

TASK DESCRIPTION



HEAD OF DEPARTMENT

MSc Thesis Task Description

Mounir Mohammad Abou Saleh

candidate for MSc degree in Electrical Engineering

Deep Reinforcement Learning based Autonomous Driving Agents

Autonomous cars establish driving strategies using reinforcement learning (RL) a powerful machine learning framework. RL is an important step toward the environment where it teaches machines through interaction with the environment and learning from their mistakes. Deep Reinforcement learning (DRL) was the most approach to simulate the act of human drivers as a self-driving car controller. Controlling a vehicle with continuous acceleration, breaking and steering angle variables.

The aim of the thesis is to provide an overall analysis of current DRL algorithms for training autonomous vehicle agents. Deep-Q-Network algorithm (DQN), which has the capacity to predict discrete actions with continuous state spaces. Proximal Policy optimization (PPO) based agent that can reliably learn to drive in a driving-like environment. In this project, we used open-source simulator (Carla) and openAI gym (CarRacing-v0) as our environment to train our agent to explore and analyze the possibilities of achieving autonomous driving.

Tasks to be performed by the student includes:

- Design and setup the environment (Open-source simulator: Carla) for testing our agent
- Improve the quality of our agent using variational autoencoders in the training pipeline.
- Build and design the network architecture for both algorithm DQN and OPP to fit in the environment.
- Experiment with training different types of neural networks, architectures, and parameters for both algorithms.
- Evaluate our agent in different modes for achieving self-driving within the chosen simulator.
- Provide metrics to compare between the models across runs.

Supervisor at the department: László Grad-Gyenge, Research Assistant Fellow
Budapest, 24 September 2021

Dr. Hassan Charaf
professor
head of department





M Ű E G Y E T E M 1 7 8 2

Budapest University of Technology and Economics

Department of Automation and Applied Informatics

abosalehmounir@gmail.com

Diploma Project

Project Title:

Deep Reinforcement Learning based Autonomous Driving Agents

Prepared By

Mounir Mohammad Abou Saleh (A2X7DS)

Supervised By

Grad-Gyenge László György

Budapest, 2021

Acknowledgements

I would like to acknowledge the valuable support provided by my supervisor in the preparation of this thesis. Grad-Gyenge László György Research Assistant Fellow who has been my guide that provided oversight on this assignment, helping to shape the approach, execution, and documentation of this project providing a great help to solve the problems encountered while working in this project.

Contents

Summary.....	9
Chapter 1: Introduction	10
1.1 Motivation.....	10
1.2 Autonomous Driving.....	10
1.3 Components of autonomous car.....	11
1.4 Thesis Objective.....	12
2 Related Work	14
3 Background	17
3.1 Machine Learning	17
3.1.1 Introduction.....	17
3.1.2 Artificial Neural Networks	18
3.1.3 Deep learning	19
3.1.3.1 Convolutional neural network.....	19
3.2 Reinforcement Learning	20
3.2.1 Introduction.....	20
3.2.2 Markov Decision Process (MDP)	21
3.2.3 Q-learning	24
3.2.4 Deep Q-Learning.....	27
3.2.4.1 Experience Replay	28
3.2.4.2 Fixed Target Network	28
3.2.4.3 DQN Implementation Process Steps.....	28
3.2.5 Proximal Policy Optimization (PPO).....	29
4 Carla Simulator.....	33
4.1 Introduction.....	33
4.2 Configuration of the simulation and of the actors.....	33
4.3 Synchronous Mode Configuration on CARLA Simulator.....	35
5 Deep Reinforcement Learning (DQN) in Carla Simulator	38
5.1 Setup.....	38
5.2 DQN Carla Environment	38
5.3 State, Action, Reward	40
5.4 DQN Agent Model.....	41
5.5 Experiments	42
5.6 Results.....	44

6 Proximal Policy Optimization (PPO) in Driving-Like Environment..... 47

6.1 Setup..... 47

6.2 Environment..... 47

6.3 Algorithm 49

6.4 Experiments 50

6.4.1 Image Preprocessing 50

6.4.2 Network Architecture..... 50

6.4.3 Optimization 52

6.4.4 Recurrent Model 53

6.4.5 Variational Autoencoder 53

6.5 Results..... 55

7 Conclusion And Future Work 58

7.1 Contributions..... 58

7.2 Future work 59

References 60

List Of Figures

Figure 3-1: Reinforcement Learning Loop	21
Figure 3-2: Q-Learning algorithm steps.....	25
Figure 3-3: Actor-Critic Proximal Policy Optimization Model[16]	31
Figure 4-1:CARLA client-server communication[2].....	33
Figure 4-2: Carla simulator Town3.....	34
Figure 4-3: Vehicle Spawn in Town3	34
Figure 4-4: CARLA simulator: Town 4.....	35
Figure 4-5: Vechicle(ILincoln MKZ2017) spawned for starting collecting data images, 10000 images as limited.....	36
Figure 5-1: Car spawned with camera sensor position at 2.5 forward and 0.7 up	40
Figure 5-2: 64X3 CNN Architecture	41
Figure 5-3: Xception Model Accuracy	43
Figure 5-4: 64x3 CNN Model Accuracy reach at 85%	44
Figure 5-5: 64x3 CNN Model Epsilon.....	45
Figure 5-6: 64x3 CNN Model Loss	45
Figure 5-7:64x3 CNN Model Reward Average	45
Figure 5-8:64x3 CNN model Reward max	46
Figure 5-9: 64x3 CNN model Reward min.....	46
Figure 6-1:State Space seen by the agent.....	48
Figure 6-2: Screenshot of CarRacing-v0 environment	48
Figure 6-3: Frame Stack Model Architecture With Four Stacked Frames	51
Figure 6-4: Recurrnet Model for PPO algorithm	53
Figure 6-5: evaluate the reconstruction loss of VAE train model.....	54
Figure 6-6: Cumulative reward of the models over number of epochs	55
Figure 6-7: Evaluation of error value for the models.....	56
Figure 6-8:Break action mean-FrameStack model	57
Figure 6-9: Acceleration mean action-FrameStack model	57

List Of Algorithms

Algorithm 1: Q-learning algorithm26

Algorithm 2: DQN Algorithm.....27

Algorithm 3: Actor-Critic PPO Algorithm32

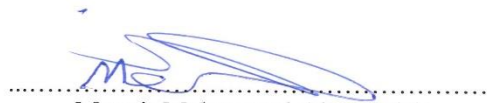
DECLARATION

I, **Mounir Mohammad Abou Saleh**, the undersigned, hereby declare that the present MSc thesis work has been prepared by myself and without any unauthorized help or assistance. Only the specified sources (references, tools, etc.) were used. All parts taken from other sources word by word, or after rephrasing but with identical meaning, were unambiguously identified with explicit reference to the sources utilized.

I authorize the Faculty of Electrical Engineering and Informatics of the Budapest University of Technology and Economics to publish the principal data of the thesis work (author's name, title, abstracts in English and in a second language, year of preparation, supervisor's name, etc.) in a searchable, public, electronic and online database and to publish the full text of the thesis work on the internal network of the university (this may include access by authenticated outside users). I declare that the submitted hardcopy of the thesis work and its electronic version are identical.

Full text of thesis works classified upon the decision of the Dean will be published after a period of three years.

Budapest, 18 December 2021



Mounir Mohammad Abou Saleh

Summary

In this thesis, we will investigate the present landscape of state-of-the-art approaches for training self-driving cars using deep reinforcement learning. Autonomous driving has recently inspired the interest of researchers, governments, and companies since it has the potential to solve several issues in today's world. Examples of such issues include individuals spending a lot of time stuck in traffic because of blockage, and people and companies have to pay for costly car accidents caused by human mistakes. Machine learning advancements are improving autonomous cars, and we have already seen many huge companies in the automotive and artificial intelligence sectors benefit from this by having autonomous vehicles travel several kilometers on public highways without accident. Reinforcement Learning (RL) is a general-purpose framework that is developing with the new concepts from deep learning. It is used to train agents without the use of datasets as in supervised learning, but rather via environment interaction and learning from their failures.

The main objective of this thesis is to provide a comprehensive analysis of current approaches for training autonomous vehicle agents using deep reinforcement learning (DRL). And our key contribution is to present two working implementations of DRL algorithms. First, we present a Deep Q-Network (DQN)-based agent capable of reliably learning to drive in the CARLA urban driving simulator. Second, Proximal Policy Optimization (PPO) to teach an agent to drive in a driving-like environment (CarRacing-v0). Through our work, we will discuss the fundamentals of reinforcement learning theory and then go further into general-purpose deep reinforcement learning techniques for handling complicated tasks. We will experiment with many types of neural network designs and experimentally compare and explain the advantages and disadvantages of each of these models throughout the development of DQN and PPO utilizing models. Additionally, setup of the CARLA simulation to train the DQN model using a different reward formalizations, scenarios, weather, and traffic conditions in order to get the optimal result.

Chapter 1: Introduction

1.1 Motivation

In today's world, we have seen an increasing desire to use technology to make the world safer and more reliable to live. Creating intelligent systems that can safely drive a car from A to B points without any human input is one of the desire approaches. While the idea of self-driving car is old and well-known, many companies and researchers start to consider these technologies seriously. Along with other computer vision and artificial intelligence concerns, developing autonomous vehicles is now considered feasible because of the numerous improvements in artificial intelligence made possible by deep learning.

1.2 Autonomous Driving

When we talk about self-driving cars, we usually mean vehicles that drive themselves from point A to point B with no human interaction in the vehicle's control system. The Society of Automotive Engineers[1] has defined a set of levels of increasing automation. Levels as follow:

- Level 0 - No Automation: In this case, the vehicle's control system is only controlled by the human driver. Even if a system has warning signs or intervention devices built in, it is still deemed level 0.
- Level 1 - Driver Assistance: The human driver and the automated system share control of the vehicle. A car with Adaptive Cruise Control, a system in which the car sets its own speed, but the human driver is responsible for steering.
- Level 2 - Partial Automation: The system assumes complete control of the vehicle's steering and speed, but a person must be ready to act instantly if the system fails.
- Level 3 - Conditional Automation: The driver does not need to be ready for instant intervention, but they must respond within a certain amount of time if the system requires it.
- Level 4 - High Automation: The car can drive itself in certain geographic areas and in limited sorts of driving scenarios. If the vehicle is unable to continue, the human driver take control.
- Level 5 - Complete Automation: There is no need for any human interaction.

Automobiles of today are generally composed of combination of level 0 (“no automation”) and level 1 (“hands on”) autonomous cars. Certain companies, such as Tesla and KIA, have currently produced cars with capabilities classified as level 2 (“hands-off”) automation, such as autonomous lane changing. Additionally, there are a few instances of commercial and non-commercial cars equipped with level 3 (“eyes off”) automation, including Audi’s A8’s “traffic jam Pilot”, which is capable of driving the car autonomously up to 60 kilometers per hour in traffic jams.

1.3 Components of autonomous car

Self-driving car required a systems and components which solve their own individual task.

Sensor, Compute and Control

When building an autonomous car, we need to think about what components and sensors are required to drive optimally, ensuring that our vehicle is energy efficient for driving for lengthy periods of time with limited battery capacity. We will also need to work on the cars control algorithms as well as the interfaces it uses to connect with its sensors.

Mapping and Localization

Finding out how to localize the car on a high-definition map and analyze the topology of the car’s surrounding environment is the goal of mapping and localization. On high-definition maps, positioning technologies such as the Global Positioning System (GPS) may be used, and this information may be combined with information extracted from the environment. For example, given a rough estimate of its location on a map, the car may be able to identify a more specific location by combining information gathered from its surroundings with its predictions of where neighboring structures should be based on the vehicle's internal map.

Simulators For Training and Validation

Before deploying our control and perception algorithms in the actual world, it's critical to test and verify them. The primary goal of creating simulators is to use them to test our algorithms before deployment.

Gym, developed by Open-AI is a popular open source of artificial intelligent projects used for reinforcement learning environments and development tools. It has become a typical benchmarking tool for RL algorithms due to its curated selection of challenges and ease of use.

Open-AI gym provide carRacing-v0 environment which is continuous action space. Simulators must also precisely match the physics and appearance of real-life situations. For autonomous driving research, there are several open-source simulators available. CARLA[2] is the most well-known.

Planning and Control

Planning refers to the process of making actions such as acceleration, break, and steering to drive the vehicle from one area to another [3]. Where vehicle is controlled by the output from its sensor, or the model perception that provides the vehicle with crucial information on the driving environment to determine the series of control signal that will lead to the target point, the vehicle should be able to avoid obstacles and optimizing.

Safety and Security

It is important to ensure the safety of autonomous systems before they are used. Simulators are helpful in testing methods, but deep learning-based agents are difficult to comprehend. In the case of an accident, we must be able to reconstruct out why the agent failed. So, safety is about ensuring that our systems are interpretable. Security is also a crucial aspect of autonomous vehicle safety; we must ensure that our autonomous vehicle technologies are not exploited or intercepted by external parties.

1.4 Thesis Objective

Our main objective in this thesis to investigate state-of-the-art approaches in reinforcement learning, with the goal of applying this theory to develop reinforcement learning models that are capable of safely and reliably maneuvering themselves within the simulators, as well as other applications. The work is dependent on the subject of machine learning, specifically the category of reinforcement learning, which is focused on learning from mistakes while interacting with the surrounding environment.

The thesis will investigate two reinforcement models. First, deep Q-Network based on training a convolutional neural network to predict the table Q-values of an action at a current state. Second, Proximal Policy Optimization (PPO) that based on actor-critic networks, PPO operates by optimizing the policy, actions taken by the agent over time.

Testing and evaluating the models will be performed in the two simulator CARLA and CarRacing-v0. The deep Q-Network model will be implemented in CARLA a real-life environment,

the training process will be in Town3 with different scenarios. While for the PPO method the model implemented in the OpenAI CarRacing-v0 an environment developed for testing RL algorithms.

In the pursuit of this objective, we set the following contributions and questions to answer:

- Propose Principles and algorithms of deep reinforcement learning. And is modern reinforcement learning approaches able to teach a car to drive in a supervised and safe manner?
- Build and setup CARLA environment with different scenarios to fit our model, in addition for synchronous mode configuration for collecting data.
- What objectives, measurements, and state representations should a reinforcement learning environment provide to meet the goals of autonomous driving researchers?
- How successful is the internal neural network model of these approaches in terms of training speed and performance, and are there other models with desirable qualities over the tried and tested models employed in these works?
- How the variational encoder will be train, and how it will enhance the performance of a deep reinforcement agent?

