Visual Query Optimization: Algorithms and Software Systems

by

Stephen A. Zekany

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
2023

Stephen A. Zekany

szekany@umich.edu

ORCID iD: 0000-0003-1914-9318

To my parents, Donald and Barbara Zekany, who were my first teachers.

ACKNOWLEDGEMENTS

I don't think I would have managed to complete my degree without the unwavering support of my colleagues, family, and friends.

First I am grateful for my advisors, Tom Wenisch and Ron Dreslinski, who supported my work throughout the years. I am grateful to Trevor Mudge and David Wentzloff for serving on my committee.

Second, my fellow graduate students from WenischLab (a.k.a. SanctuaryLab): Vaibhav Gogte, Hossein Golestani, Kevin Loughlin, Amir Mirhosseini, Harini Muthukrishnan, Akshitha Sriraman, Ofir Weiss, and Brendan West, and from CSE, especially Dan Rings, Nate Harada, Jon Beaumont, Tutu Ajayi, and Austin Rovinski.

My friends who have been there for me from childhood through adulthood: Colin Forgacs, Shawn Pate, Tim Kieras, Bob Geier, Meg and Jim Wallace, Amelia Jelic, Tristan Greathouse, Jakub Kraus, and Daniel Mishins.

Additionally, my friend Thomas Larsen, who was truly invaluable as a research assistant. I think he's better at coding than I am.

Lastly, my parents, Donald and Barbara, for their unwavering support throughout my schooling, my siblings, Anna, Peter, and John, as well as Mary and Tom Christensen, and Faye and Charles Zekany.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

The development of inexpensive digital cameras and availability of local and cloud data storage has lead to incredible amounts of digital video production. As of May 2019, YouTube processes more than 500 hours of video per minute, a number than has grown by 40% since 2014 [41]. Meanwhile, the amount of compute used for frontier machine learning workloads is growing even faster than Moore's Law [40].

In this thesis, I focus on the problem of visual analytics for dashboard camera video. Novel algorithms and systems which make large-scale video search tractable are highly important for autonomous vehicle development, which requires curating a library of video of the vehicle performing in real-world conditions. I first show and evaluate techniques to reconstruct, classify, and index vehicle maneuvers for dashboard camera video, and then evaluate search techniques for visual queries which are independent of these algorithms.

First, I present our work describing a novel approach for detecting vehicle maneuvers from monocular dash-cam video building by leveraging deep learning techniques to estimate frame-accurate ego-vehicle movement and techniques to search video efficiently for disjoint computer vision operations. We show a technique to process these motion estimates into a vehicle trajectory, and compare trajectory segments to the reference maneuvers. We then extend the idea of motion estimation for maneuver detection to lane changes, highway events, and acceleration. We discuss and evaluate a "min pool" approach to allow each classifier to select the closest match between several reference maneuvers, which we find outperforms a single reference maneuver classification approach.

We then build a classifier pipeline to operate the individual classifiers and prioritize detected maneuvers, and show statistical results. Then, I present work on an end-to-end software system to process human-semantic queries for dashboard camera video leveraging visual processing techniques and models. We adapt computer vision models for use in an existing distributed video processing framework by writing API functionality for object detection, semantic segmentation, depth, and optical flow. We design and demonstrate a query parser to dynamically parse, plan, and execute visual queries. This infrastructure then allows us to focus on how to do cost planning for these visual analytics queries by adapting database cost planning metrics to the use of computer vision techniques.

CHAPTER I

Introduction

1.1 Trends in Digital Video and Computer Vision

The development of digital video capture has enabled the proliferation of high-quality, inexpensive video recording devices such as camera phones and home security cameras. Simultaneously, the development of cloud computing and decreasing cost of digital storage has democratized video storage and distribution via applications such as YouTube. As of May 2019, YouTube processes more than 500 hours of video per minute, a number than has grown by 40% since 2014 [41]. This democratization of video recording and storage on a previously unimaginable scale has enabled the use of video recording in new applications. Television shows can shoot hundreds of hours of raw footage and edit it down to a single 30-minute episode. Businesses and homeowners can record months of security camera footage. Dashboard cameras are inexpensive to be widely available to most consumers. Meanwhile the field of computer vision, as a subset of machine learning, has achieved significant progress in the past decade thanks to new techniques such as deep learning, improved hardware for neural networks such as state-of-the-art GPUs, and widely available datasets. The amount of compute used for frontier machine learning workloads is growing even faster than Moore's Law [40]. Computer vision models achieving high accuracy now exist for tasks such as object detection, semantic segmentation, depth estimation, and similar problems.

While the opportunity to collect digital video has increased and many computer vision problems have achieved significant improvement in accuracy, tools to search video have not kept pace. For example, use of even a single computer vision model on real data can prove cumbersome as these models, built by domain experts for the purpose of achieving high accuracy and/or performance, are not optimized for use as an end-to-end systems to process large amounts of data. As such, other research has focused on improving relevant aspects such as data storage, retrieval, and representation (e.g. downsampling images), or slight speed/accuracy trade-offs such as image classification cascades. A second issue is that using models in tandem with each other to answer certain questions often proves to be difficult. Even as simple an issue as a model designed in a different machine learning framework (e.g. Tensorflow vs. PyTorch) can become a significant barrier since the software does not share common data structures or APIs.

1.2 Importance for Autonomous Vehicle Development

Development of autonomous vehicles demonstrates an important use case for collecting massive amounts of video following this paradigm. State-of-the-art autonomous driving systems are built using massive data sets for training and evaluating vision and control algorithms. These algorithms, with the need to be debugged just like any other program, are expected to perform both in usual circumstances as well as unusual edge cases. A salient requirement of the debugging process is the ability to replay the environmental circumstances of any situation where the control agent performs abnormally. Thus, a challenge for self-driving vehicle researchers is to manage and curate a massive video library of the vehicle performing under all circumstances, perhaps even from multiple camera angles, to capture both the environment and the agent behavior. These video libraries can rapidly grow to enormous sizes; for example, the publicly available Berkeley DeepDrive data set [52] comprises 100,000 video segments and over 1,100 hours of driving, while proprietary data sets can grow much larger.

A common task in the development workflow of autonomous systems is searching such a video archive to extract segments that meet particular criteria. Such searches might be used to find test scenarios to evaluate algorithm performance under unusual circumstances, for example, a segment where strong braking is needed to avoid a collision. Alternatively, when building a training data set for deep learning algorithms, it may be desirable to oversample video segments for situations that arise rarely in practical driving. Existing data management systems are well suited to execute fast searches and queries over relational (tabular) data, but typically cannot do so for unstructured data like video. State-of-the-art video processing systems, like Scanner [34], include optimizations like frame skipping and efficient delta-frame decoding, but still largely resort to brute-force search over frames. Developments in deep learning, such as object detection and semantic segmentation, present new opportunities for video search systems to leverage at near-human accuracy [13, 16, 25]).

1.3 Video Search for Road-View Video

In this thesis, I focus on the problem of video search for road-view video; that is, video recorded from the point-of-view of the driver or passenger of an automotive vehicle. Dashboard cameras, once subject to the limitations of hardware and storage described earlier, have become inexpensive commodity consumer devices (often retailing for around $100 USD). Novel algorithms, techniques, and systems to make large-scale video search a tractable problem are highly important for autonomous vehicle development. Video search techniques are relevant for other transportation-related video applications, for example insurance companies are interested in monitoring high-risk customers for safe driving. Fleet operators are interested in the driving performance of their employees. Municipal officers are interested in the driving behavior and traffic patterns on their local roadways.

In all these cases, it is necessary to go beyond the current paradigm of a single computer vision model to solve a problem for several reasons. First, developing a computer vision model is expensive, requiring expert design and monitoring through the course of training. Deep learning models, which are currently the highest-performing type of model for many benchmarks, require enormous amounts of training data and specialized hardware (e.g. high-end GPUs or custom ASIC hardware). Developing a new model when several off-the-shelf models could mimic the same functionality does not make sense. Second, due to limitations of computer hardware, it does not necessarily make sense to combine tasks from disparate tasks into a single model when multiple models can make full use of a GPU or CPU node. State-of-the-art object detection systems, for example, typically utilize all available hardware resources to run a single instance of the model due to the internal complexity of the neural network necessary to learn the benchmark suite as accurately as possible.

However, developing a system of models is challenging as well. Different models have different data input expectations and different outputs, for example different confidence levels about a particular classification instance. Hardware-level challenges exist as well, such as the movement of data between devices on a single system, such as just-in-time memory transfer from CPU to GPU. Building a distributed system adds additional layers of complexity regarding how to shard data across multiple machines and how to collate results. While some of this is just software engineering with obvious correct answers, much of it is a hardware/software optimization problem requiring specific design.

A final challenge is evaluating the statistical validity of such systems. Generally machine learning classifiers focus on the idea of accuracy for a particular problem set, where accuracy can be measured in multiple ways depending on the problem definition and scope. Simple right/wrong accuracy is usually not the correct measurement, as a trivial system can achieve high accuracy on a data set containing very few true positives simply by classifying every sample as false. A common measurement is to look at the difference between the false positive rate and true positive rate at different thresholds, which is a measurement of the classifier's performance independent of a single threshold. A common way to measure this is with a receiver operator characteristic curve, plotting false positive rate vs true positive rate and integrating the area under the curve. A high-performing classifier will ideally have an ROC curve with numeric area close to 1 – which is to say it has a high true positive rate at all false positive rates.

With the problem of video search, it is useful to consider the ability of a human compared to a software agent. Humans can identify objects or the motion of a vehicle, for example, and can do so quite accurately. The best object-detection systems, for example, achieve approximately human-level performance or even slightly exceed it. In most other computer vision domains, a human can still outperform a machine in terms of accuracy. However even the best human cannot review video many times faster than real-time, where the best object-detection models can exceed 200 frames per second on commodity hardware (approximately 6X faster than real-time). Ultimately it is possible to scale up either humans or machines for a large video library, but the best approach is to leverage the strengths of both. Which is to say: leave the easy choices to the machine, and allow final determinations to be made by a human. In practice, this means considering the confidence level of a particular decision, and allowing the machine to eliminate video frames that are either not applicable to the query. A statistical way to approach this is to consider the *recall* of a search – the ratio of true positives selected to all selected items. By comparing to precision (the number of true positives selected out of all items) we can find the total number of excluded frames. Computing a weighting of precision and recall in this way is formally considered an F-score, but in contrast to an F1-score, the maximum harmonic mean of precision and recall across all thresholds, we consider the number of excluded frames for particular values of recall. For example, we may find we can exclude 50% of frames at recall of 0.98, or 80% of frames with a recall of 0.90. This quantitative measurement, we argue, is particularly important for this problem where low probability events can be completely excluded, but edge cases still need to be examined by a human.

1.4 Thesis Outline

In Chapter 2, we present a novel approach for detecting vehicle maneuvers from monocular dash-cam video building upon a deep learning visual odometry model to estimate frame-accurate ego-vehicle movement. Our system does not rely on GPS, accelerometry, or other metadata to determine the motion of the ego-vehicle. Instead, we utilize monocular dashcam video and an existing deep learning model, DeepV2D [42], to find translational and rotational camera motion between successive frames. We process these pose estimates into a vehicle trajectory, and compare trajectory segments to the reference maneuvers. The system proceeds in two high-level phases: In the first, we apply the DeepV2D model to reconstruct high-resolution ego-vehicle trajectory from monocular front-facing dashcam videos. In the second phase, we compare the extracted trajectories to the reference maneuvers and label matching video clips. In this way, the reconstructed trajectory data acts as an index for the video library, enabling efficient search for vehicle maneuvers. We classify movement sequences against reference maneuvers using dynamic time warping and simple heuristics. We show that using deep learning visual odometry to estimate location is superior to consumer-grade high-resolution GPS for this application. We describe and implement a greedy approach to classify maneuvers and evaluate our approach on non-trivial road maneuvers, finding an overall AUROC value of 0.84.

Chapter 3 expands on the idea of motion estimation for maneuver detection to lane changes, highway events, and acceleration. We also discuss and evaluate a “min pool” approach to allow each classifier to select the closest match between several reference maneuvers, which we find outperforms a single reference maneuver classification approach. We build an overall classifier to operate the individual classifiers and prioritize detected maneuvers. When multiple maneuvers coincide in time (e.g., deceleration while turning) the overall classifier applies priority rules to resolve conflicting output from individual classifiers and produce the best labeling. In addition to classifying sequences directly, we describe how the system can be tuned to eliminate as many uninteresting frames as possible while capturing *possibly relevant* video segments (i.e., tuning a particular recall threshold), reducing workload for a human evaluator.

Chapter 4 discusses the design of an end-to-end software system to process human-semantic queries for dashboard camera video using visual processing techniques. We adapt different computer vision models for use in an existing video-processing framework by writing API functionality for object detection, semantic segmentation, depth, optical flow, in addition to the vehicle maneuver and lane change techniques discussed in Chapters 2 and 3. Additionally we built a query parser to incorporate new keywords and functionality needed. We designed and evaluated an end-to-end video query processing system which dynamically builds video processing pipelines and performs optimizations based on the latency (cost) of executing different models. We designed and demonstrated a query language which allows the user to specify search predicates. We evaluate this system with a library of dashboard camera videos and computer vision models to search for the presence, locations, and movement of other objects in the frame. Overall we found a 3.6x speedup over frame-by-frame search and 1.2x speedup for the best query optimization technique (“estimated root contribution”) compared to cost-based optimization.

CHAPTER II

Detecting Vehicle Turns from Road-View Video

2.1 Introduction

State-of-the art systems for autonomous driving are built using massive data sets for training and evaluating vision and control algorithms [3]. A key challenge for self-driving vehicle researchers is to manage and curate these massive data sets. Video libraries associated with autonomous vehicles rapidly grow to enormous sizes; for example, the publicly available Berkeley DeepDrive data set [52] comprises 100,000 video segments and over 1,100 hours of driving, while proprietary data sets can grow much larger [44].

A common task in the development workflow of autonomous systems is searching such a video archive to extract segments that meet particular criteria. Such searches might be used to find test scenarios to evaluate algorithm performance under unusual circumstances, for example, a segment where strong braking is needed to avoid a collision. Alternatively, when building a training data set for deep learning algorithms, it may be desirable to oversample video segments for situations that arise rarely in practical driving. Existing data management systems are well suited to execute fast searches and queries over relational (tabular) data, but typically cannot do so for unstructured data like video. State-of-the-art video processing systems, like Scanner [34], include optimizations like frame skipping and efficient delta-frame

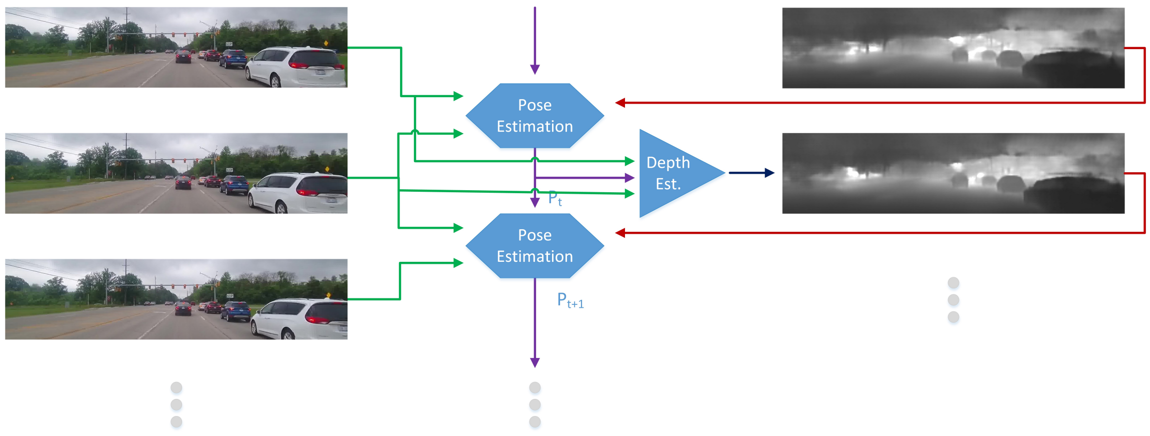


Figure 2.1: DeepV2D's pose estimation creates a depth map for each frame, and this depth map then provides the basis for the next frame's pose estimate.

decoding, but still largely resort to brute-force search over frames. Developments in deep learning, such as object detection and semantic segmentation, present new opportunities for video search systems to leverage, but do not yet attain human-level accuracy [13, 16, 25].

In this paper, we develop a processing pipeline that facilitates generating an index to search for vehicle maneuvers from dashboard camera video. The system classifies video frame sequences against a set of reference video clips, provided by the user, that demonstrate the vehicle maneuvers (e.g., right turn, U-turn, driving in reverse) to be indexed. Our system does not rely on GPS, accelerometry, or other metadata to determine the motion of the ego-vehicle. Instead, we utilize an existing deep learning model, DeepV2D [42], to find translational and rotational camera motion between successive frames. We process these pose estimates into a vehicle trajectory, and compare trajectory segments to the reference maneuvers. The system proceeds in two high-level phases: In the first, computationally more expensive phase, we apply the DeepV2D model to reconstruct high-resolution ego-vehicle trajectory from monocular front-facing dashcam videos. In the second phase, we compare the extracted trajectories to the reference maneuvers and label matching video clips. In this way, the reconstructed trajectory data acts as an index for the video library, enabling effi-

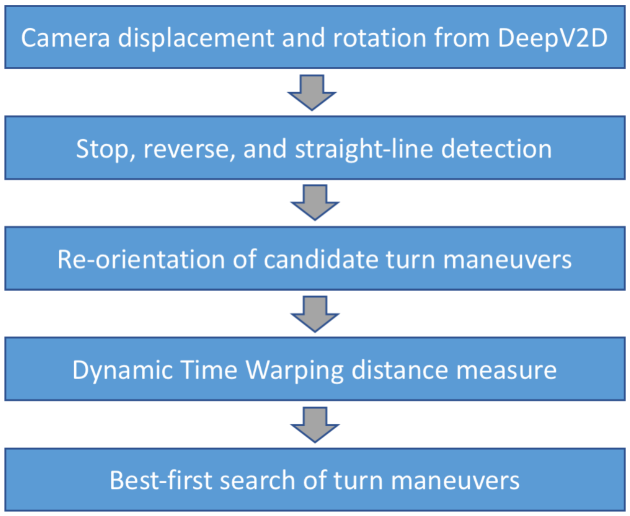

Camera displacement and rotation from DeepV2D
Stop, reverse, and straight-line detection
Re-orientation of candidate turn maneuvers
Dynamic Time Warping distance measure
Best-first search of turn maneuvers

Figure 2.2: Our software pipeline obtains output location estimates from DeepV2D, combines these into candidate trajectories, and uses a greedy approach to classify the best fit of a maneuver at any given time.

cient search for vehicle maneuvers. We evaluate our approach against human-labeled ground truth for seven common road maneuvers, and compare against a classification approach that uses measured GPS rather than reconstructed trajectories.

We make the following specific contributions:

- We describe a technique for searching for vehicle maneuvers using only dashcam video.
- We implement and evaluate our search technique.
- We demonstrate that for this search technique, ego-vehicle trajectory estimated via deep learning is superior to GPS.

2.2 Prior Work

In this section, we review two similar areas of work: aggressive driving detection and SLAM (simultaneous localization and mapping) from video.

2.2.1 Maneuver Classification

A number of studies have explored methods to evaluate *driving style* by analyzing vehicle characteristics and maneuvers. Driving style is typically a characterization of how "aggressive" a specific driver is relative to an aggregate set of reference drivers. This work is of particular interest to safety agencies and insurance companies, who wish to mitigate risk via measurement of individual or aggregate behavior. While our work does not focus on driving style or safety specifically, it is relevant because we utilize similar techniques for measuring vehicle maneuvers. Johnson et al. [20] describe a method to measure aggressiveness for turns and straight-line motion using dynamic time warping with a smartphone. Later work extended this approach with Bayesian classification [11], support vector machines, and random forest analysis [22]. Other work has focused on building a driver-centric model based on a series of maneuvers using a sensor array [9] or detection of aggressive driving using deep learning techniques [15]. In contrast to our work, which requires only dashcam video with no additional metadata, this body of work typically utilizes smartphone sensors, such as an accelerometor, GPS, magnetometer, and gyroscope, which must be on-board the vehicle [48, 26].

