Car Automation using Raspberry Pi

A THESIS PRESENTED BY

TRAN LE VINH BAO

UNDER THE SUPERVISION OF

PROFESSOR DOCTOR ANDREAS PECH
PROFESSOR DOCTOR PETER NAUTH

TO

DEPARTMENT 2

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

MASTER OF ENGINEERING

IN THE SUBJECT OF

INFORMATION TECHNOLOGY

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

FRANKFURT AM MAIN, HESSEN
JULY 2018

I hereby warrant that I am the sole author of the thesis/dissertation; that the thesis does not violate any right of privacy; that the thesis/dissertation does not contain any matter which is libelous or obscene; that the thesis does not infringe upon any statutory or common law copyright; and that I am legally empowered to enter into this agreement. All the external source of information will be given in the Bibliography.

Frankfurt am Main, Hessen
July 2018
Signature

Car Automation using Raspberry Pi

Abstract

Artificial Neural Network (ANN) is one of the most popular approaches for machine learning. Among its branches, Convolutional Neural Network (CNN) is a modern ANN class developed especially for processing images. When processing images, ANN alone cannot capture spatial information of the images as they must be flattened into an array before entering ANN. CNN on the other hand, can capture these spatial information themselves using a set of pre-trained kernels.

The car uses a Raspberry Pi 3 to control its motors using the Pi's GPIO pins and send live video frames from a Raspberry Pi Camera module to a computer. This computer then apply pre-trained CNN models to process these frames and based on these processing results send commands, which instruct the Raspberry Pi on how should the car operate.

This master thesis contains the following:

- A brief introduction of ANN and the Pi.
- My knowledge concerning CNN.
- Report of obtained results and possible improvements in the future.

Contents

1 Introduction 1

1.1 The Raspberry Pi 2

1.2 ANN and CNN 3

1.3 The car's system 4

2 Convolutional Neural Network 5

2.1 Overview 5

2.2 Loss Function 6

2.3 Optimization Method 7

2.4 Hyperparameters 10

2.5 Regularization 10

2.6 Layers 14

2.7 Common Architecture Pattern 20

3 Steering Model 22

3.1 Preprocessing 23

3.2 Training 25

3.3 Application 28
3.4 Limitations 29

4 Sign Recognition Model 34
4.1 Choosing method 34
4.2 Architecture 36
4.3 Training 37
4.4 Application 38
4.5 Limitations 39

5 Graphical User Interface 42

6 Conclusion 45

References 48

Listing of figures

1.1.1 RaspPi on Palm. 2

2.1.1 Convolution in CNN. 6

2.3.1 Inappropriate learning rate. 8

2.3.2 Nesterov visualization. 9

2.5.1 Dropout layer. 12

2.5.2 Basic image augmentation methods 13

2.6.1 Max Pooling. 19

2.6.2 FC layer 20

2.7.1 The architecture of VGG16. 21

3.1.1 Frame Preprocessing. 23

3.1.2 White Patches. 24

3.1.3 Black Noises. 25

3.2.1 Samples of data from each class. 26

3.2.2 Noise Addition. 27

3.2.3 Performance of first CNN. 28

3.2.4 Training Loss and Accuracy. 31
3.3.1 SteeringOperation. 31
3.3.2 SteeringOperation. 32
3.4.1 SteeringOperationFailure. 33

4.1.1 Grid of boxes. 35
4.3.1 Generating cropped sections. 37
4.3.2 Training Loss and Accuracy. 39
4.4.1 SignRecognition. 40
4.5.1 SteeringOperationFailure. 41

5.0.1 Main menu. 42
5.0.2 Standby menu. 43
5.0.3 Noise Addition. 44

List of Tables

3.2.1 Steering model hyperparameters 27
3.2.2 Steering model structure 29
3.2.3 Steering model hyperparameters 30
3.2.4 Steering model training result. 30

4.2.1 Sign model structure 36
4.3.1 Sign recognition model hyperparameters 38
4.3.2 Steering model training result. 38

This thesis is dedicated first and foremost to my parents, who have always tirelessly provided me with emotional and financial support.

To my best friends, Huu Tan Vu and Huynh Dang Khoa, who have inspired me with new ideas along the course of this journey.

And last but not least, to my mentors, seniors and friends who have shared words of advice and encouragement to finish this thesis.

Acknowledgments

I would first like to thank my advisors, Prof. Dr. Andreas Pech and Prof. Dr. Peter Nauth. They consistently allowed this work to be my own, but steered me in the right direction when the need arised.

I would also like to thank my Senior, Mr. Ngoc An Ha and my friends Huu Tan Vu, Dang Khoa Huynh and Tra Trong San Le. It was fantastic to be able to work and exchange ideas together with you.

And last but least, I would like to thank my family members for their constant support throughout my life until now.

Success is steady progress toward one's personal goals.

Jim Rohn

1 Introduction

Nowadays, the presence of automation has become increasingly widespread in every field, from home automation which automates household objects such as windows, lights, heaters, to automation in industrial uses, such as automated assembly lines with robot arms, and even reach as far as autonomous rovers for space exploration. It continues to seep into almost every aspect of human lives and this trend is only ever growing due to the rapid increase in computer processing power, the availability of digitalized data, the ability to share data through the Internet, and the boom of machine learning in the last two decades. Currently automation is a very popular topic for both researchers and engineers, thus making it a very promising field for workers.

Uber's fleet of self-driving car is one of the iconic examples of this field. Though the fleet is still on test runs, there are high expectations for its future. Inspired

by this project, I decided to work on building a simple automated electric car for my master thesis. This car uses deep learning algorithms to process live frames from a camera on board and the result helps it steer between two white lanes and recognize some traffic signs simultaneously.

1.1 The Raspberry Pi

The Raspberry Pi is a series of palm-sized single-board computers, and Raspberry Pi 3 is the latest version at the time this thesis is written. Though it is only as large as the palm of a human, the Pi boasts a 1.2 GHz quad-core CPU, a 400 MHz GPU, 1 GB RAM, 802.11n Wireless LAN, and many other features. It can also run an operating system and interact with mouses, keyboards and monitors through USB and HDMI ports.

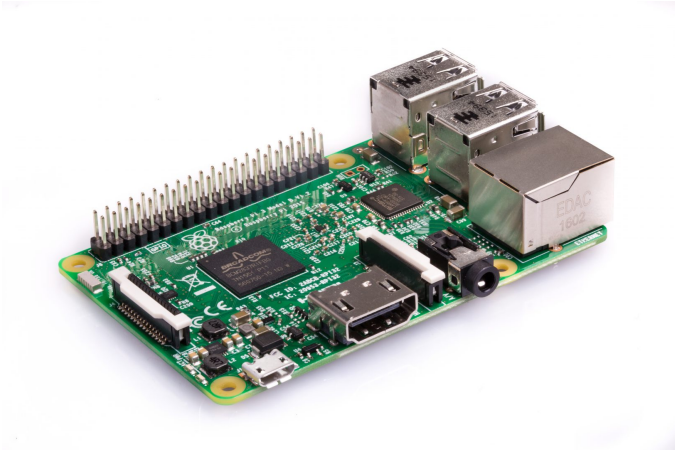


Figure 1.1.1: A Raspberry Pi 3. This small device packs the power to run its own operating system, connect to a WiFi, interact with keyboards, mouses, monitors, and many other abilities.

1.2 ANN and CNN

The concept of neural networks was inspired by human's nervous system, which is made up of layers of biological nerve cells (also called neurons). The basic task of a neuron is to let an input signal get through if it is activated, and pass that signal to its connected neurons. Arificialtificial Neural Networks (ANNs) were the very first effort of trying to mimic this natural property in around as early as 1940s. However, the computation power and algorithms at that time were not enough to develop and implement these models. Until 1970s, an optimization method called "backpropagation" was invented, which enabled the ability to train neural networks, and opened a new era for deep learning.

Image processing was one of the reasons why deep learning was introduced. Before neural networks became popular, scientists and engineers used hand-crafted feature extraction algorithms, such as thresholding, edge and corner detector to extract most useful features of an image, and those features were used to train a traditional machine learning model to classify an object of interest in that image. These techniques require a lot of time and efforts to preprocess, extract and train the model, but the result was not as good as expected.

On the other hand, artificial neural networks can outperformed the traditional machine learning approaches by providing enough examples with their corresponding labels. ANNs can learn the internal features of an object of interests automatically from those examples and predict the unobserved images with high accuracy. However, ANNs have a number of drawbacks that make them less ideal for object classification. ANN is basically a fully connected network of neurons (see figure), thus, the fact that ANN can only receive a list of values as its input forces images to be flattened before entering the network. Therefore, ANNs assume the input nodes (image pixels in image processing context) to be independent from each other. As a result, the arrangement of the pixels in the image, or the spatial information is lost.

