DOKUZ EYLÜL UNIVERSITY

GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

AUTONOMOUS VEHICLE SIMULATION WITH DEEP REINFORCEMENT LEARNING IN VIDEO GAME ENVIRONMENT

by

Ahmet Onur GİRAY

May, 2023

İZMİR

AUTONOMOUS VEHICLE SIMULATION WITH DEEP REINFORCEMENT LEARNING IN VIDEO GAME ENVIRONMENT

A Thesis Submitted to the
Graduate School of Natural And Applied Sciences of Dokuz Eylül University
In Partial Fulfillment of the Requirements for the Degree of Master of Science in Electrical and Electronics Engineering

by

Ahmet Onur GİRAY

May, 2023
İZMİR

M.Sc THESIS EXAMINATION RESULT FORM

We have read the thesis entitled "**AUTONOMOUS VEHICLE SIMULATION WITH DEEP REINFORCEMENT LEARNING IN VIDEO GAME ENVIRONMENT**" completed by **AHMET ONUR GIRAY** under supervision of **ASSOC. PROF. DR. HATICE DOĞAN** and we certify that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

.....

Assoc. Prof. Dr. Hatice DOĞAN

Supervisor

.....	.....
Asst. Prof. Dr. Nalan ÖZKURT	Asst. Prof. Dr. Neslihan AVCU
(Jury Member)	(Jury Member)

Prof.Dr. Okan FISTIKOĞLU
Director
Graduate School of Natural and Applied Sciences

ACKNOWLEDGEMENTS

I would like to thank my supervisor Assoc. Prof. Dr. Hatice Doğan for her continued faith and support. I also would like to express my gratitude to Gamze Çağlayan, my dear partner, for constantly encouraging me to move forward when I'm feeling low.

This thesis work is supported by Dokuz Eylül University Scientific Research grant 2021.KB.FEN.005.

Ahmet Onur GİRAY

AUTONOMOUS VEHICLE SIMULATION WITH DEEP REINFORCEMENT LEARNING IN VIDEO GAME ENVIRONMENT

ABSTRACT

In recent years, research on the development of autonomous vehicles has gained momentum in order to ensure comfortable and safe driving. Autonomous vehicles can move without the driver's input by using sensors and cameras. This thesis proposes the use of deep double Q-learning algorithm with experience replay to enable an autonomous vehicle to travel on a highway lane at a specific speed in a video game environment. The reinforcement learning algorithm uses the vehicle's angle to the road center line, vehicle speed, distance to the road center, and acceleration as inputs for state information. Therefore, this information must be determined with high accuracy. Convolutional neural networks, which show remarkable performance in many fields, can be used to obtain the position of the lanes and the angle of the vehicle. CNN models are trained with datasets that are too large to be collected and labeled manually. This thesis also presents a method for the automatic collection of a labeled dataset from a video game environment to be used for the detection of road lanes on a highway. With 745823 automatically collected images, different CNN models such as ResNet50, VGG16, Xception, and InceptionV3 networks are trained using the Transfer Learning method. According to the performance comparisons made with the data set obtained from a highway not included in the training, ResNet50 performs best.

Keywords: Convolutional neural networks, autonomous driving, road lane detection, deep reinforcement learning

VİDEO OYUN ORTAMINDA DERİN PEKİŞTİRMELİ ÖĞRENME İLE OTONOM ARAÇ SİMULASYONU

ÖZ

Son yıllarda, rahat ve güvenli sürüşü sağlamak için otonom araçların geliştirilmesi üzerine araştırmalar ivme kazanmıştır. Otonom araçlar, sensörler ve kameralar kullanarak sürücü girdisi olmadan hareket edebilirler. Bu tez, otonom bir aracın video oyunu ortamında belirli bir hızda otoyol şeridinde seyahat etmesini sağlamak için deneyim tekrarı ile derin çift Q pekiştirmeli öğrenme algoritmasının kullanımını önermektedir. Pekiştirmeli öğrenme algoritması, aracın yol merkez çizgisine olan açısını, araç hızını, yol merkezine olan mesafeyi ve ivmeyi durum bilgisi girdileri olarak kullanır. Bu nedenle, bu bilgiler yüksek doğrulukla belirlenmelidir. Birçok alanda dikkate değer performans sergileyen evrişimli sinir ağları, şeritlerin konumlarını ve aracın açısını elde etmek için kullanılmıştır. CNN modelleri, manuel olarak toplanıp etiketlenmesi mümkün olmayan çok büyük veri kümeleriyle eğitilir. Bu tez aynı zamanda bir otoyolda yol şeritlerinin tespiti için kullanılmak üzere bir video oyun ortamından etiketli veri setinin otomatik olarak toplanması için bir yöntem sunar. Otomatik olarak toplanmış 745823 adet görsel ile, Transfer Öğrenme yöntemi kullanılarak ResNet50, VGG16, Xception ve InceptionV3 gibi farklı CNN modelleri eğitilmiştir. Eğitimde yer almayan bir otoyoldan toplanan veri kümesiyle yapılan performans karşılaştırmalarına göre, ResNet50 en iyi performansı sergilemiştir.

Anahtar kelimeler: Evrişimli sinir ağları, otonom sürüş, yol şerit tespiti, derin pekiştirmeli öğrenme

CONTENTS

Page

M.Sc THESIS EXAMINATION RESULT FORM ii
ACKNOWLEDGEMENTS iii
ABSTRACT iv
ÖZ v
LIST OF FIGURES ix
LIST OF TABLES x
LIST OF SYMBOLS xi
ABBREVIATIONS xii

CHAPTER 1 – INTRODUCTION 1

CHAPTER 2 – BACKGROUND 7

2.1 Machine Learning Algorithms 7
2.2 Q-Learning 9
2.3 Double Q-Learning 15
2.4 Deep Q-Learning 17
2.5 Convolutional Neural Networks 22
2.5.1 InceptionV3 25
2.5.2 Xception 27
2.5.3 VGG16 28
2.5.4 ResNet50 28

CHAPTER 3 – DETECTION OF ROAD INFORMATION 30

3.1 Extracting Game Data 31
3.2 Automated Data Collection Script 33
3.3 Data Collection 34
3.4 Data Analysis 36

CHAPTER 4 – REINFORCEMENT LEARNING AGENT 38

4.1 State Space 38
4.2 Action Space 40
4.3 Reward Signal 41
4.4 Environment Setup 42

CHAPTER 5 – RESULTS 44

5.1 CNN Model Training 44
5.2 CNN Training Results 45
5.3 RL Agent Training 49
5.4 RL Agent Training Results 49
5.5 Discussion 51
5.6 Conclusion 55

REFERENCES 56

LIST OF FIGURES

Page

Figure 1.1 Variety of driving scenarios in the video game 2
Figure 1.2 Illustration of initial environment setup 4
Figure 2.1 Basic visualization of reinforcement learning 9
Figure 2.2 Algorithm of basic Q-learning 15
Figure 2.3 Algorithm of Q-learning with experience replay 16
Figure 2.4 Algorithm of double Q-learning 17
Figure 2.5 Algorithm of double Q-learning with experience replay 18
Figure 2.6 Difference of Q-learning and deep Q-learning 19
Figure 2.7 Algorithm of deep Q-learning with experience replay 20
Figure 2.8 Algorithm of deep double Q-learning with experience replay 21
Figure 2.9 Visualization of the convolution operation 22
Figure 2.10 Visualization of an example CNN 24
Figure 2.11 Visualization of an Inception module 26
Figure 2.12 Visualization of an InceptionV3 network 26
Figure 2.13 Visualization of an Xception network 27
Figure 2.14 Visualization of a VGG16 network 28
Figure 2.15 Visualization of a ResNet50 network 29
Figure 3.1 Semi-automated labelling tool 30
Figure 3.2 Nodes (left) and links (right) shown on game's map 32
Figure 3.3 Example of image-label pair from the dataset 35
Figure 3.4 Road lanes marked in game 36
Figure 4.1 Visualization of vectors used for the state information 39
Figure 5.1 Three points marked by the model 46
Figure 5.2 Three points marked by the model 47
Figure 5.3 Detection of lane boundaries while lanes are partially visible 48
Figure 5.4 Average rewards calculated during DDQN training 50
Figure 5.5 Label file with vehicle bounding boxes 53

Figure 5.6 Label file information displayed on its image.....54
Figure 5.7 Example output of the MobileNet on a video game image 54

LIST OF TABLES

Page

Table 4.1 Action space and key press durations 41
Table 5.1 Results of the first trainings 45
Table 5.2 Predicted and actual results for Figure 5.1 46
Table 5.3 Predicted and actual results for Figure 5.2 47
Table 5.4 Predicted and actual results for Figure 5.3 48
Table 5.5 Parameters in first stage training 50

LIST OF SYMBOLS

π	: Policy function
γ	: Discount factor
α	: Learning rate
θ	: Network weights for Q-network

ABBREVIATIONS

RADAR	: Radio Detection and Ranging
LIDAR	: Light Detection and Ranging
GTA	: Grand Theft Auto
AI	: Artificial Intelligence
CNN	: Convolutional Neural Network
FPS	: First Person Shooter
TD	: Temporal Difference
DQN	: Deep Q-Network
DDQN	: Deep Double Q-Network
RGB	: Red, Green and Blue
GPU	: Graphics Processing Unit
ReLU	: Rectifier Linear Unit
MSE	: Mean Squared Error
HUD	: Heads-Up Display
API	: Application Programming Interface
ID	: Identity Document
3-D	: 3-Dimensional
2-D	: 2-Dimensional
COCO	: Common Objects in Context

CHAPTER 1
INTRODUCTION

According to the World Health Organization (2018), 1.35 million people die in traffic every year around the world, and related studies have shown that 97.5% of road accidents are caused by human factors (Pakgohar, Tabrizi, Khalili, & Esmaeili, 2011). In this aspect, it is expected that autonomous vehicles would eliminate the human factor and reduce casualties significantly. The automotive industry is continuously developing toward autonomous vehicles as more companies invest in the development of self-driving vehicles. Development, testing, and deployment of autonomous vehicles in real-life scenarios on public roads are costly, time-consuming, and can be dangerous.

This study aims to provide a cheaper and more efficient alternative to this problem by using a popular video game called Grand Theft Auto V developed and published by Rockstar Games, which provides high-quality realistic visuals and traffic environments (GTAV, 2022). GTA V offers a wide variety of scenes with different lighting conditions, day and night cycles, weather conditions, and a large number of vehicle and pedestrian models. The virtual city in the video game called Los Santos is based on real-life Los Angeles with astonishing details. Therefore, the sceneries, types of roads, and traffic patterns are replicas of the real city. Examples of different driving scenarios can be seen in Figure 1.1.

In addition to the selected video game, there are open source simulation environments. Microsoft AirSim, Nvidia AutoSim and Carla are some examples (Dosovitskiy, Ros, Codevilla, Lopez, & Koltun, 2017; Microsoft, 2022; NVIDIA, 2022). However, when compared with GTA V, these are not as visually realistic and they do not have AI vehicles that behave realistically. However, due to being a video game, GTA V has its own challenges in terms of data collection and the development of research tools. These issues are also investigated and methods to overcome them are presented in this thesis.

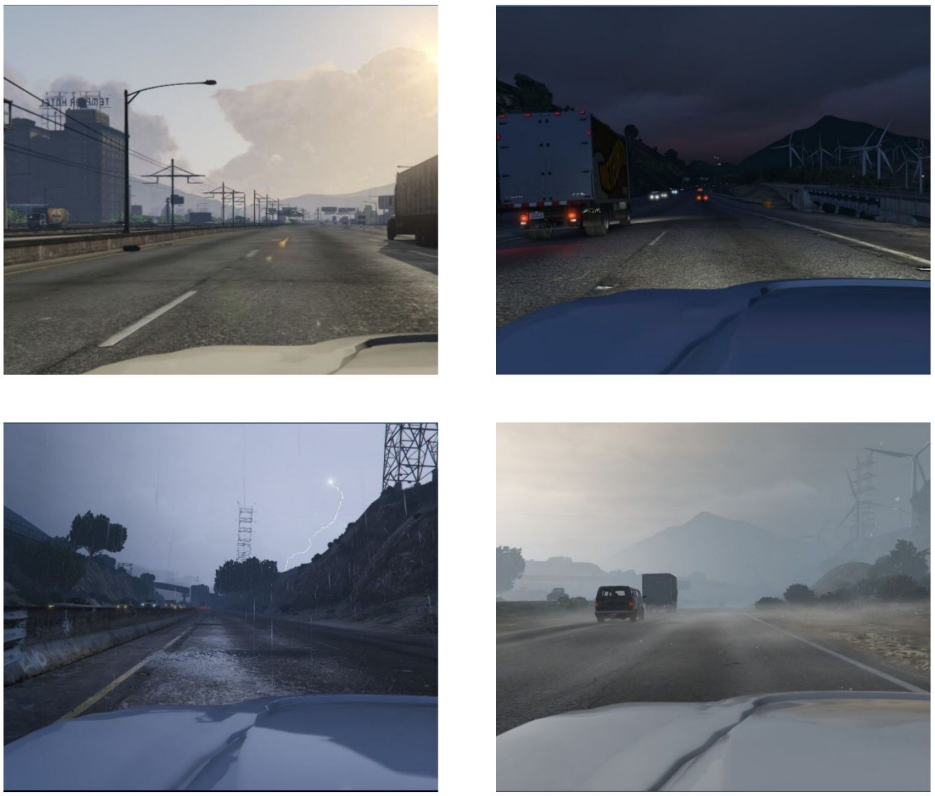


Figure 1.1 Variety of driving scenarios in the video game

Two major topics are investigated in the development of an autonomous vehicle in a video game.