Other work has been done with additional sensors, which allow for more fine-grained detection of maneuvers; for example, classification of lane changes (cut-ins and overtaking) using hidden Markov models (HMMs) with data collected from radar and LIDAR [27]. Work has also been done using HMMs to predict maneuvers [17].

2.2.2 Visual Odometry

Another significant body of work from the computer vision community has focused on the idea of *visual odometry* [33]. Borrowing ideas originating from SLAM, various techniques have focused on efficient ways to extract ego-vehicle motion from video, often for the purpose of mapping a route. ORB-SLAM is a modern monocular SLAM system that outputs camera motion [31]; our work builds on the contributions of this community. Specifically, DeepV2D [42], the deep learning algorithm we use to find camera motion, has operational similarity to ORB-SLAM.

2.3 Technique

Figure 2.2 shows an overview of our vehicle maneuver classification pipeline. We first use DeepV2D to extract translational and rotational camera movement at each frame to obtain a time series of X-Y coordinates corresponding to the ego-vehicle's trajectory. We then use simple heuristics to mark stop, reverse, and straight-line motions. We then compare trajectory segments against reference maneuvers using dynamic time warping to compute a distance measure. Finally, we perform a best-first match of turn maneuvers using a greedy approach. We will describe each of these steps in detail in the following sections.

2.3.1 DeepV2D

DeepV2D [42] is a deep learning network design for estimating camera motion and depth from video. DeepV2D consists of a Stereo Module, to perform stereo reconstruction from images and a camera motion estimate, and a Motion Module, which uses depth to estimate camera motion. The motion module finds initial estimates for the sequence of frames using a pose regression network to estimate the transformation parameters between images. These initial estimates are then refined in an iterative process using a projective warping function on the differentiable transformation of the input image to produce a warped feature map. The estimated feature map is then compared to the original image feature map and the difference is used to update the pose estimate for the next frame (see Figure 2.1).

We use a model of the DeepV2D architecture trained via RMSprop [45] and the Kitti dataset [12] (with ground truth motion estimated by ORB-SLAM2 [30]) to infer the motion of the ego-vehicle in our dataset. The input to DeepV2D is a series of five video frames and the output is the estimated depth map and motion between the third and fourth frames. We therefore process each series of five frames from the recorded video library to obtain a motion track at the same framerate as the original video.

2.3.2 Dynamic Time Warping

Dynamic time warping [38] (DTW) is a measure of the similarity of two signals that may differ in duration. DTW performs a sequential matching between the signals while ignoring time differences. In effect, it "stretches" ("warps") one signal (or parts of it) to match another and computes the difference between matched values as the distance measure. (Figure 2.3 shows an example of two right turns). Whereas the original DTW algorithm is of O(n^2) computational complexity (where n is the number of matched points), implementations such as FastDTW can approximate DTW at O(n) complexity [37]. The reduction in computation time, along with comparative simplicity relative to other methods of pattern recognition, has enabled DTW to be widely and efficiently used for applications like speech recognition [21], gesture recognition [43], and detection of vehicle maneuvers [20]. We use DTW to quantify the similarity of vehicle trajectories from DeepV2D against the reference maneuvers.

DeepV2D produces a three-dimensional rotation matrix and translation vector for each sequence of five frames. We found using DeepV2D to produce camera motion estimates at the original 30 frames-per-second of our test video produced the highest-quality motion estimates. We discard the Z component and keep a time series of ego-vehicle X-Y coordinates at each frame. Because of the high temporal resolution of the video, we can reconstruct camera motion estimates at equal or better resolution than is typically available from consumer-grade GPS devices.

Maneuver	**Heuristic**
Left Turn	Forward and horizontal distance traveled must be $\geq 80\%$ of reference maneuver and same direction
Right Turn	Forward and horizontal distance traveled must be $\geq 80\%$ of reference maneuver and same direction
U-Turn	Horizontal distance traveled must be $\geq 80\%$ of reference maneuver
K-Turn	Must contain at least half-second (15 frames) of reverse motion

Table 2.1: Heuristics to accelerate classification

	AUROC Value with Heuristic	**AUROC Value without Heuristic**
Left Turn	0.90	0.92
Right Turn	0.89	0.88
U-Turn	0.75	0.50
K-Turn	1.0	0.96
Processing time (seconds)	1098	7816

Table 2.2: Classification accuracy with and without heuristics

2.3.3 Endpoint Detection and Candidate Maneuvers

Since we have no information about when a given maneuver may start or end, we perform automatic and simple endpoint detection for each DTW sequence. As the DTW matching process has no a priori information—knowing nothing about the positional track in advance—it must consider the possibility of a maneuver starting at every time step (video frame). We assume the length of a candidate maneuver can be 50% to 150% the length of the reference maneuver in 10% increments, which allows for normal variation in ego-vehicle velocity. We allow the start and end time to be any even-numbered frame in the recording, as we find this has no effect on accuracy. While the processing time to calculate the DTW matching score for an individual reference maneuver is relatively short, performing comparisons of all possible candidate maneuvers is computationally expensive. As a strategy to reduce processing time, we enforce a few simple filtering heuristics on the four types of turns processed with DTW (summarized in Table 2.1). For left and right turns, the overall motion must match 80% of the candidate maneuver's movement along the x- and y-axes. For U-turns, there must be sufficient x-axis movement, and K-turns (three point turns) require at least a half second (15 frames) of reverse motion. We find these simple heuristics slightly improve maneuver classification accuracy while reducing computational time by an order of magnitude (see Table 2.2).

In addition to the maneuvers detected using dynamic time warping, we detect three additional maneuvers using other statistical techniques: Straight-line motion, no motion (stops), and reverse motion. We cannot use DTW to detect these maneuvers, as DTW requires sufficient variation in the compared signals to have any sensitivity. Using DTW for forward and reverse motion incurs too many false positives (e.g., portions of turns are straight), and stops cannot be represented as x-y motion. To detect reverse motion maneuvers heuristically, we simply find frame sequences where the Y-translation is negative. To tolerate noise and uncertainty, we enforce the condition that 25 of 30 frames in a 1-second sequence must be negative. To find stops, we look for one-second periods where Euclidean distance traveled is below a certain threshold (0.1 meter). To find straight-line forward motion, we use linear regression on 5-second periods of motion with a threshold of $R^2 \geq 0.985$. Examples of stop and straight-line forward maneuvers are shown in Figure 2.4.

2.3.4 Reference Maneuvers and Best-First Search

Our system allows the user to define a reference maneuver for as many categories as desired. DeepV2D extraction is expensive but need only be done once; a user can re-run subsequent analysis with a new set of reference maneuvers. For our evaluation, we define four: left turn, right turn, U-turn, and K-turn (examples are shown in Figure 2.4). (Note that the K-Turn looks odd because DeepV2D incorrectly estimates rotational speed in reverse, simply because the model isn't trained on reverse motion. However the error is consistent across maneuvers, so classification is still accurate.)

The user is responsible for defining the start and end point of each reference maneuver and selecting the baseline length for candidate maneuvers (which will then be scaled as appropriate). Each maneuver is effectively its own classifier: We compute all candidate DTW distance measures and select non-overlapping maneuvers with the lowest distance measures first. To evaluate quality, we use the area under the receiver operating characteristic curve (AUROC) [49]. The ROC curve is a plot of false positive rate vs. true positive rate, and integrating this curve gives a measure of quality of the classifier without requiring a numeric threshold. While ideally we would hope to have a high true positive rate and low false positive rate, we designed our classifier to increase the true positive rate even at the expense of additional false positives, as human observers can screen out false positives much faster than search for false negatives missed by the classifier.

Data Collection Location	Ann Arbor, MI
Total Road Distance	32 miles
Maximum Speed	45 mph
Number of Left Turns	96
Number of Right Turns	76
Number of U-Turns	29
Number of K-Turns	26

Table 2.3: Descriptive statistics of our test dataset

2.4 Evaluation Methodology

To evaluate our analysis flow, we collected dashcam video for processing with DeepV2D while simultaneously collecting high-resolution GPS positional information for a baseline comparison. We chose GPS as a fair comparison for location tracking, as GPS metadata is readily available in a number of consumer devices. We then manually labeled all recorded videos for elementary maneuvers, and selected a random subset of maneuvers from the recorded video to use as reference maneuvers. Finally, we performed cross-validation on five different samples of reference maneuvers to evaluate our best-first matching algorithm.

Our dashcam videos were recorded with a Garmin Dash Cam 55, fixed to the center of the windshield of a Toyota Sequoia SUV at approximately eye-height. This model records GPS location at a rate of 1 Hz, so we also used a GoPro Hero 5 Black to obtain 18 Hz GPS metadata time-aligned with our video. The GoPro was fixed to the roof of the vehicle for better reception, and allowed a 10-minute warmup prior to recording to properly calibrate to the satellite signals.

We evaluate on a combination of commercial and residential streets in Ann Arbor, MI. Table 2.3 describes the road and driving conditions. A total of five videos were recorded, with each driving route containing approximately 50-60 maneuvers. These videos were then converted to JPEG frames for processing by DeepV2D, running on an Intel Core i7 workstation with an NVIDIA Titan XP GPU. This workstation ran Tensorflow 1.4 with CUDA 8.0 on Ubuntu 16.04. The higher-resolution GPS data extracted from the GoPro was time-aligned with the Dash Cam 55 video.

Prior to classification, the trajectory data must be reoriented. While DeepV2D motion estimation allows knowledge of the ego-vehicle's orientation at any arbitrary point, GPS does not. Orientation is important, as a maneuver cannot be detected if the positional track is rotated. Therefore, we estimate orientation from GPS data by taking the average direction vector of the half-second period prior to the start time of interest. To allow fair comparison, we use the same half-second reorientation algorithm for DeepV2D trajectories.

An example comparison of the DeepV2D positional data vs. GPS is shown in Figure 2.5. We find overall that DeepV2D data appears cleaner and more consistent than GPS data. While the GPS data is quite good, the typical GPS accuracy of approximately 3 meters [47] is simply not sufficient for tight maneuvers, as a standard U.S. highway lane is 3.7 meters wide [50] while city roads are narrower. Therefore, a maneuver like a U-turn is difficult to reconstruct accurately if the maneuver is performed on a two-lane road that is at most 8 meters wide. If the GPS device is located in the center of the vehicle, this may further reduce the width of the maneuver relative to the GPS resolution.

We selected ten percent of all maneuvers at random to be reference maneuvers. We manually selected ground truth endpoints for both GPS and DeepV2D data to best capture the given maneuver. We then used five-fold cross-validation with this randomly chosen maneuver set from each of five different videos to evaluate the system. Following the "leave one out" protocol, these reference maneuvers were not included in the overall statistical calculation.

Detected \ Actual	Left Turn	Right Turn	U-Turn	K-Turn	Stop	Reverse	Straight
Left Turn	59%	2%	23%	0%	0%	0%	0%
Right Turn	12%	55%	1%	0%	0%	0%	8%
U-Turn	6%	1%	61%	0%	0%	0%	0%
K-Turn	2%	1%	8%	100%	0%	0%	0%
Stop	0	0%	0%	0%	88%	14%	4%
Reverse	0%	0%	0%	0%	0%	82%	0%
Straight	0%	0%	0%	0%	12%	0%	88%
Forward/ Unclassifiable	20%	40%	7%	0%	0%	5%	0%

Table 2.4: Confusion matrix of Actual vs. Detected maneuvers. Actual is shown at top and each column sums to 100%.

2.5 Results

We assess the accuracy of each maneuver's classification as follows: A reference maneuver of a particular category is chosen at random. The classifier then finds the best matches to this reference maneuver throughout the duration of the video, allowing no maneuvers to overlap. These candidate maneuvers are compared to the human-annotated ground truth. If a maneuver is detected within four seconds of the ground truth annotation, it is considered correctly classified; otherwise it is considered a false positive. We then compute the ROC curve using the list of DTW distance measures of detected maneuvers. False negatives (that is, ground-truth maneuvers not detected by the classifier) are included in the ROC calculation with the maximum-detected DTW value (to penalize the classifier for failing to detect these maneuvers).

The non-weighted average ROC curves [39] for each of the four maneuvers detected with DTW in this cross-validation experiment are shown in Figure 2.6. Each individual ROC curve for a particular maneuver (shown in thin blue lines) is averaged with equal weighting at each step (shown in the bold blue line). An interval of one standard deviation above and below the mean is shown in grey.

Figure 2.6 also shows in green the ROC curve for the same classifier using 18 Hz GPS data. We find the DeepV2D motion estimates consistently outperform our GPS measurements, due to the spatial and temporal resolution advantage of DeepV2D's 30 Hz estimates relative to the 18 Hz 3-meter resolution of the GPS. As seen in Figure 2.5, the overall jitter of the GPS trajectories means these maneuvers will typically have much higher distance measures and are less well oriented than the DeepV2D trajectories.

We use a greedy approach to classify the seven elementary maneuvers. Stops, reverses, and straight motion have highest priority, as these are the simplest maneuvers and least computationally expensive to detect. The greedy classifier then chooses up to two maneuvers of different types for each period of unlabeled time in the video recording. We allow the system to choose up to two maneuvers because often one maneuver can contain another: e.g., many U-turns have an embedded component that aliases as a left turn. This left turn often has a lower DTW distance measure than the U-turn itself, meaning that if we chose only one maneuver we will miss the correct one. As our goal is to avoid false negatives even at the expense of increasing the rate of false positives, we simply evaluate a greedy classifier allowed to choose up to two maneuvers, as this does not require defining additional heuristics or priorities for each maneuver.

The two-maneuver greedy approach has an overall AUROC of 0.84; the ROC curve is shown in Figure 2.7. A breakdown of all seven individual maneuvers is provided as a confusion matrix in Table 2.4. We find that the greedy classifier approach frequently has false positives for left and right turns compared to the labeled ground truth; these arise due to curves in the road. Figure 2.8 shows an example of a road curving to the left, which is detected by the classifier as a left turn even though a human labeler marked it as forward/unremarkable.

Left turns also have a higher prevalence of aliasing as U-turns and K-turns, as each of these begins with ego-vehicle movement to the left. To better distinguish curves from turns, we must fuse our trajectory-based method with other information sources, such as road segmentation or map data; we are pursuing such multimodal data fusion in ongoing work. As previously mentioned, U-turns often alias with left turns, and K-turns overall are quite accurately detected due to the reverse motion not present in other maneuvers.

The AUROC is higher than the confusion matrix results would indicate because it accounts for the DTW distance metric (confidence) in classifying each maneuver. It is important to note that our classifier uses only one reference maneuver from each maneuver category for classification. It is therefore unrealistic to expect this approach to work perfectly, as the geometry of intersections can vary quite substantially in terms of both size and angle of intersection.

2.6 Future Work

We have demonstrated a mechanism for indexing vehicle maneuvers by trajectory, but further work remains to enable dashcam video search more broadly. First, we wish to extend our vehicle maneuver work from surface roads to highways, allowing detection of maneuvers such as merges and exits. Second, we intend to utilize other modern deep learning techniques to enable new video search capabilities. For example, use of depth estimation to allow detection of lane changes, and object detection and geometric semantics to discover interactions with other vehicles and pedestrians. Finally, we are interested in optimizing the computational efficiency of this end-to-end system and exploring new ways to accelerate it.

2.7 Conclusions

We have designed and implemented a system that determines vehicle trajectory state in terms of seven common road maneuvers. Movement is derived via a deep learning model from monocular dashcam video, which produces translational and rotational motion for each frame. These instantaneous movement vectors are processed into candidate maneuvers. Following detection of stops, reverses, and straight-line motion segments, we detect turns via dynamic time warping, computing a distance measure for each maneuver. We evaluate the best choice for each possible maneuver category using a greedy classifier on the distance measures. We test our greedy classifier using cross-validation and find it has an overall AUROC value of 0.84. Finally, we utilize our turn classifiers on high-resolution GPS data collected in parallel with the dashcam video and find that that DeepV2D's estimated motion allows for more accurate classification of vehicle motion.

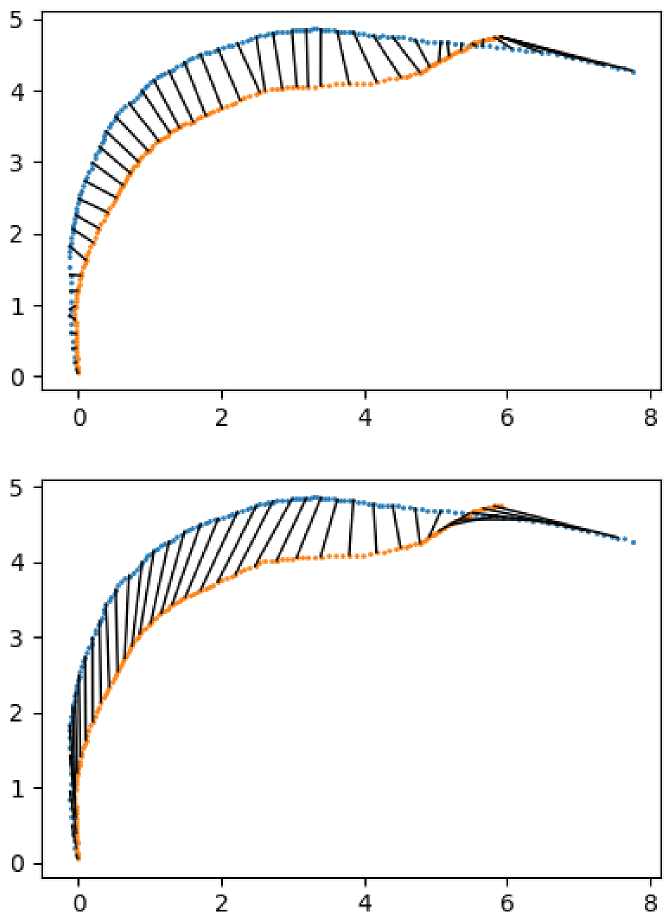


Figure 2.3: An example of two right turns (in orange and blue) collected from our experiments. The location tracking is produced by DeepV2D and the maneuvers are automatically reoriented based on preceding motion. Rather than comparing maneuvers using Euclidean distance between discrete locations at a given time (top), we use Dynamic Time Warping (bottom), visualized here with lines connecting a sample of the match points.

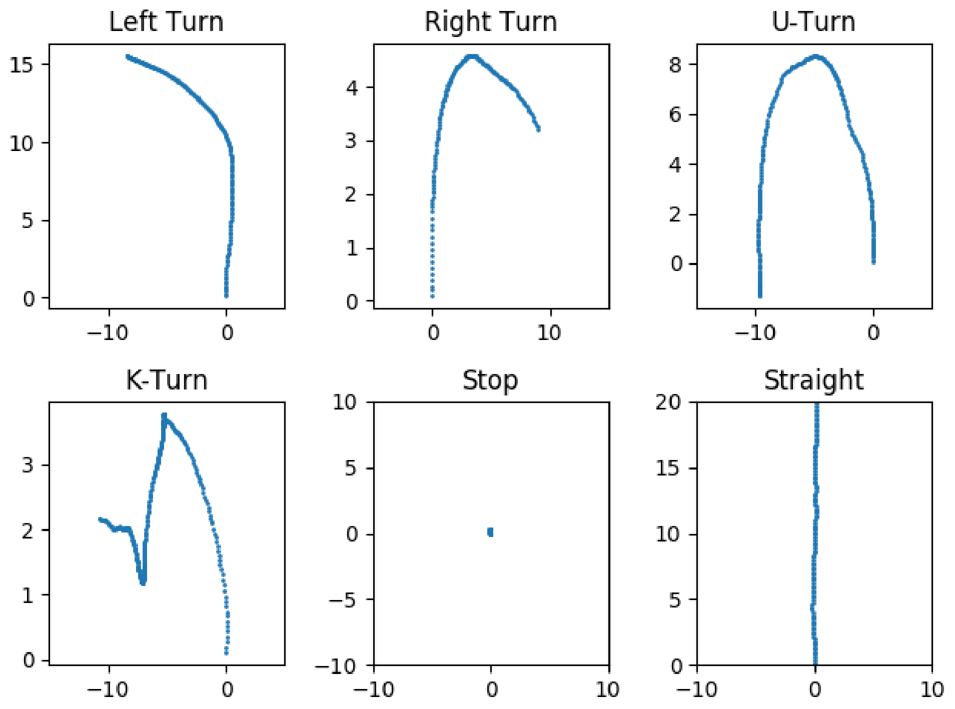


Figure 2.4: Sample vehicle maneuver trajectories using location estimates from DeepV2D. X and Y scales are nominally in meters, though DeepV2D overestimates distance in some cases.

