UPPSALA
UNIVERSITET

UPTEC F 23006
Examensarbete 30 hp
November 2023

Evaluation of Deep Q-Learning Applied to City Environment Autonomous Driving

Jonas Wedén

Civilingenjörsprogrammet i teknisk fysik

UPPSALA
UNIVERSITET

Evaluation of Deep Q-Learning Applied to City Environment Autonomous Driving

Jonas Wedén

Abstract

This project's goal was to assess both the challenges of implementing the Deep Q-Learning algorithm to create an autonomous car in the CARLA simulator, and the driving performance of the resulting model. An agent was trained to follow waypoints based on two main approaches. First, a camera-based approach, which allowed the agent to gather information about the environment from a camera sensor. The image along with other driving features were fed to a convolutional neural network. Second, an approach focused purely on following the waypoints without the camera sensor. The camera sensor was substituted for an array containing the agent's angle with respect to the upcoming waypoints along with other driving features. Even though the camera-based approach was the best during evaluation, no approach was successful in consistently following the waypoints of a straight route. To increase the performance of the camera-based approach more training episodes need to be provided. Furthermore, both approaches would greatly benefit from experimentation and optimization of the model's neural network configuration and its hyperparameters.

Teknisk-naturvetenskapliga fakulteten

Uppsala universitet, Utgivningsort Uppsala/Visby

Handledare: Magnus Lundstedt Ämnesgranskare: Roland Hostettler

Examinator: Tomas Nyberg

Populärvetenskaplig Sammanfattning

I detta projekt har en metod från förstärkningsinlärning vid namn Deep Q-Learning (DQN) implementerats med hjälp av Python och PyTorch. DQN är förstärkningsmetoden Q-Learning parad med neuronnät. Syftet med denna metod var att lära en modell att köra en bil i en stadsmiljö genom att följa vägpunkter. Stadsmiljön samt bil- och körfysiken har tillhandahållits av simulatorn CARLA.

Projektet har fokuserat på implementation och optimering av förstärkningsinlärningsmetoden DQN, för att skapa en självkörande bil i CARLA. Målet var att bedöma både utmaningarna med att använda DQN för att träna modellen, samt framförandet av den självkörande bilen från den tränade modellen.

Under projektets gång uppstod två huvudsakliga tillvägagångssätt. Det första innebar att agenten som styr bilen huvudsakligen fick information om världen från en kamerasensor placerad på bilen. Agenten fick också information om hur långt bort nästa vägpunkt är, agentens vinkel gentemot vägbanans riktning, samt agentens fart. Bilden från kamerasensorn matades till ett faltningsnätverk där utdatan sammanfogades med resten av informationen innan allt matades till det första lagret i neuronnätet. Därefter approximerade neuronnätet den bästa handlingen för stunden.

Det andra tillvägagångssättet var riktat till att göra det lättare för agenten att enbart lära sig följa vägpunkterna, genom att avlägsna kamerasensorn. Istället använde sig agenten av information tillhandahållen enbart från en lista. Informationen bestod av vinkeln mellan agentens bil och de nästa 15 vägpunkterna, agentens avstånd till mitten av vägbanan, agentens vinkel gentemot vägbanans riktning, och agentens fart. Denna informationslista matades till neuronnätet som approximerade den bästa handlingen.

De två tillvägagångssätten, där det sista sättet utfördes två gånger; en gång utan byggnader i simulationen och en gång med, tog ungefär 48, 40 samt 40 timmar att träna 16000, 40000 och 40000 episoder. Även om tillvägagångssättet med kamerasensorn gav det bästa resultatet under utvärderingen, lyckades ingen av dem kontinuerligt följa vägpunkterna på en rak väg. Metoden med kamerasensorn hade med fördel behövt fler träningsepisoder för att förbättra sin prestation, vilket tiden inte räckte till för. Utöver detta, skulle experimentering med neuronnätets uppställning, samt dess hyperparametrar för att optimera modellen ha störst betydelse för tillvägagångssättens inlärning.

Acknowledgements

I would like to express my gratitude to my supervisor Magnus Lundstedt for his helpful advice during the project. I would also like to express my gratitude to my subject reviewer Roland Hostettler for providing invaluable feedback on this report.

Contents

1 Introduction

1.1 Background

Machine learning is an increasing part of our day-to-day lives. It can help us solve many problems with greater ease and speed. Everything from identifying melanoma [1] to companies tailoring their commercials to a certain demographic [2]. Another particular issue that can be improved by machine learning is the one of road safety.

According to WHO global status report on road safety 2018 [3], deaths on the roads have increased every year since 2000 and is the leading cause of death for people between the ages 5-29. Among the key risk factors for accidents are speed and driving under the influence. Additionally, humans have a tendency to lose attention, be uncertain of traffic rules, or have slow reaction times. These risk factors are all a consequence of our inherent flaw of human error. This flaw cannot be removed, but it can be mitigated in certain situations.

Autonomous driving is one such way to mitigate human error when driving. In particular, autonomous driving refers to the concept of a vehicle driving using machine learning, without a human controlling it. The vehicle uses sensors such as cameras, LiDAR, radar, etc., to gather data from the surrounding environment [4]. Many accidents could be avoided if a self-driving vehicle, trained to obey all traffic rules, without being distracted, and simultaneously taking swift actions with quicker reaction times than humans, were to be put into action in the real world. Driving under the influence would no longer be an issue, and driving while fatigued never a risk.

Several car manufacturers have already incorporated parts of autonomous driving into their cars, e.g. cruise control or lane assist, and some have begun the journey towards complete autonomous driving [5]. However, this is a difficult task to accomplish, but by using machine learning methods such as reinforcement learning, it can become a reality.

Reinforcement learning is an area of machine learning that has grown popular within the category of games. It consists of an agent exploring an environment to find a set of actions that would maximize a cumulative reward. It is a relatively recent introduction within machine learning, having been introduced in the 1980s [6]. Only recently, in 2016, a reinforcement learning agent by the name AlphaGo [7] beat a master at the complex game of Go.

Reinforcement learning mainly constitutes a trial and error process. Q-Learning is one method within reinforcement learning and the focus of this project. This

method calculates and saves the reward for a specific action taken in a specific state in a table. Essentially, this is how an agent learns which actions to take. However, what happens when there are many states and many actions? In that case, the action-reward table would become too computationally expensive to use. This is what deep reinforcement learning (DRL) aims to solve by using a deep neural network.

