PATH FINDING OF AUTO DRIVING CAR USING DEEP LEARNING

by

Chih Yung Tseng

A Thesis

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Master of Science in Engineering

Department of Electrical and Computer Engineering

Fort Wayne, Indiana

August 2020

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Guoping Wang, Chair

Department of Electrical and Computer Engineering

Dr. Chao Chen

Department of Electrical and Computer Engineering

Dr. Bin Chen

Department of Electrical and Computer Engineering

Approved by:

Dr. Chao Chen

For my lovely family who support me unconditionally in my studies abroad.

ACKNOWLEDGMENTS

I would like to thank Dr. Wang, my thesis advisor, for his ongoing support and encouragement for me to pursue this work. His guidance and expertise throughout the entire process allowed me to adjust and improve myself and my work. I would also like to thank my friends, Willie and Ronald, for their inspiration and motivation (especially in coding logic, program optimized, and extra hand to set equipment environment). Last but not least, I would like to thank my family for their unconditional support of my studies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

CNN	Convolutional Neural Networks
GPU	Graphical Processing Units
BP	Backward-Propagation
Tanh	Hyperbolic Tangent
ReLU	Rectified linear activation unit
YOLO	You Only Look Once
IOU	Intersection over union
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
Restnet	Residual Network
PDE	partial differential equation
SGD	Stochastic gradient descent
MSE	Mean square error
MAE	Mean absolute error
FAIR	Facebook's AI Research Lab
TP	True Positive

LIST OF SYMBOLS

f_{ij}	The Coefficient of Convolution Kernel
d_{ij}	The Corresponding Value of The Pixel.
F	Sum of The Coefficient of The Kernel
V	Output Pixel Value
z_i	Data Point in Loss Function
w	Update Vector.
μ_L	Momentum.
y & $\hat{y}$	Real Value and Prediction Value
p	Parameters
g	Gradient
v	Velocity
μ	Mentum.

ABSTRACT

Self-driving cars can provide a sustainable, safe, convenient and congestion free transportation system. By using artificial intelligence, especially in machine learning, it can approach the achievement that we want. However, being able to infallibly recognize objects is just one of several challenges which artificial intelligence must solve. One of the solutions is to apply deep learning and computer vision. Convolutional Neural Networks' (CNN) deep architecture classification approaches have gained popularity recently for its ability to learn middle and high level image representations. In previous experiments, there were many similar examples using different CNN models to train the robot for graphic recognition and obstacle avoidance. However, there is still much room for improvement in the department of image recognition, especially pertaining to its accuracy

In this project, CNN has been applied as a training tool to process image classification and object avoidance on remote robotic cars built with the Nvidia Jetson Nano developer kit. The kit was programmed using the wireless programming environment, Jupyter notebook. In addition, two different CNN models have been applied to analyze the output result performance. The main purpose is to train the robot to identify objects and improve its accuracy. The recognition and accuracy rate under different conditions can be observed by comparing the two models with different graphic inputs conditions. This project adopts the pre-train model for real time demonstrations and can be executed in a cloudless environment (without networks involved). As a result, the robot can achieve a high accuracy rate in both CNN models output result performance. Moreover, the pre train model can execute in local service to accomplish cloudless.

1. INTRODUCTION

In our current day traffic system, human-caused traffic accidents happen all around the world. Reckless behaviors such as driving with a mobile phone, impaired driving and violations of traffic rules can be credited as the main causes for traffic accidents. Despite efforts to enforce traffic rules and ensure safety guidelines that we must follow while driving, accidents continue to occur and show no signs of improvements. Although human errors can never be eliminated, there are other methods that can minimize this issue. The use of technology is one of the solutions for this issue. In many scientific and technological research, self-driving cars have become one of the most mainstream developmental projects. Many companies such as Google and Tesla have invested considerable efforts in this area.

Self-driving cars can be used to transport people or products to certain points of destination. With little control over human error, safety becomes a top priority for researchers to focus on. In order to navigate in a complex environment and reach the target destination safely, an auto-driving car needs to be able to avoid obstacles on its path not only quickly, but more importantly, with accuracy. Due to the loud and unpredictable nature of the road environment, regular controllers represent an inadequate solution to this problem. Consequently, this portrays the necessity for development of intelligent controllers which can handle these uncertain situations. In order to develop this controller, machine learning techniques can be applied into this field. Among them, CNN is one of the more prominent and notable mainstream technologies in this area.

The strongest feature of CNNs is that they can automatically learn features from training samples, thereby eliminating the need to manually select visual features for the model. Another benefit of CNNs is that they utilize the 2D structure of the image which can allow for more accuracy than the standard flattened neural networks. This method works better than explicit feature decomposition, such as detecting lanes or adjacent cars, since the network determines the best feature to extract from the image.

1.1 Review of self-driving car in ML

There have been many recent research and development surrounding auto-driving cars. One such example is auto-car driving training in CNN networks using Raspberry Pi as an executive board to train CNN model [1]. In addition, for the image input section, the raspberry pi camera is applied as an input with image pixel resolution of 320x240 into the database. The convolution network used has 128 input nodes, 2 hidden layers with 32 nodes each, and finally an output layer with 4 nodes in each of the four outputs. For the software architecture, OpenCV and inclusive image process function has been used to train CNN. The experimental results show the unstable bias phenomenon when making a right and left turn. Unstable accuracy presents as a limitation that can be a significant issue in this experiment. [2] applied raspberry pi 3 and ultrasonic sensor HC-SR04 for obstacle detection. The CNN models' architecture consists of four convolution layers and two fully-connected vanilla layers. However, in this research, the CNN model is not of region-based CNNs, which limits computation ability and results in unpredictable security issues.

The objective of this research was to develop a mobile robotic system that coordinates with software and camera to process images with proper feedback movement. One of the goals is to achieve cloudless that process a real-time model in the local GPU. Cloudless can be defined as no networks involved which model would execute in local service. In addition, two types of convolutional neural networks architecture have been applied in this project that build and train the model with the software. One CNN model has been trained for object detection while the other model has been trained and processed in following the central traffic line. The other goal of this project is to achieve a high accuracy rate during the direction decision process.

The self-driving car process can be separated into three steps: model building, training process, and real-time demonstration. Each step focuses on a different component which affects model performance. For example, the quantity and quality of the image are the fundamental factor for model development. Moreover, imagery quantity indirectly affects and influences the output model accuracy. In training models, adjustment parameters, such as the number of training epochs and the loss function which are discussed in the following sections, can determine the accuracy of the final results. Finally, in the real-time demonstration, analysis and recording the output result can be considered as the main purpose for the next adjustment.

In the following chapter, chapter two describes the structure, operation method and pre model of CNN. Chapter three covers the common CNN structure models which include three kinds of structures that are used in this project (AlexNet, Resnet18, and Resnet34). Chapter four and chapter five illustrate the operation logic and method of the descent gradient and loss equation in the training process. In chapter six, the preparation process of the self-driving car for obstacle avoidance is the main achievement through utilized data collection and training model. In addition, Alexnet, which has 5 convolution layers and 3 full connection layers, is used to train our CNN model. With the corresponding program codes, the process is step-by-step described.

Chapter seven also provides the step-by-step operation method and detail code description. Two CNN models, Restnet18 and Resnet34 are trained for two different purposes: object recognition and road following. Chapter eight compares and analyzes the experimental results. Road following accuracy and object detection sensitivity are the main observation target in this project. The expectation is to achieve a high accuracy rate for both training models within cloudless mode. Finally, in Chapter nine, the conclusion of this experiment is given.

2. CONVOLUTIONAL NEURAL NETWORKS

Due to the large number of graphics recognition involved for autonomous self-driving vehicles, the theory and concept of deep learning is widely utilized in this field. One specific and more prominent method is CNN. With a method of feature extraction, CNN can accurately identify what is inside an image. The accuracy must rely on the characteristics of a large amount of data to continue to learn repeatedly, in learning to meet the established conditions as far as possible, to achieve the goals. The accuracy can exceed that of humans. In this chapter, the architecture inside CNN and the logical method of operation are explained in detail.

2.1 Convolutional neural networks architecture

Convolutional neural networks (CNN) normally have a standard structure - stacked convolutional layers that are optionally followed by contrast normalization and max-pooling. Then, the structure is followed by one or more fully-connected layers [3]. Specifically, [4] provided another way to describe CNN as a biologically-inspired class of deep learning models. A single network can substitute all three stages and train end to end from raw pixel values to classifier outputs. Fig. 2.1 below displays the architecture of the CNN model.

Due to computational improvement, CNNs have been applied to relatively small scale image recognition problems (on datasets such as MNIST, CIFAR-10/100, NORB, and Caltech-101/256) [4]. Furthermore,[3] and [5] mentions other computer vision field applications such as object detection [6], facial recognition [7], and pedestrian detection [8].

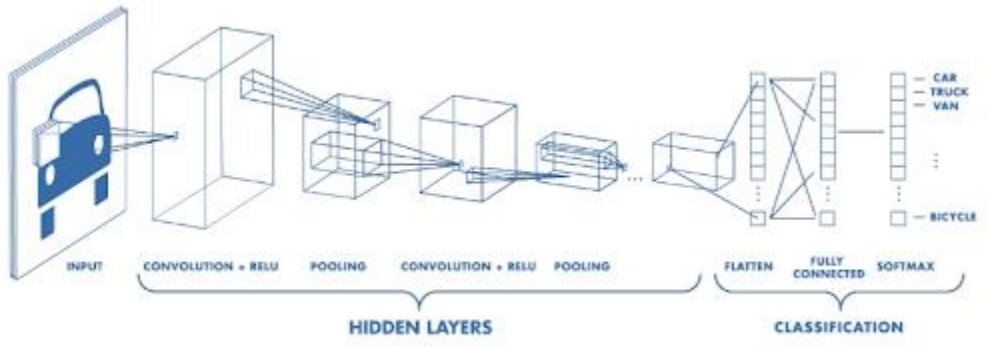

Fig. 2.1 Architecture of CNN model [21]

2.1.1 Convolution Layer

As a main building block of CNN, the convolution layer maintains the relationship between pixels by learning image features using small squares of input data. In addition, it can also be described as a mathematical operation to merge two sets of information. For example, Figs 2.2 and 2.3 demonstrate the process of convolution. Two matrices represent the input image (55) and filter kernel (33). The function for an image processing is defined as Equation 2.1.

$$V = \left| \frac{\sum_{i=1}^{n} (\sum_{j=1}^{m} f_{ij} d_{ij})}{F} \right| \tag{2.1}$$

Where f_{ij} is the coefficient of a convolution kernel at position i, j. d_{ij} is the data value of the pixel that corresponds to f_{ij}. Dimensions of the kernels are represented as n and m, assuming a square kernel (if n = 3, the kernel is 3 × 3). Sum of the coefficient of the kernel is F (value equal to 1 if sum is 0). V is the output pixel value. Different kernels can become a crucial factor for image processing.

After applying the convolution, max pooling can be utilized as the preparation for a fully connected layer. Max Pooling is mainly used on the pooling layer's side. Max Pooling is a simple concept that simply picks out the maximum value in the matrix. Max Pooling's main benefit is that if the picture is panned a few Pixels as a whole, it will have no impact on judgment and good noise resistance. Fig. 2.4 illustrates the concept of the pooling layer. In addition, average pooling is another common pooling method in this area.