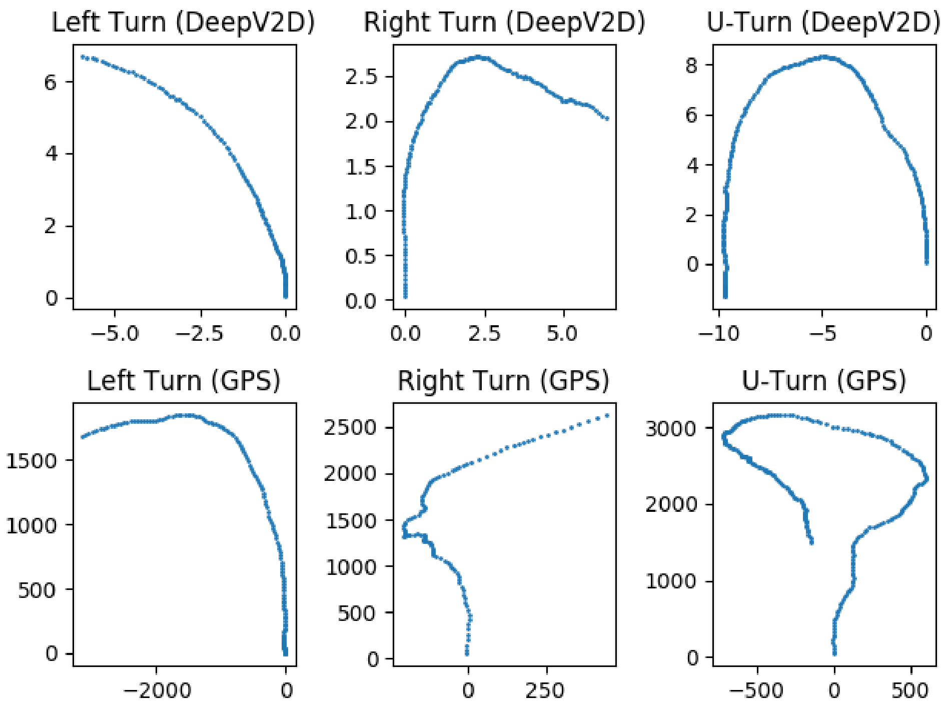


Figure 2.5: Vehicle trajectories with motion estimated by DeepV2D compared to concurrent GPS measurements. Whereas neither technique perfectly matches the geometry of the maneuver, the DeepV2D data provides a more consistent match. DeepV2D scale is nominally meters, GPS scale is ten-millionths of degrees of latitude/longitude.

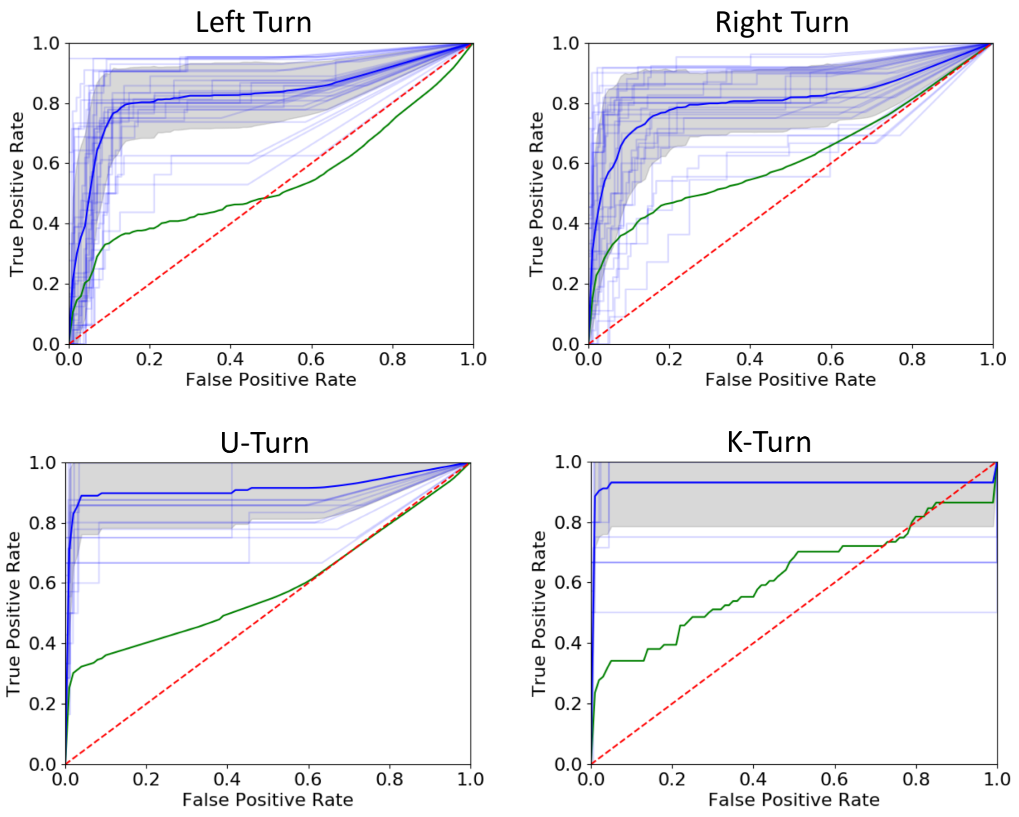


Figure 2.6: Non-weighted average ROC of cross-validation DTW classification for DeepV2D motion estimates (blue) and GPS (green).

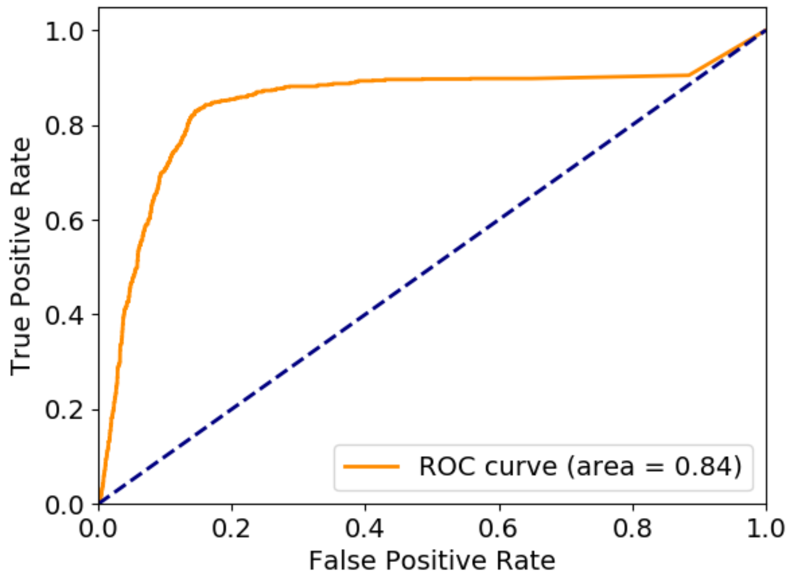


Figure 2.7: Overall Receiver Operating Characteristic of greedy classifier.

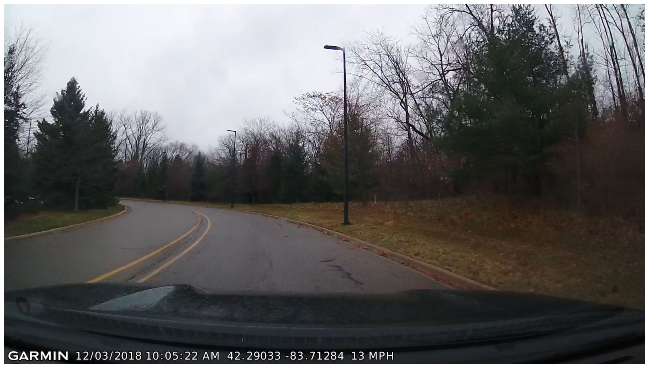


Figure 2.8: Example of a road curve to the left, which aliases as a left turn.

CHAPTER III

Vehicle Maneuver Classification from Road-View Video

3.1 Introduction

In Chapter II, I describe my work determining and classifying vehicle turns and simple maneuvers. From experience working with video and finding, by hand, the types of maneuvers that may be detectable from dashcam video, I developed ideas for new maneuvers we could detect from dashboard camera video:

- Could we use these techniques to perform road-surface analysis? E.g. could we detect a gravel road, or pothole, or speed bumps?
- How could we detect maneuvers that don't depend specifically on vehicle motion, or are different motion each time? For example, a lane change maneuver is different if performed on a curved road vs. a straight road?
- How could we detect novel, non-maneuver based road events, such as deer crossings or highway travel?
- How can a classification system consider all these different types of maneuvers, the priorities, and whether multiples are allowed to coexist simultaneously?

I embarked on a journey to create more classifiers to answer these questions. As we (myself and my summer research assistant) discovered, the answer to the first question is “no”. 30 or 60 fps video, along with vibration reduction in the camera and the vehicle itself and the short duration of an event such as hitting a pothole, prevented us from accurately detecting such events.

Luckily, we had success with the other three questions. We found, for example, dynamic time warping was a valid way to match lane change events, without any depth estimation at all, lending credence to the idea of dynamic time warping as a general approach to the maneuver-matching problem. We created a definition for coexistence of different maneuvers, and an overall classification scheme to make use of all the classifiers we developed.

Frequently, when developing the classifiers described in this chapter, the final design we arrived at was quite different from our initial design. To give one example, I struggled with the problem of highway entrances and exits for quite a long time, even discussing the problem with industry engineers. At one point, I even looked up the legal requirements for an interstate highway entrance in Michigan, thinking perhaps we could design different classifiers for different designs. Even this turned out to be a dead end, because variations from the code are sometimes approved! (Such a highway entrance exists in Ann Arbor: from Barton Hills onto M-14.) In the end, we discovered image classification was a better approach (as described later in this chapter).

In this chapter, I describe the processing pipeline that facilitates generating an index to search for vehicle maneuvers from dashboard camera video. As in Chapter II the system classifies video frame sequences against a set of reference video clips, provided by the user, that demonstrate the vehicle maneuvers (e.g., right turn, U-turn, driving in reverse) to be indexed. Our system does not rely on GPS, accelerometry, or other metadata to determine the motion of the ego-vehicle. Instead, we utilize monocular dashcam video and an existing deep learning model, DeepV2D [42], to find translational and rotational camera motion between successive frames. We process these pose estimates into a vehicle trajectory, and compare trajectory segments to the reference maneuvers. The system proceeds in two high-level phases: In the first, computationally more expensive phase, we apply the DeepV2D model to reconstruct high-resolution ego-vehicle trajectory from monocular front-facing dashcam videos. In the second phase, we compare the extracted trajectories to the reference maneuvers and label matching video clips. In this way, the reconstructed trajectory data acts as an index for the video library, enabling efficient search for vehicle maneuvers. We evaluate our approach against human-labeled ground truth for common road maneuvers, and compare against a classification approach that uses measured GPS rather than reconstructed trajectories. In addition to classifying sequences directly, we describe how the system can be tuned to eliminate as many uninteresting frames as possible while capturing *possibly relevant* video segments (i.e., tuning a particular recall threshold), reducing workload for a human evaluator.

We describe, implement, and evaluate classifiers for these common road maneuvers. We find utilizing a "min pool" approach to allow each classifier to select the closest match between several reference maneuvers outperforms a single reference maneuver. We also build an overall classifier to operate the individual classifiers and prioritize detected maneuvers. When multiple maneuvers coincide in time (e.g., deceleration while turning) the overall classifier applies priority rules to resolve conflicting output from individual classifiers and produce the best labeling.

We make the following specific contributions:

- We use reconstructed X-Y position to search a video library for turn maneuvers using dynamic time warping (DTW).
- We show DTW is a useful algorithm to measure maneuver similarity beyond X-Y position by using projected lane position and estimated vehicle speed to

detect lane changes and deceleration maneuvers, respectively.

- We implement and evaluate individual classifiers for the maneuvers described in (1) and (2).
- We describe, implement, and evaluate an overall classifier to compose results of individual maneuver classifiers and produce the best label for each time instant (video frame).

Overall we find estimations of vehicle maneuvers from monocular dashcam video is an effective technique to classify video frames.

3.2 Turn Detection Technique

Figure 2.2 shows an overview of our vehicle maneuver classification pipeline. We first use DeepV2D to extract translational and rotational camera movement at each frame to obtain a time series of X-Y coordinates corresponding to the ego-vehicle's trajectory. We can, optionally, use simple heuristics to label stop, reverse, and straight-line motion [53]. We then compare trajectory segments against reference maneuvers using dynamic time warping to compute a distance measure. Finally, we perform a best-first match of turn maneuvers using a greedy approach. We will describe each of these steps in detail in the following sections.

3.2.1 DeepV2D

DeepV2D [42] is a deep learning network design for estimating camera motion and depth from video. DeepV2D consists of a Stereo Module, to perform stereo reconstruction from images and a camera motion estimate, and a Motion Module, which uses depth to estimate camera motion. The motion module finds initial estimates for the sequence of frames using a pose regression network to estimate the transformation parameters between images. These initial estimates are then refined in an iterative process using a projective warping function on the differentiable transformation of the input image to produce a warped feature map. The estimated feature map is then compared to the original image feature map and the difference is used to update the pose estimate for the next frame (see Figure 2.1).

We use a model of the DeepV2D architecture trained via RMSprop [45] and the Kitti dataset [12] (with ground truth motion estimated by ORB-SLAM2 [30]) to infer the motion of the ego-vehicle in our dataset. The input to DeepV2D is a series of five video frames and the output is the estimated depth map and motion between the third and fourth frames. We therefore process each series of five frames from the recorded video library to obtain a motion track at the same framerate as the original video.

3.2.2 Dynamic Time Warping

Dynamic time warping [38] (DTW) is a measure of the similarity of two signals that may differ in duration. DTW performs a sequential matching between the signals while ignoring time differences. In effect, it "stretches" ("warps") one signal (or parts of it) to match another and computes the difference between matched values as the distance measure. (Figure 2.3 shows an example of two right turns). Whereas the original DTW algorithm is of $O(n^2)$ computational complexity (where n is the number of matched points), implementations such as FastDTW can approximate DTW at $O(n)$ complexity [37]. The reduction in computation time, along with comparative simplicity relative to other methods of pattern recognition, has enabled DTW to be widely and efficiently used for applications like speech recognition [21], gesture recognition [43], and detection of vehicle maneuvers [20]. We use DTW to quantify the similarity of vehicle trajectories from DeepV2D against the reference maneuvers.

DeepV2D produces a three-dimensional rotation matrix and translation vector for each sequence of five frames. We found using DeepV2D to produce camera motion estimates at the original 30 frames-per-second of our test video produced the highest-quality motion estimates. We discard the Z component and keep a time series of ego-vehicle X-Y coordinates at each frame. Because of the high temporal resolution of the video, we can reconstruct camera motion estimates at equal or better resolution than is typically available from consumer-grade GPS devices.

3.2.3 Endpoint Detection and Candidate Maneuvers

Since we have no information about when a given maneuver may start or end, we perform automatic and simple endpoint detection for each DTW sequence. As the DTW matching process has no a priori information—knowing nothing about the positional track in advance—it must consider the possibility of a maneuver starting at every time step (video frame). We assume the length of a candidate maneuver can be 50% to 150% the length of the reference maneuver in 10% increments, which allows for normal variation in ego-vehicle velocity. We allow the start and end time to be any even-numbered frame in the recording, as we find this has no effect on accuracy. While the processing time to calculate the DTW matching score for an individual reference maneuver is relatively short, performing comparisons of all possible candidate maneuvers is computationally expensive. As a strategy to reduce processing time, we enforce a few simple filtering heuristics on the four types of turns processed with DTW (summarized in Table 2.1). For left and right turns, the overall motion must match 80% of the candidate maneuver's movement along the x- and y-axes. U-turns must have sufficient x-axis movement, and K-turns (three point turns) require at least a half second (15 frames) of reverse motion. We find these simple heuristics slightly improve maneuver classification accuracy while reducing computational time by an order of magnitude (see Table 2.2).

3.2.4 Reference Maneuvers and Best-First Search

We define four turn categories: left turn, right turn, U-turn, and K-turn (examples are shown in Figure 2.4). (Note that the K-Turn looks odd because DeepV2D incorrectly estimates rotational speed in reverse, simply because the model isn't trained on reverse motion. However the error is consistent across maneuvers, so classification is still possible.) Each turn category is effectively an independent classifier on the camera-motion data produced by DeepV2D. DeepV2D motion estimation is expensive but need only be done once; a user can re-run subsequent analysis with a new set of reference maneuvers.

At least one reference maneuver must be defined for each turn classifier. For a reference maneuver, the user is responsible for defining the start and end time and selecting the baseline length for candidate maneuvers (which will then be scaled as appropriate). We compute DTW distances and select non-overlapping maneuvers (with an optional enforced "dead time" between maneuvers) with the lowest distance measures first. This means each time in a video will be scored with the lowest-distance match to the reference maneuver, and to find the true maneuvers we can evaluate different choices of thresholds for distance measure to capture a high number of true positives while eliminating as many false negatives as possible.

We believe this video search technique functions best as an analogy to a *Bloom filter* [4] (a data structure to determine whether a value is possibly in a set or definitely not in a set). Like a Bloom filter, our turn search techniques function best as an *augmentation* of human searching: we wish to eliminate as many uninteresting frames as possible without eliminating any maneuvers that we seek.

3.2.5 Multiple Reference Maneuvers with Minpool

We desire a distance measure that quantifies the similarity of two maneuvers. Since every maneuver is slightly different there is no perfect reference maneuver that defines a left turn – instead there is a broad range of what can be considered a left turn. For example, left turns can be "sharp" (more than 90 degrees) or "slight" (less than 90 degrees) in addition to a standard 4-way perpendicular intersection. We improve our model using multiple reference maneuvers and calculating the minimum distance measure to any of them. We call this the "min pool" technique as inspired by the convolutional neural network technique of selecting the minimum pixel from a set of nearby pixels in pooling layers. Let R be the set of reference maneuvers, c be a candidate maneuver, and $dtw(c, r)$ be the dynamic time warping distance between c and r. We define a new distance:

$$d(c, R) = min_{r \in R} dtw(c, r)$$

We find this technique of including multiple reference maneuvers produces higher-quality results than a single reference maneuver at the expense of additional computation, as we will elaborate in Section 3.5.

3.3 Individual Classifiers

We have described the concepts used to find turn maneuvers, especially DTW of candidate maneuvers and best-first search. We find these concepts can be used to detect additional maneuvers. However, it is often the case that common maneuvers are not completely defined by X-Y position. In this section we describe three additional maneuver detection techniques. We find that DTW can work well with time-series data besides X-Y position – we show this by detecting sharp deceleration events and lane changes. We also describe a novel technique for finding highway merges and exits.