2 Related Work

In this chapter, we will summarize the researcher's earlier work and studies in the area of autonomous vehicles utilizing various reinforcement learning methods. They employed a variety of approaches, simulations, and reward functions to improve the agent car training.

Self-driving automobiles are one of the most common examples of autonomous systems. The primary goal of a vehicle that is completely self-driving car to be without interaction from the human driver. A Reinforcement learning algorithm called deep Q-network used by the authors in paper [4] to develop a training strategy in CARLA a simulated environment and collect rewards in a challenge with huge state space. Finally, with the help of DQN, the author handles the issue of an exploding state-space.

Reinforcement learning used in paper[5] to control a virtual automobile driven by the algorithm DQN. Using a simulated environment, authors train an agent without pre-feeding it with data. This strategy nearly identical to ours study work of implementing the algorithm DQN on CARLA simulator. On the other hand, the authors train the model using JavaScript racing game instead of actual traffic issues, where the agent driving behavior might be affected by an infinite number of variables.

Using deep reinforcement learning, the authors handle the challenge of developing long-term driving methods in[6]. The author shows two major challenges. First enhancing functional safety. Second, the Markov Decision Process model causing unpredictable behavior of other agents in this multi-agent scenario. The authors demonstrate the problem by using the policy gradient iteration, decomposed into a formation of a policy for desires, and introduce a hierarchical temporal abstraction refer to option graph.

The study in [7] provides a method for teaching an autonomous car agent to safely control rounds in the CARLA simulator by implementing the Q-Learning RL algorithm. The authors create a real learning approach for successive decision-making in CARLA simulator, the Markov decision process (MDP) is utilized in the paper to design the analysis of autonomous vehicle behavior in navigating a roundabout safely using the Q-learning method in a simulated environment. Moreover, in this study, machine learning algorithms and a set of practical driving information were employed to teach the agent to make a decision. to achieve ideal driving performance, a trained system has

been built to learn and determine if a certain action has positive or negative and reinforcing it via a predetermined rewards policy.

The Open Racing Simulator (TORCS) used in paper [8] in order to investigate deep Q-Learning, the authors build a neural network model to train an agent that can navigate a wide variety of track designs. Train and test the model on two validation tracks using reward function that maximizes longitudinal speed while reducing transverse speed and divergence from track center. The scientists believe that by simulating long-term state dependencies in addition to a memory on a specific track, LSTM networks will be able to learn better policies and increase performance on following curves and laps. Furthermore, a coordinate transformation used to make real-world sensor data similar to TORCS simulations.

In [9] the authors collect a large dataset of highway, thousands of frames with vehicle bounding boxes and lane annotations. The authors used deep learning and computer vision techniques to solve difficulties such as lane recognition and automobile detection. A CNN architecture capable of recognizing all lanes and vehicles in a single forward pass was then trained on this data. This paper demonstrates how current convolutional neural networks (CNNs) may be utilized to recognize lane and vehicle in real-time systems. For highway lane and car recognition, CNN algorithms work well as a result for the paper.

Deep reinforcement learning for lane keeping assist is introduced in paper [10] based on multiple kinds of algorithms employed in the TORCS simulator. The authors demonstrate a discrete action that deal with deep Q-Network, and continuous action deal with Deep Deterministic Actor Critic Algorithm (DDAC). When driving on a curvy route with other cars and variety of curves, simulations show that a vehicle learn to drive by itself. It's a new field of study which investigates how constrained restriction effect the learning process of the algorithms.

For continuous steering control, the authors in [11] concentrate on establishing the steering smoothness and then proposing an additional reward penalty since it is hard to transition between distinct discrete steering values at close time in real. In TORCS and CARLA simulators, DDPG and A3C experiments are carried out. In both studies, the new penalty was shown to be more effective than the old one. Results demonstrate that the suggested penalty increases the smoothness of steering operations in both algorithms.

In [12]. Learning to drive in a day, autonomous vehicle trained to follow a rural road using just one monocular front-facing camera and a few episodes of training. The authors define the reward as the vehicle forward speed, and they end the episode when the car drifts too far to the sides

of the road (the termination pulse is issued by the human driver). Determining the reward in that's way pushes the vehicle to travel as far as possible. The algorithm used in this paper is actor-critic DDPG training a convolutional neural network. In addition, they discovered that encoding the pictures using variational autoencoder (VAE) was for effective than training the convolutional layers in conjunction with all the actor-critic parameters.

In [13], the paper describes an imitation learning model that can learn to traverse arbitrary routes utilizing different actor networks, depending on what the vehicle's current move should be. The authors created a model that teaches sub-policies for three different commands left, straight, right. Based on the expert actions, a chosen branch will optimize throughout the training process. The authors tested the model on CARLA simulator as well a small RC car. As a result, the model had an episodic accuracy rate of 88 percent in their testing.

In [14], Deep Q-learning networks are proposed as an inverse reinforcement learning strategy for solving large state space issues. This technique is evaluated in a simulation-based autonomous driving scenario by the author. Decisions are made via the Markov Decision Process. The output of the networks was to steer to the left, right, or move forward. After several learning episodes, the authors were willing to perform collision-free actions.

Most of the research in reinforcement learning that we've looked at so far has been about how to solve problems in video games and with robotics. There has been little study on the use of deep reinforcement learning for autonomous driving on real life because of the difficulty of safely training a reinforcement agent in the actual world, since such an agent must explore its environment to learn. However, in [12] the authors demonstrated that a vehicle can be trained to effectively follow a landscape using just a single monocular front-facing camera.

in this thesis, the most researchers relevant to our work of implemented DQN in the CARLA simulator was paper [5] and, but the authors used JavaScript while we used Python, also the concept and understood of Markov decision process present in paper[7], while for implementing PPO in the CarRacing-v0 inspired by the papers [15] and [16].

3 Background

3.1 Machine Learning

3.1.1 Introduction

Machine learning is a sub-field of artificial intelligence that support the development of new algorithms and systems that learn to solve problems from examples, many of today's leading companies consider machine learning a central part of their operations. Machine learning frequently outperforms hand-crafted algorithms for problems with some degree of non-triviality, such as detecting objects in photos [17]. Because the theory behind machine learning algorithms is frequently general-purpose, the same theory may sometimes be re-purposed for a range of systems, as long as the relevant data for the specific problem is available.

Some machine learning algorithms work by estimating the model from example inputs and then using that model to generate predictions. It's important to collect data for training and testing. Data represents some type of information that machines, or humans can understand, such as pictures, stock prices.

Supervised learning is one of the main types of machine learning. In this type, the machine learning models are trained with labeled dataset, which helps the models to learn and improve over time. For example, a model would be trained to classify between the pictures of cats and dogs. The most task of supervised learning is to create a regression model that maps input data to some output data. Create a Function F , $F(X)=Y$, where X represent the inputs pictures, pictures of the animals. And Y refer to the corresponding labels for each X , an expert must manually label a group of X and Y pairs to be reasonable size. Given that our dataset (the complete array of labeled data points) is well-formed, regression models that approximate and generalize $F(x)$ can be created. Artificial neural networks (ANNs) are frequently used for this algorithm. Researchers have decided to make their manually gathered datasets available to the public. Some famous datasets used nowadays include MNIST[18], ImageNet[17], for self-driving car, such as KITTI [19], Apollo[20].

Moreover, unsupervised learning is also a type for machine learning where we solve the equivalent $F(x)=Y$ regression problem with unsupervised learning, but without any labeled data Y . Clustering is a common form of unsupervised learning, in which we try to group data points based

on statistical features of the data. Semi-supervised is a common between supervised and unsupervised where small amount of X is label

In this project, we will study the reinforcement learning algorithms which is a type of machine learning. Reinforcement learning (RL) is based on training a model to make a sequence of decisions. Which we consider an agent lives in environment where the agent can behave. The purpose of RL is to teach the agent to maximize the total reward. In case of autonomous car, the agent vehicle can be learned to drive between two desire points with avoiding obstacles. In this case, the agent utility is a metric that represents how well it can get from the start point to the target pint without collusion.

3.1.2 Artificial Neural Networks

Artificial neural networks (ANNs) are computer models based on the human brain's neuron networks. They are also known as learning algorithms for modeling input-output interactions. In machine learning, artificial neural networks refer to a directed-acyclic graph with weights along each edge.

ANN is a system of interconnected nodes. The nodes are sorted and organized into liner arrays known as layer, in ANN there are three layers, input layer, hidden layer, and the output layer. When we feed a data point Xi to an ANN, it propagates across the network, and we get a prediction $F(Xi) = Yi$. In machine learning, the goal is to find the ideal configuration of network weights, usually known as W , with respect to some objective.

Gradient decent: Gradient Descent is an optimization technique that is used to improve ANNs based models by minimizing the cost function. It operates by computing the gradient of the loss function, L , in terms of the network's weights, W . We next shift our weight variables in the direction of greatest descent as fellow:

$$w_{ij} \leftarrow w_{ij} - a \frac{\partial L}{\partial w_{ij}} \quad (3.1)$$

Where a is a hyperparameter that determines how much we should nudge our weights, and $\frac{\partial L}{\partial w_{ij}}$ is the steepest ascent along the loss function's surface with respect to a given weight w_{ij} .

3.1.3 Deep learning

The simplified formulation of ANNs shown above is incapable of generalizing to new data, since this kind of ANN is really nothing more than a linear combination of transformations on the input X across the layers' weight matrices w . As it turns out, this difficulty is resolved by developing more complex, non-linear models. We integrate these concepts in deep learning by developing deep networks in which we apply non-linear activation functions to our neurons, allowing the network to infer non-linear correlations in the input data. The sigmoid function and the Rectified Linear Unit (ReLU) function are two frequently used non-linear activation functions.

3.1.3.1 Convolutional neural network

Convolutional Neural Networks (CNNs) are deep neural networks algorithms that perform convolutional operation on the input form. Image classification and object recognition applications are now more scalable approach to CNN, there is three main types of layers in CNN, a convolutional layer is the first layer in the model, and it can be followed by adding to it a pooling layer, while the final layer is the fully connected layer, the more layers added to the CNN model the more complex our model will be.

Convolutional layer: The convolutional layer is the most important component of CNN where the most majority of the processing takes place. It requires input data, a filter, and a feature map. For example, the input is a color image, which is formed of a 3D matrix of pixels. This means the input will have three dimensions: height, width, and depth, which related to the RGB color space of a picture. A feature detector, also known as a kernel or a filter, will simulate across the image's receptive fields, checking for the presence of the feature. Convolution is the term for this procedure.

The feature detector is a weighted two-dimensional (2-D) array that represents a part of the image. The receptive field is determined by the size of the filter, which is generally a 3x3 matrix. After that, the filter is applied to a section of the image, and the dot product between the input pixels and the filter is determined. The output array receives this dot product. The filter then moves by one stride, and the procedure is repeated until the kernel has swept across the whole picture. A feature map, activation map, or convolved feature is the eventual output from a series of dot products generated by the input and the filter.

Pooling layer: pooling layer is similar to convolutional layer where it works on reducing the number of the parameters in the input image, but the difference is that the filter of the pooling layer

does not have a weight. The advantage of pooling layer it came by avoiding overfitting and reduce the complexity. There are two types of pooling

- Max pooling: while the filter passes across the input image, the pixel with the highest value is chosen to send to the output array. This method is frequently used.
- Average pooling: while the filter passes across the input image, it computes the average value of the receptive field and sends it to the output array.

Fully-connected layer: each node in the output layer is connected directly to a node in the previous layer. This layer performs classification tasks based on the characteristics obtained by the pervious layers and their various filters. While convolutional and pooling layers often apply ReLU functions to classify inputs, fully connected layers typically use a Softmax activation function to provide a probability from 0 to 1.

3.2 Reinforcement Learning

3.2.1 Introduction

Reinforcement learning is a subset of tasks that have the following characteristics:

1. There is an agent that can act in environment
2. When the agent acts, the environment provides feedback, which is referred to reward.
3. The agent can observe the current state returned by the environment to make actions
4. The policy is the rule which applies by the agent to make the decision for the next action

We're talking about a reinforcement learning challenge when a problem can be defined by the characteristics listed above. As shown in (Figure 3-1), we generally think of this framework as a loop.

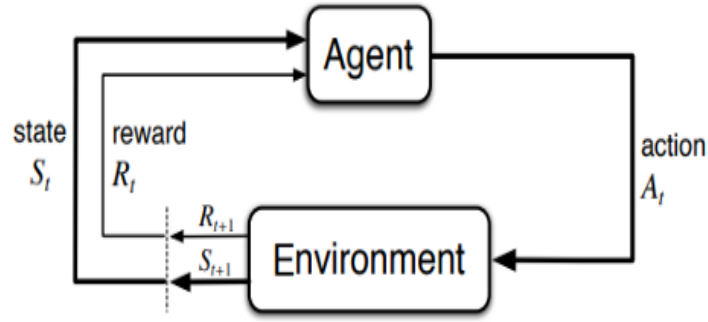


Figure 3-1: Reinforcement Learning Loop

The figure above describes the learning loop of reinforcement, the agent begins at time-step t by observing the state S_t and its reward R_t in real time, the environment in return, provides a new rewards R_{t+1} and a new state S_{t+1} based on the action A_t of the agent, R_{t+1} and S_{t+1} represent the reward and state in time-step $t+1$ respectively.

3.2.2 Markov Decision Process (MDP)

Markov Decision Processes properly defined processes that are stated in terms of agents, environments, and rewards. The 4-tuple (S, A, P, r) may be used to simulate a Markov decision process, which is a time discrete stochastic control process:

- S is set of states in MDP.
- A is set of action in MDP, also A_s represent the action in the current state S
- $P(s'|s, a)$ is the probability of observing the state s' when performing the action a in state s , where $s, s' \in S$ and $a \in A$.

A time discrete MDP means that we go step-by-step through time, when an agent makes action a_0 and the environment respond with a reward r_1 , and then the agent does another action a_1 and the environment respond with another reward r_2 , and so on. Most of the time, in episodic tasks we begin at the beginning of the time-step and terminate after the terminal state has been achieved. An episode is a sequence of events that occurs between the times $t=0$ and $t=\text{terminal}$. In continuous setting, the process will never achieve a terminal state. Which means that the process will continue to run until $t \rightarrow \infty$.