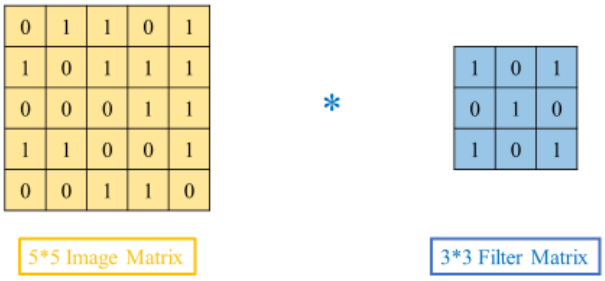

Fig. 2.2 Image matrix multiplies filter kernel matrix

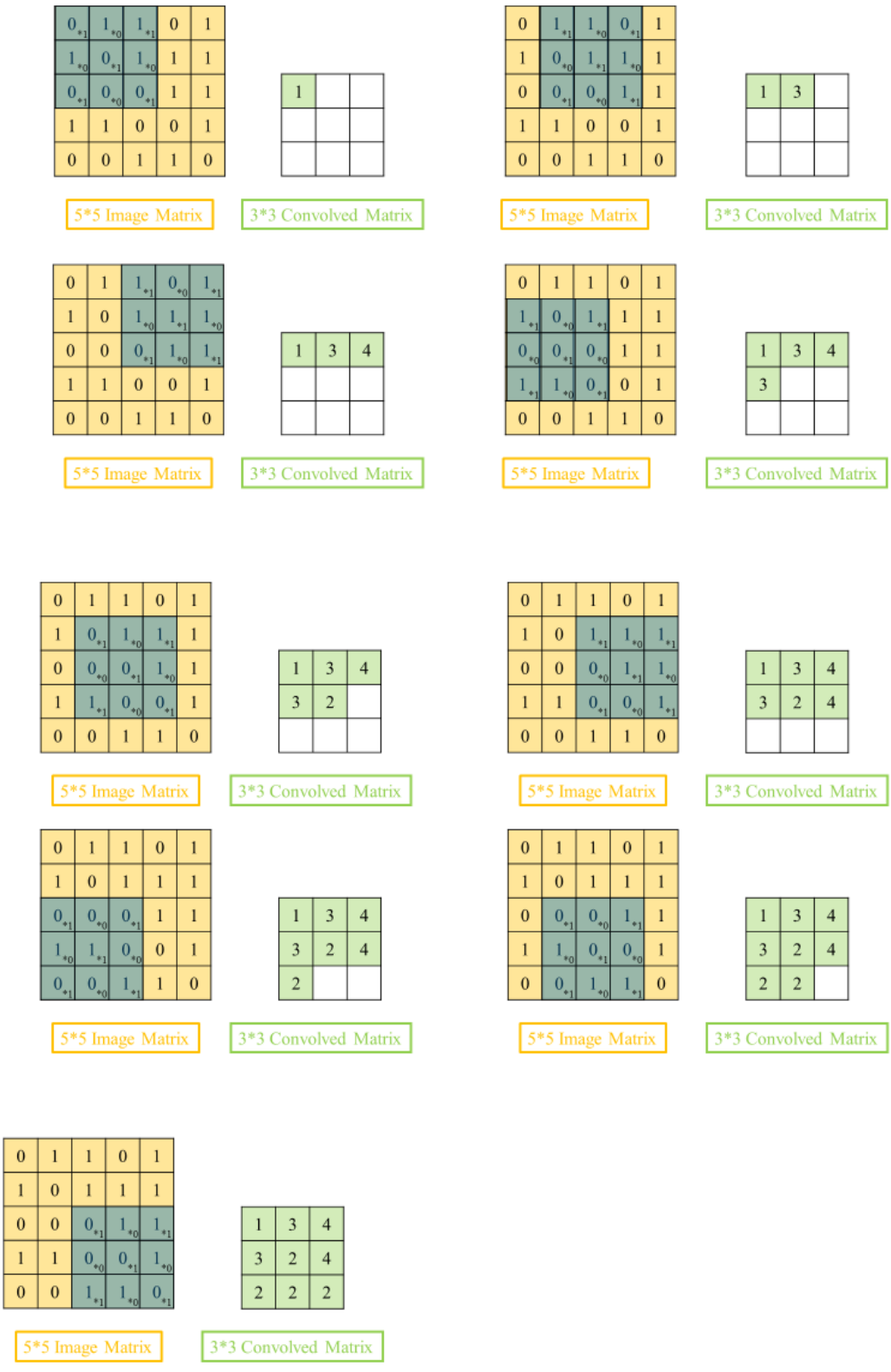

Fig. 2.3 Convolution processing step by step architecture

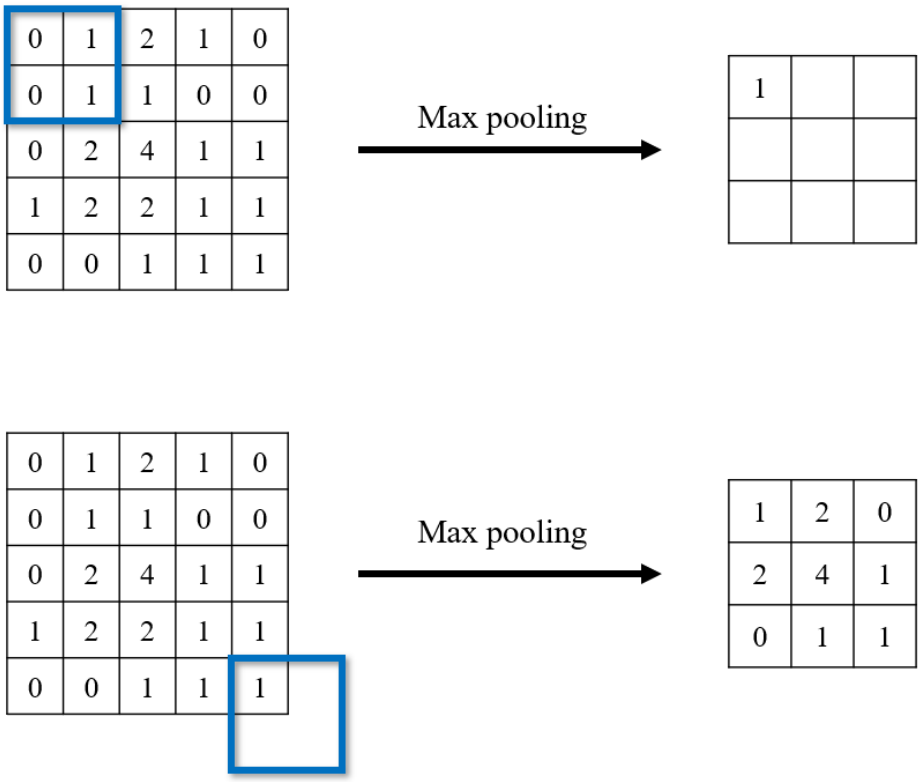

Fig. 2.4 Max Pooling Function

2.2 Pre train model

When we confront a situation with the same pattern of graphics in the input dataset, instead of building a model from scratch to solve a similar problem, using pre-trained models which have been previously trained on large datasets, allow us to directly use the weights and architecture obtained and apply the learning to our problem statement. For example, image recognition [5] and object detection [9] which categorizes as image processing can be solved through a pre-train model.

Specifically, the other method to describe a pre-train model is that created by other person who solves a similar problem or comparable pattern. The purpose of using a pre-train model is to save time on similar questions that we face. For example, CNN models, such as Googlenet and ImageNet, provide a large pre-train dataset and model that is trained in multiple situations. Moreover, by applying this CNN model, the performances of object detection and segmentation tasks significantly improve over previous methods [3][9][10]. On the other hand, the accuracy may not be 100% in any application because of the condition differences in a range of issues.

2.3 Activation function

The activation function is mainly used in the neural network to approach the non-linear problem by using the non-linear equation. Without the use of the activation function, the neural network combines the operations in a linear way. Because the hidden layer and the output layer are the inputs of results from the upper layer, the linear combination calculation is used as the output of this layer so that there is only a linear relationship between the output and the input. In reality, however, nonlinear problems are more common than linear. Therefore, the model trained by the neural network would be meaningless without the inclusion of a non-linear activation function.

Gradient descent algorithm is a very widely used optimization algorithm which will be discussed in chapter 4. Generally supervised neural networks use error backward-propagation (BP) to update the weights of the network. First, the loss corresponding to the output layer is calculated, then, loss is continuously transferred to the upper layer as a derivative and the corresponding weight parameters are modified to reduce the loss. Three common activation functions are described in the following sections which are Sigmoid, hyperbolic tangent (Tanh), and Rectified linear activation unit (ReLU) functions.

Alternatively, since there are too many layers in the deep network, activation functions, such as the Sigmoid function, often change the derivative to 0 gradually which causes the weight parameters to have difficulty updating smoothly and makes the neural network not optimized. In addition, ReLU functions are most commonly used for activation that can avoid gradient disappearance, explosion, and convergence.

2.3.1 Sigmoid function

Mathematically, the Sigmoid function takes a real number in any range and returns an output value in the range of 0 to 1. The S-shaped function produces S shaped curves, which are also used for statistics, using cumulative distribution functions with output range 0 to 1. The formula can be presented as Equation 2.2

$$S(t) = \frac{1}{1+e^{-x}} \tag{2.2}$$

However, the Sigmoid function has three major drawbacks:

1. Gradient vanishing is prone to gradient disappearance
2. Function output is not zero-centered:

In the Sigmoid function, when the input of the following neurons are all positive, the gradient value is always positive when evaluating the gradient of the weight value. Therefore, in the BP process, the weights are updated either in the positive direction or in the negative direction resulting in an uneven convergence curve, forming a bundling phenomenon and affecting the convergence speed of the model.

3. Exponential operations are time consuming

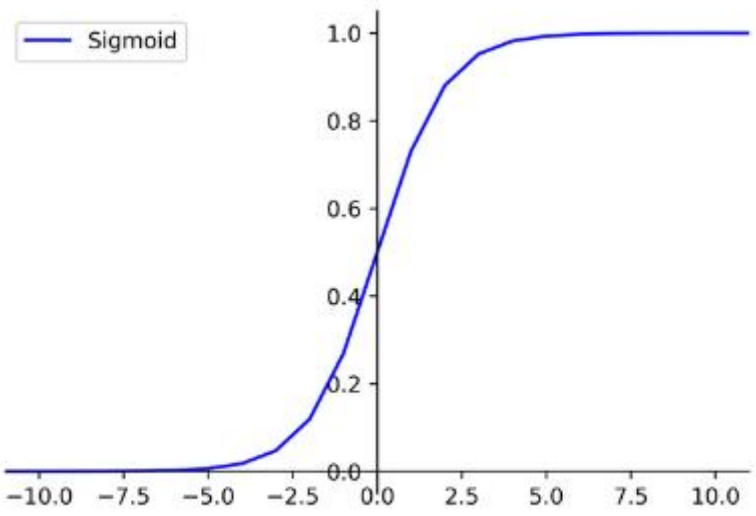

Fig. 2.5 Sigmoid function

2.3.2 Hyperbolic tangent (Tanh)

Hyperbolic tangent can resolve the output zero centered issue in Sigmoid function. Equation 2.3 below displays the tanh function. However, the problem surrounding exponential operation

and gradient vanishing are still unsolvable. As a result, this function can be considered as an early age process in deep learning.

$$\tan h(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \tag{2.3}$$

Three main characteristics, boundedness, parity, and non-periodic function can be described with the Tanh function. The graph of the hyperbolic tangent function is clamped between the horizontal straight lines y=1 and y=-1, and when the absolute value of X is large, its graph approaches the straight line y=1 in the first quadrant and y=-1 in the third quadrant.

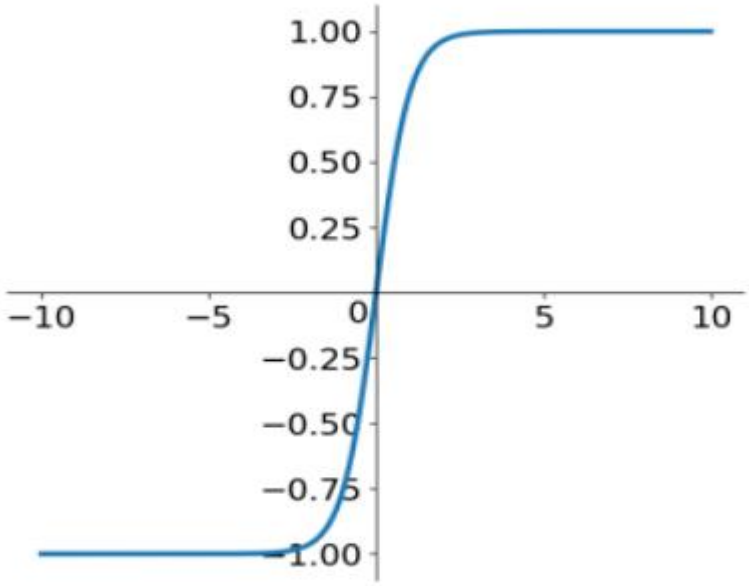

Fig. 2.6 Tanh function

2.3.3 Rectified linear activation unit (ReLU)

Fig. 2.7 illustrates the value of ReLU function. If the value is positive, then the size of the value is the output. If the value is negative, the output is 0. The ReLU function is not fully differentiable, but the non-differentiable part can be replaced by Sub-gradient, which is the most frequently used activation function in recent years. With the characteristics and ability to resolve the gradient

explosion problem, fast calculation and fast convergence and ReLU is the most frequently used activation function in recent years [11].

$$\mathbf{f}(\mathbf{x}) = \begin{cases} 0\,, if\ x < 0 \\ x\,, if\ x \geq 0 \end{cases} \tag{2.4}$$

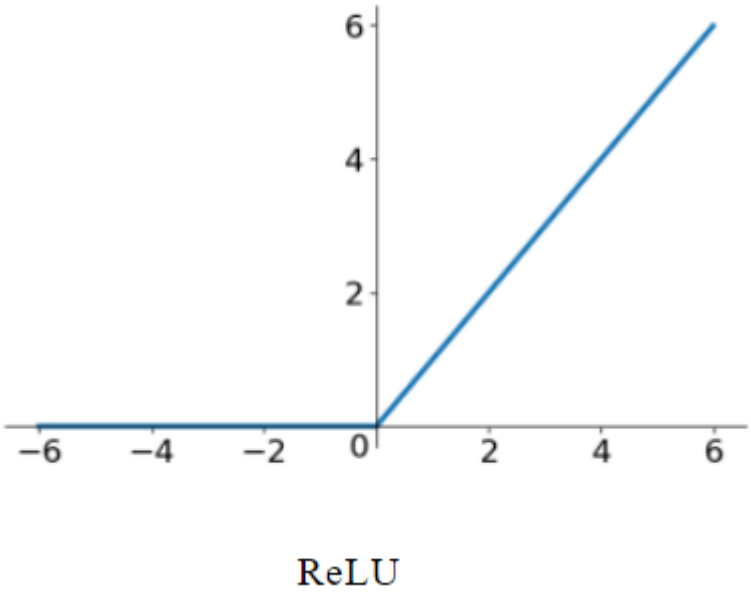

Fig. 2.7 ReLU function

For BP error rate neural network, the calculation of gradients is most important when updating weights. Sigmoid and Tanh functions have the same issue that easily eliminate gradients when input values are close to saturation. When the first-order differential value approaches zero, the gradient disappearance problem occurs which makes the error back-propagation calculation unable to update the weights effectively. The issue is more evident when the layers of the neural network increases. Therefore, it is difficult to train a deep neural network, but the piecewise linear property of ReLU can effectively overcome the problem of gradient disappearance.

Another advantage for ReLU is the computation that requires no exponential operation. Simply put, output value can be determined by whether the input value is greater than zero or not. In

addition, the ReLU excitation function makes the negative part of the output zero, rendering the network more diverse and alleviates the problem of over fitting.

2.4 Implementation

2.4.1 Object detection

Our human optical ability is fast and accurate so that we can instantly detect object location in the image [6]. This ability allows us to perform complex tasks such as walking, without colliding into objects. In order to achieve this level of movement and accuracy from a computation technical aspect, CNN is an indispensable tool and essential concept to fulfill such detection. One of the well-known object detection method is called You Only Look Once (YOLO) [6]. With unify separate object detection elements into a single neural network, it not only utilizes characteristic features from the entire image but also predicts all bounding boxes simultaneously.

The YOLO system divides the input image into an S×S grid and predicts B bounding boxes in each grid. After this process, confidence scores can be created that reflect confidential rate if the box contains an object in the model. Each bounding box consists of 5 predictions to represent coordinates that the width and height are predicted relative to the whole image. The individual box confidence predictions can be presented as Equation 2.5

$$\Pr(Class_i|Object) * \Pr(Object) * IOU_{pred}^{truth} = \Pr(Class_i) * IOU_{pred}^{truth} \quad (2.5)$$

Intersection over union (IOU) value should be between the predicted box and the ground truth. $\Pr(Class_i|Object)$ is the C conditional class probabilities prediction in each grid cell.

Fig. 2.8 displays a YOLO model that contains a single convolutional network that simultaneously predicts multiple bounding boxes. Fig. 2.9 shows the YOLO full network which are built with 24 convolutional layers followed by 2 fully connected layers

As a result, the YOLO design allows end-to-end training and real-time speeds along with maintaining high average precision. Three advantages can be delivered when compared with other CNN models [6].

1. Extremely fast processing time. The base system network runs at 45 frames per second with no batch processing which means it can process streaming video in real-time with less than 25 milliseconds of latency

2. YOLO can process the whole image and make predictions simultaneously. Compared with Fast R-CNN, YOLO makes less than half the number of background errors.

3. YOLO learns generalizable representations of objects. Along with its processing ability, it is less likely to break down when applied to new domains or unexpected inputs.