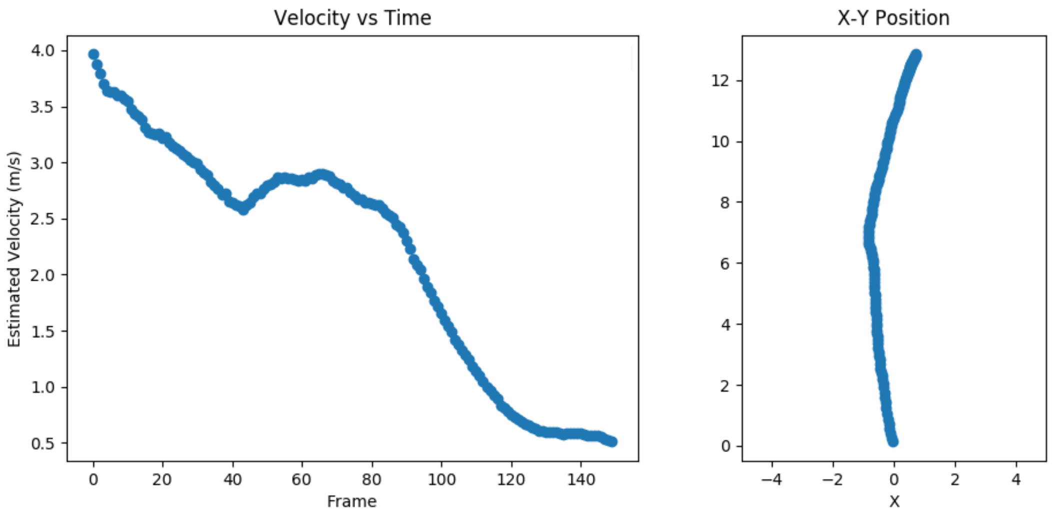


Figure 3.1: Estimation of motion and speed from a maneuver as the vehicle slows down while approaching a stop sign and comes to a stop

3.3.1 Deceleration

Our maneuver matching technique relying solely on X-Y position does not give any information about speed because DTW stretches maneuvers in time. For detecting turns, this property is useful, because a left turn can happen at any speed. Other maneuvers, however, can be defined in terms of a change in speed, such as decelerating to stop at an intersection. We can utilize speed as an additional component of measurement and search for maneuvers in the X-Y-S space. Figure 3.1 shows an example of a deceleration maneuver as the car approaches an intersection.

For each frame, we start with a coordinate (x_i, y_i) given from DeepV2D for each frame i in the video. We compute the instantaneous speed at each frame by using the position of the next frame as follows:

$$s(x_i, x_{i+1}, y_i, y_{i+1}) = \frac{\sqrt{(x_i - x_{i+1})^2 + (y_i - y_{i+1})^2}}{\Delta t}$$

Where $\Delta t = \frac{1}{30}$ seconds (since our video is 30fps) is the time between the two consecutive frames. Because speed tends have a lower absolute value than position, we apply a 5x adjustment to the speed coordinate. We find this provides a balanced normalization of the dimensions for this search technique. We generate our new X-Y-S coordinates for each frame.

$$\{(x_i, y_i)\} \rightarrow \{(x_i, y_i, 5 * s(x_i, x_{i+1}, y_i, y_{i+1})\}$$

We demonstrate the capability of our new search space by finding periods of strong deceleration, such as a sudden braking for a pedestrian, stop sign, or stoplight. We do not search for strong acceleration as this is much more unusual in normal driving behavior. As with turns, given a reference maneuver, we use DTW to find similar maneuvers in the X-Y-S coordinate system.

3.3.2 Lane Change

To detect lane change maneuvers, we use an open-source implementation of a the LaneNet lane detection model [32, 14] to obtain estimated lane line positions relative to the ego-vehicle. Then, given a reference lane change, we use DTW over the lane position to find similar maneuvers.

LaneNet is pretrained on the Tusimple Lane Challenge Benchmark[46]. LaneNet detects four lane lines per image and outputs a classification score for each. A lane "line" may be an actual painted line on the road surface or the edge of the road. We choose the three most prominent lanes based on the classification score. An example of the model's output is shown in Figure 3.2. On a two-lane surface road, it is typical to have only one lane line (or none), and using only three instead of all four reduces the likelihood of falsely detecting a nonexistent lane.

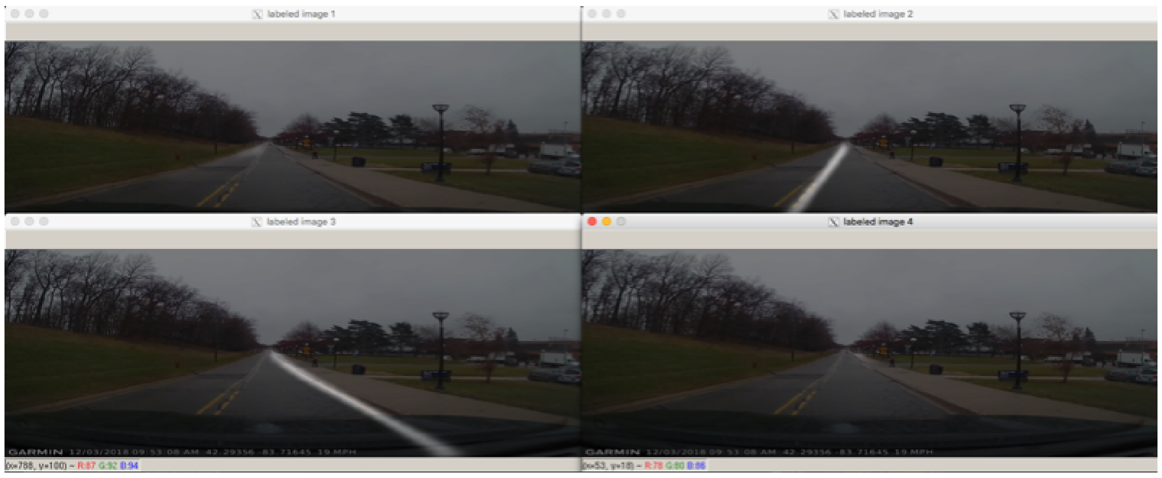
We wish to identify the position of the car relative to the lanes, so we use the position of the lane line at the base of the frame. To determine this position, we perform a Hough transform [] on the lane line segmentation output, and then use the position of the center of the line. If the lane line continues off the left or right edge of the image, we project it linearly to determine the coordinate corresponding to the baseline of the image (which is why negative pixel values can arise for the detected lane position, as in Figure 3.2). We thus obtain a three-dimensional coordinate to represent the camera's position relative to the lanes for each frame.

During a lane change, the lane lines move horizontally with respect to the car. This motion causes the positive trend in all three lane lines as seen in Figure 3.2. When the ego-vehicle is driving straight in its lane, the estimated pixel coordinate of each lane line does not vary. We distinguish between right and left lane changes based on whether estimated lane line positions follow an increasing or decreasing trend. We again apply DTW to find the similarity in trends of the lane line positions over variable length time intervals. We perform DTW on the position of each lane relative to the ego vehicle. Then, using the same procedure as previously described for turns, we identify the candidate maneuvers with the smallest DTW distance to a reference lane change, and find all lane changes.

3.3.3 Highway Detection

It is challenge to identifying highway merge and exit events directly with our maneuver matching technique, as the length and shape of highway entrance and exit ramps varies considerably. The min-pool approach only helps detect known maneuver types and cannot detect unknown ramp configurations. Detecting highway transitions using changes in speed or acceleration is also not viable, as our experiments showed DeepV2D does not perform well on highway environments, where camera pose changes considerably frame-to-frame and there are comparatively few visual features to obtain a good depth map. As such, on highways, DeepV2D yields a noisy velocity signal.

To detect highway merges and exits, we instead use a simple image classifier and k-means sorting. We recorded approximately 68 minutes of training video on local



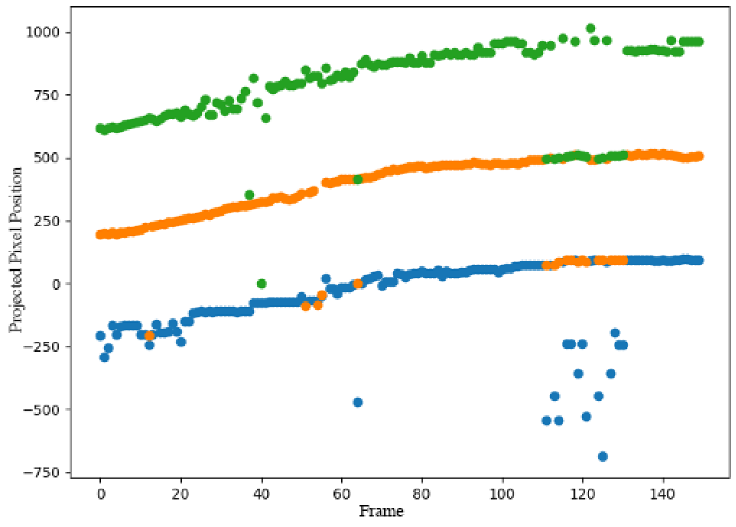


Figure 3.2: Graph of the location of each lane (in pixels) in the dash cam against time for a typical lane change. Each color shows the inferred position of a specific lane.

highway and surface roads (a disjoint set of roads from those used in our evaluation dataset). We extract a still frame at a stride of 1 second. Frames taken on merge and exit ramps were discarded, as are frames where the vehicle is not in motion. Using the resulting still frames, we use transfer learning on an Inception V3 network with two categories: "highway" (n=287) and "surface streets" (n=585) with a 10% validation set and find the model achieves 94% accuracy after 1000 training steps.

We use k-means clustering to best fit the distributions of the on-highway and surface street image clusters in our video dataset. While this is (simple) unsupervised

	Highway On/Off	**Highway Merge/Exit**
Precision	0.83	0.54
Recall	1.0	0.96
F1 Score	0.90	0.69
Count (True)	85	23
Count (False)	348	116
Effective Frames Eliminated	n/a	70.5%

Table 3.1: Highway classifier and merge/exit detector performance

learning, we nonetheless use leave-one-out testing for each video to avoid biasing the distribution. We consider time windows of 30 seconds each from our video dataset: we extract one still frame per second and use the image classifier on each. We then average the scores of these 30 frames and determine to which cluster the average score belongs. This procedure provides a “highway” vs. “surface” classification for each 30-second interval.

To identify highway merges and exits, we consider time windows of 180 seconds in length. We take the average of the first 60 and last 60 seconds, leaving the middle 60 seconds as “dead time”, as classification output is unstable during the transition, while the ego vehicle is on an on- or offramp (which we assume to last less than 60 seconds). We select these constants such that the windows are sufficiently small to detect brief periods of highway driving while still keeping the classification simple and inexpensive to compute. We report a highway transition when the road type prediction in the first 60 seconds and last 60 seconds differ. We illustrate this clustering technique for a sample video segment in Figure 3.3. Three highway transition events occur, and each is correctly identified by the classifier.

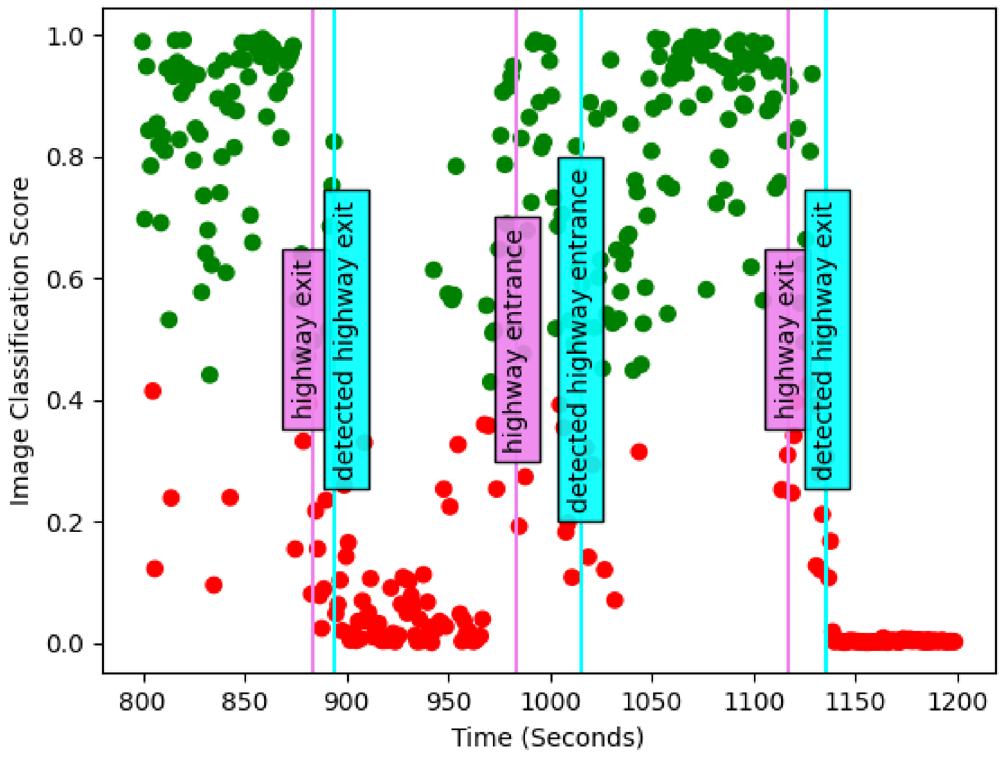


Figure 3.3: Image classification score (Y-axis) of image frames from a video including portions of on-highway and off-highway driving. K-means clustering into two groups shows likely on-highway (green) and off-highway (red) frames. The highway entrance and exit events are annotated as purple. These are marked as occurring at the earliest possible interval. Detected highway entrance and exit events are marked in teal.

3.4 Evaluation Methodology

3.4.1 Video Dataset

We collected Dashcam videos in a total of 15 driving sessions on local surface streets and highways in Washtenaw County, Michigan. Our dashcam videos were recorded with a Garmin Dash Cam 55, fixed to the center of the windshield of the vehicle at approximately eye level. We collected our own dashcam video, rather than using existing public data sets, because DeepV2D requires camera intrinsics for estimation of camera motion.

We made no preparation for environmental or traffic conditions except for avoiding snow, as DeepV2D performs poorly due to lack of adequate training data in snowy conditions. Road surfaces were mostly (>95% by time) paved, but road surface conditions were highly variable; roads are often patched or in need of repair and lane lines are often faded. Speed limits ranged from 25mph (residential streets) to 70mph (interstate highway). Of 15 video segments, 13 produced usable camera motion data. One was excluded for low-light conditions (near sunset) and the second for sunlight refracted through a dirty windshield. The 13 videos used for analysis totalled 381 minutes of driving footage.

We manually labeled all maneuvers that occur in recorded videos. Most maneuvers are simple to score unambiguously, such as turns and lane changes. When in doubt about whether a particular maneuver occurs (such as a slight left turn), we err on the side of labeling it. We found the deceleration maneuver is challenging for humans to score objectively, however, so we followed the following protocol:

- Human scorer identifies all *unambiguous* deceleration maneuvers.
- Run evaluation and obtain list of all false positive maneuvers.
- For each false positive, consider changing to a true positive if upon re-examination

the maneuver contained a speed drop of >10mph.

3.4.2 Data Preprocessing

Videos were converted to JPEG frames using ffmpeg for processing by DeepV2D, running on an Intel Core i7 workstation with an NVIDIA Titan XP GPU. This workstation ran Tensorflow 1.12 with CUDA 9.0 on Ubuntu 16.04.

3.4.3 Selection of Reference Maneuvers for Evaluation

We select ten percent of all maneuvers at random to be reference maneuvers. These reference maneuvers are excluded from the overall statistical calculations. Following the conceptual goal of using k-fold cross-validation to split our reference and test data, effectively each reference maneuver (or set of maneuvers) serves as the "training" set for a given run of a classifier and every other maneuver is part of the "test" set. We manually select ground truth endpoints to best capture the given maneuver.

3.5 Results

We assess the accuracy of each maneuver's classification as follows: A reference maneuver of a particular category is chosen at random. The classifier then finds the best matches to this reference maneuver throughout the duration of the video, allowing no maneuvers to overlap. These candidate maneuvers are compared to the human-annotated ground truth. If a maneuver is detected within four seconds of the ground truth annotation, it is considered correctly classified; otherwise it is considered a false positive. We then compute the ROC curve using the list of distance measures of detected maneuvers. False negatives (that is, ground-truth maneuvers not detected by the classifier) are included in the ROC calculation with the minimum-detected distance measure (to penalize the classifier for failing to detect these maneuvers).

To evaluate quality of the DTW classifiers, we use the area under the receiver operating characteristic curve (AUROC) [49] as our primary metric. The output from each classifier is essentially a sorted list of distance measures and segments of video. As the distance increases, there is worse similarity to the reference maneuver. Interpretation as a binary classifier requires choosing a cutoff threshold for each classifier's distance measure. Choosing a threshold is an arbitrary exercise as it is not clear what constitutes a "good" value. Therefore we instead use an ROC curve, which does not require choosing a threshold. The ROC curve is a plot of false positive rate vs. true positive rate, and integrating this curve gives a measure of quality of the classifier without requiring a specific numeric threshold.

3.5.1 Turn Classifier

The results are shown in Figure 3.4. We use a micro-average ROC across every reference maneuver, meaning each individual true or false classification is evaluated with respect to ground truth and summed. This procedure removes any potential bias for any particular reference maneuver to give better results (for example being smaller or shorter than other reference maneuvers), as opposed to a macro-average across maneuvers. However it penalizes different maneuvers for having different distributions of correct and incorrect results and thus is generally lower than the macro-average. The micro-average ROC curve is shown in orange, representing the average of every maneuver in the 10% test set.

We show the results of our "min pool" technique described in Section 3.2 as the ROC curve in green. We find the "min pool" technique exceeds the individual average, as this allows the classifier greater flexibility in matching maneuvers to a library of known maneuvers. This improvement occurs because even with randomly chosen maneuvers, different road intersections and common maneuvers will have slightly different geometry, such as a U-turn across a narrow road vs. a wide road.

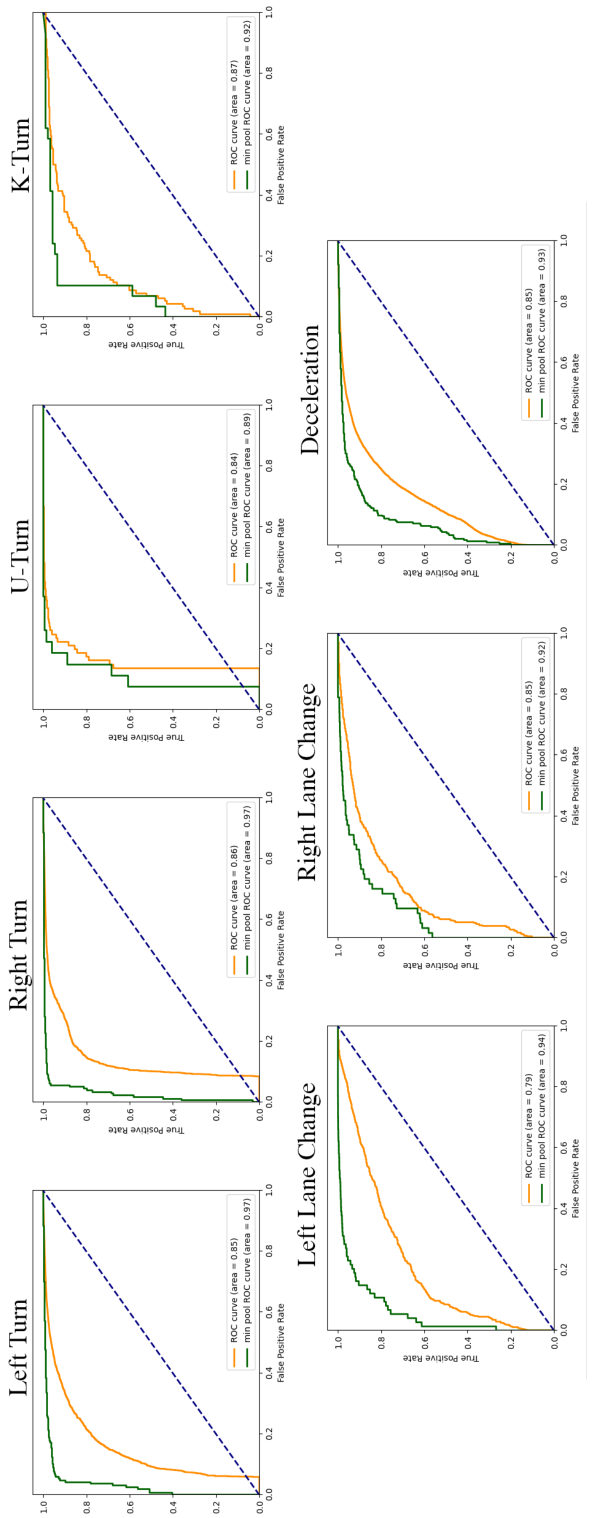


Figure 3.4: ROC curves for seven individual classifiers. We find that the min pool approach of looking for the best match among several maneuvers (green) is superior to the average of individual randomly chosen maneuvers (orange).