- Development of a convolutional neural network (CNN) to process video footage captured from a camera placed on the hood of the vehicle in order to provide state information to the deep reinforcement learning algorithm,
- Development of an artificial intelligence (AI) with deep reinforcement learning in order to decide the actions of the autonomous vehicle.

First part of this study focuses on detecting road lanes on a highway from a camera placed on a vehicle's dashboard. One of the main goals is to create a framework that generates labeled visual data from the video game that can be used to train CNNs, which are successfully used in autonomous driving tasks (Huval et al., 2015; Muller, Ben, Cosatto, Flepp, & Cun, 2005). Training CNNs requires a huge number of labeled

data.

The datasets generated using the methods described in this study do not require a manual labeling process, therefore allowing the collection of approximately eight thousand image-label pairs during one hour of driving. Data is generated by utilizing third-party tools that exploit source code functions. The labeling of lanes is done directly using game engine data and functions, so the data is collected exactly as the game generates them. This increases the accuracy of the dataset.

Using the methods explained in this study, a dataset of 745,823 frames was collected during approximately 90 hours of driving. From each frame, the angle of the vehicle, the coordinates of the center point of the lane, and the coordinates of the left and right lane boundaries are extracted.

Driving for data collection is also automated using third-party tools, and manual inputs are only required if the vehicle is to be driven to a predefined point in the world. Otherwise, the vehicle moves on the defined parkour without the need for any manual inputs. The collected data is used to train InceptionV3, Xception, VGG16, and ResNet50 models (Chollet, 2017; He, Zhang, Ren, & Sun, 2016; Simonyan & Zisserman, 2014; Szegedy, Vanhoucke, Ioffe, Shlens, & Wojna, 2016). Performance comparisons of these models are presented.

This thesis propose a unique method for the usage of the lane detection network as an information provider about the states of the environment to an agent trained with deep reinforcement learning algorithms. Video game environment is utilized in the most realistic way by extracting state information using convolutional neural networks from a video camera and shows that by improving this network, it is possible to train advanced automation systems in a virtual environment. We present our method and its results in detail, both for state detection and training the AI agent. The reasons for selecting deep reinforcement learning can be listed as follows:

- Supervised learning algorithms require huge datasets.

- If the data sets to be used in the supervised learning method are created for autonomous vehicle simulation, it is highly likely that ideal driving conditions will prevail.

- Creating data sets for many different scenarios, conditions and environments that autonomous vehicles may encounter in traffic and training a neural network using these data sets will take too long and it is possible that it will not produce successful results depending on the adequacy of the data set.

- Demonstrating that it is possible to adapt reinforcement learning algorithms to real life after training them to a safe level in a virtual reality environment, since testing algorithms on real cars is dangerous and expensive.

In order to facilitate deep Q-learning, one of the popular deep reinforcement algorithms, on GTA V, first requirement for the environment is to provide state information to the agent. To narrow down the scope of the simulation, these states are selected as the vehicle's angle to the road lane center and vehicle speed. Q-learning algorithms are commonly used for tasks that have a terminal state. However, it is possible to utilize them for problems without a terminal state such as continuous control tasks, similar to an autonomous vehicle. In the autonomous vehicle case, the agent's goal would be staying in the desired state for a long period of time, for example staying perfectly in the middle of the lane at a certain speed.

The initial idea to obtain these state information from the video game was to train a CNN. An illustration of such a system is given in Figure 1.2.

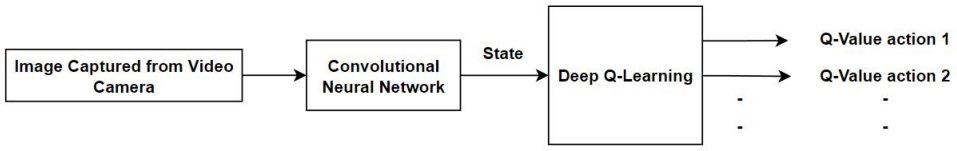


Figure 1.2 Illustration of initial environment setup

Since an autonomous vehicle in real life needs to observe state information depending on cameras and sensors, this would be the most realistic approach.

However, to train a deep Q-learning algorithm effectively, state information needs to be reliable. If state information is not reliable, the agent may be rewarded by the environment for taking wrong actions, which would then prevent algorithm from converging. Since CNNs' performance are not always 100% on the test set, the errors made by the CNN negatively affects the performance of the deep reinforcement learner. Therefore, in the training state, it is decided to directly extract state information from the video game engine rather than using the CNN outputs.

The aforementioned features offered by the video game have caused them to be used by many researchers. Shafaei, Little, and Schmidt (2016) showed that video games could be used to train and evaluate computer vision models, and that trained models could be assessed on real-world data. Using the same video game, Zou, Shi, Li, Zhang, and Shi (2020) evaluated the generalization performance of the model trained using video game data for remote sensing image segmentation and obtained promising results.

Filipowicz, Liu, and Kornhauser (2017) presented an AlexNet model to learn stop signs from the dataset generated in GTA V and investigated the success of the developed model on real-world data. It has been concluded that it is possible to transfer the models trained in the game to real-world applications. Using GTA V, Filipowicz (2016) demonstrated ways to extract video game data such as vehicle and pedestrian bounding boxes and distance of the car to lane markings for model training and creating tools to automate the processes. Richter, Vineet, Roth, and Koltun (2016) proposed a semantic labeling tool for extracting ground truths from GTA V without needing to access the source code. Martinez et al. (2017) showed that it is possible to generate datasets from GTA V that include the distances of the agent's car to the lanes and other vehicles.

Video games in general have also been used in various deep reinforcement learning research studies. Mnih et al. (2013) presented the first reinforcement learning model that can learn control policies directly from multidimensional sensor inputs. The model revealed is a CNN trained with a variant of the "Q-learning" algorithm. The model's

input is raw screen pixels and its output is a value function that predicts future rewards. The proposed model has been tested in six Atari 2600 games and its superiority over previous methods has been demonstrated. In addition, the model managed to reach a higher score than an expert player in three of the games.

Lample and Chaplot (2017) showed development of a reinforcement deep learning algorithm in a three-dimensional video game environment. The recurrent Q-learning algorithm developed for the FPS (First Person Shooter) game Doom has managed to outperform the game's own artificial intelligence.

Vitelli and Nayebi (2016) used a CNN trained with deep recurrent Q-learning algorithm for autonomous vehicle driving in a three-dimensional simulation environment. The article emphasizes that deep Q-networks are effective for an autonomous vehicle to learn navigation on the race track, but more successful results can be obtained by developing a reward function that better balances speed and vehicle stability.

CHAPTER 2
BACKGROUND

2.1 Machine Learning Algorithms

The research area of machine learning can be divided into three main learning methods. Reinforcement learning, supervised learning, and unsupervised learning.

Supervised learning algorithms use a large training data set in accordance with predetermined rules to train the artificial neural network model. The right action sets and input-output expectations that the model is supposed to learn are specified in this data set, and how well the model converges to these ideal outputs throughout the training process is evaluated. Two main problems can be resolved by using supervised learning algorithms.

- Classification problems, where mapping of inputs to discrete values is required. Outputs are usually classes or categories. An object detection problem may be an example of this.
- Regression problems, where the outputs are real numbers. Model is trained to understand the relations between variables. Regression models are helpful for forecasting numerical values, such as estimating a stock price at a specific time based on historical data.

Unsupervised learning on the other hand, is mainly trained on unlabelled data. They are able to determine hidden patterns between input data. Three main problems can be resolved by using unsupervised learning.

- Clustering problems, where grouping unlabeled data by their similarities and differences. Anomaly detection, market segmentation and social network analysis can be examples of this.

- Association problems, where relationship between different data is needed. Market basket analysis or medical diagnosis can be examples of this.
- Dimensionality reduction problems, where reducing the complexity of input data is required. It is mainly used for preprocessing purposes.

In order to complete the first part of self driving, the system should be able to process video footage from the video camera. For this purpose, CNNs will be used to process the screen images, which are the outputs of the video game. Convolutional neural networks, an important sub-branch of supervised learning, are artificial neural networks that can distinguish objects from photographs and video images and reveal dominant features. Although mainly used for classification problems, CNN models can be modified to resolve regression problems too. For the lane detection problem aimed to be resolved in this thesis, a trained CNN is expected to output 2-D position of lane markings on the image. Therefore, lane detection is approached as a regression problem.

Reinforcement learning algorithms, unlike other machine learning algorithms and methods, are algorithms used to train a piece of software called the agent how to take the right action with a trial and error method using feedback from its past actions and experiences in an interactive environment. Reinforcement learning works with dynamic environments rather than static data sets and its goal is to learn the action sets necessary to produce an optimal output. In doing so, they allow the agent to explore, interact with the environment, and learn (Sutton & Barto, 2018).

The agent monitors the current state of the environment and decides its actions based on the observed states. The environment changes its current state according to the action and generates a reward signal as a result of the action. Reward and the new state are taken from the environment by the agent and the agent decides the correctness of the action and whether it should be repeated or not based on the reward signal received from the environment. Observation, action and reward system is repeated until the learning is complete. Figure 2.1 shows a representation of environment-agent

interaction.

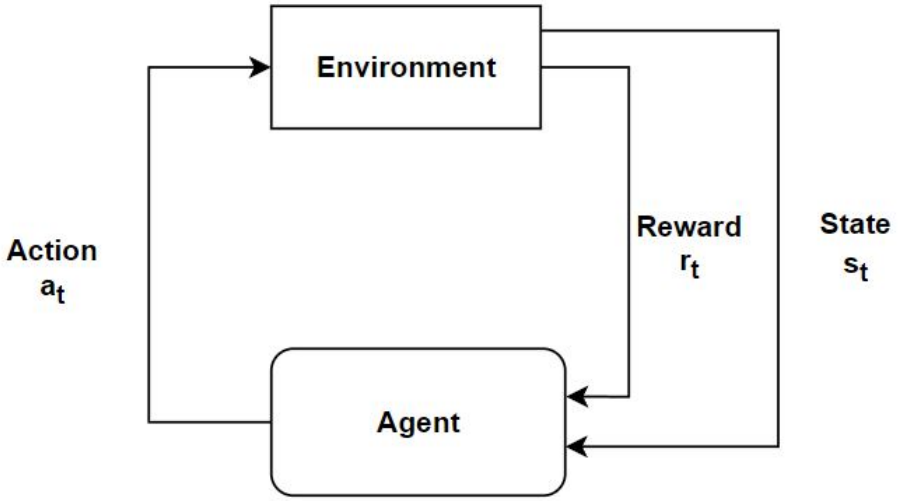


Figure 2.1 Basic visualization of reinforcement learning

The second part of self-driving is to develop a reinforcement learning model that makes use of CNN's outputs to actually derive the correct actions to be taken. The deep Q-learning algorithm is chosen as the reinforcement learning algorithm to derive the actions of the autonomous vehicle.

2.2 Q-Learning

In reinforcement learning, as shown in Figure 2.1, an agent may take an observation of states s_t, $s_t \epsilon S$, where S is the possible set of states that agent can observe from the environment at any time step t, any action a_t, $a_t \epsilon A(s_t)$, where $A(s_t)$ is the possible set of actions that agent can take in state s at time step t. The agent would then receive a numerical reward r_{t+1}, $r_{t+1} \epsilon R$, where R is the possible set of rewards and transition to a new state s_{t+1} as a consequence of its action.

A policy is a function ($\pi_t(s)$) that maps states to actions to determine the best course of action in a given state. Since reinforcement learning aims to maximize total reward over the long term, finding the optimal policy is required.

A reinforcement learning problem can be represented as a Markov Decision Process, in which the next state s_{t+1} is determined solely by the current state s_t and the current action a_t, not by previous states or actions. In model based reinforcement learning, it is assumed that information related to the environment is known probabilistically. The state transition function $P(s_{t+1}, s_t, a_t)$ gives us the probability of the environment being in state s_{t+1} in next time step if action a_t is taken in s_t. The reward received in this transition is then calculated by the reward function $R(s_{t+1}, s_t, a_t)$.

In order to adjust the affect of future rewards, a discounted total reward by discount factor (γ) can be calculated as in Equation 2.1. The discount factor determines how much importance is to be given to immediate rewards versus future rewards.

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.1}$$

Where r_t is the reward received at time step t and G_t is the total return. In this case, the optimal policy would be the one that maximizes total return.