Convolutional neural networks (CNNs) were introduced to handle these prob-

lems. Instead of flattening images, CNNs can receive the full image array as their inputs, and use the convolution operators to extract the features from the image. (See figure). CNN has proved its strength in this context by outperforming other learning algorithms, and become popular for image classification tasks nowadays. Because of its proved strengths and robustness (some references), CNNs were chosen as the main approach for my model in this thesis, and will be discussed in details in chapter 2.

1.3 The car's system

Using a Raspberry Pi 3 as the controller, a small electric car and a Pi Camera module, I built an automated car that used live image frames from the Pi Camera module to simultaneously steer between two white lanes and recognize some traffic signs. The frames, which have a size of 320 x 240 pixels, are streamed to a computer through wireless Internet connection. There they are processed by two networks, one for steering and one for sign recognition. The result will then get sent back to the Pi and it will control the car according to these results. There are two motors, one motor for acceleration and a servo motor for steering, both of which are connected to the GPIO interface of the Pi.

This system operates at an average speed of 14 frames per second (fps). The accuracy of the CNN model for steering is almost 100% while the model for sign recognition goes up to around 95% . Both models are based on VGG network architectures but are much smaller, as the frames need to be processed in a real-time manner. They keep the accuracy in a reasonable range while being extremely light compared to the VGG architectures so they works very well in this project where computing resource is constrained.

2

Convolutional Neural Network

2.1 Overview

Convolutional neural network (CNN) is a class of deep, feed-forward ANN, created for the purpose of processing images. Instead of using single weights like ANN, CNN uses kernels, which are sets of weights, to process images, thus keeping the spatial information intact. These kernels will slide through the image and results will be stored as output. This algorithm is defined in the following equation:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n K(i+m, j+n) \times I(m,n) \tag{2.1}$$

All the networks of this thesis project are built using the Keras library [7] with

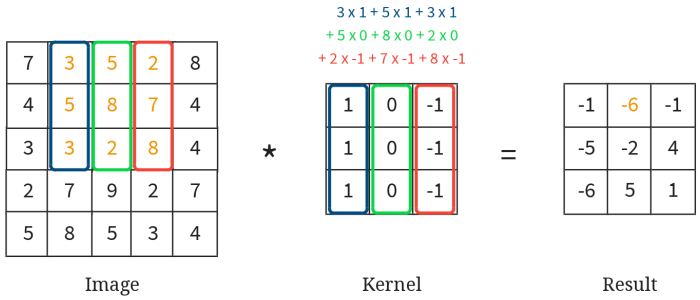


Figure 2.1.1: Convolution in CNN. The kernel will move from left to right, top to bottom until it covers every pixel of the input image. The result matrix on the right side will be saved as output of that kernel.

Tensorflow [16] (GPU version) as backend.

2.2 Loss Function

In order to train the kernels in a model, one must first determine how well that model currently performs. This is the role of loss function: It evaluates a model. The better a model is, the lower its loss would be, and vise versa, a model with high loss means that it is a poor model. For each training cycle, a model would shift the weights of its kernels a little, then calculate loss, compare that to the loss of previous cycle and use the result of this comparison as the base for how it should shift the weights in next cycle.

Loss function generally evaluates how far a model output is from its ground truth and turn that into a numerical value. Two of the most popular loss functions for classification include:

- Hinge Loss:

$$L_i = max(0, 1 + \max_{j \neq i} y_j - y_i) \tag{2.2}$$

where y_i is prediction of a correct class, $\max_{j \neq i} y_j$ is prediction of the incorrect

class with the highest score.

- Cross Entropy Loss:

$$L_i = -y_i ln(f(x_i)) - (1 - y_i) ln(1 - f(x_i)) \tag{2.3}$$

where ground truth label y_i is already one-hot encoded.

Note that the equations for classification assumed that the ground truth labels y_i have been encoded to 0 and 1 values using One Hot Encoding method.

After many experiments, my conclusion is that Cross Entropy Loss always achieve better results than Hinge Loss in both of my models. For this reason, it is the only loss function that I will use throughout the project.

2.3 Optimization Method

For optimization, the most popular method in modern deep learning is Stochastic Gradient Descent (SGD). It is evolved from Gradient Descent (GD), which is defined by the following equation:

$$W = W - a\nabla_W f(W) \tag{2.4}$$

where a stands for learning rate and $\nabla_W f(W)$ is gradient of loss function f(W).

Instead of calculating on whole dataset like GD, SGD separates a training dataset into batches, compute gradient of loss function on them and use the result to update model weights, one batch at a time. Since it only compute one part of the dataset each cycle, the result will be more noisy. However, it is much faster because the model will be able to descend many times per epoch instead of only once like GD. That means the smaller its batch size is, the faster the model can be trained. However, a batch size that is too small will make the steps too noisy, which leads to a drop in accuracy, so one must consider tuning this hyperparameter.

Like batch size, learning rate is also one of the most important hyperparameters that one must tune during the training process. If it is too high, the model will miss a local minimum and if it is too low, training time will be extended which leads a waste of resources. In the worst case, a learning rate that is too small makes the model easily falls into local minima, as depicted in figure 2.3.1

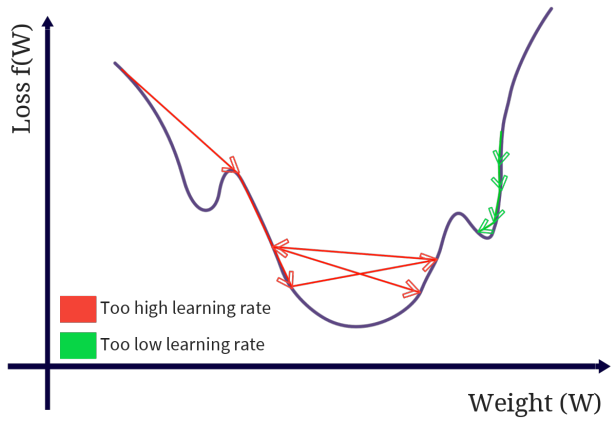


Figure 2.3.1: Inappropriate learning rate. The red arrows depict the case when learning rate is too high, and the green arrows depict when learning rate is too low. As can be seen, the red arrows hover above and will never reach the minimum value while the green arrows get trapped by a local minimum.

SGD can be further accelerated by adding momentum V to weight updates with SGD:

$$W = W + V \tag{2.5}$$

where

$$V = \gamma V - \alpha \nabla_W f(W) \tag{2.6}$$

Note that the V on the right side of equation 2.6 is the value of V from previous cycle.

This kind of simple momentum works well when a model is in a plateau far from all minima. However, if a model keeps building momentum when it approaches

a minimum point, it may go over that point if there is too much momentum and start hovering above it like the red arrows in Figure 2.3.1. For this reason, we need a smarter algorithm for adding momentum.

One of the most popular solution for this problem is Nesterov Accelerated Gradient [9]. Unlike standard momentum where the model simply make a large jump in weight update following the calculated gradient, Nesterov takes two additional steps: Calculate the gradient at the arrival point of the large jump, then use that gradient for correction. Figure 2.3.2 visualizes how Nesterov is executed and how it looks compared to standard momentum.

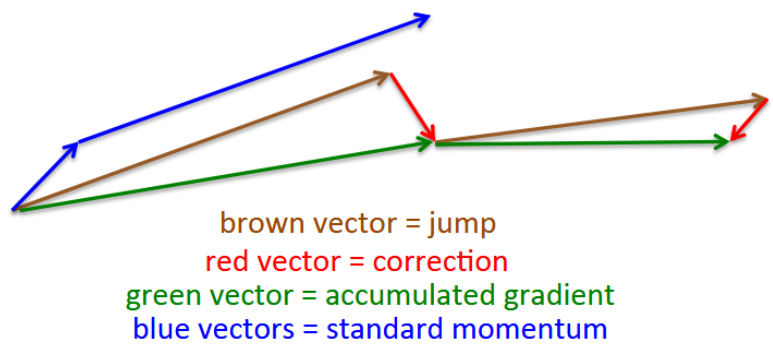


Figure 2.3.2: Nesterov Visualization from G.Hinton's lecture 6c The blue arrows depicts a standard momentum algorithm with only 1 step, while Nesterov method contains 3 steps shown in brown, red and green arrow.

However, Nesterov momentum slows down the training process significantly so it should be used with care, especially if a programmer is working with large datasets like ImageNet for example. From my experience, applying Nesterov slows down training time by two to three times. On large networks that require days to train, Nesterov may prove to be undesirable because it costs too much resources and reduces the number of tunings that can be done on those networks since time is always the most limited resource. In my case Nesterov is the optimum choice, because my network is small and the dataset is not large so an increase in training time does not affects me heavily, plus adding Nesterov significantly improve the stability when training compared to using simple momentum.