3.5.2 Comparison to GPS

To compare our technique to off-the-shelf GPS, we show the non-weighted average ROC curves [39] for each of four turn maneuvers detected in Figure 2.6. Each individual ROC curve for a particular maneuver (shown in thin blue lines) is averaged with equal weighting at each step (shown in the bold blue line). An interval of one standard deviation above and below the mean is shown in grey.

The green line represents the reconstructed trajectories when using GPS instead of the DeepV2D portion of our computational pipeline. However, we find the DeepV2D motion estimates consistently outperform our GPS measurements, due to the spatial and temporal resolution advantage of DeepV2D's 30 Hz estimates relative to the 18 Hz 3-meter resolution of the GPS. The typical GPS accuracy of approximately 3 meters [47] is simply not sufficient for tight maneuvers, as a standard U.S. highway lane is 3.7 meters wide [50] while city roads are narrower. Therefore, a maneuver like a U-turn is difficult to reconstruct accurately if the maneuver is performed on a two-lane road that is at most 8 meters wide. Figure 2.6 shows the ROC curve using 18 Hz GPS data (green) against the DeepV2D data (blue). GPS data is not nearly precise enough for good results.

3.5.3 Calculation of Discarded Frames

While we prefer AUROC as the primary metric, we are also interested in using our system as a binary classifier that finds all instances of a particular maneuver. We wish to avoid discarding positive events, meaning that we prioritize a high recall, at the expense of precision. This use case requires selection of a threshold to meet the desired recall rate, and then calculating the resulting precision and F1 score. Our DTW classifier partitions the video into many different segments, each representing a potential maneuver with a DTW score representing how close it is to the reference maneuver(s). By introducing a specific score threshold, we convert the DTW score to a binary classification. This system is intended as support for a human looking for maneuvers, so we are particularly interested in the *fraction of discarded frames* for a particular recall. We can approximate this measure as follows: Let N be the number of potential maneuvers in the partition, and $TP + FP$ be the number of events that were predicted positive by our classifier. $\frac{TP+FP}{N}$ is the fraction that is kept, so we define $F_{eliminated}$ to be the percentage of eliminated sections of video.

$$F_{eliminated} = 1 - \frac{TP + FP}{N}$$

Let R be recall and P be precision. We can easily derive $F_{eliminated}$ from a given recall and precision with the following algebra:

$$\begin{aligned} F_{eliminated} &= 1 - \frac{TP + FP}{N} \\ &= 1 - \frac{\frac{TP}{TP+FN} * (TP + FN)}{\frac{TP}{TP+FP} * N} \\ &= 1 - \frac{R * (TP + FN)}{P * N} \end{aligned}$$

Where the first step is multiplying by 1, and the second step is substitution from the definition of precision and recall. Because maneuvers differ slightly in length we refer to this as the *approximate percent of discarded frames*. We show the calculation of $F_{eliminated}$ for individual classifiers in Table 3.2.

3.5.4 Comparison of Turn Classifiers

We wish to quantify how well DTW distance measures can function as as a metric across different classes of maneuvers. This experiment shows the potential to search for multiple maneuver types instead of only one at a time. To answer this question, we use a greedy approach to classify these four turn maneuvers, recursively selecting the candidate maneuver with the lowest distance measure among all four turn types. (We

Classifier	**Recall**	**Precision**	**F1 Score**	**Frames Eliminated**
Deceleration	0.98	0.06	0.12	**34%**
	0.95	0.07	0.13	**50%**
	0.90	0.13	0.23	**74%**
	0.80	0.23	0.36	**92%**
	0.61	0.46	0.53 (max)	**95%**
Left Turn	0.98	0.05	0.10	**46%**
	0.95	0.19	0.32	**86%**
	0.90	0.34	0.49	**93%**
	0.80	0.43	0.56	**95%**
	0.74	0.52	0.61 (max)	**96%**
Right Turn	0.98	0.12	0.22	**34%**
	0.95	0.53	0.67	**86%**
	0.92	0.85	0.89 (max)	**91%**
	0.90	0.85	0.88	**91%**
	0.80	0.89	0.84	**92%**
Left Lane Change	0.98	0.01	0.02	**31%**
	0.95	0.02	0.03	**64%**
	0.90	0.02	0.04	**77%**
	0.80	0.06	0.10	**92%**
	0.33	0.61	0.43 (max)	**99%**
Right Lane Change	0.98	0.01	0.03	**56%**
	0.95	0.02	0.03	**57%**
	0.90	0.02	0.03	**62%**
	0.80	0.03	0.06	**84%**
	0.18	0.79	0.30 (max)	**99%**

Table 3.2: Individual classifier performance by recall threshold

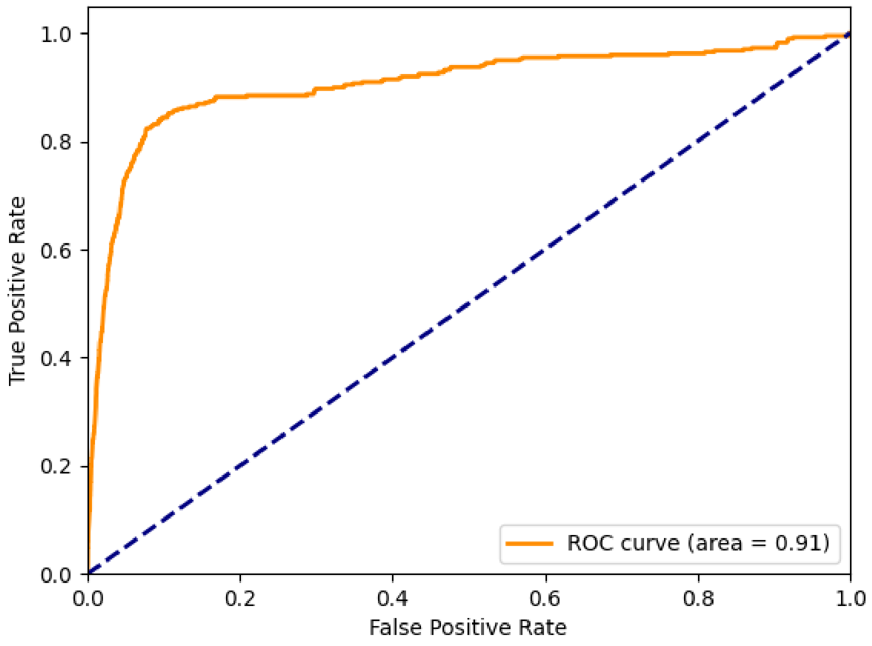


Figure 3.5: Receiver Operating Characteristic of turn classifier.

purposely ignore the fact that certain maneuvers may have different distributions of distance measure values.) We then check the time duration of this candidate maneuver to ensure no prior selected maneuver is in conflict. We find achieve an overall AUROC of 0.91; the ROC curve is shown in Figure 3.5.

3.5.5 Additional Classifier Results

3.5.5.1 Deceleration

Overall we find an AUROC of 0.85 for the deceleration classifier using this technique on 10% randomly chosen maneuvers, as seen in Figure 3.4. When augmented with the min-pool technique, we find that AUROC improves to 0.95. While the F1 score is not as high, we find the effective number of frames eliminated can be quite high for a given recall. For example, with a recall threshold of 90% we can successfully discard 74% of the video frames as uninteresting. We note this discard rate is

	Best Match Overall	**Top 2 Matches**	**Top 2 Matches (without RF error)**
Matched	628	654	703
Missed	169	143	94
Accuracy	78.8%	82.1%	88.2%

Table 3.3: Overall classifier accuracy for Best Match (Top 1) and Top 2 detection of maneuvers by time. We also show the Top 2 matches assuming the random forest model always chooses correctly.

a bit lower than for turn maneuvers; human scoring of deceleration events is itself a subjective exercise and difficult to quantify compared to the other maneuvers.

True \ Detected	**Highway Merge / Exit**	**Left Turn**	**Right Turn**	**U-Turn**	**K-Turn**	**Left Lane Change**	**Right Lane Change**	**Deceleration**
Highway Merge / Exit	22 (96%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	1 (4%)	0 (0%)	0 (0%)
Left Turn	0 (0%)	161 (89%)	0 (0%)	0 (0%)	0 (0%)	4 (2%)	15 (8%)	0 (0%)
Right Turn	0 (0%)	0 (0%)	133 (80%)	0 (0%)	0 (0%)	12 (7%)	21 (13%)	0 (0%)
U-Turn	0 (0%)	0 (0%)	1 (3%)	21 (68%)	0 (0%)	2 (6%)	7 (23%)	0 (0%)
K-Turn	0 (0%)	0 (0%)	1 (4%)	0 (0%)	26 (93%)	0 (0%)	1 (4%)	0 (0%)
Left Lane Change	1 (1%)	1 (1%)	14 (18%)	2 (3%)	0 (0%)	5 (7%)	53 (70%)	0 (0%)
Right Lane Change	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	1 (2%)	62 (98%)	0 (0%)
Deceleration	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	251 (99%)

Table 3.4: Confusion matrix matching True vs. Detected maneuvers.

Recall	**Precision**	**F1 Score**	**Frames Eliminated**
0.98	0.08	0.15	**34%**
0.95	0.10	0.17	**49%**
0.90	0.14	0.24	**65%**
0.80	0.21	0.33	**79%**
0.52	0.47	0.56 (max)	**94%**

Table 3.5: Overall classifier performance by recall threshold

3.5.5.2 Lane Change

We find an AUROC of 0.85 for both left and right lane changes. If we use 10% of of the maneuvers as reference maneuvers and take the minimum distance, the AUROC improves to 0.94 for left lane change and 0.92 for right lane change, showing that DTW is a robust technique for finding lane change maneuvers.

3.5.5.3 Highway vs. Surface Streets

We show the results in the "Highway On/Off" column of Table 3.1. We find that for a total of 85 on-highway epochs, we achieve a recall of 1.0 and precision of 0.83 for a total F1 score of 0.90. While our approach correctly identifies every on-highway epoch, we find the false positives from the image classifier occur primarily on major surface streets that have certain features in common with a highway (e.g. medians, few businesses or buildings, trees set back far from the road). As our overall goal is to maximize recall even at the cost of lower F1 score, we find this to be an acceptable tradeoff in practice.

3.5.5.4 Highway Merges and Exits

The overall performance of the merge/exit classifier is shown in the "Highway Merge/Exit" column of Table 3.1. We find that this technique gives high recall (0.96) and precision of 0.54, allowing us to eliminate approximately 70.5% of the total frames in our video dataset.

3.6 Overall Maneuver Classification

Whereas each maneuver classifier can be used individually, we also wish to compose classifiers to determine the best labeling for every point in a video.

3.6.1 Technique

Building an overall classifier is not as simple as running each classifier individually and comparing the results. We begin by defining a coexistence matrix specifying which maneuver label should be preferred when they overlap (e.g., prefer K-turn over turn, as a turn may be part of a K-turn), and whether two labels may coexist (deceleration may coincide with a turn). For example, we disallow turn maneuver labels when the scene is classified as highway driving. The coexistence matrix can be used to optimize computational performance (skipping classifications that are disallowed by the matrix) or to improve classification accuracy by eliminating false positives.

When classifiers produce a label combination disallowed by the coexistence matrix, we must resolve the conflict. We apply several heuristics:

- If the evaluation scores of two classifiers are comparable, we can choose the more confident score. For example, if we find a time segment that is similar to a left turn with a distance measure of 100, and similar to a right turn with distance measure of 10,000, we can prefer the lower distance score.
- We filter turns detected while driving on the highway, as the turn detector produces false positives at highway curves.
- Deceleration may overlap with any other maneuver.

- Lane DTW and turn DTW do not provide comparable distance measures, and there is no obvious heuristic for which label to prefer when both classifiers report a match. To solve this problem, we trained a simple random forest classifier (validated on an independent selection of lane changes and turns) which attempts to determine whether a particular period of time is more likely to be a lane change or turn, based on the positional change over time.

The overall classification algorithm is as follows: we run each classifier separately and obtain maneuver predictions at each timestep. For the lane-change, turn, and deceleration classifiers the prediction output is a distance measure from reference maneuvers, but the on-highway classifier is a binary classification. Therefore we perform highway classification first and mask the predicted on-highway time periods. For each subsequent classifier, we check whether coexistence with prior predictions during the same time period is allowed. For maneuvers with comparable distance measures (e.g. right turns and left turns) we choose the maneuver with the lowest distance measure. After all conflicting predictions are pruned, we add the best detected maneuver's time period to the mask to prevent an overlapping maneuver from being detected. After importing and pruning the output from each classifier, the result is a time series with the best match of maneuver(s) for each point in time.

3.6.2 Results

Table 3.5 shows the precision and F1 scores at different recall rates. As with the individual classifiers, the primary purpose of our technique is to find as many *possibly interesting* sequences of frames as possible while eliminating a high number of uninteresting frames. We find that while detecting 90% of all maneuvers we can discard nearly two-thirds (65%) of frames. At the maximum F1 score, we detect about half (52%) of all maneuvers correctly while eliminating 94% of frames.

We measure the accuracy of matching one or more detected maneuvers at a given time to the correct maneuver at that time. We first perform this matching for the best single detected maneuver at a given time. We find 654 maneuvers match correctly and 143 are missed, for a total overall accuracy of 82.1% (Table 3.3). When we expand the classifier to allow selection of the best two maneuvers, we find accuracy improves to 88.2%.

We show a confusion matrix of true vs. detected maneuvers in Table 3.4. We find left turns commonly alias with U-turns and K-turns, as each of these begins with ego-vehicle movement to the left. We also see that left and right lane changes alias frequently, due in part to some variance in the randomly-chosen reference maneuvers for each.

We also find that the overall classifier approach frequently has false positives for left and right turns compared to the labeled ground truth; these arise due to curves in the road. For example a road curving to the left can be detected by the classifier as a left turn even though a human marked it as forward/unremarkable. To better distinguish curves from turns, our trajectory-based method could be fused with other information sources, such as road segmentation or map data.

3.7 Conclusion

We have described, implemented, and evaluated classifiers for common road maneuvers. The input to these classifiers is reconstructed trajectory data from dashcam video. We evaluated our approach against human-labeled ground truth for common road maneuvers, and compared against a classification approach that uses measured GPS rather than reconstructed trajectories. We found utilizing a “min pool” approach to allow each classifier to select the closest match between several reference maneuvers outperforms a single reference maneuver. We built an overall classifier to operate the individual classifiers and prioritize detected maneuvers, with maneuver matching accuracy greater than 78% for best-fit.

CHAPTER IV

A Visual Query System for Road-View Video

4.1 Introduction

A common task in the development workflow of autonomous systems is searching a video archive to extract video segments that meet particular criteria. Such searches might be used to find test scenarios to evaluate agent performance under specific circumstances, for example, a segment where an autonomous vehicle stops at an intersection and waits for oncoming traffic to clear. The problem of video search is, in many ways, analogous to database queries over relational, tabular data. However instead of finding matching rows using criteria such as string matching and table joins, the output is frames matching specified output from computer vision operations. In contrast, a video processing *pipeline* has the purpose of efficiently processing all frames of a video through a series of computer vision operations. State-of-the-art video pipeline systems, like Scanner [34], allow configuration and execution of these processing pipelines. Additionally, Scanner includes desirable optimizations such as delta-frame decoding, caching, and frame skipping.

In this paper, we focus on the specific challenge of performing query planning for video search – i.e., a query of valid frames matching specific predicates – with the goal of minimizing the amount of computation required to evaluate the query. We consider a query to be an operation requiring evaluation of one of more computer vision models, such as object detection, optical flow, depth estimation, and lane line detection. We design a simple declarative language to map keyword predicates to computer vision operations, which can then be used in a sentence of conjunctions and disjunctions. Our key idea is that by decomposing a query into a tree structure and tracking which video frames have been evaluated for certain predicates, we can enumerate all possible query plans. Then, in the process of executing the query, we iteratively find the operation to execute with the lowest estimated cost that will resolve the largest number of frames. We develop and test several strategies for generating this cost estimation, ranging from simply picking the lowest-cost operation to analysis of the tree structure itself.

We integrate Scanner into an end-to-end query processing system to evaluate our cost-estimation strategies, thereby leveraging Scanner's video processing capabilities to dynamically create and execute operations on subsets of frames. We systematically generate a large number of random queries to use for our evaluation. We focus on the specific use case of vehicle dashboard video recording, as this is a use case where a large amount of searchable video can be generated in a relatively short period of time (e.g. evaluation of a fleet of driverless cars) and presents more challenges than stationary video such as security camera footage. We use our prior work on finding vehicle maneuvers in dashboard video recordings as one of the operations, along with searching for objects, and geometric relationships (left/right and front/back). Finally, we use our library of dashboard video recordings for the evaluation.

Our goal for system evaluation is to demonstrate *plausible* computer vision workloads operating within this system, enabling exploration of the capability of the query language and query tree execution structure. Our principal measurement for evaluation is latency, specifically the end-to-end runtime of Scanner running our custom video processing pipelines.

We make the following specific contributions:

- We develop a novel cost-optimization function and various evaluation strategies to process video queries with minimal computation.
- We build an end-to-end video query processing system to implement these optimization techniques.
- We evaluate this system using both automatically generated and handwritten queries on a library of dashboard camera videos, finding a 3.6x speedup over frame-by-frame search and 1.2x speedup for the best query optimization technique ("estimated root contribution") compared to lowest-cost optimization.

4.2 Related Work

As computer vision techniques have improved in accuracy, several approaches have been taken to improving the runtime performance of such systems. Broadly, many different software and hardware optimizations are possible. End-to-end video analytics systems are of particular importance as the amount of recorded video (e.g. YouTube) continues to grow [23].

A number of studies have taken the approach of designing a video query language with spacial semantics [23, 6]. Others focus on the idea of temporal queries over video feeds [8, 7, 5]. In the transportation domain, Koudas et al use spatial relationships to accelerate searches for fixed-camera video (e.g. "find car left of truck") [24]. Ahmed et al explore a query-based traffic monitoring application [1].

A second category of interest is video processing optimizations, e.g. cascade filters, data tiling, data reuse, online-monitoring etc [2, 36, 18, 10, 35, 28]. Moll et al evaluate the technique of "adaptive sampling": focusing on portions of video most likely to contain objects of interest [29].

Finally, a third category is recent work on video pipeline optimization, e.g. connecting different video operations in sequence, possibly with conditional paths [34, 51].

Table 4.1: List of terminology and definitions used.

Term	**Definition**
computer vision operation	A specific computer vision model which processes video frames, such as object detection or lane line detection.
operator	A keyword in the query language (such as "FRONT" or "LEFT") which accepts arguments and contains one or more computer vision operations to be executed to evaluate the query.
tree node (node)	An instance of an operator in the decomposed query tree, which also contains a frame vector and other metadata necessary for query evaluation.
frame positive rate	The rate of frames which are (in reality, in estimation, or in simulation) evaluated as true by a particular operator.
frame vector	A list of video frames. We start with all video frames in the unknown frame vector (subject to operations such as frame skipping, etc) and the system tracks frames between the unknown frame vector, the true frame vector, and false frame vector.
search algorithm	A strategy to choose which node (operator instance) in the query tree to execute next.

