from these approaches in two key aspects. First, we do not need to formulate a task as a sequential decision problem, which is natural for certain tasks such as text generation, but unnatural for others such as human pose estimation. Second, reinforcement learning approaches treat performance metrics as black boxes, whereas we assume access to the code of the performance metrics, which is a valid assumption in most cases. This access allows us to reason about the code and generate better gradients.

3.2.2 Surrogate Losses

There has been a large body of literature studying surrogate losses, for tasks including multi-class classification (Bartlett et al., 2006; Zhang, 2004; Tewari and Bartlett, 2007; Crammer and Singer, 2001; Allwein et al., 2000; Ramaswamy et al., 2013, 2014), binary classification (Bartlett et al., 2006; Zhang, 2004; Ramaswamy et al., 2013, 2014) and pose estimation (Tompson et al., 2014).

Compared to them, UniLoss removes the tedious manual effort to design task-specific losses. We demonstrate that UniLoss, as a general loss framework, can be applied to all these tasks and achieve comparable performance.

3.3 UniLoss

3.3.1 Overview

UniLoss provides a unified way to generate a surrogate loss for training deep networks with mini-batch gradient descent without task-specific design. In our general framework, we first reformulate the evaluation process and then approximate the non-differentiable functions using interpolation.

3.3.1.1 Original Formulation

Formally, let $\mathbf{x} = (x_1, x_2, \ldots, x_n) \in \mathcal{X}^n$ be a set of n training examples and $\mathbf{y} = (y_1, y_2, \ldots, y_n) \in \mathcal{Y}^n$ be the ground truth. Let $\phi(\cdot; w) : \mathcal{X} \to \mathbf{R}^d$ be a deep network parameterized by weights w that outputs a d-dimensional vector; let $\delta : \mathbf{R}^d \to \mathcal{Y}$ be a decoder that decodes the network output to a possibly discrete final output; let $\xi : \mathcal{Y}^n \times \mathcal{Y}^n \to \mathbf{R}$ be an evaluator. ϕ and δ are applied in a mini-batch fashion on $\mathbf{x} = (x_1, x_2, \ldots, x_n)$; the performance e of the deep network is then

$$e = \xi(\delta(\phi(\mathbf{x}; w)), \mathbf{y}). \tag{3.3}$$

3.3.1.2 Refactorized Formulation

Our approach seeks to find a surrogate loss to minimize e, with the novel observation that in many cases e can be refactorized as

$$e = g(h(f(\phi(\mathbf{x}; w), \mathbf{y}))), \tag{3.4}$$

where $\phi(\cdot; w)$ is the same as in Eqn. 3.3, representing a deep neural network, $f : \mathbf{R}^{n\times d} \times \mathcal{Y}^n \to \mathbf{R}^l$ is differentiable and maps outputted real numbers and the ground truth to l comparisons each representing the difference between certain pair of real numbers, $h : \mathbf{R}^l \to \{0, 1\}^l$ maps the l score differences to l binary variables, and $g : \{0, 1\}^l \to \mathbf{R}$ computes the performance metric from binary variables. Note that h has a restricted form that always maps continuous values to binary values through sign function, whereas g can be arbitrary computation that maps binary values to a real number.

We give intermediate outputs some notations:

- Training examples $\mathbf{x}, \mathbf{y}$ are transformed to scores $\mathbf{s} = (s_1, s_2, \dots, s_{nd})$, where $\mathbf{s} = \phi(\mathbf{x}; w)$.
- $\mathbf{s}$ is converted to comparisons (differences of two scores) $\mathbf{c} = (c_1, c_2, \dots, c_l)$, where $\mathbf{c} = f(\mathbf{s}, \mathbf{y})$.
- $\mathbf{c}$ is converted to binary variables $\mathbf{b} = (b_1, b_2, \dots, b_l)$ representing the binary outcome of the comparisons, where $\mathbf{b} = h(\mathbf{c})$.
- The binary variables are transformed to a single real number by $e = g(\mathbf{b})$.

This new refactorization of a performance metric allows us to decompose the metric e with g, h, f and ϕ, where ϕ and f are differentiable functions but h and g are often non-differentiable. The non-differentiability of h and g causes e to be non-differentiable with respect to network weights w.

3.3.1.3 Differentiable Approximation

Our UniLoss generates differentiable approximations of the non-differentiable h and g through interpolation, thus making the metric e optimizable with gradient descent. Formally, UniLoss gives a differentiable approximation $\tilde{e}$

$$\tilde{e} = \tilde{g}(\tilde{h}(f(\phi(\mathbf{x}; w), \mathbf{y}))), \tag{3.5}$$

where f and ϕ are the same as in Eqn. 3.4, and $\tilde{h}$ and $\tilde{g}$ are the differentiable approximation of h and g.

We explain a concrete example of multi-class classification and introduce the refactorization and interpolation in detail based on this example in the following sections.

3.3.2 Example: Multi-class Classification

We take multi-class classification as an example to show how refactorization works. First, we give formal definitions of multi-class classification and the performance metric: prediction accuracy.

Inputs are a mini-batch of images $\mathbf{x} = (x_1, x_2, \dots, x_n)$ and their corresponding ground truth labels are $\mathbf{y} = (y_1, y_2, \dots, y_n)$ where n is the batch size. $y_i \in \{1, 2, \dots, p\}$ for $i = 1, 2, \dots, n$ and p is the number of classes, which happens to be the same value as d in Sec. 3.3.1. A network $\phi(\cdot; w)$ outputs a score matrix $\mathbf{s} = [s_{i,j}]_{n\times p}$ and $s_{i,j}$ represents the score for the i-th image belongs to the class j.

The decoder $\delta(\mathbf{s})$ decodes s into the discrete outputs $\tilde{\mathbf{y}} = (\tilde{y}_1, \tilde{y}_2, \dots, \tilde{y}_n)$ by

$$\tilde{y}_i = \underset{1\le j\le p}{\operatorname{argmax}}\, s_{i,j}, \tag{3.6}$$

and $\tilde{y}_i$ represents the predicted label of the i-th image for $i = 1, 2, \dots, n$.

The evaluator $\xi(\tilde{\mathbf{y}}, \mathbf{y})$ evaluates the accuracy e from $\tilde{\mathbf{y}}$ and $\mathbf{y}$ by

$$e = \frac{1}{n}\sum_{i=1}^{n}[y_i = \tilde{y}_i], \tag{3.7}$$

where $[\cdot]$ is the Iverson bracket.

Considering above together, the predicted label for an image is correct if and only if the score of its ground truth class is higher than the score of every other class:

$$[y_i = \tilde{y}_i] = \bigwedge_{\substack{1\le j\le p\\ j\ne y_i}} [s_{i,y_i} - s_{i,j} > 0], \quad \text{for all } 1 \le i \le n, \tag{3.8}$$

where $\wedge$ is logical and.

We thus refactorize the decoding and evaluation process as a sequence of $f(\cdot)$ that transforms s to comparisons—$s_{i,y_i} - s_{i,j}$ for all $1 \le i \le n, 1 \le j \le p,$ and $j \ne y_i$ ($n \times (p - 1)$ comparisons in total), $h(\cdot)$ that transforms comparisons to binary values using $[\cdot > 0]$, and $g(\cdot)$ that transforms binary values to e using logical and.

Next, we introduce how to refactorize the above procedure into our formulation and approximate g and h.

3.3.3 Refactorization

In Eqn. 3.4, after we transform the training images into scores $\mathbf{s} = (s_1, s_2, \ldots, s_{nd})$, we get the score comparisons (differences of pairs of scores) $\mathbf{c} = (c_1, c_2, \ldots, c_l)$ using $\mathbf{c} = f(\mathbf{s}, \mathbf{y})$. Each comparison is $c_i = s_{k_i^1} - s_{k_i^2}, 1 \leq i \leq l, 1 \leq k_i^1, k_i^2 \leq nd$. The function h then transforms the comparisons to binary values by $\mathbf{b} = h(\mathbf{c})$. The h is the sign function, i.e. $b_i = [c_i > 0], 1 \leq i \leq l$. The function g then computes e by $e = g(\mathbf{b})$, where g can be arbitrary computation that converts binary values to a real number. In practice, g can be complex and vary significantly across tasks and metrics.

Given any performance metrics involving discrete operations in function ξ and δ in Eqn. 3.3 (otherwise the metric e is differentiable and trivial to be handled), the computation of function $\xi(\delta(\cdot))$ can be refactorized as a sequence of continuous operations (which is optional), discrete operations that make some differentiable real numbers non-differentiable, and any following operations. The discrete operations always occur when there are step functions, which can be expressed as comparing two numbers, to the best of our knowledge.

This refactorization is usually straightforward to obtain from the specification of the decoding and evaluating procedures. The only manual effort is in identifying the discrete comparisons (binary variables). Then we simply write the discrete comparisons as function f and h, and represent its following operations as function g.

In later sections we will show how to identify the binary variables for three commonly-used metrics in three scenarios, which can be easily extended to other performance metrics. On the other hand, this process is largely a mechanical exercise, as it is equivalent to rewriting some existing code in an alternative rigid format.

3.3.4 Interpolation

The two usually non-differentiable functions h and g are approximated by interpolation methods individually.

3.3.4.1 Scores to Binaries: h.

In $\mathbf{b} = h(\mathbf{c})$, each element $b_i = [c_i > 0]$. We approximate the step function $[\cdot]$ using the `sigmoid` function. That is, $\tilde{\mathbf{b}} = \tilde{h}(\mathbf{c}) = (\tilde{b}_1, \tilde{b}_2, \dots, \tilde{b}_l)$, and each element

$$\tilde{b}_i = \texttt{sigmoid}(c_i), \tag{3.9}$$

where $1 \le i \le l$. We now have $\tilde{h}$ as the differentiable approximation of h.

3.3.4.2 Binaries to Performance: g.

We approximate $g(\cdot)$ in $e = g(\mathbf{b})$ by multivariate interpolation over the input $\mathbf{b} \in \{0, 1\}^l$. More specifically, We first sample a set of configurations as "anchors" $\mathbf{a} = (\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_t)$, where $\mathbf{a}_i$ is a configuration of $\mathbf{b}$, and compute the output values $g(\mathbf{a}_1), g(\mathbf{a}_2), \dots, g(\mathbf{a}_t)$, where $g(\mathbf{a}_i)$ is the actual performance metric value and t is the number of anchors sampled.

We then get an interpolated function over the anchors as $\tilde{g}(\cdot; \mathbf{a})$. We finally get $\tilde{e} = \tilde{g}(\tilde{\mathbf{b}}; \mathbf{a})$, where $\tilde{\mathbf{b}}$ is computed from $\tilde{h}$, f and ϕ.

By choosing a differentiable interpolation method, the $\tilde{g}$ function becomes trainable using gradient-based methods. We use a common yet effective interpolation method: inverse distance weighting (IDW) (Shepard, 1968):

$$\tilde{g}(\mathbf{u}; \mathbf{a}) = \begin{cases} \dfrac{\sum_{i=1}^{t} \frac{1}{d(\mathbf{u}, \mathbf{a}_i)} g(\mathbf{a}_i)}{\sum_{i=1}^{t} \frac{1}{d(\mathbf{u}, \mathbf{a}_i)}}, & d(\mathbf{u}, \mathbf{a}_i) \neq 0 \text{ for } 1 \le i \le t; \\ g(\mathbf{a}_i), & d(\mathbf{u}, \mathbf{a}_i) = 0 \text{ for some } i. \end{cases} \tag{3.10}$$

where $\mathbf{u}$ represents the input to $\tilde{g}$ and $d(\mathbf{u}, \mathbf{a}_i)$ is the Euclidean distance between $\mathbf{u}$ and $\mathbf{a}_i$.

We selecting a subset of anchors based on the current training examples. We use a mix of three types of anchors—good anchors with high performance values globally, bad anchors with low performance values globally, and nearby anchors that are close to the current configuration, which is computed from the current training examples and network weights. By using both the global information from the good and bad anchors and the local information from the nearby anchors, we are able to get an informative interpolation surface.

We adopt a straightforward anchor sampling strategy for all tasks and metrics: we obtain good anchors by flipping some bits from the best anchor, which is the ground truth. The bad anchors are generated by randomly sampling binary values. The nearby anchors are obtained by flipping some bits from the current configuration.

Under our unified framework, there are other sophisticated choices for the interpolation methods and the sampling strategy for anchors. We adopt the current way for simplicity and an easy generalization across different tasks and metrics.

3.4 Experimental Results

To use our general framework UniLoss on each task, we refactorize the evaluation process of the task into the format in Eqn. 3.4, and then approximate the non-differentiable functions h and g using the interpolation method in Sec. 3.

We validate the effectiveness of the UniLoss framework in three representative tasks in different scenarios: a ranking-related task using a set-based metric—average precision, a pixel-wise prediction task, and a common classification task. For each task, we demonstrate how to formulate the evaluation process to our refactorization and compare our UniLoss with interpolation to the conventional task-specific loss. More experimental details can be found in the supplementary material.

3.4.1 Tasks and Metrics

3.4.1.1 Average Precision for Unbalanced Binary Classification

Binary classification is to classify an example from two classes—positives and negatives. Potential applications include face classification and image retrieval. It has unbalanced number of positives and negatives in most cases, which results in that a typical classification metric such as accuracy as in regular classification cannot demonstrate how good is a model properly. For example, when the positives to negatives is 1:9, predicting all examples as negatives gets 90% accuracy.

On this unbalanced binary classification, other metrics such as precision, recall and average precision (AP), i.e. area under the precision-recall curve, are more descriptive metrics. We use AP as our target metric in this task.

It is notable that AP is fundamentally different from accuracy because it is a set-based metric. It can only be evaluated on a set of images, and involves not only the correctness of each image but also the ranking of multiple images.

This task and metric is chosen to demonstrate that UniLoss can effectively optimize for a set-based performance metric that requires ranking of the images.

3.4.1.2 PCKh for Single-Person Pose Estimation

Single-person pose estimation predicts the localization of human joints. More specifically, given an image, it predicts the location of the joints. It is usually formulated as a pixel-wise prediction problem, where the neural network outputs a score for each pixel indicating how likely is the location can be the joint. We choose this task to validate that UniLoss can also handle pixel-wise prediction tasks.

Following prior work, we use PCKh (Percentage of Correct Keypoints wrt to head size) as the performance metric. It computes the percentage of the predicted joints that are within a given radius r of the ground truth. The radius is half of the head segment length.

This task and metric is chosen to validate the effectiveness of UniLoss in optimizing for a pixel-wise prediction problem.

3.4.1.3 Accuracy for Multi-class Classification

Multi-class classification is a common task that has well-established conventional loss — cross-entropy loss. We use this task to show that UniLoss performs comparably in such a common setting.

We use accuracy (the percentage of correctly classified images) as our metric following the common practice.

This task and metric is chosen to demonstrate that for a most common classification setting, UniLoss still performs similarly effective as the well-established conventional loss.

3.4.2 Average Precision for Unbalanced Binary Classification

3.4.2.1 Dataset and Baseline

We augment the handwritten digit dataset MNIST to be a binary classification task, predicting zeros or non-zeros. Given an image containing a single number from 0 to 9, we classify it into the zero (positive) class or the non-zero (negative) class. The positive-negative ratio of 1:9. We create a validation split by reserving 6k images from the original training set.

We use a 3-layer fully-connected neural network with 500 and 300 neurons in each hidden layer respectively. Our baseline model is trained with a 2-class cross-entropy loss. We train both baseline and UniLoss with a fixed learning rate of 0.01 for 30 epochs. We sample 16 anchors for each anchor type in our anchor interpolation for all of our experiments except in ablation studies.

3.4.2.2 Formulation and Refactorization

The evaluation process is essentially ranking images using comparisons between each pair of scores and compute the area under curve based on the ranking. Given that the output of a mini-batch of n images is s $= (s_1, s_2, \ldots, s_n)$, where s_i represents the predicted score of i-th image to be positive. The binary variables are $\mathbf{b} = \{b_{i,j} = [c_{i,j} > 0] = [s_i - s_j > 0]\}$, where i belongs to positives and j belongs to negatives.

3.4.2.3 Results

UniLoss achieves an AP of 0.9988, similarly as the baseline cross-entropy loss (0.9989). This demonstrates that UniLoss can effectively optimize for a performance metric (AP) that is complicated to compute and involves a batch of images.

3.4.3 PCKh for Single-Person Pose Estimation

3.4.3.1 Dataset and Baseline

We use MPII (Andriluka et al., 2014) which has around 22K images for training and 3K images for testing. For simplicity, we perform experiments on the joints of head only, but our method could be applied to an arbitrary number of human joints without any modification.