A value function $V(s)$, depending on the policy function, determines how good it is for the agent to be in a particular state by calculating the expected discounted future reward.

$$V(s) = max_a E\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s \right) \tag{2.2}$$

Given the best policy for the agent, E is the expectation of total discounted future reward. From here, we come to an equation known as Bellman's Equation (Bellman, 1966).

$$V(s) = max_{\mathrm{a}} E\left(r_{\mathrm{t+1}} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{\mathrm{t+k+2}} | s_{\mathrm{t}} = s, s_{\mathrm{t+1}} = s' \right) \tag{2.3}$$

$$V(s) = max_{\mathrm{a}} E\left(r_{\mathrm{t+1}} + \gamma V(s') \right) \tag{2.4}$$

Given the best policy possible, our value function is the current reward $r_{\mathrm{t+1}}$ plus the discounted value function in the next state s'. This presents a naturally recursive solution to define the value function. The value function at state s is a function of the value function at the next state s'. Then optimal policy π^* can be extracted from the value function.

$$\pi^*(s) = argmax_{\mathrm{a}} E\left(r_{\mathrm{t+1}} + \gamma V(s') \right) \tag{2.5}$$

Where argmax function returns the best action that maximizes the expectation of discounted future rewards. The value function can be resolved for a model based reinforcement learning approach using methods such as value iteration or policy iteration, which are feasible for environments with a small number of states because the recursive process of creating a value table would become too demanding as the number of states increases.

For example in value iteration, estimate of the value function can be found by using Equation 2.6.

$$V(s) = max_{\mathrm{a}} \left(\sum_{s'} P(s'|s,a)((R(s',s,a) + \gamma V(s')) \right) \tag{2.6}$$

Where $P(s'|s,a)$ is the probability of next state being s' given that current state is s and action taken is a. This probability function can be described as the "model" of the environment for Markov Decision Process.

This type of learning is not feasible for the autonomous vehicle case, where the states of the environment would be described with parameters such as vehicle speed

and the car's position on the road lane (the angle of the car to the center point of the lane in our case), since the combination of these may be huge and iterating a value table over these would be cumbersome. Also, in the case of autonomous vehicle, we would not have a model of the environment to describe the probabilities of switching to certain states given certain actions.

In order to get rid of the dependency on a model, an action-value function, which is called as quality function, can be introduced. Quality function $Q(s,a)$ represents how good it is to take a particular action while the agent is in a particular state. It is the estimated maximum reward for a given state-action pair. According to Bellman's Equation optimal action-value function would be as below.

$$Q^*(s,a) = \sum_{s'} P(s'|s,a)((R(s',s,a) + \gamma max_{a'} Q(s',a')) \tag{2.7}$$

Similarly, optimal policy would be as shown in Equation 2.8.

$$\pi^*(s) = argmax_a Q^*(s,a) \tag{2.8}$$

From here we come to Q-learning, where learning the value function (quality function in this case) is done by Temporal Difference (TD) learning method. TD learning is closer to how biological learning occurs. The basic idea behind TD learning is to use the difference between the current action-value and next action-value to update the value estimate of the current action. The step size of the update is determined by a hyperparameter called learning rate (α). Learning rate should be selected with care since a larger learning rate may cause missing the optimal action-value and a smaller learning rate may cause the action-value to converge very slowly. On value function given in Equation 2.4, applying the TD learning we get Equation 2.9.

$$V'(s) = V(s) + \alpha\left(r + \gamma V(s') - V(s)\right) \tag{2.9}$$

Where V' represents the updated value function, V represents the current value function and term $r + \gamma V(s')$ is the target estimate function and the term $(r + \gamma V(s') - V(s))$ is TD error. Applying this on quality function (Q-function) we get Equation 2.10.

$$Q'(s,a) = Q(s,a) + \alpha\left(r + \gamma max_{a'} Q(s',a') - Q(s,a)\right) \tag{2.10}$$

As can be seen from the update rule in Equation 2.10, Q-learning does not use any probability function as in model based approach but uses a trial-and-error method. By taking a random action at each time without depending on an optimal policy, agent updates and improves quality function by calculating TD error. Therefore learning is done purely on experience, making Q-learning a model-free, off policy learning of quality function (Watkins & Dayan, 1992).

An "off-policy" approach in reinforcement learning can learn the value of a new policy, whereas a "on-policy" method in reinforcement learning only learns the value of the present policy. An on-policy technique adjusts the policy by moving in the direction of a greater expected return after learning the value function for the current policy. This implies that the agent will adhere to the present policy while learning, and that the current policy will be updated using the value function that was learned. On the other hand, an off-policy technique can be used to learn the value function for a different policy and apply it to the current policy. This implies that the agent does not need to follow the current policy while learning, and can instead learn from data generated by any policy.

On-policy methods are generally more stable and easier to optimize, but off-policy methods can learn from a wider range of data and are often more sample efficient. On-policy methods are usually used when the current policy is known and the goal is to improve it, whereas off-policy methods are usually used when it is difficult or impossible to follow the current policy and the goal is to learn a good policy from data generated by another process.

In Q-learning, the agent has to balance exploration and exploitation while selecting the right action for a given state. Utilizing current knowledge or policies to create choices that maximize the expected return is referred to as "exploitation." In order to advance the current state of knowledge or policy, "exploration" refers to the practice of obtaining additional information or testing out novel strategies. These two methods conflict with each other, since maximizing the expected reward often requires using prior knowledge, while trying out new actions may lead to suboptimal performance in the short term. However, the agent needs to try out new actions in order to be able to estimate an optimal action-value function.

One of the common methods to adjust exploitation versus exploration trade-off is by using ϵ - greedy action selection strategy. The agent selects an action with highest estimated reward with probability $1 - \epsilon$ or selects a random action with probability ϵ. ϵ is called exploration rate. This allows the agent to explore the given action space to improve the estimate of action-value while taking advantage of knowledge of actions with highest estimated values. In practice ϵ value is started from a higher value and decreases over time, allowing agent to try out new actions at the beginning of the training and explore more aggressively. Exploitation increases as training goes on and the agent gains confidence in the estimated action-value function.

Given these information regarding Q-learning, we can summarize the algorithm as shown in Figure 2.2.

The hyperparameters alpha (α) and gamma (γ) control the learning rate and the discount factor, respectively.

An approach known as experience replay can be used to ensure the algorithm's convergence (L.-J. Lin, 1992). For this, experiences of the agent, current state s, the current action a, next state s' and reward r are stored in a buffer. During training of the algorithm, a random sample from this memory can be introduced to calculate the loss, which is used to update the action-value function.

In situations where the agent faces non-stationary dynamics or when the data

Initialize $Q(s,a)$, the action−value function, to some initial values

for each episode:
  Initialize the state s from state space S
  while s is not terminal:
    Select an action a using an exploration strategy, ϵ−greedy
    Execute action a and observe reward r and next state s'
    Update the action−value function $Q(s,a)$ by following update rule:
      $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma max_{a'}(Q(s',a')) - Q(s,a))$
    Set $s \leftarrow s'$
end for

Figure 2.2 Algorithm of basic Q-learning (Watkins & Dayan, 1992)

distribution evolves over time, experience replay can be especially helpful.

Experience replay can aid in decorrelation of the training samples and increase the stability of the learning process by sampling from a fixed collection of events. It enables the agent to gain knowledge from transitions that might not be immediately accessible, like those that took place in a previously finished episode. Applying the experience replay method on basic Q-learning algorithm shown in Figure 2.2, we get the algorithm shown in Figure 2.3.

2.3 Double Q-Learning

An issue during Q-learning training can be encountered where estimated Q-values are higher than the actual Q-values. This phenomenon is named overestimation (Thrun & Schwartz, 1993). The agent can choose actions that aren't actually the best ones, which would result in a smaller reward, if the Q-values are overestimated. A suboptimal policy may be reached as a result of overestimation since the agent may become "stuck" in a local optimum with higher Q-values. Some factors can contribute to overestimation.

Initialize $Q(s,a)$, the action−value function, to some initial values
Initialize replay buffer D with capacity N

for each episode:
    Initialize the state s from state space S
    while s is not terminal:
        Select an action a using an exploration strategy, $\epsilon-$greedy
        Execute action a and observe reward r and next state s'
        Store (s,a,r,s') in replay buffer D
        Sample a batch of transitions (s,a,r,s') from D
        For each transition, update Q using the following update rule:
            $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma max_{a'}(Q(s',a')) - Q(s,a))$
        Set $s \leftarrow s'$
end for

Figure 2.3 Algorithm of Q-learning with experience replay

- Improper Q-value initialization: The Q-values may be prone to overestimation if they are initialized too high.
- Learning rate: A high learning rate may result in overestimation since the Q-values are updated too quickly.
- Transitions that occur in a correlated way may cause the Q-values to be exaggerated because of the temporal or causal relationships between the transitions. Experience replay may help to reduce this affect.
- Function approximation error can result in overestimation if the Q-function is estimated using a function approximator like a neural network.

Double Q-learning is an efficient approach to solving the overestimation issue (Hasselt, 2010). When using double Q-learning, the agent keeps track of two different action-value functions. In each step, one of the action-value functions are updated with the values from the other. This has the effect of decoupling the action selection and value estimation processes, which can help to reduce overestimation of the Q-values. Algorithm can be seen in Figure 2.4.

Initialize the Q−functions $Q1$, $Q2$

for each episode:

Initialize the state s from state space S

while s is not terminal:

Select an action a using the $\epsilon-$greedy action selection strategy for $Q1(s,a)$ and $Q2(s,a)$

Execute the action a and observe the reward r and the next state s'

with a probability p:

Find best next action $a^* = argmax_a(Q1(s',a))$

Update $Q1$ using the update rule:

$$Q1(s,a) = Q1(s,a) + \alpha(r + \gamma Q2(s',a^*) - Q1(s,a))$$

with probability $1-p$:

Find best next action $a^* = argmax_a(Q2(s',a))$

Update $Q2$ using the update rule:

$$Q2(s,a) = Q2(s,a) + \alpha(r + \gamma Q1(s',a^*) - Q2(s,a))$$

Set $s \leftarrow s'$

end for

Figure 2.4 Algorithm of double Q-learning (Hasselt, 2010)

Similarly to basic Q-learning, to break correlation between successive samples, experience replay can be introduced to double Q-learning. Algorithm then changes as shown in Figure 2.5.

2.4 Deep Q-Learning

The predicted Q-values for each state-action pair are typically stored in a table or matrix by traditional Q-learning algorithms. This is not practical in situations with a big or continuous state space since it could be difficult to store all the possible state-action pairings in memory. Using a supervised neural network, a non-linear function as the function approximator for Q-learning, $Q(s,a;\theta)$, where θ are the weights in the neural network, we come to deep Q-learning (DQN) (Mnih et al., 2015). In this case action-value function Q can be called a Q-network. Without the

Initialize the Q−functions $Q1$, $Q2$

Initialize replay buffer D with capacity N

for each episode:

Initialize the state s from state space S

while s is not terminal:

Select an action a using the $\epsilon-$greedy action selection strategy for $Q1(s,a)$ and $Q2(s,a)$

Execute the action a and observe the reward r and the next state s'

Store (s,a,r,s') transition in replay buffer D

if len(**D**) > **N**

Remove oldest transition from the buffer

endif

Sample a batch of transitions (s,a,r,s') from D

with a probability p:

Find best next action $a^* = argmax_a(Q1(s',a))$

Update $Q1$ using the update rule:

$$Q1(s,a) = Q1(s,a) + \alpha(r + \gamma Q2(s',a^*) - Q1(s,a))$$

with probability $1-p$:

Find best next action $a^* = argmax_a(Q2(s',a))$

Update $Q2$ using the update rule:

$$Q2(s,a) = Q2(s,a) + \alpha(r + \gamma Q1(s',a^*) - Q2(s,a))$$

Set $s \leftarrow s'$

end for

Figure 2.5 Algorithm of double Q-learning with experience replay

requirement to discretize the state space, DQN is able to train the Q-network directly from raw sensory input by employing a deep neural network as a function approximator. This enables the agent to adapt to situations with a wide or continuous state space and learn from high-dimensional sensory input. An illustration of the difference between Q-learning and deep Q-learning is shown in Figure 2.6.

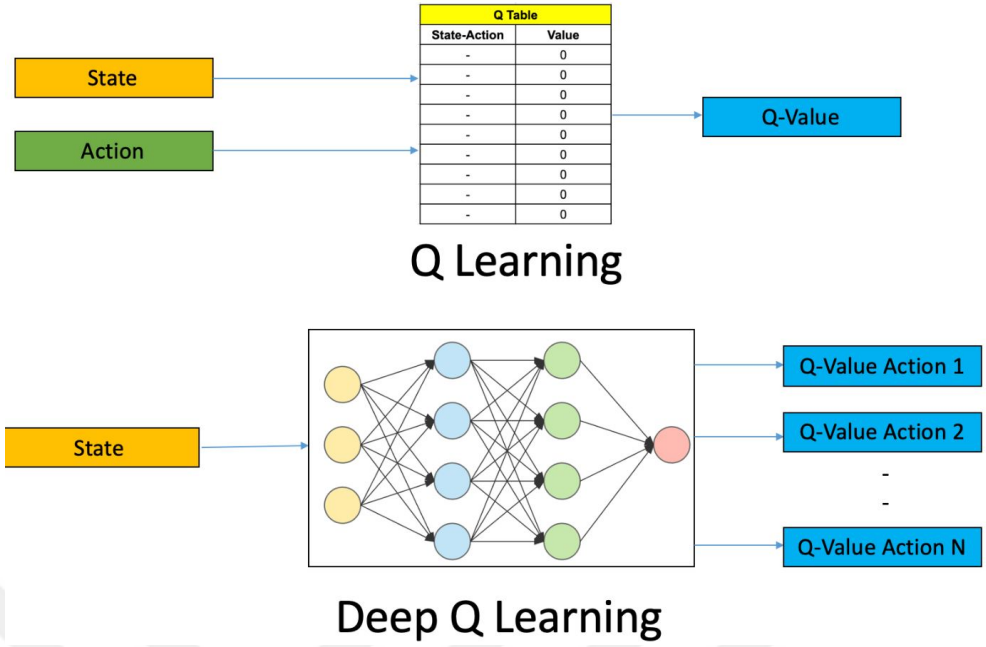


Figure 2.6 Difference of Q-learning and deep Q-learning (Choudhary, 2022)

$$L_i(\theta_i) = E_{s,a\sim p(.)}\left((y_i - Q(s,a;\theta))^2\right) \tag{2.11}$$

$$y_i = E_{s'\sim\epsilon}(r + \gamma max_{a'} Q(s',a';\theta_{i-1})|s,a)) \tag{2.12}$$

Q-network weights can be calculated by minimizing an error function $L_i(\theta_i)$ which changes in each iteration. Where i is the number of iterations.