2.4 Hyperparameters

Hyperparameters are parameters that do not change during the training process. They have an important effect on training output and the majority of time spent on tuning a network is spent on calibrating the values of these hyperparameters. There are several important hyperparameters that should be tuned:

- Learning rate: Determines how fast a network learns. High learning rate helps networks learn faster, but it may hinder them from reaching a minimum point. One can either use a fixed learning rate or introduce a decay parameter to reduce learning rate over time.
- Number of epochs: Also the number of times that the entire training set passes through the network. If there is no sign of overfit and no stagnation in validation accuracy growth then the larger the epoch number is, the better.
- Batch size: The number of samples that pass through the network in each iteration. There can be several iterations each epoch. The smaller batch size is, the faster the network learns, while a larger batch size makes the learning process more stable.
- Dropout probability: Determines the number of nodes to be randomly selected and dropped in each iteration. This helps prevent overfitting, but will make the training process unstable if the probability is too high.
- Momentum: Accelerates the training speed of networks. However, a large momentum may make the network shoots through a local minimum.

2.5 Regularization

Generalization is the ability to adapt to situations outside of the training dataset, which is the final goal for most deep learning models, including CNN. Regularizations are techniques devised for the sake of improving the generalization ability

of a model, in other words, decreasing the error rate when it is applied on unseen data. These techniques may even increase the error rate during training phase in order to achieve this. Without regularization techniques, deep learning networks like CNN can easily become overfit to its training data, as there is a huge number of parameters to calibrate in the network while training data is not always sufficient.

Although generalization is an essential ability for any model which makes regularization a very important technique, applying too much regularization may lead to underfitting, which is also an undesirable outcome. The most popular regularization techniques include:

- Weight Penalty
- Dropout Layers
- Early Stop
- Dataset Augmentation

In this project, I have implemented three techniques: Weight Penalty, Dropout Layers and Dataset Aufmentation.

2.5.1 Weight Penalty

The most standard form of regularization is applying weight penalty. Since it has the tendency to return weights to 0 or near 0 if they are not regularly supported, weight penalty is also called weight decay. From Equation 2.4, a weight decay term R(W) is added in order to achieve the following equation:

$$W = W - a\nabla_W f(W) + \lambda R(W) \tag{2.7}$$

where "λis the scale of weight decay and a hyper parameter that one must tune when training, together with learning rate "αand batch size. There are two types

of mainstream weight penalty R(W) that are most widely used, L1 and L2. This project implement L2, which is defined in the following equation:

$$R(W) = \sum_i \sum_i W_{i,j}^2 \tag{2.8}$$

2.5.2 Dropout Layer

The dropout layer will randomly remove some nodes from the network during training time. This makes it less likely for a nodes with very heavy weight to appear, which make the network biased. This node, with its large weights, may decide the output by itself without regard to other nodes, is an example of overfit. It most likely will not work outside of the training dataset. Dropout can distribute the importance of that node into several other nodes, which in turn increase the generalization ability.

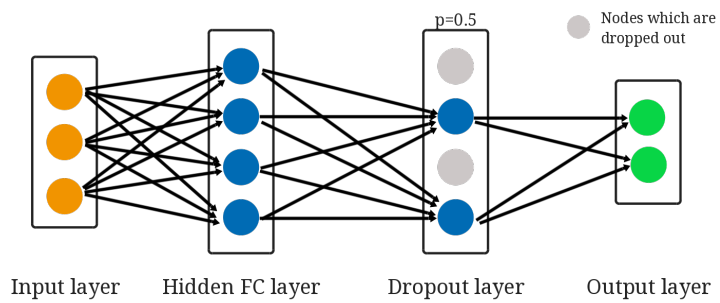


Figure 2.5.1: Dropout layer. During training time, random nodes along with their connections, are removed. The number of dropout nodes depends on the probability hyperparameter p.

Also, evolving from dropout layers, a method called Dense Sparse Dense Training Flow [3] had been devised. Although it seemed promising, implementing this would cost too much time and effort so I stopped at using normal dropout layers.

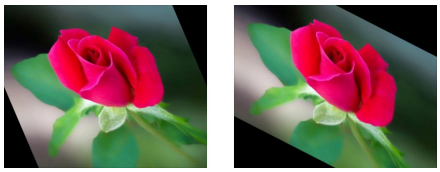
2.5.3 Dataset Augmentation

"Data augmentation encompasses a wide range of techniques used to generate new training samples from the original ones by applying random jitters and perturbations such that the classes labels are not changed." [13] It not only increases the amount of training data, but also makes a model much more robust against minute changes like viewing perspective, light condition, and salt & pepper noise (Gaussian noise) for example.

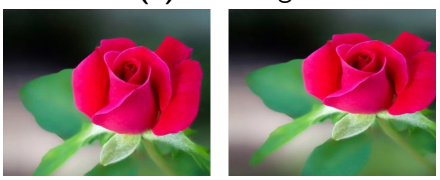
(a) Original image.

(b) Flipping.

(c) Rotating.



(d) Shearing.



(e) Scaling.

Figure 2.5.2: Basic image augmentation methods. (a) is the original image. (b) shows images that are flipped horizontally and vertically. (c) shows rotated versions. (d) shows images that are sheared horizontally and vertically. Finally, (e) shows enlarged images, one that keep its aspect ratio and one that is distorted.

Although there are more advanced techniques, they are peculiar and should only be used in certain cases. The most popular ones are still the basic techniques, which are shown by Figure 2.5.2.

The basic techniques mentioned above can easily be combined with each other in order to create even more diverse training data. In my project, I have implemented flipping and scaling to increase my training data.

2.6 Layers

This section is about the layers, or building blocks, that compose a CNN. There are many types of layers, but the ones used in this project include:

- Convolutional (CONV)
- Activation (ACT or RELU)
- Batch Normalization (BN)
- Pooling (POOL)
- Fully Connected (FC)
- Drop Out (DO)

Among these, only the CONV and FC layers contain the weights that can be trained. Others are auxiliary layers that helps the training process become faster and more accurate. Detail on each layer will be discussed in the following subsections.

2.6.1 Convolutional Layer

The CONVs are the core layers and are mandatory in any CNN structure. After all, it can be said that this layer is what differentiate CNN from ANN. Each CONV layer consists a set of K filter (or kernels) that can be learned through training.

When an image enters a CONV layer, it is treated as a 3D matrix where the depth is the number of channels. Each filter in a CONV layer will slide through its input and create a 2D output activation map, as shown in Figure 2.1.1. Then these 2D maps will be stacked together into a column, creating a 3D matrix with the depth of K and it is treated as the output of this CONV layer.

A CONV layer has several important characteristics that one must pay attention at:

- Number of filter K: This controls the depth of output produced by a layer. Therefore, it has a direct effect on network size, which is a very important characteristic. A model that is too small may fail to learn all information from a large image while training a large model on an insufficient dataset may lead to underfitting. The only way to find the correct value of K is through exhaustive trial-and-error calibration and the programmer's own experience.

- Width-Height: All filters in a CONV layer have identical width and height defined when the layer was created (i.e. 3x3, 1x1, 5x5), and their depth is equal to the depth of layer input. For example, depth of filters in the first layer is 3 if input is a RGB image and it is 1 if input is a gray scale image. And, if the first layer has K=100, then the depth of filters in the second layer would be 100. As most CNN networks use square filters and input images, later I will only use width and depth to describe their dimensions. Currently I implemented 3x3 kernel in my networks.

- Stride: This controls step length when a filter is sliding through an input matrix. 1 is the default and most used value for stride since there is no special requirements and it can be used for most cases. A stride larger than 1 will reduce the output size compared to the input and are typically used together with larger filters like 7x7 and 11x11 on high resolution input images. However, this is considered an advanced technique and should be used with care since the programmer has to carefully keep track of input and output size of

each layer. A simpler and more straightforward method, POOL layer, will be introduced in Subsection 2.6.4

- Padding Size: When either filter size or stride length is larger than 1, the width of an input matrix will be reduced after the mapping process. To prevent this, the input must be padded so its output will have the same dimensions. There are several types of padding like constant, symmetric, wrap and so on, but the most widely used is still zero padding because it is easy to implement and add no additional information to the image, therefore reducing the risk of an interference on output. When introducing padding, one must make sure that the output width W_{out} must be an integer:

$$W_{out} = \frac{W_{in} - F + 2P}{S} + 1 \tag{2.9}$$

  where W_{in} is the width of input image, F is the filter size, S is the stride length and P is the padding size.

