ΠΑΝΕΠΙΣΤΗΜΙΟ
ΘΕΣΣΑΛΙΑΣ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

Διαχείριση Δρομολογίου και Αναγνώριση Διασταυρώσεων σε Συστήματα Αυτόνομης Οδήγησης με Υιοθέτηση Μάθησης Πολλαπλών Εργασιών

Παύλος Απλακίδης

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

ΥΠΕΥΘΥΝΟΣ

……Κωνσταντίνος Κολομβάτσος……..
…….Επίκουρος Καθηγητής…….

Λαμία …… έτος 2022

ΠΑΝΕΠΙΣΤΗΜΙΟ
ΘΕΣΣΑΛΙΑΣ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

Διαχείριση Δρομολογίου και Αναγνώριση Διασταυρώσεων σε Συστήματα Αυτόνομης Οδήγησης με Υιοθέτηση Μάθησης Πολλαπλών Εργασιών

Παύλος Απλακίδης

UNIVERSITY OF
THESSALY

SCHOOL OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE & TELECOMMUNICATIONS

Path Planning and Crossroad Detection for Autonomous Driving using Multi-Task Learning

Pavlos Aplakidis

FINAL THESIS

ADVISOR

……..Konstantinos Kolomvatsos……..
…….Assistant Professor…….

Lamia …… year 2022

«Με ατομική μου ευθύνη και γνωρίζοντας τις κυρώσεις [(1)], που προβλέπονται από της διατάξεις της παρ. 6 του άρθρου 22 του Ν. 1599/1986, δηλώνω ότι:

*1. Δεν παραθέτω κομμάτια βιβλίων ή άρθρων ή εργασιών άλλων αυτολεξεί **χωρίς να τα περικλείω σε εισαγωγικά** και χωρίς να αναφέρω το συγγραφέα, τη χρονολογία, τη σελίδα. Η αυτολεξεί παράθεση χωρίς εισαγωγικά χωρίς αναφορά στην πηγή, είναι λογοκλοπή. Πέραν της αυτολεξεί παράθεσης, λογοκλοπή θεωρείται και η παράφραση εδαφίων από έργα άλλων, συμπεριλαμβανομένων και έργων συμφοιτητών μου, καθώς και η παράθεση στοιχείων που άλλοι συνέλεξαν ή επεξεργάσθηκαν, χωρίς αναφορά στην πηγή. Αναφέρω πάντοτε με πληρότητα την πηγή κάτω από τον πίνακα ή σχέδιο, όπως στα παραθέματα.*

*2. Δέχομαι ότι η αυτολεξεί **παράθεση χωρίς εισαγωγικά**, ακόμα κι αν συνοδεύεται από αναφορά στην πηγή σε κάποιο άλλο σημείο του κειμένου ή στο τέλος του, είναι αντιγραφή. Η αναφορά στην πηγή στο τέλος π.χ. μιας παραγράφου ή μιας σελίδας, δεν δικαιολογεί συρραφή εδαφίων έργου άλλου συγγραφέα, έστω και παραφρασμένων, και παρουσίασή τους ως δική μου εργασία.*

3. Δέχομαι ότι υπάρχει επίσης περιορισμός στο μέγεθος και στη συχνότητα των παραθεμάτων που μπορώ να εντάξω στην εργασία μου εντός εισαγωγικών. Κάθε μεγάλο παράθεμα (π.χ. σε πίνακα ή πλαίσιο, κλπ), προϋποθέτει ειδικές ρυθμίσεις, και όταν δημοσιεύεται προϋποθέτει την άδεια του συγγραφέα ή του εκδότη. Το ίδιο και οι πίνακες και τα σχέδια

4. Δέχομαι όλες τις συνέπειες σε περίπτωση λογοκλοπής ή αντιγραφής.

Ημερομηνία: ...../...../2022

Ο - Η Δηλ.

(1) «Όποιος εν γνώσει του δηλώνει ψευδή γεγονότα ή αρνείται ή αποκρύπτει τα αληθινά με έγγραφη υπεύθυνη δήλωση

του άρθρου 8 παρ. 4 Ν. 1599/1986 τιμωρείται με φυλάκιση τουλάχιστον τριών μηνών. Εάν ο υπαίτιος αυτών των πράξεων

σκόπευε να προσπορίσει στον εαυτόν του ή σε άλλον περιουσιακό όφελος βλάπτοντας τρίτον ή σκόπευε να βλάψει άλλον, τιμωρείται με κάθειρξη μέχρι 10 ετών.»

ΠΕΡΙΛΗΨΗ

Τα αυτόνομα οχήματα είναι ένα μείζον θέμα στον τομέα της τεχνητής νοημοσύνης. Μπορούν να λύσουν πολλά από τα προβλήματα της καθημερινότητας, και πιο συγκεκριμένα το πρόβλημα της ασφαλούς συγκοινωνίας. Έχει αποδειχθεί πως ο πιο αποτελεσματικός τρόπος για να μετατρέψει κανείς ένα αυτοκίνητο σε αυτόνομο όχημα είναι η χρήση μόνο της κάμερας ως αισθητήρα αντίληψης, σε συνδυασμό με την χρήση μηχανικής μάθησης για την λήψη αποφάσεων ως προς το πού πρέπει να κινηθεί το όχημα αυτό. Η χρήση φθηνών αισθητήρων (συγκεκριμένα κάμερα) καταφέρνει να μειώσει σημαντικά το κόστος παραγωγής και ταυτόχρονα την τιμή πώλησης, συγκριτικά με άλλους αισθητήρες όπως LiDAR, οι οποίοι έχουν πολύ υψηλό κόστος και έχουν αποδειχτεί μη απαραίτητοι. Δύο βασικές εταιρείες επιτυγχάνουν την αυτονομία οχημάτων για τον μέσο καταναλωτή, η Tesla και η Comma.ai, οι οποίες χρησιμοποιούν προηγμένη μηχανική μάθηση πάνω στις εικόνες που παρέχει η/οι κάμερα/ες σε συνδυασμό με άλλους αλγορίθμους που καθορίζουν την κίνηση του οχήματος. Συγκεκριμένα, η Comma.ai χρησιμοποιεί μια μόνο κάμερα που βλέπει μπροστά από το αυτοκίνητο, δηλαδή ένα smartphone που εφάπτεται στο παρμπρίζ (πράγμα που επιτρέπει το Openpilot, όπως λέγεται το προϊόν τους, να υποστηρίζεται από τα περισσότερα αυτοκίνητα της αγοράς). Τα ζωντανά βίντεο της κάμερας προωθούνται σε ένα τύπου end-to-end μοντέλο μηχανικής μάθησης το οποίο είναι υπεύθυνο για την κίνηση του αυτοκινήτου. Αυτή ήταν και η βασική έμπνευση της συγκεκριμένης πτυχιακής εργασίας, όπου μέσω βίντεο από smartphones που εγκαταστάθηκαν στο παρμπρίζ διαφόρων αυτοκινήτων, έγινε προσπάθεια εντοπισμού διασταυρώσεων, ορίων του δρόμου και δρομολόγηση του οχήματος με την χρήση μηχανικής μάθησης πολλαπλών εργασιών.

ABSTRACT

Autonomous vehicles is an important topic, among many others, for the AI (Artificial Intelligence) community. For many years, self-driving cars have been considered science fiction. However, with the works of companies such as Tesla (Autopilot) and Comma.ai (Openpilot), this "science fiction" seems highly achievable. These software products use simple sensors (cameras, radar, etc) and Deep Learning to eventually output a drivable path for the car to follow, and then other types of software handles the execution of the that path. One approach is more of a mid-to-mid solution (highly used by Tesla), meaning that deep learning is used to detect various objects such as pedestrians, stop signs, lane lines, etc to produce a 3D map of the car's environment. Then other algorithms are used (probably more deep learning), that take into account the generated map and output the final path the car should follow based on human driving data. This solution builds more trust to end users since they can see the map (therefore what the software "sees"), however it requires top-of-the-line expensive hardware and also causes doubts in it's scalability. Another solution uses a more end-to-end approach (meaning one big neural network takes the sensor data as input and outputs the final path), which doesn't output a fancy 3D map to the user, but promises more scalability for the future (highly used by Comma.ai). This approach basically relies more on AI and lets a machine learning model decide the path (based on previous data from human drivers) via the sensors themselves rather than creating a map of the environment. This approach has been proven very efficient on highway driving using less expensive hardware and being compatible with many car brands and models (even Tesla cars), however this open-source driving agent is not ready yet for driving in big cities, towns, etc, thus it is not

ready for full self driving capabilities. A big problem of city driving is how the car should handle crossroads. This thesis tries to provide a simple (yet not necessarily scalable) solution to how the car should handle crossroads by learning how to detect them and plan a path accordingly based on how humans drove in similar situations. There is also some experimentation with the performance of multi-task deep learning. The final application shows promising results in simple tasks such as crossroad detection, road edge detection and path planning, using just neural network(s), while also benchmarking single networks against multi-task networks in both task efficiency/accuracy and performance (speed). The only sensor used for this project was a single camera (monocular). Software libraries such as pytorch, opencv, etc have been proven very useful.

Table of Contents

CHAPTER 1 Introduction

1.1 What is autonomous driving?

Nowadays, a lot of cars ship with many assistance and automation features. The golden standard is cruise control, which keeps the car's speed at a specific value for extended periods of time, in order to make driving long distances a bit more relaxing for the driver. There is also adaptive cruise control, which is programmed to change the speed value depending on the situation (i.e. when making some turns the car needs to slow down).

Another feature that recently has come to play in a lot of modern cars is lane keeping assist, that requires some robust lane-lines detection. It can either work as a warning for when the car is shifting out of lane, so that the driver fixes his course to avoid an accident, or it can also have some autonomy in order for the car to slightly correct it's path by itself.

The features mentioned above belong in different vehicle automation levels that describe how much the driver has to participate in actual driving. There are 4 main levels (even though there could be more, depending on how one categorizes the given features). On level 1 we have the basic cruise control that was mentioned earlier. On level 2 we have adaptive cruise control and maybe some lane-line detection with warnings (or even a slight corrective turn of the wheel) for when the car strays out of it's supposed path. Here, there is a minimal level of automation and it's purpose is just to assist the driver in order to avoid accidents and make the process of driving more comfortable.

When referring to level 3 automation, one means that 99% of the time the driver does not need to take any action and the car successfully drives itself, with some probability of error. Level 3 features include longitudinal and lateral path planning and full controls autonomy, meaning that the car can fully moderate it's speed and steering in order to drive in different situations, while the driver only needs to pay attention so that he can take over in case the system fails at any moment.

Finally, at level 4 there should be no humans on the driver's seat. The car should be able to fully drive itself like a human (or even better) without the probability of a system failure or any accident. This level unfortunately still belongs in the category of science-fiction, for the time being. This is why modern autonomy algorithms require the driver to be engaged at all times, so that if the car itself fails to successfully drive in a given scenario, the driver can always take over and correct it.

Today, there is a number of companies that have achieved autonomy close to level 3. Most notably, Comma.ai and Tesla have done so by minimizing the cost, relying on less expensive sensors such as cameras and radar (as opposed to the costly LIDAR sensors) and utilizing deep learning so that the car actually learns to drive by the example of humans.

1.2 Sensors used

Human driving consists roughly of 3 steps, perception, planning and control. The same goes for the computer/machine that drives the car. For perception we use various sensors such as cameras, GPS, radar, etc. In this project, only a single smartphone camera was used, just like in Openpilot. Monocular (i.e. using one camera) has been proven to be enough for many self-driving tasks, and can be more usable since the end user only has to mount a smarphone on the windshield instead of installing a bunch of cameras in the car. Comma.ai claims that they achieve great results without even using the full potential of monocular self-driving.

LIDAR is not practical either, since it is expensive and quite unnecessary. Humans drive by mostly using their vision for perception, therefore an HD map of the whole environment of the car is overkill and impractical. There are also sensors such as RADAR, that can assist the process of self-driving while on the path for level 4, however eventually the algorithms should be able to drive a car without relying on it. For now, it is being used for corrective purposes, in order to assist the machine learning algorithms in depth estimation, leading car detection, etc.

This thesis focuses more on perception (monocular detection of crossroads and road edges) while implementing a simple but not so practical path planner. There is no implementation of controls, since this was never the focus of this project, so other possible car sensors are ignored.

CHAPTER 2 Previous Work on Self-Driving Cars

This thesis is heavily inspired by the works of Tesla and Comma.ai in the market of self-driving cars, with their products (Autopilot and Openpilot from each company respectively) showing great results and a large fleet of users. Both self-driving agents use machine learning algorithms and rely on cheap sensors such as cameras in order to decide the overall driving policy of the car. There are 3 basic steps for the autonomy stack in cars regarding software, perception, planning and controls.

Tesla utilizes many multi-task deep neural networks in order to cover the whole stack. For perceptions, many large neural networks are used in order to detect cars, pedestrians, signs, road edges, lane lines, obstacles, etc. This leads to the creation of a detailed map of the environment of the car. Many other approaches use static algorithms on LIDAR sensor readings in order to create an HD map of the environment, which is quite impractical since LIDAR sensors are expensive and the algorithms used are not robust to noise or edge cases. The machine learning approach tries to replicate human behavior so that the car can handle itself well enough in most situations without following static rules on how to drive. After creating a map using machine learning, the same techniques are used in order to plan the path for the car to follow. For controls, since Tesla makes it's own cars, the software has full access to steering and gas/break pedals. The last part, however, does not concern this thesis so it shall not be focused on.

On the other hand, Comma.ai's approach skips the perception step and uses one deep multi-task neural network in order to plan the path, based just on raw video data from a dashboard camera. This approach is called end-to-end and has been focused on by Nvidia and other companies as well. It is the most promising, but still requires a lot of research and especially a lot of data. It scales better since it let's the complex machine learning model what to focus on from the video feed, instead of relying on a map of the environment. With this approach, the car can drive itself in a smoother way, replicating the behavior of a human more accurately. This approach was focused on the most for this project, since the main goal was to help an end-to-end path planner make better decisions when meeting crossroads, by training a multi-task model alongside a crossroad and road-edge detector.