In autonomous driving, the states are normally represented by high-dimensional data from cameras and other sensors. Instead of calculating the action-reward for every state and saving it to a table, DRL uses a deep neural network to approximate the reward of each action from the high-dimensional data, making it possible to solve complex problems that require a large amount of data [8], [9]. This method is known as Deep Q-learning (DQN) [10].

Training a vehicle safely requires a realistic driving simulator such as CARLA [11]. This is an open-source simulator for autonomous driving research built upon Unreal Engine. CARLA provides many features, e.g. pre-built city maps, Python API, autonomous driving sensor suite, actors, traffic manager, among many others outside the scope of this project. These features allow the focus of this project to be on the implementation of one autonomous vehicle in a life-like city environment, given that CARLA can provide the city map, other vehicles, pedestrians, and the control of these within the program.

1.2 Project Goals

This project focuses on the implementation and optimization of the reinforcement learning method deep Q-learning, to construct an autonomous vehicle in the open-source autonomous driving simulator, CARLA. The goal is to assess both the challenges of using DQN for an autonomous vehicle in a city environment, and the driving performance of this algorithm when the environment moves towards more realistic scenarios. Mainly, an environment consisting of other vehicles, traffic lights, and pedestrians. This goal is achieved by answering the following questions:

- What is the rate at which the DQN algorithm converges during training?
- How stable is the DQN algorithm during training episodes?
- How well will the trained autonomous vehicle follow the designated route?
- How consistent will the trained autonomous vehicle be when following the same route?
- Can the autonomous vehicle avoid collisions and follow traffic rules?

1.3 Related Work

Recent work in the field of autonomous driving through neural networks have showcased significant progress. This section provides an overview of some noteworthy developments in this domain. For more information on the topic, the survey in [12] is a great starting point.

In the pursuit of more streamlined autonomous systems, researchers have explored end to end approaches using Convolutional Neural Networks (CNNs). This approach, presented in [13], involves training a CNN to map raw pixels from a front-facing camera directly to steering commands. This system has shown that CNNs are able to learn the task of lane following in diverse conditions, which signifies remarkable progress in the world of self-driving vehicles.

However, in [13], the network was given control over lane and road following only, when the agent approached an intersection or a lane change was required, a human driver had to take over. A solution to this is presented in [14], where an agent is trained via imitation learning and at test time is driving based on visual input as well as by responding to navigational commands. In theory, this solution would allow a passenger to provide input on which lane the agent should drive in or which turn to take.

Online training of a reinforcement learning agent, where the agent gathers data from the environment first-hand, is time-consuming. By training multiple agents in parallel, the training time can be decreased. However, for CARLA, this is especially resource intensive. In [15], two distributed asynchronous multi-agent reinforcement learning algorithms are presented to address this issue. It is demonstrated that the methods proposed can accelerate the online training of an agent on the CARLA simulator. This method could further increase the speed at which self-driving vehicles are developed.

Furthermore, recent work in autonomous driving using DRL and CARLA consist of the implementation and comparison between the algorithms DQN and Deep Deterministic Policy Gradient (DDPG) in [16]. This work concluded that it was successful in implementing both DQN and DDPG to control the navigation of a vehicle. They also found that DDPG obtained better performance and driving more similar to a human.

The aim of this project is to implement the DQN algorithm in a life-like scenario provided by the CARLA simulator. As opposed to the recent work in [16], the focus of this project is purely on the implementation, optimization, and evaluation of DQN when put into a realistic scenario incorporating other vehicles and pedestrians.

2 Theory

In this section, theory about the topics used in this project is provided. It begins with neural networks and the basics of reinforcement learning, and ends with the concept of Q-learning and DQN.

2.1 Neural Networks

Neural networks are the foundation of DQN. They are inspired by the human brain's network of neurons. The structure of a neural network is typically built of layers: 1 input layer, at least 1 hidden layer and 1 output layer, see Figure 1. The layers contain nodes (neurons) that are connected to each node in the next layer.

The input layer is a vector $\mathbf{x}$ with the dimensions $(n, 1)$, where n is the number of input features. The hidden layers apply weights to the information passed through the layers. Consider 1 hidden layer, for the i-th node in the hidden layer, we compute a weighted sum of the inputs and apply a non-linear activation function, such as the rectified linear unit function ($ReLU(y) = \max(0, y)$) to introduce non-linearity,

$$ReLU\left(\sum_{j=1}^{n}(x_j w_{ij}) + b_i\right),$$

where x_j represents the i-th input, w_{ij} corresponds to the weight applied between the j-th input and i-th hidden layer node, b_i is the bias term for the i-th node, which allows the network to shift the activation function. The output layer is the final layer that adjusts the output from the hidden layers to the final output, which for reinforcement learning is the approximated Q-value for each action. Similar to the hidden layer, each node in the output layer calculates a weighed sum of the hidden layer output and applies an activation function.

The neural network learns by minimizing a loss function by updating the network's weights and biases, typically using gradient descent. This process of learning is called backpropagation [17].

When using images to feed a neural network, the most common approach is to use a convolutional neural network (CNN). The point is for the neural network to extract features of the input image. There are three main layers in a CNN: convolutional layer, pooling layer, and fully-connected layer [19], see Figure 2.

The convolutional layer is the first layer the image is fed to. This layer produces a convolution of the image, which entails using a 2D filter that moves across the entire image and calculates a dot product between the weights of the filter and the pixel values spanning the size of the filter. Each filter operation is collected into

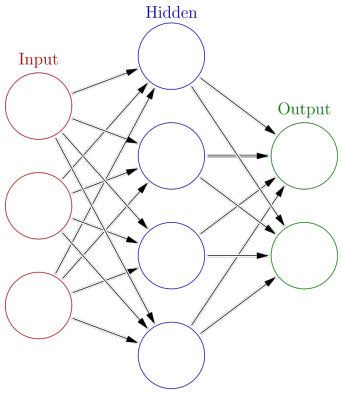


Figure 1: Simple example of a neural network [18].