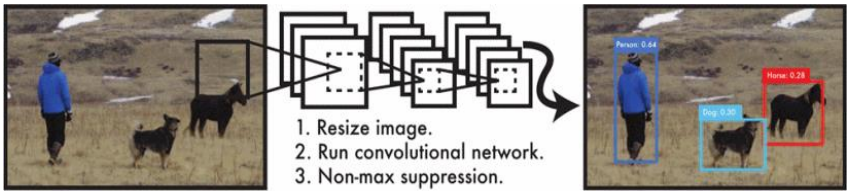

Fig. 2.8 The YOLO detection system [10]

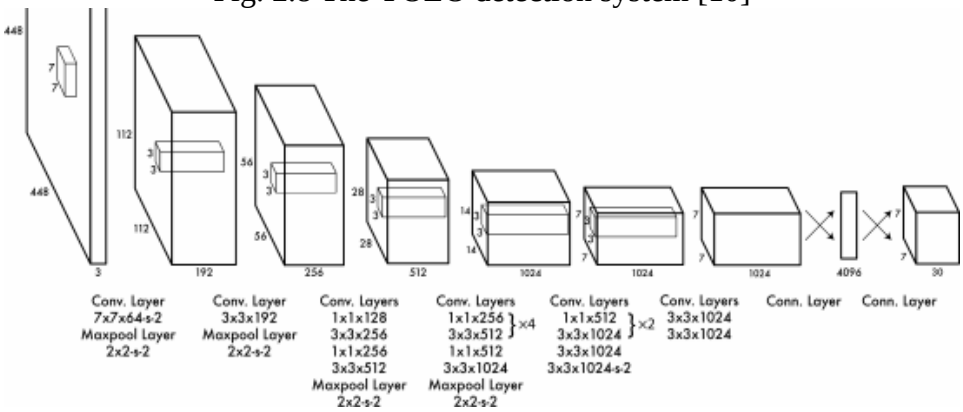

Fig. 2.9 YOLO CNN architecture [10]

3. ALEXNET AND RESNET

Due to the progress and development of science and technology, the CNN model has been continuously improved. In the performance of image recognition section, not only has the accuracy been improved, but also the processing speed has been increased. In the development of CNN models, there are some methods that have been brought special attention by researchers. In this chapter, the architecture and method of Alexnet and Resnet, which have been utilized in this research, will be discussed.

3.1 Alexnet

The AlexNet architecture was published in [12], achieving outstanding improvement performance compared with the other non-deep learning methods for ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2012. With localizing deep network and training a high-capacity model with only a small quantity of annotated detection data, object detection performs better than the systems based on simpler HOG-like features. AlexNet has five convolution layers, three pooling layers, and two fully-connected layers with approximately 60 million free parameters. In this project, AlexNet is one of CNN architectures for evaluation and analysis in the four commands classification of the project.

Compared to the previous CNN version, Alexnet has a larger computation ability with 60 million parameters and 650000 neurons and spent five to six days training with two GTX 580 gigabyte GPUs. Fig. 3.1 shows the architecture of the Alexnet model which includes 5 convolution layers and 3 full connection layers. In a single convolution layer, there are usually many cores of the same size. For example, Alexnet's first convolution layer contains 96 11x11x3 sized kernels. Note that the width and height of the kernel are usually the same and the depth is the same as the number of channels. The first two convolution layers are followed by the maximum pooling layer which will be discussed below. In addition, the third to fifth layers are directly connected.

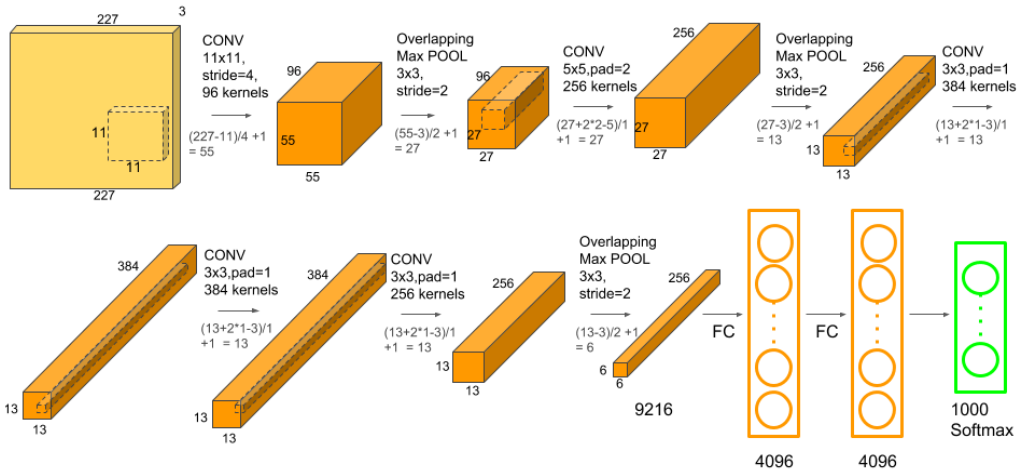

Fig. 3.1 Alexnet model architecture[12]

The maximum pooling layer is usually applied to sample the width and height of the tensor to keep the depth constant. The author utilized a 3 × 3 merging window and the span between adjacent windows is 2. The error can be reduced by 0.4% and 0.3% for top-1 and top-5 with the same output size. In addition, with exploits of ReLU function, CNN model training speed can increase tremendously than tanh or Sigmoid. [12] provides an image which displays in Fig. 3.2 that Alexnet can achieve 25% training error rate. By using cifar-10 datasets, this performance is six times of the equivalent network using tanh (dotted line).

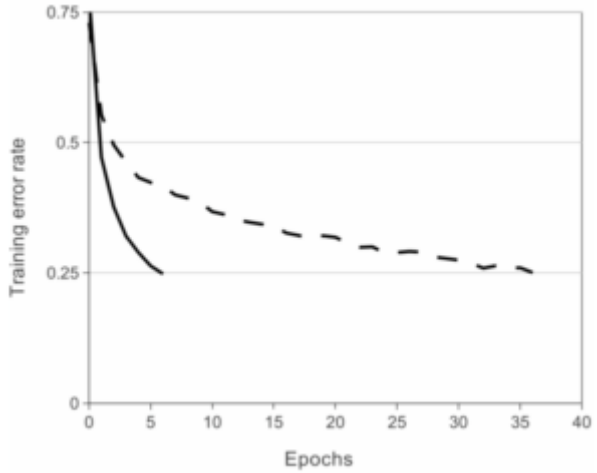

Fig. 3.2 A four-layer convolutional neural network with ReLUs (solid line) reaches a 25% training error rate on CIFAR-10 six times faster than an equivalent network with tanh neurons (dashed line).[12]

3.1.1 Enhancement dataset capacity

To avoid the over fitting problem, several methods are applied to avoid it [12]. One way is to display different changes of the same image to the neural network that helps to prevent over fitting. Mirror image can be one effective approach to increase the size of the dataset through flipping the image around the vertical axis.

The other method to increase the dataset is to randomly crop the original image into other data. For example, [12] random crop size of 256 × 256 image to extract new data into four 227 × 227 images. The four randomly cropped images look very similar, but not exactly the same. This shows the tiny movement of the neural network pixels will not change the image original characteristic. Without data expansion, the author would not be able to use such a large network because it would suffer from severe over fitting.

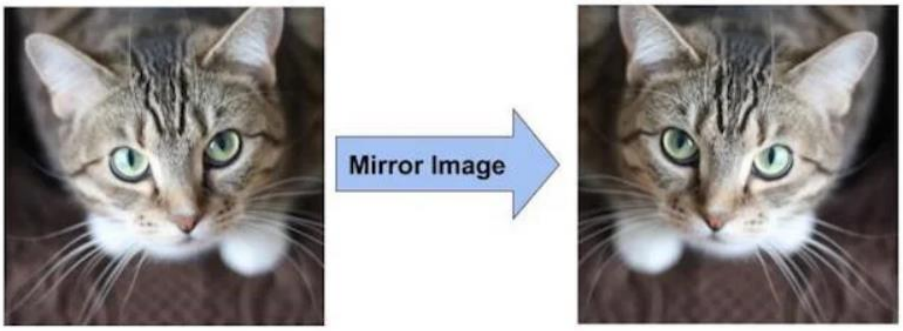

Fig. 3.3 Mirror Image to increase dataset

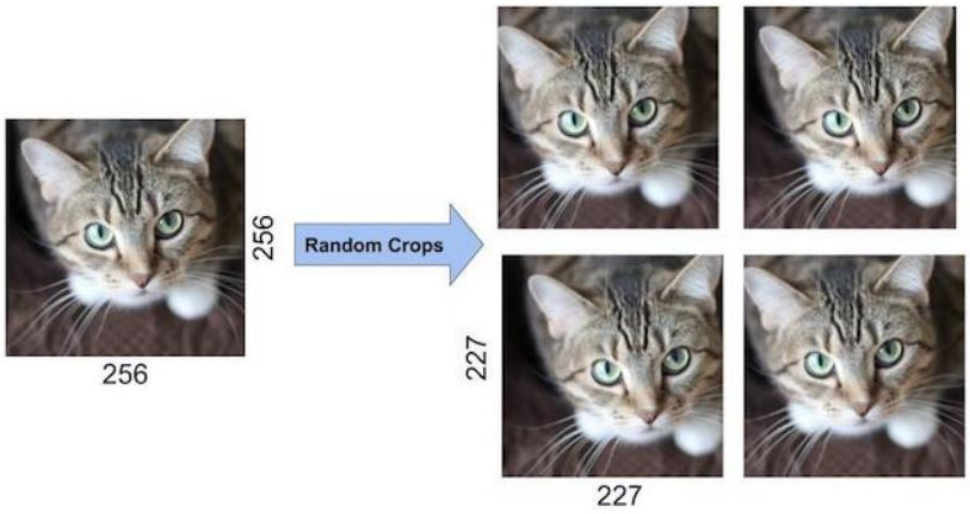

Fig. 3.4 Random Cropping Image to increase dataset

3.1.2 Dropout

The author utilized the dropout method to avoid an over fitting issue[13]. Dropout refers to that when training a large neural network, some neurons are randomly "turned off", that is, these neurons are temporarily erased from the network, which is equivalent to that in this training. These erased neurons do not participate in this training. Fig. 3.5 illustrate the concept of the network architecture.

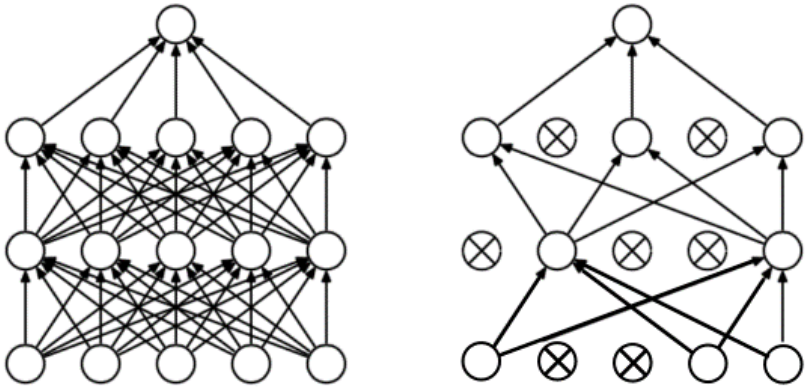

Fig. 3.5 Dropout architecture

When dropout is set to 0.5, it means that each neuron has a 50% probability of being left behind and 50% of the probability of being erased. This is equivalent to randomly sampling 50% of the nodes from the original neural network to form a new neural network. This is a sub network of the original neural network, however, the scale is much smaller than the original neural network and the training cost is also relatively small. Fig. 3.6 displays the test error trained with backpropagation using 50% dropout for all hidden layers and 20% dropout for the input layer[13].

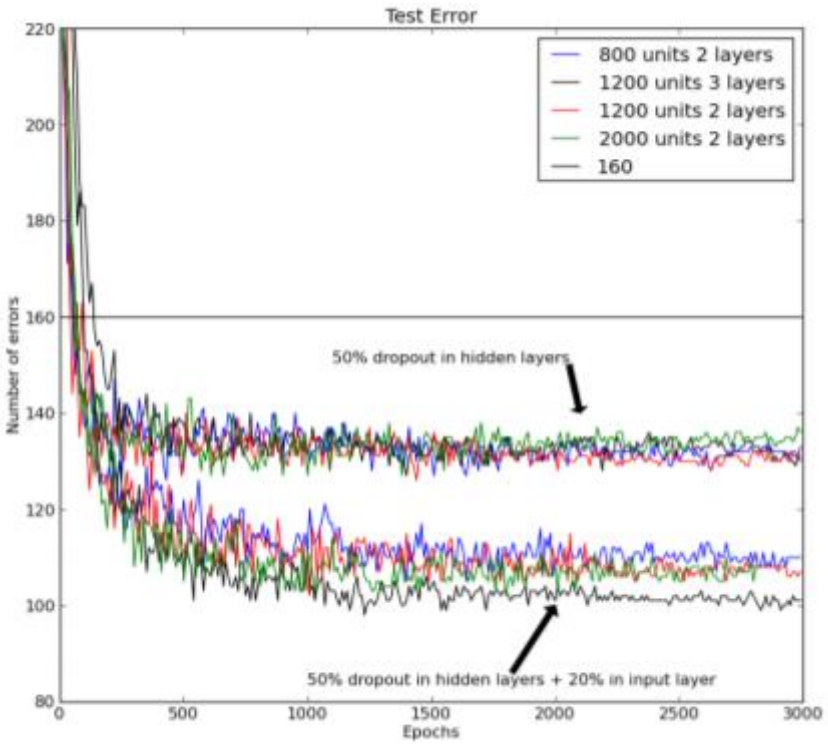

Fig. 3.6 The error rate on the test set for a variety of neural network architectures trained with backpropagation using 50% dropout for all hidden layers. The lower set of lines also use 20% dropout for the input layer.[13]

3.2 Residual Network (Restnet)

CNN models can contain multiple different layer constructions, such as AlexNet which has 5 convolutional layers. In addition, the VGG network and GoogleNet had 19 and 22 layers respectively [14][15].

According to the approximation theorem, a single layer feedforward network with sufficient capacity can represent any function. With this level of network, overfitting the data becomes a considerable issue that needs to be addressed. In this case, one method to approach this problem is that common trend in the research community is allowing network architecture to go next training section.

Because of the vanishing gradient problem which makes the gradient infinitely small, deep networks are hard to train. As a result, the network performance achieves saturated or even starts

degrading rapidly if we let the network go deeper. Fig. 3.7 displays a relationship between layer depth and performance error.

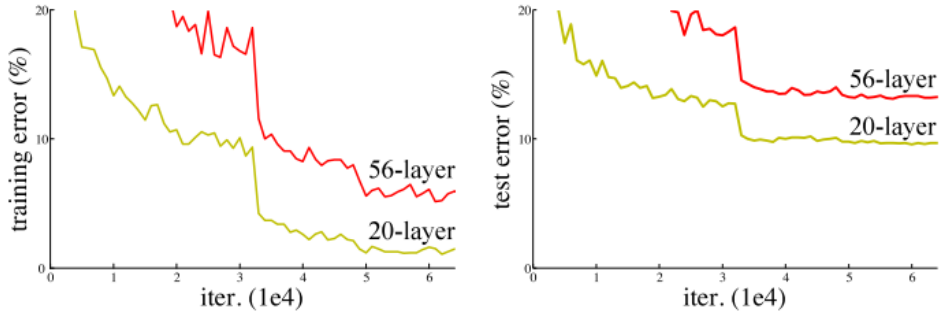

Fig. 3.7 Increasing Network Depth Leads to Worse Performance[16]

The fundamental idea of Resnet is exploiting identity shortcut connection, residual block, which skips one or more layers to deal with the vanishing gradient problem. Typically, the skipped layers contain nonlinearities and batch normalization models in between. The Equation 3.1 below describes the concept of residual block based on the partial differential equation (PDE) [16]. First of all, consider a two discrete order PDE in convolution form which shows in Equation 3.1.

$$u(x,t+1) = u(x,t) + \left[\frac{1}{2}(\sigma^2 + b), c - \sigma^2, \frac{1}{2}(\sigma^2 - b)\right] * u(x,t) \qquad (3.1)$$

the identical residual block form can be achieved by regarding the convolution kernel $\left[\frac{1}{2}(\sigma^2 + b), c - \sigma^2, \frac{1}{2}(\sigma^2 - b)\right]$ as $w(x,t)$:

$$u(x,t+1) = u(x,t) + w(x,t) * u(x,t) \qquad (3.2)$$

Any two order PDE from the above analysis can be rewritten as a residual block with the corresponding convolution kernel size equaling three. In addition, residual blocks with larger convolution kernel can be achieved by converting a higher order PDE.