Where $p(s,a)$ is the probability distribution over s and a sequences which is called behaviour distribution and y_i is the TD target at iteration i. Usually the behaviour distribution is selected by an ϵ - greedy strategy as shown for off-policy Q-learning algorithms in previous section. Algorithm of deep Q-learning can be shown as in Figure 2.7.

Deep Q-learning can be combined with double Q-learning with experience replay in order to reduce the overestimation problem (Van Hasselt, Guez, & Silver, 2016). In deep double Q-Learning (DDQN), two networks called target and online are used. The online network is used to select the actions, while the target network is used to evaluate the actions selected by the online network. The goal is to reduce the overoptimism that

can occur in traditional deep Q-learning, which can lead to poor performance. DDQN algorithm is shown in Figure 2.8.

Initialize the Q−network with random weights
Initialize replay buffer D with capacity N

for each episode:
    Initialize the state s from state space S

    for each time step:
        Select an action a using the $\epsilon-$greedy action selection strategy
        Execute the action a and observe the reward r and the next state s'
        Store (s, a, r, s') transition in replay buffer D

        if len(**D**) > **N**
            Remove oldest transition from the buffer
        endif

        Sample a batch of transitions (s, a, r, s') from D
        Set target y:
            if s' terminal:
                $y = r$
            else
                $y = r + \gamma max_{a'} Q(s', a'; \theta)$
        Perform a gradient descent step on $(y - Q(s, a; \theta))^2$ to update the network weights
    end for
end for

Figure 2.7 Algorithm of deep Q-learning with experience replay (Mnih et al., 2015)

Initialize online Q−network Q with random weights θ

Initialize target Q−network Q' with random weights θ'

Initialize replay buffer D with capacity N

for each episode:

Initialize the state s from state space S

for each time step:

Select an action a using the $\epsilon-$greedy action selection strategy using Q:

$a = argmax_a Q(s, a; \theta)$

Execute the action a and observe the reward r and the next state s'

Store (s, a, r, s') transition in replay buffer D

if len(**D**) > **N**

Remove oldest transition from the buffer

endif

Sample a batch of transitions (s, a, r, s') from D

Set target y:

if s' terminal:

$y = r$

else

$y = r + \gamma max_{a'} Q'(s', a'; \theta')$

Perform a gradient descent step on $(y - Q(s, a; \theta))^2$ to update weights theta

Every C steps set $\theta' = \theta$

end for

end for

Figure 2.8 Algorithm of deep double Q-learning with experience replay (Von Dollen, 2017)

2.5 Convolutional Neural Networks

Convolutional neural networks are a type of neural networks that specialize in processing data with a grid-like topology, such as images. They are well suited for image classification tasks as they are able to automatically extract features from the input data. This is achieved by using convolution operation using a window called kernel or filter. This operation enables them to identify patterns or features in images. CNNs are commonly used for tasks like image classification, object detection and segmentation.

Kernel is a small matrix of weights that is moved through the image while calculating the dot product between its parameters and input data. This operation is called convolution. If a kernel is designed to detect edges on an image, output of convolution is a higher value where an edge is present in the input and vice versa. Output of the convolution layer is called a feature map. Using these feature maps, classification of images or object detection tasks are possible. During the training weights of the kernel is learned. Visualization of the convolution operation is shown in Figure 2.9.

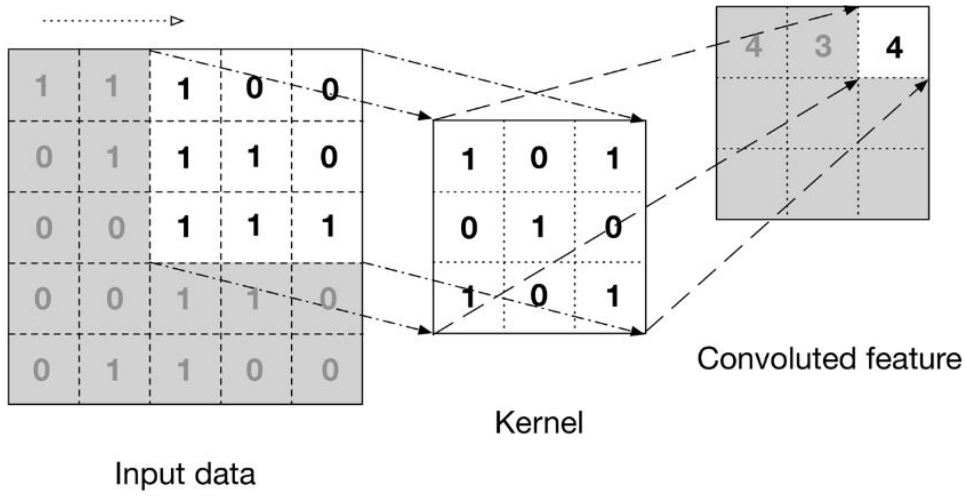


Figure 2.9 Visualization of the convolution operation (Manav, 2022)

Three feature maps are produced when the kernel is shifted among the channels for RGB inputs. These matrices are combined to create the feature map of the image.

Convolutional layers are usually followed by a pooling layer. Pooling layer is used to down sample the feature maps, allowing the network to focus on the most important features and also reducing the computation costs. They also help with improving generalization performance by reducing overfitting. There are two types of pooling layers.

- Max pooling, output of the pooling layer is the maximum value in the input layer, framed by a window. This method is more robust against noise and outliers in the input data.
- Average pooling, returns the average of the values within a window. This method acts as a noise suppression mechanism.

Last type of layers used in CNNs are fully-connected layers. By connecting each neuron in the layer to the previous layer's nodes, they allow the network to learn complex correlations between input and output data. During the training phase, the weights of the connections between neurons are learned. Typically, fully-connected layers are applied towards the end of the network and after convolutional and pooling layers in a CNN. They are used to incorporate the information that the earlier layers have learned in order to forecast or classify data. Due to large number of parameters to be learned, training CNNs are computationally expensive and often requires specialized hardware such as a GPU. Example of a CNN is shown in Figure 2.10.

Activation functions are used in CNNs to introduce the non-linearity to the network, without them network would only learn linear relationships between the inputs and outputs. Several commonly used activation functions are ReLU, sigmoid and softmax functions. Each of these functions have different characteristics that are suitable for different kinds of tasks.

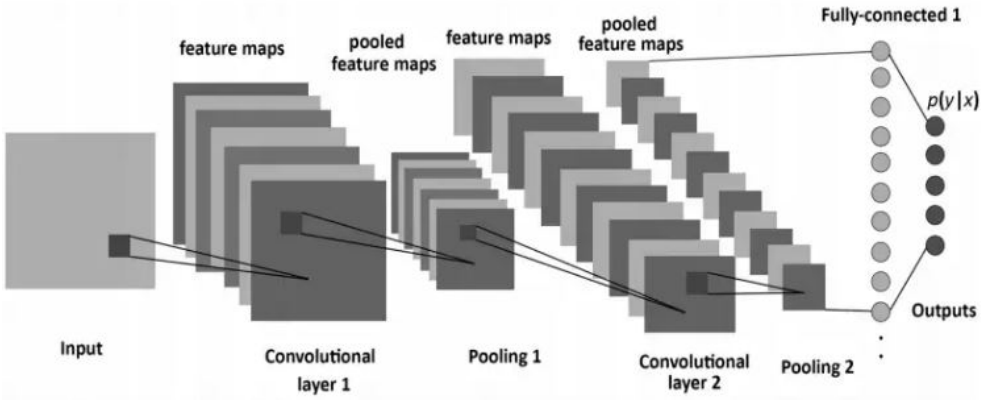


Figure 2.10 Visualization of an example CNN (Stewart, 2022)

Training CNNs usually require a large amount of labeled data, images and their corresponding labels for example. Network's weights are adjusted by minimizing a loss function over the training data. Loss function is a measure between network's predicted output and actual labeled output. Some of the loss functions commonly used are mean squared error (MSE), binary cross-entropy and categorical cross-entropy. Selection of the loss function, similar to activation functions, is dependent on the nature of the task. MSE is commonly used for regression problems whereas binary cross-entropy is used for binary classification problems.

Steps of training a CNN can be summarized as below.

- Initiating the weights of the network.
- Introducing input from the dataset and calculating the loss function between predicted outputs and actual outputs.
- Backpropagating the error, calculating the gradient of the loss function with respect to weights.
- Updating the weights depending on an optimization algorithm.
- Running training until the loss function converges to a desired value.

Before the training starts, dataset is separated into three subsets, training dataset, validation dataset and test dataset. Each of these datasets contain unique data. During

the training weights are updated according to the loss function calculated over training dataset. After each epoch, updated weights are run across samples from validation dataset, results are used to evaluate the generalization capabilities of the network. Lastly, test dataset is used after the training is complete in order to evaluate overall success of the network. This process is used to prevent CNN from overfitting, a phenomenon observed when CNN is able to perform better on training dataset but not on validation and test datasets. Increasing the dataset size, reducing model complexity and stopping the training process earlier may help to reduce overfitting problems.

Training CNNs may take huge amounts of time even on best hardware due to being computationally expensive and requirement of large datasets. In order to reduce the training times and lower the amount of data required, transfer learning methods can be used. Transfer learning is basically re-purposing a pre-trained CNN for a new task. Since the pre-trained model may have learned features related to the new task, it can be re-used by training only top layers of it on a small amount of newly collected data.

Several pre-trained CNNs such as InceptionV3, Xception, VGG16 and ResNet50 are re-purposed for lane and vehicle position detection task in the scope of this thesis work using transfer learning method.

2.5.1 InceptionV3

InceptionV3 is a CNN architecture developed for image classification problem based on original Inception model (Szegedy et al., 2015). Model uses multiple "inception" blocks which consists of convolutional and pooling layers. Main idea of the Inception model is to use multiple different sized kernels. Separating larger kernels into smaller ones is called factorization. This helps to capture details in a wider level. Szegedy et al. (2015) also suggest using a 1x1 kernel before using larger ones which helps to reduce number of parameters in the network by applying dimensionality reduction. This is illustrated in Figure 2.11. In order to reduce

parameters even further, max pooling layers are used and the fully connected layers at the end are replaced with a global average pooling layer. Compared to AlexNet's nearly 63 million parameters, InceptionV3 only has nearly 24 million (Krizhevsky, Sutskever, & Hinton, 2017).

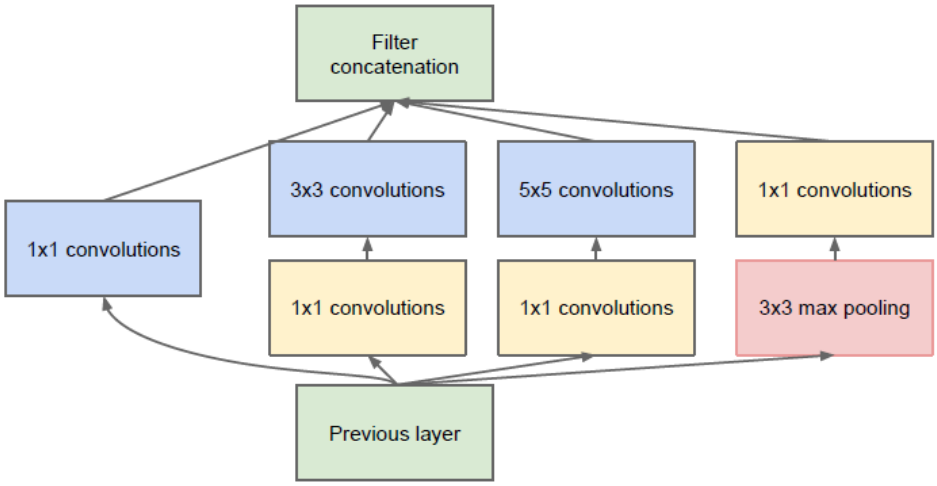


Figure 2.11 Visualization of an Inception module (Szegedy et al., 2015)

InceptionV3 was trained on the ImageNet dataset, which consists of millions of images classified with 1000 different classes. On the ImageNet classification challenge, it produced cutting-edge results, and it has been extensively employed in numerous other applications, including object detection and image segmentation. Summary of the model is shown in Figure 2.12.