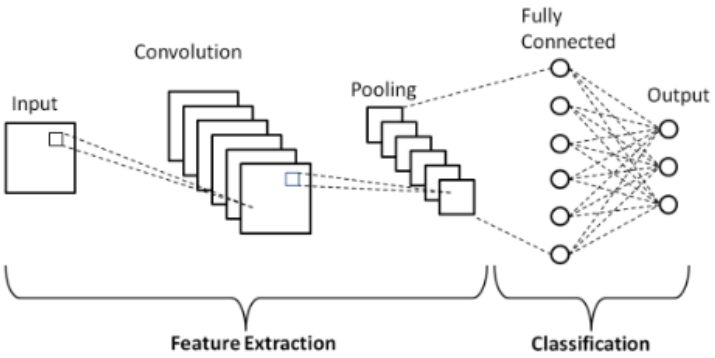


Figure 2: Diagram of basic CNN architecture [20].

an output array and is known as a feature map. To establish nonlinearity in the model, a ReLU function is applied to the complete output array.

The pooling layer reduces dimensionality of the array to help reduce the complexity and improve the efficiency of the network. Like the convolution layer, the pooling layer places a filter on top of the array, but instead of calculating the dot product between the weights of the filter and the array, it applies either max pooling or average pooling. Max pooling involves taking the maximum of the pixel values within the filter to send to the output array, and average pooling taking the average of the pixel values.

The final layer is the fully-connected layer. This layer is in charge of classification based on the features extracted from the previous layers. For this project, this is the layer that finally produces the approximated Q-values for each action available.

2.2 Reinforcement Learning

Reinforcement learning is a collective term for a multitude of algorithms that aim to instruct an agent to take actions in order to maximize a cumulative reward. The focus of this project is the algorithm known as Q-learning, and, when paired with neural networks, Deep Q-Learning.

The agent in reinforcement learning is an entity that takes actions. We can think of this agent as a person. The environment could be a city this person has never been to. The problem could be that the person is set free in the middle of the city with the goal to find the closest restaurant. What the person observes is what is called the state, which, of course, is derived from the city environment. We allow the agent to explore the city and send them rewards when they walk in the right direction. Hopefully, the person will figure out, based on the rewards, which actions lead to the correct direction and finally reach the restaurant. This is the essence of reinforcement learning.

Figure 3 shows a diagram of the interaction between the agent and the environment, as described in the previous paragraph. The agent takes an action, which affects the agent's relationship to the environment. Based on the effect of the agent's action, the agent receives a reward and the next state. The process is then repeated.

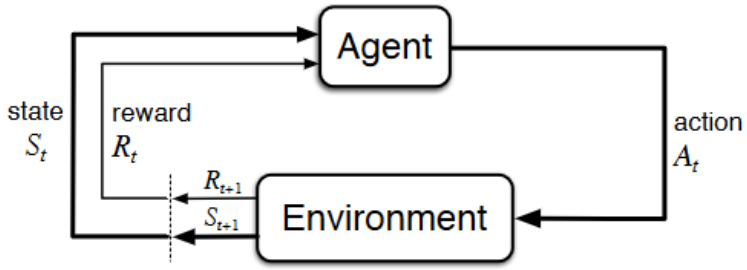


Figure 3: The interaction between the agent and the environment [6].

2.2.1 Markov Decision Process

The decision process of a reinforcement learning agent can be described by the Markov Decision Process (MDP), see [6, Chapter 3]. The MDP is a mathematical framework essential in reinforcement learning to model and solve decision-making problems.

The MDP is defined as a 4-tuple $(\boldsymbol{S}, \boldsymbol{A}, p, \boldsymbol{R})$, where:

- $\boldsymbol{S}$ is the state space, containing a set of states,

- $\boldsymbol{A}$ is the action space, containing a set of actions,
- $p(s', r \mid s, a)$ is the probability that action a in state s will lead to the next state s' and the reward r,
- $\boldsymbol{R}$ is the reward received when moving from state s to the next state s'.

The probability distribution denoted as

$$p(s', r|s, a) = Pr\{S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a\}, \tag{2.1}$$

plays a vital role in reinforcement learning and MDPs. In this equation, the variables are defined as follows: $S_{t+1} = s'$ represents the next state, $R_{t+1} = r$ the next reward, $S_t = s$ signifies the current state, and $A_t = a$ corresponds to the action chosen in the current state, see Figure 3.

The probability, states, and actions, are what defines the dynamics of an MDP. The goal is often to maximize the received rewards. Therefore, given the quantities in (2.1), the agent's decision-making process can be optimized to identify the actions that yield the greatest cumulative rewards.

2.2.2 Q-Learning

The Q-learning algorithm paired with a neural network is known as Deep Q-learning, which is the focus of this project. Before describing the theory behind DQN, the basis of the regular Q-learning algorithm will be explained in this section.

The Q-Learning algorithm is a fundamental technique used to train an agent to make decisions within an environment. The agent explores the environment with the goal of learning an optimal policy, often denoted π. The policy π is found by estimating the value of each action for every state, known as the Q-value. The expected future reward of taking a specific action in a given state is what the Q-value represents. The policy π specifies which action the agent will take. Throughout the learning process, the agent continually updates the Q-values [6].

The process is as follows:

1. **Initialization of Q-Table**: Initialize the Q-table for each state-action pair, arbitrarily. This process is executed once at the start of the learning phase.
2. **Action Selection**: At each step, the agent selects an action, denoted as a, based on its current state s following a specified policy π.

3. **Action Execution and Observation**: The agent executes action a, interacts with the environment, and observes both the immediate reward r, and the subsequent state s', as a consequence of taking action a.

4. **Q-Value Update**: The Q-value for the current state-action pair (s, a) is updated according to the Q-learning equation, see [6, Chapter 6.5]:

$$Q(s,a) = Q(s,a) + \alpha[r + \gamma \max_a(Q(s',a)) - Q(s,a)]. \tag{2.2}$$

   Here α represents the learning rate, γ the discount factor applied to future rewards, and $max_a(Q(s',a))$ signifies the maximum Q-value achievable in the net state s' by considering all possible actions.

5. **State Transition**: The agent updates its current state s to be the next state s' observed after taking action a.

With the completion of this process, the agent has a fully updated Q-table. With this Q-table, the agent can determine the next action for each state by selecting the action associated with the highest Q-value.

2.2.3 Deep Q-Learning

Regular Q-Learning is most useful for simpler tasks, with an environment that possess a limited amount of actions and states. When the number of actions and states increase, the Q-table will become too computationally expensive. Therefore, the incorporation of a deep neural network to Q-learning has been contrived, with the idea that the neural network will approximate the Q-values for each action.