Fig. 3.8 below demonstrates the difference between regular block and architecture**.** With input of x, f(x) is the ideal mapping for the path we want to obtain. Practically, the residual mapping is often easier to optimize.

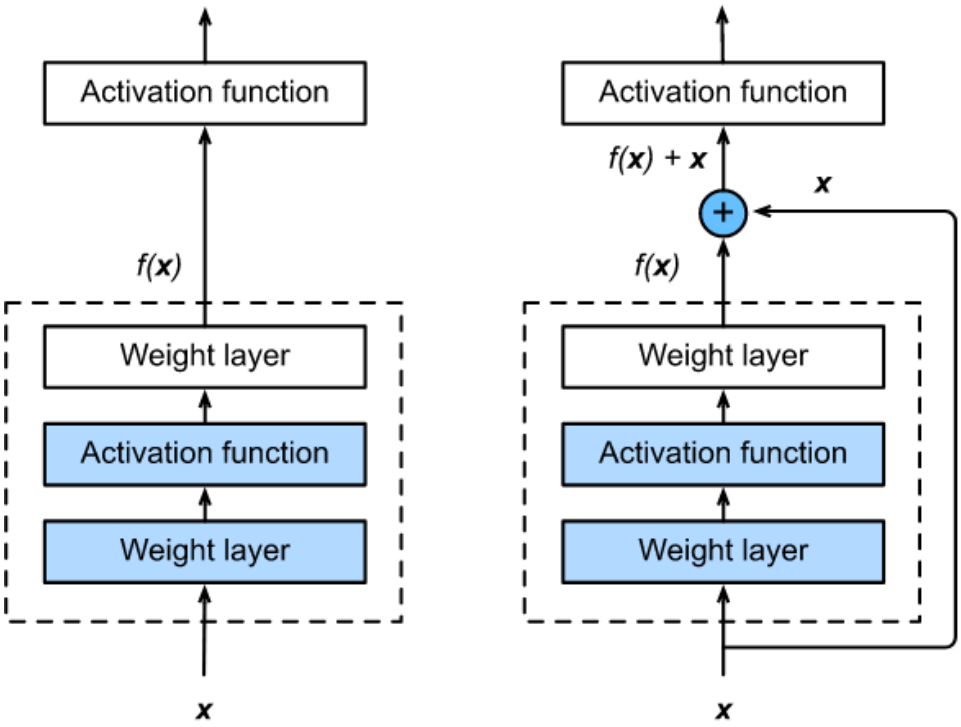

Fig. 3.8 Regular Block (Left) And Residual Block (Right)[8]

With the skipping connections between layers, as a result, the model has the ability to train much deeper networks than what was previously possible. Inspired from the VGG[16][17] network, two rules are followed in Resnet architecture that used to test the Skip Connections:

1. The filter map depth remains the same if the output feature maps have the same resolution.
2. The filter map depth is doubled if the output feature map size is halved.

Fig. 3.9 illustrates the design of a 34-layer residual network. The dotted skip connections represent multiplying the identity mapping by the linear projection term to align the dimensions of the inputs.

Through the use of residual blocks, the advantages of Resnet, which have been solved the over fitting issue, are fully demonstrated. In addition, because of the utilization of pooling function, a large number of parameters are omitted from the calculation to save computing resources which can be considered as the other strong advantage. On the other hand, Resnet also has room to be strengthened. In the original version, the last step in Resnet was to process the activation function ReLU. However, if the order of residual blocks needs to change, the output result will always be non-negative, which indirectly limits the expressive power of the model.

For the application aspect, the author utilized Resnet to execute image classification [21]. With 2698 images as input dataset, output result can achieve 93.59 percent of accuracy at Resnet18, 91.58 percent at Resnet34 and 92.67 percent at Resnet50 by adjusted training and testing parameters. Fig. 3.10 displays the output result of the image classification.

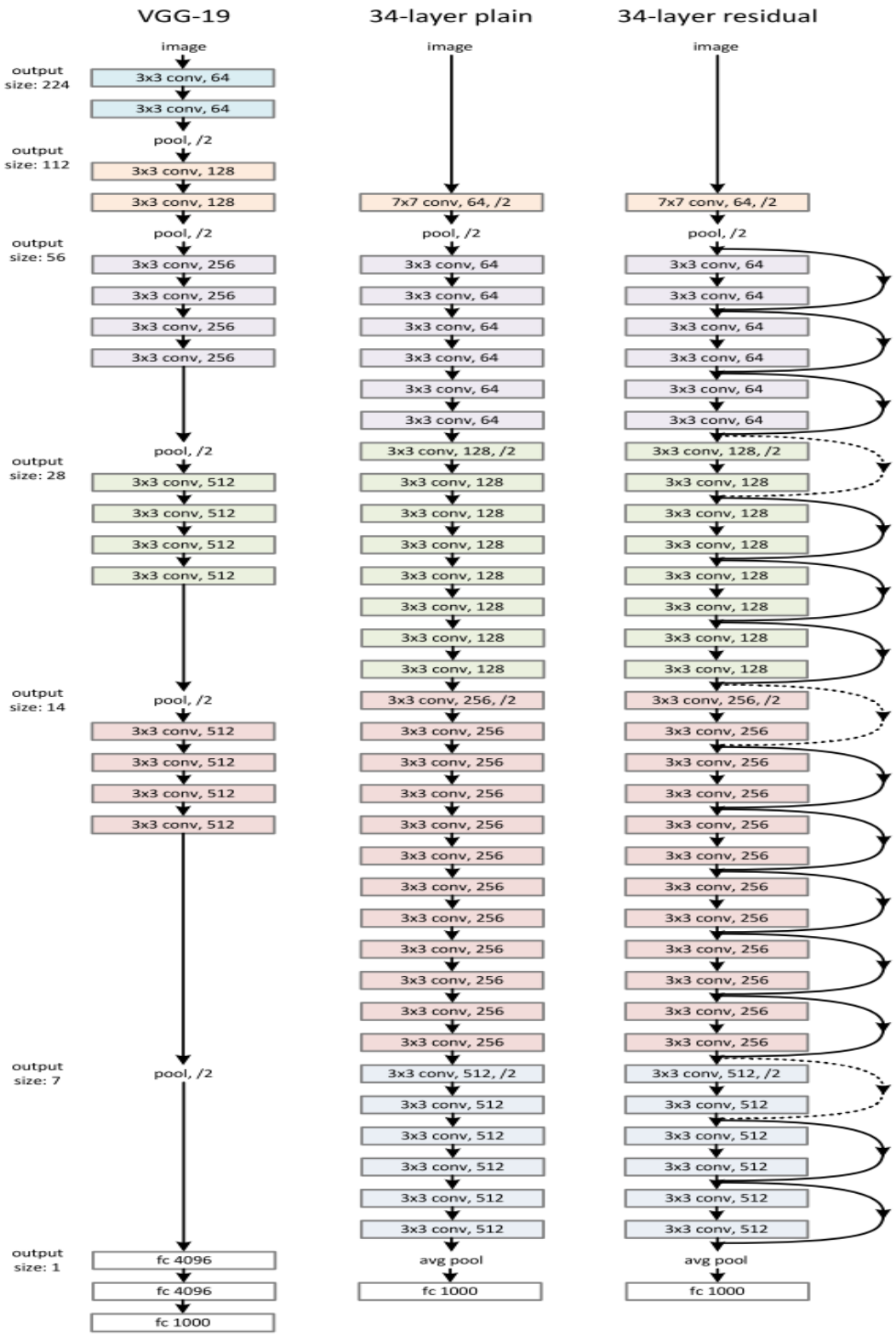

Fig. 3.9 34-Layer Residual Network Architecture

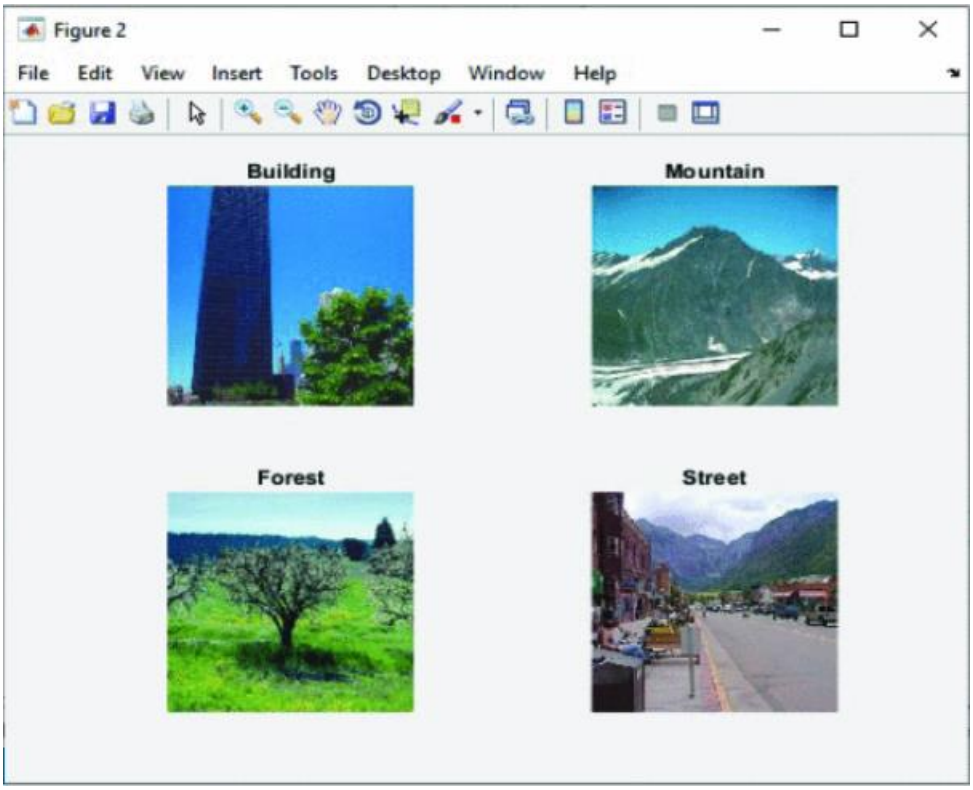

Fig. 3.10 Image Classification Result [21]

4. GRADIENT DESCENT AND LOSS FUNCTION

4.1 Concept of Stochastic Gradient Descent

Stochastic gradient descent (SGD), which is used in this project, is a very popular and common algorithm used in various Machine Learning algorithms, and most importantly forms the basis of Neural Networks. Gradient, in plain terms means slope or slant of a surface. Therefore, gradient descent represents descending a slope with expectation of the lowest point on that surface. In addition, gradient descent is an iterative algorithm that starts from a random point on a function and travels down its slope in steps until it reaches the lowest point of that function.

For the curtail algorithm, it can be determined to six steps. Starting with a random point, the purpose is to search for a different approach to update this point.

1. Find the slope of the objective function with respect to each parameter/feature. In other words, compute the gradient of the function.
2. Pick a random initial value for the parameters.
3. Update the gradient function by plugging in the parameter values.
4. Calculate the step sizes for each feature as Equation 4.1.

$$\text{step size} = \text{gradient} * \text{learning rate} \tag{4.1}$$

5. Calculate the new parameters as Equation 4.2.

$$\text{new params} = \text{old params} - \text{step size} \tag{4.2}$$

6. Repeat steps 3 to 5 until gradient is almost to zero.

4.2 Stochastic Gradient Descent

Limited computation ability can be considered as a downside of the gradient descent algorithm when inputting a large sized data set. Stochastic gradient descent, on the other hand, would randomly pick one data point from the whole data set at each iteration to reduce the computations enormously. To discover a balance between the advantage of gradient descent and speed of SGD, 'mini-batch' function is commonly utilized with sampling a small number of data points instead of just one point at each step.

4.3 Momentum

With an initial random point on the error function, gradient descent attempts to move to the global minimum of the function. Any step in a downward direction will take us closer to the global minimum. On the other hand, realistic issues are typically complex with numerous local minimal points. Fig. 4.1 displays differences between an idea and real situations which is trapped in one such local minimum point. In order to disengage this circumstance, momentum is one of the beneficial methods.

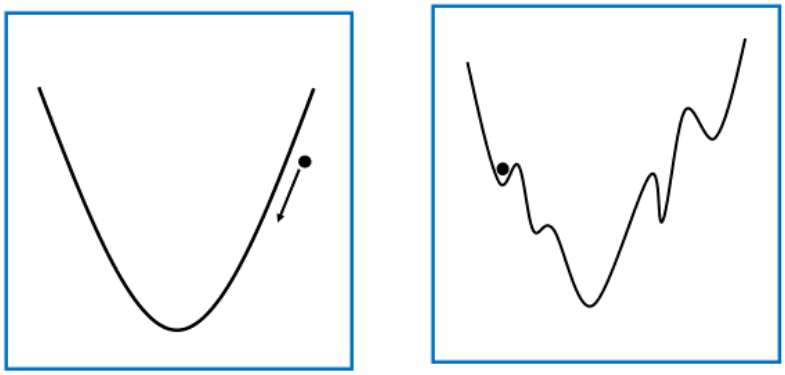

Fig. 4.1 (Left) Idea Gradient Descent; (Right)Real Gradient Descent with Local Minimum. Ball Represents Current Location of Execution Point.

Gradient Descent with momentum considers the past gradients to smooth out the update. Average exponential weighted of the gradient will be computed to improve weighted. The method can accomplish faster than the standard gradient descent algorithm. The concept of momentum is introduced by Polyak[18], which is ubiquitous in deep learning implementations. For some $\mu_L \in [0,1]$, it takes the following form which is shown in Equation 4.3.

$$w_{t+1} - w_t = \mu_L(w_t - w_{t-1}) - \alpha_t \nabla_w f(w_t; z_{i_t}) \quad (4.3)$$

The component function $f_i(w)$ represents a loss function evaluated on a specific data point z_i. Considering one term at a time, the gradient will be calculated to update the vector w. μ_L represented as a momentum.

4.4 Loss Function Algorithm

In the concept of deep learning, the purpose of the optimization algorithm is to approach the maximized or minimized value which is called 'object function'. In addition, it means that we are searching for a candidate solution that has the highest or lowest score respectively. Specifically, algorithms that are used in neural networks will mostly be loss functions.

The loss function can be basically divided into two aspects (classification and regression), which approaches the minimum value as close as possible. In actuality, predicting the same value as the actual value can be very difficult to achieve. Statistics use the term of residual to describe this unsatisfied statement.