4.3 Video Processing Pipeline and Computer Vision Operations

To test and evaluate our ideas for query decomposition and planning we assemble an end-to-end pipeline for processing video queries, composed of computer vision models. We chose these models as examples of useful search pragmatics, e.g. detection

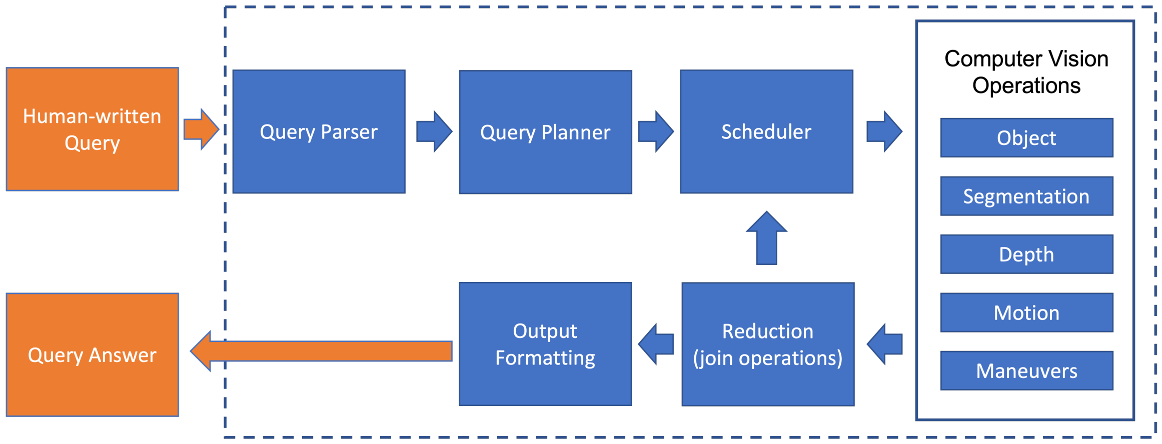


Figure 4.1: Overall system pipeline (blue) with inputs and outputs (orange).

of objects and lane lines. We use Scanner, an open-source video processing framework, as our primary component to perform the actual video processing operators and call the relevant computer vision models.

4.3.1 Comparison with DBMS Query Planning

We draw inspiration for query processing from database management systems (DBMS), especially SQL-based systems. The SQL language uses relational algebra to specify exactly what data should be returned from a particular query. The DBMS enumerates and optimizes the query, using estimates such as cardinality of the underlying tables, the cost of performing certain operations, and the underlying data structures such as indexes (typically B-Trees) and data ordering.

While this has certain similarities and provides the starting point for our language design and query processing, there are necessarily certain differences. First, we assume our videos have no underlying structure aside from being sequential, compressed MPEG4 files. This is quite different, than say, a text or numeric field defined in a table. In our system, each video is effectively a table, and the columns are the binary video frame as well as the computational results of each operation... not just the raw machine learning model output, but also the reduction or mapping of this to the predicate we wished to solve. This is somewhat similar to intermediate results that might be produces by a database engine.

Secondly, database engines attain high performance through particular attention to memory usage and caching; especially keeping track of which pages are in memory and reusing output from one operation as the input to the next operation without needing to evict and reload those pages later. We rely on Scanner to make optimal choices when possible, but multiple loads of the same video frame, for example, is difficult to avoid because image data has a larger footprint. Moreover, the computer vision operations can be quite expensive compared to the process of loading and formatting the input data. This is a direct contrast to a typical database query, where a predicate comparing two integers or checking a Boolean value is ultimately a single assembly code instruction, and even operations like string matching are computationally simple in comparison. In these cases, the architectural challenge is efficient management of a memory-bound process efficiently.

Thirdly, this marked difference in performance attributes allows consideration of alternative query processing techniques. In particular, as execution cost is dominated by computation rather than memory, keeping data resident in cache and memory for the next operation is no longer as high a priority. Certainly this would improve performance to some extent, but releasing this constraint allows focus to be places on how to efficiently explore and resolve a tree query graph in-place, instead of on rebalancing and transforming it to optimize for query operations.

As we describe in Section 4.5, our strategy is to choose which node of the query to process next, effectively breaking the query into tranches of simpler pipelined operations. Instead of query planning and enumerating all possible trees to compare costs, this means the focus is on estimating the cost of each operator, selecting the best operator to perform next, and keeping track of which video frames have been resolved to true or false.

4.3.2 Scanner

Scanner integrates open-source software libraries such as ffmpeg, grpc, and protobuf to create a distributed video "database" capable of executing pipelines of arbitrary video-processing operations. Scanner is designed for high-performance video processing at cluster scale, using efficient data structures, existing libraries, and incorporating ideas like data reuse, frame skipping, and parallelism that are difficult and tedious to implement from scratch. For example, the Scanner API supports selection of arbitrary frames from a video, and has built-in functions for common operations such as resizing images, striding, and writing output. An important feature of Scanner is the ability to implement custom, arbitrary video frame operations in Python, including those using TensorFlow and PyTorch operation graphs. These operations can then be used in pipelines alongside the built-in Scanner operations. This allows us to focus our experiments and measurements on the query language, query planning, and latency of execution, independent of the many variables inherent to video processing, as Scanner has already been optimized to load, decode, and process video frames.

4.3.3 Geometric-Related Computer Vision Models

For the purpose of this study we are interested in queries on geometric properties of objects contained in the video frame, e.g. whether two objects are next to each other, or whether an object is in the same lane as the ego vehicle. To support these operations we assembled a collection of open source, off-the-shelf computer vision models to use within our system from broad categories of geometric-related models that are of interest in studying dashboard video. First we have an object detector, which produces bounding boxes for different categories of objects. We used Mask-RCNN trained on the COCO dataset. A lane line detector model gives information about whether an object lies in the current lane, or to the left or right. A depth-estimation model allows us to estimate whether one object is in front or behind another. And finally, an optical flow model provides pixel-level estimations of motion between sets of frames. Additionally, we use our vehicle maneuver classifier as a queryable dataset.

We elaborate on the exact language semantics in Section 4.4. We then built and validated custom Scanner operations for each model. Each operation consists of a main function implementing a simple pipeline of operations and a kernel class implementing a *load* function to perform any necessary setup (e.g. load a TensorFlow graph) and then an *execute* function performing the forward inference pass of the model. Most of our operations (all except vehicle maneuver detection) use a GPU to run inference. While we try to keep each operator as simple as possible, some call arbitrary code as needed by the author of the model.

The goal of our project was not to choose the most efficient or most accurate models, but to show that the difference in latency of execution for different models can be exploited in the query planning and execution process to improve the over latency of a particular query. Moreover, while the operations we choose can indeed be performed in parallel, our goal is not to reduce latency through massive parallelism but to show that better choices in query execution can lower execution time. This is particularly relevant for edge processing, for example, where resources are be more constrained than in a datacenter.

Table 4.2: Descriptions and computer vision models used by each operator.

Function	**Type**	**Functional Description**	**Model(s)**					
			O	L	S	D	M	V
Left/Right	(object vs object)	compare bounding box locations	X					
Left/Right	(ego vs obj)	check if bounding box is over lane line	X	X				
Front	(object)	object exists in frame	X					
Front/Back	(object, object)	depth of two objects	X			X		
In	(object, segmentation class)	check if bounding box is (nearly) enclosed by segmentation class	X		X			
Moving	(object, direction)	check if majority of pixels in bounding box are moving in a direction	X				X	
ego.maneuver	Comparison	check for specified maneuver						X

4.4 Language and Operators

4.4.1 Language

We have developed a simple scripting language to use as the declarative input to our system. The language has three different operator types:

- **Functions**: have a keyword defining the function, followed by one or more arguments in parenthesis. Functions drive one or more computer vision operations, e.g. comparing geometry of two objects or establishing that an object is in-frame.
- **Comparisons**: check whether a keyword is equal to some defined value. Used primarily for vehicle maneuver attributes, as the word "ego", referring to the car from which video is being recorded, is a special keyword mapping to maneuver searches.
- **Logical Operators**: "AND" and "OR" operators, which allows logical combinations of predicates.

Table 4.3: Language Keywords

Category	**Keyword(s)**	**Description**
Ego	ego.maneuver	Ego vehicle state, matching with known maneuvers
Object	(classes of objects), e.g. car, pedestrian, bike, sign, traffic light, etc.	Known classes of objects detectable with current model operations
Temporal	BEFORE, AFTER	Allows for detection of sequences of events
Spatial	LEFT, RIGHT, FRONT, BACK	Geometric position of known objects in frame.

4.4.2 Predicate Functions

In this section we describe each operator. For reference of which operators use different underlying computer vision operations, see Table 4.2.

- **Front(object)**: checks to see whether an object is in-frame (object being a particular object class). The simplest operator in our library, which is just checking for the existence of a bounding box on a particular frame.

- **Front(object_1, object_2)**: checks whether one object is in front of another using average depth contained within the bounding box. We take the depth of all pixels in the bounding box, create two clusters (assuming an object and a background) and compare the average of the two nearer clusters.

- **Left(object_1, object_2)**: an operator to see whether two objects are in-frame at the same time, and one is geometrically to the left of the other.

- **Left(ego, object)** or **Right(ego, object)**: we wish to see if an object is to the left or right of the ego vehicle. To do this, we find the nearest lane boundary on each side, presuming this is an unambiguous marker of geometric correctness, and

- **In(object, segmentation_class)**: identity whether a particular bounding box is entirely or nearly surrounded by a particular segmentation class

- **Moving(object, direction)**: identify whether a bounding box has majority movement in a particular direction.

- **ego.maneuver = <maneuver>**: identify whether the vehicle from which video is being recorded is performing a particular maneuver, of which we briefly describe our technique below

4.4.3 Vehicle Maneuver Detection

We use a previously-published vehicle maneuver detection scheme as one of our operators. In the simplest cases, we detect simple vehicle maneuvers (left turn, right turn, U-turn, and K-turn) using estimates of the camera motion. We compare trajectory segments against reference maneuvers using dynamic time warping to compute a distance measure. In the simplest case, we can then check the distance measure against a known threshold. For validation we can check against all other distance measures in the video for that maneuver, and against other maneuvers types. For the purpose of this study, we assume the camera motion is precomputed and we are only computing the DTW results on-demand.

4.4.3.1 Deceleration

Our maneuver matching technique relying solely on X-Y position does not give any information about speed because DTW stretches maneuvers in time. For detecting turns, this property is useful, because a left turn can happen at any speed. Other maneuvers, however, can be defined in terms of a change in speed, such as decelerating to stop at an intersection. We utilize speed as an additional component of measurement and search for maneuvers in the X-Y-S space.

4.4.3.2 Lane Line Hough Transform

We wish to identify the position of the car relative to the lanes, so we use the position of the lane line at the base of the frame. To determine this position, we perform a Hough transform [19] on the lane line segmentation output, and then use the position of the center of the line. If the lane line continues off the left or right edge of the image, we project it linearly to determine the coordinate corresponding to the baseline of the image (which is why negative pixel values can arise for the detected lane position, as in Figure 3.2). We thus obtain a three-dimensional coordinate to represent the camera's position relative to the lanes for each frame.

4.4.3.3 Highway Entrance/Exit

Identifying highway merge and exit events directly with our maneuver matching technique is challenging, as the length and shape of highway entrance and exit ramps varies considerably. Detecting highway transitions using changes in speed or acceleration is not viable, as our experiments showed motion estimation does not perform well on highway environments. To detect highway merges and exits, we use a simple image classifier and k-means sorting. We use transfer learning on an Inception V3 network with two categories: "highway" (n=287) and "surface streets" (n=585) with a 10% validation set and find the model achieves 94% accuracy after 1000 training steps. To identify highway merges and exits, we consider time windows of 180 seconds in length. We take the average of the first 60 and last 60 seconds, as the actual on- or off-ramp transition is often metastable.

4.5 Video Query Execution

4.5.1 Query Tree Generation

We construct the query tree given the input text of a query. The tree is constructed by looking for parentheses surrounding predicate phrases. Each set of parentheses is assumed to be a child node or subset of nodes in the query, connected to a parent "and" or "or" node. We form the query by initializing an "or" node as the root node, and then by pushing operators onto a stack as we find them, and recursively generating a new subtree each time we encounter a new set of parenthesis containing multiple predicates. Ultimately we create a single tree where operator nodes are demarcated from "joining" nodes ("AND" and "OR" nodes). Operator nodes are leaves in the tree, and contain one or more computer vision operations along with the arguments or the comparison tokens to be used when calling the operations.

4.5.2 Tree Search Algorithms

4.5.2.1 Lowest-Cost Search (LC)

We choose the node (operator instance) in the tree with the expected lowest cost based on latency of loading and executing the operations. Choosing the lowest-cost node is not the optimal choice in all circumstances, though for small queries this strategy performs quite well.

4.5.2.2 False Low Bound (FLB)

For each node, we can consider the cost of evaluation based on whether an individual frame is true or false. For a typical query and video library, we expect the common case will be most frames resolving as false for most operators (though obviously this is speculative and could vary depending on the dataset). For a given node, we can calculate the lowest-possible cost of evaluating that node if every unknown frame in every child resolved to false. We call this calculation the "false low bound". (A corresponding "true low bound" is similarly easy to calculate, but much less useful.) For leaf operators the cost of resolving a frame to true or false is identical, but FLB is useful for joining nodes with multiple children. Intuitively, a frame is resolved at an "AND" node, for example, as soon as one of the child nodes resolves that frame to false, because any logic sentence conjoined with "false" must be false.

Once we compute the cost at each leaf node, we can easily compute the FLB for each parent node in the tree hierarchy:

- FLB for 'OR' Node is the sum of all unexecuted child operators over all unknown frames
- FLB for 'AND' Node is the minimum of all unexecuted child operators over all

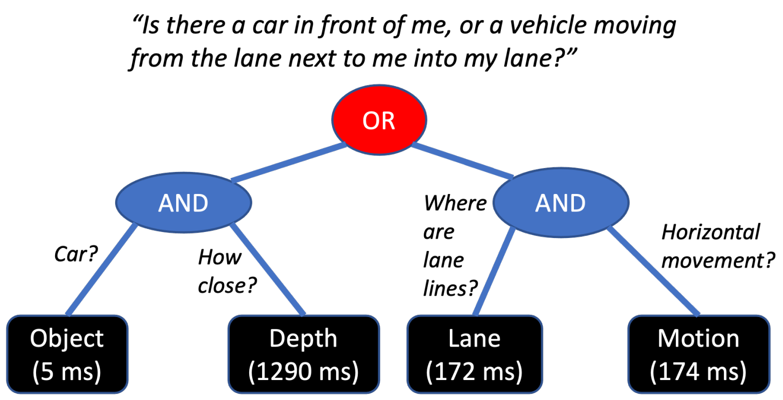


Figure 4.2: A simple example of how choosing the lowest cost operation is not necessarily the best choice. Leaf nodes (in black) are computer vision operations with a specific latency. If we choose the operation with latency (cost) 2, we will next need to evaluate the operation with latency 14. A better choice is to start with the operation with cost 6.

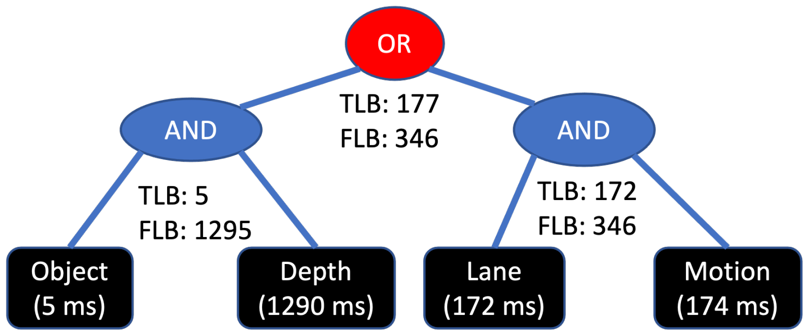


Figure 4.3: Example False Lower Bound and True Lower Bound calculations for each node in a query.

unknown frames

We calculate FLB for each node at each iteration in the query process, and then choose the node with the lowest FLB to process next.

4.5.2.3 Expected Contribution to Root (ECR)

We can choose a node by the impact resolving its unknown frame vector will have on the unknown frame vector of the top level node. For example, a child node of the root will have more impact than a descendant deeper in the tree, because the descendant's results will traverse other joining nodes on the way up the tree hierarchy to the root. Obviously we cannot know the results in advance, but can make intelligent guess with only one parameter: expected true frame rate for a given operator. For each joining node traversed to the root which has other children, we naively assume those children could potentially feed a result impacting whether an individual frame resolves to true or false. We estimate that each child is equally likely to affect its parent, in other words, we estimate the probability that a specific child resolves its parent is $\frac{1}{c(u)}$, where $c(u)$ is the number of children of vertex u. We want a heuristic $H : V \to \mathbb{R}$ that gives us an estimate of how important the leaf vertex v is to evaluate. We define this heuristic as follows:

$$\mathrm{H(v)} = \left[n_f(1-p) \prod_{u \in OR(v)} \frac{1}{c(u)} + n_f p \prod_{u \in AND(v)} \frac{1}{c(u)} \right] / \left[c + n_f d \right]$$

Where:

- n_f is the number of frames.
- p is an estimated probability that a frame resolves to true.
- $OR(v)$ gives all of the OR nodes on the path from the root to vertex v in the tree. $AND(v)$ does the same for AND nodes.
- $c(u)$ gives the number of children of vertex u.
- c is the fixed cost of running the associated ML model.
- d is the cost of inference on a single frame

The calculation is thus performed as follows: we find the cardinality of the unknown frame vector for a particular node. Then, for the subset of frames expected to be false, we find the "OR" nodes to be traversed on the path to the root. For each we divide by the total number of children of the node (effectively assuming each child has an equal contribution). We do not need to do this for "AND" nodes because a frame resolved to false in one child does not need to be resolved in the other children. For the subset of frames expected to be true, we do a similar function except only for "AND" nodes.

4.5.2.4 Frame-by-Frame (FBF)

For a worst-case benchmark comparison, we compute the cost of executing every operator contained in the query. Importantly this still includes frame skipping and utilizes Scanner's internal optimizations to reduce the overall computational time.

4.5.3 Caching

In many queries the same computer vision model is used multiple times with different unknown frame vectors. We wish to reduce the latency of our operator execution by caching intermediate results for certain frames. We thus have two simple caching strategies depending on the type of operation:

4.5.3.1 Value Cache (VC)

Certain computer visions operations reduce the amount of data by orders of magnitude relative to input. Lane-line detection, for example, reduces a 1080x1920 8-bit image to 20 floating-point values: four orders of magnitude less data. For operations that follow this execution paradigm, we place these results in a hash table. The cache has a "get" and "put" function for each computer vision operation. The input to the "get" function is a list of frames which are checked against a hash table of previously resolved frames for that operation, and if a frame has been resolved the result vector is returned. The input to the "put" function is the array of result vectors which are inserted into the hash table.

4.5.3.2 Image Cache (IC)

Some operations do not reduce the size of the input data at all, effectively producing an output frame of similar size and resolution to each input frame. Examples include pixel-level estimates of motion and depth. To “cache” this data, we leverage Scanner’s ability to store and load columns of video frames, keeping the output data in Scanner and then loading it as needed without calling the expensive machine learning operation that produced it. We keep a map of frame IDs in our system to match with the sequential frame storage of Scanner and retrieve frames when needed. From Scanner we load all frames needed for the current operation into memory and pass this data back to our post operation function. We track the latency of this operation and include this in the total query latency.

4.5.4 Frame Vector Tracking

Each node in the tree has three lists of frames: a list of unknown, true, and false frames. When a node is selected to run, we choose to resolve all unknown frames to attain the lowest-possible amortized cost of the function. The function runs and returns a result vector with a Boolean value for each frame.

