University of Applied Science Merseburg

Department of Engineering and Natural Sciences

Comparison of assistance systems for autonomous driving using Convolutional Neural Networks

A master thesis submitted for the degree of

Master of Engineering
in Computer Science and Communications System

Tran, Hoai Viet

Advisor:
Prof. Dr. Eckhard Liebscher
Prof. Dr. Andreas Spillner

Merseburg, October 2022

Abstract

After building the neural network, hyperparameters tuning is an important step in Machine Learning to improve the model performance or to customize model hyperparameters to better suit the dataset. There are different tools and packages that use grid or random search algorithms for hyperparameters optimization. But these algorithms do not indicate the importance of different hyperparameter combinations or the correlation between hyperparameters and the loss function. Deep learning models consist of multiple layers with fully-connected individual neurons that makes it complicated to understand why the model learns it that way. That is why finding hyperparameters importance is necessary to define which factors have positive or negative impacts on the model.

A deep learning model in this project will take images from the camera in the simulator as input and predict steering values. The aim of this work is to optimize the hyperparameters tuning process of CNN model. Instead of choosing and combining randomly, different sets of hyperparameters are selected systematically through multivariate quadratic regression.

Statutory Declaration

I herewith declare that I have completed the present thesis independently making use only of the specified literature. Sentences or parts of sentences quoted literally are marked as quotations, identification of other references with regard to the statement and scope of the work is quoted.

_____	_____
Location, Date	Signature

Table of Contents

List of figures

List of tables

Listings

List of abbreviations

ADAS	Advanced Driving Assistance Systems
CNN	Convolutional neural network
MSE	Mean squared error
MAE	Mean absolute error
GD	Gradient descent
SGD	Stochastic Gradient Descent
Mini-batch SGD	Mini-batch Stochastic Gradient Descent
ADAM	Adaptive moment estimation
RMSProp	Root Mean Squared Propagation
EWMA	Exponentially weighted average

Mathematical notation

- Numbers are represented by letters written in non-bold form: $x,\ y$
- Vectors are represented by bold lowercase letters: $\mathbf{x}^{(k)}$, $\mathbf{y}^{(k)}$
- Matrices are represented by capital letters: $X,\ Y$

$\mathbf{x}^{(k)}$	kth vector in a training set, round bracket represents the order of samples
$\mathbf{y}^{(k)}$	kth vector in output set
x_i	ith element in input vector
y_i	ith element in output vector
$\mathbf{w}^{[L]}$	Vector of weights at Lth layer, square bracket represents the order of layer
$\mathbf{b}^{[L]}$	Vector of biases at Lth layer
$\mathbf{a}^{[L]}$	Vector of activation function at Lth layer

1. Introduction

The groundwork of this project is based on the end-to-end network architecture of NVIDIA's research for autonomous driving [1]. A traditional pipeline for autonomous driving consists of several different components: Sensor Fusion, World Model, Behavior Generation, Planning and Vehicle Control [2].

Firstly, the input data coming from sensors and cameras will be processed in the Sensor Fusion layer to extract the relevant features, then all of these features will be combined in the World Model layer to create a complete picture of the surrounding environment. From this stage, the system must choose decisions for the vehicle in layer Behavior Generation and Planning. Finally, the system sends control values through the Vehicle Control layer to Actuator Interface modules [3].

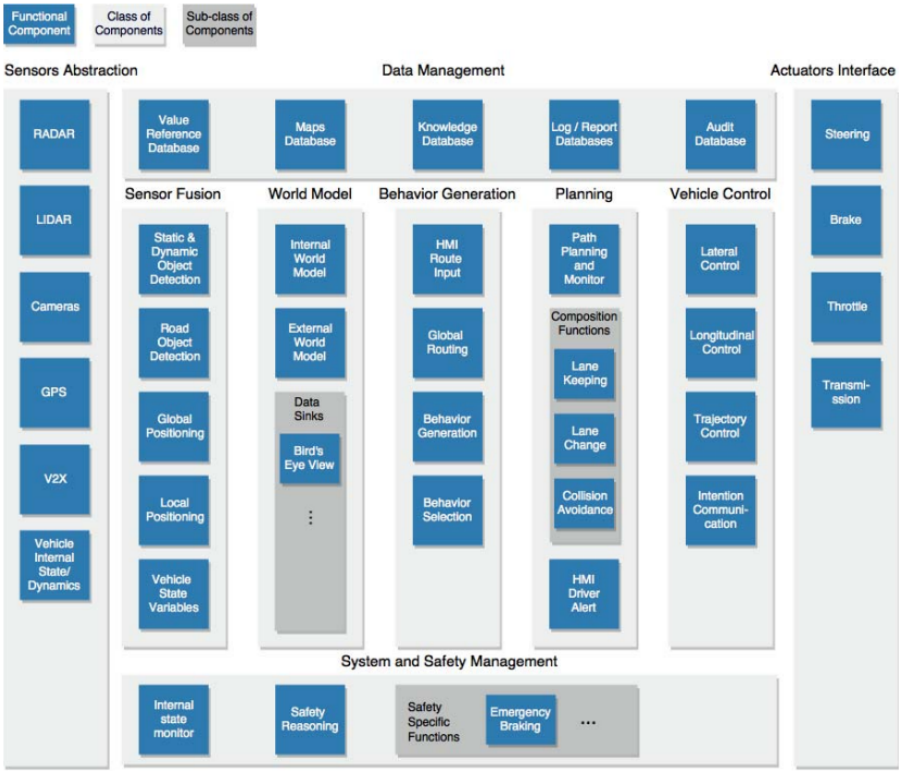

Fig. 1.1: Autonomous driving system by Alexandru Serban [3]

With end-to-end deep learning it can take all those multiple stages and replace them with a single model. The end-to-end model is able to control the autonomous car directly from the pixels provided by the embedded cameras [1]. Based on the basic

network architecture of NVIDIA, this project will apply different variants and tune various hyperparameters.

1.1 Definition of autonomous driving

In contrast to automated vehicles, which only take over part of the control, autonomous cars replace the drivers completely. The autonomous vehicles have no influence from humans and can decide for themselves such as how to behave around the curve (brake, adjust the steering wheel) or how to react to a specific situation (pedestrians or animals appear suddenly).

Many advantages of autonomous driving can be considered. It offers the elderly or people with disabilities the chance to drive by themselves when the public transport or taxis are uncomfortable or too expensive. In addition, when all autonomous cars connect together through IoT (Internet of Things), it can reduce the number of crashes on the roads. The cars will receive signals from each other and adjust their speed by themselves before the crossroads to avoid collision. For this scenario, cyber security plays an important role.

1.2 Levels of autonomous driving

According to current industry standard categorized by the Society of Automotive Engineers (SAE), there are 6 levels of autonomous driving, based on their degree of automation [7]:

- Level 0: no automation
- Level 1: very light automation (cruise control)
- Level 2: automation but requires human attention all the times
- Level 3: can self-drive but require intervention in some conditions
- Level 4: highly autonomous
- Level 5: completely autonomous

Level 1: Driver assistance
At this level the vehicles can perform only a basic assistant task at any given time like braking or accelerating.

Human requirements: all the times
Features: adaptive cruise control, lane keeping

Level 2: Partial driving automation
The vehicles have some form of Advanced Driving Assistance Systems (ADAS). ADAS can take control simultaneously of steering, braking and acceleration systems.

Human requirements: the driver must pay attention and take control in many situations
Features: lane keeping and adaptive cruise control concurrently

Level 3: Conditional driving automation
The vehicles at this level can run mainly by themselves and require human intervention in some extreme environments or failures. In 2022 the Drive Pilot system of Mercedes - Benz is the world's first fully certified level 3 and now can be ordered in Germany as an option in S-class or EQS models [8]. It can navigate the traffic and detect the weather conditions.

Human requirements: the driver must intervene during extreme conditions
Features: Traffic Jam Chauffeur

Level 4: High driving automation
At this level the system can drive on its own and human intervention is not necessary. The limitation of this level is geography. The system can be applied in particular areas and in some weather conditions would also affect these vehicles and would likely disturb their operation. Google's autonomous vehicles are targeted to work at this level

Human requirements: not require
Feature: driverless bus or taxi

Level 5: Fully driving automation
At this highest level of autonomous driving the vehicles are not bounded by geofences and can drive by themselves in all conditions. Currently, there are no real examples of this level outside of science fiction.

1.3 General structure of autonomous driving

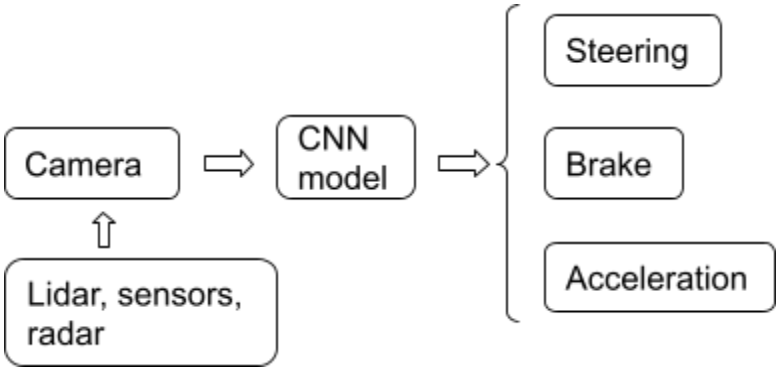

Fig. 1.2: Autonomous driving system

- At the beginning, the autonomous vehicle collects the data through various components (lidar, radar, sensors,...).

- These data are feeded forward a deep learning model. Through repeated training, the deep learning model improves and can independently create appropriate values for the given situation

- The vehicle uses these values to control steering angles, brake pressure, acceleration, etc.

There are different combinations of hardware and software in autonomous cars. Here is just the basic components:

- Camera: is used to collect dataset for deep learning model

- Lidar (light detection and ranging): emits laser signals, up to 150,000 pulses per second, to measure distance. Laser signals with the wavelength between 500 - 600 nm beam at the objects and measure the time it takes for the laser to return to its source. This is why lidar is more accurate and faster than radar

- Radar (radio detection and ranging): measures the long distances to other traffic. The system works almost the same as Lidar but it uses radio waves instead of laser signals. With the wavelength between 3mm to 30cm radar can function at long distances [7].

2. Basic theory

2.1 Image processing

A computer image is a picture composed of an array of elements called pixels. In an 8-bit grayscale image each pixel occupies exactly one byte. This means each pixel has 256 possible numerical values, from 0 to 255, which each correspond to the brightness of one pixel in a picture, with 0 being black and 255 being white [9].

2.1.1 Color space

RGB color space

RGB color space is a combination of 3 colors: red, green, blue. A color image is just an extension of a grayscale image, which is stored as a 3 dimensional array with *height x width x 3* that defines red, green, and blue color components for each individual pixel. The color of each pixel is determined by the combination of the red, green, and blue intensities: $f(x, y) = [r(x, y),\ g(x, y),\ b(x, y)]$. Since the three colors have integer values from 0 to 255, there are a total of 256*256*256 = 16,777,216 combinations or color choices.

Grayscale image

To increase the computational speed the images in this project are converted from RGB color space into grayscale (black and white). This method reduces the number of pixels in images, also the size from 3 to 2 dimensions: *height* x *width* x *1*.

Fig. 2.1: A 8-bit grayscale image [9]

2.1.2 Convolutional operation

Convolution is a mathematical operation which is fundamental to many common image processing operators. Convolution provides a way of multiplying together two arrays of numbers to produce a third array of numbers. This can be used in image processing to implement operators whose output pixel values are simple linear combinations of certain input pixel values [12].

$$y[h,w] = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} k[m,n].\, x[h-m,\ w-n]$$

y: output image matrix

x: input image matrix

k: kernel matrix.

h and *w*: indices the image matrices

m and n: indices of the kernel.

The size of output image [12]:

$$(n_{height} - k_{height} + 1),\ (n_{width} - k_{width} + 1)$$

With input image size $n_{height} \times n_{width}$ and kernel size $k_{height} \times k_{width}$. In *Tensorflow* convolutional layer can be implemented by this function:

```
Conv2D(filters, kernel_size, strides, input_shape, activation)
```

Filter: Integer, the dimensionality of the output space

Kernel_size: A list of 2 integers, specifying the height and width of the 2D convolution window.

Stride: A list of 2 integers, specifying the strides of the convolution along the height and width

Activation: Activation function [10]

Padding

The kernels overlap many times the pixels, which are in the middle of the image. The pixels, which are at the edges or corners, are used much less in the output. That is a waste of information near the edge of the image. The output image also shrinks after the convolutional operation. After hundred convolutional layers the size of the output image becomes very small. Padding technique will solve these problems. The image is added one or many additional borders around the edges and the values of these pixels are zeroes.

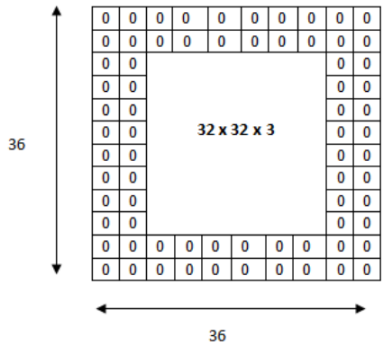

Fig. 2.2: The image with padding, $p = 2$

The size of the input image is 32x32x3 with 2 borders of zeroes around the edges that creates a new size 36x36x3. This technique minimizes the reduction of size in the output layer or it is a way of increasing the size of an image to counteract the fact that stride reduces the size. In general, if a total of p_h rows of padding (roughly half on top and half on bottom) and a total of p_w columns of padding (roughly half on the left and half on the right) are added, the output shape will be [12]:

$$(n_{height} - k_{height} + p_{height} + 1),\ (n_{width} - k_{width} + p_{width} + 1)$$

Stride

When the convolutional operation begins, the window at the upper-left corner of the input image, and then slide it over all locations down and to the right, by default sliding step one element at a time. However, either for computational efficiency or downsample, the window moves more than one element at a time, skipping the

intermediate locations. This is particularly useful if the convolution kernel is large since it captures a large area of the underlying image.

Stride is the number of rows and columns traversed per slide. When the stride for the height is s_h and the stride for the width is s_w , the output shape is [12]:

$$[(n_{height} - k_{height} + p_{height} + s_{height})/s_{height}],\ [(n_{height} - k_{height} + p_{height} + \ s_{width})/\ s_{width}]$$

2.1.3 Filter

Image filtering changes the range (the pixel values) of an image, so the colors of the image are altered without changing the pixel positions. The goal of using filters is to modify or enhance image properties and to extract valuable information from the pictures such as edges, corners, and blobs. There are 2 types of filter: low-pass filter and high-pass filter

- *Low-pass filter:* in the frequency domain it suppresses high frequencies and preserves the low frequencies. It is used for smoothing the image, e.g., mean filter, Gaussian filter.

- *High-pass filter:* opposite to low-pass filter it suppresses low frequencies and preserves the high frequencies. It is used for sharpening the image or detecting the edges, e.g., Sobel filter.

The width and height of the filter must be an odd number, so that the pixels being worked on are always in their center. A Gaussian filter, which is used in this project to smooth and preprocess the images before training, will be described in the following section.

Gaussian filter

The Gaussian smoothing operator is a 2-D convolution operator that acts as low-pass filter and is used to blur images and remove high spatial frequency components from an image (noise). It uses the shape of a Gaussian (bell-shaped). In 2-D Gaussian filter has this formula [11]:

$$G(x, y) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x^2+y^2)}{2\sigma^2}}$$

With σ is the standard deviation of distribution. Gaussian distribution has a bell curve, as a 2-D Gaussian kernel looks like this:

1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	16	26	16	4
1	4	7	4	1

Fig. 2.3: Gaussian kernel [11]

The idea of Gaussian smoothing is: the pixels nearest the center are given more weight than the other far away from the center. Since the image is stored as a collection of discrete pixels, therefore it is necessary to produce a discrete approximation to the Gaussian function before performing the convolution.