CHAPTER 3 Algorithms Used

3.1 Artificial Intelligence

3.1.A Learning from humans

Many people think that to make a car drive itself we need to define an enormous amount of static rules (traditional programming) such as how much distance it needs to leave from the leading car, or where exactly to stop if it sees a stop-sign. This is mostly how cars drive in video-games, where the environment of the car is quite predictable and the car itself is not required to do any hard tasks rather than just roam without crashing. But we want to deploy a self-driving car into the real world, which can be very unpredictable. Even if a decent traditional programming solution is proposed (which is unpractical since it requires a lot of unnecessarily complex code), it will not be able to handle edge cases that surprisingly happen a lot on a daily basis.

The biggest question that provides the solution is the following: "How do humans actually learn to drive?". Before we even start studying for a driving license, we already know what a stop sign is and what to do when we meet one on the road. Why is that? Well, we learn basic driving before we even get a hold of a wheel by observing other people (such as parents, etc) drive. Why not apply that method for the self-driving agent? Basically, instead of instructing the machine how to drive by providing static rules, we teach it by showing it how humans drove in a variety of situations. Instead of telling it to stop when a stop sign is present, we should show it scenarios where an actual person stopped while encountering one.

This solution should achieve autonomous vehicles that drive like actual human beings, in order to handle any scenario and achieve more "natural" movements. Humans are good at judging several situations, such as when the car doesn't have to or even should not follow the lane-lines and instead drive either towards the center or the outer side of the lane. In a scenario like this, a static algorithm wouldn't be able to judge the situation correctly and thus would keep following the lane-lines. By using artificial intelligence in order to teach the car how to drive itself like a human, we can just show the car what to do in every situation so that it decides for itself, based on it's given data, what to do. The main goal is to make a self-driving car that drives like an actual human without any error, which will eventually achieve safer driving and less accidents. Most accidents don't happen because a driver can not handle his/her car, but due to distractions and lack of awareness. A computer will always pay attention and combining it with a correct and safe driving policy, this would make the roads safer.

Section 3.1.B Machine Learning

With this solution proposed, another question arises: “How do computers learn to complete tasks based on real-world data?”. The answer is “By using machine learning”. But what exactly is machine learning and why is it becoming such a big deal for artificial intelligence the last decade?

There are three types of machine learning: supervised, unsupervised and reinforcement learning. This project uses supervised machine learning, which means that the computer learned the given tasks based on previous data it was shown. For these tasks, human driving videos (dash-cam) were used as training data.

There are two basic types of tasks a machine learning model can do: regression and classification. For regression, the model is given a set of inputs and some continuous numeric values as training data. After some iterations of the training data and a suitable machine learning model (which is more like a big mathematical function), the computer learns not only what to output given an input it has already seen, but can also generalize from the given dataset in order to give correct outputs from inputs similar to what it has already “seen”. There are different machine learning models for such tasks, such as linear or logistic regression, etc.

For classification, the model is once again given a training dataset that consists of inputs and their expected outputs, however this time the outputs are not continuous values (for example numbers), but different classes (e.g. male, female, dog, cat, etc). There are two types of classification, binary and multi-class. Binary classification happens when the output classes are only two (e.g. yes or no, male or female, etc), while multi-class when we have more that two possible classes in the dataset. There are different classification models/algorithms, such as decision trees, Bayesian classifiers, etc.

Unsupervised machine learning algorithms, such as clustering, are used in situations where we are given only input data, in order to find various correlations between the tuples/rows. These algorithms automatically extract statistics and other characteristics of the given data resulting in a more detailed analysis.

In reinforcement learning, an intelligent agent is deployed in an environment and, given a specific set of controls, tries to learn by trial and error. The programmer defines a reward function, so that the model knows when it is doing the right things or the wrong ones. After performing some sets of actions and episodes, the agent calculates it’s reward and evaluates it’s policy. This type of machine learning can perform great in games whose rules are deterministic, such as mazes, tic tac toe, etc, where the agent plays a lot of rounds, tries, fails and learns from it’s mistakes in order to win the given game or complete a task. Such type of learning algorithms have been used in mini-projects for self-driving cars in video games, where after a large amount of episodes (trials and errors until the game is over), the car could drive itself without crashing, even though the actual driving was not really elegant. This method can work in a video-game where the environment is quite predictable and the rules are predefined, however in the real world it could be dangerous and highly sub-optimal.

The algorithms/models mentioned above have shown great results and performance in simple tasks, however for autonomous driving tasks, a more sophisticated solution has been proven to work greatly. The models used for this

project belong in the category of Neural Networks, they are part of a subcategory of machine learning, called Deep Learning.

3.1.C Deep Learning and Artificial Neural Networks

With the huge improvements that have been made in computer hardware, specifically on GPUs and memory, the concept of Artificial Neural Networks has been tackled and researched intensively. Today, they are widely used for various complex machine learning tasks. Basically, Deep Learning is the use of artificial neural networks for machine learning problems instead of the more traditional algorithms/models.

But what exactly is an artificial neural network? In simple words, it is an attempt to model the human brain (which is proven by research that it is a big network of neurons) using computers. Like the human brain, artificial neural networks have many connected neurons that receive and transmit signals from and to other ones. In reality, this explanation is merely a simple visualization of such models, a concept that was followed during their creation. These networks consist of many layers of neurons that are inter-connected, where the number of layers dictates their depth. Deep learning is associated with the research and use of these deep artificial neural networks for various tasks, thus it's name.

A more pragmatic explanation of what an artificial neural network is would be the following: a large and complex mathematical function f(x) = y where x is the input and y the output of the network. How f is defined is really arbitrary and widely researched until even today, but simply put: f is visually the architecture of the network (the calculations that the function executes).

To understand neural networks and their architecture, one must first understand what a single neuron is. A neural network that consists of one neuron is called a perceptron (since it is not really a network, no actual connections exist). A simple perceptron is basically a linear regressor, which means it utilizes the mathematical function of y = w*x + b, where w is called the weight and b the bias. So, a simple perceptron model tries to find a straight line that best suits and generalizes the line the training data creates. It achieves that by finding the right weight and bias for the given task. W and b are initialized randomly (they can however be initialized by using other algorithms, depending on the training data and machine learning task), and during training the inputs x from the given dataset are "fed" forward to the perceptron (and therefore to the whole network when we are dealing with more than one neuron). Y (the output of the model) is calculated through the given function, and then is compared with the corresponding Y of the training data. The difference between Y_train and Y_out is called Error. The goal of the training process is to minimize that error (also called loss), therefore improving accuracy, while being able to generalize on similar previously unseen data.

There are many ways to define the error and they are called loss functions. Some examples for regression are Mean Squared Error (1/n * Σ(yi yi')^2), Mean Absolute Error (1/n * Σ|yi – yi'|), etc. The weight w and bias b are updated with

each iteration of a tuple/row (i.e. one input x and it's corresponding output y) depending on the value of the error.

But how are they updated? By using a method called backpropagation that uses gradient descent:

$W_{new} = W + \Delta W$

$\Delta W = a * Error * \sigma(W*x) * (1 - \sigma(W*x)) * x$

$Error = y - f(x)$

where y is the correct value from the dataset and f(x) the output of the perceptron/network .

Where a is called the learning rate, which determines how much the value of the weights will change, i.e. how fast the network learns. If the value is too big, the network will make huge changes to it's weights and not be able to match the training data properly, while if it is too small, the network won't make changes drastic enough to learn the task. The ideal value of a is a matter of experience and experimentation.

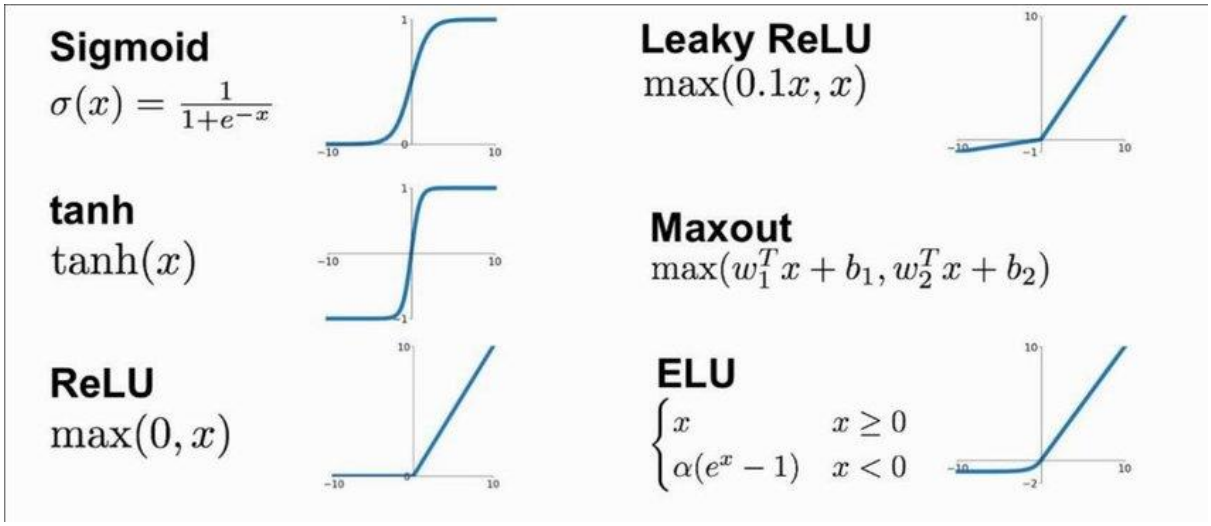
3.1.D Learning more complex models by using Activation Functions

Perceptrons and neural networks are good enough for simple problems. The architecture that has been explained has some problems, such as disappearing gradiens, inability to match more complex mathematical functions, etc. These problems have been solved by using activation functions.

An activation function works like a gate for the output of a single neuron. It is a mathematical function which takes as input the output of a perceptron and, depending on the function's type, outputs a different value. There are many activation functions used, such as Sigmoid, Softmax, Tanh, ReLU, ELU, etc. The activation functions that were used in this project will be explained later on.

What these function allows the neural networks to do is achieve better logistic regression, thus learning more complex tasks. The simple perceptron without an activation function can only perform linear regression. By using these functions, it can perform different types of classification, binary (using sigmoid, which clamps the output between 0 and 1) or multi-class (using softmax).

In conclusion, activation functions are used for more complex regression or for classification. The output of a binary classifier neural network or perceptron is a single value, 0 or 1, while the output of a multi-class classifier is a vector of length equal to the number of possible classes. Each value in the vector represents the probability of the class in the corresponding index, the highest one is the one the network eventually chooses.



Some of the most basic and commonly used activation functions in neural networks

3.2 Neural Networks Architectures Overview

3.2.A Data

As mentioned previously, the data that was used in this project is in the form of video files (mp4) from a single dashboard camera, thus the name monocular crossroad detection. Data is a very crucial, if not the most important, piece of the puzzle, since it defines the overall behavior of the model. The network learns and matches the training data while also performing relatively good on similar scenarios/videos that it has never been exposed to before.

The videos' quality is mostly HD, however training on actual HD data is memory demanding and highly impractical. Neural networks do not require HD frames to be accurate, therefore all the frames of each videos are preprocessed and converted to a lower resolution, specifically 320x160. Most of the data is urban driving, where crossroads are frequent, along with stop signs and traffic lights.

The overall preprocessing of the data consists of breaking the mp4 file into individual frames (using PIMS library) that are downscaled as mentioned above. All frames from all available videos are stored to RAM, which is hardware demanding (over 16GB of memory is used by the training script), along with the corresponding label or annotation (road edges, path, etc) for each and everyone of them.

As mentioned before, importance of data is tremendous. Correct labeling of that data is crucial and needs to be done with precision and efficiency. Big companies have either a large team for manual labeling the given data, or have developed a sophisticated auto-labeling stack. During the development of this project, manual labeling using various tools such as the custom crossroad labeler script and computer vision annotation tool has proved to be enough for this thesis but not scalable for large production.

Specifically for labels, crossroads were just two possible values (binary classification), 0 or 1, thus labeling was quite simple, by using a custom script that utilizes opencv python library. For road edge detection and path planning, polylines had to be drawn on each frame. Therefore, a tool called computer vision annotation tool proved very useful, since it required a static amount of points per polyline, and the lines themselves were tracked automatically. The neural network has to regress the points for each polyline. Scripts for various tweaking and preprocessing of the annotations were written, with the most important tasks being serializing/flattening polylines's points into one large vector that matches the output of the neural network, and deserializing that output back into the polylines.

Each polyline consists of about 8 points with 2 coordinates (i.e. x, y). So the annotations of each frame are one 3D array (1st dimension indicates the polyline, 2nd dimension indicates the point and 3rd dimension the coordinate). The algorithms used for this type of preprocessing do not have the most optimal time complexity (O(n^3)), although training performance was quite safisfying, in terms of time efficiency.

3.2.B Computer Vision and Convolutional Neural Networks

Since the given data consists of videos, specifically a series of frames, computer vision algorithms seem to be the solution to crossroad detection and the rest of the given tasks. However, traditional computer vision algorithms such as edge detection, static feature detectors, etc are not good enough for the complexity of the given problem.

Each image/frame consists of pixels. A pixel basically is the smallest possible square on the image or generally on the screen itself. In programming and digital image processing, a greyscale image is a 2D matrix, each cell representing a pixel, which is an integer, an arithmetic value that determines how light or dark the specific part of the screen will be. For colored images, the frame is represented by a 3D matrix in the form of RGB (Red Green Blue). This 3D matrix therefore consists of 3 2D matrices that follow the same pattern as the greyscale image, i.e. each cell of those matrices represents the intensity of each pixel (how much Read, Green or Blue exists in that pixel). So in conclusion, a colored image consists of 3 different images combined together and creating many other colors through combinations. For this project, each frame is likewise a 3D image of 320*160*3 pixels.

For simple problems such as low resolution hand-written digits recognition (MNIST dataset), were the input matrix would be a flatten/serialized vector of each image's pixel, a simple neural network can perform quite well without any modification. But for an input vector of size 320*160*3 = 153600, computation is impractical. However, not all of the pixels of a frame are useful. We only need specific features. As mentioned above, in computer vision there are many static algorithms that detect specific features (such as corners), each feature being a small neighbourhood of pixels, resulting in a much smaller input vector for the neural network. This method unfortunately does not perform very well since we are telling the net to look for very specific and static things on each image and there are many ways it can go wrong.