We adopt the Stacked Hourglass (Newell et al., 2016) as our model. The baseline loss is the Mean Squared Error (MSE) between the predicted heatmaps and the manually-designed "ground truth" heatmaps. We train a single-stack hourglass network for both UniLoss and MSE using RMSProp (Hinton et al., 2012) with an initial learning rate 2.5e-4 for 30 epochs and then drop it by 4 for every 10 epochs until 50 epochs.

3.4.3.2 Formulation and Refactorization

Assume the network generates a mini-batch of heatmaps $\mathbf{s} = (\mathbf{s}^1, \mathbf{s}^2, \ldots, \mathbf{s}^n) \in \mathbf{R}^{n \times m}$, where n is the batch size, m is the number of pixels in each image. The pixel with the highest score in each heatmap is predicted as a key point during evaluation. We note the pixels within the radius r around the ground truth as positive pixels, and other pixels as negative and each heatmap $\mathbf{s}^k$ can be flatted as $(s^k_{pos,1}, s^k_{pos,2}, \ldots, s^k_{pos,m_k}, s^k_{neg,1}, \ldots, s^k_{neg,m-m_k})$, where m_k is the number of positive pixels in the k-th heatmap and $s^k_{pos,j}$ $(s^k_{neg,j})$ is the score of the j-th positive (negative) pixel in the k-th heatmap.

Table 3.1: PCKh of Stacked Hourglass with MSE and UniLoss on the MPII validation.

Loss	MSE	$\sigma = 0.1$	$\sigma = 0.5$	$\sigma = 0.7$	$\sigma = 1$	$\sigma = 3$	$\sigma = 5$	$\sigma = 10$	UniLoss
PCKh		91.31	95.13	93.06	95.71	95.74	94.99	92.25	**95.77**

PCKh requires to find out if a positive pixel has the highest score among others. Therefore, we need to compare each pair of positive and negative pixels and this leads to the binary variables $\mathbf{b} = (b_{k,i,j})$ for $1 \leq k \leq n$, $1 \leq i \leq m_k$, $1 \leq j \leq m - m_k$, where $b_{k,i,j} = [s^k_{pos,i} - s^k_{neg,j} > 0]$, i.e. the comparison between the i-th positive pixel and the j-th negative pixel in the k-th heatmap.

3.4.3.3 Results

It is notable that the manual design of the target heatmaps is a part of the MSE loss function for pose estimation. It heavily relies on the careful design of the ground truth heatmaps. If we intuitively set the pixels at the exact joints to be 1 and the rest of pixels as 0 in the heatmaps, the training diverges.

Luckily, Tompson et al. (2014) proposed to design target heatmaps as a 2D Gaussian bump centered on the ground truth joints, whose shape is controlled by its variance σ and the bump size. The success of the MSE loss function relies on the choices of σ and the bump size. UniLoss, on the other hand, requires no such design.

As shown in Table 3.1, our UniLoss achieves a 95.77 PCKh which is comparable as the 95.74 PCKh for MSE with the best σ. This validates the effectiveness of UniLoss in optimizing for a pixel-wise prediction problem.

We further observe that the baseline is sensitive to the shape of 2D Gaussian, as in Table 3.1. Smaller σ makes target heatmaps concentrated on ground truth joints and makes the optimization to be unstable. Larger σ generates vague training targets and decreases the performance. This demonstrates that conventional losses require dedicated manual design while UniLoss can be applied directly.

Table 3.2: Accuracy of ResNet-20 with CE loss and UniLoss on the CIFAR-10 and CIFAR-100 test set.

Loss	CIFAR-10	CIFAR-100
CE (cross-entropy) Loss	91.49	65.90
UniLoss	91.64	65.92

3.4.4 Accuracy for Multi-class Classification

3.4.4.1 Dataset and Baseline

We use CIFAR-10 and CIFAR-100 (Krizhevsky and Hinton, 2009), with 32×32 images and 10/100 classes. They each have 50k training images and 10k test images. Following prior work (He et al., 2016), we split the training set into a 45k-5k train-validation split.

We use the ResNet-20 architecture (He et al., 2016). Our baselines are trained with cross-entropy (CE) loss. All experiments are trained following the same augmentation and pre-processing techniques as in prior work (He et al., 2016). We use an initial learning rate of 0.1, divided by 10 and 100 at the 140th epoch and the 160th epoch, with a total of 200 epochs trained for both baseline and UniLoss on CIFAR-10. On CIFAR-100, we train baseline with the same training schedule and UniLoss with 5x training schedule because we only train 20% binary variables at each step. For a fair comparison, we also train baseline with the 5x training schedule but observes no improvement.

3.4.4.2 Formulation and Refactorization

As shown in Sec. 3.3.2, given the output of a mini-batch of n images s $= (s_{1,1}, s_{1,2}.., s_{n,p})$, we compare the score of the ground truth class and the scores of other $p-1$ classes for each image. That is, for the i-th image with the ground truth label y_i, $b_{i,j} = [s_{i,y_i} - s_{i,j} > 0]$, where $1 \leq j \leq p$, $j \neq y_i$, and $1 \leq i \leq n$. For tasks with many binary variables such as CIFAR-100, we train a portion of binary variables in each update to accelerate training.

Table 3.3: Ablation study for mini-batch sizes on CIFAR-10.

Batch Size	8	16	32	64	128	256	512	1024
Accuracy	87.89	90.05	90.82	91.23	**91.64**	90.94	89.10	87.20

Table 3.4: Ablation study for number of anchors on the three tasks.

Task	#Anchors in Each Type	Performance
Bin. Classification (AP)	5	0.9988
	10	0.9987
	16	0.9988
Classification (Acc)	5	91.55%
	10	91.54%
	16	91.64%
Pose Estimation (PCKh)	5	94.79%
	10	94.95%
	16	95.77%

3.4.4.3 Results

Our implementation of the baseline method obtains a slightly better accuracy (91.49%) than that was reported in (He et al., 2016)—91.25% on CIFAR-10 and obtains 65.9% on CIFAR-100. UniLoss performs similarly (91.64% and 65.92%) as baselines on both datasets (Table 3.2), which shows that even when the conventional loss is well-established for the particular task and metric, UniLoss still matches the conventional loss.

3.4.5 Discussion of Hyper-parameters

3.4.5.1 Mini-batch Sizes

We also use a mini-batch of images for updates with UniLoss. Intuitively, as long as the batch size is not extremely small or large, it should be able to approximate the distribution of the whole dataset.

We explore different batch sizes on the CIFAR-10 multi-class classification task, as shown in Table 3.3. The results match with our hypothesis—as long as the batch size is not extreme, the performance is similar. A batch size of 128 gives the best performance.

3.4.5.2 Number of Anchors

We explore different number of anchors in the three tasks. We experiment with 5, 10, 16 as the number of anchors for each type of the good, bad and nearby anchors. That is, we have 15, 30, 48 anchors in total respectively.

Table 3.4 shows that binary classification and classification are less sensitive to the number of anchors, while in pose estimation, fewer anchors lead to slightly worse performance. It is related to the number of binary variables in each task: pose estimation has scores for each pixel, thus has much more comparisons than binary classification and classification. With more binary variables, more anchors tend to be more beneficial.

3.5 Conclusions

We have introduced UniLoss, a framework for generating surrogate losses in a unified way, reducing the amount of manual design of task-specific surrogate losses. The proposed framework is based on the observation that there exists a common refactorization of the evaluation computation for many tasks and performance metrics. Using this refactorization we generate a unified differentiable approximation of the evaluation computation, through interpolation. We demonstrate that using UniLoss, we can optimize for various tasks and performance metrics, achieving comparable performance as task-specific losses.

3.6 Appendix

3.6.1 Binary Classification

3.6.1.1 Network Architecture

The network takes a 28×28 image as input and flatten it to be one dimension with 784 elements. It then pass the flattened input through two fully-connected layers, one with 500 neurons and one with 300 neurons. Each fully-connected layer has a ReLU activation layer followed. The prediction is then given with an output layer with a 2-way output.

3.6.1.2 Training Details

The baseline loss is a 2-class cross-entropy loss. Our loss is formulated as described in Sec. 4.2. The h function is the sigmoid relaxation of the binary variables. The g function is the interpolation over anchors of the binary variables. For example, the anchor that gives the best performance is that all positive examples have higher scores than negative examples—all $b_{i,j} = 1$. Following Sec. 3.4, three types of anchors are generated randomly. The good anchors and nearby anchors are obtained by flipping one binary bit from the best anchor and the anchor at the current training step respectively.We sample 16 anchors for each type.

We train models with the baseline loss and UniLoss with SGD using the same training schedule: with a fixed learning rate of 0.01 for 30 epochs.

3.6.2 Pose Estimation

3.6.2.1 Network Architecture

We use the Stacked Hourglass Network architecture. It takes a 224×224 image as input and passes it through one hourglass block as described in (Newell et al., 2016). It outputs a heatmap for each joint. A heatmap essentially gives scores measuring how likely is the joint to be there for each pixel or region centered at that pixel.

3.6.2.2 Training Details

The baseline loss is the Mean Squared Error (MSE) between the predicted heatmaps and the manually-designed "ground truth" heatmaps. More specifically, the target "ground truth" heatmap has a 2D Gaussian bump centered on the ground truth joint. The shape of the Gaussian bump is controlled by its variance σ and the bump size. The commonly chosen σ is 1 and the bump size 7.

Our loss is formulated as in Sec. 4.3. The h function is the sigmoid relaxation of the binary variables. The g function is the interpolation over anchors of the binary variables. For example, the anchor that gives the best performance is that all binary values to be 1. Following Sec. 3.4, three types of anchors are generated. For good anchors, we flip a small number of bits from the best. Nearby anchors are flipped from the current configuration by randomly picking a positive/negative pixel in the current output heatmaps and flipping all bits associated with this pixel. We sample 16 anchors for each type.

We train the model both UniLoss and MSE with RMSProp (Hinton et al., 2012) using an initial learning rate 2.5e-4 for 30 epochs and then divided by 4 for every 10 epochs until 50 epochs.

3.6.3 Classification

3.6.3.1 Network Architecture

We use the ResNet-20 architecture (He et al., 2016). The network takes a 32×32 image as input. The input image first goes through a 3×3 convolution layer followed by a batch normalization layer and a ReLU layer. It then goes through three ResNet building blocks with down-sampling in between. After an average pooling layer, the output layer is a fully-connected layer with a 10-way output.

3.6.3.2 Training Details

Our baseline loss is a 10-way cross-entropy (CE) loss. Our loss is formulated as in Sec. 4.4. The h function is the sigmoid relaxation of the binary variables. The g function is the interpolation over anchors of the binary variables. For example, the anchor that gives the best performance is that all binary values to be 1, meaning that for each image, the ground truth class has higher score than every other class. The good anchors and nearby anchors are obtained by flipping one binary bit from the best anchor and the anchor at the current training step respectively.We sample 16 anchors for each type.

We use the same data augmentation for both with random cropping with a padding of 4 and random horizontal flipping. We also pre-process the images with per-pixel normalization. We train both models with SGD using an initial learning rate of 0.1, divided by 10 and 100 at the 140 epoch and the 160 epoch, with a total of 200 epochs on CIFAR-10. On CIFAR-100, we train baseline with the same training schedule and UniLoss with 5x training schedule but 20% binary variables at each step.

CHAPTER 4

Synthetic Data Generation for Small-data Object Detection [1]

4.1 Introduction

Synthetic Data Generation methods (Shrivastava et al., 2017a; Yang and Deng, 2020; Sankaranarayanan et al., 2018) has been proposed in various tasks to address the difficulties to collect real data, including semantic segmentation, gaze estimation, 3D reconstruction and so on.

In this chapter, we explore a new method to use synthetic data generation to improve small data object detection. Object detection (Lin et al., 2018b; Ren et al., 2015) requires a large amount of training data to obtain good performance. But for many object detection tasks, large datasets are difficult to obtain due to rare objects and difficulties in obtaining object location annotations. One common example is with medical images – disease detection has very little labeled object bounding box data because the diseases by nature are rare, and annotations can only be done by professionals, and thus are costly. Solving such rare data object detection problems is valuable: for example, for disease localization, a good disease detector can help provide assistance to radiologists to accelerate the analysis process and reduce the chance of missing tumors, or even provide a medical report directly if a radiologist is not available.

In this chapter we explore using generative models to improve the performance in small-data object detection. Directly applying existing generative models is problematic. First, previous work on object insertion for generative models often needs segmentation masks, which are often not available e.g. in disease detection tasks. Second, GANs are designed to produce realistic

[1]This chapter is based on a joint work with Michael Muelly, Jia Deng, Tomas Pfister and Li-Jia Li

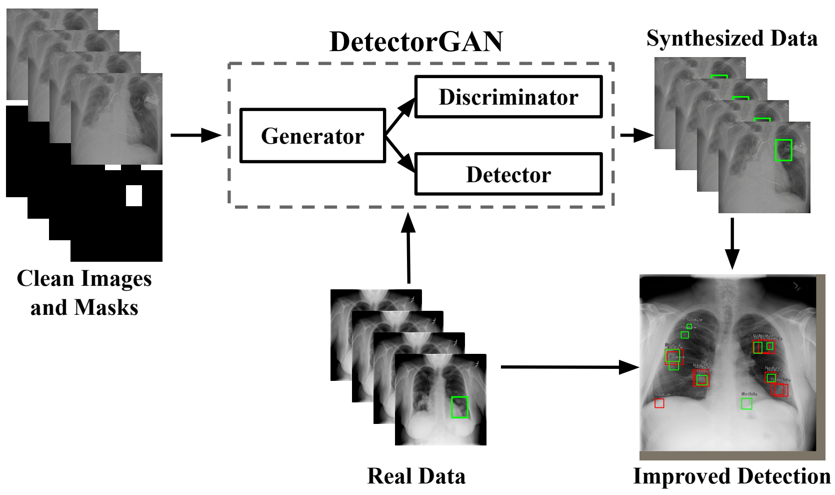


Figure 4.1: DetectorGAN generates object-inserted images as synthesized data to improve the detection performance. DetectorGAN integrates a detector into the generator-discriminator loop.

images (indistinguishable from real images), but realism does not guarantee that it can help with the downstream object detection task. In particular, there is no direct feedback from the detector to the generator; which means the generator cannot be trained explicitly to improve the detector.

To address this, we propose a new DetectorGAN model (shown in Fig. 4.1) that connects the detector and the GAN together. This joint model integrates a detector into the generator-discriminator pipeline and trains the generator to explicitly improve the detection performance.

DetectorGAN has two branches after the generator: one with discriminators to improve realism and interpretability of the generated images, and another with a detector to give feedback on how well the generated images improve the detector. We jointly optimize the adversarial losses and detection losses. To generate images that are beneficial for the detector, the loss formulation is non-trivial. One difficulty is that our goal is for the generated images to improve the detector performance of real images, but the generator cannot receive gradients from the detection loss on real images because the real images are not generated. To address this, the proposed method bridges this link between the generator and the detection loss on real images by unrolling one forward-backward pass of the detector training.

We demonstrate the effectiveness of using DetectorGAN to improve small-data object detection in two datasets for disease detection and pedestrian detection. The detector-integrated GAN model achieves state of the art performance on the NIH chest X-ray disease localization task, benefiting from the additional generated training data. In particular, DetectorGAN improves the Average Precision of the nodule detector by a relative 20% by adding 1000 synthetic images, and outperform the state of the art on localization accuracy by a relative 50%. We also show that the proposed framework significantly improves the quality of the generated images: a radiologist prefers generated images by DetectorGAN over alternative methods in 96% of cases. The detector model can be integrated into almost any existing GAN models to force them to generate images that are both realistic and useful for downstream tasks. We give the pedestrian detection task and the associated PS-GAN (Ouyang et al., 2018) as an example, demonstrating a significant quantitative and qualitative improvement in the generated images.

Our contributions are:

- To the best of our knowledge, this work is first to integrate a detector into the GAN pipeline so that the detector gives direct feedback to the generator to help generate images that are beneficial for detection.
- We propose a novel unrolling method to bridge the gap between the generator and the detection performance on real images.
- The proposed model outperforms GAN baselines on two challenging tasks including disease detection and pedestrian detection, and achieves the state-of-the-art performance on NIH chest X-ray disease localization.
- We are the first few works to explore generating synthetic data using GANs in small-data object detection.