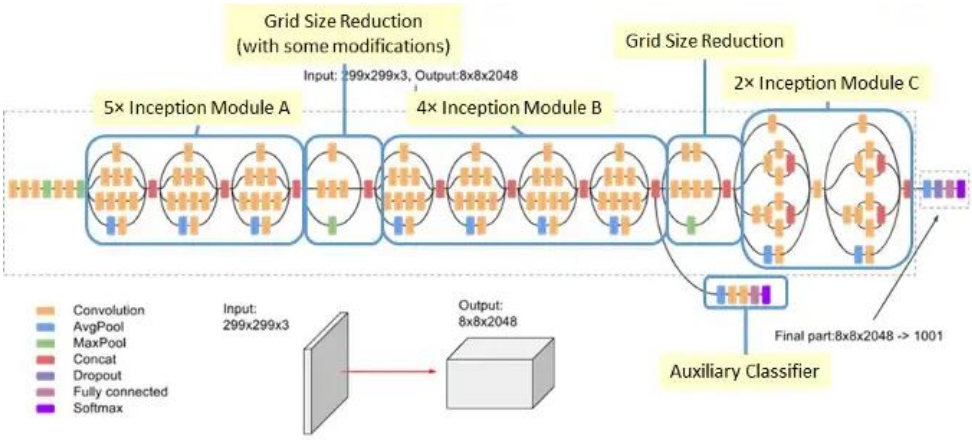


Figure 2.12 Visualization of an InceptionV3 network (Tsang, 2022)

2.5.2 Xception

Inspired by InceptionV3, Chollet (2017) presented Xception network architecture using depthwise separable convolutions. Depthwise separable convolutions consists of spatial convolution applied on each channel independently, followed by a pointwise convolution. Depthwise separable convolutions are more efficient than standard convolutions because they require fewer parameters and computations, making them faster and more efficient to compute. Xception reverses the order of depthwise separable convolution by first applying the pointwise convolution (1x1 kernel) as it is done in Inception modules.

Xception architecture outperformed InceptionV3 in ImageNet dataset and has slightly lower number of parameters. Architecture of the network is given in Figure 2.13.

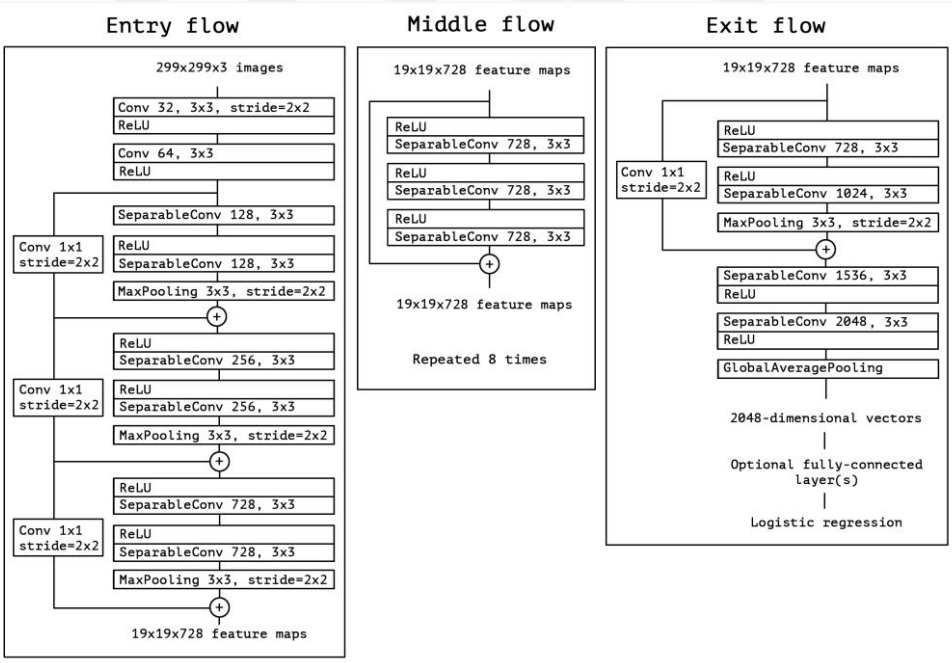


Figure 2.13 Visualization of an Xception network (Chollet, 2017)

2.5.3 VGG16

VGG16 model consists of 16 layers, 13 convolutional layers and 3 fully-connected layers. The convolutional layers are organized into 5 blocks, with each block consisting of 2 or 3 convolutional layers and a max pooling layer. The first block uses 64 filters, the second uses 128 filters, the third uses 256 filters, the fourth and fifth blocks use 512 filters. The two fully connected layers at the end have 4096 units each and last fully connected layer has 1000 units for ImageNet classification. Architecture of VGG16 relatively simpler than previously mentioned networks as it consists of linearly stacked convolutional and fully connected layers. Compared to previous methods VGG16 has also a larger number parameters around 140 million. Architecture of the network is shown in Figure 2.14.

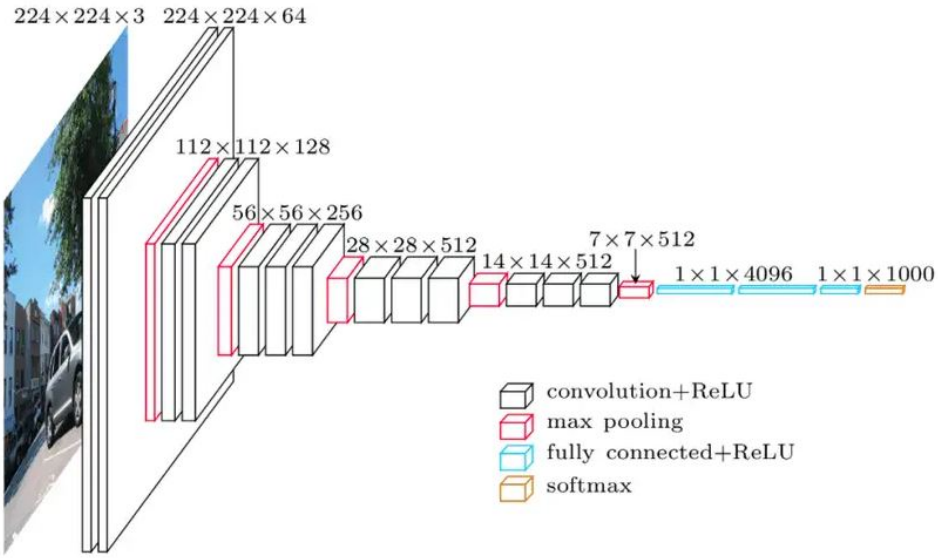


Figure 2.14 Visualization of a VGG16 network (Džakula & Bezdan, 2019)

2.5.4 ResNet50

ResNet architecture consists of 152 layers (He et al., 2016). It uses residual blocks which consists of a series of convolutional layers followed by a skip connection. A skip connection allows the gradient to bypass one or more layers. Skip connections are

typically implemented as a simple element-wise sum of the input to the skip connection and the output of the skipped layers. This makes it easier for the network to learn deeper architectures and helps to reduce the vanishing gradient problem, which is a phenomenon that can occur when training very deep networks and can make it difficult for the network to learn. As networks get deeper the affect of the gradient in earlier weights get lower, leading to a slower learning.

ResNet50 is a variant of ResNet that is trained on ImageNet dataset. It consists of 50 layers organized in 5 stages consisting of convolutional layers and shortcut connections. 50 layer configuration has 3, 4, 6 and 3 residual blocks and each residual block consists of three convolutional layers. ResNet network architecture is given in Figure 2.15.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				

Figure 2.15 Visualization of a ResNet50 network (He et al., 2016)

CHAPTER 3
DETECTION OF ROAD INFORMATION

While collecting images from a video game is simpler, manually labeling lane borders on each image is nearly impossible. To streamline the labeling process, the initial idea was to use more traditional and widely used methods such as Canny Edge Detection and Hough Transform (Canny, 1986; Hough, 1962). To partially automate the labeling process, a tool that can modify the parameters of these methods in real-time is used (Gale, 2022). The tool captures one image from the game's window and asks user to modify parameters used in the algorithm. However, visual inspections indicated that under different lighting conditions and when lanes are only partially in the scene, the parameters of these methods need to be re-adjusted to yield acceptable results. In most complicated traffic scenes, the lanes could not be detected by these methods. Even if the method was fully successful in its task, it would take a huge amount of time to manually edit parameters and save thousands of images. Therefore, it was decided not to follow this method and find another way to fully automate dataset generation. An example use case of the tool is shown in Figure 3.1.

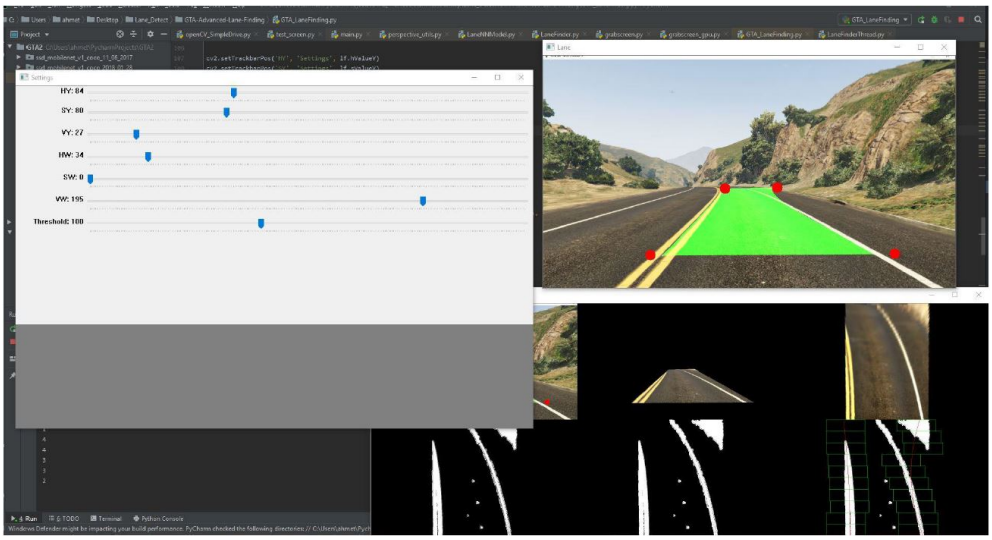


Figure 3.1 Semi-automated labelling tool

Using third-party tools such as Script Hook V and Rage Plugin Hook, it is possible to extract information directly from the video game's engine (Blade, 2022b; Rage

Plugin, 2022). Rage Plugin Hook connects to the game instance and acts as an interface between the game's engine and custom user codes. It also provides an in-game menu to facilitate the dataset collection process such as teleporting the player character to a certain point in the world, changing weather and time conditions, and spawning a vehicle. With the help of this menu, it is also possible to hide the game's HUD (heads-up display) to obtain more realistic images. Script Hook allows using any .NET language to develop scripts that run in the game. It provides a huge database that includes APIs (Application Programming Interface) of the game's engine, researched and reverse-engineered by the game's fan base (Blade, 2022a). Native APIs from the game engine can provide vital information about where the lanes are placed and their 3D world coordinates within the game. By converting 3D world coordinates to 2D screen coordinates, a huge dataset that consists of images and the coordinates of four key points can be collected. These points are:

- The angle of the vehicle to the road,
- Left lane boundary,
- Right lane boundary, and
- Center of the lane.

3.1 Extracting Game Data

The virtual world of GTA V is filled with various types of AI-controlled vehicles such as passenger cars, busses, trucks, and motorbikes. Hundreds of such entities can surround the player's character at any given time in virtual traffic. These vehicles behave similarly to real life, they can be slower or faster and change lanes. They are controlled by in-game pathfinding algorithms and such functions can be exploited to understand how the roads are detected for the AI vehicles.

The two main pieces of information used are nodes and links. Nodes are individual points scattered around the virtual world. They are placed in the middle of roads and

connected with entities called links. The links contain information about the type of road (highway, urban, seaway, etc.) and other information such as the number of lanes on the road and the width of each lane. A game file called “paths” keeps the information about nodes and links, along with their 3D world coordinates. By parsing this file, one can determine how pathfinding information is scattered in the virtual world.

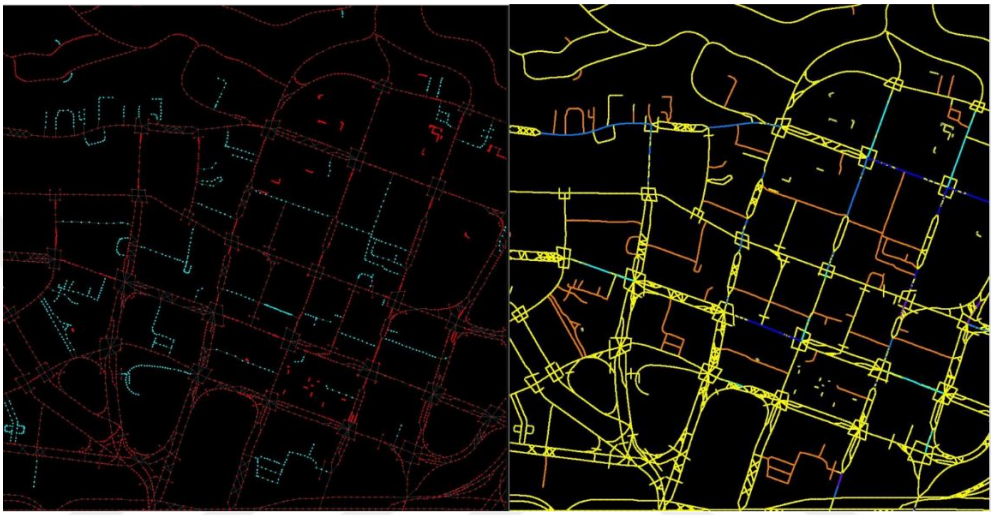


Figure 3.2 Nodes (left) and links (right) shown on game’s map

Using auxiliary functions such as GET_NTH_CLOSEST_VEHICLE_NODE_ID, it is possible to detect where the player’s character is in the virtual world and how close the character is to a vehicle node or vehicle link. Since nodes and links are labeled with unique IDs, it is possible to detect the properties of the road the player is currently driving. From the road type, lane width, and the number of lanes, the exact positions of left, right, and middle lane points are determined in real-time. By using vector calculation APIs such as GET_ANGLE_BETWEEN_2D_VECTORS, even the angle of the vehicle’s forward vector to the road center can be calculated. Calculated ground truths can be displayed in the virtual world for debugging and visual inspection.