The simplify formula below illustrates the loss function as an Equation 4.4.

$$\text{Loss} = \text{y} - \hat{y} \quad (4.4)$$

where y and $\hat{y}$ represent as real value and prediction value. The model can be determined through the value of loss where lower is better.

On the classification aspect, similarly, the error rate for the model should be as low as possible which indicates the method to perfectly separate the different types of data. Error rate equation shown in Equation 4.5.

$$\text{Error rate} = \frac{\sum_{i=1}^{n} sign(y_i \neq \hat{y}_i)}{n} \, , sign(y_i \neq \hat{y}_i) = \begin{cases} 1 \, , y_i \neq \hat{y}_i \\ 0 \, , y_i = \hat{y}_i \end{cases} \quad (4.5)$$

Where y and $\hat{y}$ represent as real class and prediction class.

However, the error rate can not be directly taken as the loss function for optimization of classification questions. Two main reasons below illustrate the downside of the error rate.

1 Differences between the two models are difficult to observe with error rate alone

2 The subsequent model learning has difficulty obtaining a better learning direction. The model will not know whether there are many or few errors in the current model, for this reason, data can only be used to identify errors. In addition, the model does not know the direction of the best model and how much to update when learning

4.4.1 Mean square error (MSE)

With different sizes of dataset, positive and negative values can be produced during the average difference calculation. Directly adding the value can be an issue for loss function. One method to prevent this problem is exploiting MSE which represents a mean square value of difference between predicted and real value. MSE can be described as the Equation 4.6 below where n represents number of data [22].

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \quad (4.6)$$

With the learning purpose of eliminating (minimizing) the residuals, the largest residuals can be considered the first element to start with. Because the characteristics of the outlier weight will be larger than the others, the parameters will be updated in the direction of outlier error in the update which is more likely to cause model performance deterioration,

4.4.2 Mean absolute error (MAE)

The other method to approach the minimum value is to apply MAE. By utilizing absolute value, the difference value can transform into positive value. Equation has displayed in 4.7.

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{4.7}$$

Fig. 4.2 below illustrates the difference between MSE and MAE. With real value equal to 0 as example and prediction value range from 10 to -10, it can be found that the loss of the square is very large (red axis), the loss of the absolute value is relatively small (blue axis), and the change curve of the loss of the square is more linear (V word) [22].

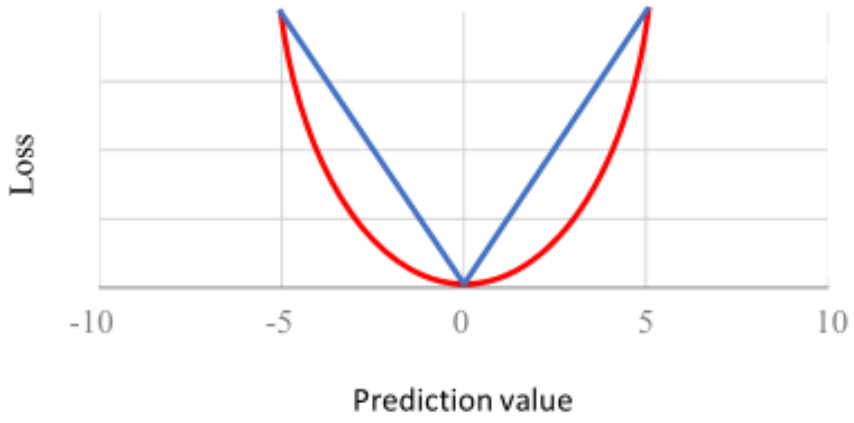

Fig. 4.2 MSE (Red Line) And MAE (Blue Line) With Real Value As 0

5. EXPERIMENT ENVIRONMENT

In studies related to auto-driving cars, researchers [1][2] have used similar methods to approach auto-driving cars. The most common method is to apply a microcomputer, like the raspberry pi, as the GPU processing center. Library of programming functions (OpenCV or Pytorch) has been utilized to train the CNN models. Depending on the direction of the study, the ultimate goal may be graphic recognition, obstacle avoidance, or direction control.

In this study, the two different CNN models, Alexnet and Resnet, are used to train object recognition and road tracking respectively. The purpose is to enable the auto-driving car to follow the midline when the environment does not detect the obstacles (traffic cones). On the other hand, when there are obstacles, it can switch modes to obstacle avoidance judgments (turn left, right or return). In addition, accuracy is one of the main objectives of this experiment. Finally, through the pre-training model, the final demonstration can be performed on a local server, reaching the goal of not using the cloud.

5.1 Software

In this project, we apply PyTorch as the library to build our CNN models which was developed by Facebook's AI Research lab (FAIR). PyTorch is an open source deep learning framework with the ability of modular and building flexibility. It contains a tape based auto-grab system to build a fast, flexible experimentation environment for Python execution.

Specifically, we applied three CNN models (Alexnet, Resnet18, and Resnet34) to build our architecture. Training process is separated into two different parts. First, Alexnet is implemented to a central divider line following with 2 different classes as output in a fully connected layer. Second, we exploit Resnet18 and Resnet34, which have 512 input features and 4 output categories, to train the image classification (traffic cone).

5.1.1 Jupyter Notebook

Jupyter notebook is an integrated architecture between IPython and notebook. In addition, it is an application environment between Editors and IDE (spider, pycharm, VIM). The platform allows you to write programs with its literal features, achieve high interactive execution results and easily present data visual execution. Two components, web application and notebook documents, can be the main factor for Jupyter notebook. Web application is a browser based interactive creation and application tools, including computing, mathematics, document creation and rich multimedia output. Notebook documents display all contents in the above web application, including input and output of calculation, document description and explanation, mathematical operation and formula, pictures and all rich multimedia contents.

In this project, Jupyter notebook has been utilized as the programing platform. NVIDEA develop kit support remote control function which allows connecting and programing in the Jupyter notebook environment. Specifically, the robot can connect the platform by navigating to http://<jetbot_ip_address>:8888

5.2 Hardware

In the choice of hardware, because the self-driving car needs a lot of graphic recognition, the strength of the GPU will directly affect the output performance. The more sophisticated the GPU, the higher the resolution and the faster and smoother the motion. For example, Raspberry Pi can be considered as a common processing board consider of low-price limitation. In addition, researchers also utilized the personal computer as a CPU processing center. Embedded lenses are the most common method for image input. Infrared sensors have also appeared inside [2]. The sensor function can be used as an auxiliary tool for object sensing detection.

In the following sections, hardware devices which are will be used in this study will be discussed including NVidia Jetson Nano Developer Kit, camera model and wireless adapter. Fig. 5.1 below shows the model configuration of the self-driving car after completion.

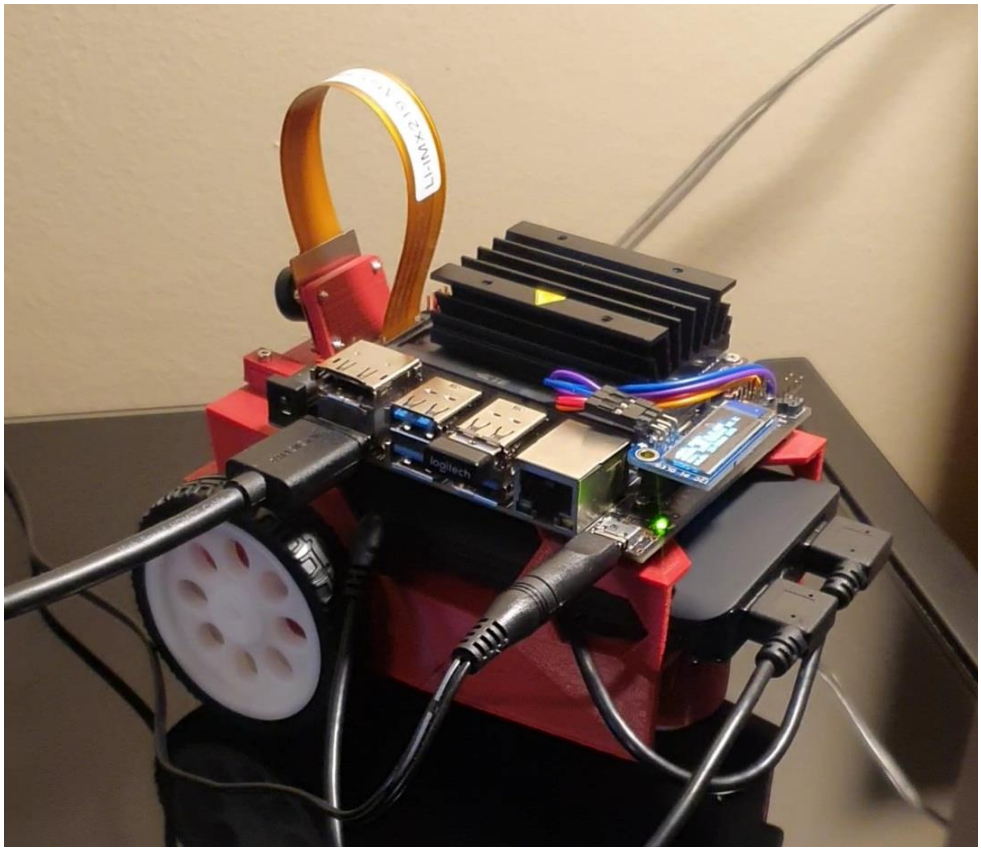

Fig. 5.1 Auto Driving Car Configuration

5.2.1 NVIDIA JETSON NANO

One of the highlight features for Jetson is the CPU-GPU heterogeneous architecture [19], combining with CPU to boot up the firmware and the CUDA-capable GPU potential to accelerate complex machine-learning tasks. In order to achieve full potential of Jetson and attain real-time performance, both Jetson hardware and CNN algorithms need to be optimized.

In this project, we applied Jetson Nano Developer Kit to develop the CNN model. The Jetson Nano has been presented in June 2019 especially for target applications where reducing the board size, power consumption, and price is important[6]. It is equipped with a NVIDIA Maxwell GPU with a peak performance of 472 GFLOPs which benefit from hardware acceleration. However, deep learning specific accelerator are not included. Two power modes can be applied (5W and 10W). Fig. 5.2 below displays the Jetson Nano hardware platform.

Fig. 5.2 Jetson Nano develop kit

5.2.2 Camera model

A camera model LI-IMX219-MIPI-FF-NANO is included inside Jetson Nano Developer Kit which displays in Fig. 5.3. The sensor type is Sony IMX219 8.08MP Color sensor and its maximum resolution can reach 3280 (H) x 2464 (V) pixel. In this project, in consideration of the memory size and execution ability, input image is set up to 224 x 224 pixels.

Fig. 5.3 LI-IMX219-MIPI-FF-NANO camera model

5.2.3 Wireless Communication system

To provide an effective training environment for the experiment, we use Intel wireless-AC 8265 adapter which is shown as Fig. 5.4. It provides three times faster network speed (867 Mbps) than 802.11n. In addition, 2.5G and 5G Hz bandwidths are supported in this adapter. Maximum operating temperature of 80 degree Celsius needs to be considered as a performance factor since the wireless adapter is directly attached at the develop kit board.

Fig. 5.4 AC 8265 Wireless Communication adapter

5.3 Trajectory Environment with Traffic Cone

In this project, the trajectory model has been built by a range of 135 x 95 cm black mat. We exploited the double side type to represent a road scenario. Fig. 5.5 illustrates the road experiment environment. In order to improve image training process, contrasting colors would be easier to identify the difference such as edge detection and central divide line.

In addition, traffic cones on tracks are the main obstacles. The location of the traffic cone can change to any place in the trajectory which achieves the mobile obstacles. The main purpose is to let the auto driving car pass through the trained model to determine the location of the traffic cone and avoid it.

Fig. 5.5 Trajectory Environment

5.4 Procedures of Robot Car Driving

Figs. 5.6 and 5.7 below illustrate the robot process logic architecture at third action movement (real time demonstration). Starting with the camera catching current time image and preprocess, four classification probabilities will be given by robot pre-train model. If the possibility of ‘free’ commend is higher than 25 percent, robot will process the road following function. On the other hand, the frame counter will increase by one and set the current statement to ‘stop’. At the same time, the robot will identify counter number statements that are greater or equal to 20. With multiple times of error shooting, number of 20, which means robot remaining at ‘stop’ command for 20 frameworks, is more appropriate for robot processing. Two consequences are produced by following the result of counter identification. The robot will execute a formula with error loss to determine right or left turn power if the stop frame counter does not reach 20. Otherwise, the robot

will proceed into exploring mode which makes a formidable right or left turn. The purpose of exploring mode is to allow the robot to escape an immovable situation that struggle by obstacle (traffic cone). Furthermore, U-turn will be executed after the number of times right and left turn equal to three in exploring mode. The intention of U-turn, similarly, is to let the robot break free from confinements with a more effective way.