4.2 Related Work

4.2.1 Image-to-image Translation.

Based on a conditional version of Generative Adversarial Networks (GANs) (Goodfellow et al., 2014), Isola et al (Isola et al., 2017) pioneered the general image-to-image translation task. Afterwards multiple other works have also exploited pixel-level reconstruction constraints to transfer between source and target domains (Zhang et al., 2017; Wang et al., 2018). These image-to-image translation frameworks are powerful, but require training data with paired source/target images, which are often difficult to obtain. Unpaired image-to-image translation frameworks (Zhu et al., 2017; Liu et al., 2017a; Shrivastava et al., 2017b; Kim et al., 2017; Lee et al., 2018b) remove this requirement of paired-image supervision; in CycleGAN (Zhu et al., 2017) this is achieved by enforcing a bi-directional prediction between source and target. The proposed DetectorGAN falls in the category of unpaired image-to-image translation frameworks. Its novelty is that it integrates a detector into GAN to generate images as training data for object detection.

4.2.2 Object Insertion with GANs.

The idea of manipulating images by GANs has been explored recently (Lee et al., 2018a; Hong et al., 2018; Chien et al., 2017; Lin et al., 2018a; Ouyang et al., 2018; Lee et al., 2019; Lin et al., 2018a). These works use generative models to edit objects in the scene. In contrast, (1) our method doesn't require any segmentation information; and (2) our goal is to gain quantitative improvement on object detection task while prior works focus on qualitative improvement such as realism.

4.2.3 Integration of GANs and Classifiers.

Beyond the basic idea of using adversarial losses to generate realistic images, some GAN models integrate auxiliary classifiers into the generative model pipeline, such as Auxiliary Classifier GAN (ACGAN) and related works (Odena et al., 2017; Bazrafkan and Corcoran, 2018; Dash et al., 2017; Bousmalis et al., 2017; Hoffman et al., 2018). At a first glance, these models bear some similarity with our integration with detector. However, we differ from them both conceptually and technically. Conceptually, these methods only improve the realism of the generated images and have no intention to improve the integrated classifier; in contrast, the purpose of our integration is to improve the detection performance. Technically, our loss formulation is different: ACGAN minimizes classification losses only on synthetic images and has no guarantee for improving performance on real images, whereas ours optimizes losses on both synthetic and real by adding unrolling step. Nevertheless, we construct a baseline with ACGAN-like losses, which only minimizing detection losses on synthetic images, and show that our proposed method outperforms it.

4.2.4 Data Augmentation for Object Detection

There are some works using data augmentation to improve object detection. A-Fast-RCNN (Wang et al., 2017b) uses adversarial learning to generate hard data augmentation transformations, specifically for occlusions and deformations. It differs from our method in two major ways: (1) It is not a GAN model – it does not generate images but instead adds adversarial data augmentation into the detector network. In contrast, our model has a discriminator and detector that work together to generate synthetic images. (2) Its goal is to 'learn an object detector that is invariant to occlusions and deformations'. In contrast, our method focuses on generating synthetic data for the problem setting where the amount of training data is limited.

Perceptual GAN (Li et al., 2017a) generates synthetic images to improve the detection. However, it is designed specifically for small-sized object detection by super-resolving the small-sized objects into better representations. Their method does not generalize to general object detection.

Concurrent unpublished work PS-GAN (Ouyang et al., 2018) is most closely related: synthetic images are generated to improve pedestrian detection. They generate synthetic images using a traditional generator-discriminator architecture. In contrast, we add a detector in the generator-discriminator loop and have direct feedback from the detector to the generator.

4.3 DetectorGAN

Our DetectorGAN method generates synthetic training images to directly improve the detection performance. It has three components: a generator, (multiple) discriminators, and a detector. The detector gives feedback to the generator about whether the generated images are improving the detection performance. The discriminators improve the realism and interpretablity of the generated images; that is, the discriminators help to produce realistic and understandable synthetic images.

4.3.1 Model Architecture

We implement our architecture based on CycleGAN (Zhu et al., 2017). The generator in DetectorGAN generates synthetic labelled (object-inserted) images that are fed into two branches later: the discriminator branch and the detector branch. We consider clean images without objects belong to domain X, and labelled images with objects belong to domain Y.

Generators. We use a ResNet generator with 9 blocks as our generators G_X and G_Y following (Zhu et al., 2017; He et al., 2016). The forward generator G_X takes two inputs: one is a real clean image, which is used as the background image to insert objects. The other one is a mask where the pixels inside the bounding box of the object to insert are filled with ones while the rest are zeros. The output of the generator is a synthetic image with the input background and an object inserted at the marked location. Inversely, the backward generator G_Y takes a real labelled image and a mask showing the object location, and outputs an image with the indicated object removed.

Plausible inserting locations of objects are difficult to obtain. For the NIH disease task, we obtain these locations by pre-processing and random sampling. In theory, the location could be in any position in the lung area, but since in practice we do not have segmentation mask for the lung area, we first match each clean image to the most similar labelled image with bounding box and then randomly shift the location around to get the sampled ground-truth box location. For the pedestrian detection task, we follow the setup in the previous work (Ouyang et al., 2018). It is notable that the selection of mask locations does not change our method – as an alternative one could use trainable methods to predict plausible locations.

Discriminators. Our method contains two global discriminators $DIS_{globalX}$ and $DIS_{globalY}$ as in Cycle-GAN (Zhu et al., 2017), and a local discriminator DIS_{localX} for local area realism (Lee et al., 2019; Li et al., 2017b). The global discriminator $DIS_{globalX}$ and the local discriminator DIS_{localX} discriminates between real labelled images and synthetic labelled images (generated by G_X), globally on the whole images or locally on the bounding box crops. $DIS_{globalY}$ discriminates what G_Y generates (synthetic clean images by removing objects from real labelled images) and real clean images. DIS_{localY} is not needed because conceptually we do not care much about the local realism after removing an object. We use 70×70 PatchGAN following (Johnson et al., 2016; Isola et al., 2017; Zhu et al., 2017) for all of our discriminators.

Detector. The detector DET takes both real and synthetic labelled images with objects as input and outputs bounding boxes. In our implementation we use the RetinaNet detector (Lin et al., 2018b). But we are not only limited to RetinaNet: as long as the detector is trainable, we can integrate it into the loop.

4.3.2 Train Generator with Detection Losses

The objective of the generator G_X is to generate images with objects inserted that are both realistic and beneficial to improve object detection performance. One of our main contributions is that we propose a way to backpropagate the gradients derived from detection losses back to the generator to help the generator to generate images that can better help improve the detector. In other words, the detection losses give the generator feedback to generate useful images for the detector.

We note the detection loss (regression and classification losses) as $L_d(\cdot)$, where $\cdot$ is a labelled

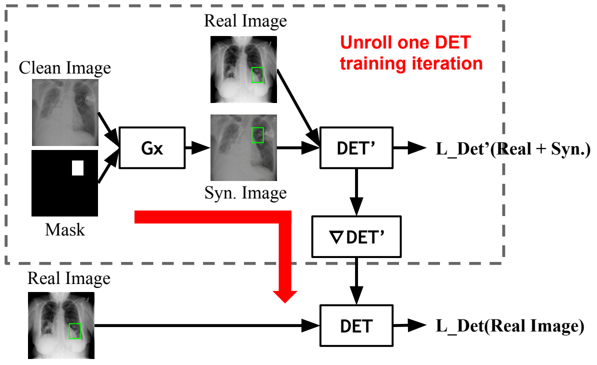


Figure 4.2: The illustration for Eqn. 4.3 – unrolling one forward-backward pass for training DET to bridge the link between G_X and L_{Det}^{real} (detection loss on real images). Detection loss on real images has no direct link to the generator G_X. Last step of training old DET (noted as DET' in the figure, refers to same DET module but in the previous training step) is unrolled as in the dotted rectangle. The red arrow represents the fact that there is a differentiable link between G_X and L_{Det}^{real} after the unrolling.

image, either real or synthetic. The detection loss on real images and synthetic images are:

$$L_{Det}^{real}(DET) = E_{y \sim p_{data}(Y)}[L_d(DET(y))] \tag{4.1}$$

$$L_{Det}^{syn}(G_X, DET) = E_{x \sim p_{data}(X)}[L_d(DET(G_X(x)))] \tag{4.2}$$

Unroll to Optimize Detection Loss on Real Images. Intuitively, given a real image y, the goal of G_X is to use generated images to help minimize the detection loss on real images. That is, G_X should be trained to minimize the loss L_{Det}^{real} in Eqn. 4.1. However, there is no G_X involved at the first glance – the loss L_{Det}^{real} does not depend on the weights of the G_X so G_X cannot be trained. But we observe that even though there is no direct link in one forward-backward loop from G_X to real images, the detector is trained by synthetic images generated by G_X in the previous step. We propose to bridge the link between G_X and the real image detection loss L_{Det}^{real} by unrolling a single forward-backward pass of the detector as shown in Eqn. 4.3. A visualization of this unrolling process is shown in Fig. 4.2. This allows us to train G_X with respect to the loss L_{Det}^{real}.

$$\begin{gathered}\tilde{L}_{Det}^{real}(G_X, DET) = E_{y\sim p_{data}(Y)}[L_d(DET(y))] \\ \text{where weights of } DET\text{, } W_{DET}\text{, is updated with} \\ \frac{\partial(L_{Det}^{real}(DET) + L_{Det}^{syn}(G_X, DET)))}{\partial W_{DET}}\end{gathered} \tag{4.3}$$

Specifically, we train the weights DET with synthetic images and real images for one iteration and obtain the gradients on DET. These gradients are linked to the generated synthetic images and thus to the weights in the generator G_X. Then we use the updated DET to get the L_{Det}^{real} loss and gradients. In this way, we obtain a link from G_X to DET and then to L_{Det}^{real}.

Intuitively, this Eqn. 4.3 can be seen as a simple estimation of how the change in G_X will change detection performance on real images in Eqn. 4.1.

Detection Loss on Synthetic Images. The generator aims to make the synthetic images helpful for the detector. It maximizes the detection loss on synthetic images (Eqn. 4.2) to generate images that the detector has not seen before and cannot predict well. In this case the generated images can help improve the performance.

One might think the generator should instead minimize the detection loss on synthetic images. This shares some similar ideas with ACGAN-like losses, where the auxiliary classification loss on synthetic images is minimized to improve realism. But for our goal to improve the detection performance on real images, minimizing detection losses on synthetic images may not help, or may even hurt the detection performance on real images. The intuition behind this is that synthetic objects may distract away from the optimization goal of the detector. In our experiments, we show that minimizing synthetic image losses like ACGAN harms detection performance on real images.

Table 4.1: Nodule AP on expanded annotation setting on the test set with IoU = 0.1 for NIH. Baseline is using only real training data. We add 1000 synthetic images from CycleGAN and GAN-D for training.

Training data	Nodule AP	Nodule Recall
Real data only	0.124	0.184
Real + syn from ACGAN-like losses	0.154	0.607
Real + syn from CycleGAN + BboxLoss	0.196	0.541
Real + syn from DetectorGAN - unrolling	0.203	0.544
Real + syn from DetectorGAN	**0.236**	**0.649**

Table 4.2: Localization accuracy with different T_{IOU} on "old annotations" test set for NIH. The latter three uses RetinaNet as the model.

T_{IOU}	0.1	0.2	0.3	0.4	0.5	0.6	0.7	Avg
Wang et al. (Wang et al., 2017a)	0.14	0.05	0.04	0.01	0.01	0.01	0.00	0.04
Zhe et al. (Li et al., 2018)	**0.40**	0.29	0.17	0.11	0.07	0.03	0.01	0.15
real	0.15	0.15	0.15	0.08	0.08	0.00	0.00	0.09
real + syn from CycleGAN + BboxLoss	0.31	0.31	0.23	0.23	0.00	0.00	0.00	0.15
real + syn from DetectorGAN	0.31	**0.31**	**0.31**	**0.23**	**0.23**	**0.15**	**0.08**	**0.23**

4.3.3 Overall Losses and Training

Overall Losses. The objective of the generator G_X is to generate images with an object inserted at the indicated location in background images. The generated images should be both realistic and beneficial to improve object detection performance. In other words, the DetectorGAN model should generate images that: (1) can help to train a better detector; (2) have an object inserted; and (3) are indistinguishable from real images, globally and locally.

We have introduced losses to help the detector above. For inserting an object, we use an L1 loss to minimize the loss between the synthetic object crop and the real object crop (refered as BboxLoss):

To generate realistic images, we have adversarial losses for the global discriminators and the local discriminator. For $DIS_{globalX}$, the adversarial loss is $L_{GAN}(G_X, DIS_{globalX})$ as in Eqn. 4.4. $L_{GAN}(G_Y, DIS_{globalY})$ and $L_{GAN}(G_X, DIS_{localX})$ are similar.

$$\begin{aligned} L_{GAN}(G_X, DIS_{globalX}) = {} & E_{y \sim p_{data}(Y)}[\log DIS_{globalX}(y)] \\ & + E_{x \sim p_{data}(X)}[\log(1 - DIS_{globalX}(G_X(x)))] \end{aligned} \tag{4.4}$$

In addition, we use cycle consistency losses and identity losses to help preserve information from the whole image.

Here G_X and G_Y aim to fool the discriminators while the discriminators aim to discriminate between fake and real images. The generator and discriminators thus optimize

$$\begin{aligned} min_{G_X,G_Y} max_{DIS_{globalX},DIS_{localX},DIS_{globalY}} & [L_{GAN}(G_X, DIS_{globalX}) \\ & + L_{GAN}(G_Y, DIS_{globalY}) + L_{GAN}(G_X, DIS_{localX})] \end{aligned} \tag{4.5}$$

We update the weights of the detector DET by minimizing the detection losses for both real images and synthetic images: it minimizes Eqn. 4.1 and Eqn. 4.2.

Training. In summary, when updating the discriminators, the goal is to maximize the discriminator losses on generated images and minimize the losses on real nodule images. When updating the detector, the goal is to minimize the detection losses for both real and generated nodule images. When updating the generator, the goal is to: (1) minimize the discriminator losses on generated images; (2) minimize detection loss on real object images, (3) maximize detection loss on generated images.

We use a history of synthetic images (Shrivastava et al., 2017b), and for faster convergence we pretrain the discriminator-generator pair and the detector separately and then train them jointly. When we have a labelled image without bounding box annotations, we still update the discriminator $DIS_{globalX}$ to improve global realism.

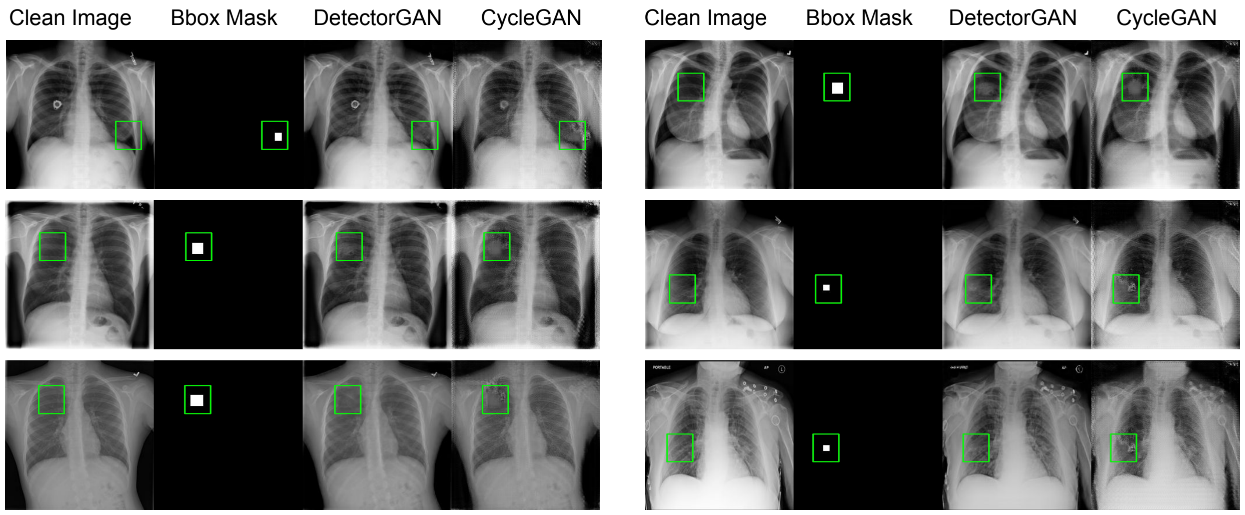


Figure 4.3: Example generated images from CycleGAN and DetectorGAN for NIH. The details are high-lighted in green boxes (added for visualization). Both methods generate synthetic images from clean images and bounding box masks. DetectorGAN generates nodule inserted images with better local and global quality.