The input to the neural network will be the state s, and in turn the network will approximate the optimal Q-value function

$$Q^*(s,a) = \max_\pi \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots \mid s_t = s, a_t = a, \pi], \tag{2.3}$$

which is the maximum sum of rewards r_t discounted by γ at each time step t, achieved by a policy π after an observation s and taking an action a [10]. The agent will choose the action with the highest Q-value, as in regular Q-learning, but through gradient descent the weights in the neural network will be updated instead of a Q-table [21].

When a neural network is used to approximate the Q-values, challenges related to stability and convergence arises. There are several causes to the instability. First, often in reinforcement learning, an agent experiences correlated observations over time. Learning from correlated data can introduce bias, which can lead to instability. Second, when the Q-function is updated within the neural network,

small changes can change the policy π. This can affect the data distribution of the observed states and actions, which can make learning unstable. Third, the correlations between the Q-values and the target values $r + \gamma \max_a Q(s', a)$ can affect the learning process and lead to instability.

The instabilities are mitigated by two techniques, experience replay, and target network. Experience replay stores the agent's experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ at each time-step t in a data set $D_t = \{e_1, \ldots, e_t\}$. Q-learning updates are applied on randomly drawn mini-batches of experience from the stored samples to break the correlation between consecutive experiences. A target network is a separate neural network copy of the primary network used to stabilize the training. This network computes the target values periodically and is updated less frequently than the primary network, reducing correlations with the target.

During training, the loss is calculated from the Mean Squared Error (MSE) between the target value $r + \gamma \max_a Q(s', a)$, and the approximated value $Q(s, a)$. Minimizing this loss leads to the optimal Q-values.

The process of the DQN-algorithm during training starts by initializing the main Q-value network, the target Q-value network, and the experience replay storage. The agent is trained in an episodic fashion, where for every time-step of an episode, the agent either chooses the most optimal action or a random action with probability ϵ. At every time-step of an episode, the agent interacts with the environment, takes an action, observes the next state and reward, and stores these into the experience storage. During training, the agent samples a random mini-batch from the storage to update the network. The target network is updated less frequently than the primary network, and an MSE loss is calculated between the two. Afterward, gradient descent is used to update the network's weights.

Furthermore, a technique called ϵ-greedy with ϵ-decay can help the agent learn faster by forcing the agent to explore new actions. At the beginning of the training, ϵ is set to a high value, e.g. $\epsilon = 0.7$, to make it more probable that the agent chooses a random action. However, for every time-step, ϵ is decreased a small amount until it reaches a minimum ϵ. This is called ϵ-decay. This technique ensures the agent maximizes exploration at the beginning of training while slowly converging to a more greedy approach, where the agent only chooses the most optimal actions.

The DQN-algorithm's behavior and performance is influenced by a set of parameters known as hyperparameters, which are set before the training commences. Some of these hyperparameters are [22]:

- Learning rate α: This determines the step-size taken during the gradient descent optimization process. A higher value could result in faster convergence,

but could also overshoot the optimal value.

- Discount factor γ: To determine the weight given to future rewards, the discount factor is used. A value closer to 0 results in more weight to immediate rewards, while a value closer to 1 gives more weight to future rewards.
- Batch size: Number of experiences pulled from the experience storage and used during each training update. Larger batch sizes can help the agent learn more quickly, but requires more computation.

The hyperparameters require tuning to find the values that give the best results.

2.3 Using Deep Q-Learning to Learn Driving

To teach an agent to drive using the DQN-algorithm, the problem first needs to be structured as an MDP to model the agent's decision-making and define the states, actions, and rewards. In DQN, the primary objective is to learn the Q-value function, see (2.2), and then find an optimal policy that maximizes the expected future rewards, see (2.3).

For the agent to be able to learn well, the action space and reward function needs to be well-defined. In our context, actions should consist of various turn radii and acceleration values, to allow the agent to navigate the environment. Equally, the design of the reward function is vital to serve as a guide towards desirable driving behavior.

Neural networks play an important role within DQN to approximate the Q-value function, where a state defined in the MDP serves as the input to the neural network. When driving, the agent can observe the state from onboard sensors, often a camera, resulting in an image for each time-step, where its pixel values form the input to the neural network. The network's output consists of the approximated Q-values for every possible action.

During training, the weights of the neural network are updated using gradient descent driven by the MSE loss between the predicted Q-values from the primary network and the Q-values from the target network. To stabilize training and break correlation in observations, the agent's experiences (state, action, reward, next state) at every time-step are stored in a replay buffer, and is randomly sampled in mini-batches during the training.

3 Method

In this section, the method of using a DQN together with CARLA is explained.

The basic overview for the method of teaching an agent to drive a car was to use CARLA to handle the simulation of the city, its vehicles, pedestrians, and the agent vehicle. CARLA provided the sensors to receive data of the current state, in this case an RGB image, see Figure 7. The sensors also detected whether the vehicle has collided or violated traffic rules by crossing a solid lane marker. Furthermore, CARLA also provided the means to control the agent vehicle.

To teach the agent, the DQN-algorithm described in section 2.2.3 was implemented together with a CNN. The CNN received the RGB image from the camera sensor as input and output approximated Q-values for the actions, see Figure 4. The agent chose the best action approximated by the CNN at every step in an ϵ-greedy policy, and learned from experience using the experience replay introduced in the DQN-algorithm.

Furthermore, a simpler approach, comparable to the similar approach in [16] was also explored. This approach did not use an image sensor to observe the environment, hence it did not need a CNN, only a regular neural network. For this, the input state consisted of an array of various vehicle- and waypoint-information, see Figure 5. The output and the rest of the algorithm worked in similar fashion to the more advanced camera-based approach.

To maximize the computational speed, the code used the PyTorch package Torch CUDA (v. 1.13.1+cu117) to utilize a dedicated GPU for computations. The training and validation was running on the following system specifications:

- GPU: GTX 1070.
- CPU: i7-7700k @ 4.20 GHz.
- RAM: 16 GB DDR4.
- OS: Windows 10 v. 19045

3.1 CARLA

This section is dedicated to presenting how CARLA (v.0.9.13) was used in this project.

The map used in CARLA is called "Town10", and is a city containing junctions with traffic lights, crosswalks, stop signs, and different lanes, among others, see Figure 6. Namely, an urban environment filled with typical traffic rules belonging to a city. In this project, only this map has been considered, primarily because it possesses every main property of city-driving.