In reinforcement learning the models can be built by transition probabilities where MDP is a stochastic process. Mathematically the MDP equation shown as:

$$P[S_{t+1}|S_t] = P[S_{t+1}|S_1, S_2, \dots, S_t] \quad (3.2)$$

$S[t]$ refer to the current state of the agent and $S[t+1]$ the next state, Essentially, the transfer from state $S[t]$ to state $S[t+1]$ is completely independent of the past, according to equation (3.2). As a result, if the system exhibits Markov property, the RHS of the equation is equivalent to the LHS. This means that our present state has already all the information from the previous states, which is intuitively correct. Now we can define the transition probability for Markov state from $S[t]$ to $S[t+1]$ as follow:

$$P_{ss'} = P[S_{t+1} = s' | S_t = s] \quad (3.3)$$

We convert the state transition probability into matrix by:

$$P = \begin{bmatrix} p_{11} & p_{12} & \dots & p_{1n} \\ p_{21} & p_{22} & \dots & p_{2n} \\ & \dots & \dots & \\ p_{n1} & p_{n2} & \dots & p_{nn} \end{bmatrix}$$

Elements in the matrix represent the probability values from moving from a state to another state, for example P_{12} refer to the transition from state 1 to state 2. The summation of the probabilities values in each row is equal to one. $S[1], S[2], \dots, S[n]$ are random states with a Markov property. So, it's just a Markov sequence of states. It has a collection of states and a transition probability matrix (P). Where the States (S) and Transition Probability matrices determine the environment's dynamics (P).

In addition, one of the main purposes of MDP is to identify the best control policy for choosing actions, which mean that MDP it's also a control process in this perspective. The policy defines the set of rules that the agent observes to choose what action to take in response to a given state, and it's symbolized by $\pi(a|s)$. It's the optimal policy for maximizing cumulative reward, in mathematically way, to identify the optimal policy, we must find policy that maximizes the value of the following equation(3.4):

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \quad (3.4)$$

Where γ is the discount factor that prevents the total reward from increasing to infinity. Choosing a greater γ value between 0 and 1 indicates that agent's actions are more dependent on future reward. On the other hand, a smaller γ value results in actions that mostly concerned with the instantaneous reward r_t .

Exploration and Exploitation

Reinforcement learning is a subset of MDP issues in which the agent does not know the state-transition probabilities $P(s'|s, a)$ or the reward function $r(s, a)$. This implies that the agent must investigate the environment to discover a link between state-action pairing and reward. This leads us to the principle of reinforcement learning exploration and exploitation. This is the principle states to reach an optimum policy, an agent must explore and exploit the environment. Exploring requires performing non-optimal or random acts to explore the environment it usually happens at the beginning of the training process because the state and reward are unknown. While exploitation is defined as taking the most preferable action giving the current policies. After seeing several state and rewards, the agent exploits the environment and get closer to reach a solution. Generally exploiting take place in the latter stages of the training process to reach an optimal policy.

Discrete and Continuous state

According to the environment the state and the action spaces might be discrete or continuous. For example, chess is a discrete game with state and action spaces. In chess there are finite number of potential state and action at any moment. While for continuous state and actions autonomous car with a sensor module is an example, where the action and the state is represented by real values. The action contains the vehicle steering it can be left, right, and forward, the value calculated with the angle degree. The state can be represented the frame image the location of the vehicle.

Policy Gradient

It would be better if the policy optimize directly. To find a policy, policy optimization approaches focus on looking for a policy in the subgroup of the policy space. Both gradient-based and gradient-free optimization can be used for this approach. We will concentrate on gradient-based approaches, although gradient-free methods like optimization algorithms have also shown to be effective. Williams presented in his paper the study of the policy gradient optimization algorithms for

connectionist reinforcement learning in 1992[21]. Williams deduces the scientific framework of gradient-based policy optimization. And group of algorithms known as REINFORCE is introduced in the paper that employs gradient following in feedforward networks, widely known today as artificial neural networks (section 3.1.2). In a summary, these approaches gather large number of trajectories, and then utilize gradient ascent to estimate the probability of optimal trajectories in the policy.

On-policy and Off-policy

Using the on-policy technique, the same policy is utilized to decide the value of the policy and to command and control the agent in the same process. On-policy methods include the Proximal Policy Gradient, which is an example of a policy gradient. Off-policy approaches make use of various policies for assessing the policy and for modulating the agent to get the desired results. An example for off-policy method is the use of Q-learning.

Model-based and model-free

Methods that develop a model about how the environment functions are known as model-based methods. When an agent performs action a in a current state s , it moved to s' and given reward r . These data will be used to estimate $P(s'|s, a)$ and $r(s, a)$. An MDP-solving technique such as Bellman algorithms equations would be used once an approximation model has been created to determine the best policy given our present model of the environment. In another hand, Model-free approaches don't need to model the environment in order to optimize the policy. It is hard to develop MDP models without making approximations since the state and action can be continuous, giving infinite values. In addition to directly optimizing the policy, we used the function optimization methods like gradient descent to identify efficient policies

3.2.3 Q-learning

Q-learning algorithm[22] is a solution for reinforcement issues, model-free since it doesn't need to model the environment. It operates by keeping track of Q-values in a table. A state-action pair is mapped to a Q-value that indicates the return of doing action a in state s via the Q-table. Q-learning works only with discrete state and action spaces, based on bellman formula shown in equation (3.5):

$$New\ Q(s, a) = Q(s, a) + \alpha[R(s, a) + \gamma \max_{a'} Q'(s', a') - Q(s, a)] \quad (3.5)$$

- Learning rate(α): define the amount you accept the new value than with the pervious value. Multiplying the ratio among new and old by the learning rate
- $R(s, a)$: reward value after accomplishing action a at state s
- Discount factor γ , it used to find a balance between immediate and futures reward. γ range is between 0.8 to 0.99

States are linked to actions value via the bellman equation. Using this tool, we can navigate the environment and choose the optimal policy values. The form of Q-values is a matrix of states and action, the Q-matrix is randomly initialized, the agent starts acting in the environment to calculate the reward for each action taken by the agent. The matrix is then updated based on the Q-values that have been seen.

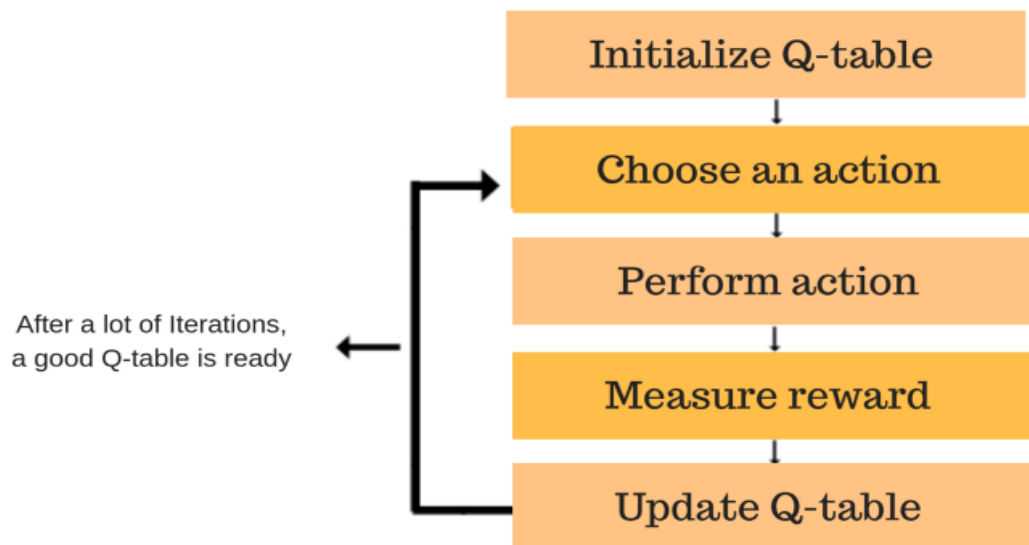


Figure 3-2: Q-Learning algorithm steps

Figure 3-2 show the steps for Q-learning algorithm:

Step 1: we will start by making a Q-table ($M \times N$) where M rows equal to the number of states, and N columns number of actions, at the beginning set the values to zero.

Step 2-3: The sequence of actions is repeated for infinite amount of time. This indicates that this process will continue until the training is completed or the loop is terminated. On the Q-table, we will pick an action at a current state. However, as previously stated, when the episode first begins, the Q-values are set to 0. the idea of exploration and exploitation trade-off now takes hold.

Step4-5: now we have done an action, we can see the results and reward. The Q function need to be updated $Q(s, a)$ every step. A new state s' and a reward r are returned to us when we do the random action in step 3. As the bellman equation is applied to update the Q-table.

Algorithm 1 Q-learning algorithm

```

1: Initialize  $Q(s, a)$  arbitrarily
2: for each episode do
3:   Initialize  $s$ 
4:   for each step of episode do
5:     Choose  $a$  from  $s$  using policy derived from  $Q$  (e.g.  $\epsilon$ -greedy)
6:     Take action  $a$ , observe  $r, s'$ 
7:     Update  $Q(s, a) \leftarrow Q(s, a) + \alpha [r(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a)]$ 
8:      $s \leftarrow s'$ 
9:   end for
10: end for

```

Algorithm 1: Q-learning algorithm

The whole Q-learning method is shown in Algorithm 1. The exploration vs. exploitation trade-off provided by an ϵ -greedy strategy is a key aspect of this method. An ϵ -greedy policy is the one that chooses between the optimum. Typically, we want to set $\epsilon=1$ and start to reduce its value to 0 throughout the duration of the episodes. This means that the agent thoroughly studies the environment before settling on a final policy. Q-learning is an off-policy method since it selects action randomly using ϵ -greedy policy and determines the value of the current policy using $\max Q'(s', a')$ policy.

3.2.4 Deep Q-Learning

Deep Q-learning represent the utilization of deep neural network with reinforcement learning. At DeepMind technologies paper[23] introduced the first example of deep reinforcement learning model that effectively learn to control polices using high-dimensional state spaces. The method has been successfully used to train for a diverse range of Atari games utilizing input frames and reward signal. In the games they analyzed, DQN reach a high performance equal to the human behave.

Deep Q-learning function by teaching a neural network to estimate the Q-values of an action at a current state. Instead of keeping (s, x, a) table, we train a model with deep convolutional neural network to predict the table of the Q-values. In summary. The model is built in a way to predict the actions by combining states, actions, and reward pairings with its previous experiences.

Algorithm 2 Deep Q-learning with experience replay

```

1: Initialize replay memory  $\mathcal{D}$  with capacity  $\mathcal{N}$ 
2: Initialize action-value function  $Q$  with random weights
3: for episode = 1,  $M$  do
4:   Initialize sequence  $s_1 = x_1$  and preprocessed sequence  $\phi_1 = \phi(s_1)$ 
5:   for  $t = 1, T$  do
6:     With probability  $\epsilon$  select a random action  $a_t$ 
7:     otherwise select  $a_t = \max_a Q^*(\phi(s_t), a; \theta)$ 
8:     Execute action  $a_t$  in emulator and observe reward  $r_t$  and image  $x_{t+1}$ 
9:     Set  $s_{t+1} = s_t, a_t, x_{t+1}$  and preprocess  $\phi_{t+1} = \phi(s_{t+1})$ 
10:    Store transition  $(\phi_t, a_t, r_t, \phi_{t+1})$  in  $\mathbb{D}$ 
11:    Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $\mathbb{D}$ 
12:    Set

$$y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$$

13:    Perform gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$ 
14:  end for
15: end for

```

Algorithm 2: DQN Algorithm

Algorithm 2 DQN and Q-learning algorithm have a lot in common. With our greedy approach to support exploration where the agent starts with random action to explore the environment, action spaces still need to be defined in a discrete form, although the state space will become continuous.

In contrast to Q-learning, which increases exponentially, the approach of DQN method varies linearly with the state-space.

3.2.4.1 Experience Replay

Used a replay memory which is a reinforcement learning strategy in which we record the agents behaviors at each time-step, $e = (s_t, a_t, r_t, s_{t+1})$ which refer to [state, action, reward, next state] in a data-set $D = e_1, e_2, \dots, e_N$, after that we stored this data in a replay memory across several episodes. The memory is then sampled randomly for a minibatch of experience, which is used to train an off-policy as with DQN. The method of replay memory is used when autocorrelation caused unstable training.

3.2.4.2 Fixed Target Network

When the target and the predicted Q-values are both calculated by the same network, there is a risk of the network becoming unstable due to a feedback loop between the two values. So instead of using a single network, we used two, one for predicting the Q-values and one for estimating the target Q-values called target network. Target networks have the same architecture as the primary network. The weights in the target network are updating to the main model network after a certain number of time steps.

3.2.4.3 DQN Implementation Process Steps

1. Replay memory capacity set to zero
2. Initialize the network with random weights
3. For each episode, initialize the starting state. And for every step take an action by exploration or exploitation. After exploring we choose an action for the next state by the help of the target network where we obtain the max number from the target Q-values, such as:

$$a = \operatorname{argmax}(Q(s, a, \theta)) \quad (3.6)$$

4. Execute selected action observe reward and next state
5. Store transition in the replay memory, transition as $\langle s, a, r, s' \rangle$ refer to [state, action, reward, next state]

6. Policy network should be notified when a batch of preprocessed states is received.
7. Calculate the loss between the Q-value that represent the action from the transition tuple and the target Q-value for this action. The formula used for computing the loss:

$$LOSS = (r + \gamma \max Q(s', a'; \theta') - Q(s, a; \theta'))^2 \quad (3.7)$$

8. Update the weights of the real network from the target network weights after n steps.

It's also worth mentioning that DQN has a number of better applications and modifications, like double DQN[24], Dueling DQN[25], DQN with experience replay[26]

3.2.5 Proximal Policy Optimization (PPO)

PPO is one of the most popular RL algorithms was released by the openAI team in 2017[27]. PPO offer higher convergence qualities than other reinforcement learning algorithms like DQN method, PPO based on creative mix of clipping the policy loss and computing the loss in the form of probability ratio. It stabilizes training by combining trust region optimization with gradient descent, resulting in a loss function that almost ensures that the agent policies will progress linearly. PPO aims to enhance on Trust Region Policy Optimization[28], which establishes an objective function that determines the update step inside a lower-bound know as trust region.