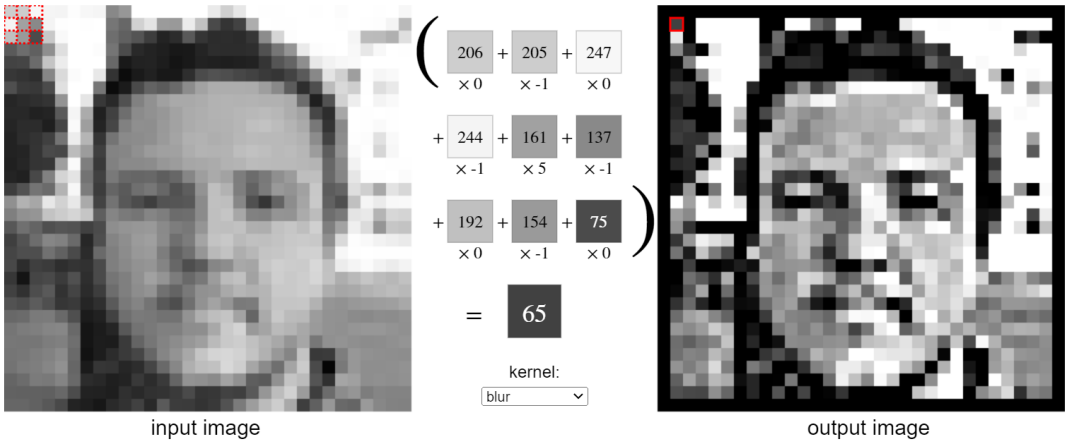

Fig. 2.4: 3x3 Gaussian filter [11]

In figure the red top-left kernel on the left image makes a convolutional operation with the pixels to create the small top-left new pixel on the right image.The numbers in the

squares are the values of pixels and the numbers under the square are values of the Gaussian blur kernel.

In *cv2* library this function is used for Gaussian filter:

```
cv2.GaussianBlur(src, ksize)
```

The input parameters:

src: input image

ksize: the size of the kernel

2.2 Supervised learning and neural network

Machine Learning is a field of research that gives computers the ability to solve the problem by themselves without being explicitly programmed. The goal of machine learning is to create a model and tune its parameters to the given data in order to apply it to the unknown data to generate predictions or recommendations. These processes can be optimized.

Machine Learning can be divided into 4 categories, which suit different problems: supervised learning, unsupervised learning, semi-supervised learning and reinforcement learning. In this project supervised learning is the main focus.

2.2.1 Supervised learning

In supervised learning, the goal is to learn a mapping from inputs *x* to outputs *y*, given a set of labeled input-output pairs $D = (x_i, y_i)\ with\ i = 1 \ldots m \mid m \in N$, where D is the training set, i is the ordinal number of training samples. Each training input x_i and output y_i is a element of vector $\mathbf{x} = [x_1, x_2, \ldots, x_m] \mid x \in R$ and $\mathbf{y} = [y_1, y_2, \ldots, y_m] \mid y \in R$ [12]. In this project **x** is a vector of training images and **y** is a vector of steering values. Each input x_i represents a size of image, therefore it has 3 dimensions: $height$ x $width$ x $channels$.

2.2.2 Neural network

Neural network is a part of supervised learning. Their name and structure are inspired by the human brain and the way biological neurons transmit signals to one another. Artificial neural networks consist of a node layer containing an input layer, one or more hidden layers, and an output layer. Each node is connected to another and has an associated weights w and biases b.

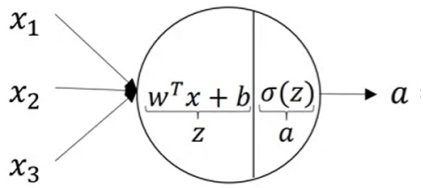

Fig. 2.5: Structure of a perceptron [6]

Like representation of figure [6], there are a vector of input $\mathbf{x^{(k)}} = [x_1, x_2, x_3] \mid x \in R$ with k is kth example in training set and a vector of weight $\mathbf{w^{[i]}} = [w_1^{[i]}, w_2^{[i]}, w_3^{[i]}] \mid w \in R$ with i is ith layer. A linear combination between inputs $\mathbf{x^{(k)}}$, weights $\mathbf{w^{[i]}}$ and biases $\mathbf{b^{[i]}}$:

$$\mathbf{z^{[i]}} = \mathbf{w^{[i]}}.\mathbf{x^{(k)}} + \mathbf{b^{[i]}}$$

Activation function adds non-linearity to the neural network and it is also an output of this layer:

$$\mathbf{a^{[i]}} = \sigma(\mathbf{z^{[i]}})$$

Then this output becomes an input of the next layer and this process repeats until it reaches the output layer.

2.2.3 Vectorization

Instead of looping each example to the whole training set like this pseudo code:

For j = 1 to m:

$\mathbf{z^{[i](j)}} = \mathbf{w^{[i]}}.\mathbf{x^{(k)}} + \mathbf{b^{[i]}}$

$\mathbf{a^{[i](j)}} = \sigma(\mathbf{z^{[i](j)}})$

This implementation costs much computational time O(m) with m is the number of examples in the training dataset. For a better solution, these input vectors can be stacked together in a matrix X = [$\mathbf{x}^{(1)}$, $\mathbf{x}^{(2)}$, $\mathbf{x}^{(3)}$, … $\mathbf{x}^{(m)}$] and implement all of them with:

$$Z^{[i]} = w^{[i]}.X + b^{[i]}$$
$$A^{[i]} = \sigma(Z^{[i]})$$

In *Numpy* library, it supports the function *dot()* to multiply 2 matrices. In these following lines of code the matrices a and b multiply together, each matrix has 1.000.000 elements. The time of implementation between vectorized and unvectorized version is compared.

```
size_trainingset = 1000000
a = np.random.rand(size_trainingset)
b = np.random.rand(size_trainingset)
tic = time.time()
c = np.dot(a, b)
toc = time.time()
print(f"Result: {c}")
print("Vectorized multiplication: ", ((toc-tic)*1000), "ms")
```

```
Result: 250157.01333635935
Vectorized multiplication: 1.966714859008789 ms
```

```
tic = time.time()
for i in range(size_trainingset):
    c += a[i] * b[i]
toc = time.time()
print(f"Result: {c}")
print("Unvectorized version: ", ((toc-tic)*1000), "ms")
```

```
Result: 250157.0133363599
Unvectorized version: 398.0224132537842 ms
```

2 methods have the same result but the time of implementation vectorized version is faster than unvectorized version more than 200 times. In this project all of the training data are vectorized before doing computational things.

2.2.4 Activation function

An activation function is a function that is added into an artificial neural network in order to help the network learn complex patterns in the data. It takes in the output signal from the previous cell and converts it into some form that can be taken as input to the next cell.

Activation functions are useful because they add non-linearities into neural networks, which allows the neural networks to learn powerful operations. There are different reasons to use activation function in neural network:

- The input of activation function is $W.X + b$ where W is weights of cell, X is the input and b is the bias. If the result of this equation is not restricted to a certain limit, it can go extremely high especially in the case of very deep neural networks that have millions of parameters.

- If the activation functions were removed from a feedforward process, the entire network could be refactored to a simple linear operation or matrix transformation on its input and it would no longer be capable of performing complex tasks such as image recognition.

Rectified linear activation (ReLu)
The Relu function is calculated by:

$$f(x) = max(0, x)$$

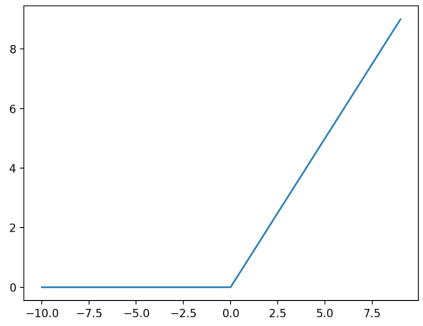

Fig. 2.6: Relu function

If the input x is negative, then $f(x) = 0$, otherwise $f(x) = x$ [6]

Hyperbolic Tangent (Tanh)
Tanh is calculated by:

$$Tanh\, x = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

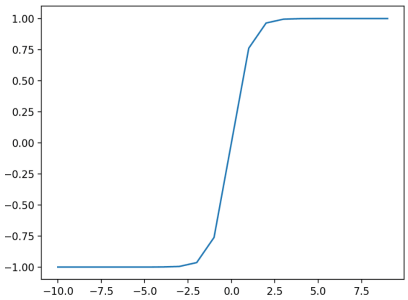

Fig. 2.7: Tanh function

It is very similar to the sigmoid activation function and even has the same S-shape but the range is different from -1 to 1. The larger the input (more positive), the closer the output value will be to 1.0, whereas the smaller the input (more negative), the closer the output will be to -1.0 [6].

2.2.5 Loss function

A loss function in Machine Learning is a measure of how accurately a model can predict. The loss function takes two items as input: predicted outputs from the model and actual outputs of the dataset. The output of the loss function is a measure of how well the model predicts the outcome. A high value for the loss means the model performed very poorly and a low value for the loss means the model performed very well.

The mean squared error (MSE) is the most common and simple loss function. It takes the difference between predicted values and actual values, square it and average it across the whole dataset [13].

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

N: the number of samples in the dataset
y: the target value
$\hat{y}$: the predicted value.

The MSE puts larger weight on these errors due to the squaring part of the function. Large errors are emphasized and have a relatively greater effect on the value of the performance metric. At the same time, the effect of relatively small errors will be even smaller [13]. Besides that MSE is used in this project because it is differentiable, which makes it easy to perform mathematical derivative in comparison to a non-differentiable function like mean absolute error (MAE).

2.2.6 Regularization

In order to avoid overfitting, regularization techniques play an important role in Machine Learning. The main aim of regularization is to reduce over-complexity of Machine Learning models. After this process the model performs well not only on the training dataset, but also has the ability to generalize to the new examples, which it has not seen in training dataset. Dropout regularization is used in this project.

Dropout regularization
Dropout is also the regularization technique, which is used in this project. Dropout refers to the process of ignoring certain neurons in the network during training.

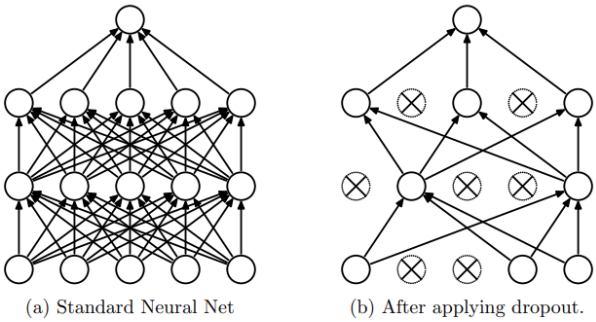

Fig. 2.8: Fully-connected network and dropped out network [20]

The network on the left is a fully-connected network, where all units are activated. On the right, some units in the network are dropped out of the model - the values of their *weights* and *biases* are not considered during the training. When applying dropout to the network, which drops some units in the hidden layers randomly at the time of training. Each time the gradient is updated and the new different units are dropped based on the probability hyperparameter p. In Tensorflow dropout is applied by this function:

```
tf.keras.layers.Dropout(rate)
```

where *rate* is probability that the given units will be dropped.

2.2.7 Backpropagation

The backpropagation algorithm generally consists of passes (epochs) with each pass consisting of two phases [24]:

- *Feedforward phase:* during this phase, the values of training data are transported from input layers through the hidden layers to the output layer. The predicted output $\hat{y}$ are created through *weights* and *activation functions*.

- *Feedbackward phase:* the predicted output $\hat{y}$ and the target output y are compared through loss function MSE. In this phase the process goes back and adjusts *weights* and *biases* for the purpose of reducing *loss function*.

Backpropagation always starts from the output layer and updates *weights* and *biases* in each layer. It adjusts the parameters in neurons throughout the network to get the desired output in the output layer. As described in *2.2.4 Loss function*, *loss function* MSE is used in this project:

$$L = \frac{1}{N}\sum_{i=1}^{N}(y_i - a^{(L)})^2$$

N: the number of samples in the dataset
y: the target value
$a^{(L)}$: activation from the last layer

In order to improve the network, it must be calculated how a change in *weights* and *biases* affect loss function.

$$z^{(L)} = w^{(L)}.a^{(L-1)} + b$$
$$a^{(L)} = \sigma(z^{(L)})$$

$w^{(L)}$: weight parameters of neuron from the last layer
b: bias parameter of neuron
$a^{(L-1)}$: activation function from second last layer

The algorithm runs backward to optimize the loss function MSE. The chain rule is applied, since *weights* and *biases* can not be derived directly from loss function. This calculation is based on the partial derivative of loss function in relation to *weights* and *biases* [24].

$$\frac{\delta L}{\delta w^{(L)}} = \frac{\delta L}{\delta a^{(L)}} \frac{\delta a^{(L)}}{\delta z^{(L)}} \frac{\delta z^{(L)}}{\delta w^{(L)}}$$

$$\frac{\delta L}{\delta b^{(L)}} = \frac{\delta L}{\delta a^{(L)}} \frac{\delta a^{(L)}}{\delta z^{(L)}} \frac{\delta z^{(L)}}{\delta b^{(L)}}$$

$\frac{\delta L}{\delta w^{(L)}}$: the derivative of loss function with respect to weights

$\frac{\delta L}{\delta b^{(L)}}$: the derivative of loss function with respect to biases

$\frac{\delta L}{\delta a^{(L)}}$: the derivative of loss function with respect to activation function

$\frac{\delta a^{(L)}}{\delta z^{(L)}}$: the derivative of activation function with respect to z equation

$\frac{\delta z^{(L)}}{\delta w^{(L)}}$: the derivative of z equation with respect to weights

$\frac{\delta z^{(L)}}{\delta b^{(L)}}$: the derivative of z equation with respect to biases

These equations measure the ratio of how a particular weight and bias affect the loss function, which is optimized. w can not directly found in loss function, that is why a change of w in z equation is considered, because z equation holds w. And the change of z equation in activation function a will affect the change of activation function a in loss function L. The new weights and biases are updated in each iteration. This optimizing process is described in more detail in the next section.

2.2.8 Optimizer

Optimizers are algorithms used to change the attributes of a neural network such as *weights* and *biases* in the layers in order to reduce the losses. The purpose of the optimizer in this project is to minimize the difference between predicted steering values and actual steering values *(MSE)*. There are many different algorithms to optimize the loss function.

2.2.8.1 Gradient Descent (GD)

Gradient Descent is one of the most popular optimization algorithms. It takes the derivatives of *loss function* with respect to the parameters such as *weights* and *biases*. Two or more derivatives of the same function are called *a gradient.* This gradient will descend to the lowest point of the *loss function.* That is why this algorithm is called *Gradient Descent [14].*

$$w_{new} = w_{old} - \alpha \frac{\partial L}{\partial w}$$

$$b_{new} = b_{old} - \alpha \frac{\partial L}{\partial b}$$

w_{new}: updated *weight* in every new step

w_{old}: the old *weight*

b_{new}: updated *bias* in every new step

b_{old}: the old *bias*

α: learning rate. Learning rate is one of the hyperparameters that needs to be tuned. When *the learning rate* is too small, the learning process will be longer, but when the learning rate is too large, it will overshoot the minimum of the loss curve.

$\frac{\partial L}{\partial w}$: partial derivative of *loss function* with respect to *weight*. It is the rate of change of *loss function* to the change in *weight*.

$\frac{\partial L}{\partial b}$: partial derivative of *loss function* with respect to *bias*. It is the rate of change of *loss function* to the change in *bias*

In order to reach the lowest point of the loss function the updated *weights* need to go to the opposite direction of the derivative, that is why there is a minus in this algorithm. But GD is not efficient for big data because it costs a lot of time and computational resources when this algorithm runs through all the samples of a big dataset. Instead of using GD Stochastic Gradient Descent (SGD) is applied for a large dataset.