2.6.2 Activation Layer

An ACT layers always follow immediately after a CONV layer. It is often referred to as RELU since ReLU is the most popular option for activation function. The purpose of ACT layers is to check if the output is "fired" or not, so that nodes in the network can mimic neurons in our brain, which is already proven to be a powerful system.

There are three most well-known activation function:

- Step Functions:

$$f(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.10}$$

  The most simple and intuitive activation function. However, its gradient is 0 and is not differentiable at 0, which may lead to problems if used together

with SGD.

- Sigmoid Function.

$$f(x) = \frac{1}{1 + e^{-t}} \tag{2.11}$$

  Compared to Step Function, Sigmoid Function is very smooth and differentiable everywhere, which makes implementation of SGD together with it easy. However, this function has a fatal flaw: Its gradient becomes almost non existent after it reaches the saturation point.

- Rectified Linear Unit (ReLU) Function.

$$f(x) = max(0, x) \tag{2.12}$$

  ReLU is the most popular activation function in recent years because it outperforms the other two families in most cases. However, it still has a weakness: ReLU is not differentiable at 0, although this is normally not a problem because it is very rare for outputs, which are real numbers, to reach the value of exactly 0.

2.6.3 Batch Normalization Layer

As its name suggested, BN is used to normalize the output of its previous layer before passing it to the next layer in the network. This helps stabilizing the network by reducing interval covariance shift [5], which in turn reduces the number of epochs required and decreases final losses because fluctuations throughout the network caused by the diversity in training datasets is regularized.

BN also enables higher values of learning rate. If a network without batch normalization uses a high learning rate, the diversity in training data may get amplified and cause imbalance in weight distribution inside the network, which decreases the number of useful nodes and reduce network capacity.

The normalized value of each x_i from the input matrix is calculated in each batch β

using the following equation:

$$\hat{x} = \frac{x_i - \mu_\beta}{\sqrt{\sigma_\beta^2 + \varepsilon}} \tag{2.13}$$

where

$$\mu_\beta = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{2.14}$$

$$\sigma_\beta^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\beta)^2 \tag{2.15}$$

The ε in Equation 2.13 is a very small value, normally $1e^{-7}$. It is added to avoid a division by 0.

2.6.4 Pooling Layer

There are two methods of pooling: max pooling and average pooling. Both reduce the spatial size of input data, but max pooling extracts strong features and discard the rest, which effectively destroy spatial information. Average pooling preserves more spatial information but cannot extract features as much as max pooling. In my case, the white lanes and traffic signs are fairly simple object with less spatial information, therefore I use max pooling for every POOL layer in my networks.

Each POOL layer has two parameters: receptive field width F and stride length S. There are two ways to implement the pooling methods above:

- Overlapping pooling where S<F: This method preserves more information but does not reduce that much input size, thus it is normally used in networks where input images are small(i.e. 64x64, 32x32 pixels and below).
- Non-overlapping pooling where S=F: This method heavily reduce input size, thus it discards a lot of information and is typically used when input images are large.

In the case of my project, even though the input images are small (64x64 and 20x20), I still chose to use non-overlapping pooling where S=F=2 because I need to minimize the deployment time even if some accuracy must be sacrificed.

Output size of a POOL layer can be calculated using the following equation:

$$W_{out} = \frac{W_{in} - F}{S} + 1 \tag{2.16}$$

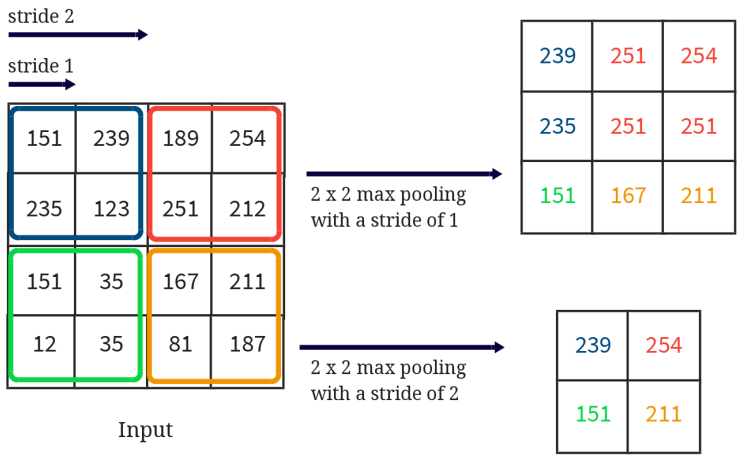


Figure 2.6.1: Visualization of max pooling. We can see how a pool size of 2 reduce the input spatial size and how it can be further reduced by using a longer stride. S=F=2 can reduce 75% of the parameters.

2.6.5 Fully Connected Layer

This layer follows the standard for feedforward neural networks: Each of its neuron is fully-connected to every node from the previous layer. FC layers are always placed at the end of a CNN network after the CONV layers.

Note that since FC layers can only accept arrays as input, the output from the last CONV layer must be flattened into an array before passing it to the first FC layer.

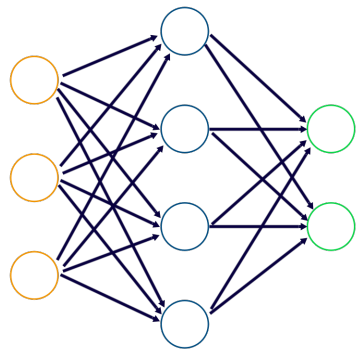


Figure 2.6.2: A simple visualization of three FC layers (lightly edited from source image on WikiPedia). The neurons in FC layers are fully-connected just like the nodes in this network.

2.6.6 Dropout Layer

DO layer is a type of regularization, which aims to prevent overfitting even at the cost of training accuracy by randomly dropping nodes from the network during training process. The only parameter in a DO layer is the drop probability p, which signifies how many percent of the nodes will be randomly dropped from the network during training process.

For more details, please refer to Subsection 2.5.2.

2.7 Common Architecture Pattern

Modern CNN networks are constructed using "building blocks", which are groups of layers that are placed repeatedly in the network.

According to [6]: "The most common CNN architectures follows the pattern: $INPUT \rightarrow [[CONV \rightarrow RELU]*N \rightarrow POOL?]*M \rightarrow [FC \rightarrow RELU]*K \rightarrow FC$ where the $*$ indicates repetition, the $?$ indicates an optional layer. More over, $N \geq 0$ (and usually $N \leq 3$), $M \geq 0$, $K \geq 0$ (and usually $K<3$)."

As an example, we shall convert the architecture of VGG [14], and VGG16 in par-

ticular into this pattern:

$INPUT \rightarrow [[CONV \rightarrow RELU] * 2 \rightarrow POOL] * 2 \rightarrow [[CONV \rightarrow RELU] * 3 \rightarrow POOL] * 3 \rightarrow [FC \rightarrow RELU] * 2 \rightarrow FC$

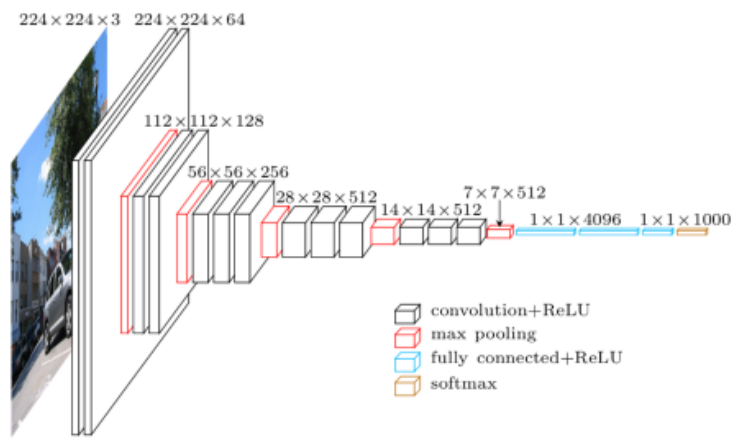


Figure 2.7.1: The architecture of VGG16 by Davi Frossard. VGG is a robust family of neural network with proven record. It achieved an astounding 6.8% error with ImageNet, a colossal and challenging dataset that consists of around 14 million images in more than 20 thousand classes. VGG16 consists of 13 CNN layers and 3 FC layers.