PPO includes gathering several experiences from connecting with the environment and utilizing that collected experiences to update the agent decision-making policy over time. When the policy is updated using this batch of experiences, the pervious experiences are removed, and a completely new batch of experiences is gathered again with the new policy. Because of this, PPO is a on-policy learning strategy in which the trajectories samples obtained can only be used once to update the current policy. The advantage of PPO is to ensure that the new update will not diverge too far from the previous policy. As a result, training is less variable, but there is some bias, but it is smoother and prevents the agent from making senseless action. PPO demands on two model's actor-critic models, both deep neural networks.

Actor Model: The actor model is tasked with determining what action to do under a certain state observed from the environment. In this thesis work we will be using the RGB images frames

from the simulator to be the input to the actor model and gives action as an output. The action can be referred to steering, brake, acceleration.

Critic Model: When the actor model chosen an action to do, the environment should send a reward signal to the agent. The critic model takes this reward as an input, the main function of the critic model is to evaluate the action done by the actor model whether it improved the environment to be in a better state or not, and its input is utilized to improve the performance of the actor model. Critic model generates a numeric value that represents rating for the pervious state action. The agent uses this rating to compare its present policy to a new policy and determine how it might enhance the actor model to do better actions.

Finally, those two models are used for interacting with the environment for a period that define as PPO steps, and collect experiences data, refer to states, actions, and rewards, these experiences will be used for training and to update the policy to a new one.

Figure 3-3 illustrates the workflow of Actor-Critic PPO procedure carried out in a worker at the ending of each episode. The Actor-Critic PPO models starts by collecting a finite mini-batch of consecutive samples from the trajectory memory, the initial tuple is picked at random, while the other tuples must be continuous. Equation (3.8) show the function L^p for the policy gradient RL:

$$L^p(\theta) = \hat{E}[\log \pi_{\theta}(a_t|s_t)\widehat{A}_t] \quad (3.8)$$

Where π_{θ} refer to the current policy, \hat{E} is the average of the finite batch of samples, and \widehat{A}_t is the rating estimator of the action perform at time step t . The generalized advantage estimator (GAE) [29] is used to determine \widehat{A}_t using the discount factor $\gamma \in [0,1]$:

$$\widehat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1}^V + (\gamma\lambda)^2\delta_{t+2}^V \dots (\gamma\lambda)^{U-t+1}\delta_{U-1}^V \quad (3.9)$$

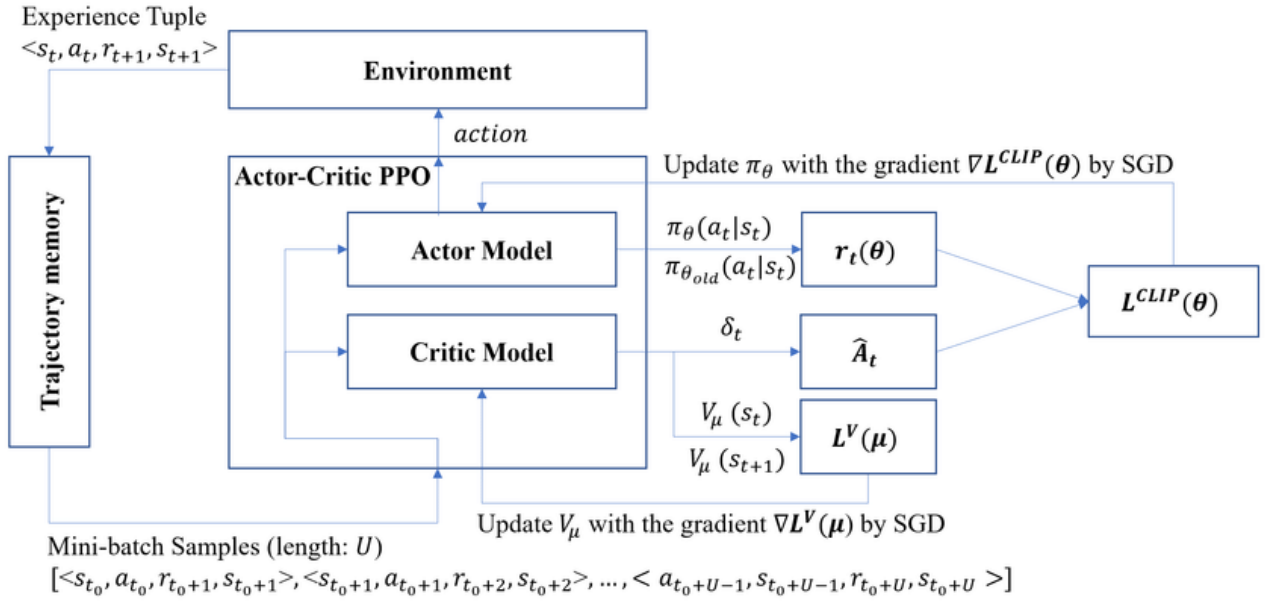


Figure 3-3: Actor-Critic Proximal Policy Optimization Model[16]

Instead of the method stated in equation (3.8), the actor model of TRPO and PPO employs role sampling to calculate the probability of samples collected from the old policy under the new policy to improve it. They aim for the following surrogate objective function to be maximized L^{CPI} :

$$L^{CPI}(\theta) = \hat{E} \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t \right] \quad (3.10)$$

The benefit of this policy gradient reformulation is that we can now utilize gradient descent while putting a trust region condition on the loss function in terms of both the new and old policies.

Where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$. The following equation been proposed, clipped loss function

$$L^{CLIP}(\theta) = \hat{E} [\min(r_t(\theta) \hat{A}_t, \text{clip}(r_t(\theta), 1 - \varepsilon, 1 + \varepsilon) \hat{A}_t)] \quad (3.11)$$

The parameter ε define how much the new policy may differ from the old policy each update, toward better polices. The *min* term calculates the minimum of both clipped and unclipped, in the equation the old loss $L_{\theta_{old}}^{IS}$ is the unclipped objective, where the probability ratio interval $[1 - \varepsilon, 1 + \varepsilon]$ clipped the objective. So that cautious modifications may be made. As long $A > 0$ and $r_t(\theta) > 1$, or both are negative. The *min* term makes it such that our updater is controlled by the clipped aim. As an alternative if $A < 0$ and $r_t(\theta) > 1$ or $A > 0$ and $r_t(\theta) < 1$, we shall push the existing policy parameters

through the parameter of the preceding policy, therefore canceling the modifications made in the previous update.

Using differentiable loss function enable us to apply gradient descent to the optimization issue. As compared to TRPO, the parameterize of critic can be now with θ . An additional value function loss $L^{VF} = (V(s_t, \theta_v) - R_t(\tau))^2$ is introduced by PPO in order to improve the critic. The entropy is also included by adding the term, $-\frac{1}{2}(\log 2\pi\sigma^2) + 1$. Finally, the final loss function is written as follow:

$$L^{CLIP+VF+S}(\theta) = -\hat{E}_t \left[L^{CLIP}(\theta) - \alpha L^{VF}(\theta) - \beta \frac{1}{2}(\log 2\pi\sigma^2) + 1 \right] \quad (3.12)$$

Algorithm 3 PPO, Actor-Critic Style

```

1: for iteration=1,2,... do
2:   for actor=1,2,...,N do
3:     Run policy  $\pi_{\theta_{old}}$  in environment for  $T$  timesteps
4:     Compute advantage estimates  $\hat{A}_1, \dots, \hat{A}_T$ 
5:   end for
6:   Optimize  $L^{CLIP}$  wrt  $\theta$ , with  $K$  epochs and minibatch size  $M \leq NT$ 
7:    $\theta_{old} \leftarrow \theta$ 
8: end for

```

Algorithm 3: Actor-Critic PPO Algorithm

Algorithm 3 show a loop executes the gradient update on random minibatch samples across K epochs, M represent the size of the minibatch and N the number of environments across the model. The estimator advantage \hat{A}_t is computed in the meanwhile.

4 Carla Simulator

4.1 Introduction

CARLA is an open-source driving simulator that has been used for training and validation autonomous driving algorithms. It has flexible settings with the help of an API. It is developed in C++ and uses the Unreal Engine to create the driving scenarios. It includes digital assets and full control over map actors, as well as environmental conditions, map production, and server-client communication. CARLA also provides a Python API module, so the simulator works as a server, while the Python API manages the client-server interaction see (Figure 4-1).

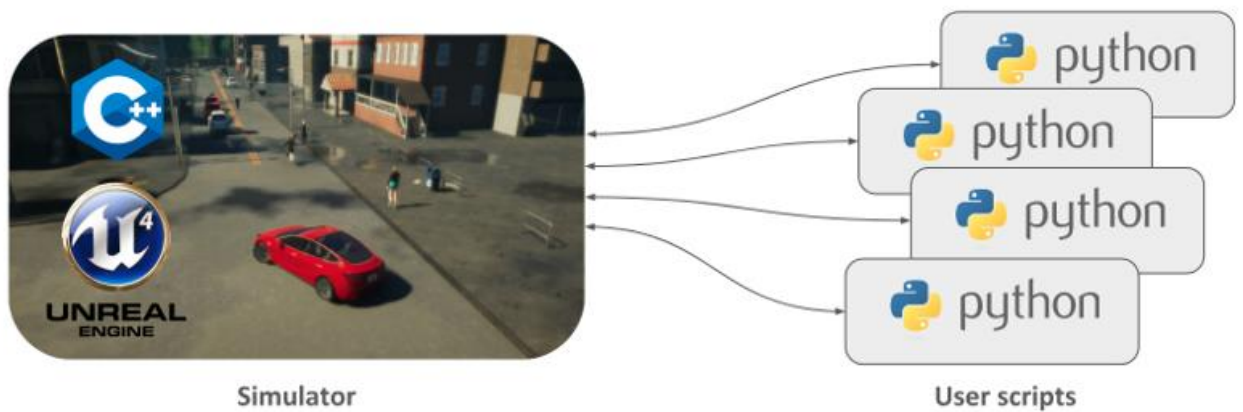


Figure 4-1:CARLA client-server communication[2]

The python API provide access to simulation features. Python script may be used to obtain raw data from CARLA sensors installed to the self-car, analyze the data, calculate all the parameters required by the controller, and transmit the commands for throttle, brake, and steering to the CARLA simulator

4.2 Configuration of the simulation and of the actors

The interconnection with the server, as well as all the simulation and actor settings, must be completed Before the control algorithm start. A variety of functions are available here, including retrieving information about the environment, spawning actors at random positions and orientations, attaching sensor devices to the eye of the vehicle, deleting, or spawning more characters. For

spawning a car in the Town3 (Figure 4-2) we run the python script *manual_control.py* see Figure 4-3), control the car with the keys WASD.



Figure 4-2: Carla simulator Town3



Figure 4-3: Vehicle Spawn in Town3

4.3 Synchronous Mode Configuration on CARLA Simulator

One of the main aspects of this study is to build an algorithm on CARLA for image collection when we are driving in the Town4 simulation. The primary goal of collecting these data is to use them as training datasets for the variational encoder. As a result, the variational encoder will be used in future work, as part of the training pipeline for the proximal policy gradient reinforcement learning method. This thesis will focus mainly on the data collection and training processes.

The world has been received from the client object using *get_world()* function and the synchronous mode has been active as a result. After being set, the simulation is delayed at the end of each step until a tick message has been successfully received. This signal may be transmitted to the simulator by using the *world.tick()*. This is very helpful when dealing with our sensors that are based on cameras. The simulator must operate with a constant time-step to utilize the synchronous mode consistently and predictably. The step size in our mode is 0.5 seconds. An important step of the configuration that the actor information can be accessed by executing *world.get_spectator()* function. Where it's easy to adjust the spectator location and orientation, for instance get the desired perspective of the ego car. The ActorBlueprint class is used to create the actor's information objects, it includes all the tags and characteristics for every single actor in CARLA. The *get_blueprint_library()* function used to access the list of all blueprints, by executing *blueprint_library.find(tag)*, where tag here identifies a blueprint with three words separated by a dot, there are seven world in CARLA. In this thesis. The map 4 has been chosen (Figure 4-4), And the spawn vehicle type was Lincoln MKZ2017 (Figure 4-5)



Figure 4-4: CARLA simulator: Town 4



Figure 4-5: Vehicle(Lincoln MKZ2017) spawned for starting collecting data images, 10000 images as limited

The Python code below describe the main functions on CARLA for the configuration of the simulation and of the actors:

```
1 def main():
2
3
4     # connect to client
5     client = carla.Client('localhost', 2000)
6     client.set_timeout(20.0)
7
8     world = client.get_world()
9
10    # Synchronizing a camera with synchronous mode.
11    settings = world.get_settings()
12    settings.synchronous_mode = True
13    world.apply_settings(settings)
14
15    try:
16        m = world.get_map()
17        start_pose = random.choice(m.get_spawn_points())
18        waypoint = m.get_waypoint(start_pose.location)
19
20        blueprint_library = world.get_blueprint_library()
21
22        #Create vehicle and attach camera to it
23        vehicle = world.spawn_actor(
24            random.choice(blueprint_library.filter('vehicle.lincoln.mkz2017*')),
25            start_pose)
26        actor_list.append(vehicle)
27        vehicle.set_simulate_physics(False)
```

```

28     # Get spawn location
29     camera_bp = blueprint_library.find('sensor.camera.rgb')
30     camera_transform = carla.Transform(carla.Location(x=1.6, z=1.7))
31     camera_bp.set_attribute('image_size_x', '1280')
32     camera_bp.set_attribute('image_size_y', '720')
33
34
35     # Spawn the camera and attach it to the vehicle
36     camera_rgb = world.spawn_actor(
37         camera_bp,
38         camera_transform,
39         attach_to=vehicle
40     )
41     actor_list.append(camera_rgb)
42
43     # Make sync queue for sensor data.
44     image_queue = queue.Queue()
45     camera_rgb.listen(image_queue.put)
46     world.tick()
47     world.wait_for_tick()
48     frame = None
49
50     # save image to the PC disk
51     waypoint = random.choice(waypoint.next(1.5))
52     vehicle.set_transform(waypoint.transform)
53     image.save_to_disk('_out/%08d' % image.frame_number)
54
55
56     # MAIN LOOP
57     except KeyboardInterrupt:
58         print('\nExit by user.')
59     finally:
60         if args.synchronous_mode:
61             print('\ndisabling synchronous mode.')
62             settings = world.get_settings()
63             settings.synchronous_mode = False
64             world.apply_settings(settings)
65
66     print('destroying actors.')
67     for actors in actor_list:
68         actor.destroy()
69     print('done.')
70

```


5 Deep Reinforcement Learning (DQN) in Carla Simulator

After we discussed the DQN algorithm of reinforcement learning in section (3.2.4). As well Carla configuration and setup in section (4.2). We will get started in this chapter with implementing the DQN algorithm on Carla Simulator. The main propose in this chapter is to reach a self-driving car with a driving policy. The strategy depends on training a deep neural network to predict the best action to perform and extract the maximum reward from a wide state space.