To place a vehicle in the city, the type of vehicle was chosen from the CARLA

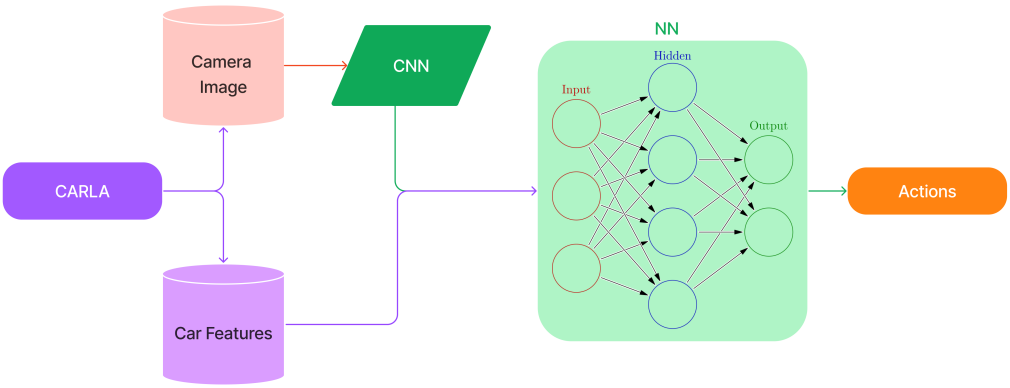


Figure 4: The structure of choosing an action for the approach with camera.

blueprints and was spawned on a chosen coordinate. The destination was chosen as coordinates to another part of the city.

Using the built-in route planner, waypoints from the location of the vehicle spawn to the destination were generated with a chosen spacing in meters. The waypoints took the appearance of green rectangles, which distinguished them from the rest of the environment.

CARLA provides various sensors that can attach to the car. Some examples are: RGB camera, collision sensor, depth camera, radar, LiDAR, and lane invasion. In this project, only the RGB camera, collision sensor, and lane invasion sensor were used. The RGB camera provided data of the current state to the CNN, the collision as well as the lane invasion sensor recorded any collision or violation of lane marking which ended the episode.

The vehicle was controlled using built-in methods. These methods were applied to the vehicle as the actions that were taken. Controlling a vehicle in reality is more advanced than discrete actions, but because DQN works best with discrete actions and to avoid having many actions, they have been limited to four types in the camera-based approach.

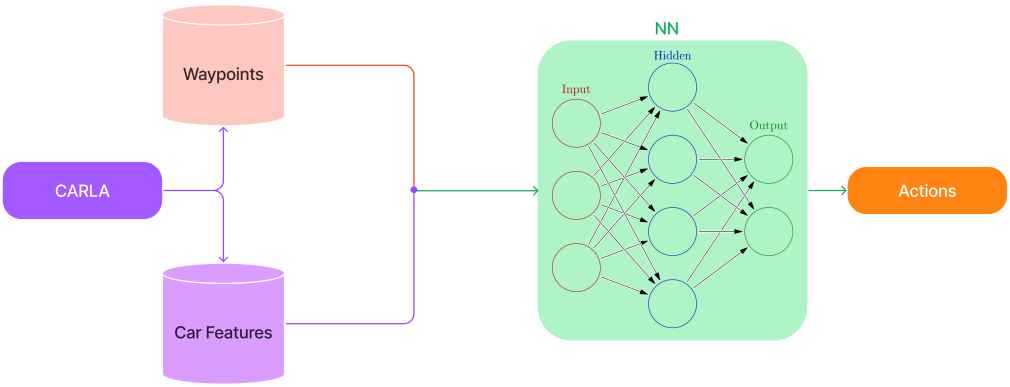


Figure 5: The structure of choosing an action for the approach without camera.

Figure 6: Bird's-eye view of CARLA's Town10.

3.2 Deep Q-Learning

Because the agent observed the environment using a camera sensor provided by CARLA in the camera-based approach, see Figure 7, the DQN algorithm was implemented with a CNN to be able to extract features from the camera image. The CNN was implemented using the PyTorch library (v. 1.13.1+cu117) [23]. In the camera-free approach, the convolutional part of the CNN which deals with images was not needed.

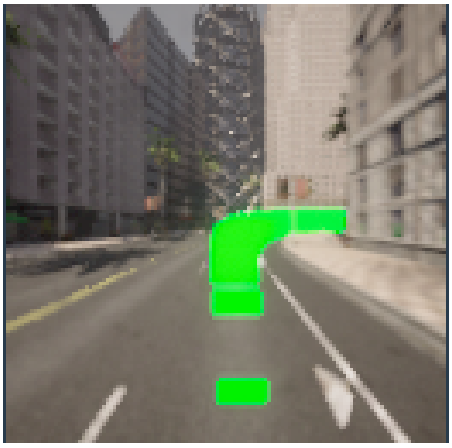


Figure 7: The vehicle's perspective from the camera sensor in 128x128 RGB.

3.2.1 Markov Decision Process

The problem considered in this project can be structured as an MDP, see Section 2.2.1. At every step, the agent observes the current state of the environment, takes an action based on this, and receives a reward for that action. This repeats until the goal is reached. The MDP 4-tuple was defined as:

- State space $\boldsymbol{S}$: The state at each step was what the agent observed through the camera sensor, as well as additional features extracted from the environment, namely,
  - d, the agent's Euclidean distance to the next waypoint in m,
  - $\theta \in [0, 360]$, the agent's angle to the lane heading,
  - v, the agent's speed in km/h.

  The image obtained from the camera sensor was fed to the CNN, and the output was concatenated with the additional features before the first fully connected layer.

For the camera-free approach, the state space had to be modified such that the agent could learn where to go without using a camera sensor. Therefore, the state space here consisted of

  - An array of relative angles between the agent and the next 15 waypoints. The array is similar to a queue. When the first waypoint was reached, it was removed from the array and another waypoint was appended at the end, always maintaining an array of 15 waypoints.
  - d_c, Euclidean distance to the center of the lane in m.
  - $\theta \in [0, 360]$, the agent's angle to the lane heading.
  - v, the agent's speed in km/h.