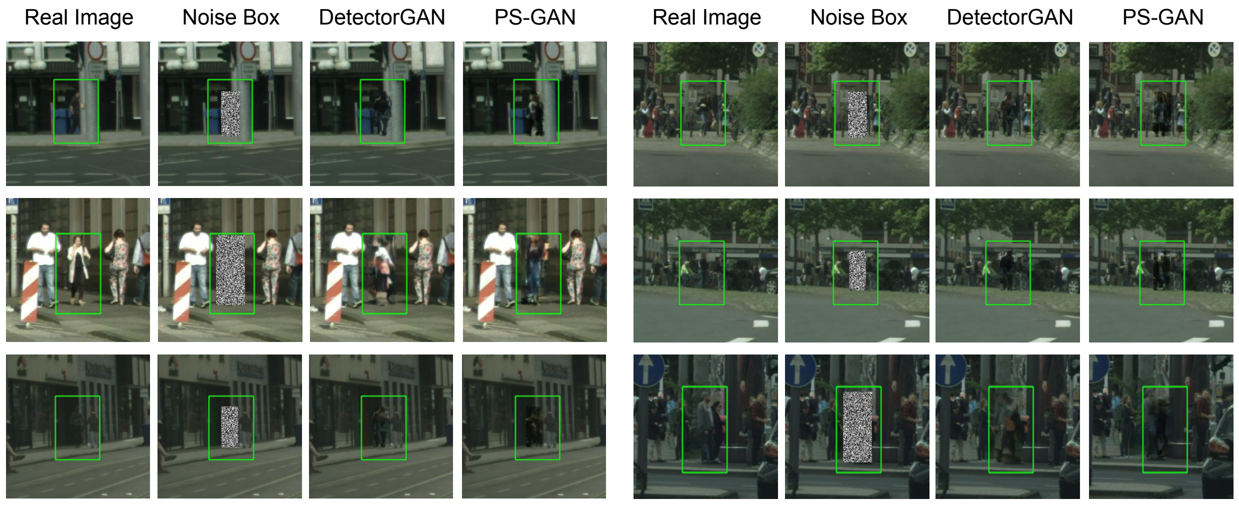


Figure 4.4: Examples of generated synthetic images from PS-GAN and DetectorGAN. The details are high-lighted in green boxes (added for visualization). We see qualitative improvement for pedestrian task as well.

4.4 Experiments

In this section we demonstrate the effectiveness of DetectorGAN on two tasks: nodule detection task with the NIH Chest X-ray dataset and pedestrian detection with the Cityscapes dataset.

Table 4.3: Localization accuracy with different T_{IOBB} on the "old annotations" test set on NIH.

T_{IOBB}	0.1	0.25	0.5	0.75	Avg
Wang et al. (Wang et al., 2017a)	0.15	0.05	0.00	0.00	0.04
Zhe et al. (Li et al., 2018)	**0.40**	0.25	0.11	0.07	0.18
RetinaNet: real	0.15	0.15	0.08	0.08	0.09
RetinaNet: real + syn from CycleGAN + BboxLoss	0.31	0.31	0.00	0.00	0.12
RetinaNet: real + syn from DetectorGAN	0.31	**0.31**	**0.23**	**0.23**	**0.22**

Table 4.4: User study on the NIH Chest X-ray dataset.

Method	Prefer [%] ↑		Mean Likert ↑		Std Likert	
	Object	Whole	Object	Whole	Object	Whole
CycleGAN + BboxLoss	20	4	1.31	1.18	0.68	0.53
DetectorGAN	**80**	**96**	**2.69**	**3.88**	0.85	0.73

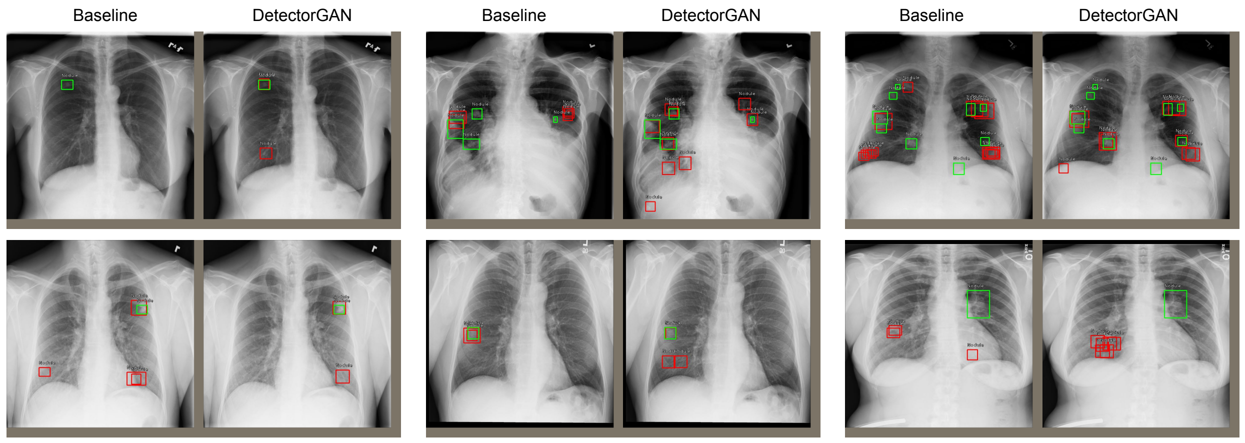


Figure 4.5: Comparison showing that adding synthetic images can help detect nodules in NIH Chest X-ray more accurately. Here, green boxes are ground truth and red are predictions.

We obtain significant improvements over baselines and achieve state of the art results on the nodule detection task.

4.4.1 Disease Localization

4.4.1.1 Dataset

We use the NIH Chest X-ray dataset (Wang et al., 2017a) and focus on the nodule detection task. The NIH Chest X-ray dataset contains 112,120 X-ray images – 60,412 clean images and 51,708 disease images, 880 of which have bounding boxes. For the nodule class, there are 6,323 nodule images, 78 of which have bounding boxes.

Improved and Extended Annotations. The bounding box annotation for this dataset is however not satisfying due to the following issues: (1) In the original paper and previous work (Wang et al., 2017a; Li et al., 2018), there is no standard train/test/validation split. (2) The bounding box annotations are not complete; that is, for each image there is only at most one bounding box for each class annotated, while there are actually many nodules present in the image. (3) Even with a standard train/test/validation split, the test and validation sets are too small to obtain stable and meaningful results.

To address these problems, we make the following efforts to make the disease detection task more standard and easy to conduct research on: (1) Generating a no-patient-overlap train/test/validation split with 0.7/0.2/0.1 portion of the data, yielding 57/13/9 images with 57/13/9 object instances. (2) Asking radiologists to re-annotate the current validation and test images using additional images from labeled images in the test/validation sets. These efforts result in 36 images and 80 images in validation and test sets accordingly, with 159 and 309 object instances. These splits and extended annotations will be published online to facilitate future research into this topic. We did not re-annotate or expand the training set as we want to demonstrate the effectiveness of the proposed method in learning small-data object detection tasks.

We refer to the 9/13 validation/test settings as "old annotations" and the 36/80 validation/test settings as the "new annotations". We obtain the detection AP on the "new annotations" and localization accuracy on the "old annotations" for fair comparison with previously published results.

Baselines and Previous Work. The baselines are: training with only real images, with additional synthetic images generated from CycleGAN and BboxLoss, and with additional synthetic images generated from ACGAN-like losses. The ACGAN-like losses refers to that in addition to discriminator losses, we also minimize the detection loss on synthetic images, similar to what ACGAN does for a classifier. We compare these methods on the new high quality annotations. In addition, we compare to two previously published best-performing works (Wang et al., 2017a; Li et al., 2018) using their evaluation split and their annotations (the "old annotations").

Evaluation Metrics. We use the standard object detection metric, average precision (AP), as the evaluation measure for the detection task. For comparisons to previous work, we also use their metric: localization accuracy, which is defined as the percentage of images that obtain correct predictions. An image is considered having correct predictions if the intersection over union (IOU) ratio between the predicted regions (can be non-rectangle) and the ground truth box is above threshold T_{IOU}. Another metric that is used by these works is to replace the IOU with intersection over bounding boxes $IOBB$. However, we encourage researchers to use the proposed new annotations and evaluation metric in the future for standard comparisons.

4.4.1.2 Quantitative Comparison

New Annotation with Average Precision. In Table 4.1, we compare the results of using only real data, using synthetic data from the proposed method as well as from other baseline GAN models. We observe that DetectorGAN significantly improves the average precision. Compared to training on real data only, the AP nearly doubles from 0.124 to 0.236, and recall over triples from 0.184 to 0.649. Compared to ACGAN-like losses and CycleGAN + BboxLoss, we obtain relatively 50% and 20% improvement.

We notice that ACGAN-like losses performs more poorly than using discriminator losses only, even though it has an additional loss to improve the detection performance on synthetic images. One explanation is that the generator and the detector learn only to detect synthetic objects, which is different from the goal of detecting real objects, leading to poor performance.

To further demonstrate the benefits of using the unrolling step to bridge the gap between the generator and the detection performance on real images, we also experiment with a 'DetectorGAN - unrolling' network without unrolling. We observe a significant boost for adding the unrolling step, from 0.203 to 0.236 AP.

Old Annotation with Localization Accuracy. For comparison with previous work, we evaluate detection results using the localization accuracy metric with different IOU and $IOBB$ thresholds. Results are shown in Table 4.2 and Table 4.3. We significantly outperform competing methods by relative 50% and 22%.

4.4.1.3 Qualitative Analysis

Generated Image Quality. We show DetectorGAN's generated images, along with CycleGAN-generated images in Fig. 4.3. We observe that images are much better in terms of realism and blend-in.

Detected Nodules. We show that the detector helps to detect undetectable nodules in Fig. 4.5. We observe that every nodule captured by the baseline (trained on real images only) is also captured by the model trained using synthetic images. Meanwhile, adding synthetic images helps capture more nodules that baseline cannot capture. Moreover, the box locations are generally more accurate.

4.4.1.4 User Study

We also conduct user study with a radiologist to evaluate the quality of the generated images. We ask the radiologists to rate the realism of the inserted nodule and the global image on a Likert scale (scale 1–5, with 5 indicating highest quality). As shown in Table 4.4, the images from DetectorGAN are better than those from CycleGAN + BboxLoss in 96% of cases, with generated objects (nodules) better in 80% of cases. Moreover, the average Likert scores are significantly

Table 4.5: Pedestrian detection AP trained with real data, synthetic data generated by PS-GAN, pix2pix and DetectorGAN.

Data	Real	+DetectorGAN	+PS-GAN	+pix2pix
AP	0.593	**0.613**	0.602	0.574

higher: 2.69 vs 1.31 for the objects, and 3.88 vs 1.18 for the whole image, demonstrating the benefits of our method.

4.4.2 Pedestrian Detection

As a demonstration of the applicability of DetectorGAN to other datasets and problems, we apply it to pedestrian detection with a different base architecture. We follow PS-GAN (Ouyang et al., 2018) to synthesize images with pedestrians inserted and improve pedestrian detection. We demonstrate a quantitative and qualitative improvement in the generated images by adding the detector into the loop.

Dataset. We use the Cityscapes dataset, which contains 5,000 urban scene images with high-quality annotations. We follow the instructions in the PS-GAN paper to filter images with small or occluded pedestrians obtain about 2,000 images with about 9,000 labeled instances.

Baseline and Architecture. We use PS-GAN (Ouyang et al., 2018) as the backbone architecture and add the detector into the model. The PS-GAN uses the standard pix2pix framework with local discriminators. This also shows that the DetectorGAN idea is versatile — it can be integrated with different GAN models. We fine-tune the model from the pretrained PS-GAN model.

Quantitative Results. Table 4.5 shows that we improve the detection performance for pedestrian detection as well. We observe that DetectorGAN further improves the performance over PS-GAN.

All models here are trained using the same setting. The real-images-only baseline performance is slightly different from what is reported in the PS-GAN paper because we do not have access to the exact details of the detector setting used in the PS-GAN paper.

Qualitative Results. Qualitative results are shown in Fig. 4.4. We observe that DetectorGAN can generate qualitatively better images with less artifacts.

4.5 Conclusion

In this chapter we learn to generate new images with associated bounding boxes to explore the object detection problem in the small data regime. We have shown that simply training an existing generative model does not yield satisfactory performance due to it optimizing for image realism instead of object detection accuracy. To this end we developed a new model with a novel unrolling step that jointly optimizes a generative model and a detector such that the generated images improve the performance of the detector. We show that this method significantly outperforms the state of the art on two challenging datasets.

CHAPTER 5

Architecture Search for GANs [1]

5.1 Introduction

In this chapter, we explore architecture search for Generative Adversarial Networks (GANs). Neural Architecture Search (NAS) (Zoph et al., 2018; Zoph and Le, 2017) aims to reduce the human intervention by automating network design. Researchers use Reinforcement Learning (Zoph et al., 2018; Zoph and Le, 2017; Pham et al., 2018), Evolutionary Algorithm (Real et al., 2019), gradient based (Luo et al., 2018; Liu et al., 2019b), Random Search (Xie et al., 2019) and progressive search (Liu et al., 2018) for image classification task. In addition, there are also recent works applying NAS on segmentation (Liu et al., 2019a; Chen et al., 2018), machine translation (So et al., 2019), and transfer learning (Wong et al., 2018).

There is however little effective effort has been made for GANs, especially considering the difficulties and instabilities (Arjovsky et al., 2017; Heusel et al., 2017) to train large-scale GANs—as a two-player game, generator or discriminator could easily dominate the training and cause the training signal to vanish or explode. With bigger networks and higher resolutions, this issue gets worse with more complex network parameter space.

Recent work introduced the progressive growing of GANs (Karras et al., 2018) to ease the training for large GANs. It starts with generating low-resolution images with a small network and then progressively adds new layers to the network to generate higher-resolution images. As demonstrated by (Karras et al., 2018), it makes training large GANs easier and more stable and

[1]This chapter is based on a joint work with Yuting Zhang, Jia Deng and Stefano Soatto

improve the quality of generated images in high resolutions.

However, in this progressive-growing process for architectures, the layers and growing schedules are predefined by researchers. There is a large space of the growing strategy that still remains under-explored. For example, are symmetric generator and discriminator optimal? Are the layer choices optimal?

In this chapter, we propose an automatic method to combine the architecture search idea with the progressive-growing strategy. We explore this rich architecture space by dynamically growing a GAN during training, optimizing the growing strategy together with its network parameters.

Our Dynamically Grown GAN (DGGAN) method embeds architecture search techniques as an interleaving step with gradient-based training to periodically seek the optimal regarding the balancing between the generator and discriminator, choice of network units, and growing strategy. That is, we alternate between optimizing the generator and discriminator architecture and training the new architecture. More specifically, when optimizing the generator and discriminator architecture, our method grows layer(s) from previous architecture, with where to grow (generator or discriminator or both) and how to grow in the automatic framework. The new architecture is then trained with weight inheritance from the previous architecture.

Our method enjoys the benefits of both easing the optimization by progressively growing the architecture and exploring more architecture design space by architecture search. This combination is beneficial to discover well-performing GANs, especially with high-resolution images.

Compared to progressively growing GANs (Karras et al., 2018) with a manually-designed growing strategy, our dynamic growing method explores much richer architecture space and growing strategies. More specifically, our method allows the generator and the discriminator to grow alone or together dynamically in the process, creating diverse and unconventional balance between them.

Compared to prior work AutoGAN (Gong et al., 2019) and AGAN (Wang and Huan, 2019), which combine architecture search with GANs, we complement progressive growing with architecture search to eases the training of GANs with complicated architectures and high resolutions.

This novel perspective enables our method to work on higher resolutions while these prior works only worked on at most 48×48.

Our experiments show that we achieve the new state-of-the-art on CIFAR-10 and the best performance among non part-based GANs on LSUN for image generation. With further analysis of the thousands of models in our search procedure, we observe several practical conclusions on the GAN model design such as generator-discriminator balance and convolutional layer choices.

Our main contributions are:

- We propose a method to dynamically grow GANs, easing the training of GANs as well as exploring unconventional network architecture space;
- We present the first automatic GAN that works on high-resolution images;
- We provide constructive conclusions of generator and discriminator design choices using thousands of searched models;
- We achieve new state-of-the-art image generation performance.

5.2 Related Work

5.2.1 Improving GANs

Since the proposal of GANs (Goodfellow et al., 2014), there have been several lines of work to improve GANs, including improving loss functions (Arjovsky et al., 2017; Deshpande et al., 2018; Berthelot et al., 2017), regularization techniques (Miyato et al., 2018; Gulrajani et al., 2017; Zhang et al., 2020), generation strategy (Lin et al., 2019), and architecture designs (Li et al., 2019; Nguyen et al., 2017; Zhang et al., 2019; Radford et al., 2016; Karras et al., 2018). Improved loss functions and regularization techniques are orthogonal to our work. Our work belongs to the architecture design line of work. While these works improve GAN training, they use manually-designed architectures.