Momentum

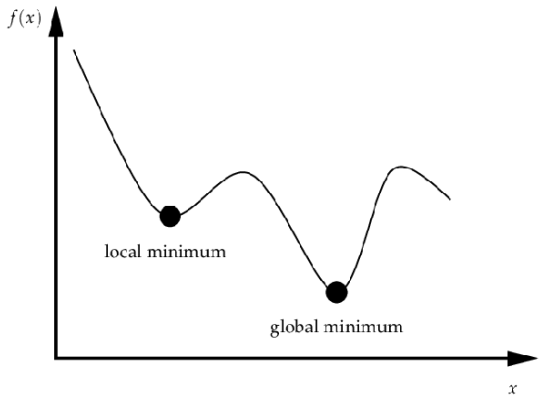

Fig. 2.9: Local and global minimum.

The goal is to reach the global minimum, the lowest point on the function. But sometimes GD is stuck at the local minimum, because at this point the slope $\frac{\partial L}{\partial w}$ is zero that makes the learning process stop. In this case the momentum is added to push the point to continue rolling to the global minimum. This accumulated speed is equivalent to the exponentially weighted average (EWA) of past gradients [16].

$$m_{(w,\,t)} = \beta . m_{(w,\,t-1)} + (1 - \beta) . \frac{\partial L}{\partial w_t},\ with\ 0 \leq \beta \leq 1$$

$$m_{(b,\,t)} = \beta . m_{(b,\,t-1)} + (1 - \beta) . \frac{\partial L}{\partial b_t},\ with\ 0 \leq \beta \leq 1$$

$m_{(w,\,t)}$: current exponentially weighted average gradient of the *weight*

$m_{(w,\,t-1)}$: previously exponentially weighted average gradient of the *weight* until the time $t - 1$

$m_{(b,\ t)}$: current exponentially weighted average gradient of the *bias*

$m_{(b,\ t-1)}$: previously exponentially weighted average gradient of the *bias* until the time $t\ -\ 1$

$\frac{\partial L}{\partial w_t}$: current gradient of *weight* at time t

$\frac{\partial L}{\partial b_t}$: current gradient of *bias* at time t

β: weightage. It is given more weight to the current gradient than to the previously accumulated gradient.

When the point reaches the local minimum in *figure 2.9* , the slope $\frac{\partial L}{\partial w}$ becomes zero and the equation of current momentum is $m_{(w,\ t)} = \beta . m_{(w,\ t-1)}$, not equal to zero like GD. The new weights and biases are not equal to zero and continues to update:

$$w_t = w_{t-1} - \alpha . m_{(w,\ t)}$$

$$b_t = b_{t-1} - \alpha . m_{(b,\ t)}$$

w_t: current weight

w_{t-1}: previous weight

b_t: current bias

b_{t-1}: previous bias

α: learning rate

2.2.8.2 Root Mean Squared Propagation (RMSProp)

The other problem of GD is the learning rate is constant during the learning process. The optimal scenario to accelerate the optimization processes is the learning rate is updated after each iteration [17]. The gradient $\frac{\partial L}{\partial w}$ and $\frac{\partial L}{\partial b}$ changes during the training process and this rate of change is applied to update learning rate [17]:

$$v_{(w,\ t)} = \beta . v_{(w,\ t-1)} + (1\ -\ \beta) . \left(\frac{\partial L}{\partial w_t}\right)^2,\ with\ 0 \leq \beta \leq 1$$

$$v_{(b,\ t)} = \beta . v_{(b,\ t-1)} + (1\ -\ \beta) . \left(\frac{\partial L}{\partial b_t}\right)^2,\ with\ 0 \leq \beta \leq 1$$

$v_{(w,\,t)}$: current exponentially weighted average squared gradient of the *weight*. The more recent $\frac{\partial L}{\partial w_t}$ is, the more impact it has on v_w

$v_{(b,\,t)}$: current exponentially weighted average squared gradient of the *bias*

β: weightage. It can be self defined, but in paper Hinton advised β should be 0.9 [17].

The *weights* and *biases* are updated after each iterations:

$$w_t = w_{t-1} - \frac{\alpha}{\sqrt{v_w + \epsilon}} \frac{\partial L}{\partial w}$$

$$b_t = b_{t-1} - \frac{\alpha}{\sqrt{v_w + \epsilon}} \frac{\partial L}{\partial b}$$

ε: error term. It is added to v so that the denominator does not become zero, as default $\varepsilon = 10^{-8}$

α: learning rate

Through the process the sum of squared gradients v_w and v_b always increase and make the learning rate α decrease. Over the time with iterations the learning rate α tend to zero and makes $w_t \approx w_{t-1}$ and $b_t \approx b_{t-1}$ that leads to slower convergence.

2.2.8.3 Adaptive moment estimation (Adam)

Adam is the most popular and effective of all the optimizers. It is a combination between GD with momentum and RMSProp [18]:

$$m_t = \beta_1 . m_{t-1} + (1 - \beta_1) . \frac{\partial L}{\partial w_t}$$

$$v_t = \beta_2 . v_{t-1} + (1 - \beta_2) . \left(\frac{\partial L}{\partial w_t}\right)^2$$

m_t: current exponentially weighted average gradient of the *weight*

β_1: parameter of Momentum, as default $\beta_1 = 0.9$ [18].

v_t: current exponentially weighted average squared gradient of the *weight*.

β_2: parameter of RMSProp, as default $\beta_2 = 0.999$ [18].

2.3 Convolutional neural network

Convolutional neural network (CNN) is a sequence of layers, which reduce the images into easier form to process without losing features for getting a good prediction. It is important to design a model architecture, which is not only good at learning features but also scalable in big data. There are three main types of layer to build CNN architecture: Convolutional layer, Pooling layer and Fully-connected layer [31].

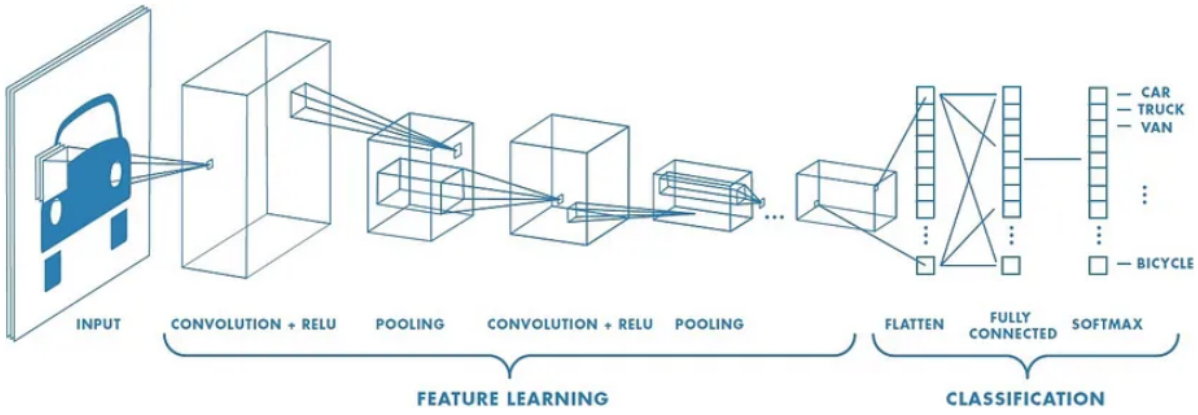

Fig. 2.10: Three main layers in CNN model [31]

2.3.1 Convolutional layer

This layer applies convolutional operation, described in *2.1.2 Convolutional operation*, to the input images. After this layer the image size is decreased and it brings all the information in the field into a single pixel. The purpose is size reduction as well as extraction of high-level features such as edges of the input images. The deep dimension of output after this layer is depend on the number of applied filters [$height$ x $width$ x $filters$] *[31].*

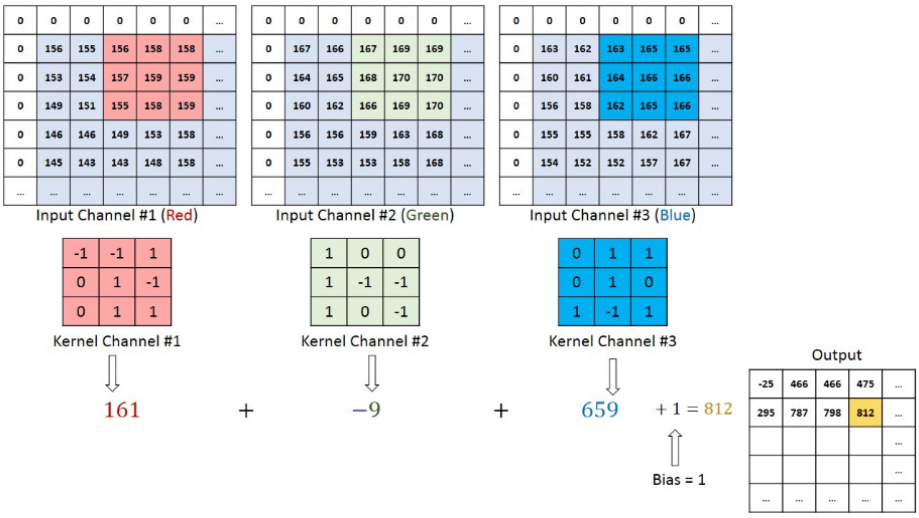

Fig. 2.11: Convolutional operation on three channels images RGB with 3x3x3 kernel

In this project the color space of input images are converted from RGB to Grayscale color space in order to decrease the dimension and speed up computational time, described in *2.1.1 Color space.* In fig. 2.13 is the convolution operation between the grayscale image 5x5x1 and the kernel 3x3x1

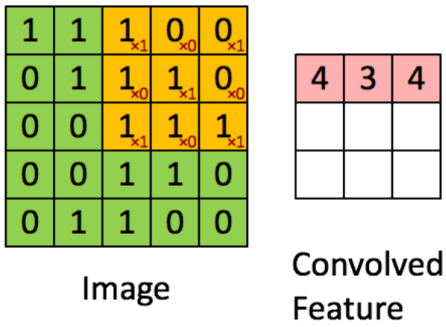

Fig. 2.12: Convolutional operation for grayscale image

2.3.2 Pooling layer

Similar to the purpose of the Convolutional layer, the Pooling layer is responsible for downsampling the size of convolved features. This layer reduces the amount of parameters and computation in the network, therefore it also controls overfitting. There are two types of pooling: Max pooling and Average pooling.

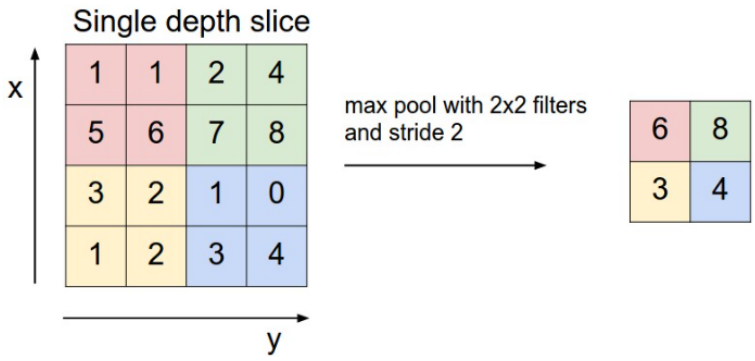

Fig. 2.13: Max pooling with 2x2 filter and stride 2 [21]

Max pooling takes max values from the portion of images covered by the kernel, while the average pooling returns the average of all the values from the portion of images covered by the kernel. In figure. Max pooling decreases the image size from 4x4 to 2x2 with 2x2 filter and stride 2.

2.3.3 Fully-connected layer

In a Fully-connected layer every single neuron in every layer connects to each other (a multilayer perceptron). Like described in *2.2.2 Neural network,* it is a combination of equation between input values and parameters of the network .The purpose of this layer is to tune the weights w and biases b parameters in each layer by using *backpropagation* and *optimizers*.

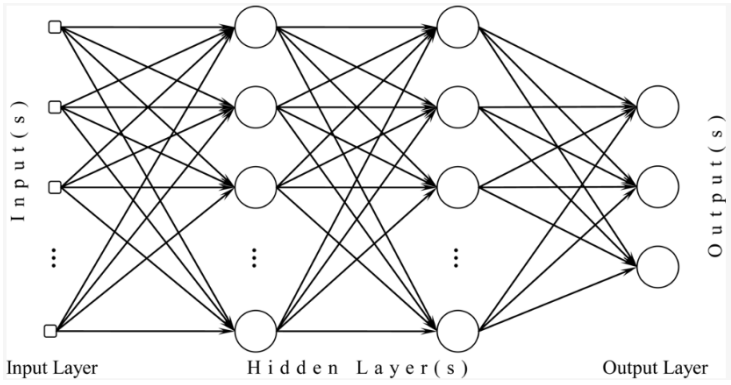

Fig. 2.14: Fully-connected layer with two hidden layers [21]

2.4 Regression

Regression is a method to measure the relationship between dependent variables (regressors) and independent variables (target). It captures the correlation between variables observed in the data set and determines whether those correlations are statistically significant or not. There are different variants of regression: simple regression, multivariate regression, polynomial regression, …

Simple regression:
A model for simple regression with n observations (x_1, y_1), (x_2, y_2), … (x_n, y_n) are pairs of regressors and target [23]:

$$y_i = \beta_0 + \beta_1 . x_i + \epsilon_i$$

y_i: ith target variable, $i = 0, \ldots . n$

β: coefficients, where β_0 is the constant term of the model

x_i: ith regressor, $i = 0, \ldots . n$

ϵ_i: ith noise term or random error

Multivariate regression:
A multivariate regression is an extension of simple regression with more than one independent variables:

$$Y = X\beta + \epsilon$$

Y: response vector, $n \times 1$ dimension
X: regressor vector with m regressors, $n \times m$ dimension
β: coefficient vector, $n \times 1$ dimension
ϵ: random error vector, $n \times 1$ dimension

Or in detail:

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \ldots \\ y_n \end{pmatrix} = \begin{pmatrix} 1 & x_{11} & x_{12} & \ldots & x_{1m} \\ 1 & x_{21} & x_{22} & \ldots & x_{2m} \\ 1 & x_{31} & x_{32} & \ldots & x_{3m} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 1 & x_{n1} & x_{n2} & \ldots & x_{nm} \end{pmatrix} \begin{pmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \\ \ldots \\ \beta_n \end{pmatrix} + \begin{pmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \ldots \\ \epsilon_n \end{pmatrix}$$

As described in *2.2.5 Loss functio*n, mean squared error is also used for the regression model as the evaluator

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

N: the number of samples in the dataset
y: the target value
$\hat{y}$: the predicted value.

2.5 Gaussian kernel smoothing

Smoothing is a technique to remove noise or certain frequencies in time series which improves data quality. The main idea of smoothing is averaging the data points with their neighbors. Gaussian kernel, which puts different weights for averaging in the sliding window, is chosen in this project. The shape of the kernel is Gaussian distribution or bell curve.