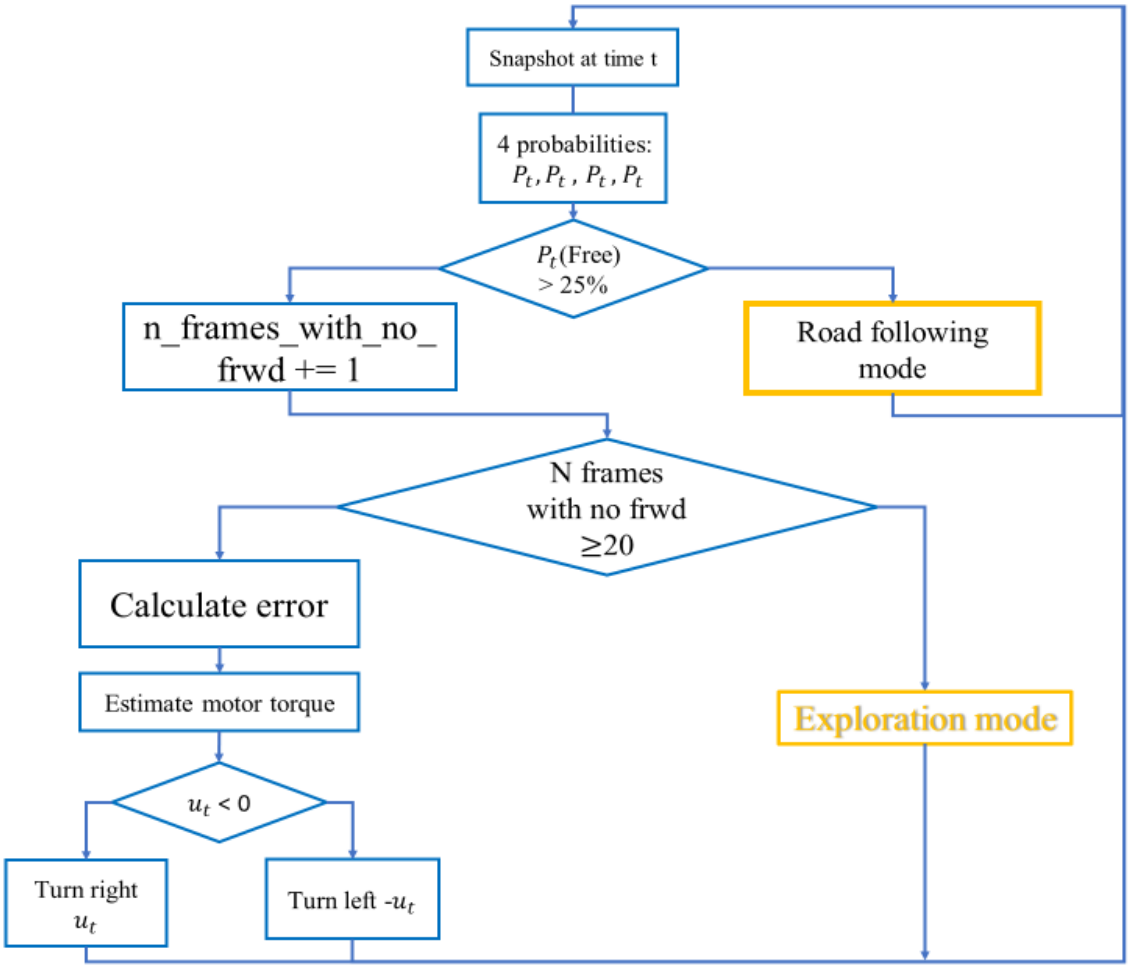

Fig. 5.6 Robot Logic Architecture

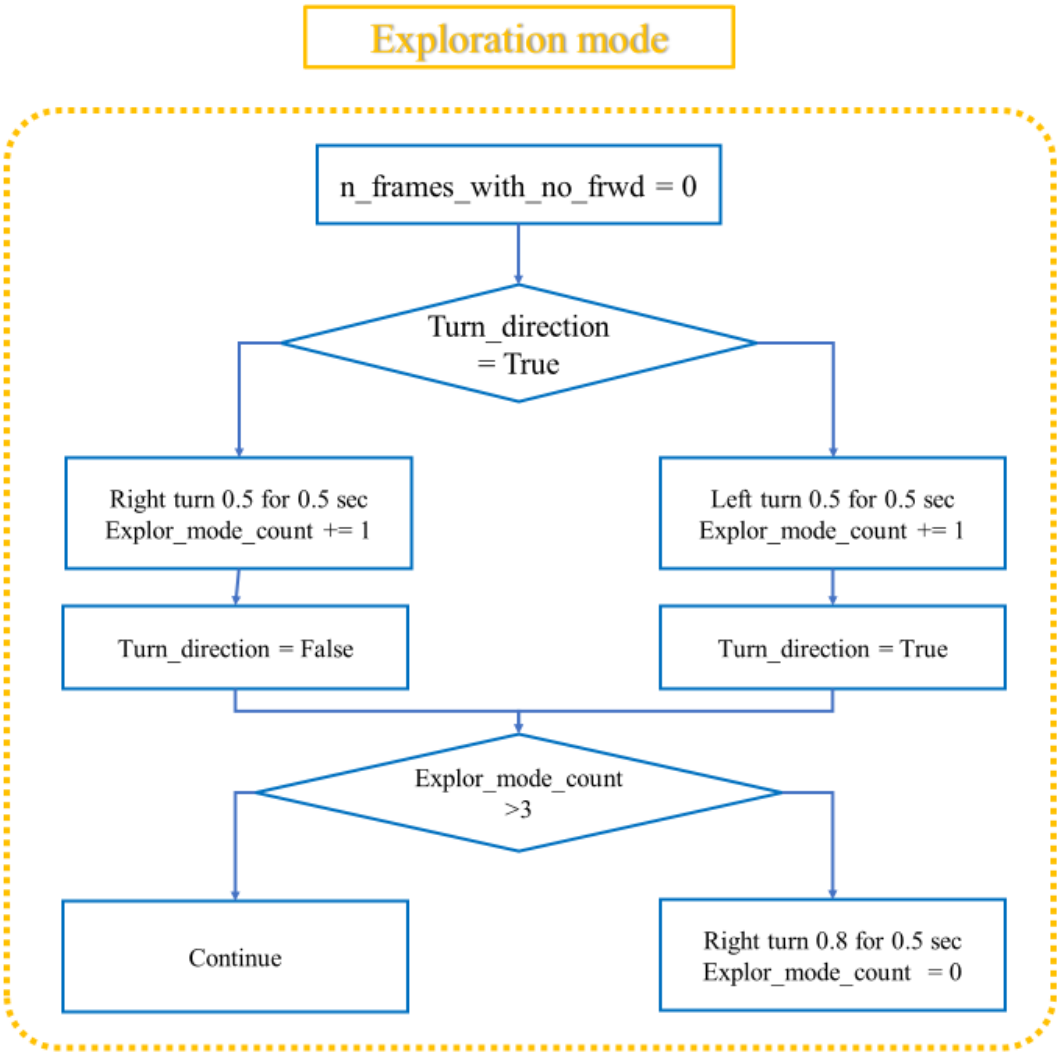

Fig. 5.7 Robot Logic Exploration Mode Architecture

6. ALEXNET APPROACH TRAFFIC CONE AVOIDED

In the first kind of model training, obstacle avoidance is the main goal that this research strives to achieve. The objective can be achieved through a three-step process: collecting data, training models, and real-time demonstration. In the training model, Alexnet, which has five convolution layers and two fully-connected layers, is used as the main training method. Fig. 6.1 below displays the three step process.

Fig. 6.1 Three Steps Process

6.1 Data Collection

In the data collection process, in order to allow the robot in identifying turning angles, the dataset contains four different classes to achieve our goal: Free, Left, Right, and Stop. Each category has 700 to 720 images in 224 x 224 pixels. The robot is manually placed in appropriate scenarios where it is suitable for four categories. Likewise, a snapshot is saved along with this label.

1. Free

The ‘Free’ statement represents a probability of no obstacles in front of the robot. In this scenario, the robot is allowed to move forward safely without collision.

2. Left

The ‘Left’ statement illustrates a scenario that probability of turning left. Robot is spinning counterclockwise which drives the right side motor. The movement depends on the current probability difference which probability turning right or left.

3. Right

The ‘Right’ statement illustrate a scenario that probability of turning right. Robot is spinning clockwise which drive left side motor. The movement depends on current probability difference which probability turning right or left.

4. Stop

The ‘Stop statement represents that the current environment has been blocked by an object (traffic cone) and expresses through probability. Framework will start counting in block situations. When counting numbers reach 25, the robot will switch to exploit mode.

In order to increase accuracy, input images need to be sampled and collected under various conditions. For example, the robot can be displayed in different orientations (e.g. sharp right vs slight right, closer to the cone or further away from it, etc.), lighting, and different textured floors. Specifically, each class input image has been placed in five to seven different environments including light difference and angle.

```
!zip -r -q dataset_cones.zip dataset_cones
```

Fig. 6.2 Zip Operation Function

After the image input process to each class finishes, the dataset is zipped to .zip file. -r flag in the zip command indicates *recursive* so that we include all nested files, the -q means *quiet* so that the zip command doesn't print any output.

Fig. 6.3 below illustrates four different floor scenarios which are wood floor, white carpet, gray wool chair, and original mat trail.

Fig. 6.3 Input with Different Floor Environment (Top Left) White Carpet, (Top Right) Gray Wool Chair, (Bottom Left) Wood Floor, (Bottom Right) Original Mat Trail.

6.2 Training model

In the training model stage, five main steps have been applied:

1. Preprocess the input images, such as uniform size and normalization.

2. The database is divided into a training set and testing set. We used 50 percent to separate the training and test set.

3. Set the number of batches we need through the ribbon

4. Determined the required CNN model and gradient descent learning rate

5. Placed Softmax function to adjust the sum of output probability to 1

In the following sections, the description for steps process based on actual operation code. Fig. 6.4 below illustrates the image pre-process function. With the ImageFolder function, input images will transform into a dataset for the training process. Colorjitter function will randomly change the brightness, contrast and saturation of an image. After resizing the image, normalization function will be used to eliminate odd numbers and restrict the number into a specific range.

```
    dataset = datasets.ImageFolder(
        'dataset_cones',
        transforms.Compose([
            transforms.ColorJitter(0.1, 0.1, 0.1, 0.1),
            transforms.Resize((224, 224)),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        ])
)
```

Fig. 6.4 Dataset Imagefolder

Dataset is randomly separated into two categories: train dataset and test dataset which is shown in Fig. 6.5. The test set will be used to verify the accuracy of the train model. The size of the test dataset is placed to 50.

```
train_dataset, test_dataset = torch.utils.data.random_split(dataset,
        [len(dataset) - 50, 50])
```

Fig. 6.5 Split dataset function

Fig. 6.6 below describes the function that can be considered as the heart of PyTorch data loading utility which represents an iteration function. Iteration is a process wherein a set of instructions or structures are repeated in a sequence a specified number of times or until a condition is met. To train our model more efficiently, the dataset is split into multiple parts called "batch". Using gradient descent which is an iterative process, the training model will process through whole batch to finish one epoch. In the project, batch size is defined to 16. Num_workers means the number of subprocesses to use for data loading.

```
train_loader = torch.utils.data.DataLoader( train_dataset, batch_size=16,
   shuffle=True, num_workers=4 )

test_loader = torch.utils.data.DataLoader( test_dataset, batch_size=16,
   shuffle=True, num_workers=4 )
```

Fig. 6.6 Dataloader for tran and test dataset

For four directions classification, model Alexnet, which displays in Fig. 6.7, has been applied to the output which comes from the 6th layer of the classifier. Bool function set to 'true' for returning a model pre-trained on ImageNet. Second parameter in the classifier, which is 4, refers to the number of outputs.

```
model = models.alexnet(pretrained=True)
model.classifier[6] = torch.nn.Linear(model.classifier[6].in_features, 4)
```

Fig. 6.7 Alexnet NN model commend

With number of 50 epochs, model can be optimized through Fig. 6.8 below that will hold the current state and will update the parameters based on the computed gradients. learning rate(lr) of 0.001 and momentum of 0.9 will be used for all parameters.

```
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
```

Fig. 6.8 Optimize Stochastic gradient descent function

Stochastic gradient descent has been applied in the function. Moreover, [20] provides the Nesterov momentum formula. Considering the specific case of momentum, the update can be written as Equation 6.1 and 6.2.

$$v_{t+1} = \mu * v_t + g_{t+1} \tag{6.1}$$

$$p_{t+1} = p_t - v_{t+1} \tag{6.2}$$

where *p*, *g*, *v* and μ denote the parameters, gradient, velocity, and momentum respectively.

Fig. 6.9 illustrates the loss function. Zero_grad clears old gradients from the last step which prevent to accumulate the gradients from all backward calls. Cross_entropy combines two different functions, LogSoftmax and NLLLoss(negative log likelihood loss), in one single class.

```
optimizer.zero_grad()
loss = F.cross_entropy(outputs, labels)
loss.backward()
optimizer.step()
```

Fig. 6.9 zero_grad and cross entropy finction

LogSoftmax can rescale an n-dimensional input Tensor so that the elements of the n-dimensional output Tensor lie in the range [0,1] and sum to 1 which define as Equation 6.3.

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \tag{6.3}$$

Obtaining log-probabilities in a neural network is easily achieved by adding a *LogSoftmax* layer in the last layer of your network. However, this module doesn't work directly with NLLLoss, which expects the Log to be computed between the Softmax and itself. The loss can be described with Equation 6.4

$$\text{loss}(\text{x}, \text{class}) = -\log\left(\frac{\exp(x[class])}{\sum_j \exp(x[j])}\right)$$
$$= -\exp(x[class]) + \log\left(\sum_j \exp(x[j])\right) \quad (6.4)$$

X is the input number of images. Class is the classification at the last output layer. Output loss can be sent back through backward function.

With the function ‘step’, a one-time calling function when the gradients are computed, the parameters can be updated to the next loop section.

Last step for the training process, which displays in Fig. 6.10, by applying feedback loss value in labels, error number can be counted through sum of absolute difference value of labels and maximum output element. Dividing with the length of the test dataset, test accuracy value can be achieved which complets one epoch. If test accuracy is smaller than the previous one, it would replace and save into the ‘best accuracy’ element until it runs through whole epochs

```
test_error_count += float(torch.sum(torch.abs(labels - outputs.argmax(1))))
test_accuracy = 1.0 - float(test_error_count) / float(len(test_dataset))
```

Fig. 6.10 Error rate function.

6.3 Robot Real-time Demonstration

```
def preprocess_1(camera_value):
    global device, normalize
    x = camera_value
    x = cv2.cvtColor(x, cv2.COLOR_BGR2RGB)
    x = x.transpose((2, 0, 1))
    x = torch.from_numpy(x).float()
    x = normalize(x)
    x = x.to(device)
    x = x[None, ...]
    return x
```

Fig. 6.11 Image Preprocess for Four Classification

In order to match the trained model format, real time input image need to do some *preprocessing* which are described in the following five steps.

1. Convert from BGR to RGB. This step lets the image process become easier. Convert from HWC layout to CHW layout
2. Normalize using the same parameters as the training. Camera provides values in the [0, 255] range and training loaded images in [0, 1] range.
3. Transfer the data from CPU memory to GPU memory
4. Add a batch dimension

After image preprocessing, the self-driving car will follow the main processing architecture to execute the following instruction which is provided in chapter six.

7. RESNET MODEL STEP PROCESS FOR ROAD FOLLOWING

In this chapter, we will discuss the second part of the experiment: road centerline tracking for auto driving cars with no obstacles involved. Similarly, the main concept is to utilize the idea of image recognition. After collecting and creating an image dataset, the CNN model will use the data, parameters, and certain training epoch to generate a pre-train model. Finally, the accuracy of road tracking is observed through real-time demonstration. In particular, two different models have been applied, Resnet18 and Resnet34, to compare the impact of accuracy under different models.