My networks also follow this pattern, albeit with a few addition (BN and DO layers). Steering Network:

$INPUT \rightarrow [[CONV \rightarrow RELU \rightarrow BN] * 2 \rightarrow POOL \rightarrow DO] * 2 \rightarrow FC \rightarrow RELU \rightarrow BN \rightarrow DO \rightarrow FC$

Sign Recognition Network:

$INPUT \rightarrow [CONV \rightarrow RELU \rightarrow BN] * 2 \rightarrow POOL \rightarrow DO \rightarrow CONV \rightarrow RELU \rightarrow BN \rightarrow DO \rightarrow FC \rightarrow RELU \rightarrow BN \rightarrow DO \rightarrow FC$

3

Steering Model

This section describes the model that steers the car so it stays between two white lanes. The model is run by the GPU of the computer on 320x240 input images streamed from the Raspberry Pi at a speed of 20 fps. From these frames, the model must recognize whether the car should keep moving straight forward, turn left, turn right or stop moving, which mean the model is dealing with a classification problem. Then, the result is sent back to the Pi and it will drive the car accordingly. This result may change due to which signs are recognized by the second model, and will be discussed in detail in chapter 4

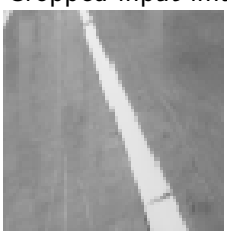
3.1 Preprocessing

To decide the steering direction of the car, only the positioning of white lanes on a frame is relevant. For this reason, I decided to add a preprocessing step in order to eliminate all unnecessary information from the frames. There are 4 steps in total, as shown in Figure 3.1.1:

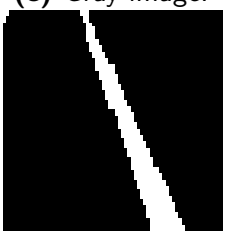
(a) Input image.

(b) Cropped input image.

(c) Gray image.



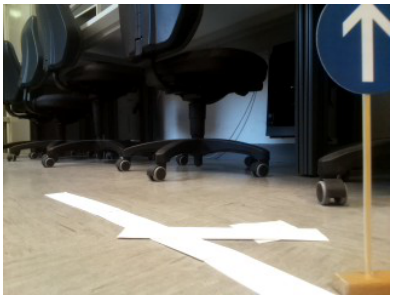
(d) Resized gray image.



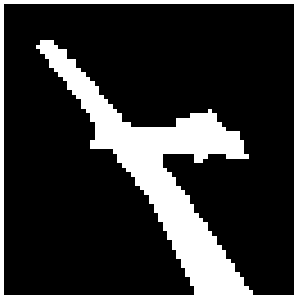
(e) Binary image.

Figure 3.1.1: Frame Preprocessing. First, the top of input frame (a) is cropped because it does not contains any information about the lanes. The cropped image (b) is then converted to gray scale which creates image (c). After that, (c) is resized to the 64x64 image (d) and finally, a filter is applied on (d) to extract the white lanes, which results in the 64x64 binary image (e). This image will be the input of the steering model.

This method of extracting white lanes from the image is simple and straightforward, thus it is very fast and easy to implement. However, its disadvantage is the susceptibility to noises caused by fluctuations in the lighting condition, as well as white object that are placed near the track. Figure 3.1.2 shows the situations where these noises are encountered.



(a) Input with noise caused by object.



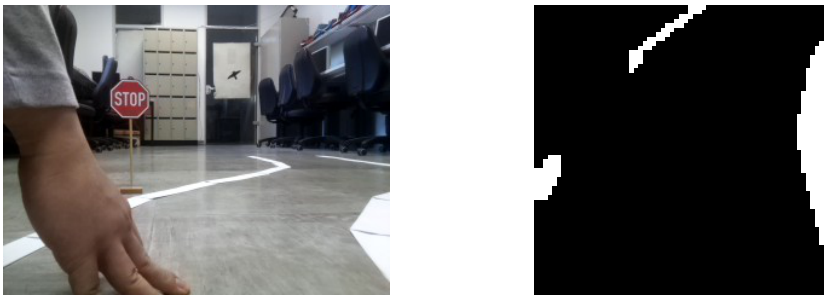
(b) Processed.

(c) Input with noise caused by lights.

(d) Processed.

Figure 3.1.2: White patches caused by white object and light sources. (a), (b) shows the case where noises are caused by white objects and (c), (d) shows the case of strong light source. The cause of noises in (d) is the automatic adjustment of the camera. In (c), it can be seen that the camera is pointing in a dark place so it increases the brightness of the image to compensate for the lack of light. As a result, the floor becomes too bright and cause the noise in (d). Compared to the floor in (a), one can see how unnaturally bright it is in (c).

However, I managed to overcome this disadvantage by using data augmentations during the training process of the model. Details will be described in section 3.2,



(a) Input with black patch noise. **(b)** Processed.

Figure 3.1.3: Black patch noises appear when the white lanes are covered by other objects. Even shadows can cause black patch noises if they are dark enough.

but basically, artificial black and white patches will be added to the training set in order to simulate the noises in real situations.

3.2 Training

3.2.1 Overview

For data collection, a track is required. For this purpose, white A4 papers were folded and connected to each other to form a pair of white lanes. Then, the car with the built-in camera was placed between the lanes at different positions to capture images of the track from various perspectives. These images were then labeled with my own decision whether the car should turn left, turn right or go straight based on its position. The following figure shows some sample data for each class:

3.2.2 Data Augmentation

For this model I used two techniques:

- Flipping: The images are flipped horizontally to double the training data.

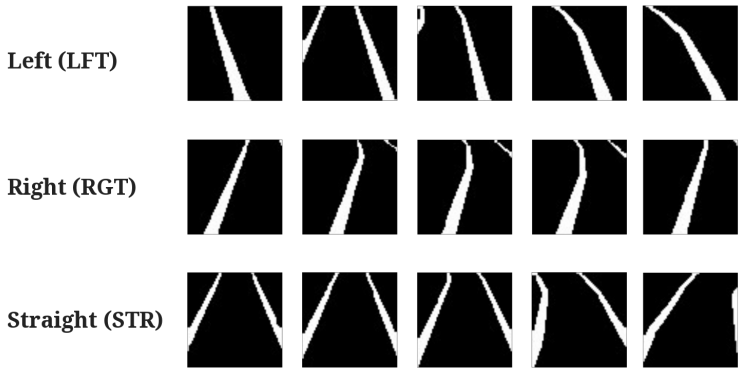


Figure 3.2.1: Samples of data from each class.

- Noise addition: Black or white noises are added to training images with random density inside squares which have random location and size.

3.2.3 Network Implementation and Tuning

Because this model has to quickly solve a classification problem using limited computing power, I decided to build a small network based on the architecture of VGG16 [14]. There are also other famous models like GoogleNet [15] and ResNet [4], however they are too complicated to use as a base for my network.

The first CNN consisted of only a CONV layer with ReLU activation and a FC layer with softmax output. It was trained on a small dataset with 3 classes and 60 images for each class. The hyperparameters of that first network are shown in Table 3.2.1 and its performance is shown in Figure 3.2.3:

Tuning hyperparameters for this network was the most time-consuming job in this entire project because at that time I was new to CNN and did not have experience on how much should the hyperparameters be increased/decreased.

(a) Addition of white noise.

(b) Addition of black noise.

Figure 3.2.2: Noise addition. White noise simulate reflection from strong light sources and white object; Black noise simulate shadows and objects lying on the lanes.

Hyperparameter Name	Value
Epoch number	100
Learning rate	0.1
Learning rate decay	0.0
Learning momentum	0.0
Batch size	32

Table 3.2.1: List of hyperparameters in the first CNN model. For a small network and dataset, 100 epoch is too many and a learning rate of 0.1 is too large.

After many experiments, I arrived at the final structure which is shown by table 3.2.2 and the parameter list in Table **??**:

3.2.4 Hyperparameters

Note that all the CONV layers use weight decay l2 with l=0.01 and same padding.

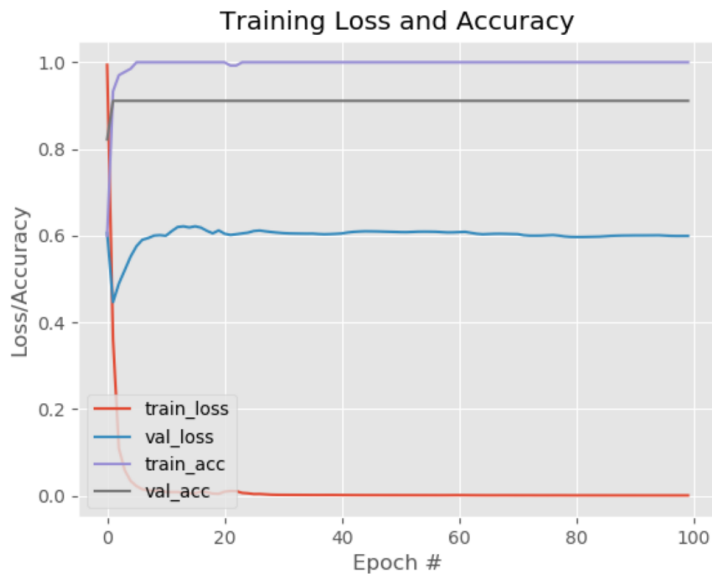


Figure 3.2.3: Performance of first CNN. As it did not have any layer to boost training accuracy and the hyperparameters are poorly set, the training process did not go well. The network was heavily overfitted.