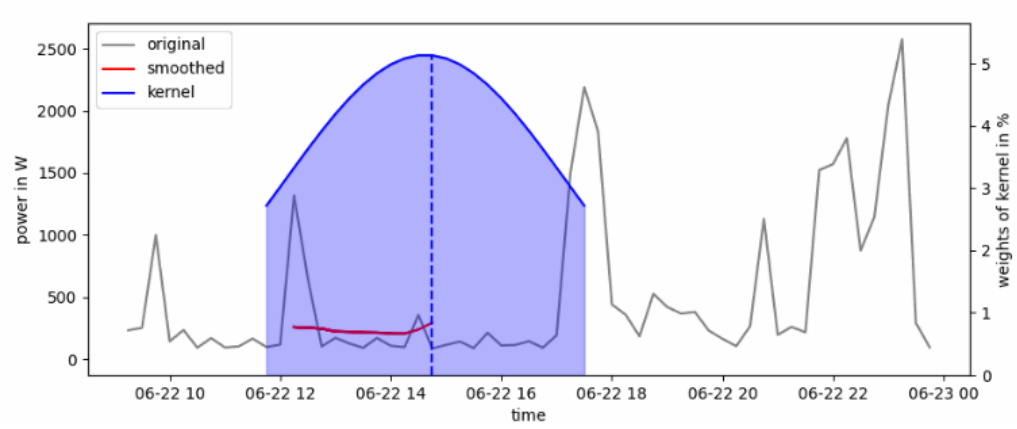

Fig. 2.15: Gaussian kernel smoothing [30]

Gaussian kernel smoothing uses a weighted average of the data points based on their distance from the center point. The weights are determined by a Gaussian function,which assigns higher weights to points closer to the mean value of kernel and lower weights to points further. The closer points have a greater influence on the smoothed data point. Gaussian kernel formula:

$$K(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

x: input data point
μ: mean value of kernel
σ: standard deviation

When Gaussian is used for smoothing, it is necessary to define Full Width at Half Maximum (FWHM). The FWHM is the width of the kernel at half of the maximum the height of the Gaussian. The standard deviation is depend on the width of sliding window or FWHM:

$$\sigma = \frac{FWHM}{\sqrt{8ln(2)}}$$

3. Concept

3.1 Main concept

This project will mainly focus on optimizing the convolutional neural network (CNN) for autonomous driving by choosing the closest exact values of hyperparameters. The images are taken from the simulator and put into the CNN model to predict steering values. Different combinations of hyperparameters create different loss values. The loss is the difference between actual steering values and predicted steering values. Instead of randomly tuning neural networks, this project will optimize the tuning process more systematically and logically by multivariate quadratic regression. The regression is built to determine relationships between hyperparameters and the loss.

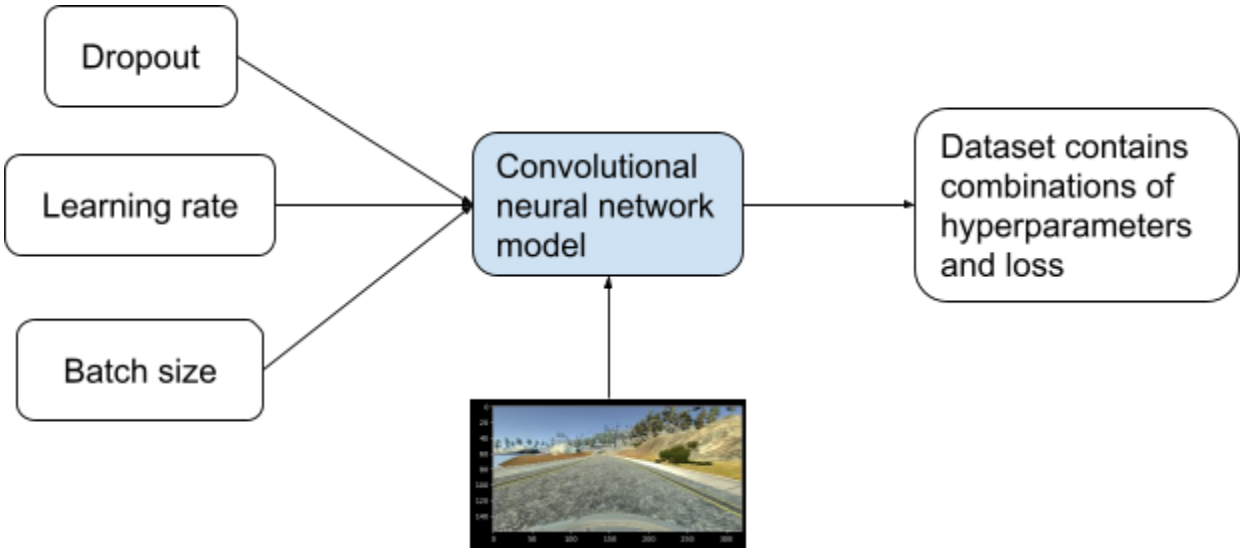

Fig. 3.1: Running the experiments to generate the dataset for regression model

Firstly in order to generate a dataset for a regression model, a number of experiments are run by CNN model, which contain the range of hyperparameter values (e.g, dropout, learning rate, batch size, *Tab. 5.1: The values of hyperparameters for data collecting*). After running the experiments, a dataset is created (*Tab. 5.2: Dataset of hyperparameters*), which includes hyperparameter values and loss. Regression model takes hyperparameters and loss from CNN as input. This model is built based on the hyperparameters as regressors and loss as target variable (more detail in *3.6 Hyperparameter tuning using multivariate quadratic regression*).

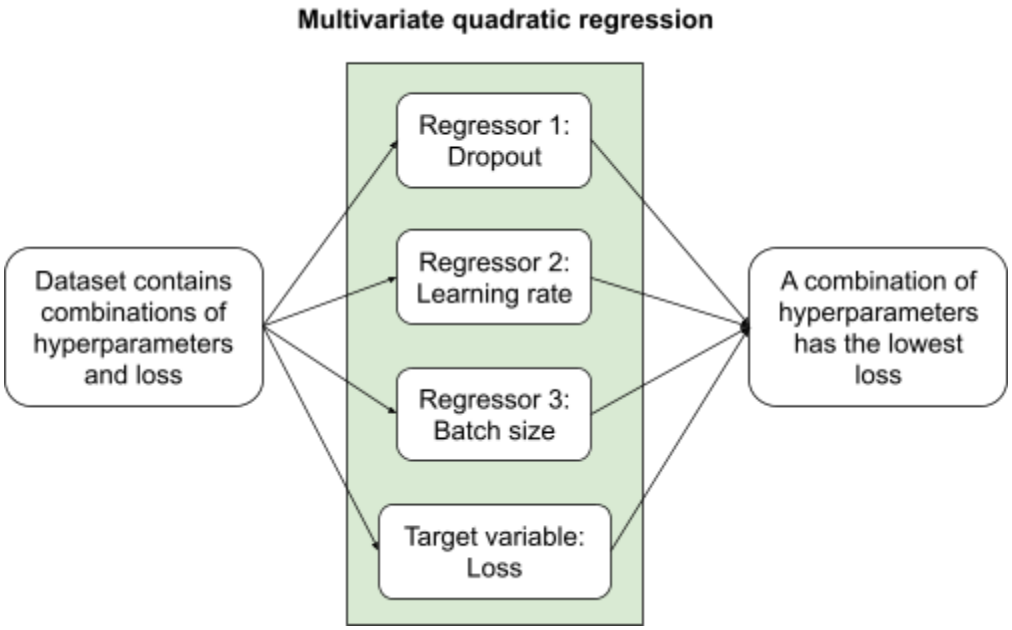

Fig. 3.2: Optimizing process with regression

After defining the regression model, this model is derived and the system of equations is solved with three variables: batch size, dropout and learning rate in order to find out which combination has the lowest loss (more detail in *3.6 Hyperparameter tuning using multivariate quadratic regression*). The appropriate strategy for tuning is chosen to find the best performance model. After training, the CNN model takes images from the camera as input to predict steering values, which are converted into steering angles in the Udacity simulator program.

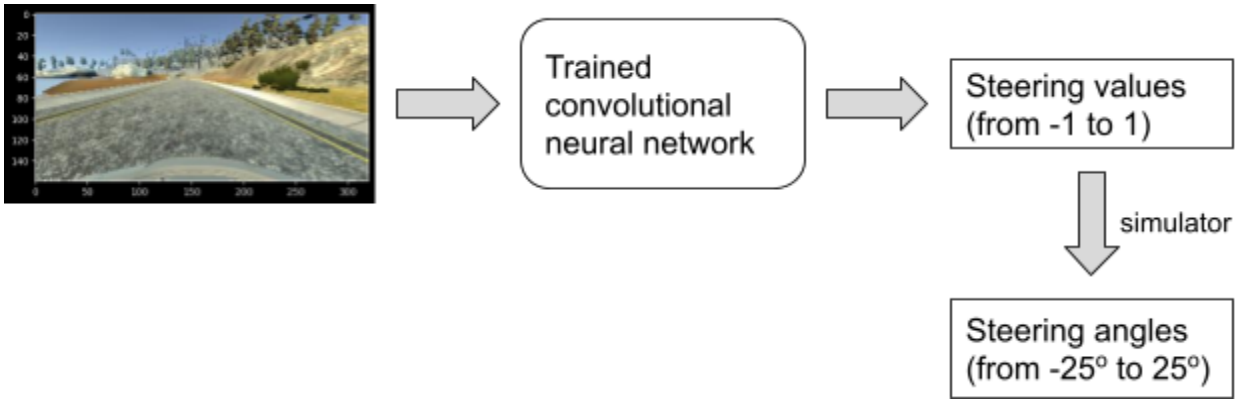

Fig. 3.3: The workflow of the CNN model after training

3.2 Component diagram of the program

Component diagram of the program

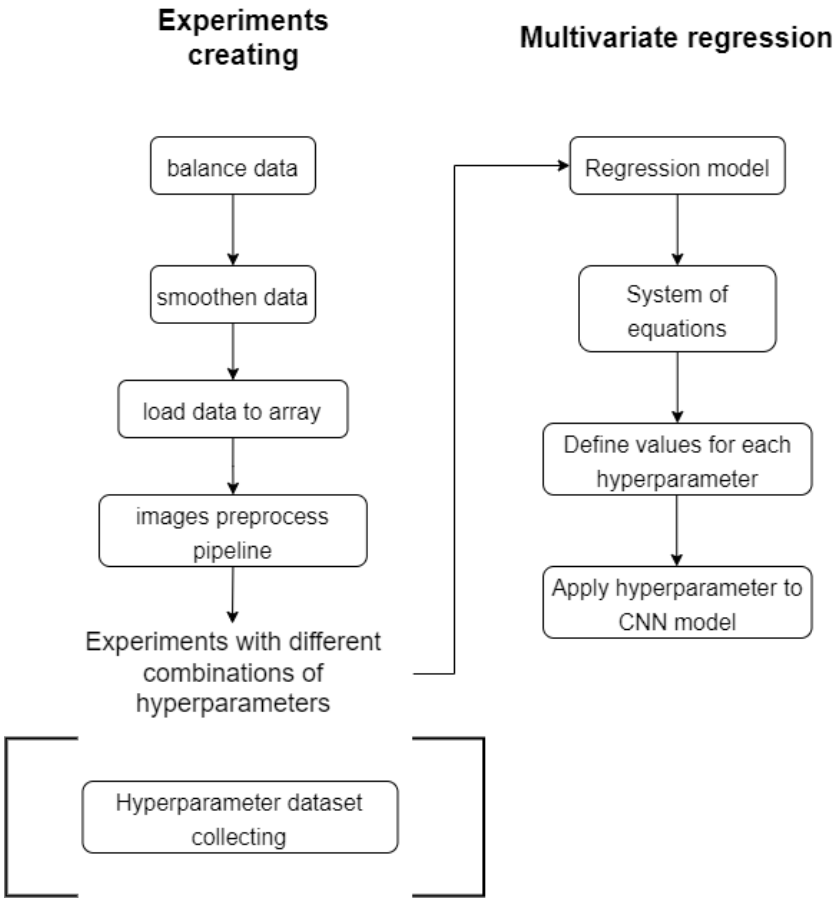

Fig. 3.4: The component diagram of the project

For easier to manage the source code and debug the whole program is divided into two small components: *experiments creating, multivariate regression*. In each component consists of different functions for different purposes:

- *Component experiments creating:* consists of different functions, which preprocesses the data and runs the experiments with different combinations of hyperparameters automatically in order to create a dataset for regression model.
    - Function *balance data:* the dataset needs to be balanced because the steering angles around zero are much higher than other values (more detail in *3.3.2 Data preprocessing - balancing steering data*)

    - Function *smoothen data*: because of limitations of simulator program, the steering data need to be smoothened (more detail in *3.4.2 Data preprocessing - smoothing steering data*)

    - Function *load data to array:* in order to increase computational speed a single datapoint needs to be loaded into *Numpy array* (more detail in *2.2.3 Vectorization*)

    - Function *images preprocess pipeline:* in this pipeline the images are preprocessed and augmented before training (more detail in *4.2 Images augmentation and preprocessing*)

    - Function *hyperparameter dataset collecting:* after running experiments, a dataset of hyperparameter and loss is collected and fed into a regression model in order to optimize.

- *Component multivariate regression:* builds the multivariate quadratic regression model and solves a system of equations to find the best combination of hyperparameters for the CNN models by taking a dataset from "hyperparameters data collecting" in component *experiments creating* as input.

3.3 Advantages and disadvantages of software tools

The advantages and disadvantages of software tools and programming language are discussed in this section. The software tools, which are used in this project, include: Python, Udacity simulator, Tensorflow.

Python

Advantages:

- Python is the most popular and simple high-level programming language. That helps the users debug easier and improves code quality

- Moreover a wide range of open-source machine learning libraries and frameworks are built based on python language such as: Tensorflow, Pytorch, Scikit-Learn.

- Python is a flexible language that can be built and run on different operating systems like: Linux, Windows, Mac.

Disadvantages:

- Python is an interpreted language that makes Python have slower runtime than compiled languages like C++ or Java. This can be a significant disadvantage when working with a large dataset.

Udacity simulator

Advantages:

- Udacity simulator provides a realistic simulation environment for driving conditions. This allows developers to test machine learning models for autonomous driving in a safe and controlled environment.

- This simulator is an open-source platform that is freely available to the public.

- In comparison with other simulators, Udacity simulator is more lightweight and without needing a GPU to install.

Disadvantages:

- Because of its lightweight, Udacity simulator can not capture all of the complexities in real world driving conditions. That leads to algorithms that perform well in simulation but fail in the real world.

- The quality of images in this simulator is not good enough to use for the training model in the real world. Because of the limited hardware resource Udacity simulator is still used in this project.

- The simulator does not allow the car to interact with other vehicles, which does not reflect the behavior of the real world drivers.

Tensorflow

Advantages:

- Tensorflow is an open-source machine learning framework developed by Google. That means it is free to use and easy to customize the code.

- Tensorflow is scalable and can handle small and large datasets. It is a good choice for large scale machine learning projects.

- It is also a flexible framework which supports a wide range of use: images processing, natural language processing and time series analysis. It is specially suitable for deep learning tasks, such as convolutional neural networks.

Disadvantages:

- This framework has limited interoperability, which means it can be challenging to integrate Tensorflow with other softwares or frameworks.

- Tensorflow requires powerful hardware to train machine learning models efficiently, especially using GPU for training. That is why cloud-based resource Kaggle Notebook is used in this project to speed up computational power (more detail in *4.1.4 Kaggle*)

3.4 Dataset

3.4.1 Data collecting

The data is collected from *Udacity's self - driving car simulator*. In this software the Training mode is chosen and the users have to drive the car manually 10-12 laps by themselves. While driving the necessary data is recorded, e.g. images, speed, steering angle and saved in Excel file. For the purpose of a simpler training process, the images of the central camera are training input and steering values are labeled as training output.

Fig. 3.5: A view in Udacity software

The dataset collected for training comprises 5855 rows (equivalent 5855 training images) and 5 columns: images of center camera, steering angle, throttle, brake and speed. To simplify and speed up the training process the throttle, brake and speed columns are reduced.