3.2 Automated Data Collection Script

In order to develop a fully automated dataset collection process, a rewarder script written in C++ on a publicly available project called DeepGTAV is re-written and significantly extended for multiple lane point detection in C# language, using the native functions and tools mentioned (Ruano, 2022). For each script to be developed for Rage Plugin and ScriptHook, a main function with certain keywords has to be defined.

- OnTick: A periodic function that keeps running when the developed plugin is activated within the game console.
- OnKeyDown: A function that runs when a certain key is pressed.
- OnKeyUp: A function that runs when a certain key is released.

As the collection process is meant to be continued until terminated by the user, OnTick function is used. The main function creates objects of player's characters and their vehicle to determine their positions on the world map. It also handles the local directory to save the dataset and period for saving individual data. When first initiated from within the game, the function loads all vehicle nodes on the entire map and makes a call to "populateNodes" function.

populateNodes function creates a large database in the shape of dictionaries by parsing paths XML file. This database consists of flags and locations for nodes and links. For nodes, information are stored such as 3-D world coordinates, node unique IDs, flags to indicate if a node is placed on a highway or a tunnel etc. For the links, information is stored such as link unique IDs, two reference IDs to nodes that the link connects, width of the road and number of lanes.

Another function called setLinePoints goes through each stored link and using the width and number of lanes information, calculates where each road line resides for that particular link. Therefore for each link, database also stores lane marking locations in terms of 3-D world coordinates.

Every function explained until this point except OnTick executes only once when the script is initiated. OnTick function is configured to only call populateNodes function once to generate the database. After that, main function cyclically calls DatasetCollect function with a configurable period, which first detects vehicle's current coordinates and goes through several nearby nodes. From nearest detected nodes, link information is reached, along with each link's line points from previously generated database. Comparing the detected links' direction with the vehicle's forward vector (a unit vector showing the vehicle's direction), only line points that are on the vehicle's lane are detected.

DatasetCollect function also calculates the angle between vehicle's forward vector and vector created between lane center point and one of the lane points using GET_ANGLE_BETWEEN_2D_VECTORS function. This information shows how far vehicle is facing away from the lane center.

Detected line points are in 3-D world coordinates, they are converted to 2-D screen coordinates using native function WORLD3D_TO_SCREEN2D.

Lastly, script saves the label file containing line point coordinates and vehicle's angle to the road center, along with a captured image from the video game.

3.3 Data Collection

By default, the video game does not provide a 'dashboard camera'. To provide that view to the player another third-party tool called Hood Camera is used (Hood Camera, 2022). To further speed up the dataset collection process another script named Self Drive Plugin is used (Self Drive Plugin, 2022). This script allows placing a way-point on the game's map and when the initiation button is pressed, the car is driven to the waypoint with customizable speed. The script uses in-game functions similar to the developed dataset collection tool and the player's vehicle behaves like any other AI-controlled vehicle in the game.

The video game is run at the highest possible graphics settings to create more detailed and realistic visuals, but at 800x600 resolution to reduce the size of the dataset.

The dataset collected in this study focused on highway driving in the middle of two lanes on pre-defined parkour. By the use of the developed and above-mentioned complementary scripts and tools, a dataset collection procedure is defined. After starting the game, a vehicle is spawned using the Rage Plugin Hook tool. The view is switched to the dashboard camera. Both time and weather are set to dynamic mode, meaning the weather is changed randomly as time passes by. The vehicle is teleported to the starting point of the parkour on the highway. A waypoint is placed at a point on the parkour and the Self Drive Plugin is initiated, each time with different speeds so the captured frames are diversified as much as possible. The developed dataset collection script is initiated via the in-game console provided by the Script Hook V tool. Once the waypoint is reached, another one is placed automatically on the parkour, and so on. A folder called 'dataset' is automatically created at the beginning of the process which holds the image files in jpeg format and corresponding text files that hold the labels as shown in Figure 3.3.



Figure 3.3 Example of image-label pair from the dataset

3.4 Data Analysis

By using the in-game entities such as markers as seen in Figure 3.4 and subtitles or help text displayer functions of the game's engine, an analysis of the developed scripts is conducted.



Figure 3.4 Road lanes marked in game

By manually driving the vehicle in the virtual world, visual inspections are done for placed markers, and the vehicle's angle to the road is displayed as a subtitle. When driving straight on the highway, it is seen that the script can generally detect 4 or 5-point pairs (left-right lane boundaries) in front of the vehicle. When the range of nodes and links that are considered as inputs to the script is increased, it is possible to determine more pairs of points belonging to lanes. However, this results in incorrect lane detection when the vehicle passes near a connector road or where the node-link density is lower within the vehicle's range. This is because the game has a larger number of nodes and links on the connector roads to provide more paths to AI-driven vehicles. Since the developed script only considers a predetermined number of nodes

and links from the nearest to the farthest, having many nodes and links clustered on the path causes confusion, thus making lane detection difficult. So it was decided to use only one pair of lane points in front of the vehicle, which is correctly detected almost in every label file.

Since GTA V is not a research tool, debugging and developing the script is a task that must be done using in-game assets such as the markers shown in Figure 3.4. As a result, developed scripts have some bugs that cause some nodes-links to be missed or not detected at all, especially when a vehicle is turning on a highway and the nearest node is not visible on the screen. Sometimes nodes are registered multiple times, which may be caused by the existence of several links yielding the same positional information of the lanes. Depending on the position of the vehicle, the developed script may detect one lane point (left or right boundary) earlier (registered as Point0 in Figure 3.3) than another one. This may cause confusion during training because Point0 may be a right lane border in one label file and a left lane border in another.

To reduce the effects of inconsistencies in the training, a label post-processing tool is developed in Python. Tool removes multiple detected points and compares X-axis information within detected points and orders left-right lane boundaries in a way that Point0 output is always the left lane boundary where Point1 is the right lane boundary. If the nearest node is not visible on the screen and therefore not logged in the label file, it is removed from the dataset. Finally, post-processing converts the text files in the dataset folder into a single csv file, including the ID (which corresponds to the file name), angle to the road center, left and right lane points, and the center lane point.

CHAPTER 4
REINFORCEMENT LEARNING AGENT

4.1 State Space

As explained in the previous chapters, since training of deep Q-learning algorithms require highly reliable data and due to CNN models' inability of producing perfectly correct outputs all of the time, it is decided to directly extract data from the video game environment in the training phase of the autonomous agent.

Data extracted from the video game engine consist of state observations and a numerical reward signal. State observations in our case are vehicle's speed, its angle to road direction, acceleration value and its distance to the center of the road. Automated data collection script developed in scope of this thesis work is edited to facilitate such information. DatasetCollect function of the script is renamed as ComputeReward which now produces these data using in-game functions.

Similarly to dataset collection script, the angle between the road's direction and vehicle's forward vector is calculated. However in dataset collection version, script calculates the angle between two vectors. First vector is between the center of the road and closest lane point detected and the second vector is vehicle's forward vector. This results in a 90 degree angle when vehicle is directly facing the center of the road. This causes some problems when vehicle is not exactly within the lane points. This is not a problem for the dataset collection script since the vehicle is perfectly driven by the in-game AI during dataset collection but it is a problem for deep learning training because the vehicle will also be taking arbitrary actions that cause lane changes or moving in between lanes. In order to avoid this, ComputeReward function is edited to calculate the angle between closest link's direction vector and vehice's forward vector. With this change, when vehicle is facing perfectly to the road's direction, even if it is not within a lane, angle is measured as 0 degree.

ComputeReward function calculates the vehicle's speed directly by using in-game

functions. By using the angle to the road, lateral and longitudinal components of the speed is also calculated. Vectors used for extracting state information is visualized in Figure 4.1.

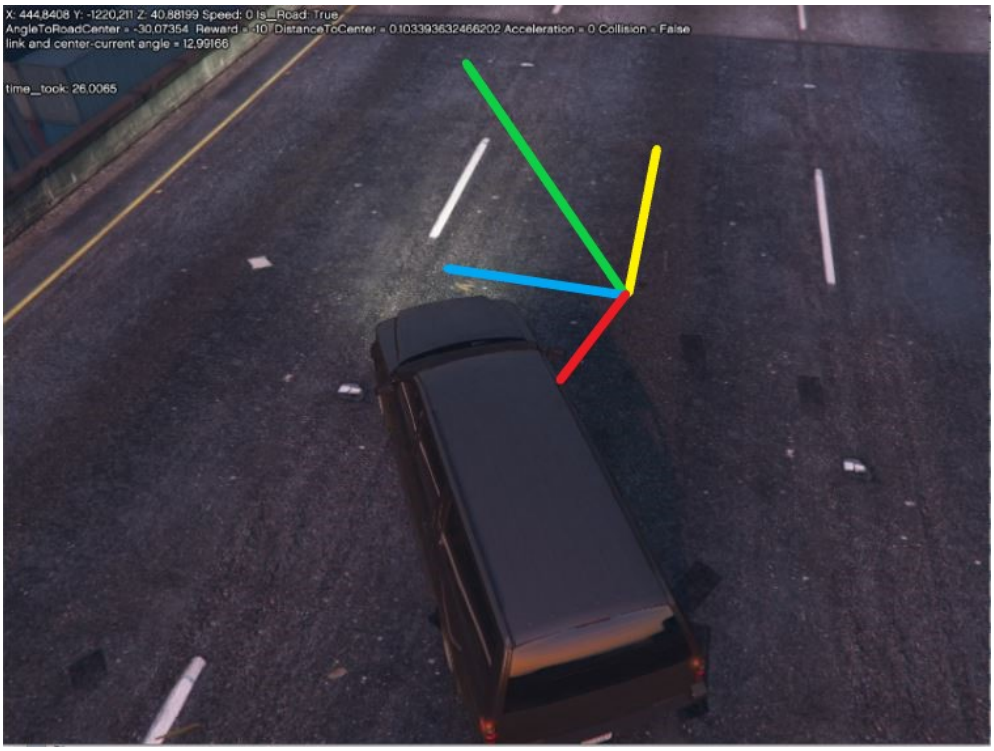


Figure 4.1 Visualization of vectors used for the state information

In Figure 4.1, green vector is the vehicle's forward vector extracted directly by using native functions. Forward vector is normally a unit vector pointing in the direction of the vehicle, in the figure it is scaled up and origin is placed on the center of the road for clarity. Yellow vector is extracted from paths file which shows the direction of the link that is closest to the vehicle's position. One of the state observation is directly the angle between green and yellow vectors. Road angle is a negative value when vehicle is directed to the left of the road's general direction and vice versa. This is also useful for reinforcement learning agent in order to decide if turning left or right is better for aligning with the road direction. Road angle information is scaled with a value of 90 degrees to ensure it remains between 0 and 1, in order to ease the training of the Q-network.

Another state observation is the vehicle's distance to the road center. This value is

not directly extractable from the video game engine as the road lanes are not directly considered as unique objects in the game's code. Also, since the links are scattered on the road with a distance between each other, it is not possible to directly use the distance between vehicle's position and one of the lane points detected. Therefore the red vector shown in the Figure 4.1 is used. Using the angle between red and yellow vectors with the size of the red vector and trigonometry, it is possible to manually calculate the distance of the vehicle to the road's center line. Similarly to road angle information, distance to road center information is also scaled with a value of 10 to ensure it remains within the range of 0 and 1.

Last bit of useful state information is the acceleration flag. This flag is directly extracted from the video game engine using native functions. Flag has a value of 1 while vehicle is moving forward and a value of -1 while vehicle is reversing.

In total four variables, angle to road direction, vehicle speed, distance to the road center line and acceleration flag are extracted from the environment with high accuracy.

4.2 Action Space

When a human is driving in GTAV, actions such as steering, acceleration and breaking are handled by four discrete inputs. These are key presses to four keyboard buttons, W, A, S and D. In order to allow the reinforcement learning agent to interact with the environment, these key presses need to be simulated. This cannot be achieved directly by using Python packages such as PyAutoGui since the video game engine expects the key press inputs to be scan codes. A wrapper written in an open source project is used in order to simulate these (Kingsley, 2022). With this code, it is possible to simulate key press and release actions separately.

For the driving task, the time between a key press and release is crucial since while vehicle is moving slowly it is needed to press a certain key longer to make a valuable impact to the vehicle. On the contrary, while vehicle is moving faster even a very short

key press can have a huge impact in terms of vehicle's stability and general direction.