3.2.5 Result

The training result of this model is very good, as shown in Figure 3.2.4 and Table 3.2.4:

3.3 Application

The finished model is robust and can handle unexpected situations like noises. As long as the car is not outside of the track, which is an environment that it has never been trained in, and there is not too much noise on the lanes, the model almost never fail to recognize the correct steering direction. Figure 3.3.1

Layer Type	OutputSize	Filter Size/Number
INPUT IMAGE	64x64x1	N/A
CONV1, ReLU	64x64x32	3x3/32
BN	64x64x32	N/A
CONV2, ReLU	64x64x32	3x3/32
BN	64x64x32	N/A
Max POOL	32x32x32	2x2
DO, p=0.25	32x32x32	N/A
CONV3, ReLU	32x32x64	3x3/64
BN	32x32x64	N/A
CONV4, ReLU	32x32x64	3x3/64
BN	32x32x64	N/A
Max POOL	16x16x64	2x2
DO, p=0.25	16x16x64	N/A
FC5, ReLU	640	N/A
BN	640	N/A
DO, p=0.25	640	N/A
FC6, SoftMax	3	N/A

Table 3.2.2: Full structure of steering model. The network has 4 CONV layers and 2 FC layers. Its input image is a 64x64 binary image, as explained in 3.1. Output is composed of 3 integers, which will be translated to classes by reversing the one hot coding. This is the optimal network that I found after many experiments. If it is down-sized any further, the accuracy will drop as well. Compared to its root the VGG16 network, it has additional layers which are BN and DO.

3.4 Limitations

However, this model still has limitations:

- If the car accidentally runs out of track (for example, when the image stream from the Pi to the computer is interrupted), the predictions will become random because I have not trained the car for this situation yet. The current car is programmed to stop immediately if it detected that it has gone out from the track.

Hyperparameter Name	Value
Epoch number	50
Learning rate	$8\ e^{-4}$
Learning rate decay	$1.60\ e^{-5}$
Learning momentum	0.9
Batch size	256

Table 3.2.3: List of hyperparameters in the final steering model.

Training loss	1.1713
Validation loss	1.1678
Training accuracy	0.9993
Validation accuracy	1.0000

Table 3.2.4: Steering model training result. The training loss and accuracy are worse than those of validation because Drop Out layer and Weight Decay l2 are implemented.

- When there is simply too much noise, the model will fail to recognize the correct steering direction, as shown in Figure 3.4.1

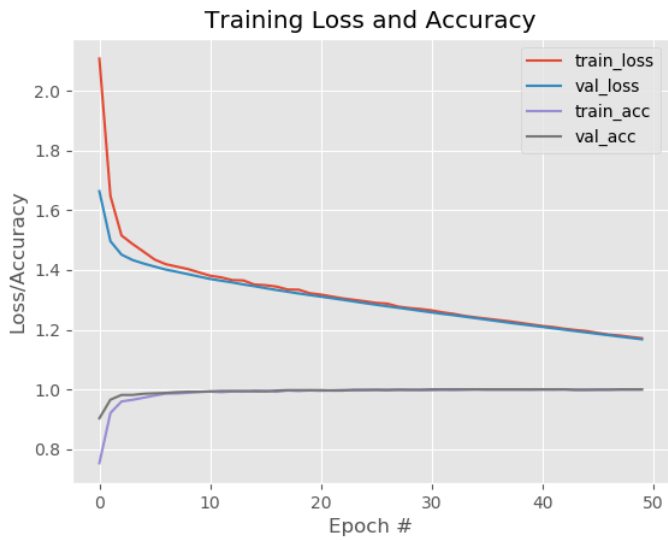


Figure 3.2.4: Training Loss and Accuracy of the steering net. Since the training data is simple and training batch is quite large, the learning curves are stable even though Drop Out layers and Weight Decay l2 are applied.

Figure 3.3.1: Steering model application. This model rarely fails to recognize the correct steering direction if the car is not outside of the track and noises are in a moderate level.

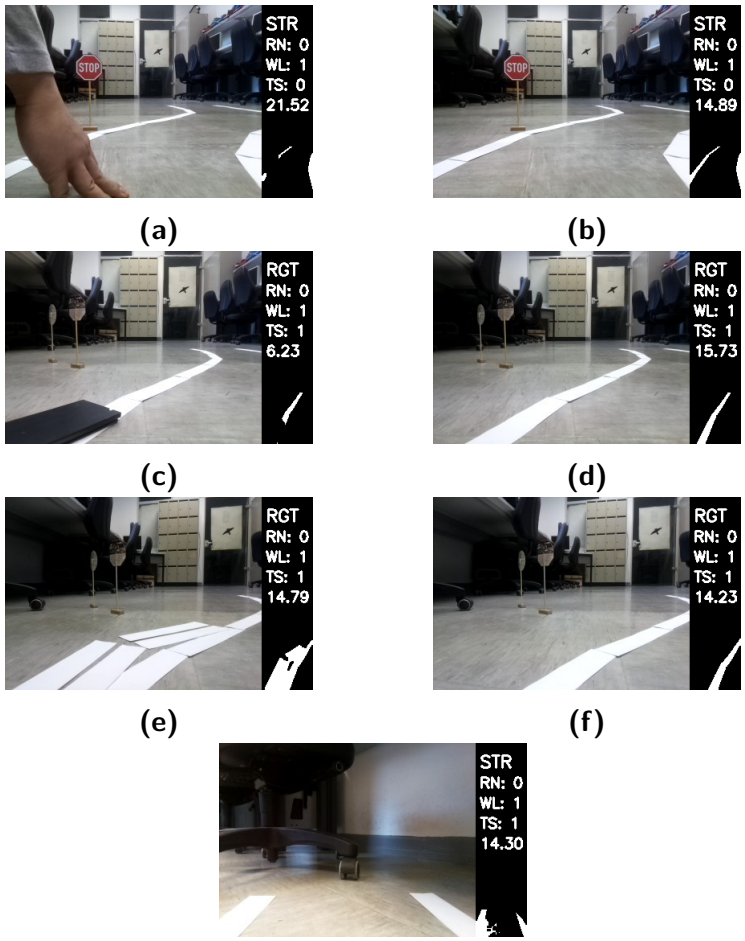
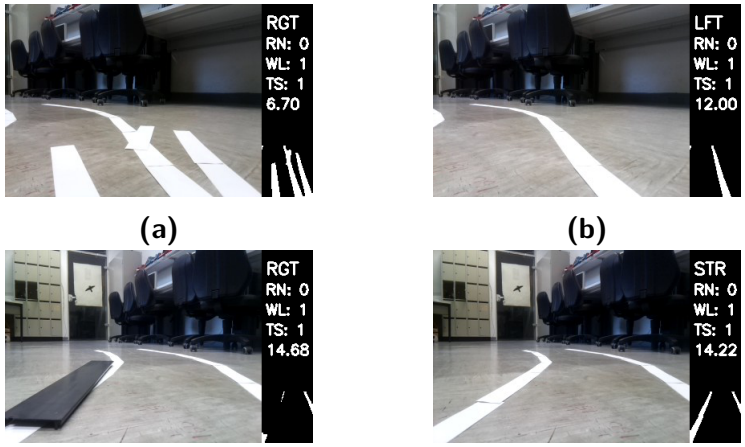


Figure 3.3.2: Steering model application in noisy condition. The model is robust enough to handle noises well. The right column contains images in normal working condition and the left column contains their noisy counterparts.



(a) **(b)** **(c)** **(d)**

Figure 3.4.1: Failure cases of the steering model. The right column depicts normal working conditions and the left column depicts their noisy counter part. If the noise becomes too much, the model would fail to recognize the correct steering direction.

4

Sign Recognition Model

4.1 Choosing method

Currently there are two mainstream techniques for object detection: region proposal based techniques [2] [1] [12] and single shot techniques [11] [10] [8]. However, I cannot use any of those due to the following reasons:

- Processing time: region proposal techniques run an exhaustive search for possible locations of objects, thus it is quite slow and not suitable for this project since I need to process an image in real time.
- Training time: single shot techniques can satisfy the processing time requirement since they look at the whole picture at once without any region proposal. However, in order to train a network using this technique, I must

create a dataset from scratch and annotate every picture by myself, and annotating a picture for single shot technique is hard: First, one must take a picture, then find the bounding boxes of each object that appears inside. The image is separated into a grid of boxes, and one must find the grid boxes which have the largest overlap regions with those bounding boxes, then annotate them with the class and the offset from those bounding boxes. And last but not least, one must check those annotation for errors to prevent them from disrupting the training process. Since I spent a considerable time training and fine tuning the steering model, there is not enough time left to annotate a dataset with hundreds of images for each class, then train tune a network from scratch.