	Center	Steering	Throttle	Brake	Speed
6	center_2022_11_11_17_45_15_222.jpg	0.0	0.000000	0	3.212186
8	center_2022_11_11_17_45_15_429.jpg	0.0	0.000000	0	3.141726
11	center_2022_11_11_17_45_15_775.jpg	0.0	0.000000	0	3.035858
13	center_2022_11_11_17_45_16_016.jpg	0.0	0.722739	0	4.252429
19	center_2022_11_11_17_45_16_764.jpg	0.0	1.000000	0	12.707510
...	...	...	...	...	...
11753	center_2022_11_12_20_21_12_289.jpg	0.0	0.000000	0	17.993510
11754	center_2022_11_12_20_21_12_364.jpg	0.0	0.000000	0	17.851470
11755	center_2022_11_12_20_21_12_436.jpg	0.0	0.000000	0	17.710550
11757	center_2022_11_12_20_21_12_580.jpg	0.0	0.000000	0	16.932590
11760	center_2022_11_12_20_21_12_859.jpg	0.0	0.000000	0	16.445130

5855 rows × 5 columns

Fig. 3.6: Dataset for training.

3.4.2 Data preprocessing

Data preprocessing is an integral step in Machine Learning as the quality of data and the useful information that can be derived from it directly. That affects the model to learn and it is important to preprocess data before feeding it into the model.

Balancing steering data

The range of steering value is from -1 to 1, corresponding with -25^{O} to 25^{O} angle degrees in the simulator. When the car turns all left, steering value is -1 and turns all right, steering value is 1 and when it goes straight, the steering angle is zero. Most of the time the car goes straight that is why the distribution of steering angle around zero is extremely higher than other values. That will affect the learning process.

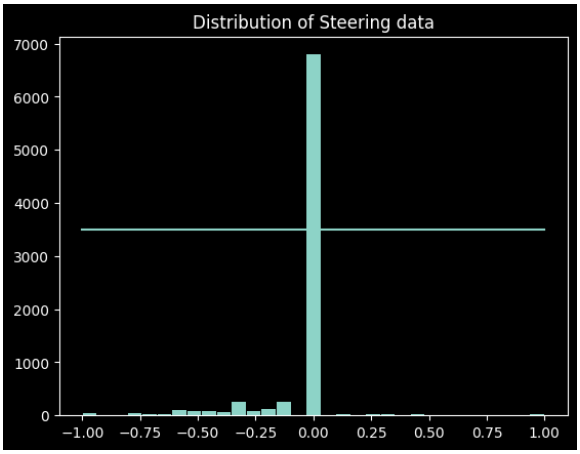

Fig. 3.7: The distribution of steering data

The distribution of value around zero is almost 7000 values and it will be dropped to equal the horizontal linear 3500

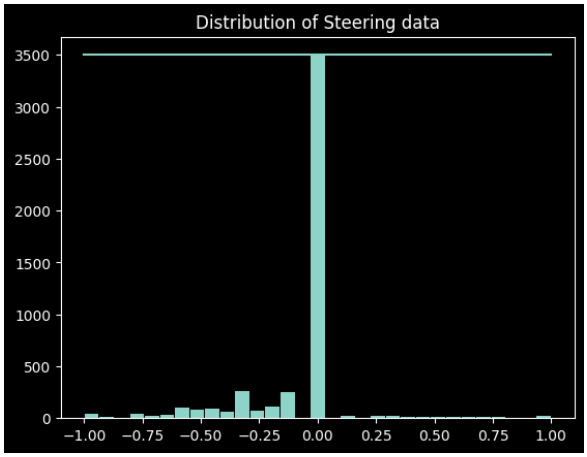

Fig. 3.8: The distribution of steering data after balancing

After dropping unnecessary data the original form of distribution is still the same, with most of the time steering angle is around zero. In this testing track, the distribution of data turning left is more than turning right.

Smoothing steering data

Because of the limitation in the simulation program the steering values need to be smoothened. In the simulator program a keyboard is used when the user manually turns the car right or left, which makes the steering values swing between 0 and 1 or between 0 and -1. And the step of steering values is 0.1, equal to 2.5°. These factors do not reflect the behavior of drivers in the real world and make training data inaccurate. Therefore the steering values need to be smoothened by Gaussian kernel smoothing method (window size = 70) (described in *2.5 Gaussian kernel smoothing*)

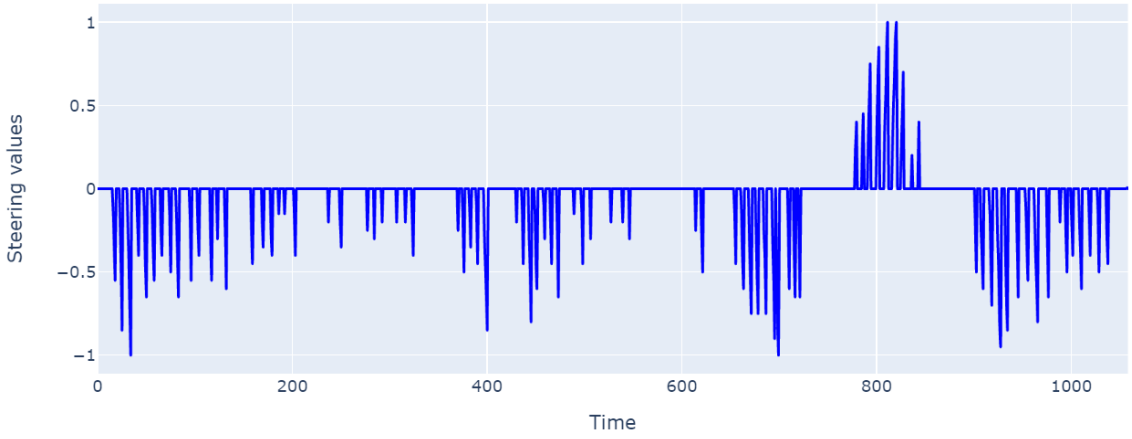

Fig. 3.9: Original steering values

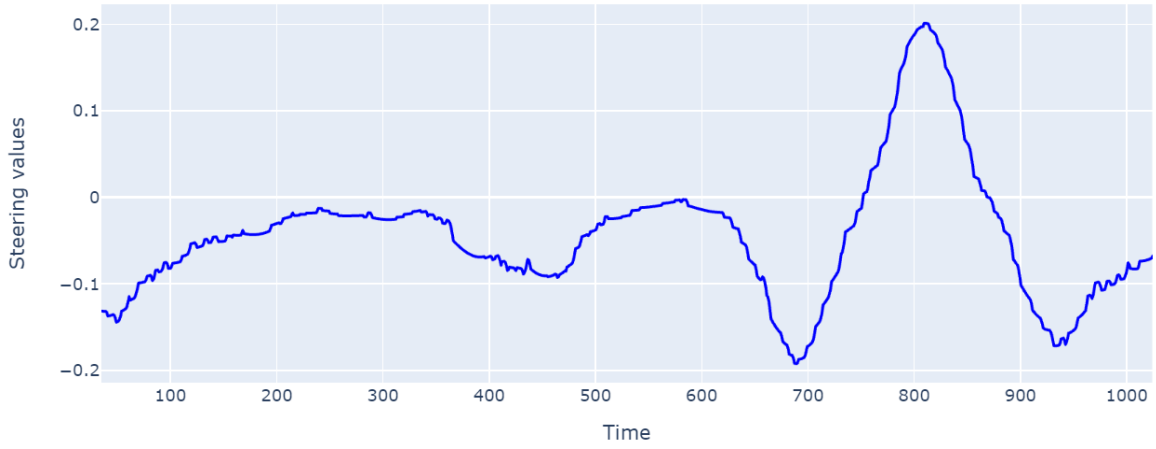

Fig. 3.10: Steering values after smoothing

Image augmentation
Deep learning networks need a large amount of training data to achieve good performance. To build a powerful training dataset with a little training data, image augmentation is considered as a boost to the performance of deep learning models. Image augmentation creates training images by different ways of processing or combination of multiple processing: shift, zoom, adjust brightness and flip the images (more detail in *4.2 Images augmentation and preprocessing*).

Images preprocessing
The purpose of this step is to standardize the images and improve their quality in order to make the training process faster and better. By preprocessing undesired distortions are suppressed and necessary features of images are enhanced to fit the particular applications (more detail in *4.2 Images augmentation and preprocessing*).

3.5 Model architecture

The neuron network architecture, which is used in this project, is based on NVIDIA's research. This model consists of five convolutional layers and four fully-connected layers. The convolutional layers have two types of filters: 3x3 and 5x5, which extract features of images. These features are flattened in a 1D array before being fed into a fully-connected layer. The last neuron of this model is just a perceptron to predict the steering value.

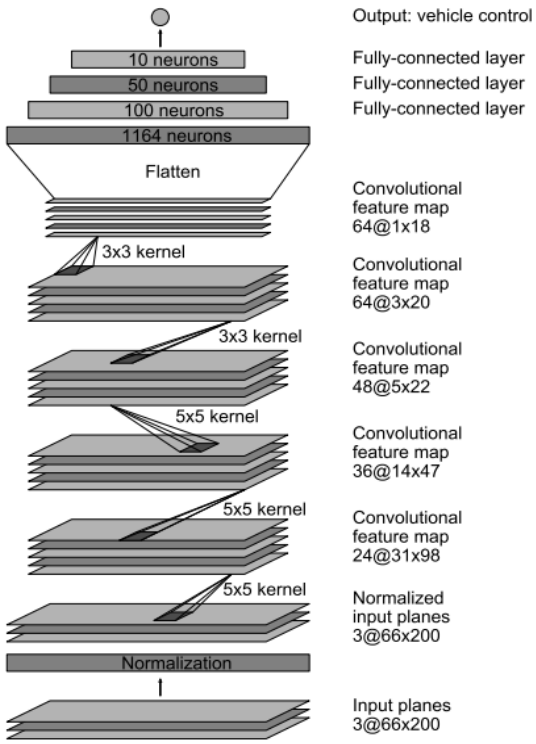

Fig 3.11: Nvidia model architecture [1]

3.6 Hyperparameter tuning using multivariate quadratic regression

After building the CNN model, hyperparameter tuning is an important step to define which values have a high effect on loss. A good tuning process can find the importance of hyperparameters and improve the model performance.

In this project the multivariate quadratic regression is used to determine the hyperparameters in CNN model. The average of training loss and validation loss of CNN model is defined as the target variable and features are the hyperparameters. In this case the variables of the multivariate regression are dropout, learning rate, batch size. The regression model:

$$y_i = f(a,b,c) = \beta_1 + \beta_2 . a_i + \beta_3 . b_i + \beta_4 . c_i + \beta_5 . a_i^2 + \beta_6 . b_i^2 + \beta_7 . c_i^2 + \beta_8 . a_i . b_i + \beta_9 . a_i . c_i + \beta_{10} . b_i . c_i + \epsilon_i \quad (3.5.1)$$

y_i: ith average loss between training loss and validation loss, $i = 0, \ldots n$

a_i: ith batch size, $i = 0, \ldots n$

b_i: ith learning rate, $i = 0, \ldots n$

c_i: ith dropout, $i = 0, \ldots n$

β: coefficients

ϵ_i: ith noise term or random error

With the conditions:

- $\beta_5 > 0$
- $\beta_6 > 0$
- $\beta_7 > 0$
- $4.\beta_5.\beta_6 - \beta_8^2 > 0$
- $4.\beta_5.\beta_7 - \beta_9^2 > 0$
- $4.\beta_6.\beta_7 - \beta_{10}^2 > 0$

After defining the coefficient $\hat{\beta}$, the regression model is derived with respect to batch size, learning rate and dropout. The system of equations is solved with three variables: batch size, learning rate and dropout. The derivative of equation (3.5.1) with respect to variables a, b, c:

$$\frac{\delta f(a,b,c)}{\delta a} = \widehat{\beta_2} + 2.a.\widehat{\beta_5} + b.\widehat{\beta_8} + c.\widehat{\beta_9} = 0$$

$$\frac{\delta f(a,b,c)}{\delta b} = \widehat{\beta_3} + a.\widehat{\beta_8} + 2.b.\widehat{\beta_6} + c.\widehat{\beta_{10}} = 0$$

$$\frac{\delta f(a,b,c)}{\delta c} = \widehat{\beta_4} + a.\widehat{\beta_9} + b.\widehat{\beta_{10}} + 2.c.\widehat{\beta_7} = 0$$

a_i: ith batch size, $i = 0, \ldots n$

$\frac{\delta f(a,b,c)}{\delta a}$: the derivative of regression model with respect to batch size

b_i: ith learning rate, $i = 0, \ldots n$

$\frac{\delta f(a,b,c)}{\delta b}$: the derivative of regression model with respect to learning rate

c_i: ith dropout, $i = 0, \ldots n$

$\frac{\delta f(a,b,c)}{\delta c}$: the derivative of regression model with respect to dropout

4. Implementation

4.1 Software

4.1.1 Udacity's self - driving car simulator

Udacity is an educational platform, they offer many courses in different fields, specially in artificial intelligence. They developed a self - driving car simulator as an open - source software to teach their students in course Self - driving car engineering [25].

This software is used for data collecting and testing autonomously driving. There are two modes: Training mode and Autonomous mode. In training mode the car is driven manually to record driving behavior and these records are used as training data. In autonomous mode the deep learning mode connects to the simulator to test how well the model can perform. At this time the simulator plays a role as a server, from which the deep learning model receives the streaming images and sends back steering angle values.

Fig. 4.1: Udacity simulator program

For more informations: https://github.com/udacity/self-driving-car-sim

4.1.2 Tensorflow and Keras

Tensorflow is an open-source machine learning library and it is also an end-to-end machine learning platform. It provides different tools for four stages when building a machine learning model: prepare and preprocess data, build models, deploy models, implement MLOps [26]. In this project Tensorflow is used only for building models. Keras is the high-level API for Tensorflow, which is integrated into Tensorflow to speed up the building deep learning process.

Tensorflow is used for *build_network()* function. The input parameters of these functions are the hyperparameters of the deep learning model. The following code builds a neural network based on the model in *3.4 Model architecture*. Dropout layers (described in *2.2.6 Regularization*) are also added between every layer to drop out unnecessary neurons in order to prevent overfitting.

Code snippet 4.1: function for creating neural network

```
from keras import Sequential
from keras.layers import Conv2D, Flatten, Dense, Dropout

def build_network(activation, optimizer, dropout):
    model = Sequential()
    model.add(Conv2D(24, (5,5), (1,1), input_shape=(70, 200, 3),
activation=activation)) # (filter, kernel, stride, input shape)
    model.add(Conv2D(36, (5,5), (1,1), activation=activation))
    model.add(Conv2D(48, (5,5), (2,2), activation=activation))
    model.add(Conv2D(64, (3,3), (2,2), activation=activation))
    model.add(Conv2D(64, (3,3), (2,2), activation=activation))

    model.add(Flatten())
    model.add(Dense(100, activation=activation))
    model.add(Dropout(dropout))
    model.add(Dense(50, activation=activation))
    model.add(Dropout(dropout))
    model.add(Dense(10, activation=activation))
    model.add(Dropout(dropout))
    model.add(Dense(1, activation='tanh'))

    model.compile(loss='mse', optimizer=optimizer, metrics=['acc'])
    return model
```

Firstly the model needs to be defined with the function `Sequential()`.The function `Conv2D(24, (5,5), (1,1), input_shape=(70, 100, 1)` creates a convolutional layer with 24 filters, 5x5 kernel, stride 1. The input shape of this layer is the shape of preprocessed images [$height$ x $width$ x $channels$]. The input shape of the next layer is automatically defined, depending on the previous layer. After the convolutional layer the dimension of data is three dimensions and it needs to be converted into a one dimension array before feeding into a fully-connected layer. `Flatten()` function flattens the output from the convolutional layer to create a single feature vector. The last layer is four fully-connected layers with one single neuron at the last layer that predicts steering angle. The activation function of the last neuron is Tanh because the steering angle is between -1 and 1

The input parameters:
activation: activation function
optimizer: pass the optimizer from build_optimizer()
dropout: the fraction of units to drop, between 0 - 1.