- Action space $\boldsymbol{A}$: The agent's actions in CARLA with the camera-based approach were defined as 4 discrete actions, as in Table 1. The camera-free approach had 27 discrete actions, see Table 2, to better compare to the method in [16]. In CARLA, the range of throttle and steer are: Throttle $\in [0, 1]$, Steer $\in [-1, 1]$. In action No. 3 in Table 1 the steering remained the same as the previous step.
- The probability function: $p(s', r \mid s, a)$, see (2.1), which was used to find the optimal policy in the agent's decision process.
- The reward function $\boldsymbol{R}$: The agent was both rewarded and punished. For the camera-based approach, the reward was received when driving straight in relation to the lane, below the speed-limit, and driving within 3 meters of the next waypoint (boolean $W \in 0, 1$). It was punished for collisions (boolean $C \in 0, 1$), crossing solid lane markings (boolean, $L \in 0, 1$), and driving faster than the speed-limit. Thus, the reward function was defined as:

  if $v <$ speed-limit:

$$\boldsymbol{R}(d, \theta, v, W, C, L) = v \cdot \cos(\theta) + 100 \cdot W - 200 \cdot C - 200 \cdot L, \tag{3.1}$$

  else:

$$\boldsymbol{R}(d, \theta, v, W, C, L) = -v + 100 \cdot W - 200 \cdot C - 200 \cdot L, \tag{3.2}$$

  where the weights were based on the weights in [16].

  The camera-free approach was different, but has some similarities. The agent was rewarded for driving straight and close to the center of the lane, and for reaching the next waypoint. It was also punished for colliding, and crossing solid lane markers. When the agent was in the lane, the reward function was

defined as

$$\boldsymbol{R}(d_c, \theta, v, W) = |v \cdot \cos\theta| - |v \cdot \sin\theta| - |v| \cdot |d_c| + 100 \cdot W - 200 \cdot C - 200 \cdot L. \quad (3.3)$$

The reward was inspired by the reward function used in [16], to be able to better compare the two models.

Table 1: A set of the available actions the agent could choose from in the camera-based approach. Throttle $\in [0, 1]$, Steer $\in [-1, 1]$. In action 3, the steering remained the same as in the previous step.

Action	Throttle	Steer
0	0.5	-0.1
1	0.5	0
2	0.5	0.1
3	0	-

Table 2: A set of the agent's available actions for the camera-free approach. Throttle $\in [0, 1]$, Steer $\in [-1, 1]$. Every throttle value was combined with every steering value, resulting in 27 actions.

No. of Actions	Throttle	Steer
27	0, 0.5. 1	-1, -0.75, . . . , 0.75, 1

3.2.2 Training and Validation With Camera

The agent was trained in an ϵ-greedy fashion on pre-determined routes, where waypoints were placed along the center of the lane the agent was to follow. The waypoints could be seen in the world and the agent could see them through the camera sensor (Figure 7). The input to the CNN were the images from the camera sensor, with the output being concatenated with the agent's distance to the next waypoint, its angle from the lane heading, and its speed, then fed into the first fully-connected layer. The states and inputs were saved in the agents' experience replay storage, helping it learn from experience. The agent was rewarded and punished based on its actions, see the reward functions (3.1) and (3.2).

Furthermore, the agent was also punished if it were to have a collision or cross solid lane markings, with the intention that the agent would avoid collision and stay within the lane. The latter two punishments were also terminations of the episode, meaning that the episode would terminate, and a new one begin, if these occur.

When all training episodes were finished, the model was saved and validated on one straight route during 50 episodes, to assess whether the agent was capable of staying within the lane and follow the waypoints. The straight route was chosen as evaluation because it was the simplest, and gave a clear indication of how well the agent had learned.

3.2.3 Training and Validation Without Camera

Similar to the camera-based approach, the agent was trained in an ϵ-greedy fashion, but on random routes. The waypoints were placed along the center of the lane the agent was supposed to follow. Because the agent was not using a camera to see the waypoints, the angles between the agent and the next 15 waypoints were sent to the agent as a part of the state. The angles towards the waypoints were updated every step. This gave the agent an idea of where to go. The state was one array, with the waypoint angles between the agent and the waypoints, concatenated with the distance to the center of the lane, the angle from the lane heading, and the agent's speed. This state array was fed into the first fully-connected layer. The agent's experiences were saved to the experience replay storage to allow the agent to learn from experience. The agent was rewarded and punished based on its actions, as defined by (3.3). Furthermore, when the agent crosses a solid lane marker, or collide, the episode would terminate, and a new episode would begin.

When the training was finished, the model was saved and validation was commenced on a straight route, to assess how well the agent had learned to stay inside the lane and follow waypoints. Validation was performed for 50 episodes.

3.2.4 Approaches

Several different training approaches, displayed in Table 3, have been considered when solving the problem.

Table 3: Training approaches considered during training of the model.

Approach	Image	Routes	Action Set	Buildings
3.1	128x128-RGB	4	Table 1	No
3.2	No Image	Random	Table 2	No
3.3	No Image	Random	Table 2	Yes

Approach 3.1 in Table 3 was using the camera sensor with additional features consisting of the agent's distance to the next waypoint, its angle from the lane heading, and its speed, concatenated to the output of the CNN before the first fully connected layer.

Approaches 3.2 and 3.3 in Table 3 did not use the camera sensor and therefore needed only a regular neural network. Instead, these approaches used a state array containing the angles between the agent and the next 15 waypoints, the distance to the center of the lane, the agent's angle to the lane heading, and the agent's speed. This state array was fed to the first fully-connected layer in the neural network.

The goal of using an RGB image was to make it easier for the agent to distinguish the waypoints and important signs from the rest of the environment compared to using a grayscale image. However, by transforming the image to grayscale, effectively removing 2 dimensions, should result in a reduction of the computational load.

With the 4 different routes approach, the training of the agent took place on 4 crafted routes. At the beginning of each episode, a route was chosen randomly, and the agent was spawned at the beginning. The "Random" value for the routes in Table 3 refers to the start position and destination being chosen completely at random among the city maps spawn points. The waypoints were generated as usual.

The last column in Table 3 named "Buildings" refers to if the map was with or without buildings. With buildings the environment was a complete city, containing multiple story buildings, parking houses, museums, etc. However, without buildings, the environment contained only the roads. It was theorized that for the camera-based approach, the buildings might interfere with the CNN and distract the agent from learning where the waypoints were. Furthermore, the buildings could also have an impact on the performance of the CARLA program.

3.2.5 Route Design

The 4-route approach was more focused on teaching the agent to specifically accomplish the driving and waypoint-following task. In essence, each route was designed for a particular action. The routes were: sharp left turn in junction, straight lane, soft right turn in lane, and soft left turn in lane. The waypoints for these routes had to be placed in the city at the same time. Hence, the waypoints had to be placed such that the agent would not see any of the waypoints belonging to the other routes, when it was driving one route. As a result, a route incorporating a sharp right turn was not feasible.