In the end, I decided to make a hybrid between the two techniques: sliding window from region proposal and the grid of single shot technique. I separate the region of interest on my image into a grid of 16x4 boxes, each box is 20x20 pixels large. The model will run a classification on each of these 64 boxes and determines if it is just a patch of background or it contains part of a certain street sign. This method is very fast and it is easy to generate a large training set for it, however it can only recognize if a sign is in the picture and their approximate location, not their exact location so it is only an object recognition model and does not have the capability for object detection.

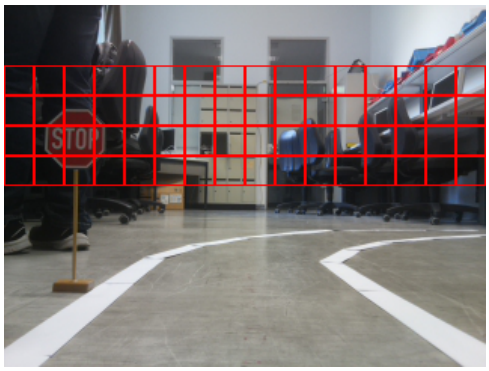


Figure 4.1.1: The grid of boxes over the region of interest.

4.2 Architecture

Again, VGG16 is chosen as the base network of this model, for the same reasons as those stated in Subsection 3.2.3. The calibration of this network was easier than the time with the Steering model because I already gained a considerable amount of experience from tuning the steering model. The whole architecture of the final network is shown in Figure 4.2.1

Layer Type	OutputSize	Filter Size/Number
INPUT IMAGE	20x20x3	N/A
CONV1, ReLU	20x20x40	3x3/40
BN	20x20x40	N/A
CONV2, ReLU	20x20x40	3x3/40
BN	20x20x40	N/A
Max POOL	10x10x40	2x2
DO, p=0.25	10x10x40	N/A
CONV3, ReLU	10x10x80	3x3/80
BN	10x10x80	N/A
Max POOL	5x5x80	2x2
DO, p=0.25	5x5x80	N/A
FC4, ReLU	400	N/A
BN	400	N/A
DO, p=0.25	400	N/A
FC5, SoftMax	1	N/A

Table 4.2.1: Full structure of sign model. There are 3 CONV layers and 2 FC layers. Input image is a 20x20 rgb image, as explained in the previous section. Output consists of 3 integers, which will be translated to classes by reversing the one hot coding.

4.3 Training

4.3.1 Strategy

Similar to the steering net described in the previous chapter, this model chosen optimization method is also SGD with Nesterov acceleration (see Section 2.3 for more detail). The training dataset contains 21412 images with 80% as training data and 20% as validation data.

4.3.2 Data Augmentation

To generate training data for this model, a window is slided through a region containing a sign, cutting out patches that contain part of that sign and save them as 20x20 images. The size and brightness of each patch is randomly altered as a form of augmentation to increase the ability to cope with unseen situation of the model.

(a) Input image. **(b)** Chosen region. **(c)** Sections.

Figure 4.3.1: Generating cropped sections. First, choose the region containing the sign (b) from (a), then slide a window through that region and save them to create sections containing parts of the stop sign (c).

The same process is repeated for other signs, backgrounds and misclassified regions to generate data. This method can produce from tens to hundreds of sections from a single image, boosting the training data size considerably.

Hyperparameter Name	Value
Epoch number	35
Learning rate	$1\ e^{-3}$
Learning rate decay	$2.86\ e^{-5}$
Learning momentum	0.9
Batch size	1024

Table 4.3.1: List of hyperparameters in the sign recognition model.

4.3.3 Hyperparameters

Note that similar to the steering net, all the CONV layers in this model also use weight decay l2 with l=0.01 and same padding.

4.3.4 Result

The training result of this model is excellent, as shown in Figure 4.3.2 and Table 3.2.4:

Training loss	0.8355
Validation loss	0.8301
Training accuracy	0.9922
Validation accuracy	0.9931

Table 4.3.2: Steering model training result. The training loss and accuracy are worse than those of validation because Drop Out layer and Weight Decay l2 are implemented.

4.4 Application

The model works well enough in normal conditions where it runs on the floor and the signs are placed to face its track. However, it is sensitive to changes in light

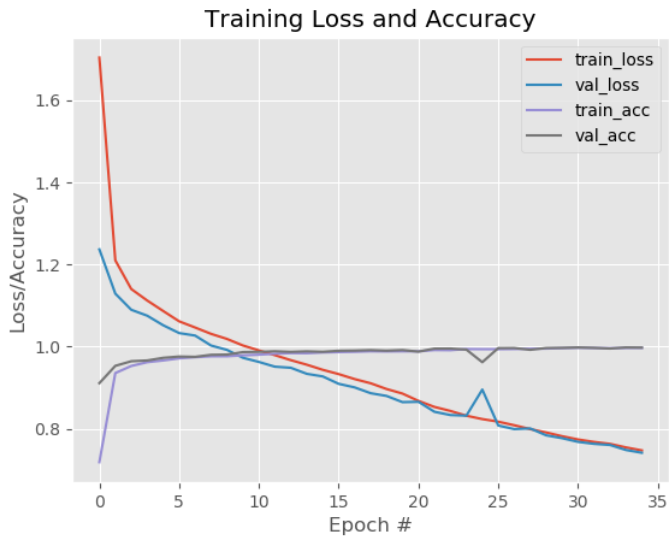


Figure 4.3.2: Training Loss and Accuracy of the sign recognition net. There are some fluctuations at epoch 17-19, which is likely caused by the regularization techniques like Weight Decay l2 and Drop Out layer. It eventually stabilize as the training data is simple and the training batch size is very large.

condition so its generalization capacity is not good.

4.5 Limitations

The limitations of this model include:

- It does not generalize well because it works poorly when the lighting condition changes too much.
- It cannot recognize very well small objects and overlapped object.

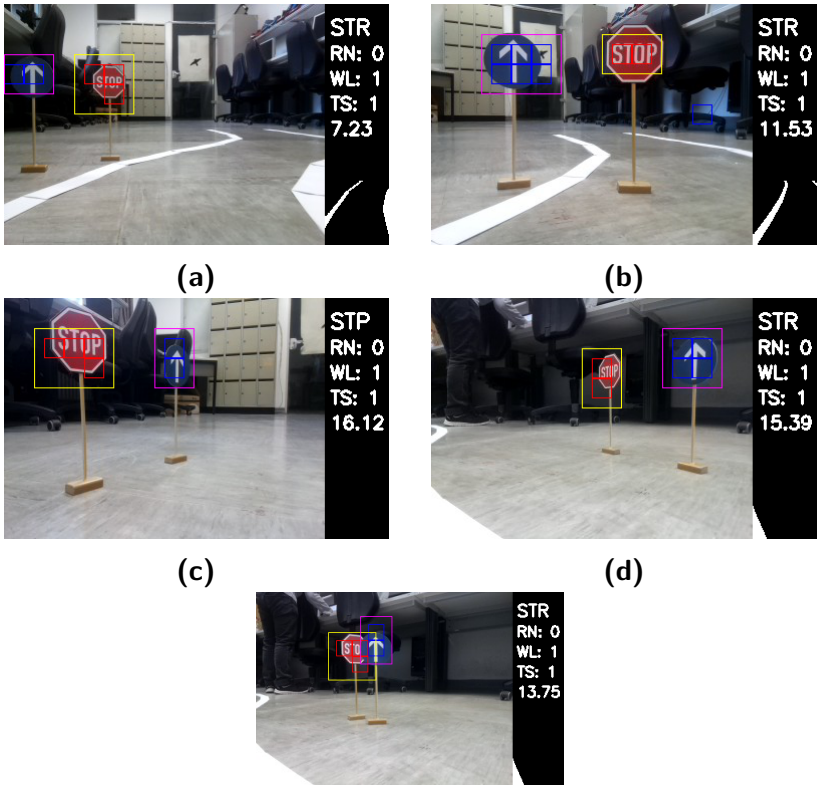


Figure 4.4.1: Testing the sign recognition model. This model performs well if there is no drastic change in light source. It can recognize accurately both normal signs and angled signs, and would often be able to recognize overlapping signs.