4.1.3 Version control

Version control is a software tool that helps track and manage changes in source code. It keeps track of every modification to the code. If something wrong happens, the developers can turn back and compare the earlier versions in order to fix the bugs or minimize the disruption. In this project two version control softwares are used: *Git* and *Weights & Biases.*

Git and Github
Git is an open-source code version control system used to track the changes in source code. It is generally used for management of source code in software development. Git provides a few of the following features:

- “Branch” is a terminology in Git, which is used to maintain the changes until they are ready. A new feature of the program can be altered on the “branch” while the “main branch” stays the same. After the work on “branch” is done, it can be merged into the “main branch”.

- It allows multiple developers to work together and track “who changed what and when”.

- Distributed version control system provides each developer with their own local repository, where they can check a full history of commits.

Github is a cloud-based repository, where the developers can store, track, manage, control the changes in the code. It is a distributed version control platform, where users can collaborate or share the code together. The source code of this project is stored in this remote repository: github.com/viettran295/cnn_thesis.

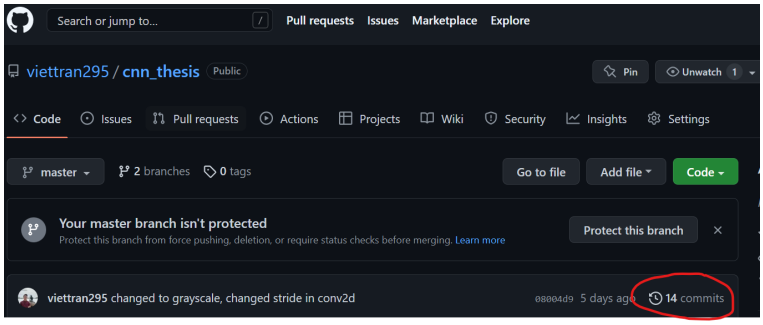

Fig. 4.2: Remote repository on Github

With the button “commits” (red circle) the other users can track what and who changed the source code. In this project there are two branches: master and test branch, which are tracked. The new features of this project are coded in the “test” branch. When everything is fine in that branch, it will be merged into the “master” branch:

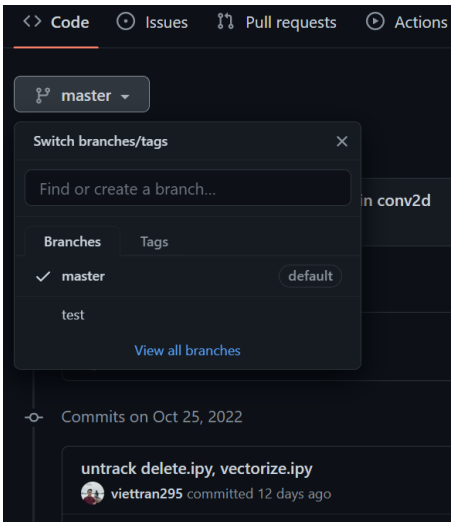

Fig. 4.3: The master and test branches in this project

Weights & Biases platform

Weights & Biases platform is a cloud-based tool to automate and track the models. It provides different useful features: compare various experiments, find and re-run previous model checkpoint, ... [27]

Hyperparameters optimization is the main focus in this project. “Sweep” feature combines different hyperparameters together to determine the model performance and track experiments. This function can also find out the correlation between hyperparameters and the metrics users care about. All of the process can be visualized. The model of this project is stored in the remote repository: https://wandb.ai/viettran/cnn_thesis/sweeps/zdm5se9n?workspace=user-viettran

Fig. 4.4: Workspace in Weights & Biases

(1): the main workspace, where all of the experiments can be tracked with their metrics
(2): “sweep” function, which combines different hyperparameters
(3): artifacts, where metadata of models are saved and can be re-run the model.
(4): charts, which track the desired metrics, e.g. loss, validation loss, accuracy, … .

4.1.4 Kaggle

Hardware is also an important factor that needs to be considered in Machine Learning training. In comparison to a CPU (Central Processing Unit), a GPU (Graphic Processing Unit) has thousands of cores that can break down the complex problem into thousands of separate tasks and compute parallelly.

Kaggle is an open cloud-based platform that provides GPUs resources on cloud and allows the maximum usage of GPU 30 hours per week. Kaggle Notebook allows to run the code on cloud servers without needing to install anything locally. Moreover Kaggle also provides access to cloud data storage through Google Cloud Storage, where the user can store the code and the training datasets. In order to boost the training process most of the tuning experiments in this project are on Kaggle. The tuning and training notebooks are stored in kaggle.com/viettrann/code

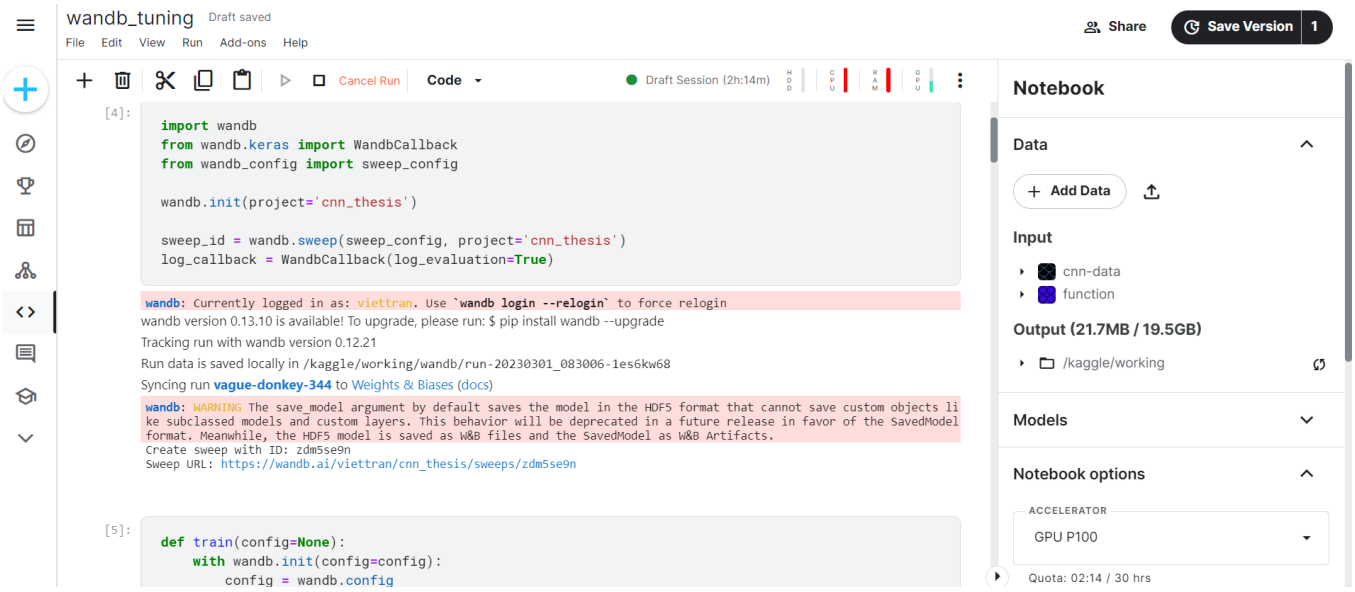

Fig. 4.5: Kaggle Notebook interface

4.2 Images augmentation and preprocessing

Images augmentation

For image augmentation *imgaug* library is used in this project, which supports a wide range of augmentation techniques [28]. To save the computational time, random values are created with the function *np.random.rand()* from the *numpy* library, instead of

implementing all of the augmentation techniques. If these random values are only smaller than 0.5, an augmentation technique is applied. With these lines of code the position of the image will change from -10% to 10% in x-axis and y-axis. The new image is shifted to the upper and left side.

Code snippet 4.2: code for shifting images

```
from imgaug import augmenters as iaa
if np.random.rand() < 0.5:
        aff = iaa.Affine(translate_percent={'x':(-0.1, 0.1),
'y':(-0.1, 0.1)})
        img = aff.augment_image(img)
```

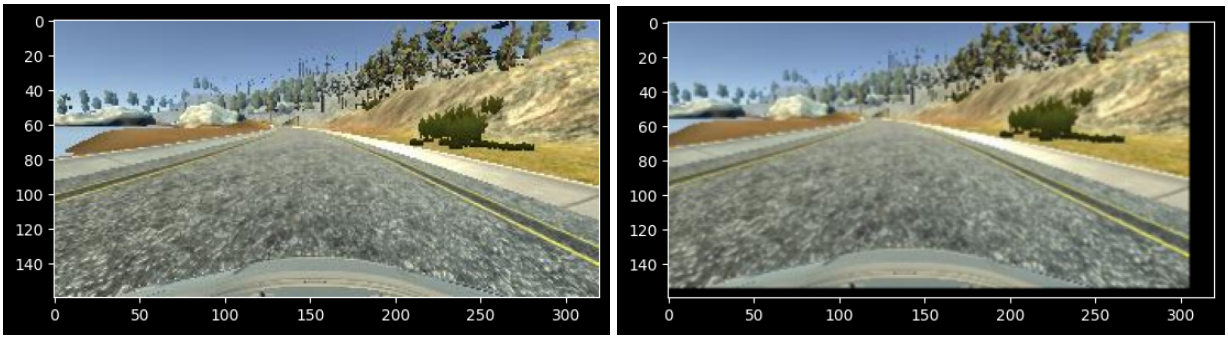

Fig. 4.6: The original image (left) and left and upper shifted image (right)

The next few lines of code zoom in process is applied with scale range from 1.2 to 1.4

Code snippet 4.3: code for zooming images

```
if np.random.rand() < 0.5:
        zoom = iaa.Affine(scale=(1.2, 1.4))
        img = zoom.augment_image(img)
```

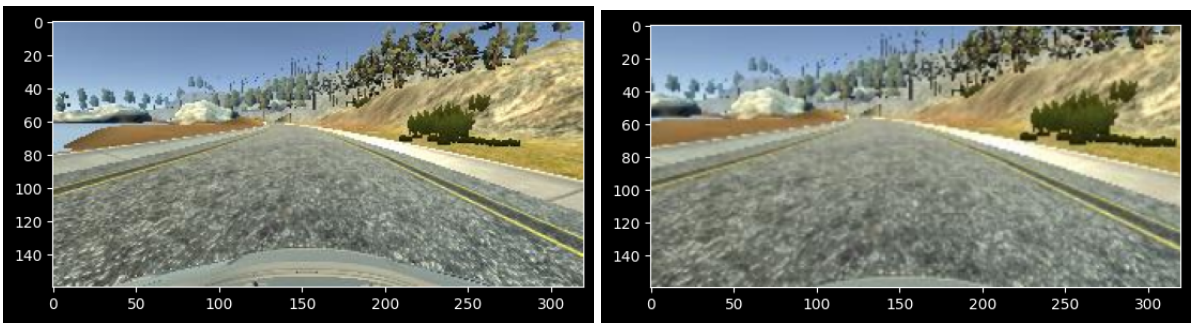

Fig. 4.7: The original image (left) and zoomed image (right)

The brightness of the image is also adjusted to make the training data more diverse with these lines of code. When the brightness value is greater than 1, the original image is brighter and it is opposite when the brightness value is lower than 1.

Code snippet 4.4: code for adjusting the brightness of images

```
if np.random.rand() < 0.5:
        brightness = iaa.Multiply((0.5, 1.5))
        img = brightness.augment_image(img)
```

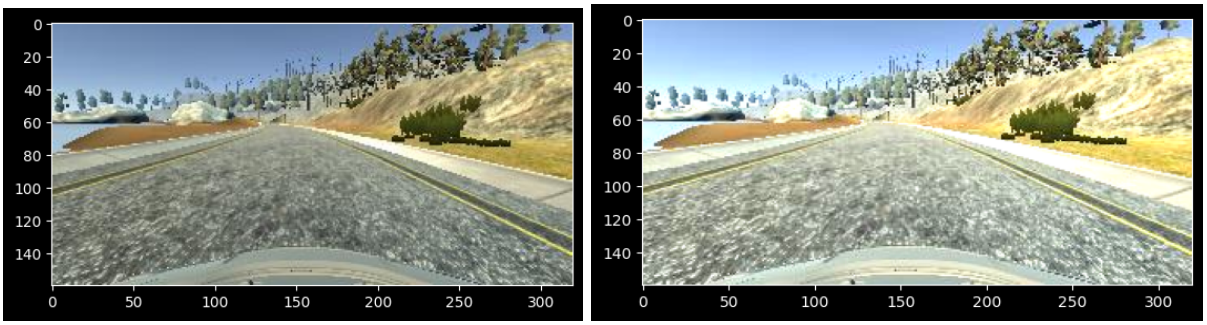

Fig. 4.8: The original image (left) brightness value is greater than 1 (right)

The training images are flipped around the y-axis with these lines of code. When flipping process is applied, steering angle must be reversed to adapt a new image

Code snippet 4.5: code for flipping images

```
if np.random.rand() < 0.5:
        img = cv2.flip(img, 1)
        steering = -steering
```

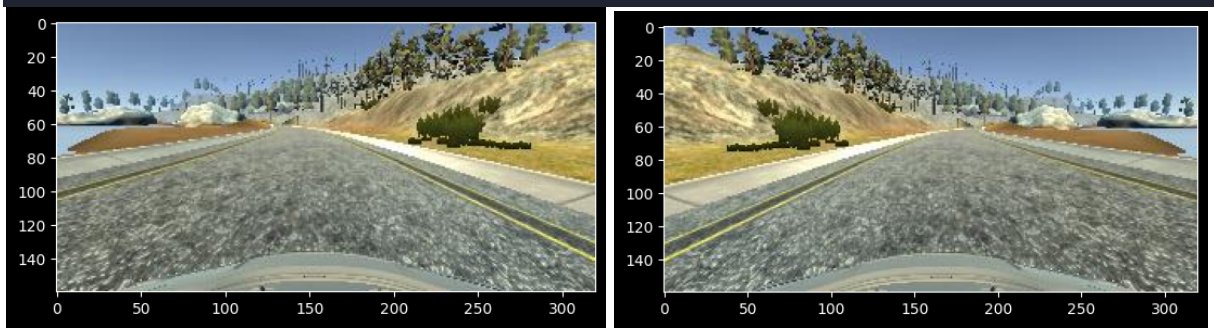

Fig. 4.9: The original image (left) and flipped image (right)

Images preprocessing

Drop image: to reduce the image size and increase computational speed the images need to be cut unnecessary areas, which is above and below the road of the image. With these lines of code it will concentrate on the field, which contains the road.

Code snippet 4.6: code for dropping the size images

```
img = img[60:130,:,:]
```

With the first and second position of this array are the rows (height) and columns (width) of the image. The remaining part of the image is from the 60th to 130th row and all of the columns.

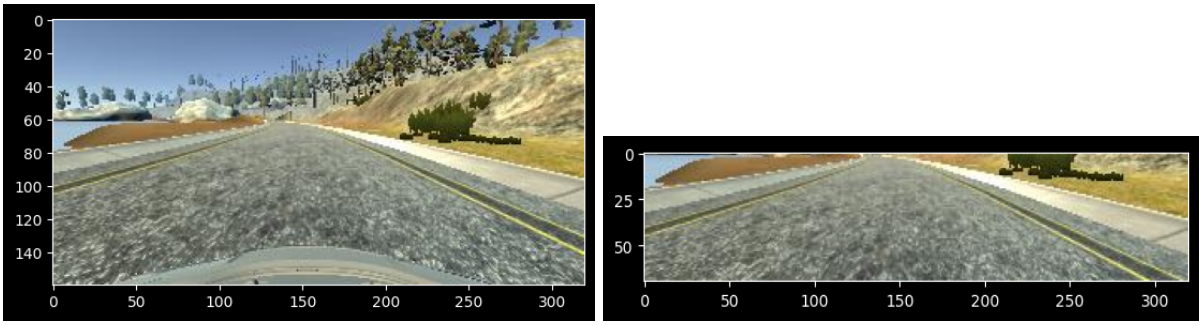

Fig. 4.10: The original image (left) and dropped image (right)

Change color space: in order to increase the computational speed, the dimensions of the images are reduced from 3 channels to 1 channel of the images, which means the color space RGB has to be changed into grayscale [9]. This color space is described in *2.1.1 Color space*.