But why not use the same philosophy of machine learning, specifically deep learning, for extracting features. Instead of statically defining what to look for, we can train the network to learn the features it needs along with how to classifiy them. This is done using a specific architecture called Convolutional Neural Network.

A convolutional neural network consists of 2 parts, the feature extractor that learns which parts of the input image it should focus and outputs a vector, and the classic neural network part that takes that vector and performs the given task, either classification or regression.

3.2.C Convolutional Layers

In digital image processing, the concept of filters is really important. They change the overall look of the image, making it sharper or blurry, extracting boundaries and many more operations. All these are done by utilizing the concept of convolution.

As mentioned above, each image consists of pixels. An image is represented by an array of them, each cell containing information about it's intensity. A filter is a relatively small kernel of pixels, an array that represents a tiny neighborhood. Typically, a filter's size is estimated at 3x3, 5x5, 7x7, etc (all odd numbers for easier application) for gray-scale images, and 3x3x3 5x5x3, etc for RGB ones (the third dimension represents the color).

But how does convolution (i.e. the application of the filter) actually work? Let us assume a 3x3 filter and a 3x3 neighborhood in an image. The pixel of the neighborhood that will be affected by the filter is the central one. The method will be explained in the following example.

K1	K2	K3
K4	K5	K6
K7	K8	K8

I1	I2	I3
I4	I5	I6
I7	I8	I8

Above is the 3x3 (K) kernel mentioned before and the 3x3 neighborhood from the image (I). The pixel that will be affected is I5. The mathematical function used is the following:

$$I5\_new = K1*I1 + K2*I2 + K3*I3 + K4*I4 + K5*I5 + K6*I6 + K7*I7 + K8*I8$$

Filter K can also be called the weight W (just like in basic neural networks). This method applies for a 3x3 portion of the image. For the filter to be applied to the whole image, the same function is used using every pixel as the center of a 3x3 neighborhood. For the boundaries of the image, where some pixels are missing since they are out of bounds, we use a method called zero-padding, meaning the missing pixels of the neighborhood are replaced by 0. There are also some different methods of padding, they are however irrelevant to this thesis.

Another concept in convolution and filter application is called stride. When we use a stride equal to 1, we basically "move" the filter by one pixel at a time, as mentioned above. Using a stride value greater than 1 will result in skipping some pixels (depending on that value). For example, with stride=2, the filter will be applied on pixel (0,0) then on (0, 2), (0, 4), etc, instead of moving right to the next ones.

The concept of convolution can be applied in convolutional neural networks, where the network attempts to learn the value of the kernel needed for the task. A convolutional layer is basically the application of a specific filter, and the parameters that can be given are the kernel size and the stride value, but also the

output channels (third dimension of the image), that is determined by the third dimension of the kernel itself.

Another concept that is used very often after convolution is pooling (max, average, etc). A 2x2 max-pooling will take a 2x2 neighborhood of the image and replace the whole 2x2 area with the max (or average in average-pooling) of that neighborhood, resulting in overall less total pixels in the image (since we want to be reducing the input image by extracting smaller features that the network can handle).

After some convolutional layers, the end result will be several feature maps (small images that represent features the network detected and focuses on). Depending on the final output channels (the number of output channels represents the number of the feature maps the layers give), we are left with smaller more manageable images, which are then serialized/flattened into a single vector that is fed to the remaining of the neural network. The final layers behave like the ones discussed in previous sections and are responsible for the actual given task.

3.2.D Residual Blocks and ResNet

A typical convolutional neural network consists of several blocks of convolutional layers (with ReLU activation function, which will be discussed further on), followed by a max or average pooling layer and sometimes batch-normalization is utilized (will also be explained later). Stacking these blocks will scale the network up, making it more powerful and able to meet the requirements of more complex tasks. This is called vertical scaling (increasing the depth), but there is also horizontal scaling, which increases the neurons per layer.

Vertical scaling has proven to be very efficient in increasing accuracy and minimizing loss in plain convolutional neural networks, however there seems to be a ceiling to that performance. After stacking several blocks (about 34 according to some experiments), accuracy stops increasing (and in some cases even decreases, just like in over-training for more epochs than needed) and the extra blocks become practically useless. This limits the networks to less complex tasks. In autonomous driving, the need for highly powerful convolutional neural networks is tremendous, so this limitation can be troublesome when trying to train on hours upon hours of complex driving data, especially when utilizing a more end-to-end approach, like comma.ai does.

In 2015, a new method and a new type of block was introduced, called the Residual Network. A residual connection/block is basically similar to a plain one, with a seemingly minor, however very effective, tweak. The input of the block is kept, and added to it's output. For instance, consider y to be the output of a residual block, x it's input, while F represents the overall function of the plain convolutional block. Then we would have the following:

$y = F(x) + x$ (or) $y = F(x, \{W_i\}) + W_s * x$

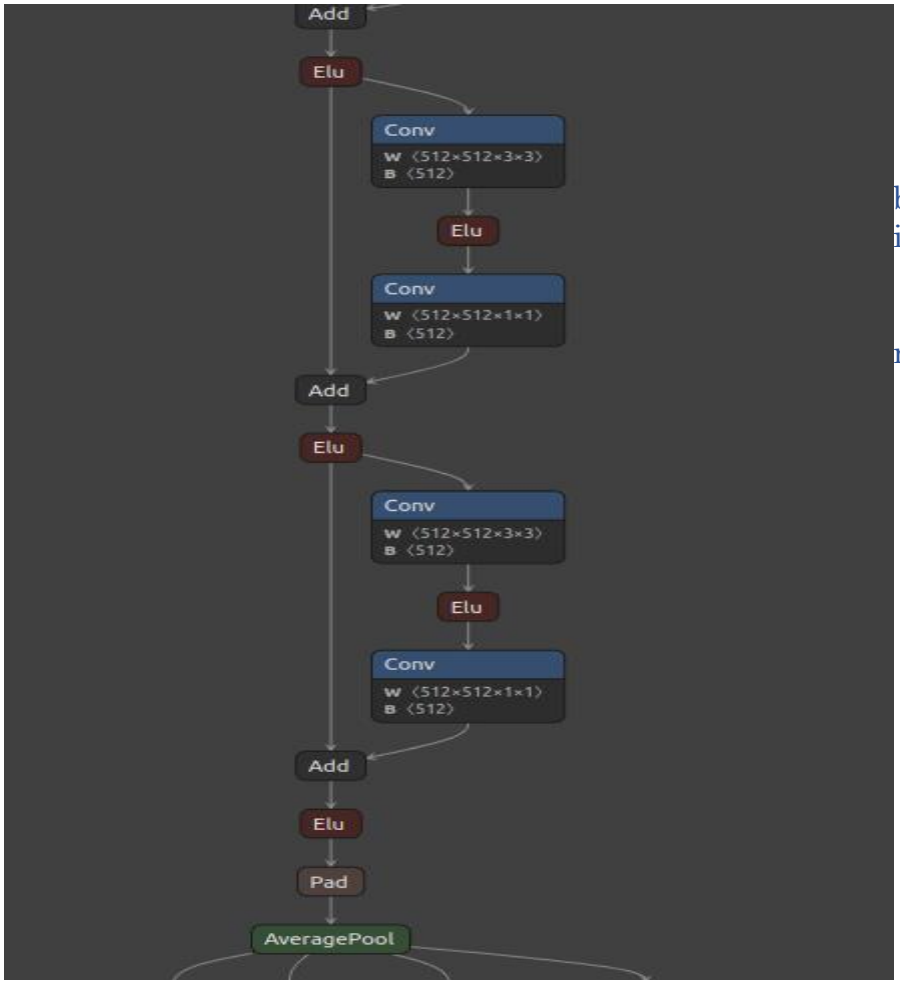
This method takes into consideration the previous state of the input, and improves vertical scalability, allowing for more layers to be added while improving overall performance and allowing this Residual Neural Network to learn much more complex tasks. However, vertically stacking layers might work in theory, a large amount of computation performance is required for this to be relevant.

Thankfully, modern hardware can support up to 152 (or more) convolutional layers is a ResNet (as it is called), while achieving amazing results, surpassing previously propsed convolutional neural network architectures such as ImageNet, CIFAR, VGG, etc.

For this project, an 18-layer ResNet was used as the backbone (feature extractor) for most of the neural networks used. It's performance has proved indeed to be amazing, however due to lack of data, it can easily over-fit some tasks. This promises great scalability for larger datasets, which only big companies such as tesla and comma.ai possess due to their large fleet of cars using their products.

Over-fitting was not discussed before, but it simply happens when a really powerful network learns the given training dataset too well. This results in great statistics in training (almost 0 loss and 100% accuracy), however the model cannot generalize the data it has learned, resulting in low accuracy during evaluation. This can be solved by either using a less powerful model (not practical for self-driving cars since it is a highly demanding task) or by feeding the network a much larger dataset, covering edge-cases along with day-to-day driving.

The following is a small portion of one of the ResNets used in this thesis:



Two example ResNet blocks used in the project's neural netowrks

3.2.E Batch Normalization in Deep Neural Networks

When dealing with deep neural networks with tens of layers, like the ones used in this project, several problems arise, besides computation power needed. The initial random weights during training can affect the whole training process negatively and combined with the need to update all the parameters from the many layers used, it results in the deeper layers' inputs to be distributed differently, making the learning process more difficult or near impossible. During some testing for crossroad detection, the used network's output always remained the same, indicating that there was no learning of the dataset.

This problem was solved by using a technique called batch-normalization. Using Batch-Normalization layers in the network helps counteract the different distribution in the inputs of the deeper layers, giving us the desired result.

To properly understand how these layers work, one needs to understand what normalization really is. In general, it is mapping some given data that has a specific range (max and min value), to a new one (e.g. [0,1]). There are different algorithms for normalization, all achieving similar results. We have min-max, z-score and normalization by decimal scaling.

In min-max normalization, the range is modified using a linear transform on the data. Suppose that Vi is a sample from the dataset and (min, max) is the range it currently belongs to. To map Vi to a new range (min_new, max_new), one needs to apply the following mathematical function:

$$Vi\_new = [(Vi - min) / (max - min)]*(max\_new - min\_new) + min\_new$$

The biggest advantage of this method is that the correlations of the data remain intact.

In z-score normalization, the average and standard deviation of the data is used. The math is as follows:

$$Vi\_new = (Vi - avg)/\sigma$$

Where avg is the average of the data and σ the standard deviation. The advantages of this method is that one does not need to specify a new range and it is resilient to outliers (which are anomalies in the dataset, data points that do not match the rest of them). Another variation of z-score normalization is by using the mean absolute deviation instead of standard deviation:

$$s = 1/n * (|v1 - avg| + |v2 - avg| + \ldots + |vn - avg|)$$

This variation achieves even better results against outliers.

So, as mentioned before, a batch normalization layer takes an input and normalizes it for the next hidden layer. Since this kind of layers belong to a neural network, they have several parameters, some are learned and others not. This differentiates them from the classic approach to normalization, they are different since they deal with a different kind of data.

The two parameters that are learned are called Beta and Gamma, while the other two non-learnable are the Mean Moving Average and the Variance Moving Average. Specifically, let's suppose that A is the input vector (activation or generally the output of a previous layer). The batch normalization layer applies the following math:

- step 1:

$\mu i = 1/M*\Sigma Ai$ (calculate mean, M is the number of samples in A)

$\sigma i = sqrt[(1/M)*\Sigma(Ai - \mu)^2]$ (calculate standard deviation)

- step 2:

Ai' = (Ai – μi) / σi (normalize the sample)

These steps are very similar to how z-score normalization works. Generally the layers might be different in how they work but there are quite some similarities to be noticed with the previously mentioned algorithms. Up until now, the learnable parameters were not used at all. The next few steps are the most important:

- step 3:

BNι = γ * Ai' + β (Scale and swift, β=beta – γ=gamma, gamma multiplication is element-wise unlike matrix multiplication)

In this step, the parameters meant to be learned (gamma and beta) are very similar to the weights and biases for basic neural network layers. BNn (the whole vector of BNi elements) is the layer's output vector, which is fed to the next one. Now, for the two other non-learnable parameters:

- step 4:

μ_movi = α*μ_movi + (1-α)*μi

σ_movi = α*σ_movi + (1-a)*σi

The above step demonstrates how the mean moving average and the variance moving average are updated. This step is not that important since these parameters are not really used during training, but they can be useful during the inference phase where evaluation of the network happens.

3.2.F Bayesian Layers

In standard neural networks, for each layer we have the following mathematical function:

$y = f(w^T*x) = f(w1*x1 + w2*x2 + \ldots)$

The learnable parameters of the network's layers are the weights (weight vector W). These weights are learned in a deterministic way, meaning that the weights are updated into new static values, without any doubt of those values. However, the network is not entirely "sure" of the updated weight values, especially in the early stages of training.

The core idea behind Bayesian layers is that instead of using weight values for the learnable parameters, the network can use a distribution of the weights. For instance, instead of updating a weight w1, we update the values of a normal distribution $N(\mu 1, \sigma 1^2)$, where μ is the mean value and σ^2 the standard deviation. So, instead of learning the weights, the network learns the parameters of the normal distribution of the weight. This introduces the concept of confidence. The mean is an estimation of the actual weight and standard deviation indicates how confident the network is about it.

$w1 \rightarrow N(\mu 1, \sigma 1^2)$

$w2 \rightarrow N(\mu 2, \sigma 2^2)$

...

Typical neural networks are deterministic in their nature, which means that for each input the net will give only one specific output. Bayesian neural networks are non-deterministic due to the Gaussian normal distributions of the weights, meaning that each input might give slightly different outputs, depending on the deviations. This achieves higher generalizability for the network, allowing it to tackle more complex problems, such as road edge detection, where they were used. In road edge detection we are trying to regress the points of each road edge that can be detected. Standard neural networks proved incapable of doing that since the data is far too complex for their capabilities, but adding Bayesian layers after the convolutional ones (instead of linear layers) proved to be much more efficient. Ofcourse, there is always room for improvements for this task.