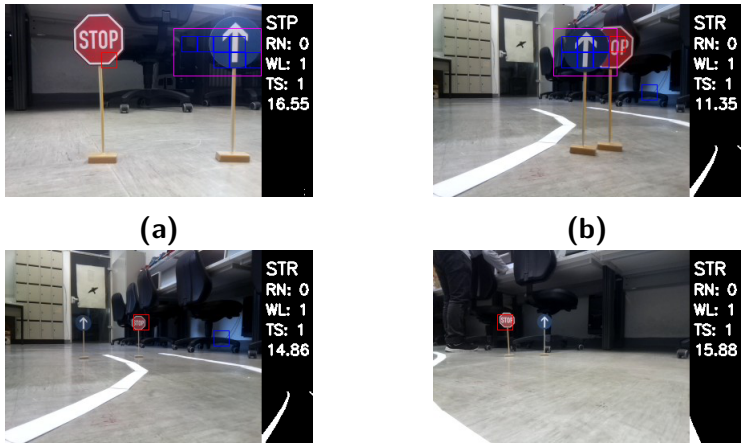


Figure 4.5.1: Failure cases of the sign recognition model. (a) fails to detect the stop sign because of the back light on top right corner, (b) fails to properly recognize the stop sign behind the forward sign, (c) and (d) miss the small forward sign.

5

Graphical User Interface

In order to make it easier to implement the models, a Graphical User Interface (GUI) was developed.

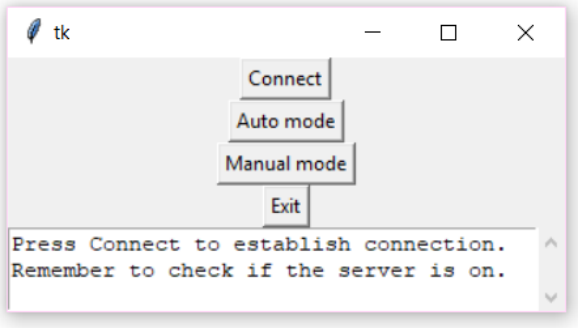


Figure 5.0.1: Main menu. Right now there is still no connection, as shown by the message box.

After connection is established by clicking the "Connect" button, the program enters stand-by mode.

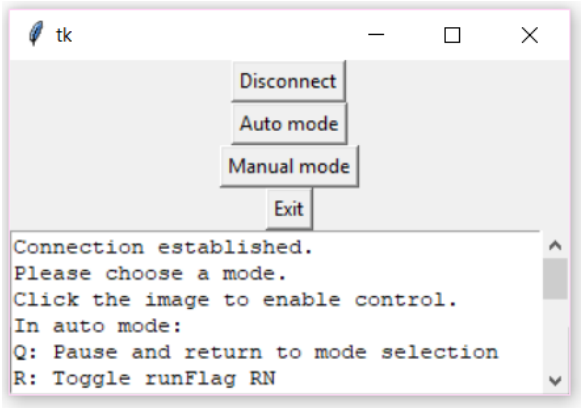
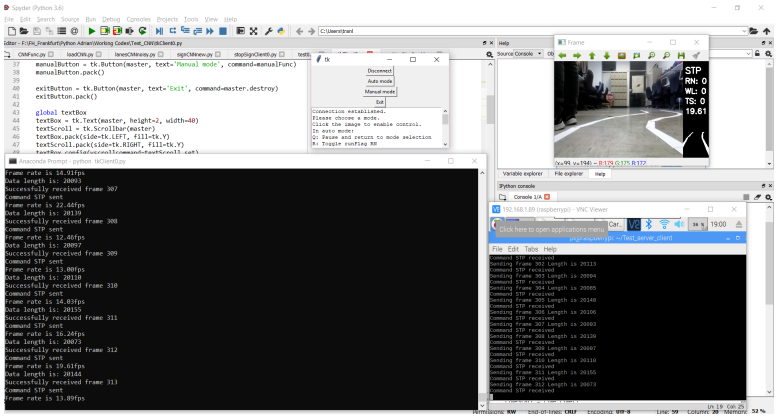
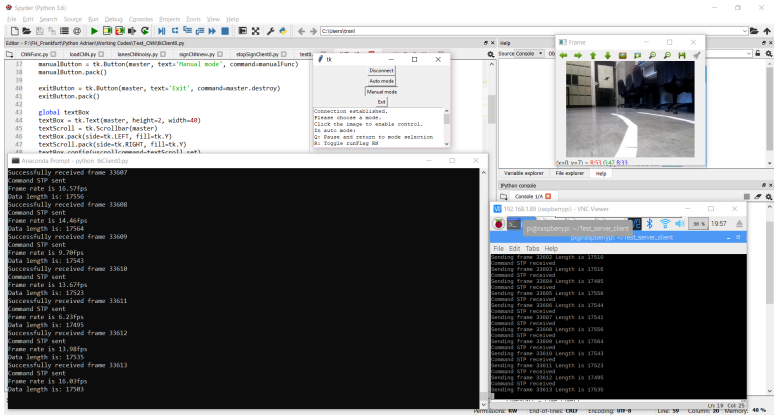


Figure 5.0.2: Standby menu. After the Internet connection is established, the tk window enters standby mode. Users can now read the user guide and choose a mode.

During the stand-by phase, the user can change to either auto mode or manual mode, or disconnect and end the program.



(a) Auto mode.



(b) Manual mode.

Figure 5.0.3: Streaming modes. The frame windows can also accept keyboard input, so it is possible to either control the car in manual mode or toggle the flags in auto mode.

6
Conclusion

Using the Raspberry Pi 3 as a controller, I successfully built a car which can steer between two white lanes, plus detecting the presence of a stop sign and a forward sign at the same time with the help of a computer. To achieve this, the car first establish a TCP connection with the computer through wireless Internet, then the control loop starts: First, the Pi sends length of the next frame to the computer, then it will save that number and send an ACK back tothe Pi. After it receives that ACK, the Pi send an image frame to the computer. Immediately after receiving a frame, the computer quickly processes it using the light weight CNNs for steering and sign recognition, then sends a command based on the results back to the Pi. The car will be driven according to that command, then the Pi start streaming the next frame. The cycle repeats until a command to suspend or end the connection arrives from the computer. The whole systems runs at 14 fps on average, but it

occasionally drops to 3-4 fps for a very short time. This is presumably caused by packet loss due to the unstable wireless connections. Because TCP enforces packet ordering, the sender will have to resend the lost packets.

As it is right now, this project still have a lot of potentials and can be developed in various directions:

- Change the classification problem of the steering network into a regression one so it can truly steer between the white lanes.
- Train a small network to stack on top of the current sign recognition model to accurately determine bounding boxes of traffic signs.
- Use one shot detector method for the sign recognition model. However, finding a suitable dataset or make one will be challenging.
- Change the wireless communication protocol from TCP to UDP, which may boost the frame rate.
- Multithreading to make image send/receive independent from image processing.

References

[1] Ross Girshick. Fast r-cnn, 2015.

[2] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation, 2014.

[3] Song Han, Jeff Pool, Sharan Narang, Huizi Mao, Shijian Tang, Erich Elsen, Bryan Catanzaro, John Tran, and Willian J. Dally. Dsd: Regularizing deep neural networks with dense-sparse-dense training flow, 2016.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[5] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[6] Andrej Karpathy. Convolutional neural networks, 2018. URL http://cs231n.github.io/convolutional-networks/.

[7] Keras. The python deep learning library. https://keras.io/, 2018. Accessed: 2018-07-15.

[8] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed4, Cheng-Yang Fu, and Alexander C. Berg. Ssd: Single shot multibox detector, 2016.

[9] Y. Nesterov. A method of solving a convex programming problem with convergence rate $o(1/k^2)$, 1983.

[10] Joseph Redmon and Ali Farhadi. Yolo9000: Better, faster, stronger, 2016.

[11] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection, 2016.

[12] Shaoqing Ren, Kaiming He, Ross Girshick, , and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks, 2016.

[13] Adrian Rosebrock. *Deep Learning for Computer Vision with Python Practitioner Bundle*, chapter 2, pages 13–29. PYIMAGESEARCH, 2017.

[14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions, 2015.

[16] Tensorflow. An open source machine learning framework for everyone. https://www.tensorflow.org/, 2018. Accessed: 2018-07-15.

Colophon

This thesis was typeset using LaTeX, originally developed by Leslie Lamport and based on Donald Knuth's TeX. The body text is set in 11 point Arno Pro, designed by Robert Slimbach in the style of book types from the Aldine Press in Venice, and issued by Adobe in 2007. A template, which can be used to format a PhD thesis with this look and feel, has been released under the permissive MIT (X11) license, and can be found online at github.com/suchow/ or from the author at suchow@post.harvard.edu.