7.1 Data collection

For the road following section, the x and y values are stored in the dataset which is shown in Fig. 7.1 below. When choosing the input image, diversity of image can directly affect the robot output performance. Commend zip is also applied in this data collection section. -r represents a for recursive function to include all nested files. Fig. 7.2 displays the zip function.

```
uuid = xy_uuid(x_slider.value, y_slider.value)
```

Fig. 7.1 X and Y Value Locator Function (uuid)

```
!zip -r -q road_following_{DATASET_DIR}_{timestr()}.zip
{DATASET_DIR}
```

Fig. 7.2 Zip Function

7.2 Training model

In the training section, similarly, the image dataset is also preprocessed for further calculation. On the other hand, the test dataset was set to 0.1 instead of 0.5. In addition, Resnet18 and Resnet34 were used as CNN models to train into two different pre-train models. The purpose is to allow output result can be compared and analyzed with two different models while demonstrating

Get_x can obtain the x axis values from the image file. Similarly, Get_y function is also applied to receive values from the y axis. The purpose to locate the x and y location is to mark the direction for further image classification calculation. Fig. 7.3 displays the Get x function.

```
def get_x(path):
    return (float(int(path[3:6])) - 50.0) / 50.0
```

Fig. 7.3 Get X Axis Value Function

Dataset is split into train and test parts. Ten percent of the original dataset is randomly separated from the test dataset. The test set will be used to verify the accuracy of the model we have trained. Different test size can affect the output model accuracy. Fig. 7.4 below shows the split dataset function.

```
test_percent = 0.1
num_test = int(test_percent * len(dataset))
train_dataset, test_dataset = torch.utils.data.random_split(dataset, [len(dataset)
- num_test, num_test])
```

Fig. 7.4 Split Dataset into Train And Test Dataset

Two different NN architectures have been applied in this training section which are Resnet 18 and 34 in Fig. 7.5 below. Training with 5 and 34 layers, a pre-trained model is placed to True which allows returning a model pre-trained on ImageNet.

```
#model = models.resnet18(pretrained=True)
model = models.resnet34(pretrained=True)
```

Fig. 7.5 Resnet 18 And Resnet 34 Model Function

Resnet model has a fully connected (fc) final layer with 512 as input features and training for regression thus output features as 2. The purpose for feature setting to two is to let output pre-train model execute image classification with two different results which are following the central line or not. Fig. 7.6 illustrates the linear function.

```
model.fc = torch.nn.Linear(512, 2)
```

Fig. 7.6 Neural Networks Linear Function

In the road following train model, the number of epochs has been set to 70. Mean square error is applied to calculate loss between each element in the input x and target y. After each epoch, the best model path would be a saved input file for next step execution.

```
loss = F.mse_loss(outputs, labels)
```

Fig. 7.7 Mean Square Error Loss Function

With division length of loading dataset, train and test loss can be received. For the expectation, loss should decrease after each epoch training process. Through loss function, we can analyze the performance of the training process in order to improve the future training process. Fig. 7.8 below illustrates the calculation process for test and train loss.

```
train_loss /= len(train_loader)
test_loss /= len(test_loader)
```

Fig. 7.8 Test and train loss

7.3 Real-time demonstration

In the real time demonstration, the process can be described by three steps: image preprocessing, initial parameters setting of the spinning of the self-driving car, and the turning track of the left and right wheels. If the forward speed is too fast, the graphics input speed will not follow up to the forward speed. In the end, the stability of the self-driving car will be affected. It is important to determine the suitable turning power in this procedure.

Fig. 7.9 below shows the image preprocessing function. When real time image transmits to process function, image will also pass through pre-process function as four direction classification. After conditional expressions output with ‘free’ direction, the image will send into the road following function to execute further movement.

```
xy_2 = model_2(preprocess_2(image)).detach().float().cpu().numpy().flatten()
x_2 = xy_2[0]
y_2 = (0.5 - xy_2[1]) / 2.0
```

Fig. 7.9 Image Preprocess for Road Following

Fig. 7.10 displays the initial value for wheels parameters. Model represents the NN architecture, where x and y values are produced as output elements for the next step to utilize. Speed gain slider is defined as initial forward speed. Steering speed is dependent on the angle of the input picture. Through executing the pre train model, a central line can be located which will affect the turning angle.

```
speed_gain_slider = 0.178
steering_gain_slider = 0.053
angle = np.arctan2(x_2, y_2)
pid = angle * steering_gain_slider + (angle - angle_last) *
        steering_dgain_slider
angle_last = angle
steering_slider = pid + steering_bias_slider
```

Fig. 7.10 Initial Two Side Steering Value

X and y values are the inputs to the above function to determine left and right steering power. With multiple times troubleshooting, speed gain value and steering gain value are defined with 0.178 m/s and 0.053m/s. turning angle can be calculated by arctangent of x / y.

Fig. 7.11 below illustrates the two sides motor turning value. Left and right motor value can be defined to provide a direction and drive the robot.

```
robot.left_motor.value = max(min(speed_slider + steering_slider, 1.0), 0.0)
robot.right_motor.value = max(min(speed_slider- steering_slider, 1.0), 0.0)
```

Fig. 7.11 Motor Value Setting

8. EXPERIMENT RESULT

8.1 Train and Test Loss

First, the training status in the model can be determined by using the loss function. Ideally, the expectation for each training session or epoch is to have a lower result than the previous one. The second point to concern is that in two different loss functions, train loss must be lower than test loss. This ensures that the results of oversaturation will not occur.

With two different input images number, Fig. 8.1 illustrates the loss value in the training section which has 70 epochs by trained with Resnet18. To abstain from over-fitting, test loss value needs to be higher than train loss value.

The red line in the Figs. 8.1 and 8.2 below represents train loss and the orange line represents test loss. In the experiment, two different input dataset conditions (images of 483 and 810) were used as experimental variables in order to observe the effect of different inputs on loss function. Within 483 inputs, the result illustrates that an overfitting in the first 20 epoch. The issue can be assumed that incomplete training or insufficient dataset. On the other hand, under images of 810 input conditions, it can be observed that almost all test loss are higher than train loss. The outcome can also indicate that the size of the database can affect the training status.

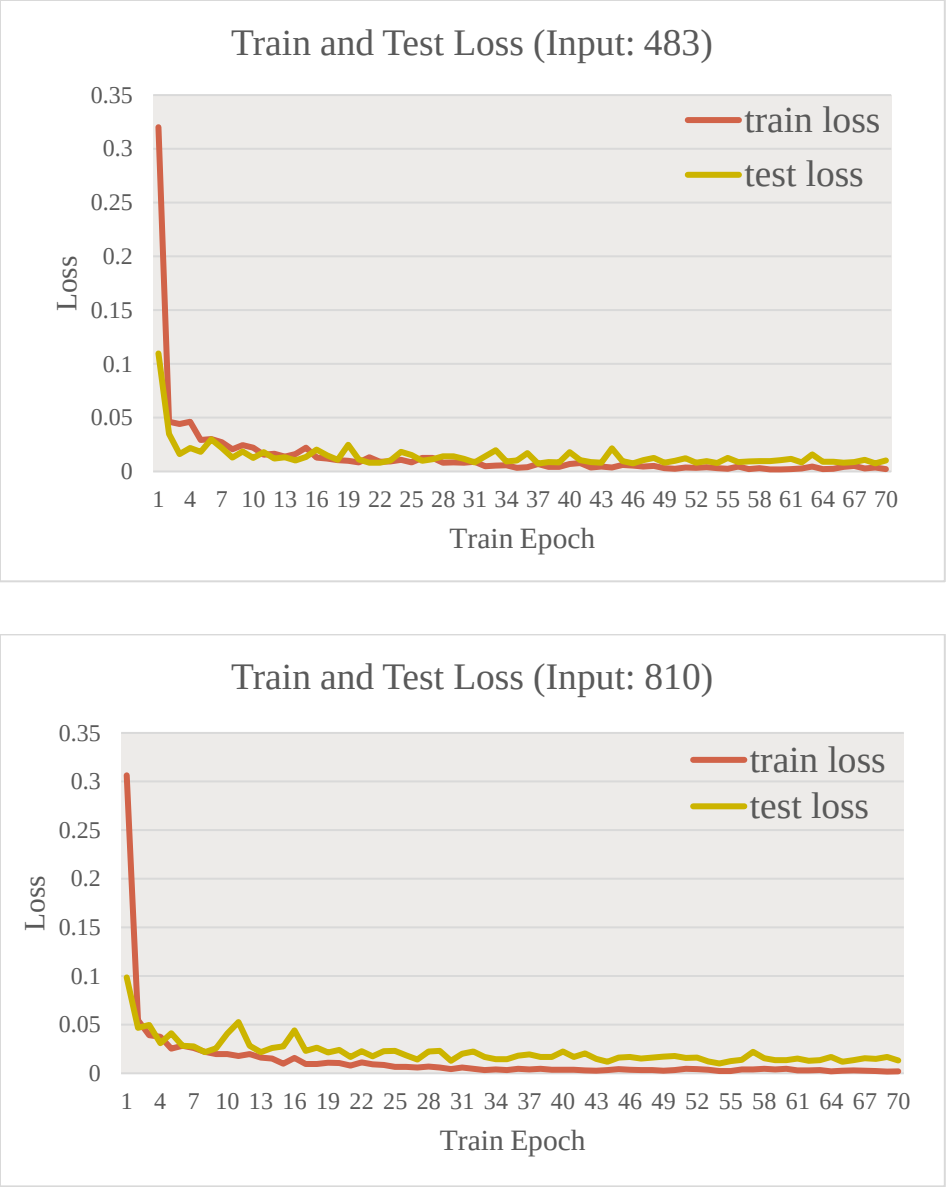

Fig. 8.1 Loss Function for Train and Test Dataset with Different Input Images

8.2 Loss in Resnet18 and 34

With the same amount of input images (810) and training epoch, Figs. 8.2 and 8.3 show the loss relation between two different NN models which are Resnet 18 and 34 in road following classification.

Through analysis, it can be found that the amount of loss is gradually reduced according to the number of training which is in line with the hypothesis before the experiment.

On the other hand, in the 65 epochs training of Resnet34, there were abnormal reaction values which were higher than previous values. The inference can be that the momentum during the gradient descent is not placed completely.

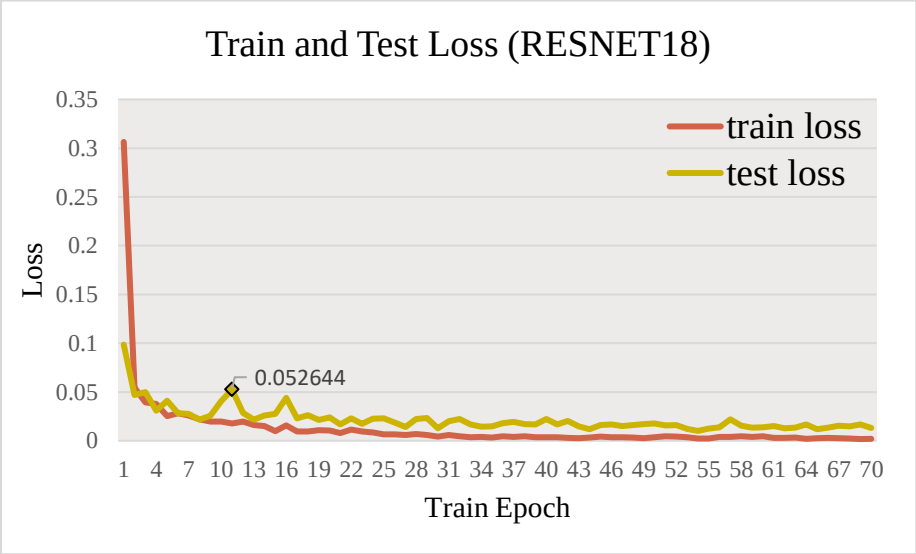

Fig. 8.2 Train and Test Loss for Resnet18 Model

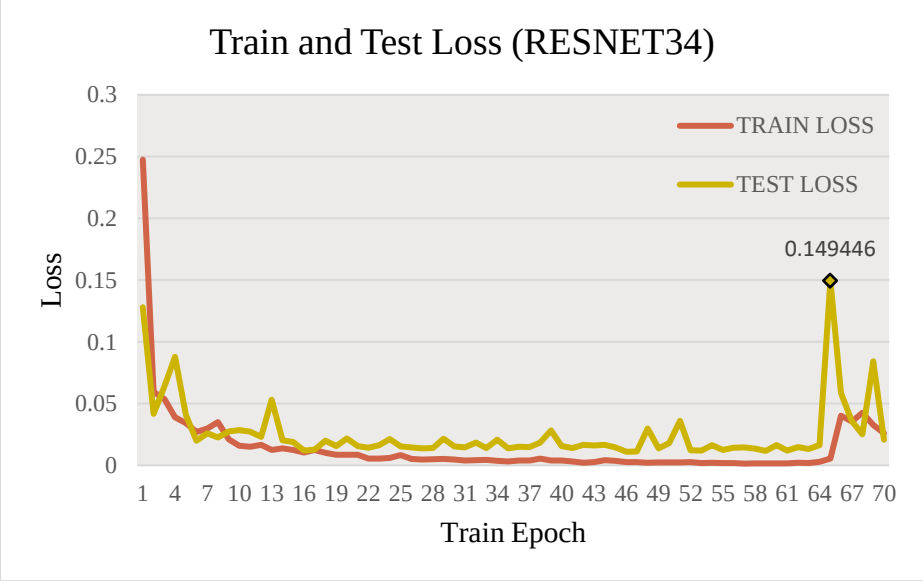

Fig. 8.3 Train and Test Loss for Resnet34 Model

8.3 Obstacle free operation efficiency

Resnet18 and 34 have been trained to operate and analyze in two circumstances: straight line and turning point. With observation the total number of output frames, accuracy rate equation is defined as Equation 8.1.

$$\text{Accuracy} = (\text{FP} + \text{FN})/(\text{TP} + \text{TN}) \tag{8.1}$$

Where used the idea of ground truth to illustrate accuracy that display in Fig. 8.4 below. For example, true positive (TP) in straight condition will be 'free' commend to execute the robot. Table8.1 displays the Resnet18 and 34 average accuracy performance in two different scenarios. Each scenario experiments 20 times epoch to obtain the average value.

For straight line accuracy, there is a crossroad in the middle of the experiment environment which displays in Fig. 8.5. Resnet18 and 34 have the same bias when approaching a crossroad which both will slightly slope to the crossroad direction. In addition, it would cause false positive (FP) happened.

For turning scenarios, Resnet34 has faster response time with average 37.3 frames for accomplishing the whole turning process. Three periods of right turn adjusting have been observed which are caused by over left turn at the beginning. On the other hand, Resnet18 presented with a lower accuracy rate (92.568%) and higher unstable performance that has six over turning reactions and two times delayed response.

		ground truth: yes	ground truth: no
system response	yes	true positive (TP)	false positive (FP)
	no	false negative (FN)	true negative (TN)

Fig. 8.4 Ground Truth

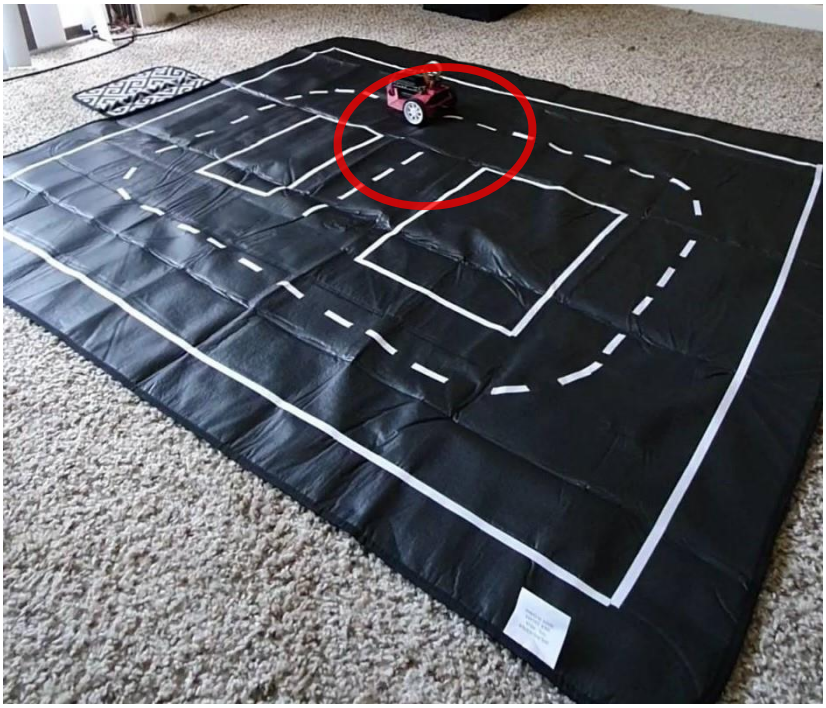

Fig. 8.5 Crossroad Environment (Red Circle)

Table 8.1 Accuracy Rate for Resnet18 and 34

	Straight line	Turning point
Resnet18	99.751 %	92.568 %
Resnet34	99.656 %	95.026 %

8.4 Obstacle collaboration Reaction time

Fig. 8.6 displays the starting and ending position which analyze the robot reaction time at right turn dilemma scenario. Line chart below illustrates the reaction time relationship between Resnet18 and Resnet34 which have maximum reaction time 20.22 and 32.51 seconds. With twenty times of testing, average reaction time for Resnet18 and Resnet34 are 11.636 and 28.995 seconds.