Code snippet 4.7: code for changing the color space of images

```
img = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
```

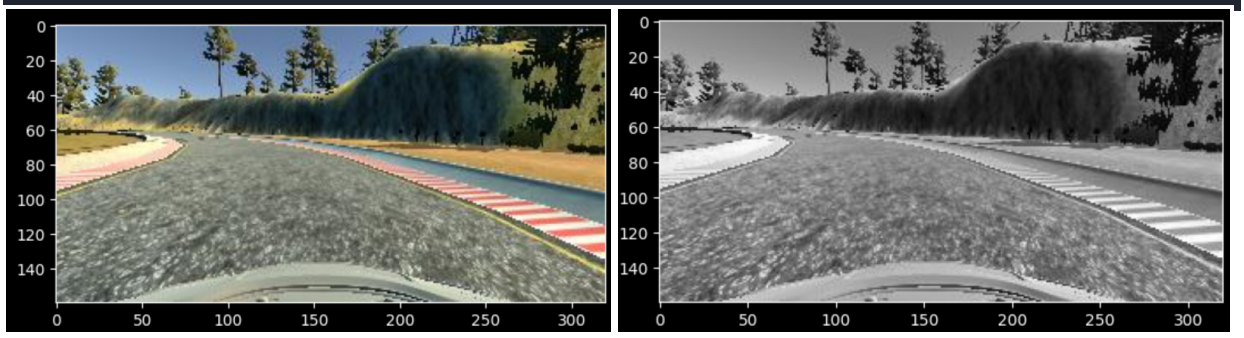

Fig. 4.11: The original image (left) and grayscale image (right)

Blur images: the low-pass Gaussian filter is applied to remove noise and blur the image. With this line of code a Gaussian kernel 5x5 and no stride is applied to the image.

Code snippet 4.8: code for blurring images with kernel 3x3

```
img = cv2.GaussianBlur(img, (3,3))
```

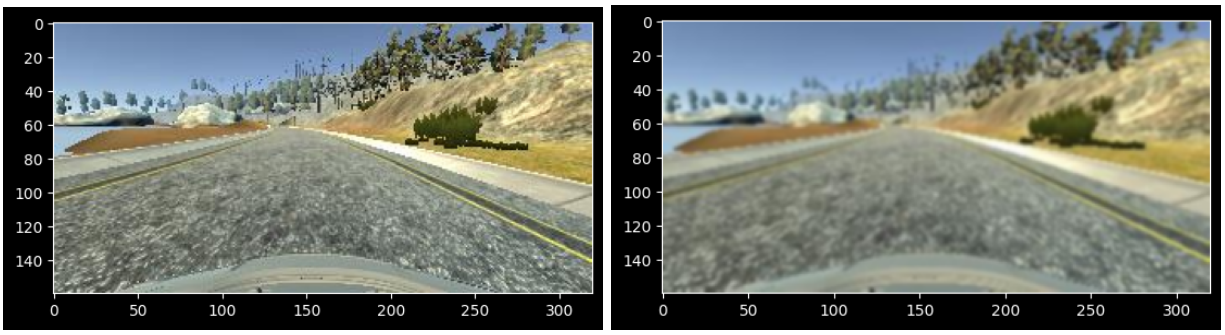

Fig. 4.12: The original image (left) and blurred image (right)

4.3 The program

Before running all of the hyperparameters need to be defined in “sweep” configuration (more detail in *4.1.3.2 Weights & Biases platform*). This configuration is stored in the python file *wandb_config.py.* The values of the hyperparameter are limited min and max values. The result of these elements is saved for the regression model.

Code snippet 4.9: configuration for hyperparameter to run the experiments

```
sweep_config = {
    'method': 'random',
    'name': 'sweep',
    'metric': {
        'name': 'val_loss',
        'goal': 'minimize'
    },
    'parameters':{
        'epochs': {
            'values': [35]
        },
        'batch_size': {
            'distribution': 'int_uniform',
```

```
            'min': 10,
            'max': 200
        },
        'dropout': {
            'distribution': 'int_uniform',
            'min': 0,
            'max': 0.6
        },
        'learning_rate': {
            'distribution': 'int_uniform',
            'min': 0.01,
            'max': 0.0001
        },
        'optimizer':{
            'values': ['Adam']
        },
        'activation': {
            'values': ['tanh']
        }
    }
}
```

The configuration parameters:
method: combine the hyperparameters randomly
metrics: the target of this tuning is to minimize the validation loss.
epochs: is an iteration over the whole dataset. To be more accurate in tuning, all of the models have the same epochs (epochs = 35).
batch_size: number of training samples per weights update.
dropout: the rate of dropped perceptrons in Dropout layers: from 0 to 0.6
learning_rate: learning rate of optimizers: from 0.0001 to 0.01
optimizer: optimizer algorithm is Adam
activation: activation functions is ‘tanh’

Component Experiments creating

This component contains all of the necessary functions. As mentioned in *4.1.2 Data preprocessing* the steering values around zero need to be cut down in order to balance with the values of steering angles. The function *balance_data()* reduces the steering values zero in the data frame.

Code snippet 4.10: function to balance the dataset

```
def balance_data(dataframe, cols_name: str, sample_remain=3500,
display=True, nbins=31):
    hist, bin = np.histogram(dataframe[cols_name], bins=nbins)
    center = (bin[:-1] + bin[1:]) * 0.5
    # remove center angle to balance dataset
    remove_list = []
    for i in range(nbins):
        bin_list = []
        for j in range(len(dataframe[cols_name])):
            if dataframe[cols_name][j] >= bin[i] and
dataframe[cols_name][j] <= bin[i+1]:
                bin_list.append(j)
        bin_list = shuffle(bin_list)
        bin_list = bin_list[sample_remain:]
        remove_list.extend(bin_list)

    dataframe.drop(dataframe.index[remove_list], inplace=True)
    return dataframe
```

The input parameters:
dataframe: Pandas DataFrame
cols_name: column name of the data frame
sample_remain: numbers of data will be kept
display: if this parameter is true, histogram of the dataset will be displayed
nbins: number of bins in the histogram

As described in *2.2.3 Vectorization* the images and the steering values are vectorized in a *numpy array* in order to speed up the computation. *Load_data_to_array()* function will pack the data from data frame into *numpy array*

Code snippet 4.11: function to vectorize the dataset

```
# load img path and steering from dataframe to np array
def load_data_to_array(path, dataframe):
    imgPath = []
    steering = []
    for i in range(len(dataframe)):
        tmp = dataframe.iloc[i]
```

```
        imgPath.append(os.path.join(path, 'IMG', tmp[0])) # first col
of Dataframe
        steering.append(tmp[3]) # third col of Dataframe
    return np.asarray(imgPath), np.asarray(steering)
```

The input parameters:
path: path, where the images are saved
dataframe: Pandas DataFrame
Before feeding into a deep learning model the images need to be preprocessed. *Img_preprocess_pipeline()* is a combination of *image augmentation and image preprocessing* (more detail in *4.1.2 Data preprocessing*). This function pairs the images as training data with the steering values as labels.

Code snippet 4.12: function to preprocess the images

```
def img_preprocess_pipeline(img_path_arr, steering_arr,
train_flag=True):
    img_batch = []
    steering_batch = []
    for i in range(len(img_path_arr)):
        idx = random.randint(0, len(img_path_arr)-1)
        if train_flag:
            img, steering = augment_img(img_path_arr[idx],
steering_arr[idx])
        else:
            img = mpimg.imread(img_path_arr[idx])
            steering = steering_arr[idx]
        img = img_preprocessing(img)
        img_batch.append(img)
        steering_batch.append(steering)
    return (np.asarray(img_batch), np.asarray(steering_batch))
```

The input parameters:
img_path_arr: path to the images, which is vectorized
steering_arr: vectorized steering values
train_flag: if the images are for training, they will be augmented and preprocessed. If the images are for testing, augmentation and preprocessing are not necessary

The experiments creating process is written in the Jupyter Notebook. The notebook consists of a sequence of cells and the code in every cell can be executed independently.

Code snippet 4.13: code for calling the function to read and preprocess the data

```
from helper import *

path = 'data'
df = load_data(f'{path}/driving_log.csv')
df = balance_data(df, 'Steering', sample_remain=3000)
```

Next step DataFrame is loaded into a vector with a support of *load_data_to _array()* function. After that the vector is splitted into training and validation data with the function *train_test_split()*: 80% of the data are used for training and 20% are used for validation. Before feeding into the model the training images need to be preprocessed through the pipeline *img_preprocess_pipeline()*. This function pairs the images as training data with the steering values as labels.

Code snippet 4.14: code for calling the function to vectorize and preprocess images

```
from sklearn.model_selection import train_test_split

img_path, steering = load_data_to_array(path, df)
# fixed value for random_state to keep train and test datasets same
each time
X_train, X_val, Y_train, Y_val = train_test_split(img_path, steering,
test_size=0.2, random_state=1)
X_train, Y_train = img_preprocess_pipeline(X_train, Y_train, 1)
X_val, Y_val = img_preprocess_pipeline(X_val, Y_val, 0 )
```

In the next cell Weights & Biases library is imported and initialized. The results and hyperparameter of each experiment is stored and tracked in a project named ‘cnn_thesis’. In order to optimize and accelerate the tuning process *sweep* feature is also initialized with the function *wandb.sweep()*. This function is described in more detail in *4.1.3.2 Weights & Biases*.

Code snippet 4.15: code for running the experiments on Weights & Biases platform

```
import wandb
from wandb.keras import WandbCallback
from wandb_config import sweep_config
```

```
wandb.init(project='cnn_thesis')
sweep_id = wandb.sweep(sweep_config, project='cnn_thesis')
```

Component Multivariate regression

This component uses data from experiments in CNN model to build a multivariate quadratic regression model. Before feeding data into quadratic regression, the original features (dropout, learning rate, batch size) need to be transformed into a new matrix consisting of the combinations of the features with the degree less than or equal to two. The new features array has a form: $[1,\ a,\ b,\ c,\ a^2,\ ab,\ ac,\ b^2,\ bc,\ c^2]$. After that `LinearRegression()` is used as a base to find the coefficients of the model.

Code snippet 4.16: code for determining the regression model

```
degree = 2
poly_reg = PolynomialFeatures(degree=degree)
poly_x_train = poly_reg.fit_transform(x_train)
regr = LinearRegression()
regr.fit(poly_x_train, y_train)
```

After defining the regression model, it is derived with respect to variables: dropout, learning rate and batch size. The system of equations is stored in an array $coef\ = [1,\ a,\ b,\ c,\ a^2,\ ab,\ ac,\ b^2,\ bc,\ c^2]$ and is solved by the numpy function `np.linalg.solve()` to find the values of hyperparameters.

Code snippet 4.17: code for solving the system of equations

```
coef_matrix = np.array([[2*coef[4], coef[5], coef[6]],
                        [coef[5], 2*coef[7], coef[8]],
                        [coef[6], coef[8], 2*coef[9]]])
depen_var = np.array([-coef[1], -coef[2], -coef[3]])
res = np.linalg.solve(coef_matrix, depen_var)
```

A callback function *ModelCheckpoint* from Tensorflow is added in this component. *ModelCheckpoint* saves a state of a model or weights and they can be loaded later to continue training from the saved state. The state is saved in the .hdf5 file.

Code snippet 4.18: code for training and saving the model

```
from keras.callbacks import ModelCheckpoint

model = build_network('tanh', 'Adam')

checkpoint_path = f'model/{opt}-{lr}.hdf5'
checkpoint_callback = ModelCheckpoint(
    filepath=checkpoint_path,
    monitor='val_loss',
    verbose=1,
    save_best_only=True
)
hist = model.fit(X_train, Y_train, validation_data=(X_val, Y_val),
batch_size=93, epochs=35, shuffle=True,
callbacks=[checkpoint_callback])
```

5. Results

Hyperparameters optimization

In order to build the multivariate quadratic regression model the data need to be collected by combining the range of following hyperparameters in CNN model. To be more accurate all of the tests are run with the same epochs (epochs = 30):

Batch size	10 - 200
Dropout	0 - 0.6
Learning rate	10^{-4} - 10^{-2}

Tab. 5.1: The values of hyperparameters for data collecting

After running 228 experiments, a table of hyperparameters with 6 columns x 228 rows is collected to feed into a multivariate quadratic regression model. The column “mean” is the average of training loss and validation loss

	Batch size	Learning rate	Dropout	Loss	Validation loss	Mean
0	182	0.005448	0.694351	0.078256	0.041367	0.059811
1	119	0.003378	0.493624	0.059851	0.039452	0.049652
2	115	0.004924	0.684175	0.062070	0.040731	0.051400
3	181	0.003706	0.536575	0.068155	0.040875	0.054515
4	143	0.004607	0.573039	0.062509	0.041824	0.052166
...	...	...	...	...	...	...
226	58	0.007874	0.317321	0.081749	0.059558	0.070653
227	35	0.001483	0.302686	0.060643	0.059847	0.060245
228	28	0.002530	0.139669	0.053006	0.059653	0.056329

Tab. 5.2: Dataset of hyperparameters.

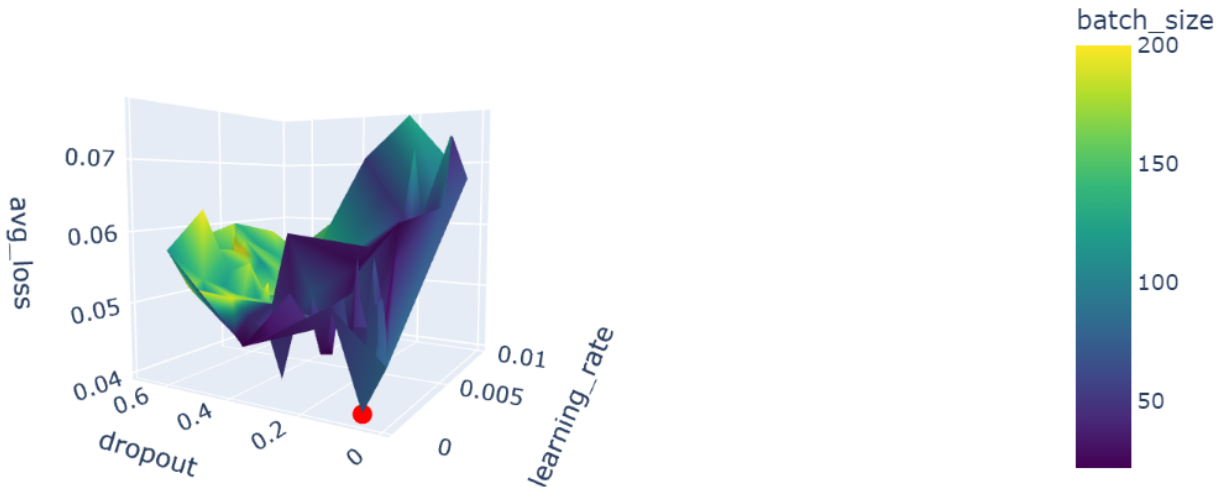

Fig. 5.1: Average loss and hyperparameters

The graph in *figure 5.1* has four dimensions with: x - dropout, y - learning rate, z - mean loss, color - batch size. The red point has the lowest mean loss 0.0475, which is found by feeding *table 5.2* into the regression model (in *Chapter 3.5 Hyperparameter tuning using multivariate quadratic regression*).