We then update each node on the path upward from the node to the root of the tree. For each node along this path, the list of true or false frames depends on the type of node:

- For “AND” nodes, the true frame vector is the ‘logical and’ of the set of all child true frame vectors. The false frame vector is the ‘logical or’ of the set of all child false frame vectors.
- For “OR” nodes, the true frame vectors is the ‘logical or’ of the set of all child true frame vectors. The false frame vector is the ‘logical and’ of the set of all child false frame vectors.

Each node's unknown frame vector is then the union of the true and false frame vectors subtracted from the set of all frames. Once we have reached the root, we then proceed in a breadth-first order, copying the unknown frame vector from the parent node to each child node. From this new unknown frame vector we subtract out each child's true and false frame vector (since each child node may have frames resolved in these vectors which are not tracked by the parent). In this way we keep an accurate list of all frames yet to be resolved in each node following each function.

4.6 An End-to-End Video Query System

Using the ideas in Sections 4.4 and 4.5, we built an end-to-end video processing pipeline. We use Scanner, a high-performance video-processing pipeline system, as the system to execute each query. Scanner integrates open-source software libraries such as ffmpeg, grpc, and protobuf to create a distributed video "database" capable of executing pipelines of arbitrary operations. This allows us to focus our experiments and measurements on the query processing and execution time independent of the many variables inherent to video processing, as Scanner has already been optimized to load, decode, and process video frames. On the front-end, we drive Scanner's API to create the operations we need after decomposing a query, and on the back-end we create our own operations for each machine learning kernel we wish to run. Additionally we create operations to handle the "post-operation" operations, such as identifying whether a bounding box lies to the left of a lane line.

4.6.1 System Components

While Scanner forms the core of the execution pipeline, additional components are required as part of the end-to-end system.

At the input step, we have a tree farmer, which takes the input text query and generates the query tree and initializes all node variables to prepare for query execution.

The tree solver finds the next node to be executed based on the search rules described in Section 4.

When considering executing a given node, the tree solver calls the cost estimator, which returns the estimated cost of loading the model and the per-frame cost given the cardinality of the unknown frame vector. The frame vector updater handles copying up frame vector to predecessor nodes after an operation has completed, and then distributing new unknown vector of frames to each node in the tree.

The Scanner driver loads and runs Scanner using the API to build pipelines of built-in and custom operations. These Scanner operations are custom code written to call and execute the machine learning models we obtained from open-source repositories or designed for this system (i.e. vehicle maneuver detection).

Following the execution of the underlying computer vision operation(s), the post-operation functions takes the raw output from each model that requires further processing to obtain the answer to the operation we are interested in. Typically the latency for this function is low compared to the model inference time. Once result values have been obtained, any unknown values are loaded into the result cache, which can then return results as needed by matching to the input frame vector.

Algorithm 4.6.1: System Execution

```
input query_sentence
input video_list
Parse query_sentence into Tree
for video in video_list:
    unknown frame vector ← list of video frames
    while  (unknown frame vector at Tree root
            node is not empty)
        n ← unexecuted node in Tree with lowest
```

```
                cost (by chosen search metric)
        for operation in n:
            Fetch cached results
            operation frame vector =
                unknown frame vector
                - frames found in cache
            Run Scanner operation
            Cache ← new frames from Scanner
            Run Post-Operation Function
            for node p from n to root:
                Update unknown frame vector of p
            for node in Tree:
                Copy down unknown frame vector
                     from parent node
                unknown frame vector -=
                    (true frame vector +
                    false frame vector)
            for node in Tree:
                Update False Lower Bound
                Update ECR
            node.is_executed ← True
output result vector
```

4.6.2 Execution

The primary execution loop of the system is shown in Algorithm IV.1. We begin with a text query and list of videos to process. The query is parsed into a tree, and then for each video we begin the main loop of processing the tree. We first select the node with lowest cost according to whichever search metric we are using (LC, ECR, FLB, etc). Once we have chosen a node to execute, we fetch any frames that we can from the cache before running the operator in Scanner on the rest of the unknown frames. We then execute the post-operation function, after which we update the frame vectors and costs.

4.6.3 Supporting Components

For the purpose of tree evaluation, we build several components to simulate the expensive inputs and outputs of the system, namely the queries (which would normally need to be written by a human) and frame-by-frame results vector (which would need to run the operations on a particular dataset in real time).

- **Query Generator**: The query generator exists to sample random valid trees from the set of all possible trees. We use two parameters: a starting "seed" probability of the current node being a leaf node, and a "decay" value for the seed at each iteration. By sampling from a list of valid operations and keywords, this component can rapidly create trees of small or large size. For our experiments, we report average parameters of each tree such as number of predicate nodes.

- **Results Simulator**: For the purpose of evaluation, we can generate a Boolean value for each frame based on several configurable parameters: a percentage of frames that are True for each operation, an option to generate clusters or purely random distributions of True frames, and, optionally, the percentage of overlap that should exist between operations.

- **Cost Calculator**: Each operator has an empirically-measured fixed cost and variable (per frame) cost for a particular hardware configuration. While this does not necessarily capture the full range of possible complexity, we nonetheless assume for the purpose of conducting our experiments that this is a known cost due to the fact that the GPU-centric operation will tend to dominate the cost of loading and decoding data.

- **Timer/Logger**: Allowing measurements of execution time for specific operations, and logging informational and error messages.

4.7 Methodology and Evaluation

4.7.1 Testing Configuration

Our software stack was run on an Intel Core i7 workstation with an NVIDIA Titan Xp GPU. This workstation ran Tensorflow 1.12 with CUDA 9.0 on Ubuntu 18.04. Scanner and all externally-developed machine learning models were procured from GitHub repositories. We ran each on test data to validate correctness, and retrained several as described in Section 4.3. We then built the Scanner operations and verified correctness again, and verified runtime correctness a third time upon using the Scanner API from our software stack. The vehicle maneuver software stack was built by the author.

The computer vision operations used are not state-of-the-art classifiers because software limitations required us to use CUDA 9.0 on our test hardware. (Adjusting a model to use a newer or older version of CUDA is quite time-consuming.) In the field of deep learning where iterative improvements produce rapid results, it may seem as though our work is significantly behind. However we note that our goal was not to procure the most accurate models to test, but to procure models which have realistic characteristics to plausibly be used in such a system. In other words, each model has attributes such as deep learning networks run on a GPU, non-trivial inference latency, and realistic input and output. The novel aspects of our system are compatible with newer deep learning systems.

4.7.2 Objectives and Measurements

Our goal for system evaluation is to show plausible workloads, explore the capability of the query language and query tree execution structure, and evaluate different execution search strategies. Our principal measurement for evaluation is latency, specifically the end-to-end runtime of Scanner running our custom video processing pipelines, as Scanner has already been optimized for data loading, processing, and movement. Scanner is capable of cluster processing, however our goal is not to hide latency but to measure and address it directly and so this is not useful for our current evaluation but and important capability for real-world systems. Therefore, we did not use cluster processing.

We are interested in the latency characteristics of the deep learning models relative to the amount of input data. Understanding the latency characteristics allows us to conduct experiments more rapidly, as knowing what each operation's latency will be means we do not need to actually run it multiple times. This allows us to perform rapid evaluations on large numbers of queries.

We show the results of these measurements in Figure 4.4. We note that beyond an initial load latency for the model (including loading Scanner libraries, initial preprocessing of the video, and copying the model to GPU), the latency is linear with respect to the number of frames.

We use a library of dashboard video camera recordings for evaluation. The 13 videos used for evaluation total 468 minutes of driving footage (about 786,000 individual frames). Scored maneuvers account for a total of 1.7 hours (about 22%) of video. Each video has resolution of 1920 by 1080 pixels, 29.97 FPS. Individual frames as resized, as necessary for input to each operation.

Table 4.4: Operation Runtime Regression Analysis

Model	**Initial Latency (seconds)**	**Per-Frame Amortized Latency (milliseconds)**	**R^2 Value**
Faster-RCNN	15.37	51	0.9963
LaneNet	14.88	174	.9992
RAFT	13.39	217	.9991
DeepV2D	20.09	1295	.9998

Table 4.5: Auto-Generated Tree Sizes

Tree Size	**Seed**	**Decay Factor**	**Number of Predicates**
Small	0.5	0.7	4.4
Medium	0.7	0.87	11.4
Large	0.8	0.9	29.4

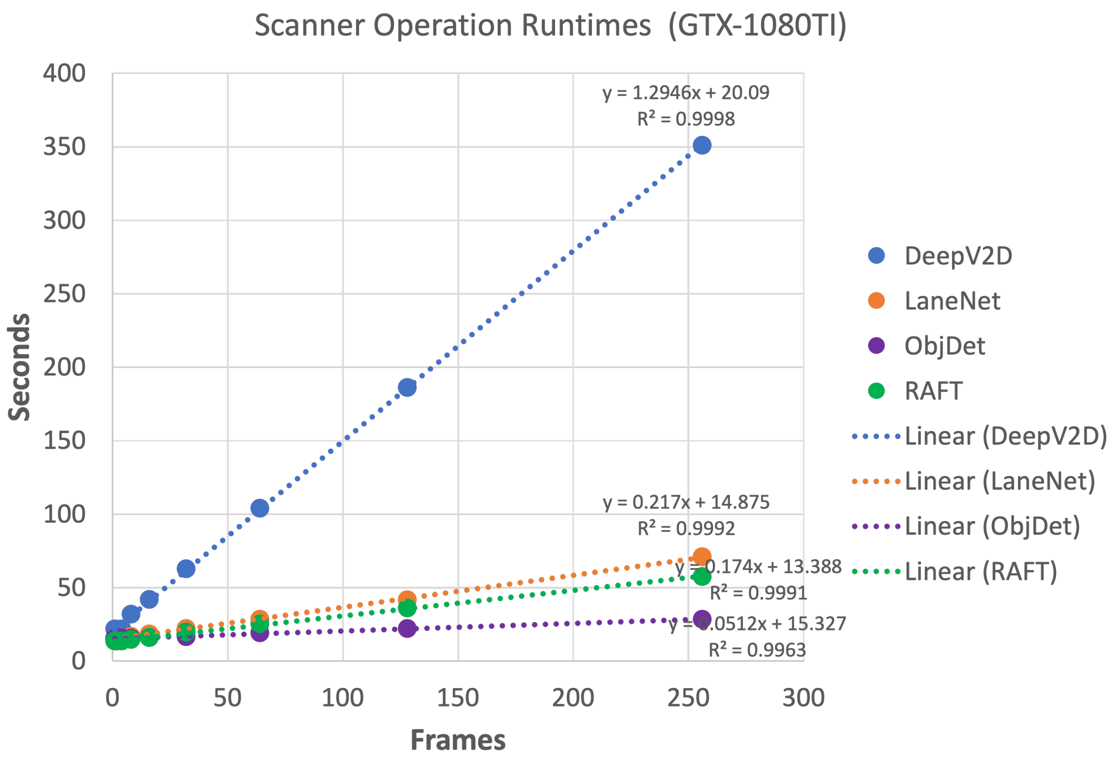


Figure 4.4: We observe a linear effect with respect to processing time for each model following an initial warmup period.

Table 4.6: Handwritten Queries

Query	**Description**
(FRONT(pedestrian,truck) AND MOVING(truck,away)) OR (FRONT(stop sign,truck) AND MOVING(truck,away))	“A truck moving toward a pedestrian or a stop sign”
(ego.maneuver=left_turn AND LEFT(ego,bicycle)) AND (FRONT(traffic_light) AND MOVING (bicyle,toward))	“The vehicle is making a left turn, with a bicycle on the left moving toward the vehicle, and a traffic light visible in the frame.”
(MOVING(pedestrian,right) AND LEFT(ego,pedestrian)) OR (RIGHT(ego,pedestrian) AND MOVING(pedestrian,left))	“Pedestrians on the left and right moving toward the vehicle.”
(FRONT(stop_sign,bus) AND LEFT(ego,pedestrian)) OR (FRONT(traffic_light,bus) AND MOVING(pedestrian,right))	“A bus approaching a stop sign or traffic light with a pedestrian on the left moving to the right.”
(LEFT(ego,pedestrian) OR LEFT(ego,truck) OR LEFT(ego,train) OR LEFT(ego,car) OR LEFT(ego,bicycle) OR LEFT(ego,motorcycle))	“A pedestrian, truck, train, bicycle, motorcycle, or car on the left of the vehicle.”
(LEFT(ego,truck) AND MOVING(truck, right) OR LEFT(ego,train) AND MOVING(train, right) or LEFT(ego,car) AND LEFT(ego,bicycle) AND LEFT(ego,motorcycle))	“A truck, train, or car on the left moving toward the vehicle.”
(FRONT(bus,stop_sign) OR FRONT(bus,traffic_light)) AND (LEFT(ego,pedestrian) OR LEFT(ego,bus))	“A bus facing a stop sign or traffic light with a pedestrian to the left of the vehicle or the bus.”
(FRONT(stop_sign, bus) AND MOVING(bus,toward)) OR (FRONT(stop_sign, bus) AND MOVING(bus, away))	“A bus in front of a stop sign while moving forward.”
(ego.maneuver=stop AND FRONT(traffic_light) AND LEFT(bicycle,ego) AND MOVING(bicycle,toward))	“A bicycle to the left moving toward the vehicle, while the vehicle is stopped in front of a traffic light.”
((FRONT(cat) OR FRONT(dog)) AND maneuver.ego=slowing_down AND NOT (FRONT(stop_sign) OR FRONT(traffic_light))	“The vehicle slows down while approaching a cat or dog, and not for a stop sign or traffic light.”

4.7.3 Query and Result Simulation

We use our query generator to sample random valid trees from the set of all possible trees. We use the "seed" and "decay" values to create three different configurations of trees: "small", "medium", and "large". We show statistics about these automatically generated trees in Table 4.5. Small trees contain a mean value of 4.4 predicates each, roughly the smallest size of a query that can plausibly benefit from a non-trivial execution plan. Medium trees contain 11.4 predicates and large trees contain a mean value of 29.4.

Table 4.7: Query Execution Time Breakdown

Tree Size	**Tree Search and Frame Vector Tracking Time (percent)**	**Scanner Operation Time (percent)**	**Post-Operation Time (percent)**
Medium	0.05 seconds (0.00%)	16 minutes, 20 seconds (99.97%)	3.2 seconds (0.03%)
Medium	0.1 seconds (0.00%)	20 minutes, 1 seconds (99.97%)	3.7 seconds (0.03%)
Large	0.4 seconds (0.00%)	21 minutes, 28 seconds (99.96%)	5.2 seconds (0.04%)

Additionally we have a results simulator which (as noted in Section 4.6) can generate random or clustered boolean values at a particular frequency to stress-test our query execution strategies. Together these modules allow us to test a large number of queries very quickly. We do not actually execute each operation for evaluation, instead we use the results found in Section 4.7 to estimate the load and execution cost for each operation. Since $r^2 > .999$ for the latency of each of these systems with respect to the number of frames, this is a very accurate estimation, and allows us to much more easily run an evaluation of our end to end pipeline.

We test results on sets of 100 random queries from each size. Additionally, we use

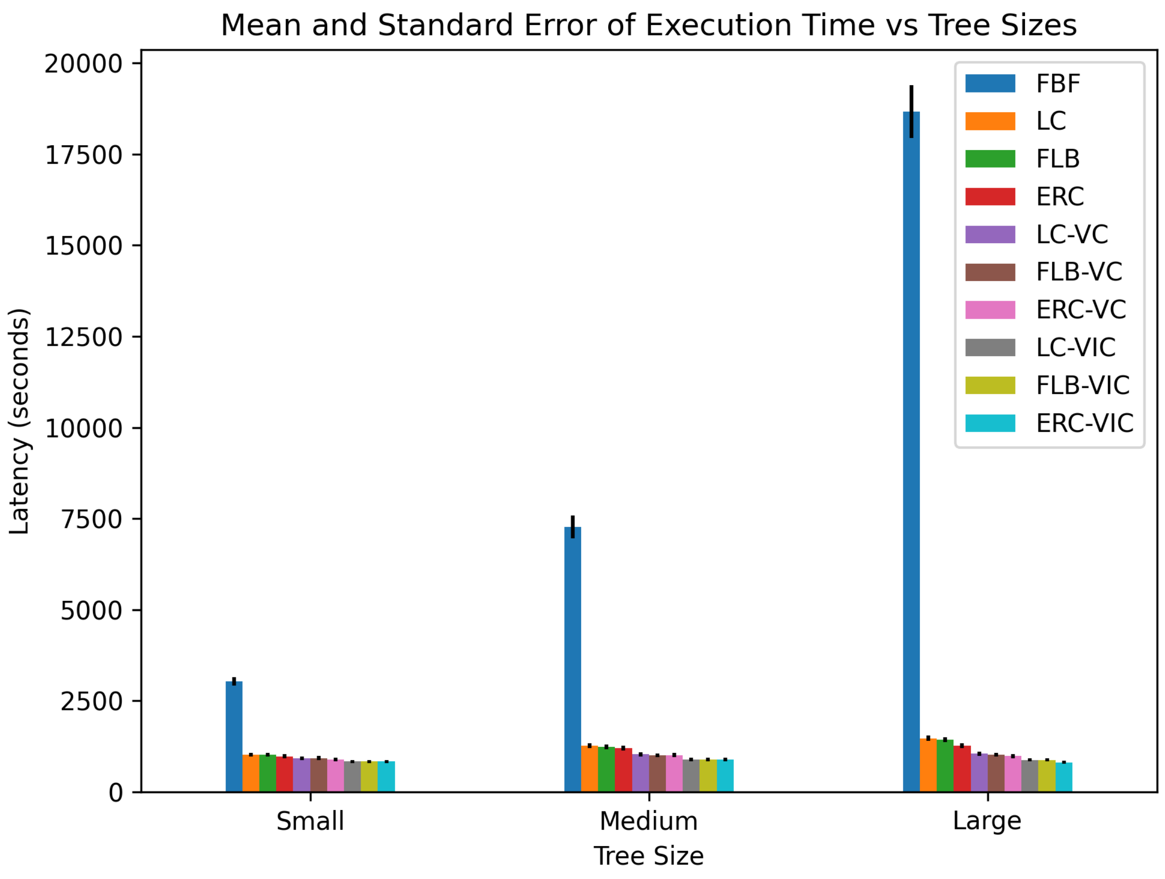


Figure 4.5: Mean Execution Time vs Tree Sizes

a set of 10 hand-written queries that test a range of operators and are designed to be plausible uses of such a system based on our conversations with industry experts.