One thing has been discovered that Resnet34 has nine times, which represent 45 percent of the experiment dataset, turning around misbehavior instead of turning right. The reason to cause turning around can be inferred to the train model and initial element that sets depend more on Resnet18. Therefore, Resnet34 has a higher response time than Resnet18.

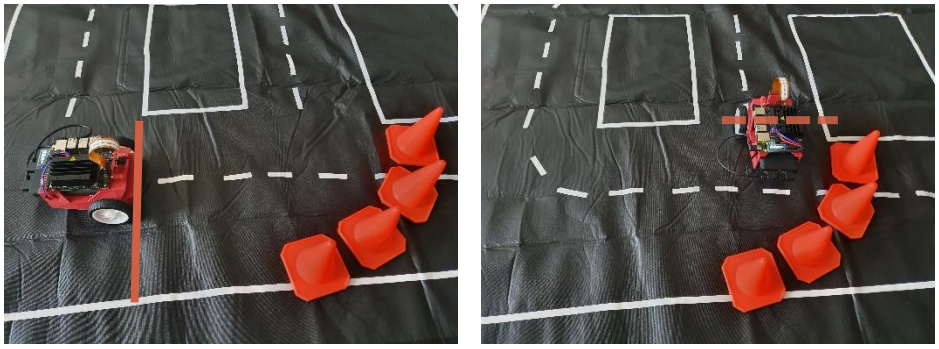

Fig. 8.6 Initial and Ending Point for Turning Left

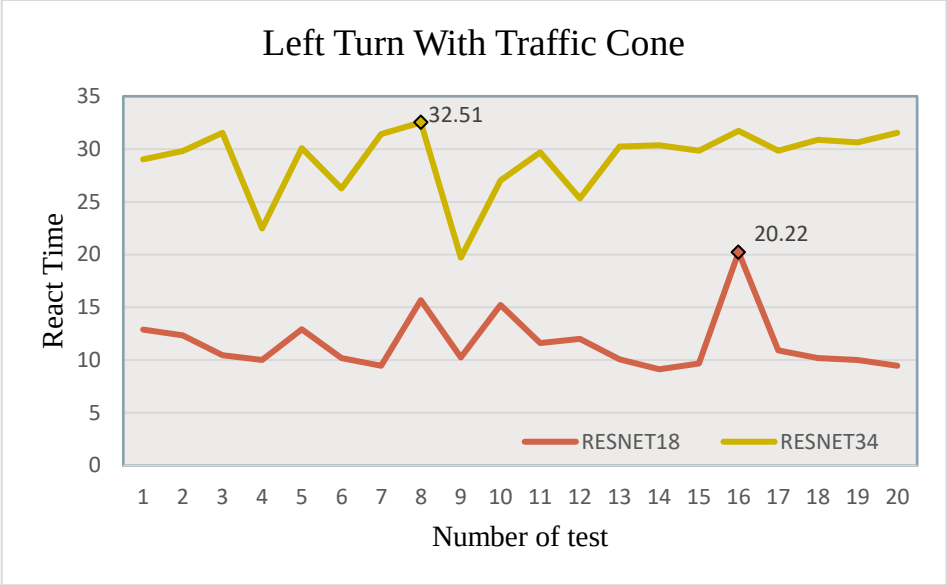

Fig. 8.7 Reaction Time for Left Turn

Fig. 8.8 shows the starting and ending position with the block road scenario. Similarly, a line chart has been provided below which presents the reaction time between Resnet18 and Resnet34. With twenty times of testing, average reaction time for Resnet18 and Resnet34 are 37.723 and 39.9225 seconds.

Resnet34 has a relatively well-behaved ability to resolve real time scenarios and can take a faster time to react after an obstacle appears. However, after turning around to track the middle line, the robot has difficulty to accurately locate the centerline, which leads to the robot approaching the boundary and needing to adjust the direction. In the result, the average reaction time is longer than Resnet18.

Fig. 8.8 Starting and Ending Point for Turning Around

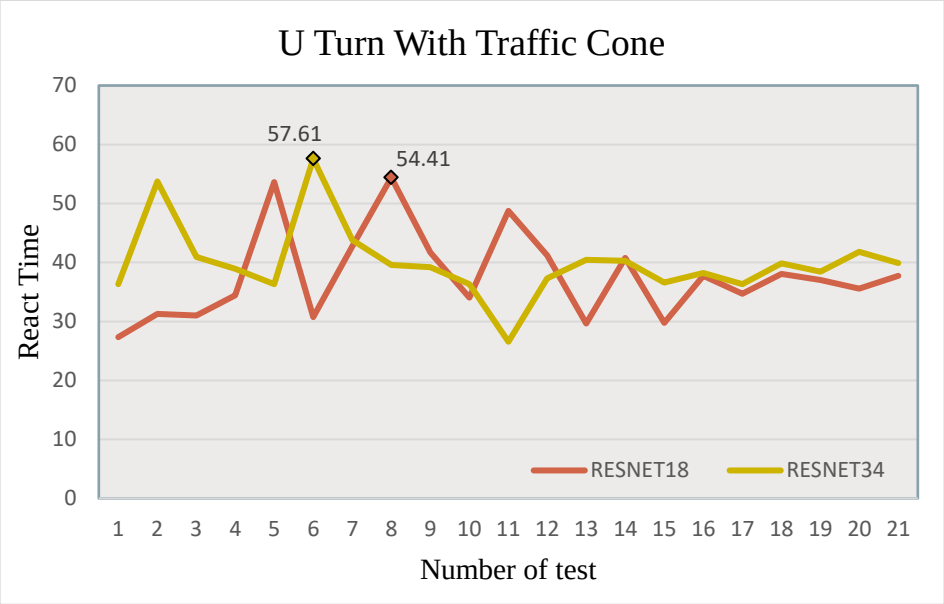

Fig. 8.9 Reaction Time for Turning Around

9. CONCLUSION

In this research, object avoidance of self-driving cars can be achieved through deep learning. The main process can be divided into three parts: data collection, training model and real-time demonstration. For our experiment environment, NVIDIA Jetson Nano Developer kit and the camera model served as the GPU processing center and input image tool. Furthermore, wireless chip AD8265 adapter has been attached into our developer kit board to provide an efficient training environment. By using Jupyter notebook and the Pytorch built-in library, the pre-train model can be created.

In the data collection section, by using the camera that is attached on the developer kit board, images can be created and stored into a dataset. Secondly, two CNN models, Alexnet and Resnet, are applied to our self-driving car as the training CNN models. With two different training modes, one following the central line in an obstacle free scenario whereas the other to detect obstacles and provide a feedback response with four different commands.

Regarding the output result testing procedures, four different testing scenarios are set for our self-driving car: straight line or turning line and obstacle (traffic cone) involved and not- involved. With a pre-train model involved, the self-driving car can achieve wireless which executes the model in local service. By recording the output result frame-by-frame, accuracy rate can be observed and analyzed.

In the performance of accuracy when there is no obstacle, the self-driving vehicle can travel along the mid-line smoothly. However, when the lighting changes, such as daylight in the daytime and indoor lights, a situation of instability presents itself. Moreover, the self-driving car will start to pause in different lighting conditions. Due to the relationship between the material of the floor track, the reflection phenomenon under the sunlight has been observed. As a result, the center line is difficult to locate correctly during interpretation, resulting in pause. In future experiments, one suggestion would be to take into account the change of light intensity and the material of the ground.

With the momentum and initial value setting biased towards correction Resnet18, it leads to a state where the reaction time of Resnet34 is higher than Resnet18 on the experimental analysis results. Another recommendation for future implementations would be to adjust training momentum value to obtain different accuracy and output results.

From the experimental results, in comparison with [2], the performance of self-driving cars in this experiment has a significant improvement. Although [2] only provided the pictures as the experimental analysis, the self-driving car would deviate (hit the track boundary) when turning. On the other hand, our research self-driving car is not easy to deviate because of the method of tracking the center line. The turning state is also relatively stable. On the other hand, because of application of the following central line CNN model, our self-driving car is more stable than [2].

In the manufacturing input dataset, the variety of pictures is limited due to environmental constraint. The accuracy of the output is therefore uncertain. For the suggestion of inputting pictures after the experiment, diversity will be a main factor to focus on (light, floor properties, etc.).

Overall, the experiment was successful. Three models were able to provide a good feedback response with significant degree of accuracy pretraining to obstacle collision or the tracking road. Regarding parameter adjustments of future experiments, it is recommended that more emphasis should be placed on the areas discussed above.

For the future research, three main directions that can be focused on. First, a variety of image collection, such as lighting conditions and background environment, is the main factor that needs to be worked on. With different input components, CNN models can observe the pattern easily and output with higher accuracy rate. In addition, quantity of the dataset can be another factor for training models. Second, parameters in the training model can be another factor to work on. For example, the number of training times and training/testing dataset allocation can affect output model results. Last, motion sensors or infrared sensors are recommended to combine into self-driving cars to increase the output model accuracy rate.

LIST OF REFERENCE

[1] A. K. Jain, "Working model of Self-driving car using Convolutional Neural Network, Raspberry Pi and Arduino," *2018 Second International Conference on Electronics, Communication and Aerospace Technology (ICECA)*, Coimbatore, 2018, pp. 1630-1635, doi: 10.1109/ICECA.2018.8474620.

[2] A. Agnihotri, P. Saraf and K. R. Bapnad, "A Convolutional Neural Network Approach Towards Self-Driving Cars," *2019 IEEE 16th India Council International Conference (INDICON)*, Rajkot, India, 2019, pp. 1-4, doi: 10.1109/INDICON47234.2019.9030307.

[3] C. Szegedy *et al*., "Going deeper with convolutions," *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Boston, MA, 2015, pp. 1-9, doi: 10.1109/CVPR.2015.7298594.

[4] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar and L. Fei-Fei, "Large-Scale Video Classification with Convolutional Neural Networks," *2014 IEEE Conference on Computer Vision and Pattern Recognition*, Columbus, OH, 2014, pp. 1725-1732, doi: 10.1109/CVPR.2014.223.

[5] C. Dong, C. C. Loy, K. He and X. Tang, "Image Super-Resolution Using Deep Convolutional Networks," in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 38, no. 2, pp. 295-307, 1 Feb. 2016, doi: 10.1109/TPAMI.2015.2439281.

[6] J. Redmon, S. Divvala, R. Girshick and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, 2016, pp. 779-788, doi: 10.1109/CVPR.2016.91.

[7] X. Qu, T. Wei, C. Peng and P. Du, "A Fast Face Recognition System Based on Deep Learning," *2018 11th International Symposium on Computational Intelligence and Design (ISCID)*, Hangzhou, China, 2018, pp. 289-292, doi: 10.1109/ISCID.2018.00072.

[8] H. Song, I. K. Choi, M. S. Ko, J. Bae, S. Kwak and J. Yoo, "Vulnerable pedestrian detection and tracking using deep learning," *2018 International Conference on Electronics, Information, and Communication (ICEIC)*, Honolulu, HI, 2018, pp. 1-2, doi: 10.23919/ELINFOCOM.2018.8330547.

[9] H. Shin *et al*., "Deep Convolutional Neural Networks for Computer-Aided Detection: CNN Architectures, Dataset Characteristics and Transfer Learning," in *IEEE Transactions on Medical Imaging*, vol. 35, no. 5, pp. 1285-1298, May 2016, doi: 10.1109/TMI.2016.2528162.

[10] K. He, X. Zhang, S. Ren and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," *2015 IEEE International Conference on Computer Vision (ICCV)*, Santiago, 2015, pp. 1026-1034, doi: 10.1109/ICCV.2015.123.

[11] Glorot, Xavier & Bordes, Antoine & Bengio, Y.. (2011). Deep Sparse Rectifier Neural Networks. Proceedings of the 14th International Conference on Artificial Intelligence and Statisitics (AISTATS) 2011. 15. 315-323.

[12] A. Krizhevsky, I. Sutskever and G. E. Hinton, "ImageNet classification with deep convolutional neural networks", *Proc. NIPS*, pp. 1097-1105, 2012.

[13] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever and R. R. Salakhutdinov, Improving neural networks by preventing co-adaptation of feature detectors, Jul. 2012, [online] Available: https://arxiv.org/abs/1207.0580.

[14] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556,2014.

[15] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition,pages 1–9,2015.

[16] Yin, Minghao & Li, Xiu & Zhang, Yongbing & Wang, Shiqi. (2019). On the Mathematical Understanding of ResNet with Feynman Path Integral.

[17] Karen Simonyan, Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. 2014.

[18] H. Zhang, Y. Luo and D. Liu, "Neural-Network-Based Near-Optimal Control for a Class of Discrete-Time Affine Nonlinear Systems With Control Constraints," in *IEEE Transactions on Neural Networks*, vol. 20, no. 9, pp. 1490-1503, Sept. 2009, doi: 10.1109/TNN.2009.2027233.

[19] C.-K. Lai, C.-W. Yeh, C.-H. Tu and S.-H. Hung, "Fast profiling framework and race detection for heterogeneous system", *J. Syst. Archit.*, vol. 81, pp. 83-91, Nov. 2017.

[20] Sutskever, I. & Martens, J. & Dahl, G. & Hinton, G.. (2013). On the importance of initialization and momentum in deep learning. 30th International Conference on Machine Learning, ICML 2013. 1139-1147.

[21] A. Mahajan and S. Chaudhary, "Categorical Image Classification Based On Representational Deep Network (RESNET)," *2019 3rd International conference on Electronics, Communication and Aerospace Technology (ICECA)*, Coimbatore, India, 2019, pp. 327-330, doi: 10.1109/ICECA.2019.8822133.

[22] J. M. Martin-Doñas, A. M. Gomez, J. A. Gonzalez and A. M. Peinado, "A Deep Learning Loss Function Based on the Perceptual Evaluation of the Speech Quality," in *IEEE Signal Processing Letters*, vol. 25, no. 11, pp. 1680-1684, Nov. 2018, doi: 10.1109/LSP.2018.2871419.