The completely random route design was only used in the camera-free approach, where the goal was for the agent to only learn to follow the waypoints. This route design was meant to help the agent learn quicker by exploring many different routes. This route design was incorporated in the later stages of this project, when

training of approach 3.1 in Table 3 had already been completed and validated. Therefore, this route design was not used in approach 3.1.

4 Results & Discussion

In this section, the results from the approaches in Table 3 will be presented, together with discussion about their results.

4.1 Evaluation Method

The evaluation method used in this project was to let the agent try to follow a straight route for 50 episodes, with a 20-second time limit. The assessment was based on how far the agent managed to drive this route without driving outside the lane. For a well-trained model, the agent should be able to complete this route within 20 seconds. An image of the route is presented in Figure 8.

However, there were various ways to evaluate the driving of the autonomous vehicle. Some examples are:

- distance driven without human intervention.
- time driven without human intervention.
- number of times the agent successfully reached a destination within a time limit.
- average distance driven between traffic violations.

These are mostly evaluations on a smaller scale. Naturally, there are larger scale evaluation methods, such as the Quantitative Evaluation for Driving (QED) method proposed in [24]. QED aims to assess several features of the driving within one scoring method. Some of the included features are the ability to stay in the center of the lane, follow the speed limit, and avoid collisions.

Since this project was focusing more towards the implementation of the DQN algorithm and its challenges for an autonomous vehicle, an advanced evaluation method, such as QED, was not considered.

4.2 Training and Validation

During training, the agent learned for a maximum of 20×10^3 episodes with an ϵ-greedy policy with ϵ starting on 0.9 and ending on 0.1, with ϵ-decay set to 4×10^{-5} in the camera-free approach. In the camera-based approach, the agent was trained on intervals of 2×10^3 episodes, and no maximum training episodes

were set. Therefore, the ϵ-decay was set to 4.4×10^{-4} such that epsilon started on 0.9 and ended on 0.1 every interval. The agent's score, average score, and loss were plotted to assess how the agent performed and learned.

During validation, the agent performed with maximum greed, i.e. ϵ set to 0. The agent was evaluated on a 50 episode run on a straight route, see Figure 8, with a 20-second time limit. Likewise, the agent's score, average score and route completion was plotted to assess how well the agent performed.

Figure 8: The straight route, the agent was evaluated on.

4.2.1 Approach With Camera

This approach was trained for 16×10^3 episodes, which took roughly 48 hours to complete.

Figure 9a shows the average score during the 16×10^3 episodes of training. The average score was slowly increasing as ϵ was decreasing, indicating that the agent was learning how to gain the most reward.

Each episode's score is displayed in Figure 9b. In some episodes, the agent received a much higher reward than in others, namely, close to episodes 3.5×10^3, 5.5×10^3, 7.5×10^3, and 12×10^3. This is most likely due to the exploration of the agent accidentally following the route exactly, or the route during these episodes was considered a simpler one by the agent.

In Figure 9c, the neural network's loss is shown. Ideally, the loss should not bounce around as much as it did, but converge around a value. There are many reasons for why the loss may not converge:

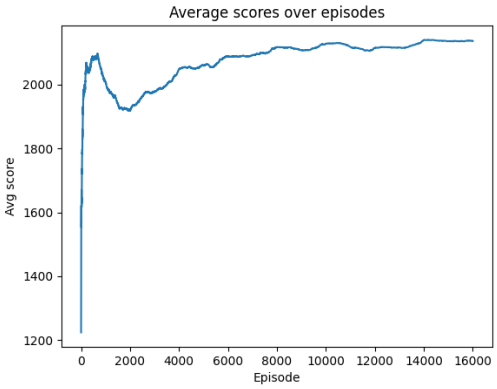
- Learning rate too large.

- Batch size too large or small.
- Neural network architecture not well-suited.
- Hyperparameter tuning.
- Not enough training episodes.

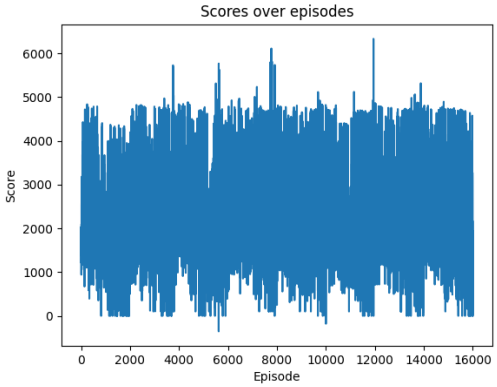
Since these reasons all affect the model's predictive capability, they would also affect the loss.

Unfortunately, because the time investment required to train the model is very large, it was beyond the scope of this projects' timetable to evaluate the effect of all these parameters.

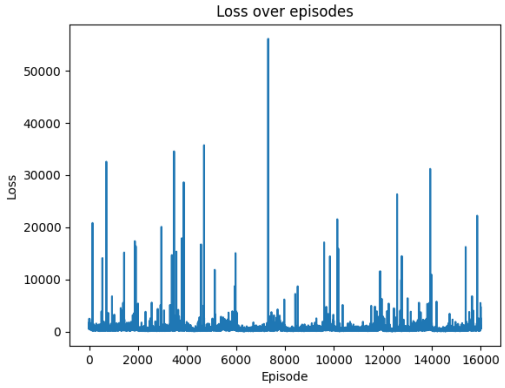
Evaluating the camera-based approach on the straight route for 50 episodes, with a time limit of 20 seconds per episode, yields the results in Figure 10. The figure that best shows how well the agent performed on the straight route is the route completion in Figure 10c. This figure displays how many waypoints of the route the agent reached before driving outside the lane or reaching the time limit. The most successful episode was episode 22, where the agent completed about 85% of the route. The average route completion during these 50 episodes was 36%. Evidently, the model had not learned enough to follow a straight route consistently. Based on the training data presented in Figure 9, it is possible that the model required additional training and experimentation with the neural network and its hyperparameters in order to enhance its performance.