Progressive GAN (ProgGAN) (Karras et al., 2018) shows that progressively growing and training GAN from a smaller scale eases the training difficulties and improves the generation quality. Motivated by this observation, the search strategy in our DGGAN is in a progressive way.

Despite this inspiration, our method is different from ProgGAN because ProgGAN is predesigned and always grows both generator and discriminator symmetrically and simultaneously. Ours automatically searches how and what to grow, and generator and discriminator may not be symmetric. This difference allows us to explore a much richer architecture space than ProgGAN, resulting in better performance. We also provide a thorough analysis in this rich space.

5.2.2 Architecture Search with GANs

AGAN (Wang and Huan, 2019) and AutoGAN (Gong et al., 2019) explore architecture search with GANs, which are closely related to our work. AGAN (Wang and Huan, 2019) learns one RNN controller with the REINFORCE algorithm to search for the architecture of both generator and discriminator simultaneously. Our DGGAN differs from AGAN because our dynamic growing strategy is more flexible: it allows to grow G alone, or D alone, or both. Moreover, our method alternate between growing an architecture and training its weight while AGAN does not have such a training strategy.

AutoGAN (Gong et al., 2019) uses RNN controllers with REINFORCE to search for the generator architecture only. Their discriminator is not searched but constructed with a predefined strategy. It is stated that when searching for both they observe "such two-way NAS will further deteriorate the original unstable GAN training, leading to highly oscillating training curves and often failure of convergence." Our DGGAN, however, deals with the difficulties in training unstable GANs and successfully searches for both generator and discriminator together.

Beyond the methodology difference, both prior works demonstrate image generation with at most 48$\times$48 resolution. Our method, however, supports high-resolution (256$\times$256).

5.2.3 Neural Architecture Search

Neural Architecture Search (NAS) (Zoph et al., 2018; Zoph and Le, 2017) aims to reduce the human intervention by automating network design. Researchers use Reinforcement Learning (Zoph et al., 2018; Zoph and Le, 2017; Pham et al., 2018), Evolutionary Algorithm (Real et al., 2019), gradient based (Luo et al., 2018; Liu et al., 2019b), Random Search (Xie et al., 2019) and progressive search (Liu et al., 2018) for image classification task. In addition, there are also recent works applying NAS on segmentation (Liu et al., 2019a; Chen et al., 2018), machine translation (So et al., 2019), and transfer learning (Wong et al., 2018).

Our method differs from them because 1) we utilize the fact that GAN has two competing components, which does not exist in the tasks above, by allowing each component to grow alone or together; 2) we embed progressive training into architecture search, which is important to models with unstable training, such as GANs. As shown in AutoGAN (Gong et al., 2019), a naive two-way NAS will deteriorate the original unstable GAN training and sometimes lead to failure of convergence.

One related work in this scope is progressive NAS (Liu et al., 2018). It does progressive search inside a cell architecture and stacks the found cell to construct the full network for evaluation. Despite the similarity of wording choice, their progressive search is layer by layer inside a cell but our progressive growing involves growing from lower resolution to higher resolution. Also, we directly construct the whole network(s) while (Liu et al., 2018) constructs a cell architecture and stack it for final architecture. In addition, as a NAS method, it also has the differences described in the last paragraph.

5.2.4 Hybrid Optimization

Our alternate optimization of the architecture and network weights shares a similar idea with the long-existing jump-diffusion processes (Grenander and Miller, 1994) solving hybrid optimization problems with a discrete component and a continuous component. Similar to jump-diffusion, we alternate gradient steps in the continuous parameter space with discrete jumps in architecture

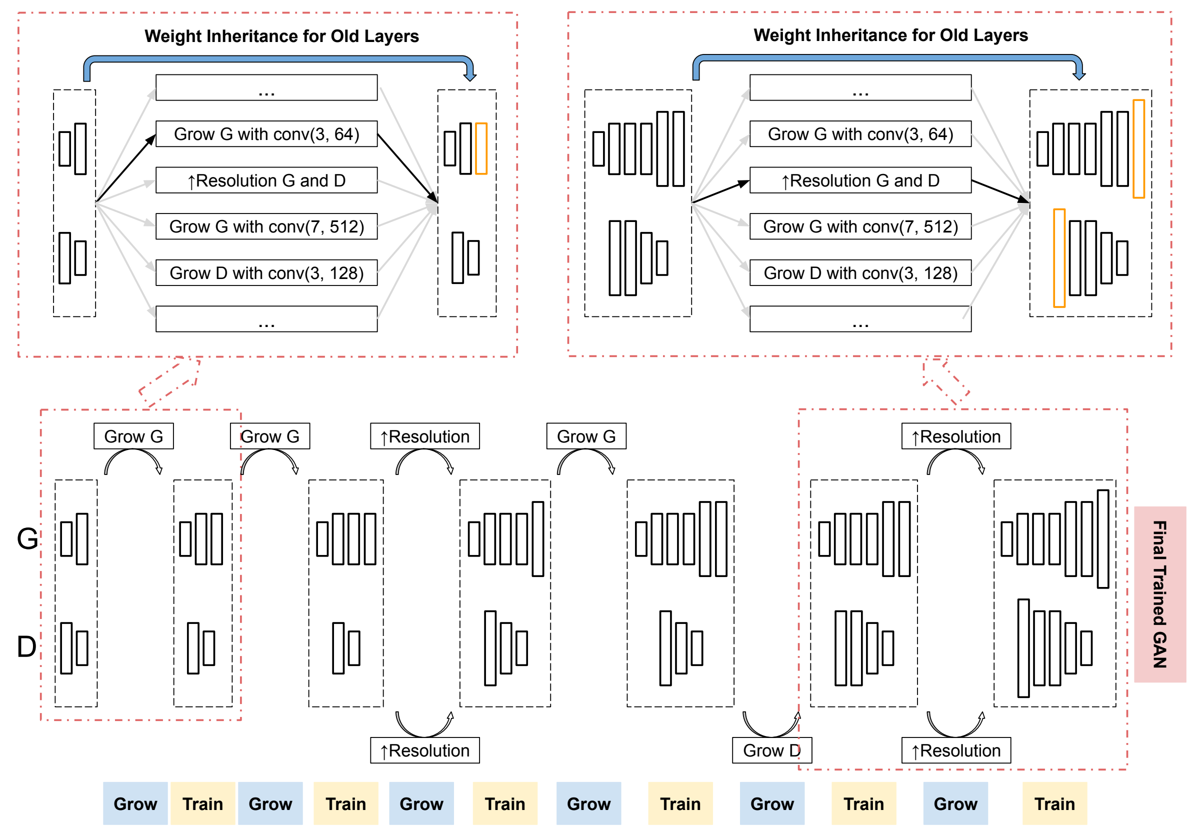


Figure 5.1: Overview of the Dynamically Grown GAN. **Bottom**: The dynamic growing process. We alternate between growing the architecture and training the weights of the new architectures. Each growing step chooses among actions including growing the generator (G) with a certain convolution layer, or growing discriminator (D) with a certain convolution layer, or growing both G and D to a higher resolution. In each training step, the new architectures inherit the weights from the old architectures. **Top**: Examples of growing steps.

space.

5.3 Dynamically Grown GAN

5.3.1 Dynamic Growing Overview

DGGAN embeds architecture search techniques with gradient-based training to periodically seek the optimal regarding network architecture and network weights. We alternate between the growing and training steps to grow a small-scale GAN to the full-scale GAN dynamically and automatically, as shown in Fig. 5.1. We use the Wasserstein distance loss with gradient penalty (Arjovsky et al., 2017; Gulrajani et al., 2017).

The growing at each step is dynamic instead of static as in the conventional progressive GAN method. At the growing step, our method determines which actions in the search space to take, i.e., *where* to grow *what* kind of layers as in Fig. 5.1(top). In contrast, in the conventional progressive GAN method, the growing schedule is fixed and manually-designed. Given existing architectures, noted as parent architectures, we expand parent architectures to new architectures by the growing actions (e.g. add a layer with 256 filters and the filter size of 3). We note the new architectures as child architectures.

At the training step, the child architectures are trained with weight inheritance from the parent architectures. More specifically, a new child architecture contains all layers in the parent architecture as well as the new layer(s) grown. We inherit the weights of the common layers from its parent architecture by initializing the weights of the child architecture with the trained weights of the parent architecture. The newly grown layer(s) that only exist in child candidates are initialized randomly. This weight inheritance method provides two benefits: it eases the optimization difficulties for the child architectures. It also reduces the training time needed for each new candidate.

5.3.1.1 Generator and Discriminator Base Architecture

Following ProgGAN (Karras et al., 2018), the base architectures of generator and discriminator are two small scale networks that operate on a low resolution with $d_0 \times d_0$ pixels. The generator takes a randomly-sampled vector as input and outputs an image with resolution $d_0 \times d_0$. The discriminator takes an image with resolution $d_0 \times d_0$ and outputs a single score. More specifically in our experiments, CIFAR-10 takes random vectors with size 128 and others take random vectors with size 512. d_0 is 8 in our experiments.

When growing a generator, we grow near the end of the network before the last convolution layer. When growing a discriminator, we grow near the beginning of the network after the first convolution layer. Growing heads are indicated as the orange layers in Fig. 5.1(top). The structures of the base architectures thus can be maintained during growing. This choice is following prior work (Karras et al., 2018) as in their study, growing in other locations does not lead to better performance. Note that our method allows us to grow the generator or discriminator alone or both. That is, they do not need to be grown symmetrically.

5.3.1.2 Action Search Space

The action search space includes:

- growing the generator with a convolution layer with filter sizes from $\{3, 7\}$ and number of filters from $\{32, 64, 128, 256, 512, 1024\}$, with padding so that the resolution does not change;
- growing the discriminator with a convolution layer with filter sizes from $\{3, 7\}$ and number of filters from $\{32, 64, 128, 256, 512, 1024\}$, with padding so that the resolution does not change;
- growing both generator and discriminator by adding a fade-in block (Karras et al., 2018) to double the resolution.

The fade-in block, introduced in ProgGAN (Karras et al., 2018), can smoothly transit the network from lower resolution to higher resolution and avoid sudden dramatic changes in well-trained low-resolution networks. Sudden dramatic changes without the fade-in block may cause the training to diverge. More specifically, to transit from lower resolution input/output to higher resolution input/output, the fade-in block uses a weighted sum of both the lower resolution path and the higher resolution path. During training, we gradually reduce the weight on the lower resolution path from 1 to 0 and increases the weight of the higher resolution path from 0 to 1, so that it transits softly.

This search space allows us to grow either or both of the generator and discriminator at each step. We thus can explore various architecture combinations of the generator and discriminator without symmetric constraint while ensuring the image resolution is consistent between both. An example of such asymmetric GAN is shown in Fig. 5.1(bottom).

5.3.2 Search Algorithm

In the growing steps discussed above, the number of potential architecture candidates will grow exponentially with respect to the search depth. Due to resource limits, it is infeasible to evaluate all of the candidates. We thus use random sampling and greedy pruning to reduce the number of candidates.

More specifically, at each growing step, we reduce the number of parent candidates by greedy pruning and reduce the number of actions by random sampling. In particular, we keep the top K parent candidates at each step and expand to child candidates only with these parents, similar to beam search. With the T actions, we randomly sample actions with a probability p. The detailed algorithm is in Algorithm 1.

This pruning algorithm reduces the number of candidates from exponential to linear, with respect to search depth. Larger p or K leads to a better exploration of the search space but also greater computational cost. We will further discuss the computational efficiency in Sec. 5.5.4.

5.4 Experimental Results

We evaluate DGGAN against manually designed ProgGAN and other recent GAN models on CIFAR-10 and LSUN. We use the most popular PyTorch implementation (Facebook) of ProgGAN

Algorithm 1: Top-K Greedy Pruning Algorithm

```
$\mathcal{A}$ = {actions};
$\mathcal{P}$ = {initial candidates};
Train and evaluate each candidate in $\mathcal{P}$;
while not reaching maximum number of layers do
    $\mathcal{C}$ = {};
    for each $(P_i, A_j) \in \mathcal{P} \times \mathcal{A}$ do
        Add $P_i + A_j$ into $\mathcal{C}$ with probability $p$;
    end
    Train and evaluate each candidate in $\mathcal{C}$;
    $\mathcal{C}'$ = keep top-$K$ candidates of $\mathcal{C}'$;
    $\mathcal{P}$ = $\mathcal{C}$;
end
```

to obtain comprehensive ProgGAN results and to implement our DGGAN. Ablative analysis is performed on CIFAR-10.

5.4.1 Datasets and Evaluation Metrics

CIFAR-10 (Krizhevsky and Hinton, 2009) contains 50k 32×32 training images in 10 categories. It is a small but effective testbed for GANs, including recent automatic GAN works (Gong et al., 2019; Wang and Huan, 2019). LSUN (Yu et al., 2015) has over a million 256×256 bedroom images for training.

We use Frechet Inception Distance (FID) (Heusel et al., 2017) as the main evaluation metric as well as the feedback criterion in the searching process. FID is a widely-used evaluation metric for the image generation task. It measures the distance between the Inception-net activations of generated images and real images. It is computed as following: $FID^2 = ||\mathbf{m}_r - \mathbf{m}_g||_2^2 + Tr(\mathbf{C}_r + \mathbf{C}_g - 2(\mathbf{C}_r\mathbf{C}_g)^{1/2})$, where $(\mathbf{m}_r, \mathbf{C}_r)$, $(\mathbf{m}_g, \mathbf{C}_g)$ denote the mean and covariance of the real and generated image feature distributions respectively. Both are modeled as multi-dimensional Gaussians. As shown in empirical studies (Xu et al., 2018; Heusel et al., 2017), FID evaluates the generation quality more effectively compared to other metrics such as Inception Score.

Following some prior works, we also report Inception Score (Salimans et al., 2016) on CIFAR-10 and the Sliced Wasserstein Distance (SWD) (Karras et al., 2018) on LSUN. The Inception Score $\mathrm{I}S = \exp(\mathbb{E}_{\mathbf{x}\sim p_g}[D_{KL}(p(y|x)||p(y))])$, however, only uses the synthetic examples and does not compare them to real-world samples. SWD is an approximation of Wasserstein distance. It computes 7×7 patch similarity at multiple scales, capturing large-scale image structures at the coarser scales and pixel-level attributes such as sharpness at the finer scales. The average across all scales indicates the overall similarity between the synthetic and real images.

Table 5.1: Quantitative evaluation on CIFAR-10, compared with ProgGAN at each resolution.

Resolution	8×8	16×16	32×32
ProgGAN	4.02	11.68	18.33
Ours	1.96	6.40	12.10
Improvement	51%	45%	34%

Table 5.2: Quantitative evaluation on CIFAR-10, compared to prior state-of-the-art works. Top: manually-designed GANs. Bottom: automatic GANs. Lower FID and higher Inception Score indicate better generation quality.

Model	FID	Inception Score
CR-GAN (Zhang et al., 2020)	14.56	-
Improving MMD-GAN (Wang et al., 2019)	16.21	8.29
DistGAN (Tran et al., 2018)	17.61	-
ProgGAN (Karras et al., 2018)	18.33[2]	8.80 ± 0.05
WGAN-GP (Gulrajani et al., 2017)	29.3	7.86 ± 0.07
DCGAN (Radford et al., 2016)	37.7	6.64 ± 0.14
AutoGAN (Gong et al., 2019)	12.42	8.55 ± 0.10
AGAN (Wang and Huan, 2019)	23.80	8.82 ± 0.09
Ours	12.10	8.64 ± 0.06

5.4.2 CIFAR-10

5.4.2.1 Implementation Details

We train initial candidates for 100k iterations and train each new candidate with 100k iterations after weight inheritance. We gradually increase the resolution from $d_0 = 8$ to 32. After reaching the final resolution, following (Karras et al., 2018), we further train the fixed architecture longer to achieve convergence. We follow the same training schedule as ProgGAN.

[2]All numbers with prior works are from original papers except this FID score is not reported in the original paper. It is obtained by the PyTorch ProgGAN implementation (Facebook).

5.4.2.2 Comparison with ProgGAN

As in Table 5.1, we improve the final resolution FID from 18.33 to 12.10 and also improve the FID on other resolutions significantly as shown in Table 5.1, by 51%, 45%, and 34% respectively. FID at each resolution is computed with resized images, thus not comparable.

5.4.2.3 Comparison with Automatic GANs

The bottom rows in Table 5.3 are prior works on automatic GANs, such as AGAN (Wang and Huan, 2019) and AutoGAN (Gong et al., 2019). Our DGGAN outperforms both and achieves the new state-of-the-art on FID.