There are many types of Bayesian layers, since their concept can be applied to many types of weights, such as Convolutional Bayesian Layers, etc. In this project, only Linear Bayesian Layers were used, in replacement of the standard linear layers at the end of the convolutional ones.

To explain this concept in more detail, let us suppose that we have a dataset $D = (xi, yi)^n_{i=1}$ and a model/neural network $F_\theta()$, using a loss function L(), such as cross-entropy (θ are the model's learnable parameters). Then for a deterministic model we have:

During Training:

$\theta^* = \text{argmax}_\theta \sum \log[p(yi \mid xi,\theta)]$ where θ^* is the optimal value for θ

or

$\theta^* = \text{argmin}_\theta \sum L(F_\theta(xi), yi)$ meaning we are trying to learn the best parameters that minimize the loss

During Prediction:

$y = F_{\theta^*}(x)$

For a Bayesian Neural Network model we have:

During Training:

$\mu^*, \Sigma^* = \text{argmax}_{\mu,\Sigma} \sum \log[p(yi \mid xi, \theta)] - KL[p(\theta, p(\theta_0))]$

where μ^* and Σ^* are the mean and standard deviation of the optimal parameter's θ^* normal distribution and KL is the regularization

$\theta \rightarrow N(\mu, \Sigma)$ $\theta_0 \rightarrow N(0, I)$

(we are trying to minimize KL and maximize the log-likelihood)

This method prevents overfitting to the data, since it ensures that the weights' are not too big

or

$\mu^*, \Sigma^* = \text{argmin}_{\mu,\Sigma} \sum L(F_\theta(xi), yi) + KL[p(\theta), p(\theta_0)]$

During prediction:

$y = 1/K * \sum F_{\theta^*_K}(x)$ meaning that we average all the individual sample predictions

The library called BliTZ proved really useful in combination with PyTorch, and provides many utilities in order to build a Bayesian Neural Network, since most deep learning frameworks do not provide such functionality.

3.3 Network Tasks and Loss Functions

3.3.A Crossroad Detection - Binary Classification

As mentioned before, crossroad detection is a binary classification problem. There are only two possible answers to the question "Is there a crossroad in front of the car?", yes or no. If it was attempted to deal with this problem as multi-class classification (treating yes and no as two separate classes) there would be unnecessary computations.

So, the output tensor (meaning the output matrix, this is how they are called in the most used deep learning frameworks for python) of the crossroad detection convolutional neural network will be a single neuron that can either be 0 or 1. For the clamping of the output value in the range of [0,1], we use the sigmoid activation function.

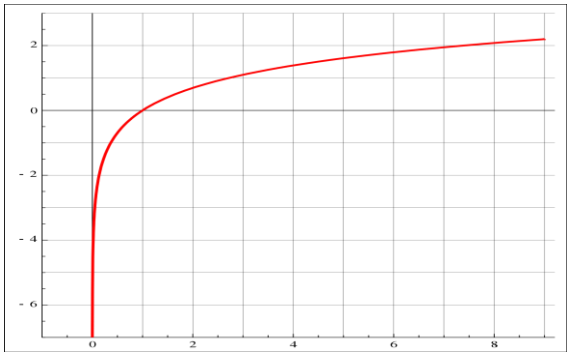
For the network to learn, a criterion is required to define the loss/error of the network's results. This is quite simple when one is dealing with continuous values, since there are many simple loss functions such as mean squared error, mean absolute error, etc. But for such kind of a task we need something different. The output tensor doesn't usually take the value of 0 or 1, but something in-between, resulting in it being a probability instead of a straight answer. Simply put, the network actually answers the question "What is the probability of a crossroad being in front of the car?". This probability is then rounded into the binary answer (with a custom threshold of 0.8 instead of 0.5, since we need it to be confident enough). The loss function used for binary classification is called Binary Cross-Entropy Loss (also called Log Loss), or BCELoss in short.

$L = - 1/N \Sigma[yi * \log(yi') + (1-yi) * \log(1-yi')]$

Where L is calculated loss, N the output size, yi the training label (correct output value), yi' the predicted probability value (network's output).

Note that by log we mean ln, natural logarithm of base e (Euler's Number) and not base 10.

Logarithms of base e are used quite a lot and with good reason, since we are dealing with probabilities. Loss calculates how wrong the network is, and in logarithm of base e, the closest to 1 the value is, the closest to 0 the loss tends to be (we take the negative of the summary calculated since we need loss to be positive). Note that we do not care for positive logarithm output (when x is greater than 1) since the output yi' is clamped between 0 and 1.



3.3.B Road Edge Detection – Dealing with Poly-lines

For this task, many possible ways to represent road_edge lines were thought of. To define the problem, we need to extract road edges from an image. A road edge is basically the boundary of the drivable space, Tesla uses it along with many other tasks to create a map of the environment. It is useful for crossroad detection, since the slight changes on the slope and the curves of the detected road edges can be used to determine whether there is a crossroad in front of the car or not. Good and robust road edge detection can also prove that actual crossroad detection is not really needed in it of itself, simply because the autonomous car only needs to know the shape and the boundaries of the road it drives on. Knowing whether the shape of the drivable area creates a crossroad or not is not really necessary. However, in this project the resources for building a robust and high-performance road edge detector were not present, although given the small amount of data that was used, the results were quite decent. This thesis' goal and main task is crossroad detection/recognition and how it can prove useful to other self-driving car deep learning tasks, so it was later on used to train along with road edge detection (multi-task learning) in hope that it might help the actual road edge detection performance. In general, crossroad detection in it of itself might not be as useful for the end user, but it can help improve the performance of other neural network tasks.

In highway driving, to represent the edges of the road, only two lines are needed. When dealing with crossroads however, more are needed. For a simple crossroad we can assume that 4 lines will be needed. In this project the maximum amount of road edge lines that was used is 6 (assuming that the road the car is driving on has 2 separate lanes with opposite flows, so we are counting in the possible obstacle that separates them), but a max number of 8 road edges can be found in some crossroads when both the roads that cross each other have 2 lanes of different flow each. The cartesian coordinate system was used for representing the points of the lines in the 2D plane of the image (x,y). Each line has 4 points, which proved to be the minimum number needed to represent the curves of crossroads. So, a 3D array of shape [6, 4, 2] is how the road edges are perceived by the algorithms used in this project. To transform this matrix into a proper output vector for the network all we have to do is flatten it, serialize it to a vector of length 6*4*2=48 floating point numbers. A function for deserialization was also needed, so that the lines could be properly processed and drawn onto the display. So in conclusion, this task is basically a regression of the various coordinates of the points that belong to the lines of the road edges.

For the loss function, a custom negative log likelihood loss was used (NLLLoss from PyTorch was incompatible for the networks). It was suitable for the Bayesian layers that were used, since linear layers with MSELoss proved inadequate for the given task. This algorithm calculates loss by using the negative natural logarithm of the given probability/likelihood, which is the output of the model.

```
# negative log likelihood loss
def neg_log_likelihood(output, target, sigma=1.0):
    dist = torch.distributions.normal.Normal(output, sigma)
    return torch.sum(-dist.log_prob(target))
```

In the image above we have the custom implementation of this loss, since NLLLoss gave some errors and was probably designed for classification tasks while we need it for regression. As it is shown, the loss takes the output of the network and creates a normal distribution with sigma=1.0 (mean=output, std_deviation=sigma). Since output is a vector with multiple values, the loss function calculates each values' natural logarithm (ln not log), sums them up and returns the negative sum. This loss in combination with Bayesian layers has shown performance far greater than MSELoss with linear layers.

3.3.C Path Planning

Path planning is the most crucial task for autonomous driving. It is the final decision of where the car should go. In this thesis, path planning was simple and most of the data was driving on a straight line.

We represent the path in the 2D image as a single line or curve, depending on the actual path the car must follow. The way path data is handled is very similar if not identical to how road edge lines were handled in road edge detection. In this task, the network tries to predict a single line, although some research has shown that predicting multiple paths and choosing the best of them could prove to be a much better solution, since it is closer to how an actual human would plan where the car should go. Because the task is very similar to the previous one, a lot of the algorithms from before were used. Here, a 3D matrix is used to represent the path curve, with a shape of [1, 8, 2], meaning 1 line with 8 points of 2 coordinates (x,y). This matrix could also be 2D but the 3rd dimension was needed to match the algorithms for the road edge lines. This also achieves scalability since it can be expanded to multiple paths instead of just one.

To match the network's output, the path label is serialized/flattened into a vector of length 1*8*2=16 floating point values, each representing a coordinate of a corresponding point. The actual output of the convolutional neural network is deserialized back into a 3D matrix in order to be more suitable for further processing, like getting drawn into the 2D image.

Since we are dealing with crossroads however, just predicting a line is not enough. Each crossroad has multiple possible paths, so we could either have the network output more than one path, as proposed above. This can also be solved by introducing a concept called desire. Comma.ai uses it in highway driving as an input from the end users when they want to change lane. For this project, desire was used to choose the direction the driver wants to go. It can either be left, right or forward. Desire is appended to the network's input after the convolutional layers, when the input itself is flattened. It is represented by a vector of length 3, which uses one hot vector encoding. In this type of encoding, we have 3 possible values desire can take: 0, 1 or 2 (forward, right and left), but instead of using a single value (vector of length 1), we use a vector of length equal to the max value + 1, meaning 3 for this task. So, value 0 will be represented as [0, 0, 1], 1 as [0, 1, 0] and 2 as [1, 0, 0]. This method allows desire to affect the input a bit more, increasing it's importance. Unfortunately, this path planner neural network was not trained on turns, so desire is constantly 0 (or [0, 0, 1]). The value of desire could be determined by the user by the blinker lights (although this will not make the car fully autonomous) or by using GPS commands.

The loss function that was used was once again negative log likelihood for the exact same reasons it was used for road edge detection. However, MSELoss with linear layers could also work quite well for this problem, since predicting a single curve that is usually a straight line is much more simple than multiple curves with different directions.

CHAPTER 4 Tools and Libraries

4.1 OpenCV and PIMS for image processing

Autonomous driving is believed to be mainly a computer vision problem. Humans drive using their eyes for perception, thus vision is the only necessary sense that is required to decide how to drive a car. Other solutions include the use of LIDAR and other expensive sensors to create HD maps of the environment, but this is simply unnecessary since cameras are much cheaper and more efficient for self-driving. This achieves lower cost for production and sale with the added benefit of emulating exactly how a human perceives the environment while driving.

Since in this project the data consists of videos from a dashboard camera (monocular driving data), a library for image processing as well as higher level computer vision tasks is required. This is where OpenCV is very useful, since it provides a great set of utilities and functions for python, allowing for many computer vision tasks to be achieved. From simply opening an image and preprocessing it for the neural networks to displaying the video and drawing the different tasks they are used for (drawing the road edges it detects, the planned path, whether or not the network sees a crossroad ahead, etc). For the deployment application, we can also display the frames per second (FPS) to see how fast those neural networks perform their tasks, in order to confirm whether their speed and efficiency or the hardware is acceptable for use in the real world. Note that self-driving cars require high performance and robust algorithms and hardware in order to avoid malfunctions and cause crashes. The whole application stack needs to operate in high speeds. In this project, 30 FPS is the average, running on a Ryzen 7 3700x CPU and an RTX 2060 6GB GPU. OpenCV is written in C++, which is a very fast compiled programming language, and also provides an Application Programming Interface (API) that connects it's functions with python. That makes it the perfect choice for matching the speed requirements of the perception tasks.

OpenCV is used mainly for opening the video files, processing the individual frames so that they can be used properly by the convolutional neural networks (input stack) and for displaying the results in video during the deployment of those networks (output stack). The whole deep learning stack sits somewhere in the middle and is the core of the project, but this will be further elaborated on in a later sub-chapter. For the data stack, the images were labeled using OpenCV for the simple task of crossroad detection and for adding desire label for path planning. For more complex annotations, Computer Vision Annotation Tool was utilized since it was more useful for drawing and tracking lines in video data for path planning and road edge detection.

Specifically, for opening the images PIMS was used. Since we are dealing with video data, we need to extract all individual frames and add them to a list. To save some lines of code, the PIMS library can extract frames so that we can parse, preprocess and add them to a list.

```
frames = pims.Video(video_path, format="mp4")
```

However, PIMS reads files in BGR format. To make this project more versatile and global, images are converted to RGB, just in case they are fed into

the network without PIMS. This also allows OpenCV to display the frames with proper colors.

```
# make pims video into actual numpy frames
def conv_frames(frames):
  imgs = []
  print("Getting frames into proper arrays")
  for frame in frames:
    imgs.append(cv2.resize(cv2.cvtColor(frame, cv2.COLOR_BGR2RGB), (W,H)))
  print("Frames converted to numpy arrays")
  return np.array(imgs)
```

Frames are converted to NumPY arrays since they are closest to PyTorch's tensor format, which makes the conversion easier. After color conversion, the images are also resized/downscaled to a more manageable resolution (320x160), to save up RAM and computational resources, while making sure that all images have the same resolution. After that, the images are fed to the network after being converted to PyTorch tensors.

OpenCV is also used for displaying the different curves of road edges and path using:

```
def draw_polylines(frame, polylines, color=(0, 0, 255)):
  for polyline in polylines:
    polyline = np.array(polyline)
    x, y = polyline.T[0], polyline.T[1]
    frame = cv2.polylines(frame, np.int32([polyline]), False, color, 2)
  return frame
```

For labeling crossroads in videos, we first read the mp4 file and parse it's individual frames. Then, every frame is displayed in a sequence (as it would if we were watching the actual video), pausing at each one so that the user can label it. If the key C is pressed, the frame is labeled as crossroad (1), otherwise if any other key is pressed (apart from Q which aborts the process) then the frame is labeled as no-crossroad (0).

The pause is achieved using key = cv2.waitKey(0), and the keystroke handler code is the following:

```
key = cv2.waitKey(0)
# if key pressed is 'c' then crossroad detected, if key is 'q' stop, of key is other
if key & 0xff == ord('c'):
  label = 1
  labels.append(label)
elif key & 0xff == ord('q'):
  break
else:
  label = 0
  labels.append(label)
print("Label", label)
```

The labels are then saved in a .txt log file and are then opened by the training scripts. Each line contains one label, meaning each line represents the label of the corresponding frame.