5.1 Setup

The implementation was written in Python 3.7 and TensorFlow 1.14 with Keras 2.2.4. Carla version was 0.9.5. The code configuration and setup Carla was done on my own PC using PyCharm IDE. Due to the low capability of my local PC, the training process was carried out on a Linux desktop provided by the supervisor which was running Ubuntu version 20.4 supported with GPU, the connection was done remotely by OpenVPN and PuTTY.

5.2 DQN Carla Environment

For training the convolutional neural network model we must first define the dataset. In our study, the dataset was gathered in different way, where we utilize Carla to emulate the driving environment server, where we can add a set of sensors on the vehicle to represent real-world self-driving car, sensors such as LIDAR, and cameras. These data sensors record the environment, and then forward the data to a function that pre-processes it. Additionally, we define the reward condition, which is whenever the agent avoids colliding.

Carla environment

As discussed before in section (4), CARLA provide a python API module, the simulator works as a server, and the Python API manages the client-server interaction. The main idea behind our strategy is to have the environment (server) and the agent (client). in section (4.3) we discussed three primary ideas of CARLA. First the Actor which include vehicles, people, and sensors, all of which play a part in the simulation and might been moved. Second the Blueprints which is used to define the properties of an actor before it can be spawned. And finally, we have the World which displays

the presently loaded map and includes tools for turning the blueprint into a live actor. The following python code demonstrates how to spawn a *Tesla model3* car using CARLA primary functions. first, we connect to CARLA, obtain the world, and access the car blueprints. In order to capture the area around the environment, we first add an RGB camera sensor on the car, and then use the blueprint function to set the attributes position of the added camera sensor. This spawning car will be the agent that will interact with the environment.

```
1 IM_WIDTH = 640
2 IM_HEIGHT = 480
3 actor_list = []
4 collision_hist = []
5 try:
6
7     # connect to client
8     client = carla.Client('localhost', 2000)
9     client.set_timeout(2.0)
10
11     world = client.get_world()
12     blueprint_library = world.get_blueprint_library()
13
14     #filter a vehicle of type tesla model3
15     bp = blueprint_library.filter('model3')[0]
16     print(bp)
17
18     #pick a spawn point randomly
19     spawn_point = random.choice(world.get_map().get_spawn_points())
20
21     #spawn the car
22     vehicle = world.spawn_actor(bp, spawn_point)
23
24     #control the car
25     vehicle.apply_control(carla.VehicleControl(throttle=1.0, steer=0.0))
26
27     #add the vehicle to our list of actors that we need to track
28     actor_list.append(vehicle)
29
30
31     # get the blueprint for this sensor, spawn location
32     blueprint = blueprint_library.find('sensor.camera.rgb')
33     # change the dimensions of the image
34     blueprint.set_attribute('image_size_x', f'{IM_WIDTH}')
35     blueprint.set_attribute('image_size_y', f'{IM_HEIGHT}')
36     blueprint.set_attribute('fov', '110')
37
38     # Adjust sensor relative to vehicle
39     spawn_point = carla.Transform(carla.Location(x=2.5, z=0.7))
40
41     # spawn the sensor and attach to vehicle.
42     sensor = world.spawn_actor(blueprint, spawn_point, attach_to=vehicle)
43
44     # add sensor to list of actors
```

```
45 actor_list.append(sensor)
```



Figure 5-1: Car spawned with camera sensor position at 2.5 forward and 0.7 up

5.3 State, Action, Reward

In addition, to the RGB sensor, we add a collision sensor to detect if the agent crashed or not. For that we create a reset method to restart the environment depending on some sort of flags. As well we wait at each episode a few seconds before activating the collision sensor to avoid collision when the car spawn in the first seconds. If a collision happen the environment will restart and return a reward of -200.

The agent states are the exact RGB frames obtained from the RGB sensor of the spawned car in the CARLA environment. The agent chooses one of the three possible actions, left, right, or straight depending on the state. Action in return, alter the state of the agent to be in a new frame. The environment deliver reward to the agent based on the action done. +1 if the frame speed > 50

KMH, -1 for frame driving < 50 KMH, and -200 for a collision. In the output of the model, we predict a Q-values in a shape of array of three index [left, straight, right]:

$$IF\ ACTION = \begin{cases} 0\ steer\ left \\ 1\ straight\ forward \\ 2\ steer\ right \end{cases}$$

The vehicle speed is converted to KMH to prevent the agent from driving in a wide circle:

$$KMH = 3.6 * \sqrt{V_x^2 + V_y^2 + V_z^2}$$

As well the reward function defines like that:

$$REWARD = \begin{cases} -200\ collision \\ -1\ if\ KMH < 50 \\ +1\ IF\ KMH > 50 \end{cases}$$

5.4 DQN Agent Model

One of the main goals of this thesis is to build a model that teach a car to drive by itself in the CARLA environment. Our agent model of 64x3 convolutional neural network (CNN), contain 4 hidden layers, 3 convolutional layers each layer consist of 64 filter and kernel size of 3x3 plus 3 average pooling, and a flatten layer which is the fourth hidden layer. The given model which is basically a CNN model will be used to analyze the frames data obtained from the RGB camera sensor connected to the car in our environment. In the output, the flatten layer will be added to make the output linear before we pass it to the dense layer, as well three neurons will be added to the output layer which represent our action direction left, right, straight.

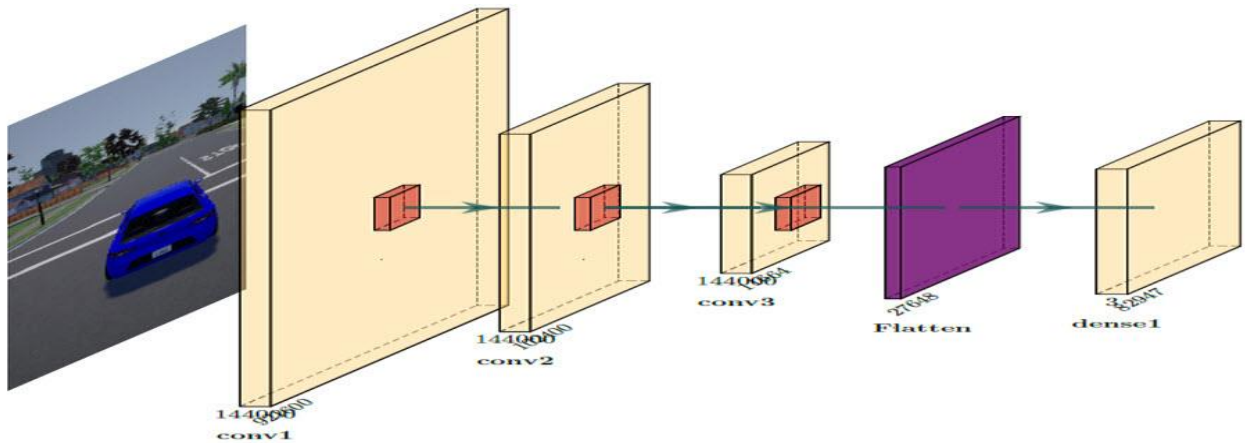


Figure 5-2: 64X3 CNN Architecture

Training

In this section, we implement all the methods covered before. Instead of using a Q-table to find values, we use DQN to output a prediction from a model, and instead of maintaining Q-table, we train our model. When we interact with CARLA environment at the starting time, we take actions randomly using the concept of exploration (section 3.2.2), after that we used the function *.predict()* to decide our next move. *.predict()* function will return three float values, which are our Q-values that correspond to actions. Next, we will execute an argmax on them, just as we did with the values in our Q-tables. Finally, using the update Q-values, we will run a *.fit()* function to update our model. Although we only want to change one of the Q-values, we will really be fitting the 3 of them.

Our model begins randomly and made update at every step, every episode. Imbalances occur which the model finds unstable and slower. As a result, we create a second model called (target model) that inherits the weights from the base model every n episode and is used to predict the future Q-values.

Implement the concept of replay memory that save the experiences (current state, action, reward, new state) at every step of an episode. For training a neural network usually we used batches. Where the agent starts collecting a finite mini-batch of consecutive samples from the replay memory. Meanwhile, a random selection of 5000 prior actions from the replay memory will fit our model.

5.5 Experiments

Initially, we chose to keep the challenge simple by trying to limit the actions of the agent to only do one of the three actions: turn left, turn right, or go straight. The first method in DQN study it was to choose the Xception model to train our agent. Xception is a model in Keras library with 71 hidden layers.

For each episode, the agent spawned at a different position in the CARLA environment. Where we chose not to spawn the agent at the same position in each episode, because of our replay memory that take batches of sample states (states define the images collected by the RGB sensor) and spawning the agent at the same location will gives each episode same group of states. In this way the agent will be train under the same set of states, which should result extremely high Q-values for those states, where the agent won't be able to function effectively with those Q-values in other states.

After running the script of the Xception model for 10000 episodes on the ubuntu server, we test the model to observe how it perform. In the 10000-episode run, we reach an accuracy with 96%(Figure 5-3) which is over high, where the agent was doing nothing just circling around in the same place, the loss function and the Q-values exploded. As result, the Xception model fail in the test, its complicated to choose a model of 71 hidden layer with reinforcement learning to train the agent to drive, deep networks models cause more overfitting. After that the solution came by changing the model to simple one. For this reason. We choose the 64x3 CNN model discuss in section (5.4). In the 64x3 CNN the trainable parameters are 3 million which is significantly less than the Xception model of 23 million. In reinforcement learning it's difficult to train an agent with tens of millions of parameters.

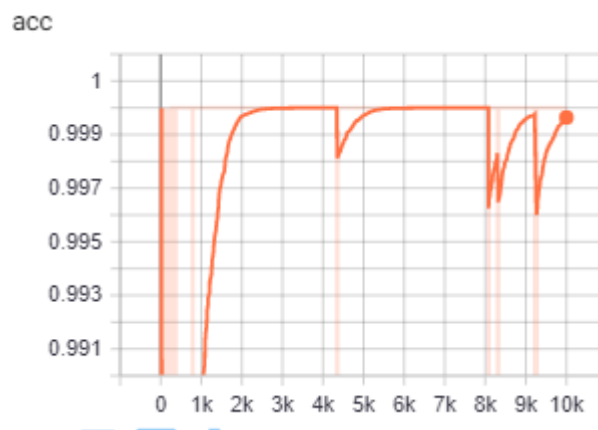


Figure 5-3: Xception Model Accuracy

The first experimentation on the model of 64x3 CNN was to train the agent with different scenarios, by changing the weather randomly overtime from a clear and sunny day to dark and wet rainy day. Additionally, spawning another 60 agents, those agents refer to vehicles of different sizes, such as cars, trucks, and motorbikes. The model fist was trained for 1000 episodes alone to determine how well it performed in various conditions. As a result, we noticed several issues where the agent falls exactly on the top of another car when we spawn it, which would stop the episode before it had a chance to understand the environment. Even though the model was unable to get the agent to travel on a straight path without clashing with other vehicles in the environment. That's because the agent fails several times before begins to learn and interact with the environment in the 1000 episode.

Second experiment in the training phase was to remove the dynamic traffic by removing the 60 agents from the environment and just keep the dynamic weather. The outcome of this experiment was better than the previous one. As before the dynamic whether settings will change, lighting and

weather conditions change randomly over time. Episodes trained under good lighting conditions improved over time, but those who were trained under dark and overcast situations failed because the RGB images from the camera sensor was different than the lighting ones, and they need additional image processing to produce reliable results. This additional image processing was an unnecessary cost for us since our computing resources were restricted, and an agent that had been trained for 1000 episodes was unable to provide acceptable results.

Finally, in order to keep things simple and decrease complexity, we chose to maintain the environment at normal sunny conditions without dynamic traffic. We saw a stable enough model after training the agent for 50000 episodes in these settings, which run very well on straight path road and even turning left and right in a decent way.

5.6 Results

As discovered before that the Xception model failed in training the agent where both the loss and Q-values were exploded. Reinforcement learning differs significantly from supervised learning, due to the fact supervised learning is assumed to be perfect ground truth. All the images we give it, as well the labels, are intended to be totally accurate. While in RL we are training a model also looking for Q-values. It's much more complex process to handle it. For that we intended to switch the model to 64x3 Convolutional neural network.

The 64x3 CNN model turned out to give the best performing, the model trained many times in the server provide by the supervisor, first for 20000 episodes after that we reload the checkpoint and continue training until we reach 35000 episodes. Finally, after 8 days we reach 50000 episodes. As a result, as shown in (Figure 5-4) the accuracy reaches over the 50k episode 90% approximately.

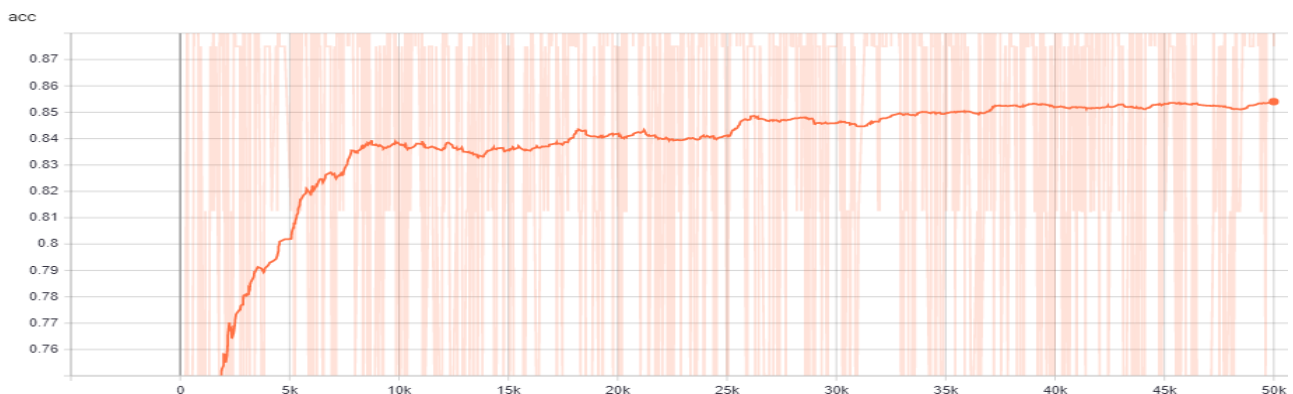


Figure 5-4: 64x3 CNN Model Accuracy reach at 85%

Figure 5-5 illustrate the concept of exploration discussed in section (3.2.2). at the start of running the model the exploration take place where our agent will take actions randomly to explore the environment. As a result, we utilize an epsilon number that allows the agent to act randomly. At the beginning the epsilon value will be 1, which means that there is a high probability to take actions randomly rather than using the neural network to predict the Q-values. As the model learn over the episodes time, we can start depending on it by taking the action from the Q-tables, and epsilon will decrease to reach the minimum value 0.01.