Preliminary trainings of the agent showed that when action space consists of longer key presses such as half a second, agent is not able to stabilize the car in the lane. When action space is set to much shorter key presses such as 10 milliseconds or 1 millisecond, vehicle stability improved. However this resulted in agent not being able to recover from erroneous states. Similarly having only four separate key presses showed lower average speeds and increased the time it takes for the agent to re-center the vehicle to the road lane.

Therefore, selection of optimal key press durations can be considered as another hyperparameter for GTAV agent. In order to reduce the affect of the key press durations, it is decided to use multiple combinations of four keys as well as different press durations.

Final action space is displayed in Table 4.1. With these combinations total of 32 discrete actions are used during the training of the agent.

Table 4.1 Action space and key press durations

Press Duration	**Key to Press**
0.005s	W or W+A or W+D or A or D
0.01s	W or W+A or W+D or A or D
0.05s	W or W+A or W+D or A or D or noPress
0.1s	W or W+A or W+D or A or D or S
0.25s	W or W+A or W+D or A or D
0.5s	W or W+A or W+D or A or D

4.3 Reward Signal

Reward signal is directly generated by the script that runs within the game. ComputeReward function described in previous sections uses the state observations, angle to road (θ), distance to road center line (d), vehicle speed (V) and acceleration

(a) to calculate the numerical reward signal.

Using native functions V and a are directly obtained from the game engine. Speed's longitudinal (V_x) and lateral (V_y) components are calculated using θ value. A common reward function in literature that rewards for longitudinal speed and punishes for lateral speed is used to generate the reward value as shown in Equation 4.1 (Pérez-Gil et al., 2021).

$$R = \sum_{t} |V_t cos\theta| - |V_t sin\theta| - |V_t||d| \tag{4.1}$$

A punishing reward of -10 is applied if vehicle is in collision with another object, acceleration flag is -1, speed value is below 1 m/s, vehicle is not on a lane, it is moving in opposing direction to the road or speed value is above 20 m/s.

4.4 Environment Setup

In order to train the reinforcement learning agent using Python first a simulated environment needs to be created. In this study, OpenAI Gym is used to simulate the GTA V environment (OpenAI, 2022). Developed Gym environment consists of two main functions, step and reset.

In step function a random action from the action space is selected and applied with key press wrapper. The results of the action (new state observations and a reward value) is collected from the script running in the game. Episode counter of 1000 steps is reduced by 1. At the end of the function, new states, reward signal and a done flag is returned to the training agent. Done flag is only set if episode counter is reached to zero or a reset request is received from the script in the game.

Done flag is sent by the script that is running in the game and is set when vehicle is in collision with another object, acceleration flag is -1, speed value is below 1 m/s, vehicle is not on a lane, it is moving in opposing direction to the road or speed value

is above 20 m/s. Once done flag is set, reset function is called by the training agent, episode counter is reset to 1000 and a reset request is sent to the script in the game. Script moves the vehicle to the starting position and another episode of training is automatically started.

Script in the game calls a ServerStart function each cycle which creates a pipe using Windows' backend and waits for a client for 20 milliseconds. Once a client is connected, state observations, reward signal and reset flag is written to the pipe as a string separated by new line characters. Gym environment similarly tries to connect to the pipe without a timeout limitation so that if video game crashes during the training, environment hangs until communication is available again once the game is re-launched. String containing the information is decrypted and converted to floating point numbers. Lastly environment sends its own reset flag using the same communication line.

By enabling the communication between script running in the game and Gym environment, a robust way of data transfer to be used for reinforcement learning agent's training is established.

CHAPTER 5
RESULTS

5.1 CNN Model Training

For the detection of lane points, four different models such as ResNet50, VGG16, Xception, and InceptionV3 are selected. These models are trained on ImageNet Dataset and Xception model showed the best Top-5 accuracy. This study uses transfer learning methods on these models to provide a comparison and select the best suitable model for the lane detection task. To fit the purpose of lane detection, some modifications are done to the models. The input layer is rescaled to 224x224x3 pixels, and a fully connected layer consisting of seven neurons is added as the output layer of the model.

The outputs are, the angle of the vehicle to the road, the coordinates of the center, the coordinates of the left lane boundary, and the coordinates of the right lane boundary. The sigmoid activation function is used at the output layer to scale the values between 0 and 1. The parameters of the last layer are updated in the training phase.

All the images are rescaled from 800x600x3 to 224x224x3 and all labels are re-scaled with the same ratio before normalization. The angle to the road is divided by 180 degrees for normalization. Therefore, all the output values are between 0 and 1 before starting the training.

Trainings on all models are carried out by freezing all base model layers and updating weights on the newly added fully connected layer on the top. It is done using Adam Optimizer with a pre-set learning rate of 0.00001 and 60 epochs. Training is done on RTX 3080 graphics card in a total of 18646 batches for one epoch. Since the batch size of 32 images does not fit in the 10 GB video ram of the graphics card, a manual input generator that reads the data from the disk and shuffles the inputs is implemented.

5.2 CNN Training Results

Mean squared error is used as the loss function and the results of the first training are given in Table 5.1. Results showed that the best performing model is ResNet50 in terms of low validation loss, therefore it is selected to be used after a second training for fine-tuning is carried out.

Table 5.1 Results of the first trainings

Model	**Training Loss (e-3)**	**Validation Loss (e-3)**	**Training Duration**
ResNet50	8.7169	8.8481	14h 24m
Xception	9.4252	9.5401	14h 8m
InceptionV3	9.6134	9.7153	12h 26m
VGG16	9.9814	9.932	13h 36m

The first 165 layers of the ResNet50 model are frozen and the remaining last block of the base model is joined in the training. Similarly, Adam Optimizer is used but this time learning rate of 0.00001 is decreased in every 10 epochs by multiplying with $(0.85)^{epoch/10}$. The second training is carried out for 50 epochs and took approximately 12.5 hours and resulted in a validation loss of 3.4737e-3 which is significantly lower than the initial training.

Using the developed tools, a test dataset consisting of 754 images is collected on a completely different highway from the one used for the training and validation datasets. When the trained model is evaluated on the test dataset, the loss is calculated as 0.01231.

To visually evaluate the outputs of the trained network, another tool based on Python's MSS module is created to capture the live footage from the game's window. Sample images from the test set and detected lane points are shown in Figure 5.1 to Figure 5.3. Their respective ground truth and predicted values of the seven outputs are given in Table 5.2, Table 5.3 and Table 5.4.



Figure 5.1 Three points marked by the model

Table 5.2 Predicted and actual results for Figure 5.1

Parameter	**Actual**	**Predicted**
Angle to road	89.85872	91.25470
Center-X	450	466
Center-Y	376	385
Point0-X	331	332
Point0-Y	376	385
Point1-X	569	612
Point1-Y	376	385

The results show that the model can find lane points when driving straight on a highway it hasn't seen before.



Figure 5.2 Three points marked by the model

Table 5.3 Predicted and actual results for Figure 5.2

Parameter	**Actual**	**Predicted**
Angle to road	90.26978	88.40917
Center-X	427	490
Center-Y	355	364
Point0-X	340	409
Point0-Y	355	367
Point1-X	514	583
Point1-Y	356	362

The positive aspect of extracting data from the game is that the lane points can be marked even when a larger vehicle blocks the agent's view, as can be seen in Figure 5.3. Such cases do not cause a problem, because the dataset also includes many similar cases where there is traffic and lanes are only partially visible or not visible at all.



Figure 5.3 Detection of lane boundaries while lanes are partially visible

Table 5.4 Predicted and actual results for Figure 5.3

Parameter	**Actual**	**Predicted**
Angle to road	92.12427	92.27007
Center-X	506	434
Center-Y	449	417
Point0-X	262	247
Point0-Y	449	417
Point1-X	743	637
Point1-Y	449	418

5.3 RL Agent Training

For the training of the reinforcement learning agent, TensorFlow Agents reinforcement learning framework is used (Google-LLC, 2022). A double deep Q-learning agent with experience replay buffer is constructed. Multiple preliminary trainings are carried out with a simpler Q-network of two hidden layers where first layer has 100 units and the second has 50 hidden layers in order to determine the action space given in Table 4.1.

After the determination of the best action space, a Q-network of three hidden layers with 128 units each is constructed. Training of the agent with the reward function in Equation 4.1 took place in three stages where first stage is 860000 steps long and other two stages consist of one million steps each. Each episode is limited to 1000 steps in order to ease the convergence of the model. Agent's learned policy is evaluated every 20000 steps for five episodes and average reward over these episodes is calculated as the training metric. Initial training of one million steps is carried out using parameters in Table 5.5.

In the second training learning rate of 1e-3 is decreased to 1e-4 and epsilon value is fixed to 0.1. Finally in the last stage, τ parameter, which controls how hard the weights of the online network are copied to the target network, is set to a much smaller value of 1e-3 to allow softer copies in order to increase the stability of the training. Training is done on RTX 3080 graphics card and each stage of the training took approximately 30 hours. This is mainly due to action space containing larger key press durations such as 0.5s.

5.4 RL Agent Training Results

Graph of average rewards calculated every 20000 steps over five episodes for all three stages of training combined can be seen in Figure 5.4.

Table 5.5 Parameters in first stage training

Parameter Name	**Value**
Number of iterations (steps)	860000
Initial collection steps	20000
Steps collected per iteration	1
Replay buffer size	250000
Batch size	64
Learning rate	1e-3
Evaluation interval	20000
ϵ	Linear decrease from 0.75 to 0.1 over 500000 steps
γ	0.99
τ	0.5
Target update period	1000

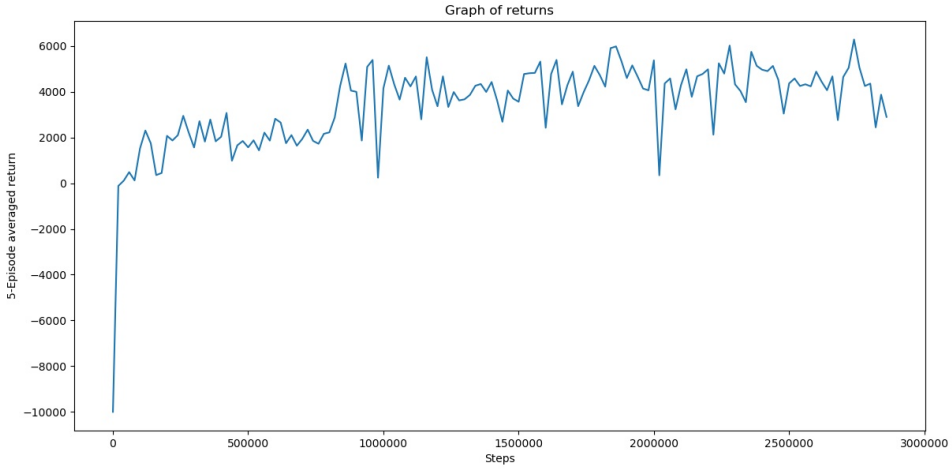


Figure 5.4 Average rewards calculated during DDQN training

According to the average rewards graph, best performing agent is obtained in step 2740000 with an average reward of 6290.417 over five episodes. Further tests on this agent are carried out. It is seen that agent is able to remain in between road lanes even on a test road that is not used during the training. When evaluated over 50 episodes on such a road, average reward is calculated as 6682.61.

Compared to an experienced human player's average reward of 9158.86 on the same portion of the road, it is seen that agent is unable to reach a human level driving ability. Visual inspections showed this is caused by slaloming behaviour observed in many similar researches using the same reward function (Güçkıran & Bolat, 2019). Since reward is decreased as vehicle deviates from the perfect zero degree road angle, agent tries to center itself perfectly to the road and given that action space is discrete, this results in slaloming behaviour. Slaloming also prevents the vehicle from reaching the top speed that is set to 20 m/s, therefore preventing the agent to reaching higher rewards.

Even with the side affects of slaloming, agent is able to keep the vehicle on the lane even on test roads with curves and with an average speed of 10 m/s.

5.5 Discussion

The presented data collection method shows how quickly large amounts of datasets can be collected for different purposes using the in-game engine and some third-party tools. One drawback of the presented solution is that the developed scripts are difficult to debug while the game is running. This can be overcome by using in-game assets to visualize the information needed for debugging.

The prediction error of lane points when a vehicle is turning on a connection road is higher than when driving straight resulting in an increase in MSE. This is due to the fact that the parkour selected during the training contains only several connection roads that the vehicle needs to turn. Also since the AI drives the vehicle in an almost perfectly straight line during dataset collection, the training data is largely dominated by road angles around 90 degrees making it difficult for the model to learn different angle scenarios. Prediction errors of the model can be reduced by using human players during the data collection instead of AI scripts. This will increase the randomness in captured scenes as well as scenarios where the car is not perfectly driven.

In the results of reinforcement learning training, it is shown that video game can be used as a machine learning environment to train autonomous vehicles. Slaloming issue encountered in this work can be overcame by using a reward function that better balances speed and stability. Also by using a similar approach continuous action spaces can be implemented using steering wheel controllers to further increase the success of the agent. Main challenge with reinforcement learning agents is the optimization of hyperparameters. Combined with very long training times it can be cumbersome to test different parameters, action spaces or reward functions. Time constraints are one of the main reasons that we were not able to develop a human-level self driving agent in the scope of this work.

In order to enhance the state detection from video game, more data can be extracted from the video game environment to be used in training more comprehensive CNNs.