4.2 NumPy

NumPy is an open-source library for python, written in C programming language, that enables users to do numerical computing while also achieving speed (C code is executed way faster than Python). It provides a wide range of utilities that proved useful for this project.

Most of the data is represented by matrices of static sizes and shapes. Python does not provide arrays, unlike many other programming languages. Instead, it has lists, which are quite useful for general purpose programming since a connected list is a data structure similar to an array, with the added benefits of having a variable length. This means that we can define a list of several data values, such as numbers, and later on add or remove them, while changing the length of the list itself.

In deep learning however we are dealing with tensors and vectors, which have specific static lengths and shapes, such as an image resolution. This makes the need of lists insignificant, especially given that operations on them are more complex than in arrays (parsing, adding and removing data from a list requires more code, while such operation on an array don't), making them performance inefficient.

So, most of the data in this project are static arrays, vectors and matrices. This is where NumPy comes in. This library does not only provide functionalities for converting python lists into static arrays, making the code faster and more memory efficient, but also for various operations on them, saving the user from writing unnecessary code for operations such as matrix multiplication, changing the dimensions of a 3D array (such as an image), etc.

To create a NumPy array in python one must first create a list and then call the corresponding function from the library:

```
import numpy as np

p_list = [1, 2, 3, 4, 5]
arr = np.array(p_list)
```

A small added benefit of high-dimensional NumPy arrays is that they are printed by default in a way that is easier to the eye, making debugging a smoother process.

PyTorch works incredibly well with NumPy, since numpy arrays can be quickly converted into tensors and then fed to the neural network in use. An input image that was read using OpenCV will have a shape of Height x Width x NumberOfChannels and given that training is done using batches of images (multiple images are fed into the network at once) the shape becomes BatchSize x Height x Width x NumberOfChannels. PyTorch however requires the following shape: BatchSize x NumberOfChannels x Height x Width. To swap the dimensions in order to match the shape needed, one could write a custom made algorithm that would require precious time and resources (Python code is not really fast in comparison to other languages). Instead, one could use the NumPy function moveaxis() that will do exactly that, with the added benefit of speed since NumPy is written in C and, like OpenCV, provides an API for Python.

```
frame = np.moveaxis(frame, -1, 0) # [batch_size, channels, height, width]
```

This library can also be used to create random samples for training (in order for the network to parse the training data in a non-serial way, which can make it more robust since it "sees" the images in a random order during each epoch).

```
rng = np.random.default_rng()
samp = rng.choice(len(frames)-1, size=BS, replace=False)
for j in samp:
  frame = frames[j]
```

Since training is done using multiple videos, we must extract all frames from all videos and concatenate them together into an X_train array. NumPy provides that functionality as well:

```
all_frames = np.concatenate((all_frames, frames), axis=0)
all_annotations = np.concatenate((all_annotations, annotations), axis=0)
```

The above code is similar to appending the frames of a new video to the array of the previously processed ones, but with numpy arrays instead of python lists.

4.3 PyTorch

Neural Networks require a lot of code. Even by using NumPy for most of the matrix computations, one would need to write thousands of lines of code in order to define the neural networks that are to be used. This is why deep learning frameworks are written. Instead of worrying about how the various layers work, the developer only has to worry about the overall architecture of the models. All one has to do is define a neural network architecture and use it either for training or deployment applications. The deep learning framework that was chosen for this project is PyTorch, although were are other choices taken into consideration such as Tensorflow.

PyTorch was inspired by the Torch deep learning framework for the Lua programming language. However, it was not widely used since a lot of programmers did not prefer Lua for their projects. Later, an AI research team was inspired by this library and implemented it in python, calling it PyTorch. Nowadays, it is developed and maintained by Facebook's AI Research lab (FAIR).

There are two basic types of programming, imperative and symbolic. The first one uses an interpreter and code is read and executed at run-time (just like in python), while the second one uses a compiler that reads the symbols of the language and outputs a binary file to be run (like in C/C++). Imperative programming is more dynamic in nature while symbolic programming is more static, this allows for python code to be more flexible during run-time. Even though symbolic programs are more efficient and allow value reusability, this project required imperative programming's flexibility.

The Tensorflow framework belongs to the symbolic programming category, even though it is used in python. It's definitions are more strictly written and a lot of tensorflow-specific functions are used. PyTorch on the other hand uses imperative programming, making it more natural for a python developer. It is better suited for research projects, due to it's flexibility, while Tensorflow is widely used for deployment in the real world. However, Tesla has been using PyTorch for the whole deep learning stack of Autopilot, which is performing at high speeds, and recently Comma.ai has converted all of it's code from Tensorflow to PyTorch claiming that they saw more benefits in that choice. PyTorch also provides a C++ library that utilizes Torchscript and achieves high speeds, which is possibly how Tesla chooses to deploy their huge models (about 48 networks with 1000 distinct predictions). However, C++ was not used for this project since the main goal was research and not deployment speed, even though self-driving cars require high performance for safety and responsiveness.

Another key feature for PyTorch is Dynamic Computation Graphs (Tensorflow provides Static Computation Graphs), which makes it a define-by-run library, meaning that the neural network's graph is generated at run-time. Tensorflow is a define-and-run library, which basically writes the whole program/neural network graph and then runs it (it is defined and assembled once and can then be run multiple times). While static graphs work better for fixed-size networks, dynamic ones work even better for more dynamic networks such as Recurrent Neural Networks (which are heavily used on video data, but they were not used for this project).

PyTorch makes debugging a bit easier, since the developer is writing more python code instead of using the framework's compiled functions. This allows for a more under-the-hood perspective that proved quite useful for this thesis' research.

To define a network, one must declare it as an ordinary python class that inherits PyTorch's torch.nn.Module. The architecture is defined in the __init__() function, where the various layers are assigned to class variables/attributes (for example self.linear1 could be the first linear layer of the model). Then, the developer needs to define the order of the network's layers, i.e. the flow of the input X through the network itself. That is done in the member function/method forward(self, x) which is called every time input is fed into the network, whether it is during training or inference. There is also another way to define the order of the layers, by using the Sequential function:

```
class Net(nn.Module):
  def __init__(self):
        super(Net, self).__init__()

        linear1 = nn.Linear(input_neurons, output_neurons)
        linear2 = nn.Linear(input_neurons, output_neurons)
        linear3 = nn.Linear(input_neurons, output_neurons)

        activation = nn.functional.Relu()

        self.network = Sequential(linear1, activation,
                                  linear2, activation,
                                  linear3)

  def forward(self, x):
        return self.network(x)
```

This simplifies the model definition process a bit, however removing some control over the flow (which was needed for adding desire in path planning). Sequential was proven useful for the Multi-Task model, which will be discussed later on. Since PyTorch was used for most of the codebase, the specific functions used for training and deployment will be explained later on as well, in the code overview.

4.4 BliTZ

As mentioned in earlier chapters, Bayesian Neural Layers, which introduce the concept of weight uncertainty in neural networks, were used for regressing the various points of curves for road edge detection and path planning. PyTorch does not offer this type of layers by default, so there is a need for a wrapper library that extends PyTorch, allowing for such functionality. This is were the library called Bayesian Layers in Torch Zoo (BliTZ) was proven quite useful.

BliTZ is designed to work with PyTorch by simply importing the library and adding @variational_estimator above the model's class definition. This allows the developer to use it's various Bayesian Layers the same way PyTorch's default layers are used. The library provides various types of Bayesian layers, such as BayesianLinear, BayesianConv1d, BayesianConv2d, BayesianConv3d, BayesianLSTM, BayesianGRU, etc, however only BayesianLinear ones were used for this project.

These layers are used for the path planner and road edge detector, right after the convolutional layers, and manage to achieve better accuracy and loss than their counterparts. As mentioned before, the loss function of choice for these layers' tasks was a custom written negative log likehood.

```
# Bayesian Layers
self.blinear1 = BayesianLinear(2048, 512)
self.b_bn1 = nn.BatchNorm1d(512)
self.blinear2 = BayesianLinear(512, 128)
self.b_bn2 = nn.BatchNorm1d(128)
self.blinear3 = BayesianLinear(128, self.n_coords*self.n_points*self.max_n_lines)
```

```
def forward(self, x):
    x = self.avgpool1(self.elu(self.bn1(self.conv1(x))))
    x = self.layer1(x)
    x = self.layer2(x)
    x = self.layer3(x)
    x = self.layer4(x)
    x = self.avgpool2(x)
    #print(x.shape)
    x = x.view(-1, self.num_flat_features(x))
    x = F.relu(self.fc_bn1(self.fc1(x)))
    x = F.relu(self.b_bn1(self.blinear1(x)))
    x = F.relu(self.b_bn2(self.blinear2(x)))
    x = self.blinear3(x)
    return x
```

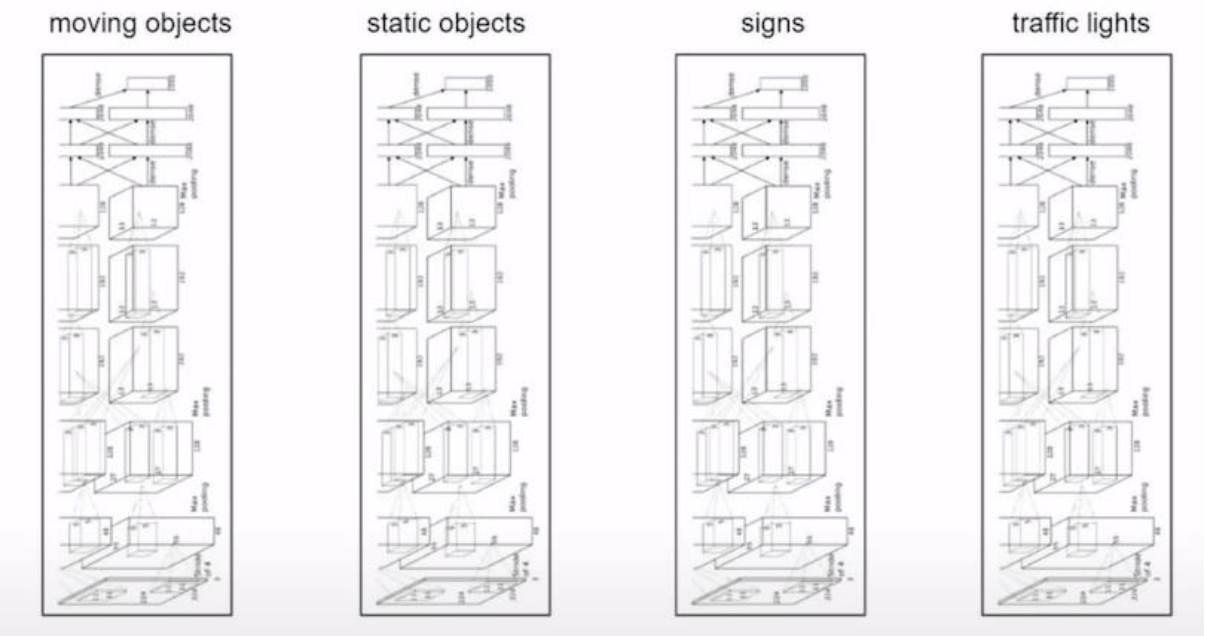
As shown above, Bayesian Linear Layers can also be combined with basic PyTorch Linear Layers, and can be used with activation functions such as ReLU with no problems.

CHAPTER 5 Multi-Task Learning

5.1 Neural Networks for multiple Tasks

There are many tasks in this project that require neural networks (crossroad detection, road edge detection and path planning), while in real world self-driving cars, companies like Tesla have about 1000 distinct predictions/tasks. But having a single neural network for each and every one of those tasks can be quite expensive computationally, which is sub-optimal for real world deployment since high speed performance of the deep learning stack is required for safety and practicality.

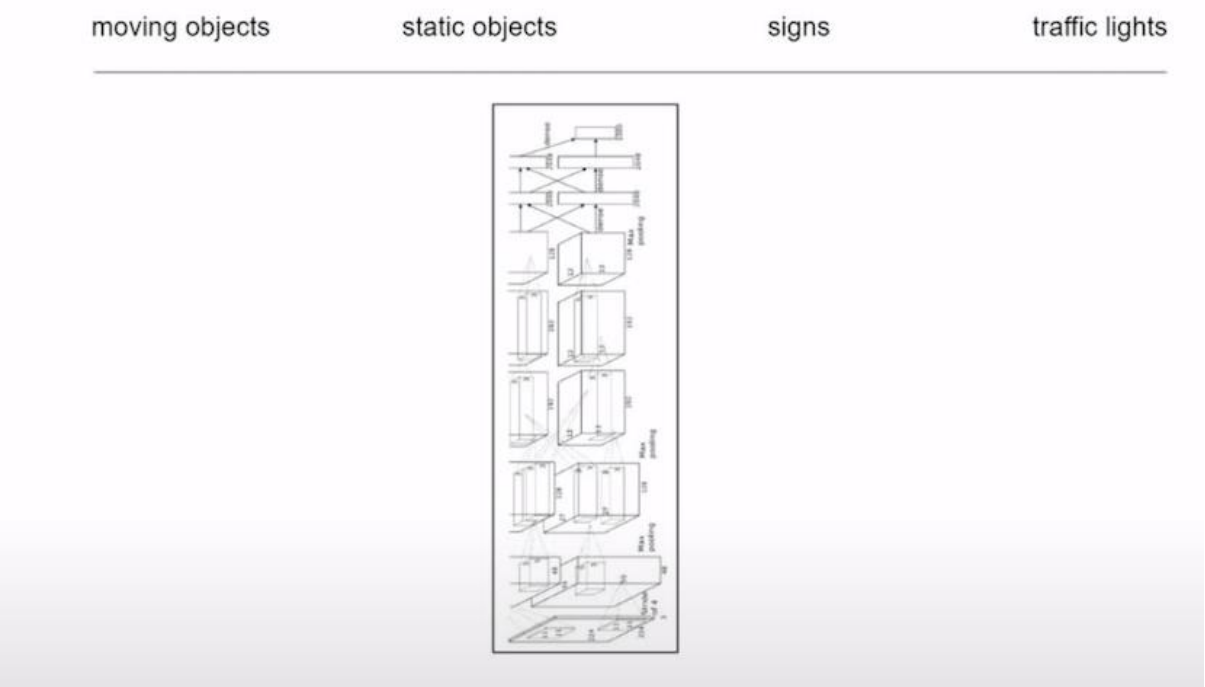
Suppose our three tasks have a single neural network each (which has been implemented in code). This allows for training individual networks separately, which results in better team workflow for each task (every person responsible for each task has to worry only about their job, since it doesn't affect others), better focus on each task and easier optimizations. Retraining and improving a single task will not have an impact on the rest of the deep learning stack and there is not need to re-evaluate all of the tasks. However, the performance cost can be quite expensive and training each task separately might result in the overfitting of some networks, since they might lack some data for their specific task.