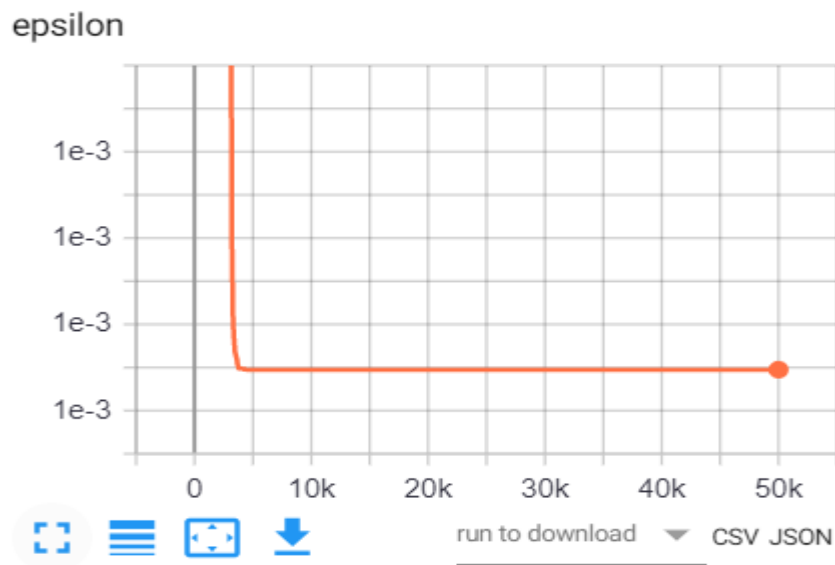


Figure 5-5: 64x3 CNN Model Epsilon



Figure 5-6: 64x3 CNN Model Loss

Figure 5-7: 64x3 CNN Model Reward Average

Figure 5-6 show the loss of our model where there were no explosions, at the last episodes we reach a loss of 8% approximately. While in (Figure 5-7) represent the average reward of the agent model improved gradually, the reward average was expected to reach a higher value over the 50k the graph show that the agent needs much time to learn and improve so we could train in more episode to increase it.

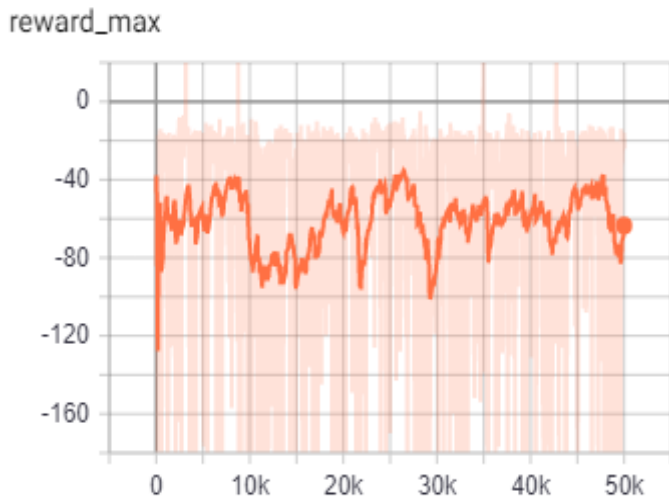


Figure 5-8:64x3 CNN model Reward max

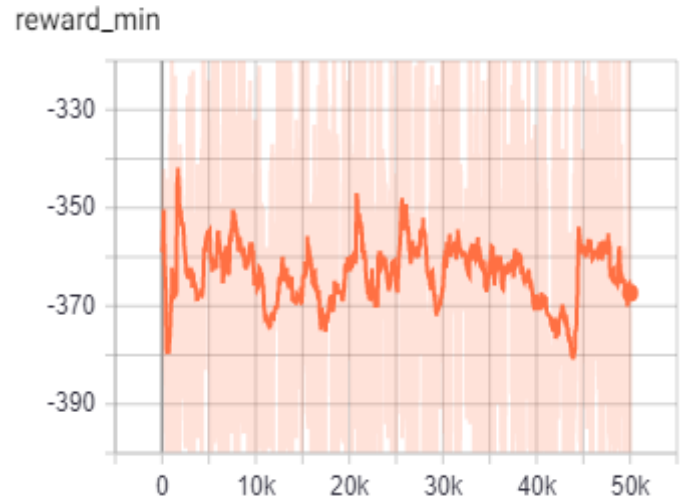


Figure 5-9: 64x3 CNN model Reward min

Figure 5-8 show the reward max over the episode as we can see the reward max it's not going up above zero just reach -20 which also need more time, also (Figure 5-9) represent the minimum reward the worst action the agent did, trends didn't appear to shift much, which isn't surprising. Sometimes the automobile is just thrown into an unfavorable situation. Finally, we can deduct from the reward average that our model improved so slowly which it needs more time.

The results did not meet our expectations, but we did improve our model slightly. As a solution, we could train the agent with different model and parameters, as increase the episodes. And by going further, we could implement another algorithm of Reinforcement learning like PPO, which have higher convergence qualities than DQN.

6 Proximal Policy Optimization (PPO) in Driving-Like Environment

In this part of the thesis, we implemented and tested the proximal policy optimization (PPO) approach in a driving-like environment. As discussed in section (3.2.5), PPO is a model-free, policy gradient-based reinforcement learning method that makes use of first-order trust region criterion to avoid variance, it learns to perform a diverse variety of continuous control tasks in a multi-joint environment. In this section, we will go through the specific's implementation of PPO on CarRacing-v0.

6.1 Setup

The implementation was written in Python 3.7 with TensorFlow 1.14. It was required to install the OpenAI gym library developed by Python, gym have a collection of environments one of them is CarRacing-v0, it used to workout RL algorithms. The training process run on my Laptop, SSD hard drive, with 2GB video memory, and 4-core CPU. One of the reasons for choosing CarRacing-v0 gym environment that it doesn't require high GPU and high video memory, it was suitable with the low PC capability.

6.2 Environment

Section (5) shows the implementation of reinforcement learning algorithm DQN on CARLA simulator, where we saw that CARLA used for the task of RL. CARLA is the most realistic urban driving simulator. However, CARLA is not a particularly light-weight environment because it focuses more on high-quality visuals and the fact that it runs on the relatively pricy Unreal Engine 4. Simple theories are more difficult to test in CARLA environment because of the high level of complexity, including traffic signals, lane markings, vehicles, and pedestrians. Instead of CARLA. The OpenAI CarRacing-v0 was used to test the models of the PPO algorithm. OpenAI gym, a collection of reinforcement learning scenarios, includes CarRacing-v0.

CaRacing-v0 is a top-down 2-dimensional ("birdseye") perspective of car on a procedurally created racing environment. Three continuous actions values of (γ, a, b) that control the car where:

- $-1 \leq \gamma \leq 1$ the steering angle
- $0 \leq a \leq 1$ acceleration

➤ $0 \leq b \leq 1$ break

The fundamentals of CarRacing-v0 are modelled using Box2D, a specialist 2D physics simulator which make the car as realistic as possible. The environment considers a number of features of the car dynamics, including the fact that it is a rear-wheel drive vehicle with ABS speed sensors, and a gyroscope. It also replicates ground friction, so quick turns and brakes on grass will cause the vehicle to slide.

For each episode, the environment creates a random track, in case the agent obtains a score of 900 over 100 iterations, the environment would consider to be “solved”. For each segment of the track the agent gets a reward equal to $1000/N$, where N refer to the total number of segments in the environment. Every frame, the agent loss 0.1 points, equating to a loss of -5 points per second. This signifies that the agent highest score is $1000 - 5t$, where t defines the shortest time for this specific track in seconds.

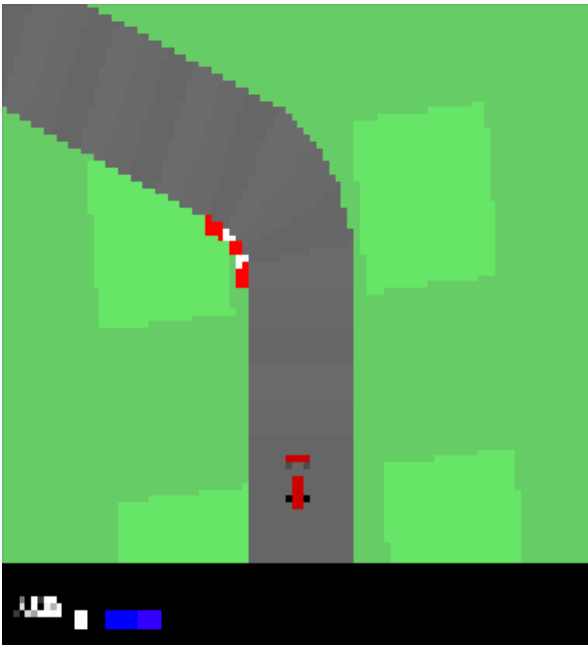


Figure 6-1: State Space seen by the agent

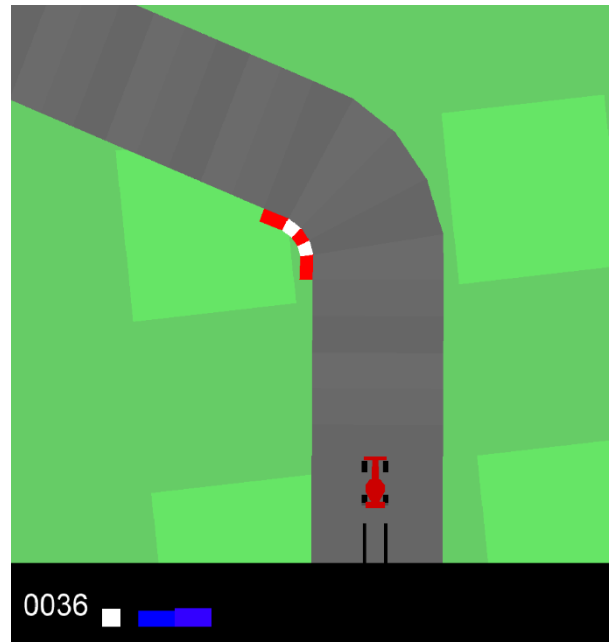


Figure 6-2: Screenshot of CarRacing-v0 environment

Figure 6-1 shown a sample of CarRacing-V0 state space. The state space in refer to the RGB images of 96x96 pixels, in which the speedometer, ABS sensors, steering angle sensors, and brake sensors are recorded as bars in the bottom 96x12 pixels of the image, respectively. With this approach, the agent will learn to evaluate the measurements actions immediately from the image, so we just need to train a convolutional neural network to teach the agent to improve his policy under a

given state. In the case of some tests, we intended to do, this seemed a little restrictive. As a result, a new modified version of the environment CarRacing-v0 will be created with the name CarRacing-v1, where it produces the following extra measurements:

- ABS sensor values for each wheel
- Steering angle
- Angular momentum

6.3 Algorithm

The main structure of the PPO algorithm is shown in Algorithm 3. It's important to remember that PPO operates by optimizing the current policy π_θ in relation to how much it deviates from the previous policy $\pi_{\theta_{old}}$. In order to sample the training data for an optimization phase, it is necessary to run single or several environments in parallel with the previous policy. The *SubprocVecEnv*, an OpenAI gym class, has been used to execute N environments in parallel, where each environment has its own thread. Before calculating the rate of the advantage estimator and updating the policy, we employ a T-step temporal difference, where T specifies the number of steps we travel along a trajectory. The advantage estimator \widehat{A}_t is determined by generalized advantage estimator (GAE) as shown in equation (3.9), where lambda λ is used to reduce the variation in the training phase and make it more stable, in our paper the value of λ will be 0.95 provides us the possibility of acting both in the near and long term, and gamma γ a constant value known as discount factor reduce the value of the future state in order to place greater emphasis on the current state. When all T steps for N environments are completed, we get NT samples from which we randomly select minibatches of size $M \leq NT$ for K number of epochs. For these minibatches of $(s_{nt}, a_{nt}, R_{nt}, \widehat{A}_{nt})$ -tuples collected from the trajectory memory are fed via an actor-critic network that uses Adam gradient descent to optimize the parameters in accordance with the $L^{CLIP+VF+S}$ loss function stated in equation (3.12). Remember that the $L^{CLIP+VF+S}$ provides a clipping parameter ϵ with the L^{CLIP} , as well value loss scaling factor α and entropy loss scaling factor β . In the Next part, we will go into further details on the construction of the two different model we evaluated.

6.4 Experiments

Reinforcement learning agents that use images as input, often represent their state space by adding consecutive frames. In this model, we chosen to generate a frame stack consisting of four consecutive frames, which would be used as the input to the model.

6.4.1 Image Preprocessing

The input to the model is 4 images of size 96x96 RGB stacked on top of each other, resulting a state space value of $(96 \times 96 \times 3 \times 4 = 110,592)$. The 96x96 RGB frames were cropped and converted into 84x84 grayscale to optimize the model and minimize the size of the state space, resulting a value of $(84 \times 84 \times 4 = 28,224 \text{ state value})$. As a result, we crop both sides left and right with 6 pixels to remove the part that are not needed to the agent, in addition with cropping the bottom part with 12 pixels, those 12 pixels encode the dashboard parameters, which should not be required when the consecutive stack frames have the information needed to deduce these parameters. Here is the code showing the step of image processing:

```
1 def crop(frame):
2     # Crop to 84x84
3     return frame[:-12, 6:-6]
4
5
6 def rgb2grayscale(frame):
7     # change to grayscale
8     return np.dot(frame[..., 0:3], [0.299, 0.587, 0.114])
9
10
11 def normalize(frame):
12     return frame / 255.0
13
14
15 def preprocess_frame(frame):
16     frame = crop(frame)
17     frame = rgb2grayscale(frame)
18     frame = normalize(frame)
19     frame = frame * 2 - 1
20     return frame
```

6.4.2 Network Architecture

Actor-Critic outputs are fed into a convolutional neural network CNN, which has a fully connected layer for each one. The paper reinforcement learning with A3C [30] influenced the

convolutional layers of the network design. In this paper the result was that deeper networks were shown to have no effect on the agent training, also deeper networks are more difficult to train and converge on. Figure 6-3) represent the frame stack model, we choose a shallower convolutional network because its fast in training and perform better or equally to deep networks. The first convolutional layer consists of 16 8x8 filters with a stride 4, and the second convolutional layer include 32 3x3 filter with a stride 2. Both layers adopt leaky ReLU activation function, $f(x) = x$ if $x > 0$ otherwise $0.01x$. It's important to keep in mind that the filters are large in comparison to those seen in deep networks like ResNet [31], Because shallow convolutional networks need big filters to expand their receptive fields, this is the case.