The multivariate quadratic regression model:

$$y = -1.255.10^{-4}a + 2.946\,b - 2.162.10^{-2}c + 4.045.10^{-7}a^2 - 6.932\,b^2 + 4.351.10^{-2}c^2 \\ 7.696.10^{-3}a\,b - 8.479.10^{-5}\,a\,c - 5.908\,b\,c \tag{5.1}$$

The regression model (5.1) is defined with the lowest mean squared error $1.4.10^{-4}$. The derivative of regression model (5.1):

$$\frac{\delta f(a,b,c)}{\delta a} = 8.09.10^{-7}.\,a + 7.696.10^{-3}.\,b - 8.479.10^{-5}.\,c - 1.255.10^{-4} = 0$$

$$\frac{\delta f(a,b,c)}{\delta b} = 7.696.10^{-3}\,a - 13.864\,b - 5.908\,c + 2.946 = 0$$

$$\frac{\delta f(a,b,c)}{\delta c} = -8.479.10^{-5}\,a - 5.908\,b + 8.702.10^{-2}\,c - 2.162.10^{-2} = 0 \tag{5.2}$$

The results of system of equations (5.2):
$a\ or\ batch\ size = 92.684$
$b\ or\ learning\ rate = 0.0034$
$c\ or\ dropout = 0.066$

Batch size number has to be a natural number. That is why the batch size is rounded to 93 and the learning rate and dropout stay the same. These hyperparameters are passed into the CNN model.

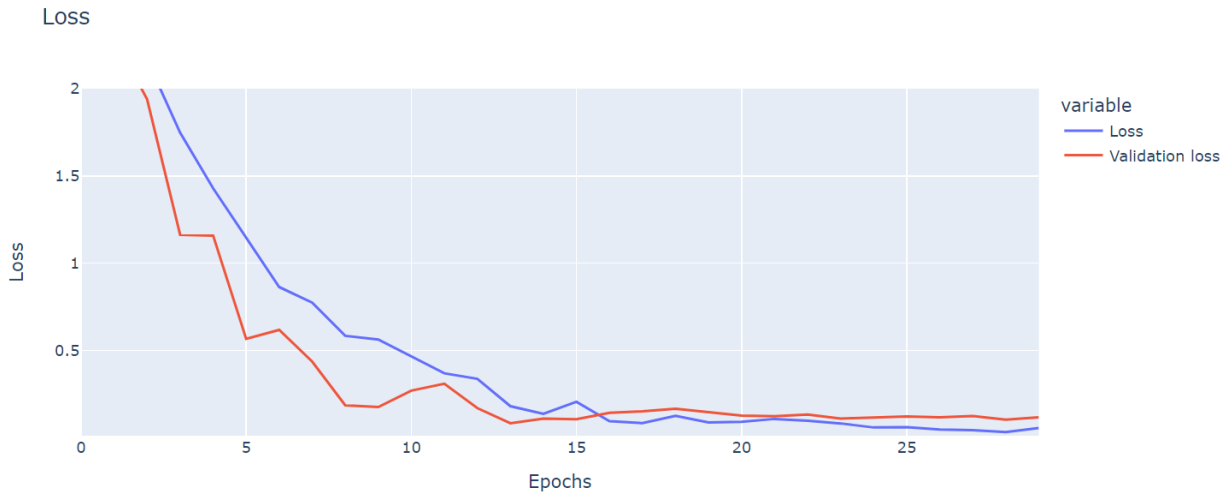

Fig. 5.2: Loss and validation loss

In this training EarlyStopping() in Tensorflow is applied, which makes the training stop if after 3 epochs the loss is no longer decreasing. The above graph shows training loss and validation loss of the model in 30 epochs. Both losses come closer together after 15 epochs. After every epoch the training dataset is shuffled that makes the loss lines unsmooth. After 15 epochs the validation loss is steady around 0.073 and training loss is approximately 0.053. The difference between loss and validation loss is not significant. The training loss 0.053 is slightly higher than 1^{O} in steering angle (max steering angle is 25^{O}, max steering value is 1)

Performance evaluation

This section evaluates the performance of the model after tuning and training. Because of the problems described in *3.3.2 Data preprocessing - smoothing steering values*, the actual steering values need to be smoothened by Gaussian kernel smoothing method with the window size is 70 [29]. The blue line is the actual steering value and the red line is the prediction value.

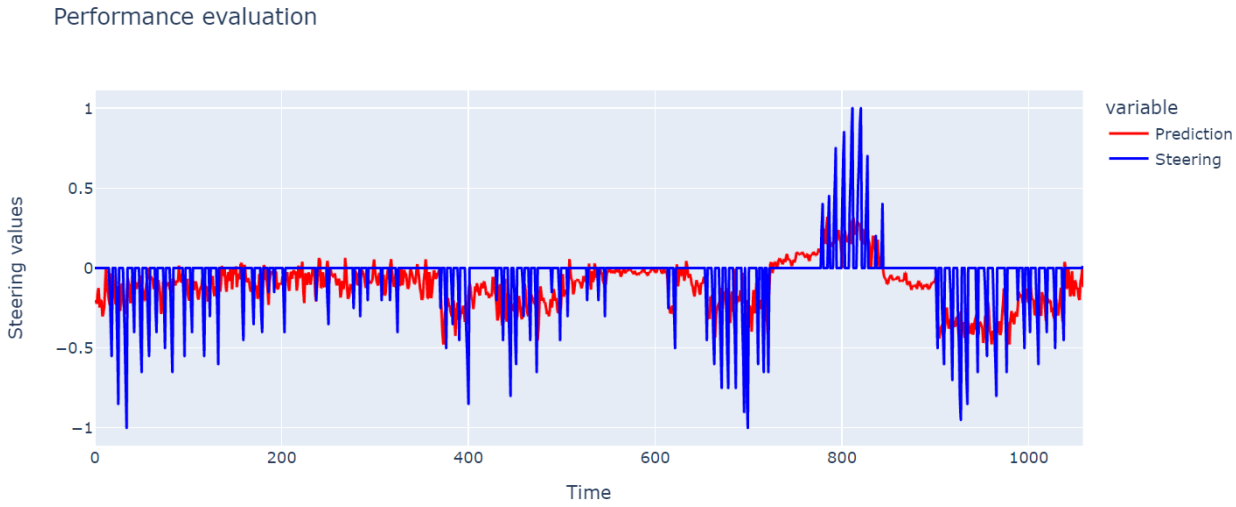

Fig. 5.3: Steering values and prediction before smoothing

The following graph shows the difference of steering values between the true values after smoothing and the prediction of the model. The mean square error between them is 0.049

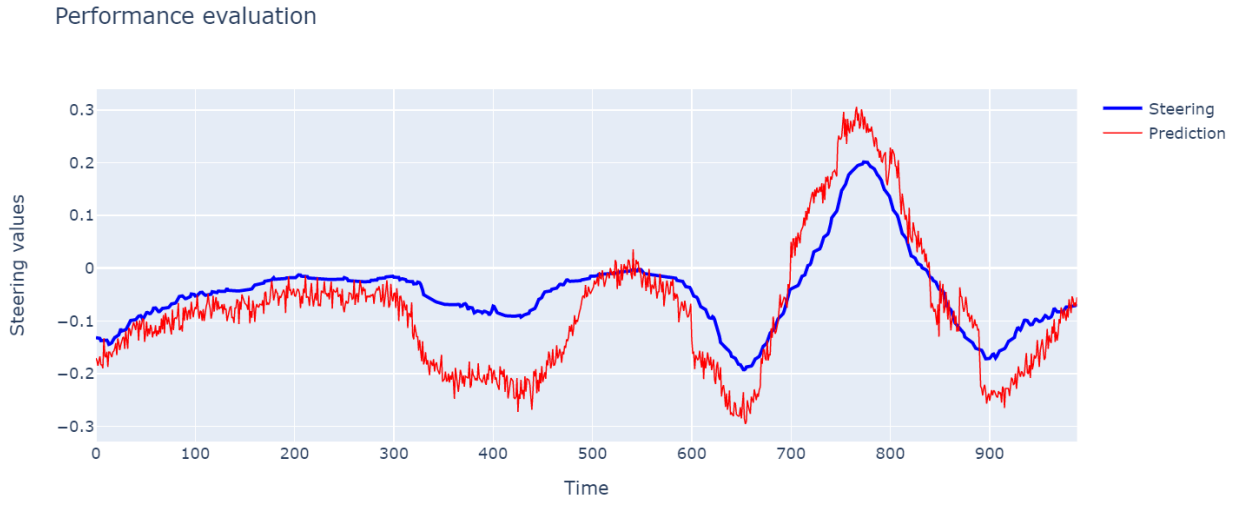

Fig. 5.4: Steering values and prediction after smoothing

Summary

In this thesis the multivariate quadratic regression is applied for the tuning process of convolutional neural networks. This method determined the values of hyperparameters, which makes the validation loss lowest. Building a regression model for the tuning process can avoid randomly choosing hyperparameters that saves much computational resources and time. This method can define the exact values and it can reduce the range to choose for every hyperparameter. It is necessary to continue the tuning process after solving the system of equations.

There are two maps in the simulator for training and testing. The training dataset is collected from the first map and then the trained model is tested in the second map. After training the CNN model can work well in the first map, from which the images are collected for the training dataset. But the model has a problem when it works in a completely new second map, which is never seen before. The model can not detect the road to predict the steering angle. This model still has limits to generalize to new and unseen dataset.

For further research it is recommended that the tuning process is optimized with other regression methods (e.g, polynomial regression with higher degree or regression tree) to compare the difference with the multivariate quadratic regression.

References

[1] M. Bojarski, D. Del Testa, and D. Dworakowski, “End to End Learning for Self-Driving Cars,” in *Ieee*, Apr. 2016. [Online]. Available: https://images.nvidia.com/content/tegra/automotive/images/2016/solutions/pdf/end-to-end-dl-using-px.pdf

[2] Net-Scale Technologies, INC, “Autonomous Off-Road Vehicle Control Using End-to-End Learning,” in *Final technical report*, Apr. 2014. [Online]. Available: http://net-scale.com/doc/net-scale-dave-report.pdf

[3] A. C. Serban, E. Poll, and J. Visser, “A Standard Driven Software Architecture for Fully Autonomous Vehicles,” in *2018 IEEE International Conference on Software Architecture Companion (ICSA-C)*, Apr. 2018. Accessed: Dec. 25, 2022. [Online]. Available: http://dx.doi.org/10.1109/icsa-c.2018.00040

[4] Q. Liu and Y. Wu, “Supervised Learning,” in *Encyclopedia of the Sciences of Learning*, Boston, MA: Springer US, 2012, pp. 3243–3245. Accessed: Dec. 25, 2022. [Online]. Available: http://dx.doi.org/10.1007/978-1-4419-1428-6_451

[5] IBM Cloud Education, “Neuronale Netzwerke", IBM, Aug. 17, 2020. https://www.ibm.com/de-de/cloud/learn/neural-networks

[6] P. Baheti, “V7,” Oct. 21, 2022. https://www.v7labs.com/blog/neural-networks-activation-functions (accessed Dec. 25, 2022).

[7] M. Krishnakumar, “The 6 Autonomous Driving Levels Explained,” *W&B*, Sep. 03, 2022. (accessed Dec. 25, 2022).

[8] A. MacKenzie, “Mercedes-Benz Drive Pilot Level 3 Autonomous First ‘Drive’: We Try a World’s First Driverless System,” *MotorTrend*, May 06, 2022.

https://www.motortrend.com/news/mercedes-benz-drive-pilot-eqs-autonomous-driverless-first-drive-review/ (accessed Dec. 25, 2022).

[9] “Blurring Images – Image Processing with Python,” *Data Carpentry - Image Processing with Python*. https://datacarpentry.org/image-processing/06-blurring/ (accessed Jan. 06, 2023).

[10] “tf.keras.layers.Conv2D ,” *TensorFlow*. https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D (accessed Jan. 25, 2023).

[11] “Spatial Filters,” *Gaussian Smoothing*. https://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm (accessed Jan. 06, 2023).

[12] A. Zhang, Z. Lipton, M. Li, and A. J. Smola, *Dive Into Deep Learning: Interactive Deep Learning Book with Code, Math and Discussions ; Implemented with PyTorch, NumPy/MXNet, and TensorFlow*. 2022.

[13] A. Botchkarev, “Performance Metrics (Error Measures) in Machine Learning Regression, Forecasting and Prognostics: Properties and Typology,” *Interdisciplinary Journal of Information*, Sep. 2018.

[14] “Reducing Loss: Gradient Descent ,” *Google Developers*. https://developers.google.com/machine-learning/crash-course/reducing-loss/gradient-descent (accessed Jan. 06, 2023).

[15] N. Seth, “Is Gradient Descent Sufficient for Neural Network Models,” *Analytics Vidhya*, Apr. 01, 2021. https://www.analyticsvidhya.com/blog/2021/04/is-gradient-descent-sufficient-for-neural-network/ (accessed Jan. 06, 2023).

[16] “Reducing Loss: Stochastic Gradient Descent ,” *Google Developers*. https://developers.google.com/machine-learning/crash-course/reducing-loss/stochastic-gradient-descent (accessed Jan. 06, 2023).

[17] G. Hinton, http://www.cs.toronto.edu/~hinton/coursera/lecture6/lec6.pdf (accessed Jan. 06, 2023).

[18] D. P. K. Ba Jimmy Lei, “Adam: A Method for Stochastic Optimization.” https://arxiv.org/pdf/1412.6980.pdf (accessed Jan. 06, 2023).

[19] K. Pykes, “Fighting Overfitting With L1 or L2 Regularization: Which One Is Better?,” *neptune.ai*, Jul. 22, 2022. https://neptune.ai/blog/fighting-overfitting-with-l1-or-l2-regularization (accessed Jan. 06, 2023).

[20] N. Srivastava, G. Hinton, and A. Krizhevsky, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting,” *Journal of Machine Learning Research*, 2014.

[21] “CS231n Convolutional Neural Networks for Visual Recognition.” https://cs231n.github.io/convolutional-networks/ (accessed Jan. 06, 2023).

[22] A. Fisher, C. Rudin, and F. Dominici , “All Models are Wrong, but Many are Useful: Learning a Variable’s Importance by Studying an Entire Class of Prediction Models Simultaneously,” Dec. 2019

[23] “Multivariate Regression,” *Brilliant*. https://brilliant.org/wiki/multivariate-regression/ (accessed Jan. 25, 2023).

[24] “Backpropagation,” *Brilliant*. https://brilliant.org/wiki/backpropagation/ (accessed Jan. 25, 2023).

[25] Udacity, “GitHub - udacity/self-driving-car-sim: A self-driving car simulator built with Unity,” *GitHub*. https://github.com/udacity/self-driving-car-sim (accessed Jan. 25, 2023).

[26] “TensorFlow,” *TensorFlow*. https://www.tensorflow.org/ (accessed Jan. 25, 2023).

[27] “Wandb,” *Documentation*. https://docs.wandb.ai/quickstart (accessed Jan. 25, 2023).

[28] “imgaug — imgaug 0.4.0 documentation.” https://imgaug.readthedocs.io/en/latest/ (accessed Jan. 25, 2023).

[29] J. Dancker, “A brief introduction to time series smoothing - Jonte Dancker,” *Medium*, Sep. 27, 2022. https://medium.com/@jodancker/a-brief-introduction-to-time-series-smoothing-4f7ed61f78e1 (accessed Feb. 08, 2023).

[30] J. Dancker, “A brief introduction to time series smoothing - Jonte Dancker,” *Medium*, Sep. 27, 2022. https://medium.com/@jodancker/a-brief-introduction-to-time-series-smoothing-4f7ed61f78e1 (accessed Feb. 28, 2023).

[31] S. Saha, “A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way,” *Towards Data Science*, Nov. 16, 2022. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53 (accessed Feb. 28, 2023).