Single-Task neural networks for each individual task (from Andrej Karpathy's presentation on multi-task learning).

In such a small project like this, where the tasks are just 3 this might not be noticeable, but in large scale production this can prove problematic. Even Comma.ai, whose autonomy solution does not require so many predictions for deciding the driving policy of the car, does not use a single network for their tasks (lane lines detection, road edge detection, stop sign and traffic lights detection and path planning among others).

So the cons of using a neural network for each separate task outweigh the pros for achieving combined real-time predictions while maintaining performance. Another approach is to use a single neural network for all tasks, where only the output vectors are different to match each task's needs. This would surely increase the overall performance speed for all predictions due to the fact that only one network is being used, making it cheaper at test time and deployment, however the various tasks will "fight" for the networks capacity. This does not happen all the time since their "relationships" can be quite complex, some tasks work well together, resulting in a synergy of accuracy (meaning that while one task improves the other does so as well, not necessarily at the same rate), while others are counterproductive to each other. This is because some predictions require some specific values from the network's learnable parameters (weights and biases), which can either benefit or work against others' requirements. Another problem would be the network's maintenance and optimization. A team of developers, each one working on a task or a set of tasks, would find it difficult to retrain and finetune the network, because one simple change for one prediction will affect all the others.



A single neural network that outputs tensors for multiple tasks (from Andrej Karpathy's Multi-Task learning presentation).

The head of Tesla's Autopilot AI team, Andrej Karpathy, gave a really interesting presentation on how the company uses multitask neural network architectures for their many predictions and how the whole team works on multiple tasks in the same models. In addition, some recent papers have shown research on the different relationships between various deep learning tasks, and how they affect each other in the same network. So, a solution to the above problems lies somewhere in the middle, where synergizing tasks share parts of the same network and then branch on their own. This creates another problem: the choice of grouping of the various tasks and what capacity of the networks they share. This is more of a trial and error problem, so one can experiment with many model architecture and, depending on the metrics, change them to better suit the tasks. There has also been some research on Neural Architecture Search for multi-task learning, which allows the automation of the models' architecture selection, however it is in an early state.

5.2 The ComboModel

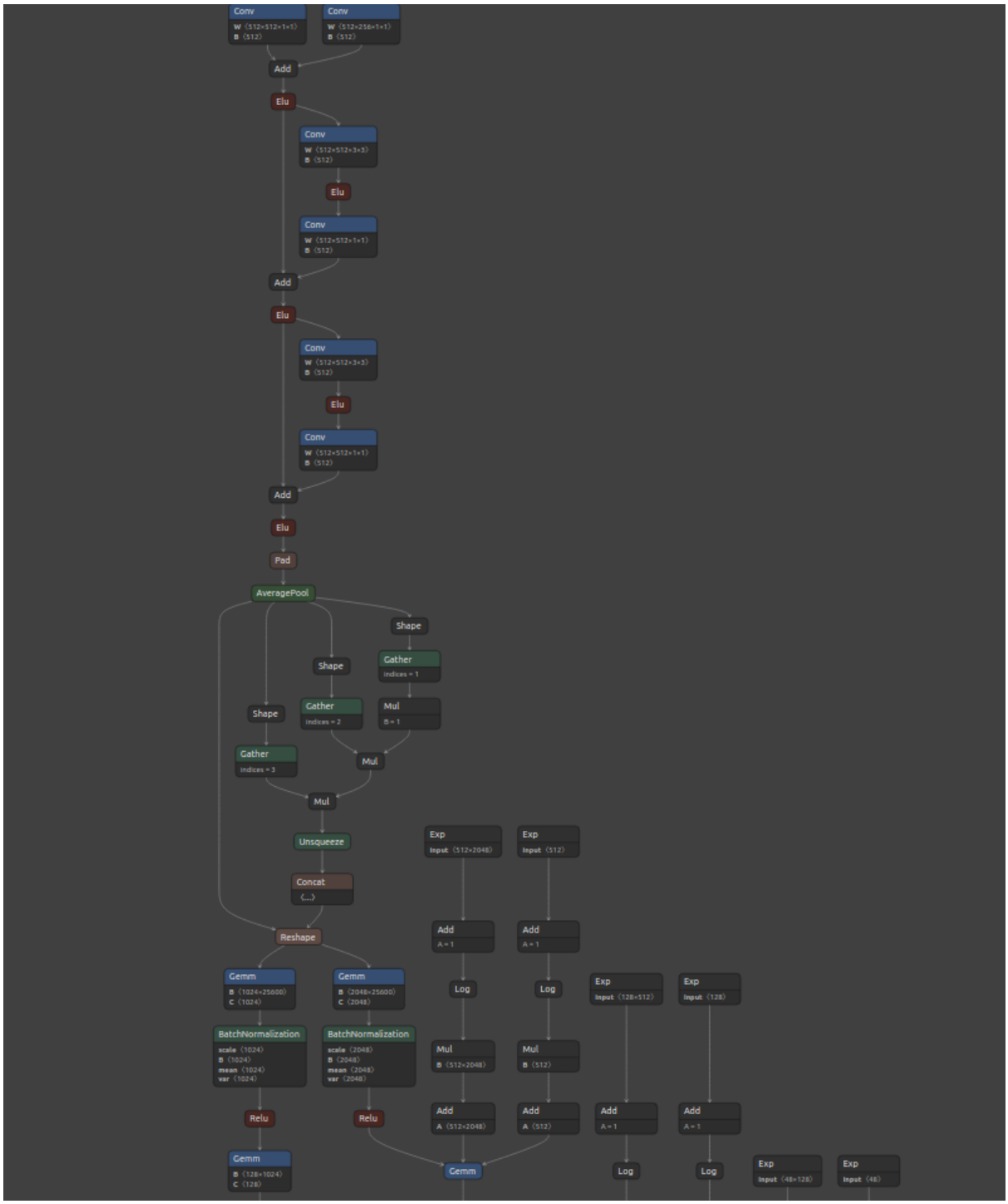
Multi-task learning has proven quite useful for self-driving cars since it saves some computational resources, increasing performance, while in some scenarios even increases the accuracy of some tasks. Thus, if done right, so that the relationships between grouped tasks not only are not damaging to each other, but help improve each other's performance, multi-task learning could provide an optimal solution not only for autonomous vehicles but for many other complex deep learning problems as well. So, this makes it really interesting regarding this thesis' area of study, which is why it was explored.

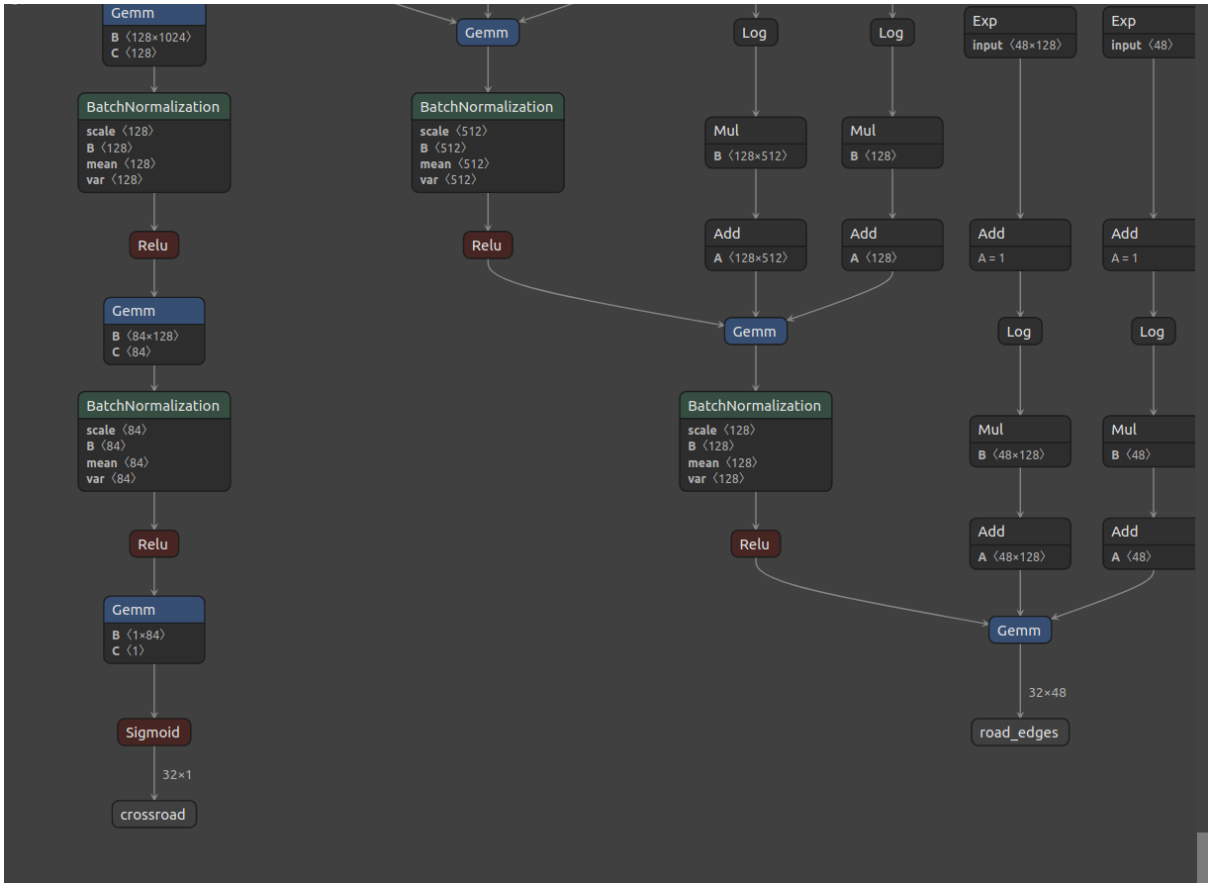
The idea was to check if crossroad detection can help improve other tasks such as road edge detection. Comma.ai uses multi-task learning in a much more simple but almost equally effective manner than Tesla. Their SuperCombo model, which is responsible for driving the car, outputs multiple predictions for lane lines, road edges, lead car detection and ofcourse path planning, by providing the tasks with one shared backbone of a large network. That backbone is the convolutional neural network part, that is responsible for detecting features from the video data. The rest of the network is separate for each task, meaning that linear and other types of layers are handled by each task individually. This is the approach that was followed in this project.

There are three tasks (road edge detection, crossroad detection and path planning) that share the same convolutional backbone, i.e. a ResNet18, but then split to their separate linear and Bayesian layers that classify or regress the training data. The main idea was that if a human was to detect whether there is a crossroad or not, they would check the same visual features they would for the edges of the road (maybe traffic lights, stop signs as well among others), and vice versa. Naturally, many tasks that seemingly should work together in synergy due to their similarities, improving the performance of each other, might not do so in practice. That's why experimentation and statistics from metrics are really important for deciding the final architecture of a multi-task neural network. The variables are too many to consider, such as which tasks to group together, how much of the network should they share, etc, so for the sake of this thesis it was important to keep it simple. Thus, simple tasks theoretically similar to each other were picked and have just the convolutional backbone in common. Ofcourse there could be more optimal architectures for this problem, but for the sake of simplicity the key idea of the ComboModel (as it was named due to it's inspiration from Comma.ai) is "Put crossroad and road edge detection together so that they focus on the same visual features and add some basic path planning along with them, in order to see if overall performance is improved". When referring to performance for multi-task learning, one must take into account that it should be the ratio of computational resource required, the speed of execution in real-world data and the accuracy of the network on that same data. Thus, there are too many variables to take into consideration, this is one reason why the concept of multi-task learning is currently under research, along with many other aspects of deep learning.