5.4.2.4 Comparison with State-of-the-art

The prior state-of-the-art methods in the top rows in Table 5.2 are manually-designed GANs that may have improved loss functions (Wang et al., 2019; Tran et al., 2018) or regularization (Zhang et al., 2020). These sophisticated losses and regularization techniques show superiority over the basic WGAN-GP loss. Even though that our models are trained with this basic WGAN-GP loss, our method still outperforms all of these methods on FID.

Note that our contribution on the architecture is orthogonal to these sophisticated losses and regularization techniques. Utilizing them, DGGAN may achieve even better performance.

5.4.2.5 Inception Score and Visualization

Even though it is shown (Xu et al., 2018; Heusel et al., 2017) that Inception Score fails to capture the distance between generated images and real images, we still evaluate with Inception Score for a thorough comparison with prior works on CIFAR-10. We show that with the large variety on FID, the Inception Scores are similar across well-performing GANs. We show randomly selected generated examples from our DGGAN, ProgGAN, and AutoGAN as in Fig. 5.3.

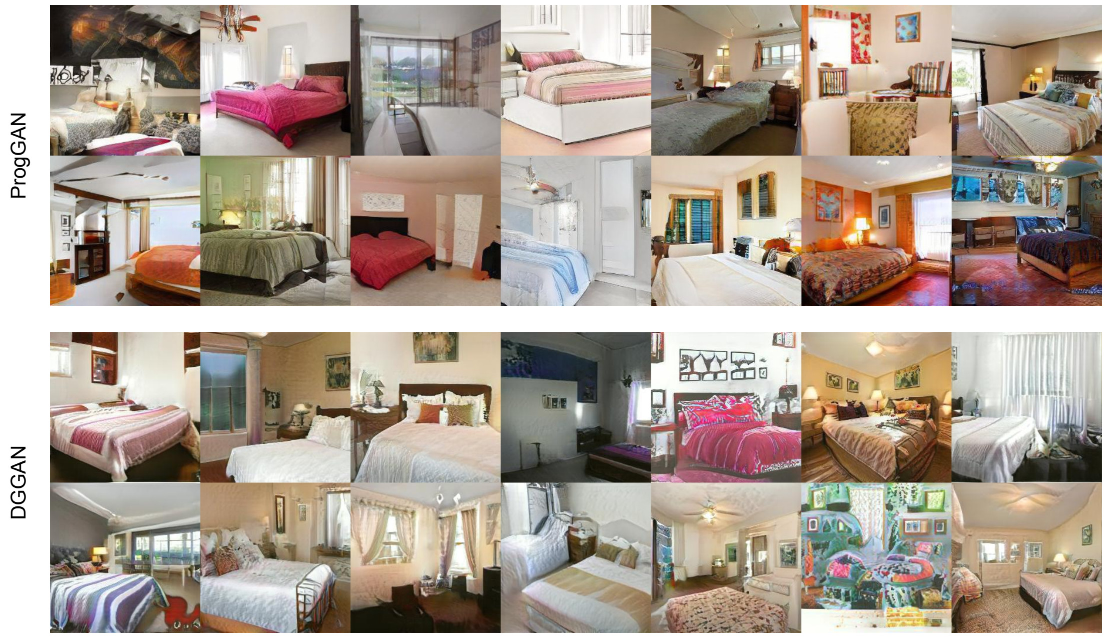


Figure 5.2: Examples of generated LSUN Images by ProgGAN and our DGGAN at 256×256 resolution. **Top**: ProgGAN, obtained from their paper. It is not stated whether it is randomly generated; **Bottom**: Our DGGAN, randomly generated.

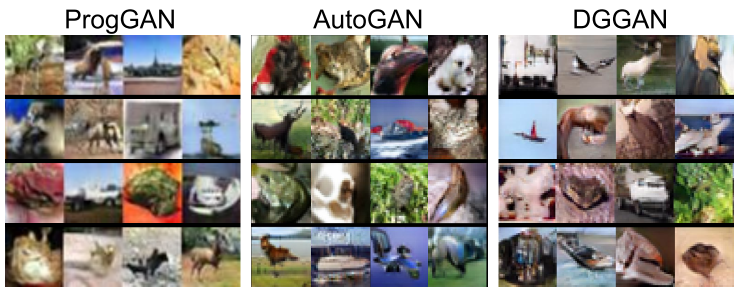


Figure 5.3: Examples of generated CIFAR-10 Images by ProgGAN, AutoGAN and our DGGAN, from left to right. ProgGAN and our DGGAN are randomly generated. AutoGAN images are obtained from their paper, which is also randomly generated.

Table 5.3: Quantitative evaluation on LSUN, compared with ProgGAN at each scale.

Resolution	8×8	16×16	32×32	64×64	128×128	256×256
ProgGAN	29.07	23.75	27.9	19.67	29.35	10.76
Ours	24.29	21.49	19.01	8.25	13.29	8.22
Improvement	16%	10%	32%	58%	46%	24%

Table 5.4: Quantitative evaluation on LSUN, compared to prior state-of-the-art works. SWD averages over multiple feature scales from 256 to 16. Lower FID and smaller SWD indicate better generation quality. All methods except COCO-GAN are non part-based GANs that learns to generate the whole images. COCO-GAN learns to generate parts of images instead.

Model	Resolution	FID	SWD $\times 10^3$					
			256	128	64	32	16	Avg
DCGAN	64×64	70.4	-	-	-	-	-	-
WGAN-GP	64×64	20.5	-	-	-	-	-	-
Improving MMD-GAN	64×64	12.52	-	-	-	-	-	-
TTUR	64×64	9.5	-	-	-	-	-	-
Ours	64×64	**8.25**	-	-	-	-	-	-
ProgGAN (reported)	256×256	8.34	2.72	2.45	2.34	2.90	9.08	3.90
ProgGAN (Pytorch)	256×256	10.76	3.74	3.78	3.53	4.04	1.58	3.33
Ours	256×256	**8.22**	3.01	2.15	3.03	4.56	1.40	**2.83**
COCO-GAN (Part-based method)	256×256	**5.99**	-	-	-	-	-	-

5.4.3 LSUN

We follow the same strategy as in CIFAR-10 except that the resolution is 256.

5.4.3.1 Comparison with ProgGAN

We demonstrate significant improvement against ProgGAN baseline at each resolution—by 16%, 10%, 32%, 58%, 46%, 24% respectively, in Table 5.3.

5.4.3.2 Comparison with Automatic GANs

We are the first automatic GAN work demonstrated with this high-resolution of 256×256. There is no prior automatic GAN work that uses a higher resolution than 48×48, potentially because of the instability of GANs and further the difficulties of searching architectures for unstable GANs, as discussed in (Gong et al., 2019).

5.4.3.3 Comparison with State-of-the-art

Recent works (Wang et al., 2019; Heusel et al., 2017) on manually-designed GANs innovate on loss function and training strategy. These methods show superiority over the basic WGAN-GP loss (Gulrajani et al., 2017; Karras et al., 2018) With the disadvantage of loss, our method achieves the best on both resolutions among these methods as in Table 5.4.

Different from the above models that generates the full images, COCO-GAN (Lin et al., 2019) is a part-based method. It is trained with image parts instead of the full image, conditioned on the spatial coordinates. It achieves the new state-of-the-art, better than other non-part based methods including ours. Combining with part-based method can be our future work with dynamic growing.

5.4.3.4 Visualization

We show examples of generated bedroom images in Fig. 5.2. Our example images are randomly generated. We see that our generated images are diverse, sharp and have good layouts most of the time. Compared to ProgGAN, it also shows better details such as wall decorations and furniture other than beds. However, we also see failure cases such as non-straight lines and occasionally completely-failed images.

5.5 Analysis

We conduct a thorough analysis of thousands of models produced by our search. Several practical conclusions and discussions are provided. More analysis can be found in the supplementary material.

5.5.1 How does the capacity of the generator (G) and the discriminator (D) affect training?

We use the number of trainable parameters as an approximated indicator of the network capacity. More specifically, we use the number of parameters in G over the number of parameters in D, noted as G2D #parameters ratio, as the G-D capacity indicator and use FID normalized by the per-scale ProgGAN baseline as the generation quality indicator.

We show how the candidates perform with the G-D ratio in Fig. 5.4(left). The X-axis is on a log scale. The extremely large normalized FIDs are the cases where the training diverges. We observe that diverged cases are rare with our algorithm.

From the zoomed-in figure in Fig. 5.4(right), we observe that *G and D do not have to be symmetric to gain good performance*. To have a better performance than the baseline, i.e. normalized FID<1, the G2D ratio can be in a wide range $[1/64, 64]$. This observation suggests that the long-existing strategy of designing symmetric G and D may miss many potentially good architecture candidates. Our DGGAN fills in this gap and explores those candidates.

To further analyze the growing process, we visualize the best performing GAN's growing route as red arrows in the plot. The flexibility to have asymmetric G-D architecture turns out to be essential for getting the best growing route.

5.5.2 How does each action affect training?

We investigate how each action, i.e. how to grow layers, affects training. We compute the improvement of normalized FID over parent architectures after each action to explore how each action affects training, as in Fig. 5.5. The normalized FID improvement is calculated by normalized FID of a child candidate over normalized FID of its parent candidate.

We compute the percentage of candidates with positive improvement after each action, and the mean and variance of the normalized FID improvement of all child candidates generated by

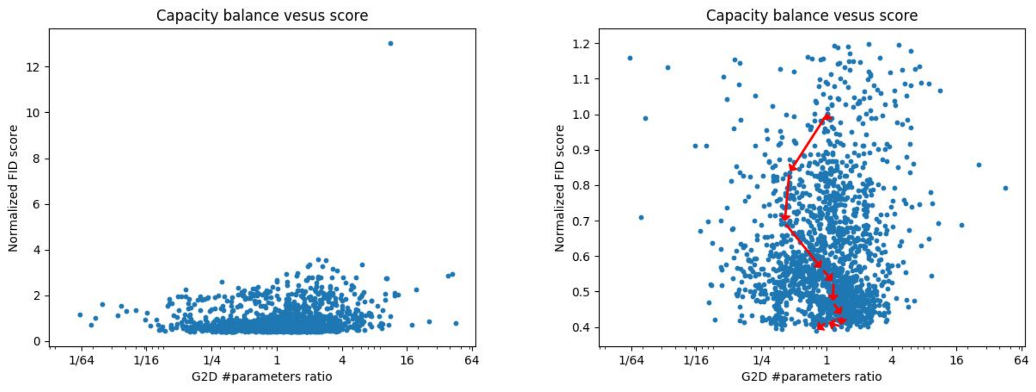


Figure 5.4: G-to-D number of parameters ratio and normalized FID for candidates on CIFAR-10. **Left**: all the candidates. **Right**: zoom-in to candidates with normalized FID smaller than 1.2. The red arrows show the growing process of the best performing GAN.

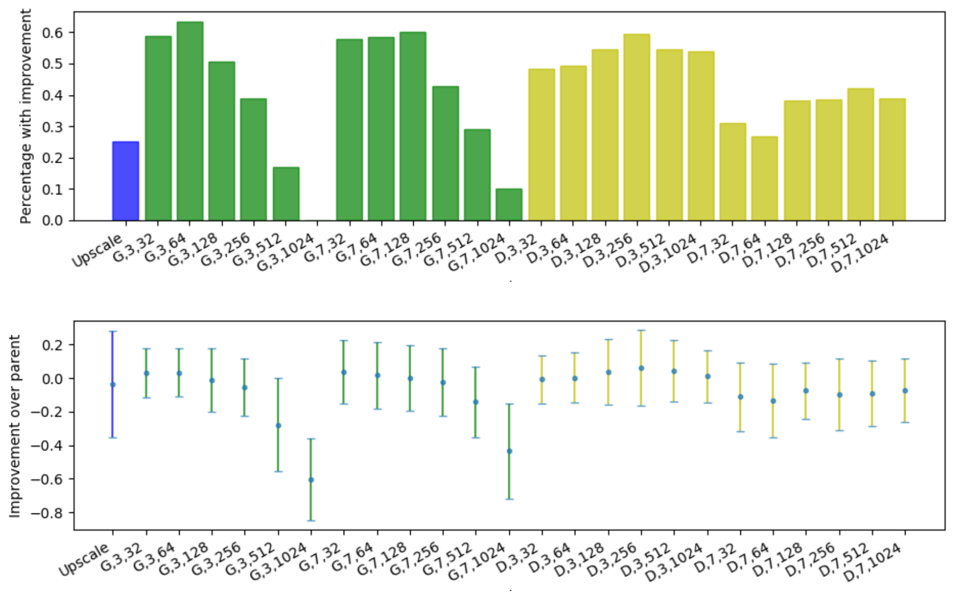


Figure 5.5: How each action affects training, i.e. how much is the improvement of normalized FID over parent. Actions include "Upscale", which means increasing the resolution of both G and D by 2, "G, a, b", which means growing a layer in G with a as filter size and b as number of filters, "D, a, b", which means growing a layer in D with a as filter size and b as number of filters. **Top**: Percentage of candidates with positive improvement over parent after each action. **Bottom**: Average and standard deviation of the improvement for each action.

performing an action as shown in Fig. 5.5.

We observe several interesting tendencies. In discriminators, larger filter size performs much

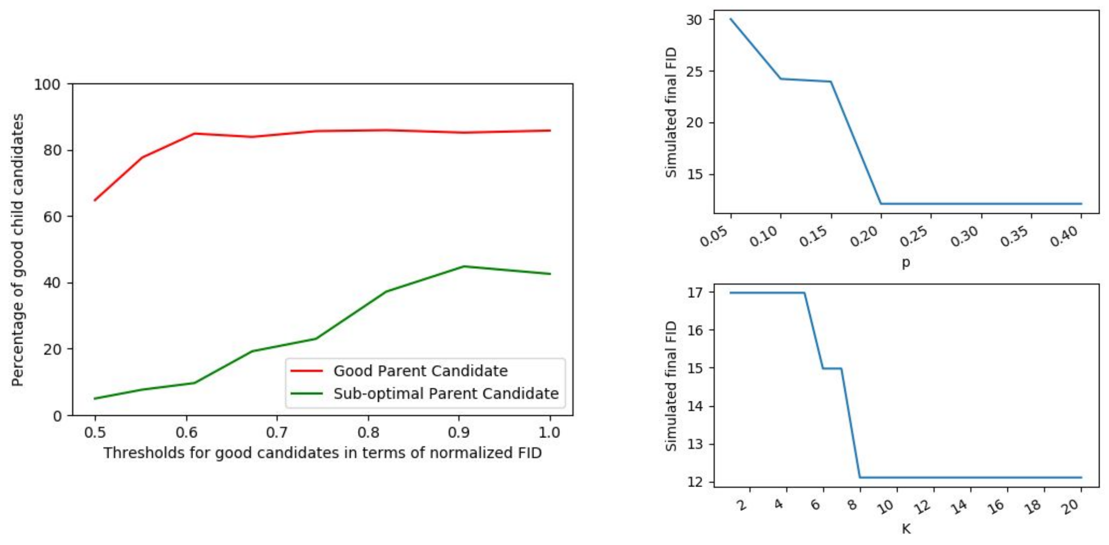


Figure 5.6: **Left**: How likely good/sub-optimal parent candidates result in good child candidates with different normalized FID thresholds. The good candidates given a threshold are defined as candidates with normalized FID larger than the threshold and the sub-optimal ones are lower than the threshold. Each threshold holds for both parent candidates and child candidates. **Right** Simulation with different hyper-parameters p and K.

worse than smaller filter size while different filter sizes perform similarly in generators. It suggests that *when designing a discriminator, a small filter size should be adopted; when designing a generator, different filter sizes are worth to explore*.

We also observe that in both discriminator and generator, more filters do not always lead to better performance. *Generator is especially more sensitive to over-complicated convolution layers*.

5.5.3 How does greedy pruning affect searching?

Our greedy pruning method prunes most of the sub-optimal parent candidates and only expand child candidates from the top-performing parent candidates. It accelerates the search significantly, from exponential to linear. However, this greedy pruning method bares the risk to miss a good search path: a sub-optimal parent candidate that is pruned may be able to develop well-performing children models in the future.

To quantify such risks, we show how likely the sub-optimal parent candidates could generate good children. We compute the ratio of good children over all children for each good or sub-optimal parent candidate using 2K+ models we trained during an extensive search process. The good candidates given a threshold are defined as candidates with normalized FID to be above the threshold and the sub-optimal ones are below the threshold. Note that the sub-optimal parent candidates used here are not too bad because the worst candidates have been pruned out during the search.