4.8 Results

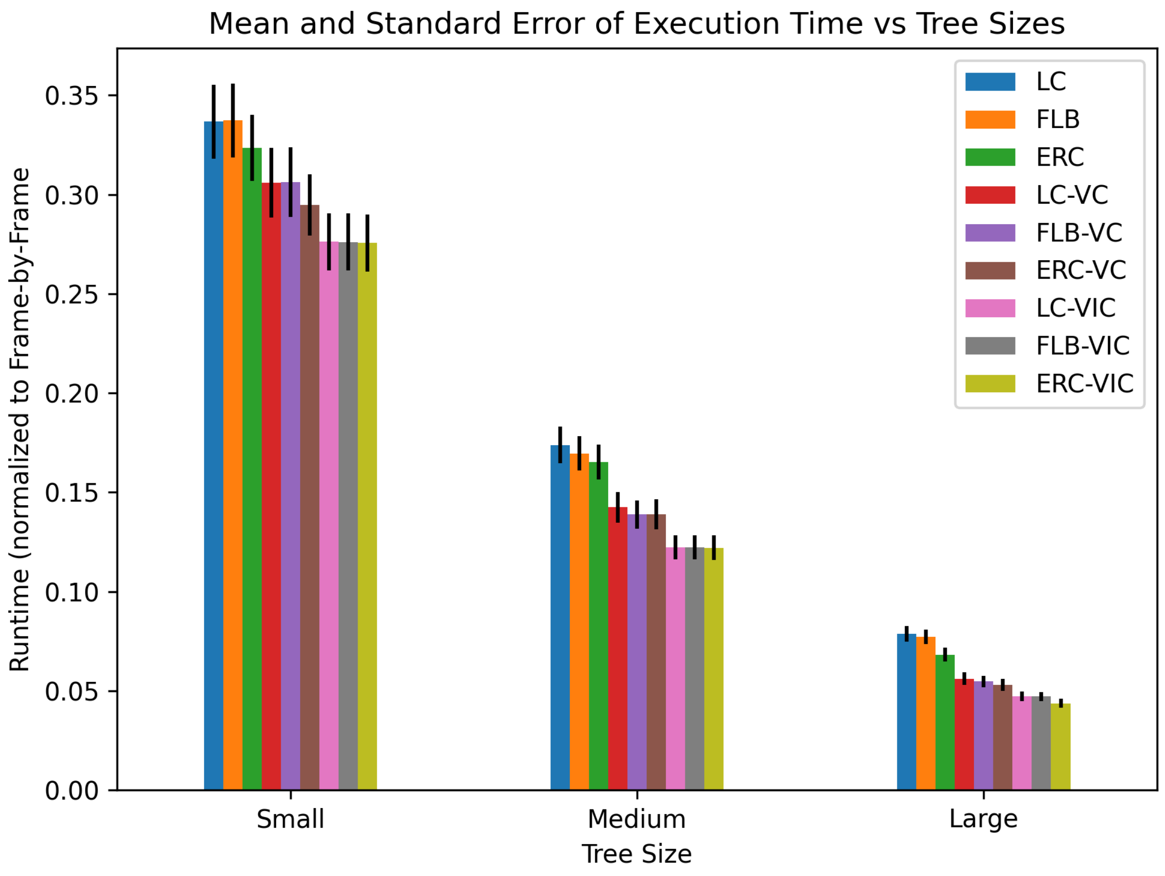


Figure 4.6: Execution time for search strategies normalized to frame-by-frame execution time

Table 4.8: Descriptive Statistics

	Small			**Medium**			**Large**		
Search	Avg. Runtime	Normalized Runtime	Avg. Nodes Explored	Avg. Runtime	Normalized Runtime	Avg. Nodes Explored	Avg. Runtime	Normalized Runtime	Avg. Nodes Explored
FBF	4137.0	n/a	n/a	10640.3	n/a	n/a	27286.3	n/a	n/a
LC	1423.6	0.344	2.50	1825.9	0.172	4.33	2106.4	0.077	7.61
FLB	1430.8	0.346	2.49	1812.9	0.170	4.32	2090.2	0.077	7.45
ERC	1387.8	0.335	2.41	1743.8	0.164	**3.92**	1841.7	0.067	7.55
LC-VC	1264.8	0.306	2.50	1508.4	0.142	4.33	1525.5	0.056	7.61
FLB-VC	1271.3	0.307	2.49	1495.7	0.141	4.30	1513.5	0.055	7.61
ERC-VC	1238.3	0.299	**2.43**	1473.6	0.138	4.13	1437.4	0.053	**7.12**
LC-VIC	1177.9	0.285	2.50	1297.1	0.122	4.33	1280.8	0.047	7.61
FLB-VIC	1178.0	0.285	2.49	1295.0	0.122	4.31	1279.3	0.047	7.58
ERC-VIC	**1162.9**	**0.281**	**2.43**	**1277.7**	**0.120**	4.20	**1168.2**	**0.043**	7.25

4.8.1 Overall Results

We show our overall results of execution time for 100 randomly generated small, medium, and large trees in Figure 4.5. This plot compares the latency for ten total combinations of query execution and caching: frame-by-frame search, and cached and non-cached versions of Lowest Cost, False Low Bound, and Expected Root Contribution. We use two caching strategies to take advantage of data reuse following classification: Value Cache (VC) and both Value and Image Cache (VIC). We show descriptive statistics in Table 4.8.

We find having a high number of predicates improves the opportunity for optimization as well as caching. For the smallest set of queries, we achieve a 3.6x speedup for the best-performing strategy. We find that while the speedup within optimizations is much smaller, we nonetheless achieve a 1.2x speedup over the lowest-cost model for these small queries.

As the number of predicates grows, we find higher speedup, especially as caching helps when the same operation is performed repeatedly in the same query. We find a 1.4x speedup over lowest-cost for medium trees, and 1.8x for large trees. Relative to frame-by-frame search, the speedup is 8.3x for medium trees and 23x for large trees.

We find, as expected, all search operations require significantly less time than frame-by-frame search when measured by latency. ERC outperforms LC and FLB in all caching conditions and in both clustered and randomly-distributed results, but the margin is typically quite small (about 2%-4%) Both the Value Cache and Value + Image Cache reduce execution time substantially.

4.8.2 Handwritten Queries

We created a list of handwritten queries, shown in Table 4.6, to compare with the randomly-generated query results. These queries test simple and straightforward human-semantic questions and contain fewer predicates than the randomly-generated

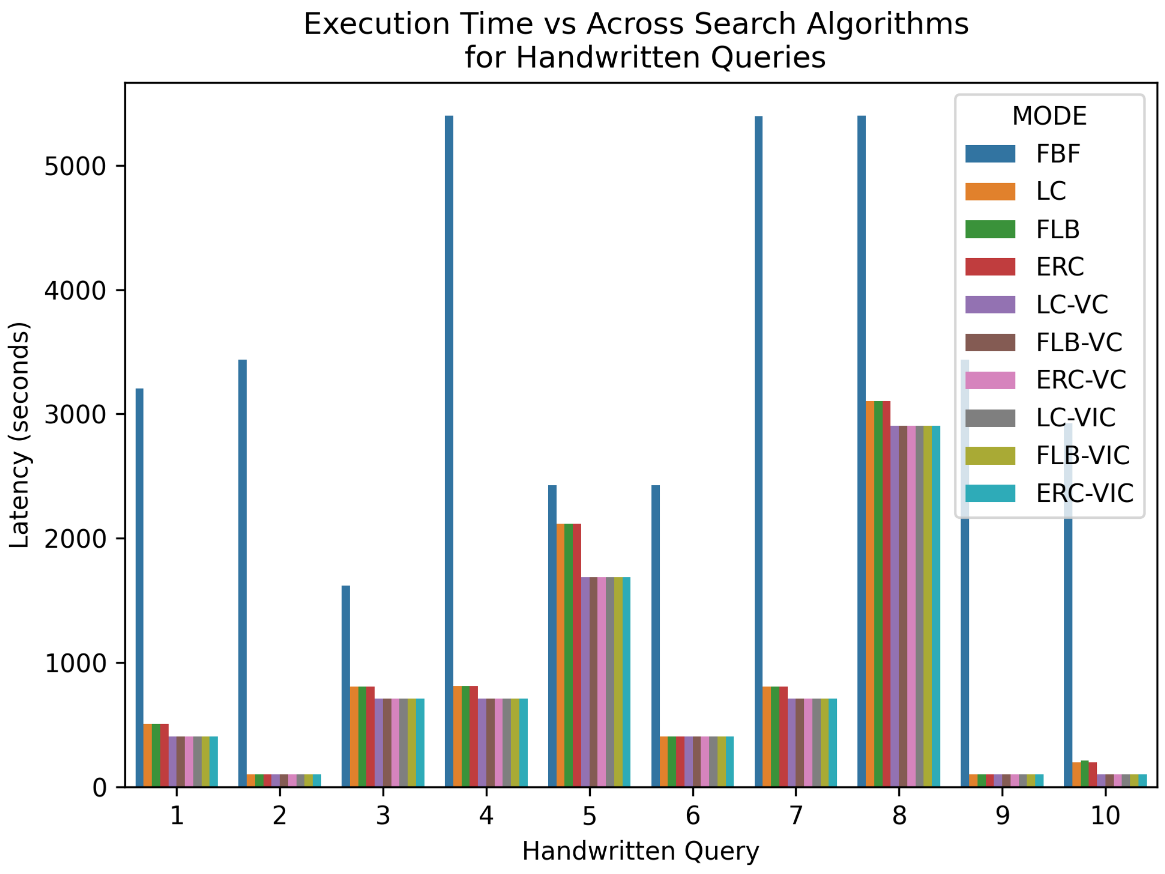


Figure 4.7: Execution Time for Search Algorithms on Handwritten Queries

queries. Nonetheless our query optimizer provides speedup to each relative to the frame-by-frame baseline as shown in Figure 4.7. We find the caches provide additional speedup relative to the cost estimation strategies. From this we conclude that even for simple queries the strategy of performing cost estimation and caching intermediate results can provide speedup.

4.9 Conclusion

We designed and evaluated an end-to-end video query processing system which dynamically builds video processing pipelines and performs optimizations based on the latency (cost) of executing different models. We designed and demonstrated a query language which allows the user to specify search predicates. We evaluated this system with a library of dashboard camera videos and computer vision models to search for the presence, locations, and movement of other objects in the frame. Overall we found a 3.6x speedup over frame-by-frame search and 1.2x speedup for the best query optimization technique ("estimated root contribution") compared to cost-based optimization.

CHAPTER V

Conclusions

The challenges of developing and validating autonomous vehicle systems has led to the need to curate large video libraries documenting the performance of the vehicle under a variety of conditions. Simultaneously, computer vision models continue to advance in multiple domains, leading to a need for techniques and software systems to build end-to-end systems of these models to help search these libraries.

This dissertation presented several techniques to address the challenges of searching video libraries in the context of autonomous vehicle development. The first part focused on the idea of using camera motion estimation and time-invariant signal similarity to search for vehicle maneuvers in dashboard camera video. The second part focused on the idea of a processing pipeline for vehicle state and motion, and extended the idea of using time-invariant matching with other computer vision techniques to detect lane changes, acceleration, and highway driving. The last part focused on a novel visual query optimization technique and demonstrated the technique in an end-to-end video processing system with a declarative query language and domain-specific computer vision models.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Ahmed, S. A., D. P. Dogra, S. Kar, R. Patnaik, S.-C. Lee, H. Choi, G. P. Nam, and I.-J. Kim (2019), Query-based video synopsis for intelligent traffic monitoring applications, *IEEE Transactions on Intelligent Transportation Systems*, *21*(8), 3457–3468.

[2] Anderson, M. R., M. Cafarella, G. Ros, and T. F. Wenisch (2019), Physical representation-based predicate optimization for a visual analytics database, in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pp. 1466–1477, IEEE.

[3] Bansal, M., A. Krizhevsky, and A. Ogale (2018), Chauffeurnet: Learning to drive by imitating the best and synthesizing the worst, *arXiv preprint arXiv:1812.03079*.

[4] Bloom, B. H. (1970), Space/time trade-offs in hash coding with allowable errors, *Commun. ACM*, *13*(7), 422–426, doi:10.1145/362686.362692.

[5] Cao, J., K. Sarkar, R. Hadidi, J. Arulraj, and H. Kim (2022), Figo: Fine-grained query optimization in video analytics, in *Proceedings of the 2022 International Conference on Management of Data*, pp. 559–572.

[6] Chao, D., N. Koudas, and I. Xarchakos (2020), Svq++: Querying for object interactions in video streams, in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pp. 2769–2772.

[7] Chen, Y., X. Yu, and N. Koudas (2020), Tqvs: Temporal queries over video streams in action, in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pp. 2737–2740.

[8] Chen, Y., X. Yu, N. Koudas, and Z. Yu (2021), Evaluating temporal queries over video feeds, in *Proceedings of the 2021 International Conference on Management of Data*, pp. 287–299.

[9] Dang, H., and J. Fürnkranz (2018), Using past maneuver executions for personalization of a driver model, in *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, pp. 742–748, IEEE.

[10] Daum, M., B. Haynes, D. He, A. Mazumdar, and M. Balazinska (2021), Tasm: A tile-based storage manager for video analytics, in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pp. 1775–1786, IEEE.

[11] Eren, H., S. Makinist, E. Akin, and A. Yilmaz (2012), Estimating driving behavior by a smartphone, in *2012 IEEE Intelligent Vehicles Symposium*, pp. 234–239, IEEE.

[12] Geiger, A., P. Lenz, and R. Urtasun (2012), Are we ready for autonomous driving? the kitti vision benchmark suite, in *Conference on Computer Vision and Pattern Recognition (CVPR)*.

[13] Geirhos, R., D. H. Janssen, H. H. Schütt, J. Rauber, M. Bethge, and F. A. Wichmann (2017), Comparing deep neural networks against humans: object recognition when the signal gets weaker, *arXiv preprint arXiv:1706.06969*.

[14] GitHub user 'MaybeShewill' (2018), Github: lanenet-lane-detection, https://github.com/MaybeShewill-CV/lanenet-lane-detection, accessed: 2019-04-14.

[15] Guo, J., Y. Liu, L. Zhang, and Y. Wang (2018), Driving behaviour style study with a hybrid deep learning framework based on gps data, *Sustainability*, *10*(7), 2351.

[16] He, K., G. Gkioxari, P. Dollár, and R. Girshick (2017), Mask r-cnn, in *Proceedings of the IEEE international conference on computer vision*, pp. 2961–2969.

[17] Houenou, A., P. Bonnifait, V. Cherfaoui, and W. Yao (2013), Vehicle trajectory prediction based on motion model and maneuver recognition, in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 4363–4369, IEEE.

[18] Ibrahim, N., P. Maurya, O. Jafari, and P. Nagarkar (2021), A survey of performance optimization in neural network-based video analytics systems, *arXiv preprint arXiv:2105.14195*.

[19] Illingworth, J., and J. Kittler (1988), A survey of the Hough transform, *Computer Vision, Graphics, and Image Processing*, *44*(1), 87–116, doi: https://doi.org/10.1016/S0734-189X(88)80033-1.

[20] Johnson, D. A., and M. M. Trivedi (2011), Driving style recognition using a smartphone as a sensor platform, in *2011 14th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, pp. 1609–1615, IEEE.

[21] Juang, B.-H. (1984), On the hidden markov model and dynamic time warping for speech recognition—a unified view, *AT&T Bell Laboratories Technical Journal*, *63*(7), 1213–1243.

[22] Júnior, J. F., E. Carvalho, B. V. Ferreira, C. de Souza, Y. Suhara, A. Pentland, and G. Pessin (2017), Driver behavior profiling: An investigation with different smartphone sensors and machine learning, *PLoS one*, *12*(4), e0174,959.

[23] Kakkar, G. T., et al. (), Eva: An end-to-end exploratory video analytics system, *ArXiv*.

[24] Koudas, N., R. Li, and I. Xarchakos (2020), Video monitoring queries, *IEEE Transactions on Knowledge and Data Engineering*.

[25] Long, J., E. Shelhamer, and T. Darrell (2015), Fully convolutional networks for semantic segmentation, in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3431–3440.

[26] Martinez, C. M., M. Heucke, F.-Y. Wang, B. Gao, and D. Cao (2018), Driving style recognition for intelligent vehicle control and advanced driver assistance: A survey, *IEEE Transactions on Intelligent Transportation Systems*, *19*(3), 666–676.

[27] Martinsson, J., N. Mohammadiha, and A. Schliep (2018), Clustering vehicle maneuver trajectories using mixtures of hidden markov models, in *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, pp. 3698–3705, IEEE.

[28] Mendoza, D., F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis (2021), Interference-aware scheduling for inference serving, in *Proceedings of the 1st Workshop on Machine Learning and Systems*, pp. 80–88.

[29] Moll, O., F. Bastani, S. Madden, M. Stonebraker, V. Gadepally, and T. Kraska (2022), Exsample: Efficient searches on video repositories through adaptive sampling, in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pp. 2956–2968, IEEE.

[30] Mur-Artal, R., and J. D. Tardós (2017), Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras, *IEEE Transactions on Robotics*, *33*(5), 1255–1262.

[31] Mur-Artal, R., J. M. M. Montiel, and J. D. Tardos (2015), Orb-slam: a versatile and accurate monocular slam system, *IEEE transactions on robotics*, *31*(5), 1147–1163.

[32] Neven, D., B. D. Brabandere, S. Georgoulis, M. Proesmans, and L. V. Gool (2018), Towards end-to-end lane detection: an instance segmentation approach.

[33] Nistér, D., O. Naroditsky, and J. Bergen (2004), Visual odometry, in *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004.*, vol. 1, pp. I–I, Ieee.

[34] Poms, A., W. Crichton, P. Hanrahan, and K. Fatahalian (2018), Scanner: Efficient video analysis at scale, *ACM Transactions on Graphics (TOG)*, *37*(4), 138.

[35] Romero, F., M. Zhao, N. J. Yadwadkar, and C. Kozyrakis (2021), Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines, in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 1–17.

[36] Romero, F., J. Hauswald, A. Partap, D. Kang, M. Zaharia, and C. Kozyrakis (2022), Optimizing video analytics with declarative model relationships, *Proceedings of the VLDB Endowment*, *16*(3), 447–460.

[37] Salvador, S., and P. Chan (2007), Toward accurate dynamic time warping in linear time and space, *Intelligent Data Analysis*, *11*(5), 561–580.

[38] Sankoff, D., and J. Kruskal (1983), The symmetric time-warping problem: from continuous to discrete, in *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison*, pp. 125–161, Addison Wesley Publishing Company.

[39] SciKit-Learn (), Receiver operating characteristic (roc) with cross validation, https://scikit-learn.org/stable/auto_examples/model_selection/plot_roc_crossval.html, accessed: 2019-04-14.

[40] Sevilla, J., L. Heim, A. Ho, T. Besiroglu, M. Hobbhahn, and P. Villalobos (2022), Compute trends across three eras of machine learning.

[41] Statista (2021), Statista: Youtube video uploaded per minute, https://www.statista.com/statistics/259477/hours-of-video-uploaded-to-youtube-every-minute/, accessed: 2021-04-14.

[42] Teed, Z., and J. Deng (2018), Deepv2d: Video to depth with differentiable structure from motion.

[43] Ten Holt, G. A., M. J. Reinders, and E. Hendriks (2007), Multi-dimensional dynamic time warping for gesture recognition, in *Thirteenth annual conference of the Advanced School for Computing and Imaging*, vol. 300, p. 1.

[44] The Verge (2018), How tesla and waymo are tackling a major problem for self-driving cars: data, accessed: 2019-04-16.

[45] Tieleman, T., and G. Hinton (2012), Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude, *COURSERA: Neural networks for machine learning*, *4*(2), 26–31.

[46] TuSimple (2019), Github: tusimple-benchmark, https://github.com/TuSimple/tusimple-benchmark, accessed: 2019-04-14.

[47] U.S. Air Force National Coordination Office for Space-Based Positioning Navigation and Timing (), Gps.gov: Gps accuracy, https://www.gps.gov/systems/gps/performance/accuracy/#how-accurate, accessed: 2019-04-16.

[48] Wahlström, J., I. Skog, and P. Händel (2017), Smartphone-based vehicle telematics: A ten-year anniversary, *IEEE Transactions on Intelligent Transportation Systems*, *18*(10), 2802–2825.

[49] Wikipedia (2019), Receiver operating characteristic (roc), https://en.wikipedia.org/wiki/Receiver_operating_characteristic #Area_under_the_curve, accessed: 2019-04-14.

[50] Wikipedia (2019), Lane, https://en.wikipedia.org/wiki/Lane, accessed: 2019-04-15.

[51] Xarchakos, I., and N. Koudas (2019), Svq: Streaming video queries, in *Proceedings of the 2019 International Conference on Management of Data*, pp. 2013–2016.

[52] Yu, F., W. Xian, Y. Chen, F. Liu, M. Liao, V. Madhavan, and T. Darrell (2018), Bdd100k: A diverse driving video database with scalable annotation tooling, *arXiv preprint arXiv:1805.04687*.

[53] Zekany, S., R. Dreslinski, and T. Wenisch (2019), Classifying ego-vehicle road maneuvers from dashcam video, *Proceedings of the IEEE Conference on Intelligent Transportation Systems*.