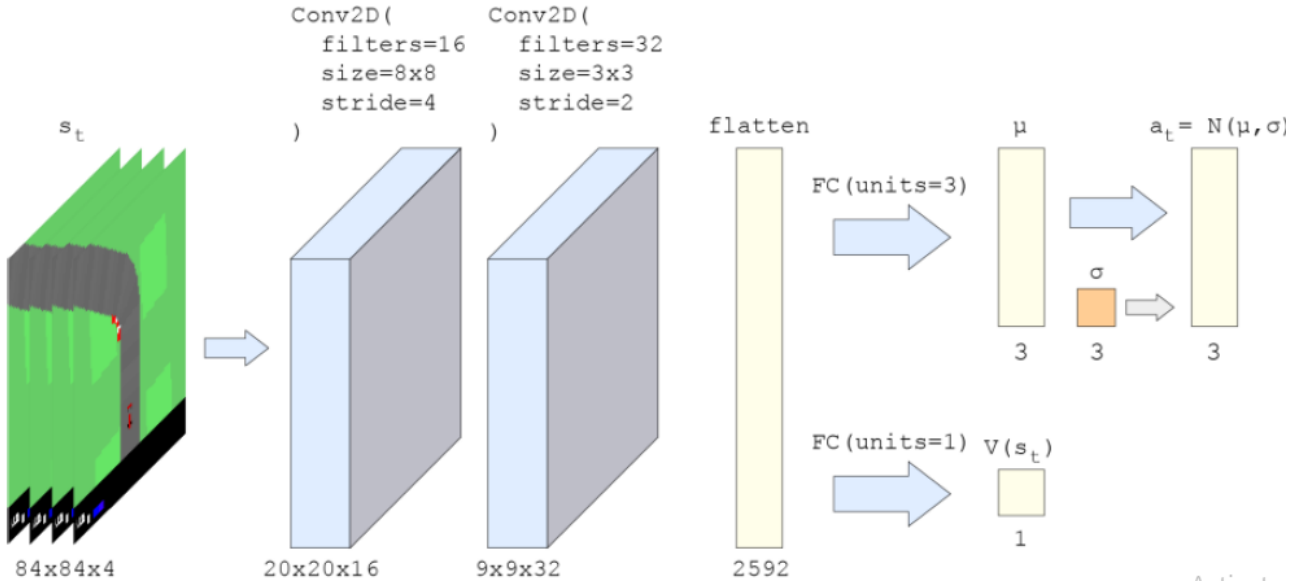


Figure 6-3: Frame Stack Model Architecture With Four Stacked Frames

Two fully-connected layers interact and share the feature maps generated by the convolutions. In the first fully-connected layer, the agent's actor is identified. It contains three output units, each representing the mean of one of the agents three actions(γ, α, b). Because CarRacing-v0 is an environment with a continuous action space, we describe the actions by a multivariate normal distribution, $\alpha_i \sim \mathcal{N}(\mu_i, \sigma_i)$.

The critic model shares the flattened feature maps of the convolutional layer. Expressed also by fully-connected layer with a single output value, which represent the value of being in a state s , $V(s)$.

6.4.3 Optimization

To compute the log probability $\log \pi_\theta(a|s)$ an action a at the state s under the policy π_θ , we used the action means and standard deviations from the network. In order to determine the clipped loss, L^{CLIP} , we must first calculate $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$. The logarithm quotient rule be used

$$\log \pi_\theta(a_t|s_t) - \log \pi_{\theta_{old}}(a_t|s_t) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} = r_t(\theta) \quad (6.1)$$

to do this:

Finally, we calculate the combined loss function, $L^{CLIP+VF+S}$, and use Adam gradient descent to optimize it.

Hyperparameter	Value
T-Steps	128
GAE parameter λ	0.95
Discount Factor γ	0.99
Clipping Parameter ϵ	0.2
Learning Rate	3e-4, decreasing by 0.85 each 10000 epoch
Value Loss Scale α	0.5
Entropy Loss Scale α	0.01
Number Of Epochs κ	10
Batch Size M	128
Number Of Environment N	8

Table 1: Hyperparameter Values

6.4.4 Recurrent Model

In the recurrent model shown in (Figure 6-4), we aimed to analyze whether the agent performance would improve if it could use one frame instead of 4 frames to the input of the model. In the recurrent model, the architecture of the model was different where we merged the latent features space vector ϕ_{t-1} from the last step with the current latent features ϕ_t .

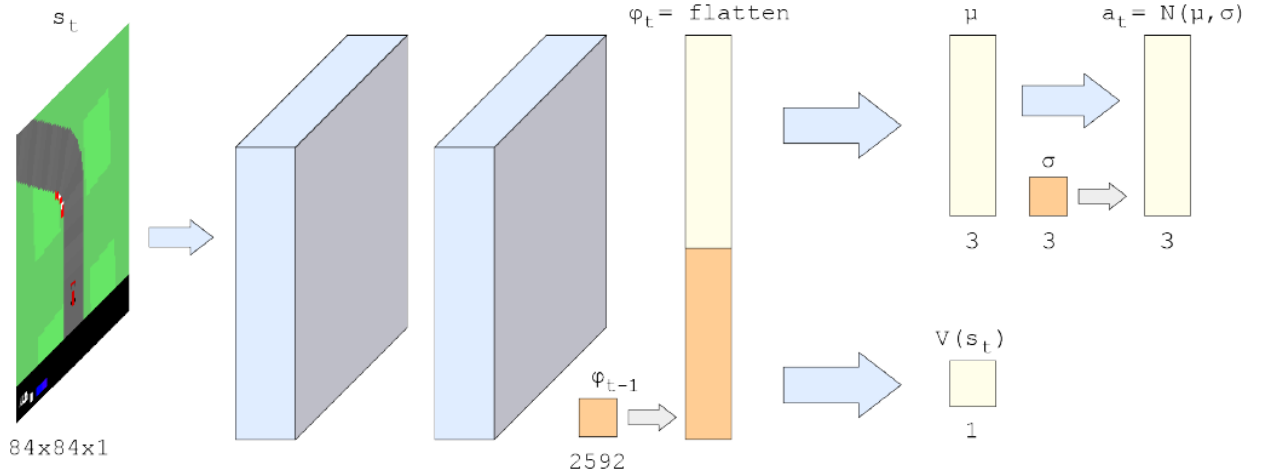


Figure 6-4: Recurrent Model for PPO algorithm

6.4.5 Variational Autoencoder

The use of a variational autoencoder, as shown in article [12], was important in reducing the overall training time. Essentially, the VAE will work as a feature extractor, disentangling and making more predictable the state space on which our agents will be able to see the frame in better way.

In future work, which will not be included in this research. We will analyze the implementation of VAE in our agent's learning pipeline, as well as how the encoded latent representation supports the agent in learning faster in the environment. For the sake of this thesis, we will just look at the areas dealing with data collection and model training. In future challenges, we will implement the VAE into the PPO's training pipeline.

The VAE model used by CNN is based on the work of paper [31]. This model begins with an encoder consisting of four convolutional of 32, 64, 128, 256 filters, with 4×4 kernels, a stride of

2, and ReLU activations. The output of the last convolutional layer in the encoder is observed and fed into two parallel fully-connected layers. With the output of the parallel layers, we choose a vector z from a gaussian distribution, which we can then transmit the vector z into the decoder for processing. The decoder begins with a fully-connected layer that is responsible for resizing z to the same size as the output of the last convolutional layer of the encoder. We do this in order to make it simpler to restore the photo to its original size using transposed convolutions later in the process. We use 4 transformed convolutional of 128, 64, 32, 1 filter, with 4x4, 5x5, 5x5, 4x4 kernels size respectively. The kernel sizes were chosen this manner so that the output of the final convolution would be a $w \times h$ picture. The VAE was improved by reducing the mean squared error (MSE) or the binary cross-entropy (BCE) between input and output.

The dataset for training the VAE were collected by driving about in the CarRacing-v0 environment manually and gathering images from the front camera sensor as we went. We gathered 10,000 photos across all environments and divided the dataset into 9000 training images and 1000 validation images.

For the purpose of training the VAE, we train the model to reconstruct the input picture from a sampled Gaussian latent vector while attempting to minimize the reconstruction loss. To achieve this goal, we use the Adam optimizer, with a learning rate of α and a rate decay of γ , as well a batch size of N . (Figure 6-5) show the decreasing of the reconstruction loss of our VAE model during the training process.

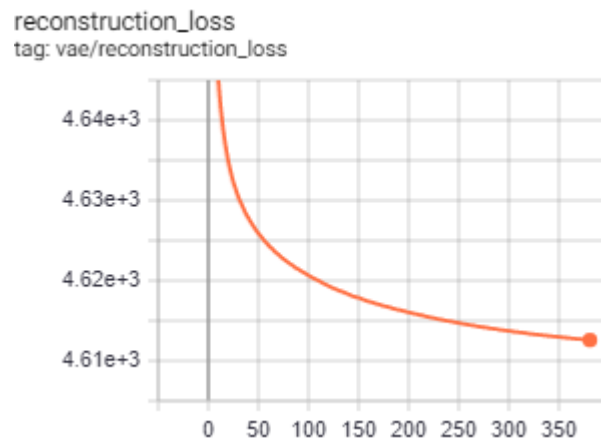


Figure 6-5: evaluate the reconstruction loss of VAE train model

6.5 Results

Figure 6-6 show the cumulative reward of each model over the number of epochs. During training, the evaluation was running every 200 epochs. The present policy is evaluated by running it on a single agent from start to finish and collecting the total reward.

Frame stack model

The use of frame stacking in CarRacing-v0 shows that it is a well-established approach in deep reinforcement learning. The frame-stack model achieves highest score than the recurrent model, a score of 800 on its best run. The vehicle maintains a consistent speed and can perform fast curves while staying on the road without exhibiting any abnormal behavior. On the other hand, recurrent model was having the ability to recover from lower states, which is important for self-driving cars. Since the exiting network is relatively shallow to optimize training time. The two models can be learned to solve in a better method by making them a bit deeper with adding a more convolutional layers with max-pooling or fully connected layers.

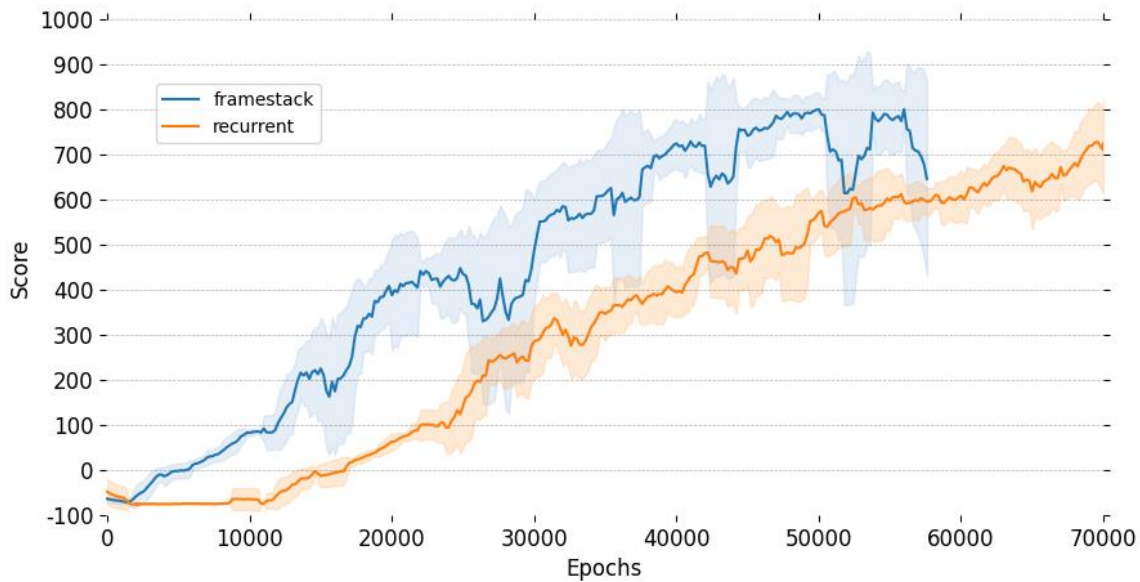


Figure 6-6: Cumulative reward of the models over number of epochs

Recurrent model

Initially, the recurrent model performed poorly, but it finally obtained results that were comparable to those of the frame stack model as shown in (Figure 6-6). The recurrent model has less variance in its performance measurements compared to the frame stack model. The ability to maintain a low level of variation throughout training and testing is a highly desired characteristic, and it is a very significant characteristic when it comes to training self-driving cars. This significant difference could be since we have doubled the number of parameters in our fully connected layers, which is due to the fact that the size of the sharing feature vector is doubled after conjunction, which may account for the difference. Regardless, we believe that this model is the most exciting model to further examine and research.