For the loss function, the most basic solution would be to add the losses of each task together and backpropagate the summary. This can give some satisfying results, however there are better ways to achieve more optimal results. Another idea would be to weight the losses depending on the importance of each task. This

will result in the task whose loss has the highest weight to dictate how much the network should change at a higher capacity. The choice of the weights depends on experience and experimentation. There are some considerations one must take into account when deciding the loss function in a multi-task model, such as the fact that some tasks have loss functions on different scales (classification vs regression), some tasks are more important, much easier than others and/or have more data and noise in their data than others.






```python
# Loss wrapper for multitask model
class ComboLoss(nn.Module):
    def __init__(self, task_num, model, device):
        super(ComboLoss, self).__init__()
        self.task_num = task_num
        self.model = model
        self.device = device
        self.log_vars = nn.Parameter(torch.zeros((task_num)))

    def forward(self, preds, cr, re, path):
        cr_loss = nn.BCELoss()
        re_loss = nn.MSELoss()
        path_loss = nn.MSELoss()

        loss0 = cr_loss(preds[0], cr)
        #loss1 = re_loss(preds[1], re)
        #loss2 = path_loss(preds[1], path)
        loss1 = neg_log_likelihood(preds[1], re)
        loss2 = neg_log_likelihood(preds[2], path)

        precision0 = torch.exp(-self.log_vars[0])
        #loss0 = precision0*loss0 + self.log_vars[0]

        precision1 = torch.exp(-self.log_vars[1])
        #loss1 = precision1*loss1 + self.log_vars[1]

        precision2 = torch.exp(-self.log_vars[2])
        #loss2 = precision2*loss2 + self.log_vars[2]

        loss = loss0 + loss1 + loss2
        #loss = loss.mean()

        return loss.to(self.device)
```

5.3 Multi-task Learning in PyTorch

PyTorch makes defining and using a multi-task neural network simple, since it gives the developers more control over the code execution and it is natural to the Python programming language. The following is an explanation of the definition of the ComboModel, which takes a 320x160 image and the desire (where the driver wants to go in the crossroad, 0: forward, 1: right, 2: left, note that desire is not fed into the network from the beginning since it is not required for all the tasks) as it's input and outputs it's crossroad detection (binary classification), the road edges it "sees" and the path it believes the car should follow (both of these tasks require regression of the curve point's coordinates). As mentioned before, these tasks share a ResNet34 convolutional backbone and then proceed to their own linear and Bayesian layers. Note that the number of layers for ResNet may change during optimization.

```
# Multitask Model
class ComboModel(nn.Module):
  def __init__(self, num_layers=18, block=ResBlock, image_channels=3):
    assert num_layers in [18, 34, 50, 101, 152], "Unknown ResNet architecture, number of laye
    super(ComboModel, self).__init__()

    # road edge polylines' shape
    self.re_n_coords = 2  # 2 coordinates: x,y
    self.re_n_points = 4  # number of points of each polyline
    self.re_max_n_lines = 6 # max number of polylines per frame

    # path polylines' shape
    self.pth_n_coords = 2  # 2 coordinates: x,y
    self.pth_n_points = 8  # number of points of each polyline
    self.pth_max_n_lines = 1 # max number of polylines per frame

    self.num_layers = num_layers
    self.cnn_output_shape = 512*5*10  # shape of the output of the cnn (input for fully conne
```

Here, some useful variables are defined. First, the code checks if the number of layers for the ResNet backbone is valid (it is 18 by default but a user can change it, the hardware used was not capable of training with more layers efficiently). Then some variables for the outputs of road edge detection and path planning are defined (the number of curves, the number of points every curve has and the number of coordinates of each point). The cnn_output_shape variable determines the shape of the output of ResNet34, which is used by the first linear layer of each task.

```
# ResNet Layers
self.layer1 = self.make_layers(num_layers, block, layers[0], intermediate_channels=64, stride=1)
self.layer2 = self.make_layers(num_layers, block, layers[1], intermediate_channels=128, stride=2)
self.layer3 = self.make_layers(num_layers, block, layers[2], intermediate_channels=256, stride=2)
self.layer4 = self.make_layers(num_layers, block, layers[3], intermediate_channels=512, stride=2)

self.avgpool2 = nn.AvgPool2d(1, 1)

# Fully Connected Layers
self.cr_head = self.get_cr_head()
self.re_head = self.get_re_head()
self.path_head = self.get_path_head()
```

After that, the actual layers of the whole network are defined (in PyTorch the network's architecture is always defined in the constructor __init__ of the class). Each ResNet layer represents a block (ResBlock) which is defined in another class (note that a ResNet34 is provided by PyTorch and the whole code was not really necessary for the network's definition, however for the purposes of studying the actual architecture, it was manually coded). Each ResBlock is basically a Residual Block that performs a few convolutions and batch normalizations, while adding the block's input to it's output, as explained in previous chapters. The make_layers function creates the residual blocks with 18 (by default) layers of specific intermediate channels and stride. In ResNet18, the number of residual block in each of the 4 main layers is 2.

```
def make_layers(self, num_layers, block, num_residual_blocks, intermediate_channels, stride):
  layers = []
  identity_downsample = nn.Sequential(nn.Conv2d(self.in_channels, intermediate_channels*self.expansion, kernel_size=1, stride=stride),
                                      nn.BatchNorm2d(intermediate_channels*self.expansion))
  layers.append(block(num_layers, self.in_channels, intermediate_channels, identity_downsample, stride))
  self.in_channels = intermediate_channels*self.expansion
  for i in range(num_residual_blocks - 1):
    layers.append(block(num_layers, self.in_channels, intermediate_channels))
  return nn.Sequential(*layers)
```

This architecture relies on the Sequential() function to save some unnecessary code for forward(self, x). For a visual representation of the network's whole architecture, one can open the .onnx model file with specialized software such as Netron.

The functions get_cr_head(), get_re_head() and get_path_head() define the layers for each task (same way they are defined in their single-task neural networks) and return them using Sequential()

```
def forward(self, x, desire):
  x = self.avgpool1(self.elu(self.bn1(self.conv1(x))))
  x = self.layer1(x)
  x = self.layer2(x)
  x = self.layer3(x)
  x = self.layer4(x)
  x = self.avgpool2(x)
  #print(x.shape)
  x = x.view(-1, self.num_flat_features(x))
  x_desire = torch.cat((x, desire), 1)
  cr = torch.sigmoid(self.cr_head(x))
  re = self.re_head(x)
  path = self.path_head(x_desire)
  return [cr, re, path]
```

The above code is the forward function of the whole network. The input image x goes through the ReNet18 layers and gets flattened into a 1D tensor. It then is passed to the crossroad and road edge detection heads (as their separate set of layers is called in such models) as it is. For path planning, x is concatenated with desire (which is one-hot vector encoded instead of a single value) and fed to the path head. After that, the predictions for each task are returned in a list, representing the outputs for each task of the ComboModel.

CHAPTER 6 Training Code Overview

6.1 Training for Crossroad Detection

The training process of the neural networks is pretty straight forward. In general, any training script includes acquiring the training dataset, preprocessing the given data and training the model with it, by feeding the whole data through the network (forwards and backwards) multiple times. Each iteration of the whole training data is called an epoch. Multiple epochs can take place during training, it is up to the developer's experience and experimentation. The goal of the whole process is for the network to learn how to match the given data as accurately as possible by optimizing it's parameters, while also being able to make accurate predictions on new data similar to the one it has been given. This is why data and the model's power so important, since one needs a model powerful enough to fit the given data (but not too powerful to avoid overfitting) and needs a lot of data with lots of variety (for example many images of different crossroads) in order to accurately describe what the model should do.

All of the data consists of .mp4 video files, therefore all the frames from each file need to be extracted, preprocessed and put in a big array of all frames from all videos. The corresponding labels need to be loaded as well, while keeping the matching indexes with the frames they describe. For crossroad detection, the label of each frame can be 0 or 1 (no-crossroad, crossroad). This happens in the function get_data(), which opens the log file for the binary labels of each frame (each line is the frame index from the corresponding video file). The naming of data files plays an important role, since the videos and their log files containing annotations, labels, etc have similar names in their beginning. They are opened together, using some python string processing functions. The frames are extracted from the videos using the PIMS library, in order to create a frames object that returns all frames without consuming too much memory. HD frames cannot be processed since they require a lot of memory, so using them as they are would be ineffective. They need to be scaled down to a new more manageable resolution (320x160), that consumes a lot less RAM, allowing the developer to add more data. In general, memory plays a big role when training deep learning models, since it determines how much data can be loaded and passed to the neural network in one session. Fortunately, 32GB of RAM was available during the implementation of this project (about 20GB were used by the training scripts, since frame arrays are expensive in memory). PIMS has also one more problem, it loads the images in BGR while OpenCV processes RGB frames mostly, so another convertion is also needed. This happens in the function conv_frames(), which takes the PIMS frames object, parses it's items and converts the frames properly while appending them to a list and then returning a numpy array of that same list.

To save a lot of time, training was done using a GPU (RTX 2060 6GB). The model used has a ResNet18 convolutional backbone, with 4 linear layers, each one followed by a batch normalization layer. The activation function for each linear layer was ReLU, while for the last one was sigmoid (since we need the output to be clamped between 0 and 1, for the binary classification task). For the convolutional layers of ResNet, the activation function used was ELU. The results of this architecture were more than good enough, meaning the model was actually

really powerful, so to avoid overfitting the number of epochs was kept low and the data high, resulting in high accuracy on training and evaluation data.

For the actual training process, the model needs all of the frames and their corresponding labels with their specific indices in order to train properly, so after using get_data(), the frames and their labels are appended/concatenated with the ones from all the other video files. The loss function of choice was Binary Cross Entropy Loss, since we are dealing with a binary classification problem. The optimizer used was Adam, since it has shown great results for the majority of deep learning tasks. In general, optimizers are various algorithms that aim to minimize the loss during training, thus being responsible for the actual learning of the model's parameters. The learning rate used was lr=0.001, which is chosen through experimentation. In general, one needs a learning rate high enough so that the model learns in a realistic rate, but not too high, because that could cause the network's loss to decrease rapidly at first, but remain the same after some time, making the learning stagnant. Only 11 epochs were required, due to the power of the model and the simplicity of the task, with a Batch Size of 128 frames. Generally, training a network by feeding it one image at a time is highly impractical and time demanding. Training in batches of images not only saves a lot of training time, but also allows the model to get better results, depending on the batch's variety of images and labels. Thus, random batches were sampled from the training data. Those samples are converted to PyTorch tensors, and are later on fed into the network. But before that, for every mini-batch we need to set the gradients to zero before backpropagation since PyTorch accumulates the gradients on subsequent backward passes. This is done using optim.zero_grad() (optim = Adam()). Then the input tensor X is fed to the model using out = model(X), the category predicted is extracted by rounding up the output (using a custom threshold of 0.8 to ensure that the network is more confident in it's predictions, thus accepting only it's most confident ones). Then, the accuracy and loss are calculated, and backpropagation is done by calling loss.backward() and optim.set(). After that, some functions are used for keeping the stats of each epoch, so that we can monitor the network's performance and make adjustments accordingly.

Evaluation is pretty straightforward. First, load the evaluation data into tensors (using batches). Then, feed them to the network, just like in training. Instead of backpropagating the data, we just keep the statistics of the model's performance on the newly seen data.

After that, the model is saved for later use, either to retrain it on more data or use it for deployment in an application.

6.2 Training for Road Edge Detection

Road edge detection is a slightly different kind of task. At it's core, it belongs to the category of regression deep learning tasks, however the challenge was processing the output tensor into 2D curves and defining a model powerful enough in order to learn the complex data. The second problem was dealt with by using Bayesian linear layers, as discussed in previous chapters in more detail. Ofcourse, the labeling of road edges could have been done in a better way, but it should suffice for some basic functionality.

As for the training script, first the necessary data is collected and stored into memory, specifically in NumPy arrays. The frames are extracted the same way as in the crossroad detection training script. The road edge curve annotations were a bit trickier to manage however. They are stored in .xml files and require a lot of preprocessing to be usable by the various scripts, either for training or deployment. At first, they are extracted from the xml file using an ElementTree from the xml library provided in python. This happens in the extract_polylines() function, which returns all the lines detected in a random way (they are sorted for easier processing by other functions) along with the index of the frame they belong to. Then, since for each frame we multiple polylines, the extract_frame_lines() returns an array whose indices represent each frame (the same way each index/line represents a frame index in crossroad detection log files). This ensures that for every frame[idx] we can get all the curves that correspond to it by using polylines[idx]. All annotations were drawn using CVAT (Computer Vision Annotation Tool) on 480x320 video files. The network requires them to match a 320x160 resolution. Resolution is important, because the network needs to learn the curve's coordinates for it's specific image parameters. They are later upscaled to a higher resolution along with the image (1920/2, 1080/2), for display in the deployment application. This conversion of the resolution for the annotations happens in convert_annotations(), by doing the following for each point's coordinates:

x_new = (x * new_W) / curr_W

y_new = (x * new_H) / curr_H

The frames and annotations are appended/concatenated to bigger arrays called all_frames and all_annotations, as they were in the crossroad detection training script, to ensure that all of the data from each video is passed at once to train().

The actual training process is simple, just like the previous script, however the annotations first need to be serialized into a vector in order to match the output tensor shape of the network. This is done by calling serialize_polylines(). The loss function used here is a custom version of negative log likelihood, which is explained in previous chapters. PyTorch's NLLLoss gave some errors since it is designed for classification tasks instead of regression. Training was done in 50 epochs with a batch size of 64. Note that in PyTorch models, each input image needs to have the shape of [batch_size, number of channels, height, width], which is ensured by using numpy.moveaxis(image, -1, 0), since the number of channels is at the end when opening the frames with PIMS and OpenCV.

At the end of the script, various stats, such as the loss for every epoch, are calculated and printed.

6.3 Training for Path Planning

For path planning, the training script is very similar to the one for road edge detection. That is because the two tasks are almost identical, the goal is to regress the coordinates of the points that belong to some curves, therefore the curves themselves. Here the difference is that there is only one curve, even though there is some support for multiple paths to be outputted by the model, as mentioned earlier.

The thing that makes path planning a little bit more unique than the other tasks is the fact that the input is not just an image. Since we are dealing with crossroads, and there are many directions the user might want to drive to, the decision was made to add another input vector called desire. Supposedly, the driver could use his blinkers when a crossroad is detected by the crossroad detector model (or the ComboModel, depending on which one is being used), in order to show the application in which direction they desire to go. Had this been implemented in a real world application, when no blinker is active while detecting a crossroad the desire will be assumed as "forward", otherwise it would be the direction the blinkers indicate. Ofcourse, since the whole project is based on monocular computer vision, it wouldn't be practical or safe for the path planner to fully drive the car, since there are other factors other than the road layout that play a big role in intersections, such as other cars, pedestrians, signs, etc. However, for this project the only focus was deciding the right path the car should follow with a somewhat end-to-end solution, based just on an input image and the desire of the car's direction.

Since desire can only include 3 main directions (there could be more ofcourse depending on the situation), forward, right and left, it can only have 3 different values. We need this input to make a higher impact to the network, so instead of just adding it as a single neuron of values 0, 1 or 2, desire is one-hot vector encoded before being concatenated to the input of the Bayesian linear layers of the second part of the neural network. So instead of feeding the network a single numerical value for desire, the model is fed a vector of length 3 and values either [0, 0, 1], [0, 1, 0] or [1, 0, 0].

The annotations for the actual path are extracted the exact same way the road edge curves are extracted from the corresponding .xml files. Desires are extracted similarly to crossroad labels (only here the values are not binary) from their log files. The same goes for the individual frames from video data.

The loss function once again is Negative Log Likelihood, although an attempt was made by using Mean Squared Error Loss, but that choice proved to be sub-optimal. The path planner neural network was trained on a GPU for 100 epochs in batches of size 64, using a random sample for each batch to ensure variety of images with or without crossroads.

At the end of the training process, the model is once again saved either for retraining/finetuning or deployment in an application.

Note that the number of epochs mentioned here for each task might have been changed during optimization of the models, and is not to be taken for granted as it is.