We vary the normalized FID threshold in the range $[0.5, 1]$ and show how likely good/sub-optimal parent candidates result in good child candidates in Fig. 5.6(left). It shows that good parent candidates are much more likely to have good child candidates, regardless of the threshold to decide good candidates. This indicates that *pruning the sub-optimal parent candidates does not introduce much risk of missing good growing routes in practice*.

5.5.4 Discussion on Efficiency

We use hyper-parameters K (top-K) and p (random sampling ratio) to budget the computation cost. Larger K or p leads to a better exploration of the search space but also greater cost. In our experiments, we choose $K = 20$ and $p = 0.4$ to maximally use our computation budget to explore a larger space and more candidates to analyze the behavior of GAN's dynamic growing. The resulting computational cost is 580 GPU days for 2k+ CIFAR-10 models and 1720 GPU days for 1k+ LSUN models.

For the optimal time cost to achieve good performance, K can be as small as 8 to achieve the same performance, and p can be as small as 0.2, as shown in our simulation in Fig. 5.6(right). We simulate to use smaller K or p in our algorithm: if a simulated candidate is not among the candidates that we have reached during our real search, we ignore it. Note that this simulation explores fewer candidates than a real run with different K and p, which means that the actual computation cost can be less to achieve the same performance.

Note that our main contribution is to propose the dynamic growing method that bridges the gap between high-resolution GAN and architecture search. We aim for high generation quality instead of good search efficiency. Upon our simple strategy such as random sampling and pruning, further work such as using learning-based methods (Liu et al., 2019b; Pham et al., 2018) can be used to improve efficiency.

5.6 Conclusions

In this chapter, we propose the Dynamically Grown GAN, growing the network architecture and optimizing its parameters together automatically. In the growing process, our method determines which component, generator or discriminator or both, to grow and how to grow. The new architecture is then trained with weight inheritance from the parent architecture. Experimental results on two datasets demonstrate competitive performance in image generation. In addition, we are the first automatic GAN method that works with high resolution images. With a thorough analysis, we provide several constructive insights on GAN architecture designs, including network balance and layer settings.

5.7 Appendix

5.7.1 Additional Analysis

We provide additional analysis of our 1k+ models on LSUN (Yu et al., 2015).

5.7.1.1 How does the capacity of the generator (G) and the discriminator (D) affect training?

We use the number of parameters in G over the number of parameters in D, noted as G2D #parameters ratio, as the G-D capacity indicator and use FID normalized by the per-scale ProgGAN baseline as the generation quality indicator.

We show how the candidates perform with the G-D ratio in Fig. 5.7(left). We observe that diverged cases are also rare with our algorithm in LSUN. From Fig. 5.7(right), to have a better performance than the baseline, the G2D ratio can be in a range of $[1/4, 4]$.

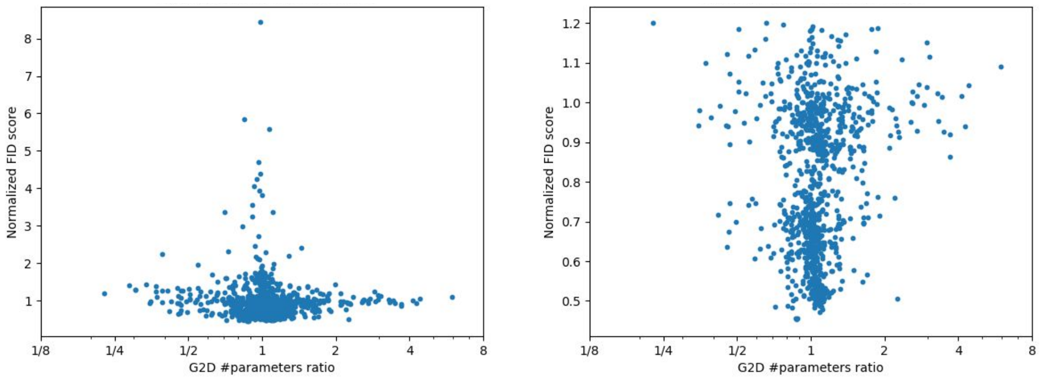


Figure 5.7: **Left**: G-to-D number of parameters ratio and normalized FID for all the candidates in LSUN. **Right**: zoom-in to candidates with normalized FID smaller than 1.2.

5.7.1.2 How does each action affect training?

We compute the percentage of candidates with positive improvement after each action, and the mean and variance of the normalized FID improvement of all child candidates generated by performing an action as shown in Fig. 5.8.

Similar with our analysis to CIFAR-10, we observe that: In discriminators, smaller filter size performs better than larger filter size while different filter sizes perform similarly in generators. We also see that in both discriminator and generator, more filters do not always have better performance.

5.7.1.3 How does greedy pruning affect searching?

In Fig. 5.9(left), it shows that good parent candidates are much more likely to have good child candidates, regardless of the threshold to decide good candidates. This result also supports the effective of our pruning strategy—that pruning the sub-optimal parent candidates does not introduce much risk of missing good growing routes in practice.

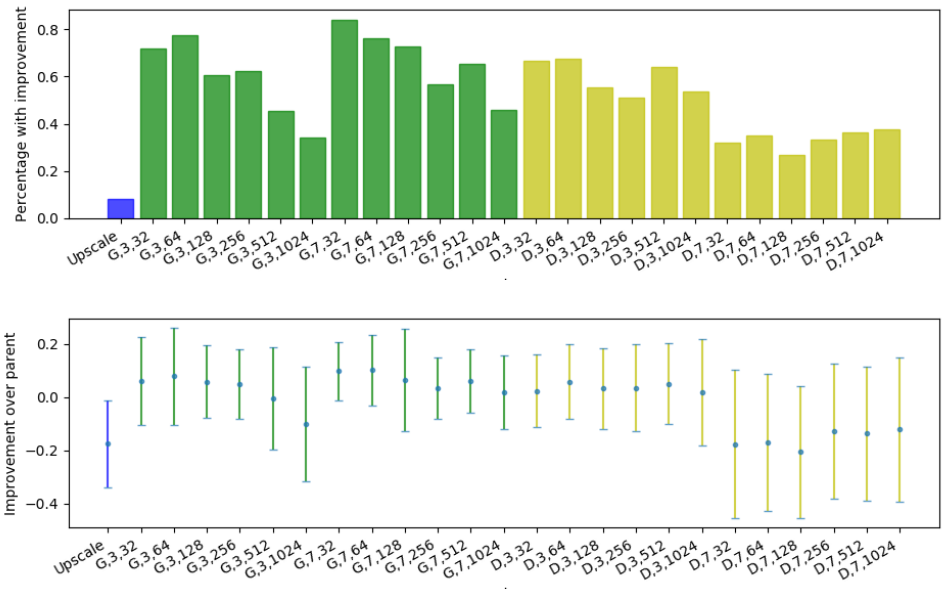


Figure 5.8: How each action affects training, i.e. how much is the improvement of normalized FID over parent. **Top**: Percentage of candidates with positive improvement over parent after each action. **Bottom**: Average and standard deviation of the improvement for each action.

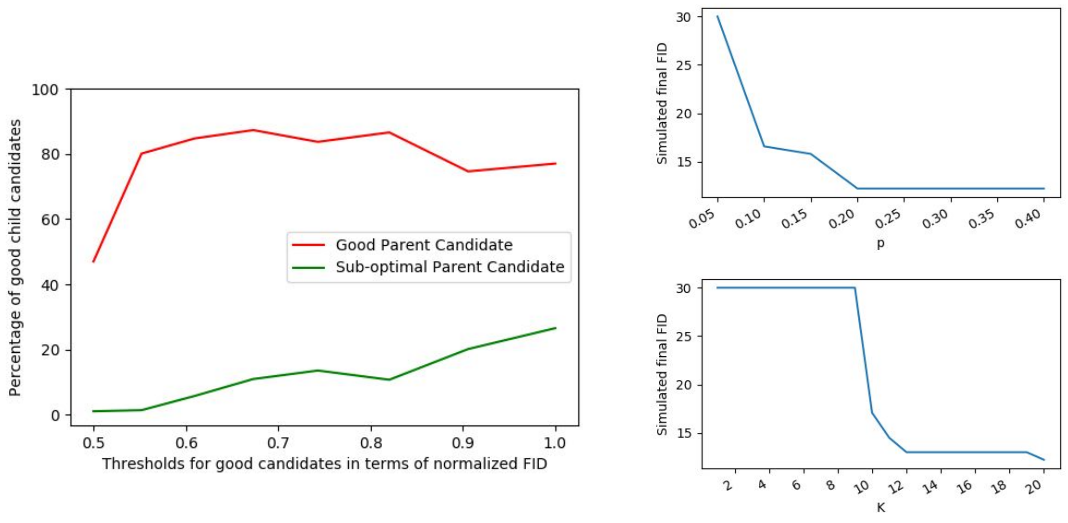


Figure 5.9: **Left**: How likely good/sub-optimal parent candidates result in good child candidates with different normalized FID thresholds. **Right:** Simulation with different hyper-parameters p and K.

5.7.1.4 Discussion on Efficiency

We again simulate to use smaller K or p in our algorithm. Note that this simulation explores fewer candidates than a real run with different K and p, which means that the actual computation cost can be less to achieve the same performance. For the optimal time cost to achieve a good performance, K can be as small as 12, and p can be as small as 0.2, as shown in our simulation in Fig. 5.9(right).

CHAPTER 6

Limitations and Future Work

Towards the ultimate goal of full automation to design a DNN, we have made efforts in four components in the automated designs of DNNs in this thesis. We propose to automate the dynamic inference with a new flexible framework. To automate the loss design with various tasks, we propose a unified framework that reduces human efforts. We also developed new solutions to synthetic data generation for small-data object detection and architecture search for generative adversarial networks. There are however some potential limitations that may limit the direct application of our methods in the practical scenarios. Below are some discussions on these limitations and also potential future directions.

6.1 Integration of Multiple Automated Techniques

Integrating multiple automated techniques is also an important future direction. Research towards automated techniques for an isolated component is often insufficient. For example, pure architecture search techniques may lead to incomplete conclusions due to the lack of extensive exploration of hyper-parameter tuning and data augmentation. We have seen architectures performing significantly better with some changes on data augmentation and hyper-parameter choices. To find an truly outperforming architecture, we may have to combine automated hyper-parameter tuning, automated data augmentation together with the architecture search.

6.2 Unified Loss

In the automated loss design topic, we have developed a unified framework to generate surrogate losses for various tasks. We achieve this by re-facorizing and decomposing the evaluation process into four steps where the non-differentiable steps can be approximated by some differentiable functions: sigmoid and differentiable interpolation. Given this general framework, one limitation is that the exploration of the approximation methods is limited. We adopt the most straightforward approximations, such as the sigmoid function and IDW interpolation. But there are other sophisticated choices for the interpolation methods, and for the sampling strategy for anchors to explore.

Another limitation is that the loss design process is not fully automated. More specifically, the refactorization step needs to be automated. Despite the amount of effort is much less than manual design from scratch, it still requires some effort to analyze the given code of the decoder and the evaluator for the refactorization. Combining automatic code analysis with our framework can further reduce human efforts in the loss design.

6.3 Architecture Search

In the architecture search topic, the evaluation of GAN architectures is difficult because there is no performance metric in image generation that is proven to be aligned perfectly with human evaluation. More research may be conducted on the evaluation side to find out a better signal to give to the search process. An example could be using multiple evaluation metrics as a tuple with certain importance rankings among each metric element.

6.4 Complete Automated System

Furthermore, to have a complete automated DNN system that can be used by users that are not familiar with DNNs, current automated techniques in all three design spaces need to be combined into a full pipeline. This pipeline requires not only the engineering work that flawlessly connects all components together, but also research towards how to define and formulate the system goal.

BIBLIOGRAPHY

E. L. Allwein, R. E. Schapire, and Y. Singer. Reducing multiclass to binary: A unifying approach for margin classifiers. In *ICML*, 2000.

A. Almahairi, N. Ballas, T. Cooijmans, Y. Zheng, H. Larochelle, and A. C. Courville. Dynamic capacity networks. In *ICML*, 2016.

J. M. Alvarez and M. Salzmann. Learning the number of neurons in deep networks. In *NeurIPS*, 2016.

M. Andriluka, L. Pishchulin, P. Gehler, and B. Schiele. 2d human pose estimation: New benchmark and state of the art analysis. In *CVPR*, 2014.

M. Arjovsky, S. Chintala, and L. Bottou. Wasserstein generative adversarial networks. In *ICML*, 2017.

J. Ba, V. Mnih, and K. Kavukcuoglu. Multiple object recognition with visual attention. *arXiv preprint arXiv:1412.7755*, 2014.

R. Bardenet, M. Brendel, B. Kégl, and M. Sebag. Collaborative hyperparameter tuning. In *ICML*, 2013.

P. L. Bartlett, M. I. Jordan, and J. D. McAuliffe. Convexity, classification, and risk bounds. *Journal of the American Statistical Association*, 101(473):138–156, 2006.

S. Bazrafkan and P. Corcoran. Versatile auxiliary classifier with generative adversarial network (vac+ gan), multi class scenarios. *arXiv preprint arXiv:1806.07751*, 2018.

E. Bengio, P.-L. Bacon, J. Pineau, and D. Precup. Conditional computation in neural networks for faster models. *arXiv preprint arXiv:1511.06297*, 2015.

S. Bengio, J. Weston, and D. Grangier. Label embedding trees for large multi-class tasks. In *NeurIPS*, 2010.

Y. Bengio, N. Léonard, and A. Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In *NeurIPS*, 2011.

D. Berthelot, T. Schumm, and L. Metz. BEGAN: Boundary equilibrium generative adversarial networks. *arXiv preprint arXiv:1703.10717*, 2017.

K. Bousmalis, N. Silberman, D. Dohan, D. Erhan, and D. Krishnan. Unsupervised pixel-level domain adaptation with generative adversarial networks. In *CVPR*, 2017.

J. C. Caicedo and S. Lazebnik. Active object localization with deep reinforcement learning. In *ICCV*, 2015.

L.-C. Chen, M. Collins, Y. Zhu, G. Papandreou, B. Zoph, F. Schroff, H. Adam, and J. Shlens. Searching for efficient multi-scale architectures for dense image prediction. In *NeurIPS*, 2018.

Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, et al. Dadiannao: A machine-learning supercomputer. In *MICRO, Annual IEEE/ACM International Symposium on*, 2014.

Y. Chen, M. W. Hoffman, S. G. Colmenarejo, M. Denil, T. P. Lillicrap, M. Botvinick, and N. de Freitas. Learning to learn without gradient descent by gradient descent. In *ICML*, 2017.

J.-T. Chien, C.-J. Chou, D.-J. Chen, and H.-T. Chen. Detecting nonexistent pedestrians. In *ICCV Workshops*, 2017.

M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele. The cityscapes dataset for semantic urban scene understanding. In *CVPR*, 2016.

K. Crammer and Y. Singer. On the algorithmic implementation of multiclass kernel-based vector machines. In *ICML*, 2001.

A. Dash, J. C. B. Gamboa, S. Ahmed, M. Liwicki, and M. Z. Afzal. Tac-gan-text conditioned auxiliary classifier generative adversarial network. *arXiv preprint arXiv:1703.06412*, 2017.

J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009.

J. Deng, S. Satheesh, A. C. Berg, and F. Li. Fast and balanced: Efficient label tree learning for large scale object recognition. In *NeurIPS*, 2011.

J. Deng, J. Krause, A. C. Berg, and L. Fei-Fei. Hedging your bets: Optimizing accuracy-specificity trade-offs in large scale visual recognition. In *CVPR*, 2012.

M. Denil, L. Bazzani, H. Larochelle, and N. de Freitas. Learning where to attend with deep architectures for image tracking. *Neural computation*, 24(8):2151–2184, 2012.

L. Denoyer and P. Gallinari. Deep sequential neural network. *arXiv preprint arXiv:1410.0510*, 2014.

E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *NeurIPS*, 2014.

I. Deshpande, Z. Zhang, and A. G. Schwing. Generative modeling using the sliced wasserstein distance. In *CVPR*, 2018.

T. Domhan, J. T. Springenberg, and F. Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *AAAI*, 2015.

D. Eigen, M. Ranzato, and I. Sutskever. Learning factored representations in a deep mixture of experts. *arXiv preprint arXiv:1312.4314*, 2013.

Facebook. Pytorch progressive GAN implementation. `https://github.com/facebookresearch/pytorch_GAN_zoo`.

P. F. Felzenszwalb, R. B. Girshick, and D. McAllester. Cascade object detection with deformable part models. In *CVPR*, 2010.

S.-W. Fu, T.-W. Wang, Y. Tsao, X. Lu, and H. Kawai. End-to-end waveform utterance enhancement for direct evaluation metrics optimization by fully convolutional neural networks. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 26(9):1570–1584, 2018.