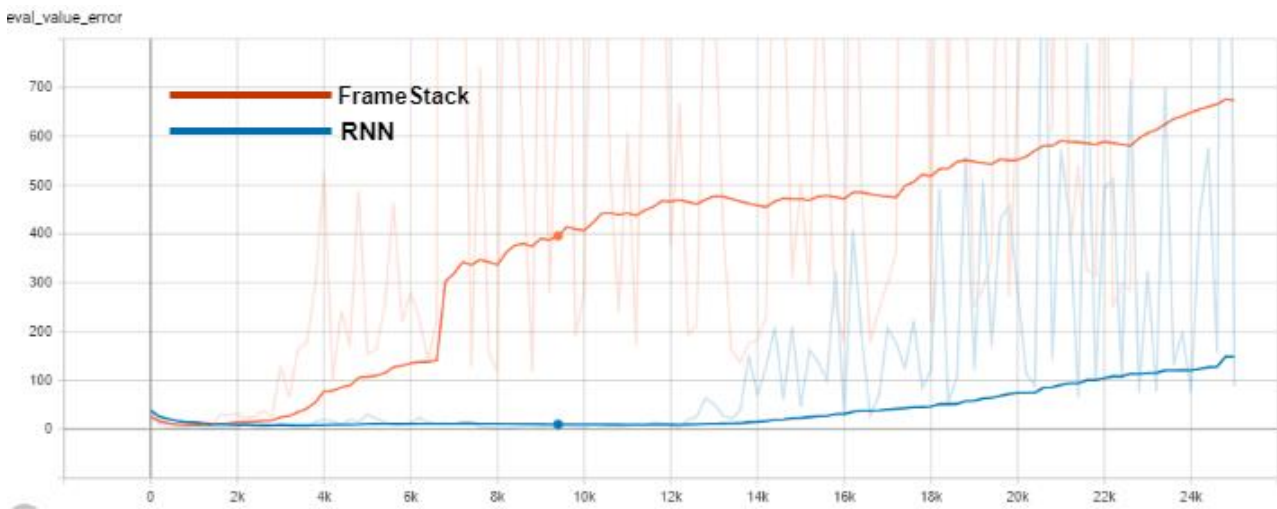


Figure 6-7: Evaluation of error value for the models

Figure 6-7 represent the estimation of the error value for the two models over the number of epochs. The frame stack model performs much faster than the RNN in evaluating the error value. As illustrated in the graph, the frame stack model begins performing immediately upon training, whereas the RNN takes much time to begin working efficiently.

Figure 6-9 illustrates the average acceleration action for the stack model throughout the number of epochs. When the average acceleration action increases, taking the acceleration action by the agent will increase. While in Figure 6-8 that shows the decreasing of the mean break action where taking break actions decrease by the agent. As a results of these two figures, we can see how our agent improves by increasing its speed and performing less breaks.

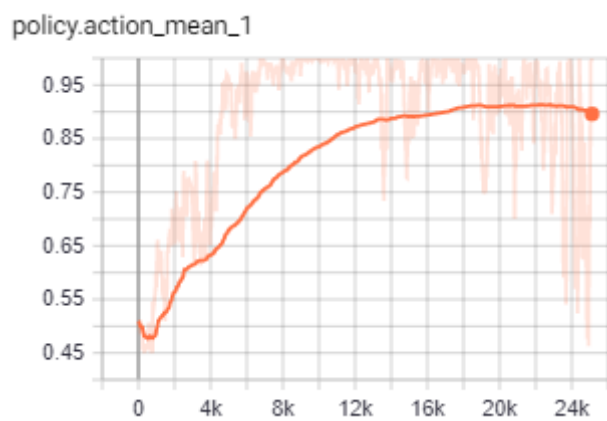


Figure 6-9: Acceleration mean action-FrameStack model

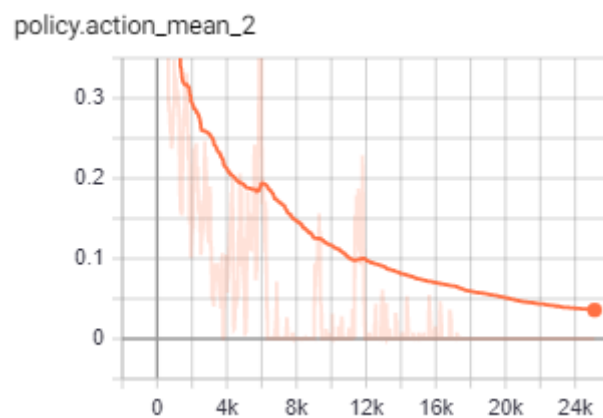


Figure 6-8: Break action mean-FrameStack model

7 Conclusion And Future Work

In this study, we explore the present state-of-the-art in reinforcement learning, as well as how reinforcement learning is currently being used to autonomous driving applications. Whereas the improvement of DQN and PPO algorithms has increased the viability of reinforcement learning in the context of autonomous driving.

7.1 Contributions

The contribution in this paper comes in the form of experimenting and testing different neural network architectures for PPO and DQN-driven agent. The first contribution comes with the study of implementing DQN on the CARLA simulator. In this study, we showed that deep neural network such as the Xception model failed in training the agent, resulting in exploding Q-values and loss. it's complicated to choose a model of 71 hidden layer with reinforcement learning to train an agent to drive, because deep models caused more overfitting. To do this, we aimed to convert to a 64x3 Convolutional neural network model, this model provides much better performance. Finally, additional time was required for the training process. The agent was trained for 8 days in the absence of dynamic weather and traffic, because when traffic was considered, the episodes failed immediately, and when dynamic weather was considered, the kernel failed most of the time due to the complexity of the RGB sensor images.

Another contribution comes in the form of a brief comparative study of the effects of employing several types of PPO model designs in the CarRacing-v0 environment. In my research, I discovered that the frame stack model gets the greatest cumulative rewards the fastest. The recurrent model, which earned equal scores to the frame stack model while having far less variation than any of the other models, might be the most exciting to investigate further. We can conclude that using reinforcement learning is a useful method for training an autonomous driving. Some of RL algorithms are applicable to handle discrete action, including DQN, while others are relevant to continuous actions, including PPO. As a result of this paper, we showed that PPO perform better than DQN. Where PPO can offer higher convergence as well it learns the policy directly, different than DQN that first learns the Q-values to used them to define the policy.

In addition, we discovered the data collection and training the model for the variational encoder. It will be implemented in future work in the training pipeline of PPO to help the deep reinforcement learning agents learn faster.

7.2 Future work

- Train the agent with several models and parameters for further episodes, and analyze the results of those models with each other, as well as at different episodic check points.
- Apply different image processing methods to enhance the image quality, to help the agent to be able to make better decisions.
- Develop a procedural generation algorithm for cities in CARLA, as well as methods for reconstruction of highways from real-world map data. This data might potentially be utilized to create a light-weight simulator with tricky road scenarios that could be used for RL.
- Design and evaluate different reward formulations in the CARLA simulator in order to achieve the desired driving behaviors from reinforcement learning agents.
- Implement PPO algorithm on the CARLA simulator with the help of the variational encoder in the training pipeline to reduce the training time and making the state space more predictable for the trained agent.
- Training using other deep reinforcement learning algorithms such as DDPG and A3C in order to get better results.
- Used temporal models such as LSTM models, and multi-agent training, with improving the exploration.

References

- [1] “ImageNet: A large-scale hierarchical image database | IEEE Conference Publication | IEEE Xplore.” <https://ieeexplore.ieee.org/document/5206848> (accessed Nov. 01, 2021).
- [2] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, “CARLA: An Open Urban Driving Simulator,” Nov. 2017, Accessed: Nov. 06, 2021. [Online]. Available: <https://arxiv.org/abs/1711.03938v1>
- [3] *, Hans Andersen 1, Xinxin Du 2, Xiaotong Shen 2, Malika Meghjani 2, You Hong Eng 2, Daniela Rus 3 and Marcelo H. Ang Scott Drew Pendleton 1, “Perception, Planning, Control, and Coordination for Autonomous Vehicles,” 2017. Accessed: Oct. 30, 2021. [Online]. Available: [file:///C:/Users/windows10/Downloads/machines-05-00006-v3 \(1\).pdf](file:///C:/Users/windows10/Downloads/machines-05-00006-v3%20(1).pdf)
- [4] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, “CARLA: An Open Urban Driving Simulator.” PMLR, pp. 1–16, Oct. 18, 2017. Accessed: Nov. 29, 2021. [Online]. Available: <https://proceedings.mlr.press/v78/dosovitskiy17a.html>
- [5] A. Yu, R. Palefsky-Smith, and R. Bedi, “Deep Reinforcement Learning for Simulated Autonomous Vehicle Control,” 2016.
- [6] S. Shalev-Shwartz, S. Shammah, and A. Shashua, “Safe, Multi-Agent, Reinforcement Learning for Autonomous Driving,” *arXiv preprint arXiv:1610.03295v1*, p. 13, Oct. 2016, Accessed: Nov. 29, 2021. [Online]. Available: <https://arxiv.org/abs/1610.03295v1>
- [7] L. G. Cuenca, E. Puertas, J. F. Andrés, and N. Aliane, “Autonomous Driving in Roundabout Maneuvers Using Reinforcement Learning with Q-Learning,” *Electronics 2019, Vol. 8, Page 1536*, vol. 8, no. 12, p. 1536, Dec. 2019, doi: 10.3390/ELECTRONICS8121536.
- [8] A. Ganesh, J. Charalel, M. das Sarma, and N. Xu, “Deep Reinforcement Learning for Simulated Autonomous Driving”, Accessed: Nov. 30, 2021. [Online]. Available: <https://www.dropbox.com/sh/>
- [9] B. Huval *et al.*, “An Empirical Evaluation of Deep Learning on Highway Driving,” Apr. 2015, Accessed: Nov. 30, 2021. [Online]. Available: <https://arxiv.org/abs/1504.01716v3>
- [10] A. el Sallab, M. Abdou, E. Perot, and S. Yogamani, “End-to-End Deep Reinforcement Learning for Lane Keeping Assist,” Dec. 2016, Accessed: Nov. 30, 2021. [Online]. Available: <https://arxiv.org/abs/1612.04340v1>

- [11] M. Bojarski *et al.*, “End to End Learning for Self-Driving Cars,” Apr. 2016, Accessed: Nov. 30, 2021. [Online]. Available: <http://arxiv.org/abs/1604.07316>
- [12] A. Kendall *et al.*, “Learning to Drive in a Day,” *Proceedings - IEEE International Conference on Robotics and Automation*, vol. 2019-May, pp. 8248–8254, Jul. 2018, doi: 10.1109/ICRA.2019.8793742.
- [13] F. Codevilla, M. Miiller, A. Lopez, V. Koltun, and A. Dosovitskiy, “End-to-end Driving via Conditional Imitation Learning,” *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 4693–4700, Oct. 2017, doi: 10.1109/ICRA.2018.8460487.
- [14] M. Vitelli and A. Nayebi, “CARMA: A Deep Reinforcement Learning Approach to Autonomous Driving,” 2016.
- [15] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. K. Openai, “Proximal Policy Optimization Algorithms”.
- [16] “The actor-critic proximal policy optimization (Actor-Critic PPO)... | Download Scientific Diagram.” https://www.researchgate.net/figure/The-actor-critic-proximal-policy-optimization-Actor-Critic-PPO-algorithm-process_fig3_339651408 (accessed Nov. 15, 2021).
- [17] J. Deng, W. Dong, R. Socher, L.-J. Li, Kai Li, and Li Fei-Fei, “ImageNet: A large-scale hierarchical image database,” pp. 248–255, Mar. 2010, doi: 10.1109/CVPR.2009.5206848.
- [18] Y. Lecun, L. Eon Bottou, Y. Bengio, and P. H. Abstract|, “Gradient-Based Learning Applied to Document Recognition”.
- [19] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, “Vision meets Robotics: The KITTI Dataset”, Accessed: Nov. 01, 2021. [Online]. Available: <http://www.cvlibs.net/datasets/kitti>.
- [20] X. Huang *et al.*, “The ApolloScape Dataset for Autonomous Driving.” pp. 954–960, 2018. Accessed: Nov. 01, 2021. [Online]. Available: <http://apolloscape.auto>.
- [21] R. J. Williams, “Simple statistical gradient-following algorithms for connectionist reinforcement learning,” *Machine Learning 1992 8:3*, vol. 8, no. 3, pp. 229–256, May 1992, doi: 10.1007/BF00992696.
- [22] C. J. C. H. Watkins and P. Dayan, “Q-learning,” *Machine Learning 1992 8:3*, vol. 8, no. 3, pp. 279–292, May 1992, doi: 10.1007/BF00992698.

- [23] V. Mnih *et al.*, “Playing Atari with Deep Reinforcement Learning,” Dec. 2013, Accessed: Nov. 09, 2021. [Online]. Available: <https://arxiv.org/abs/1312.5602v1>
- [24] G. Liu, C. Li, T. Gao, Y. Li, and X. He, “Double BP Q-Learning Algorithm for Local Path Planning of Mobile Robot,” *Journal of Computer and Communications*, vol. 09, no. 06, pp. 138–157, 2021, doi: 10.4236/JCC.2021.96008.
- [25] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, “Dueling Network Architectures for Deep Reinforcement Learning,” *33rd International Conference on Machine Learning, ICML 2016*, vol. 4, pp. 2939–2947, Nov. 2015, Accessed: Nov. 10, 2021. [Online]. Available: <https://arxiv.org/abs/1511.06581v3>
- [26] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, “Prioritized Experience Replay,” *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, Nov. 2015, Accessed: Nov. 10, 2021. [Online]. Available: <https://arxiv.org/abs/1511.05952v4>
- [27] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. K. Openai, “Proximal Policy Optimization Algorithms,” Jul. 2017, Accessed: Nov. 12, 2021. [Online]. Available: <https://arxiv.org/abs/1707.06347v2>
- [28] J. Schulman, S. Levine, P. Moritz, M. Jordan, and P. Abbeel, “Trust Region Policy Optimization,” *32nd International Conference on Machine Learning, ICML 2015*, vol. 3, pp. 1889–1897, Feb. 2015, Accessed: Nov. 12, 2021. [Online]. Available: <https://arxiv.org/abs/1502.05477v5>
- [29] J. Schulman, P. Moritz, S. Levine, M. I. Jordan, and P. Abbeel, “High-Dimensional Continuous Control Using Generalized Advantage Estimation,” *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, Jun. 2015, Accessed: Nov. 16, 2021. [Online]. Available: <https://arxiv.org/abs/1506.02438v6>
- [30] “Reinforcement Car Racing With A3C | PDF | Artificial Neural Network | Deep Learning.” <https://www.scribd.com/document/358019044/Reinforcement-Car-Racing-with-A3C> (accessed Nov. 24, 2021).
- [31] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2016-December, pp. 770–778, Dec. 2015, doi: 10.1109/CVPR.2016.90.

- [32] D. Ha and J. Schmidhuber, “World Models,” *Forecasting in Business and Economics*, pp. 201–209, Mar. 2018, doi: 10.5281/zenodo.1207631.