6.4 Training the ComboModel

Training the multi-task model is not as complex as one might think. Basically, it is a combination of the previous training scripts. At first, all the files (videos, crossroad logs, curve .xml files, etc) are selected and their data is loaded. The get_data() function returns the individual frames of each videos and their corresponding crossroad labels, road edge and path curves as well as the desires, which are used only for path planning. Then, the data of all videos is concatenated into big arrays while maintaining it's integrity and usability.

Training is once again done using a GPU, by adding .to(device) after the model and input/output tensors definitions (where device is "cuda" if an Nvidia graphics card is available). The model is trained for 50 epochs (this number might change due to optimization) in batches of 16 (since the model in it of itself is memory demanding and only 6GB of VRAM were available) using the Adam optimizer with a learning rate of 0.001. The input tensor X which represents the image is forwarded to the large neural network while the labels for each task are converted into tensors (Y1, Y2 and Y3 for crossroad label, road edges and path curves respectively). Desire is also converted as a tensor and is forwarded to the model, but it is handled only by the path planner head of the Combo Model.

The loss function used is a custom one called ComboLoss which is defined alongside the model in model.py. Details for the inner workings of the loss function are mentioned in previous chapters (note that it combines BCELoss with negative log likelihood loss by summing them up). After loss is calculated, it is propagated backwards into the neural network and optimizer.step() is called in order for the model to learn it's parameters based on the error of its output compared to the individual given labels. The ComboLoss takes as parameters the output of the model and the three training labels for each task (loss = loss_function(out, Y1, Y2, Y3)). Eventually, stats for monitoring the training process are calculated and outputted via plots and the model itself is saved for further use. The main function of train_combo.py can be seen bellow.

```
if __name__ == '__main__':
  device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
  print(device)

  base_dir = "data/videos/usable/"
  model_path = "models/combo_model.pth"

  video_files = []
  log_files = []
  annot_files = []
  path_files = []
  desire_files = []

  for f in listdir(base_dir):
    if f.endswith(".mp4"):
      video_files.append(f)
    elif f.endswith("annotations.xml"):
      annot_files.append(f)
    elif f.endswith("path.xml"):
      path_files.append(f)
    elif f.endswith("desire.txt"):
      desire_files.append(f)
    elif f.endswith(".txt"):
      log_files.append(f)
  video_files, log_files, annot_files, path_files, desire_files = sorted(video_files), sorted(log_fi

  video_files = video_files[:3] # TODO: this is a temp hack, need to get all videos' annotations
  log_files = log_files[:len(video_files)]

  print("Data Files:")
  print(video_files)
  print(log_files)
  print(annot_files)
  print(path_files)
  print(desire_files)
  assert len(video_files) == len(log_files)
  assert len(video_files) == len(annot_files)
  assert len(video_files) == len(path_files)
  assert len(video_files) == len(desire_files)
  model = ComboModel(num_layers=34).to(device).train()  # CHANGE THIS

  #for i in (t := trange(len(video_files))):
```

```
  print("Data Files:")
  print(video_files)
  print(log_files)
  print(annot_files)
  print(path_files)
  print(desire_files)
  assert len(video_files) == len(log_files)
  assert len(video_files) == len(annot_files)
  assert len(video_files) == len(path_files)
  assert len(video_files) == len(desire_files)
  model = ComboModel(num_layers=34).to(device).train()  # CHANGE THIS

  #for i in (t := trange(len(video_files))):
  for i in (t := trange(len(video_files))):
    #t.set_description("Loading from files: %s, %s, %s" % ((base_dir+video_files[i], base_dir+log_fil
    print("Loading from files: %s, %s, %s, %s, %s" % ((base_dir+video_files[i], base_dir+log_files[i]
    frames, labels, annotations, path, desires = get_data(base_dir+video_files[i], base_dir+log_files
    frames = conv_frames(frames)
    if i == 0:
      all_frames = frames
      all_labels = labels
      all_annotations = annotations
      all_paths = path
      all_desires = desires
    else:
      all_frames = np.concatenate((all_frames, frames), axis=0)
      all_labels = np.concatenate((all_labels, labels), axis=0)
      all_annotations = np.concatenate((all_annotations, annotations), axis=0)
      all_paths= np.concatenate((all_paths, path), axis=0)
      all_desires= np.concatenate((all_desires, desires), axis=0)

  frames, labels = [], []  # free up some memory
  model = train(all_frames, all_labels, all_annotations, all_paths, all_desires, model)

  save_model(model_path, model)
```

CHAPTER 7 Deployment Application, Results and Conclusions

For the purpose of testing out the results and performance of the different deep learning models that were trained in the scripts mentioned previously, a simple deployment application was written in app.py. What this application does is it loads either the individual models for each task, or the ComboModel and displays the various predictions on the frames it is given, while also giving information about the training labels, annotations, etc (if the video used by the app belongs in the training dataset).

Note that the models run in the GPU even at deployment in order to boost the overall performance and frames per second of the display algorithm. As for path planning, desire is always set to forward (0), since all of the data consists of videos driving in a straight line, however controls can be easily implemented using some OpenCV functions, same as in the scripts for labeling crossroads and desires.

First, the user specifies one of two modes for the network via environment variables. The variable MODE can either be combo or single-net, which dictates whether the large ComboModel or individual networks for each task will be used by the application. After that, the code checks if there are existing annotation files, so that it can display them alongside the network's predictions. If they exist, they are loaded the same way as in the get_data() functions for each task.

If the multi-task option is off, each model (cr_detector, re_detector and path_planner) is loaded into the GPU by first defining the model class from model.py and then loading the state dictionary from the corresponding .pth file (which is the format the networks are saved when training is done). Then, the models are set to evaluation mode by calling model.eval(), which is used for optimization and informs the algorithms that no parameters will be learned. If the multi-task parameter is set, however, only the multi-task model is loaded, the same way mentioned above.

When done with data and model preparations, the application code runs a loop that reads each frame from the video, preprocesses it the same way it was done for training and forwards it to the model(s) (note that since the networks use batch normalization layers, they need to be fed more than one frame at a time, here they are fed 2 just to avoid errors and only the second frame is displayed). If the ComboModel is used then the output of the model is handled like an array of predictions, one for each task, otherwise each model's output is stored to the corresponding variable. In order to display the curves for either road edges or the path, the polylines' points are deserialized (the opposite way that was used for serialization into a vector during training) back to their multidimensional matrices and are upscaled from their network's resolution (320x160) to the one of the display.

After that, all that remains is to show the actual display for the user to see the annotations along with the deep learning predictions. OpenCV provides some very useful functions for image processing, making the process of drawing on each frame simple. The frame that was used as input for the networks is upscaled to the resolution of the display and then the various curves are drawn using the custom coded function called draw_polylines() that iterate the curves' points'

coordinates and call OpenCV's function cv2.polylines(). That is done for both annotations and predictions for road edge detection and path planning. The predicted road edges and shown in orange, the predicted path in green and their corresponding annotations in red and blue respectively. On the top left corner, the predicted label for crossroad detection (crossroad, no-crossroad) is displayed along with the current value for desire. On the top right corner of the displayed image, the frames per second (FPS) are displayed (which are calculated by using python's time() function) in green color. At the bottom of the loop's code, the processed frame is displayed. OpenCV then proceeds to process the next frame in the video file using cv2.waitKey(1), or simply breaks the loop if the key "q" is pressed by the user. When the loop is over, the video capture is released and all OpenCV windows are destroyed (cap.release(); cv2.destroyAllWindows()). Some more information about the networks' predictions' values are printed on the terminal, in case they are needed for debugging.

```
cuda:0
Running in single-net mode
Loaded model from models/resnet_cr_detector_local.pth
Loaded model from models/re_detector_bayesian_local.pth
Loaded model from models/path_planner_desire.pth
[+] Using ground truth for road_edges
[+] Using ground truth for path
Frame: 1
[+] CR_Detection Ground Truth: 0 -> no crossroad
[~] Predicted value for cr_detection: 0.00021374197967816144
[~] Predicted value for re_detection
tensor([  11.1146,  117.0024,   33.1396,  117.7479,   53.6196,  115.1660,
          80.5056,  111.0606,  104.4806,  114.2169,  105.0168,  115.0638,
         111.5788,  114.1790,  130.3926,  109.0952,  194.7713,  117.2105,
         226.8055,  124.5453,  268.3209,  133.4470,  342.5810,  146.2717,
        -100.7529, -102.3203,  -99.0895, -101.5939,  -98.6249, -101.7294,
        -105.1368, -102.5884,  -96.4369,  -98.1900,  -96.8923,  -98.5106,
         -97.2051,  -98.7512,  -96.8178,  -98.4721, -100.7427, -100.7175,
        -100.3957, -100.5911, -100.6552, -100.6261, -100.8943, -100.8183],
       device='cuda:0', grad_fn=<SelectBackward>)
[~] Predicted value for path_planning
tensor([159.3456, 159.2567, 159.3091, 145.4720, 159.5177, 131.8574, 159.8354,
        123.5763, 160.2676, 118.4284, 160.4881, 113.8473, 160.6651, 109.9594,
        161.1745, 106.6898], device='cuda:0', grad_fn=<SelectBackward>)
FPS: 0
```

The performance and results of the various deep learning tasks were quite mixed. Crossroad detection in it of itself worked almost perfectly. The model learned the whole training dataset accurately with just a few epochs, achieving

minimal loss and without overfitting. It performed well enough on roads it had never seen before, however not that well since the data used for it's training was not all that good. Crossroad detection's single neural network was the only model that was trained on more than 3 videos, however it was not enough to make it truly robust and highly accurate for real-world deployment.

As for the path planning task, the single net model was able to regress the given curves very well, even though the loss function's value was seemingly high. This could be due to the simplicity of the data, since the cars in the videos were driving in a straight line, so the path curve was not all that complex. Note that the path planning data was annotated by hand, meaning that the path curves were manually drawn on the frames of the video file, which is suboptimal and can be inaccurate, especially when the model is deployed in the real world. A great solution would be to create an auto-labeling stack, which takes in data from videos as well as the data of GPS, IMU (like a gyroscope to determine the vehicle's pose), RADAR and visual odometry sensors (along with others as well) that most modern cars have with the same timestamps, combines them using Kalman filters, performs accurate localization and creates a path in the 3D world. Then, it can always be projected back to the 2D frame by using various algorithms. This might seem complex, but the training data for the path planner would be highly accurate (will match the human driver's behavior almost perfectly), allowing the model to truly learn to drive like a human, while also greatly improving the scalability of the data stack. If the path planning data does not require manual labeling, more hours of driving videos can be fed to the networks without hesitation, which is great for deep learning models since they require tons of data with variety in order to perform for real-world deployment. However, any other type of data besides video frames was not available and for the sake of simplicity, manual labeling was chosen. This resulted in less data being used, which is the reason why the models did not perform as good as they possibly could.

Road edge detection in it of itself did not perform that good. It proved that this task requires a powerful model to achieve accuracy, because it needs to regress a variable number of complex curves. Given the complexity of the data and the limitations of the hardware, the results were not optimal. The loss value was quite high, however when the model was tested in the custom application on the training data itself, the predictions were not far from the actual annotations. Although, the neural network needs to be highly accurate with almost zero loss on it's training data in order for it to stand a chance on real world deployment. Even so, the model performed decently considering the limitations and scale of the project, and given some optimization it could prove to be a great asset for real-world self-driving.

Finally, the multi-task model called ComboModel performed quite well on some tasks while not that good on others. Crossroad detection was good enough, considering the fact that in the multi-task model this task was trained in less data (more data was labeled for crossroad detection but not for road edge detection and path planning). The model did well on the training data for this specific task, but for the other two the performance was questionable. Path planning was quite decent due to it's simplicity (note that currently the number of training epochs is fairly low, so the reason for this problem might be the fact that the whole ComboModel was undertrained). However, road edge detection did not match the performance of it's corresponding single-task network, which again might be due

to the number of training epochs. Another problem might be the weight of the task's loss function in ComboLoss. Currently, all tasks' loss-weights are equal to 1, as if they were not weighted at all, thus weighting the loss function for road edge detection (which was negative log likelihood) more than the other ones should force the network to learn it's parameters by focusing more on it's most demanding task. This whole training phase though requires a lot of experimentation due to it's scale, in order for it to be viable for real world deployment. Even so, the concept of this project can benefit a lot of self-driving car agents (i.e. deep learning models), depending on the relationships of the various tasks that can be trained together.

In conclusion, the whole idea was to help some basic tasks of end-to-end models (such as Comma.ai's Openpilot) focus on other more specific features by training them alongside other more specific tasks. Tesla's Autopilot takes this concept to the edge by crafting huge multi-task models in order to create an accurate 3D map of the environment, something that beats the scale of this project by a lot. One last thing one should take into consideration is the fact that the whole deployment application runs at about 30 FPS, which is viable for real-world driving especially if the app itself was to be implemented in a more practical language such as C++. Thus, the performance of the deep neural networks used here is very satisfying in terms on speed and responsiveness.

BIBLIOGRAPHY

- End-to-End Motion Planning With Deep Learning [https://mankaran32.medium.com/end-to-end-motion-planning-with-deep-learning-comma-ais-approach-5886268515d3]

- End to End Learning for Self-Driving Cars [https://arxiv.org/abs/1604.07316]

- Which Tasks Should Be Learned Together in Multi-Task Learning? [https://arxiv.org/abs/1905.07553]

- Andrej Karpathy: Tesla Autopilot and Multi-Task Learning for Perception and Prediction [https://www.youtube.com/watch?v=IHH47nZ7FZU&t=367s&ab_channel=LexClips]

- Tesla Autonomy Day [https://www.youtube.com/watch?v=Ucp0TTmvqOE&ab_channel=Tesla]

- PyTorch at Tesla – Andrej Karpathy, Tesla [https://www.youtube.com/watch?v=oBklltKXtDE&t=496s&ab_channel=PyTorch]

- Comma.AI's blog [https://blog.comma.ai/]

-modeld [https://github.com/littlemountainman/modeld/tree/3340d7f86c1e5258e890af02989eb72ae39a208f]

- Project Code Link [https://github.com/pAplakidis/OpenCRD]