X. Gong, S. Chang, Y. Jiang, and Z. Wang. AutoGAN: Neural architecture search for generative adversarial networks. In *ICCV*, 2019.

I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *NeurIPS*, 2014.

J. Grabocka, R. Scholz, and L. Schmidt-Thieme. Learning surrogate losses. *arXiv preprint arXiv:1905.10108*, 2019.

K. Gregor, I. Danihelka, A. Graves, D. J. Rezende, and D. Wierstra. Draw: A recurrent neural network for image generation. In *ICML*. JMLR Workshop and Conference Proceedings, 2015.

U. Grenander and M. I. Miller. Representations of knowledge in complex systems. *Journal of the Royal Statistical Society: Series B (Methodological)*, 56(4):549–581, 1994.

I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. C. Courville. Improved training of Wasserstein GANs. In *NeurIPS*, 2017.

S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep learning with limited numerical precision. In *ICML*, 2015.

S. Gurumurthy, R. Kiran Sarvadevabhatla, and R. Venkatesh Babu. Deligan: Generative adversarial networks for diverse and limited data. In *CVPR*, 2017.

S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural network. In *NeurIPS*, 2015.

T. Hazan, J. Keshet, and D. A. McAllester. Direct loss minimization for structured prediction. In *NeurIPS*, 2010.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

P. Henderson and V. Ferrari. End-to-end training of object class detectors for mean average precision. In *ACCV*, 2016.

M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium. In *NeurIPS*, 2017.

G. Hinton, N. Srivastava, and K. Swersky. Lecture 6a overview of mini–batch gradient descent. *Coursera Lecture slides https://class. coursera. org/neuralnets-2012-001/lecture,[Online*, 2012.

J. Hoffman, E. Tzeng, T. Park, J.-Y. Zhu, P. Isola, K. Saenko, A. Efros, and T. Darrell. Cycada: Cycle-consistent adversarial domain adaptation. In *ICML*, 2018.

S. Hong, X. Yan, T. Huang, and H. Lee. Learning hierarchical semantic image manipulation through structured representations. In *NIPS*, 2018.

C. Huang, S. Zhai, W. Talbott, M. A. Bautista, S.-Y. Sun, C. Guestrin, and J. Susskind. Addressing the loss-metric mismatch with adaptive loss alignment. In *ICML*, 2019.

G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *CVPR*, 2017.

G. B. Huang, M. Ramesh, T. Berg, and E. Learned-Miller. Labeled faces in the wild: A database for studying face recognition in unconstrained environments. Technical Report 07-49, University of Massachusetts, Amherst, October 2007.

P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros. Image-to-image translation with conditional adversarial networks. In *CVPR*, 2017.

R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87, 1991.

J. Johnson, A. Alahi, and L. Fei-Fei. Perceptual losses for real-time style transfer and super-resolution. In *ECCV*, 2016.

M. I. Jordan and R. A. Jacobs. Hierarchical mixtures of experts and the em algorithm. *Neural computation*, 6(2):181–214, 1994.

T. Karras, T. Aila, S. Laine, and J. Lehtinen. Progressive growing of GANs for improved quality, stability, and variation. In *ICLR*, 2018.

T. Kim, M. Cha, H. Kim, J. K. Lee, and J. Kim. Learning to discover cross-domain relations with generative adversarial networks. In *ICML*, 2017.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015.

A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images, 2009.

G. B. H. E. Learned-Miller. Labeled faces in the wild: Updates and new reporting procedures. Technical Report UM-CS-2014-003, University of Massachusetts, Amherst, May 2014.

D. Lee, S. Liu, J. Gu, M.-Y. Liu, M.-H. Yang, and J. Kautz. Context-aware synthesis and placement of object instances. In *NIPS*, 2018a.

D. Lee, S. Yun, S. Choi, H. Yoo, M.-H. Yang, and S. Oh. Unsupervised holistic image generation from key local patches. In *ECCV*, 2018b.

D. Lee, T. Pfister, and M.-H. Yang. Inserting videos into videos. In *CVPR*, 2019.

D. Li, D. Chen, B. Jin, L. Shi, J. Goh, and S.-K. Ng. MAD-GAN: Multivariate anomaly detection for time series data with generative adversarial networks. In *ICANN*, 2019.

H. Li, Z. Lin, X. Shen, J. Brandt, and G. Hua. A convolutional neural network cascade for face detection. In *CVPR*, 2015.

J. Li, X. Liang, Y. Wei, T. Xu, J. Feng, and S. Yan. Perceptual generative adversarial networks for small object detection. In *CVPR*, 2017a.

K. Li and J. Malik. Learning to optimize. In *ICLR*, 2017.

Y. Li, S. Liu, J. Yang, and M.-H. Yang. Generative face completion. In *CVPR*, 2017b.

Z. Li, C. Wang, M. Han, Y. Xue, W. Wei, L.-J. Li, and F.-F. Li. Thoracic disease identification and localization with limited supervision. In *CVPR*, 2018.

X. Liang, Z. Hu, H. Zhang, C. Gan, and E. P. Xing. Recurrent topic-transition gan for visual paragraph generation. In *ICCV*, 2017.

C.-H. Lin, E. Yumer, O. Wang, E. Shechtman, and S. Lucey. St-gan: Spatial transformer generative adversarial networks for image compositing. In *CVPR*, 2018a.

C. H. Lin, C.-C. Chang, Y.-S. Chen, D.-C. Juan, W. Wei, and H.-T. Chen. Coco-gan: generation by parts via conditional coordinating. In *ICCV*, 2019.

T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft coco: Common objects in context. In *ECCV*, 2014.

T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár. Focal loss for dense object detection. *PAMI*, 2018b.

L. V. Lita, M. Rogati, and A. Lavie. Blanc: Learning evaluation metrics for mt. In *Human Language Technology and Empirical Methods in Natural Language Processing*, 2005.

B. Liu, F. Sadeghi, M. Tappen, O. Shamir, and C. Liu. Probabilistic label trees for efficient large scale image classification. In *CVPR*, 2013.

C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy. Progressive neural architecture search. In *ECCV*, 2018.

C. Liu, L.-C. Chen, F. Schroff, H. Adam, W. Hua, A. L. Yuille, and L. Fei-Fei. Auto-deeplab: Hierarchical neural architecture search for semantic image segmentation. In *CVPR*, 2019a.

H. Liu, K. Simonyan, and Y. Yang. Darts: Differentiable architecture search. In *ICLR*, 2019b.

L. Liu and J. Deng. Dynamic deep neural networks: Optimizing accuracy-efficiency trade-offs by selective execution. In *AAAI*, 2018.

L. Liu, M. Muelly, J. Deng, T. Pfister, and L.-J. Li. Generative modeling for small-data object detection. In *ICCV*, 2019c.

L. Liu, M. Wang, and J. Deng. A unified framework of surrogate loss by refactorization and interpolation. In *Under review*, 2020a.

L. Liu, Y. Zhang, J. Deng, and S. Soatto. Dynamically grown generative adversarial networks. In *Under review*, 2020b.

M.-Y. Liu, T. Breuel, and J. Kautz. Unsupervised image-to-image translation networks. In *NIPS*, 2017a.

S. Liu, Z. Zhu, N. Ye, S. Guadarrama, and K. Murphy. Improved image captioning via policy gradient optimization of spider. In *ICCV*, 2017b.

W. Liu, Y. Wen, Z. Yu, and M. Yang. Large-margin softmax loss for convolutional neural networks. In *ICML*, 2016.

I. Loshchilov and F. Hutter. Online batch selection for faster training of neural networks. *arXiv preprint arXiv:1511.06343*, 2015.

I. Loshchilov and F. Hutter. Decoupled weight decay regularization. In *ICLR*, 2019.

R. Luo, F. Tian, T. Qin, E. Chen, and T.-Y. Liu. Neural architecture optimization. In *NeurIPS*, 2018.

J. Ma and D. Yarats. Quasi-hyperbolic momentum and adam for deep learning. In *ICLR*, 2019.

T. Miyato, T. Kataoka, M. Koyama, and Y. Yoshida. Spectral normalization for generative adversarial networks. In *ICLR*, 2018.

V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

V. Mnih, N. Heess, A. Graves, et al. Recurrent models of visual attention. In *NeurIPS*, 2014.

V. N. Murthy, S. Maji, and R. Manmatha. Automatic image annotation using deep learning representations. In *ICMR*, 2015.

A. Newell, K. Yang, and J. Deng. Stacked hourglass networks for human pose estimation. In *ECCV*, 2016.

T. Nguyen and S. Sanner. Algorithms for direct 0–1 loss optimization in binary classification. In *ICML*, 2013.

T. Nguyen, T. Le, H. Vu, and D. Phung. Dual discriminator generative adversarial nets. In *NeurIPS*, 2017.

A. Odena, C. Olah, and J. Shlens. Conditional image synthesis with auxiliary classifier gans. In *ICML*, 2017.

X. Ouyang, Y. Cheng, Y. Jiang, C.-L. Li, and P. Zhou. Pedestrian-synthesis-gan: Generating pedestrian data in real scene and beyond. *arXiv preprint arXiv:1804.02047*, 2018.

L. Perez and J. Wang. The effectiveness of data augmentation in image classification using deep learning. *arXiv preprint arXiv:1712.04621*, 2017.

H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean. Efficient neural architecture search via parameter sharing. In *ICML*, 2018.

A. Radford, L. Metz, and S. Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. In *ICLR*, 2016.

H. G. Ramaswamy, S. Agarwal, and A. Tewari. Convex calibrated surrogates for low-rank loss matrices with applications to subset ranking losses. In *NeurIPS*, 2013.

H. G. Ramaswamy, B. S. Babu, S. Agarwal, and R. C. Williamson. On the consistency of output code based learning algorithms for multiclass learning problems. In *Conference on Learning Theory*, 2014.

M. Ranzato, S. Chopra, M. Auli, and W. Zaremba. Sequence level training with recurrent neural networks. In *ICLR*, 2016.

E. Real, A. Aggarwal, Y. Huang, and Q. V. Le. Regularized evolution for image classifier architecture search. In *AAAI*, 2019.

S. J. Reddi, S. Kale, and S. Kumar. On the convergence of adam and beyond. In *ICLR*, 2018.

S. Ren, K. He, R. Girshick, and J. Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In *NIPS*, 2015.

T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen. Improved techniques for training GANs. In *NeurIPS*, 2016.

S. Sankaranarayanan, Y. Balaji, A. Jain, S. Nam Lim, and R. Chellappa. Learning from synthetic data: Addressing domain shift for semantic segmentation. In *CVPR*, 2018.

C. N. d. Santos, K. Wadhawan, and B. Zhou. Learning loss functions for semi-supervised learning via discriminative adversarial networks. *arXiv preprint arXiv:1707.02198*, 2017.

N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *ICLR*, 2017.

D. Shepard. A two-dimensional interpolation function for irregularly-spaced data. In *ACM National Conference*, 1968.

A. Shrivastava, A. Gupta, and R. Girshick. Training region-based object detectors with online hard example mining. In *CVPR*, 2016.

A. Shrivastava, T. Pfister, O. Tuzel, J. Susskind, W. Wang, and R. Webb. Learning from simulated and unsupervised images through adversarial training. In *CVPR*, 2017a.

A. Shrivastava, T. Pfister, O. Tuzel, J. Susskind, W. Wang, and R. Webb. Learning from simulated and unsupervised images through adversarial training. In *CVPR*, 2017b.

J. Snoek, O. Rippel, K. Swersky, R. Kiros, N. Satish, N. Sundaram, M. Patwary, M. Prabhat, and R. Adams. Scalable bayesian optimization using deep neural networks. In *ICML*, 2015.

D. So, Q. Le, and C. Liang. The evolved transformer. In *ICML*, 2019.

Y. Song, A. Schwing, R. Urtasun, et al. Training deep neural networks via direct loss minimization. In *ICML*, 2016.

M. F. Stollenga, J. Masci, F. Gomez, and J. Schmidhuber. Deep networks with internal selective attention through feedback connections. In *NeurIPS*, 2014.

Y. Sun, X. Wang, and X. Tang. Deep convolutional network cascade for facial point detection. In *CVPR*, 2013.

R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*, volume 1.

R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *NeurIPS*, 2000.

C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *CVPR*, 2015.

M. Taylor, J. Guiver, S. Robertson, and T. Minka. Softrank: optimizing non-smooth rank metrics. In *WSDM*, 2008.

A. Tewari and P. L. Bartlett. On the consistency of multiclass classification methods. In *ICML*, 2007.

J. J. Tompson, A. Jain, Y. LeCun, and C. Bregler. Joint training of a convolutional network and a graphical model for human pose estimation. In *NeurIPS*, 2014.

N.-T. Tran, T.-A. Bui, and N.-M. Cheung. Dist-GAN: An improved GAN using distance constraints. In *ECCV*, 2018.

I. Tsochantaridis, T. Joachims, T. Hofmann, and Y. Altun. Large margin methods for structured and interdependent output variables. In *ICML*, 2005.

P. Viola and M. J. Jones. Robust real-time face detection. *International journal of computer vision*, 57(2):137–154, 2004.

H. Wang and J. Huan. AGAN: Towards automated design of generative adversarial networks. *arXiv preprint arXiv:1906.11080*, 2019.

T.-C. Wang, M.-Y. Liu, J.-Y. Zhu, A. Tao, J. Kautz, and B. Catanzaro. High-resolution image synthesis and semantic manipulation with conditional gans. In *CVPR*, 2018.

W. Wang, Y. Sun, and S. Halgamuge. Improving MMD-GAN training with repulsive loss function. In *ICLR*, 2019.

X. Wang, Y. Peng, L. Lu, Z. Lu, M. Bagheri, and R. M. Summers. Chestx-ray8: Hospital-scale chest x-ray database and benchmarks on weakly-supervised classification and localization of common thorax diseases. In *CVPR*, 2017a.

X. Wang, A. Shrivastava, and A. Gupta. A-fast-rcnn: Hard positive generation via adversary for object detection. In *CVPR*, 2017b.

W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li. Learning structured sparsity in deep neural networks. In *NeurIPS*, 2016.

C. Wong, N. Houlsby, Y. Lu, and A. Gesmundo. Transfer learning with neural automl. In *NeurIPS*, 2018.

L. Wu, F. Tian, Y. Xia, Y. Fan, T. Qin, L. Jian-Huang, and T.-Y. Liu. Learning to teach with dynamic loss functions. In *NeurIPS*, 2018.

S. Xie, A. Kirillov, R. Girshick, and K. He. Exploring randomly wired neural networks for image recognition. In *ICCV*, 2019.

Q. Xu, G. Huang, Y. Yuan, C. Guo, Y. Sun, F. Wu, and K. Weinberger. An empirical study on evaluation metrics of generative adversarial networks. *arXiv preprint arXiv:1806.07755*, 2018.

Y. Xu, R. Jia, L. Mou, G. Li, Y. Chen, Y. Lu, and Z. Jin. Improved relation classification by deep recurrent neural networks with data augmentation. In *COLING*, 2016.

D. Yang and J. Deng. Shape from shading through shape evolution. In *CVPR*, 2018.

D. Yang and J. Deng. Learning to generate synthetic 3d training data through hybrid gradient. In *CVPR*, 2020.

S. Yeung, O. Russakovsky, G. Mori, and L. Fei-Fei. End-to-end learning of action detection from frame glimpses in videos. In *CVPR*, 2016.

F. Yu, A. Seff, Y. Zhang, S. Song, T. Funkhouser, and J. Xiao. Lsun: Construction of a large-scale image dataset using deep learning with humans in the loop. *arXiv preprint arXiv:1506.03365*, 2015.

H. Zhang, T. Xu, H. Li, S. Zhang, X. Wang, X. Huang, and D. N. Metaxas. Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks. In *ICCV*, 2017.

H. Zhang, I. Goodfellow, D. Metaxas, and A. Odena. Self-attention generative adversarial networks. In *ICML*, 2019.

H. Zhang, Z. Zhang, A. Odena, and H. Lee. Consistency regularization for generative adversarial networks. In *ICLR*, 2020.

T. Zhang. Statistical behavior and consistency of classification methods based on convex risk minimization. *Annals of Statistics*, 2004.

X. Zhang, J. Zou, K. He, and J. Sun. Accelerating very deep convolutional networks for classification and detection. *PAMI*, 2016.

Y. Zhou, C. Xiong, and R. Socher. Improving end-to-end speech recognition with policy learning. In *ICASSP*, 2018.

J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *ICCV*, 2017.

B. Zoph and Q. Le. Neural architecture search with reinforcement learning. In *ICLR*, 2017.

B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. In *CVPR*, 2018.