One example of this is the vehicle bounding boxes. Using native function GetAllVehicles, it is possible to obtain vehicle objects that are currently generated by the video game engine. With the help of some auxiliary functions such as IS_ENTITY_ON_SCREEN and GET_ENTITY_COORDS, it is possible to write a function that extracts positional information of the vehicle entities that are visible to player's field of view. We can also extract dimensional information about a particular entity using GetDimensions. Combining the location of the entity on the world with its dimensional information, it is possible to draw vehicle bounding boxes. Similar to road lane point detection, these 3D world coordinates can be converted to 2D screen coordinates and dumped to a text file as labels (Giray & Doğan, 2023).

For each vehicle visible on the screen, four points of 2D screen coordinates that represents one corner of the vehicle's bounding box are shown in Figure 5.5. We can limit these to vehicles that are moving in the same direction with player's vehicle by comparing forward vectors. These labels then can be used to train a CNN for various tasks such as an adaptive cruise control system that requires to detect the vehicles in front.

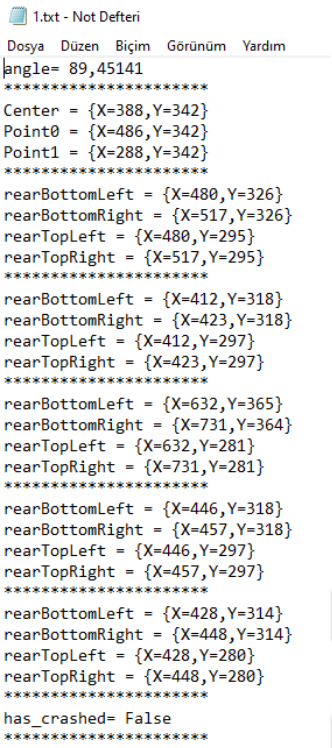


Figure 5.5 Label file with vehicle bounding boxes

One other information that can be found in the label file in Figure 5.5 is the has_crashed flag. This flag indicates if the player's vehicle is contact with any other entity such as a vehicle, pedestrian or another object. It can be used as another state parameters while conducting AI research on the video game.

An example image from the dataset and its label is visualized using a Python script to check the accuracy of these information as shown in Figure 5.6.

Where "False" on the top left corner is the has_crashed flag, 89.45141 is the angle of the car to the road center, and each red mark is either a bounding box for a vehicle in a certain range or detected road markings.

In order to further augment the virtual reality environment, pre-trained advanced object detection models can be utilized. In the scope of this thesis we have tried running one of the pre-trained models from TensorFlow Object Detection API (Huang et al., 2017). Model is trained on Common Objects in Context (COCO) dataset using a lightweight CNN called MobileNet (Howard et al., 2017; T.-Y. Lin et al., 2014).

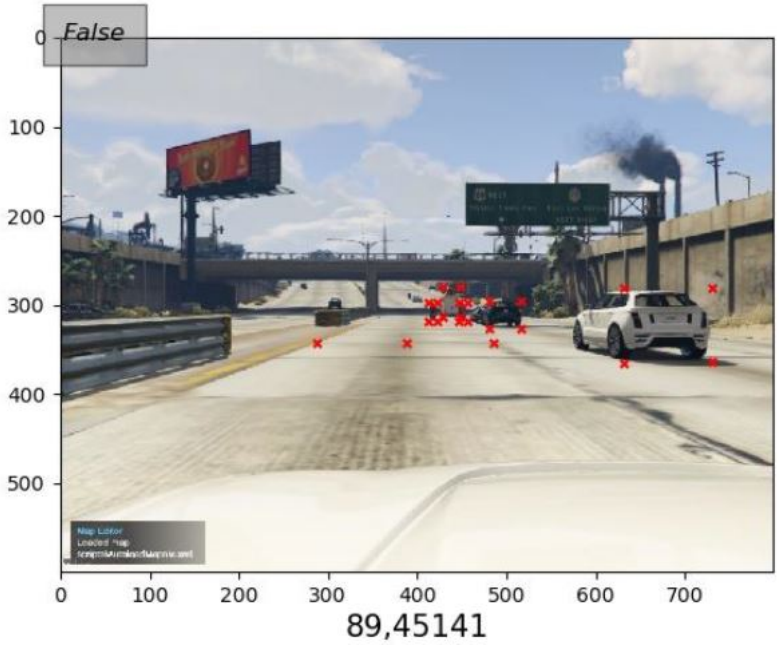


Figure 5.6 Label file information displayed on its image

Even though trained on real world images, visual inspections showed that model predicts objects in video game images with acceptable success as shown in Figure 5.7. By using more sophisticated networks object detection in the video game can be improved and this would help to add variety to state information extracted from the video game in order to be used in further reinforcement learning researches.



Figure 5.7 Example output of the MobileNet on a video game image

5.6 Conclusion

The study shows the possibility of large data collection and training of artificial intelligence models in realistic simulation environments. Using transfer learning methods, ResNet50, VGG16, Xception and InceptionV3 models, which are normally trained on real-life data, have been trained to learn the detection of lane points and road angles in a virtual world. ResNet50 has been shown to outperform the compared models for lane and road angle detection tasks on the collected dataset. The training time and computational power required for transfer learning are significantly lower than training a network from scratch.

The dataset generation methods presented in this work do not require manual labeling operations and are robust in accuracy as the data is extracted directly from the game engine. Using the presented methods, more complex and larger datasets can be generated to further increase the success rate of the developed model under different conditions.

This study also shows the possibility of using the video game as a reinforcement learning environment. A DDQN agent that is able to keep a vehicle in the lane on a highway with a reasonable speed is developed and comparison of the agent's success to an experienced player is presented. Problems encountered and possible solutions are discussed. Used methods and materials are presented in detail in order to provide a foundation for future reinforcement learning researches.

REFERENCES

Bellman, R. (1966). Dynamic programming. *Science*, *153*(3731), 34–37.

Blade, A. (2022a, Oct 28). *Gta v native db.* Retrieved from http://www.dev-c.com/nativedb

Blade, A. (2022b, Oct 28). *Script hook v.* Retrieved from https://www.dev-c.com/gtav/scripthookv

Canny, J. (1986). A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence*(6), 679–698.

Chollet, F. (2017). Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 1251–1258).

Choudhary, A. (2022, Oct 28). *A hands-on introduction to deep q-learning using openai gym in python.* Retrieved from https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/

Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., & Koltun, V. (2017). CARLA: An open urban driving simulator. In *Proceedings of the 1st annual conference on robot learning* (pp. 1–16).

Džakula, B., & Bezdan, T. (2019). Convolutional neural network layers and architectures. In *Sinteza 2019-international scientific conference on information technology and data related research* (pp. 445–451).

Filipowicz, A. (2016). *Driving school ii video games for autonomous driving* (Unpublished master's thesis). Princeton University.

Filipowicz, A., Liu, J., & Kornhauser, A. (2017). *Learning to recognize distance to stop signs using the virtual world of grand theft auto 5* (Tech. Rep.).

Gale, J. W. (2022, Oct 28). *Gta advanced lane finding.* Retrieved from https://github.com/Will-J-Gale/GTA-Advanced-Lane-Finding

Giray, A., & Doğan, H. (2023). Gelişmiş sürücü destek sistemleri uygulamalarında araç sınırlayıcı kutu tespiti için sentetik veri seti toplama yöntemi. In *Batman üniversitesi 1. uluslararası öğrenci kongresi '23.*

Google-LLC. (2022, Oct 28). *Tensorflow agents.* Retrieved from https://www

.tensorflow.org/agents

GTAV. (2022, Oct 28). *Grand theft auto v.* Retrieved from https://www.rockstargames .com/gta-v

Güçkıran, K., & Bolat, B. (2019). Autonomous car racing in simulation environment using deep reinforcement learning. In *2019 innovations in intelligent systems and applications conference (asyu)* (pp. 1–6).

Hasselt, H. (2010). Double q-learning. *Advances in neural information processing systems*, *23*.

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770–778).

Hood Camera. (2022, Oct 28). *Hood camera.* Retrieved from https://tr.gta5-mods .com/scripts/hood-camera

Hough, P. V. (1962). *Method and means for recognizing complex patterns.* U.S. Patent and Trademark Office. Retrieved from https://patents.google.com/patent/ US3069654A/en (US Patent 3,069,654)

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., & Adam, H. (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*.

Huang, J., Rathod, V., Sun, C., Zhu, M., Korattikara, A., Fathi, A., Fischer, I., Wojna, Z., Song, Y., Guadarrama, S., et al. (2017). Speed/accuracy trade-offs for modern convolutional object detectors. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 7310–7311).

Huval, B., Wang, T., Tandon, S., Kiske, J., Song, W., Pazhayampallil, J., Andriluka, M., Rajpurkar, P., Migimatsu, T., Cheng-Yue, R., et al. (2015). An empirical evaluation of deep learning on highway driving. *arXiv preprint arXiv:1504.01716*.

Kingsley, H. (2022, Oct 28). *Pygta5.* Retrieved from https://github.com/Sentdex/ pygta5

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, *60*(6), 84–90.

Lample, G., & Chaplot, D. S. (2017). Playing fps games with deep reinforcement

learning. In *Thirty-first aaai conference on artificial intelligence.*

Lin, L.-J. (1992). *Reinforcement learning for robots using neural networks* (Unpublished doctoral dissertation). Carnegie Mellon University, USA.

Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., & Zitnick, C. L. (2014). Microsoft coco: Common objects in context. In *European conference on computer vision* (pp. 740–755).

Manav, M. (2022, Oct 28). *Introduction to convolutional neural networks (cnn).* Retrieved from https://www.analyticsvidhya.com/blog/2021/05/convolutional-neural-networks-cnn/

Martinez, M., Sitawarin, C., Finch, K., Meincke, L., Yablonski, A., & Kornhauser, A. (2017). Beyond grand theft auto v for training, testing and enhancing deep learning in self driving cars. *arXiv preprint arXiv:1712.01397*.

Microsoft. (2022, Oct 28). *Airsim.* Retrieved from https://microsoft.github.io/AirSim

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *nature*, *518*(7540), 529–533.

Muller, U., Ben, J., Cosatto, E., Flepp, B., & Cun, Y. (2005). Off-road obstacle avoidance through end-to-end learning. *Advances in neural information processing systems*, *18*.

NVIDIA. (2022, Oct 28). *Autosim.* Retrieved from https://www.nvidia.com/en-us/self-driving-cars/

OpenAI. (2022, Oct 28). *Gym.* Retrieved from https://www.gymlibrary.dev/

Pakgohar, A., Tabrizi, R. S., Khalili, M., & Esmaeili, A. (2011). The role of human factor in incidence and severity of road crashes based on the cart and lr regression: a data mining approach. *Procedia Computer Science*, *3*, 764–769.

Pérez-Gil, O., Barea, R., López-Guillén, E., Bergasa, L. M., Revenga, P. A., Gutiérrez, R., & Díaz, A. (2021). Dqn-based deep reinforcement learning for autonomous

driving. In *Advances in physical agents ii: Proceedings of the 21st international workshop of physical agents (waf 2020), november 19-20, 2020, alcalá de henares, madrid, spain* (pp. 60–76).

Rage Plugin. (2022, Oct 28). *Rage plugin hook.* Retrieved from https://ragepluginhook .net

Richter, S. R., Vineet, V., Roth, S., & Koltun, V. (2016). Playing for data: Ground truth from computer games. In *European conference on computer vision* (pp. 102–118).

Ruano, A. (2022, Oct 28). *Deepgtav.* Retrieved from https://github.com/aitorzip/ DeepGTAV

Self Drive Plugin. (2022, Oct 28). *Self drive plugin.* Retrieved from https://tr.gta5 -mods.com/scripts/self-drive-plugin

Shafaei, A., Little, J. J., & Schmidt, M. (2016). Play and learn: Using video games to train computer vision models. *arXiv preprint arXiv:1608.01745*.

Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Stewart, M. (2022, Oct 28). *Simple introduction to convolutional neural networks.* Retrieved from https://medium.com/towards-data-science/simple-introduction-to -convolutional-neural-networks-cdf8d3077bac

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., & Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 1–9).

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 2818–2826).

Thrun, S., & Schwartz, A. (1993). Issues in using function approximation for reinforcement learning. In *Proceedings of the 1993 connectionist models summer school hillsdale, nj. lawrence erlbaum* (Vol. 6, pp. 1–9).

Tsang, S. (2022, Oct 28). *Review: Inception-v3 — 1st runner up (image classification)*

in ilsvrc 2015. Retrieved from https://sh-tsang.medium.com/review-inception-v3-1st-runner-up-image-classification-in-ilsvrc-2015-17915421f77c

Van Hasselt, H., Guez, A., & Silver, D. (2016). Deep reinforcement learning with double q-learning. In *Proceedings of the aaai conference on artificial intelligence* (Vol. 30).

Vitelli, M., & Nayebi, A. (2016). Carma: A deep reinforcement learning approach to autonomous driving. *Tech. rep. Stanford University, Tech. Rep.*.

Von Dollen, D. (2017). Investigating reinforcement learning agents for continuous state space environments. *arXiv preprint arXiv:1708.02378*.

Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine learning*, *8*(3), 279–292.

World Health Organization. (2018). *Global status report on road safety 2018.* Retrieved from https://www.who.int/publications/i/item/9789241565684

Zou, Z., Shi, T., Li, W., Zhang, Z., & Shi, Z. (2020). Do game data generalize well for remote sensing image segmentation? *Remote Sensing*, *12*(2), 275.