(a) Average score

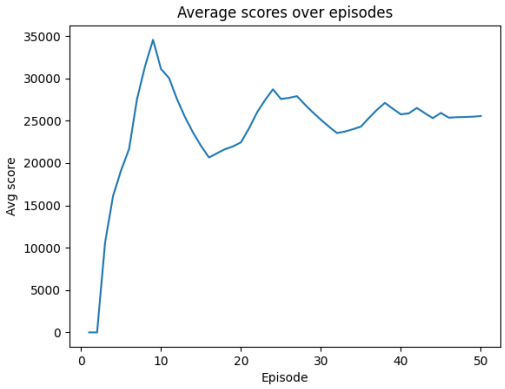


(b) Score

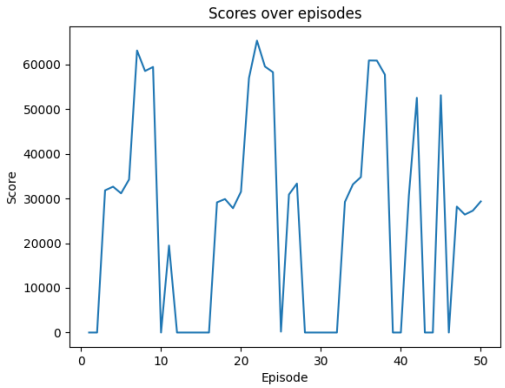


(c) MSE loss

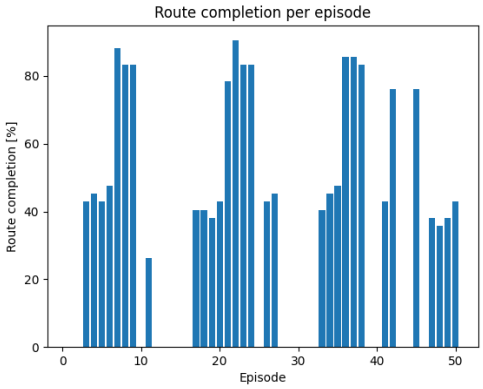
Figure 9: Results from 16×10^3 episodes training using a 128x128 RGB image as input to the CNN. Batch size 32, learning rate 1×10^{-5}.



(a) Average score



(b) Score



(c) Route completion per episode

Figure 10: Results from 50 episodes evaluation of the model trained for 16×10^3 episodes using 128x128 RGB image as input to the CNN. Batch size 32, learning rate 1×10^{-5}.

4.2.2 Approach Without Camera

Because the camera-based approach did not achieve a satisfactory result, the method without a camera sensor (3.2, and 3.3 in Table 3) was implemented based on the similar method in [16]. Both approaches were trained for 20×10^3 episodes with the same neural network structure, batch size, learning rate, epsilon start/end, and epsilon decay. Training these two approaches for 20×10^3 episodes took approximately 40 hours each.

The training result for approach 3.2 is seen in Figure 11. The learning rate was the same as the camera-based approach, but the batch size was larger. When an image no longer had to be fed to the neural network, the algorithm was faster. Therefore, a larger batch size should have provided a more stable learning experience for the agent without affecting the algorithm performance. However, it would have been beneficial to experiment with different batch sizes to find the optimal one.

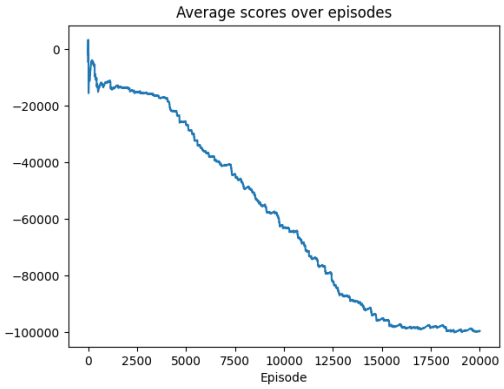
The per episode average score, score, and loss are displayed in Figure 11a, Figure 11b, and Figure 11c, respectively. In Figure 11a the average score per episode was decreasing during the whole training run except around episode 15×10^3 where it began to stabilize. The reason for the stabilization at the end is likely because this was where ϵ began to reach its minimum value and the agent was starting to use a greedy policy.

Furthermore, this approach was without the buildings present in the environment, which allowed the agent to drive off the map in certain locations because of inconsistencies with the lane invasion sensor, see Figure 15. When this occurred, the agent received a very large punishment, which can be seen in the scores in Figure 11b. Consequently, the average score was skewed because of this large punishment.

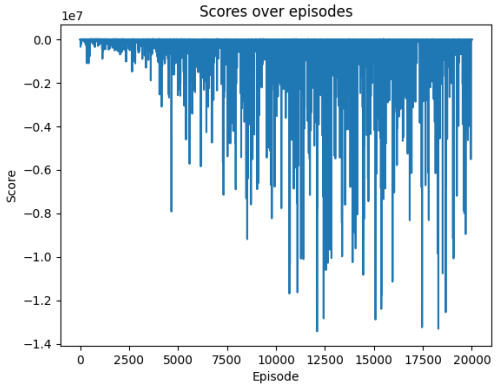
The loss in Figure 11c seems to be trending upwards, but was stabilized until right before episode 5×10^3. Comparing this to the loss in the camera-based approach, where the loss was never stable, this seems like an improvement. However, the loss for approach 3.2 overall is far greater than the camera-based approach, and would also benefit from model optimizations.

The results from evaluating approach 3.2 are displayed in Figure 12. In Figure 12c, it can be seen that the farthest the agent reached on the straight route was around 3%, if the agent managed to reach a waypoint at all. Clearly, this approach was not successful in learning to follow waypoints.

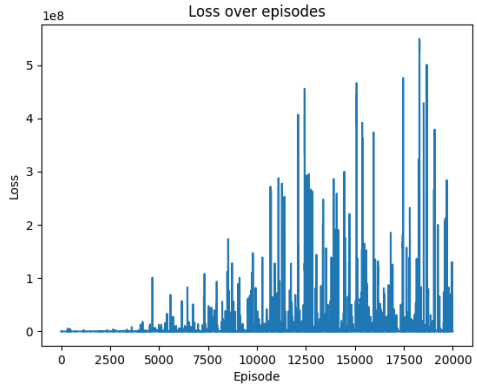
The training data after 20×10^3 episodes using approach 3.3 is displayed in Figure 13. With the buildings present in this approach, the agent would now collide with an object instead of driving off the map. Therefore, the average score in



(a) Average score



(b) Score



(c) MSE loss

Figure 11: Results from 20×10^3 episodes training using the camera-free approach, 3.2 in Table 3. Batch size 512, learning rate 1×10^{-5}.

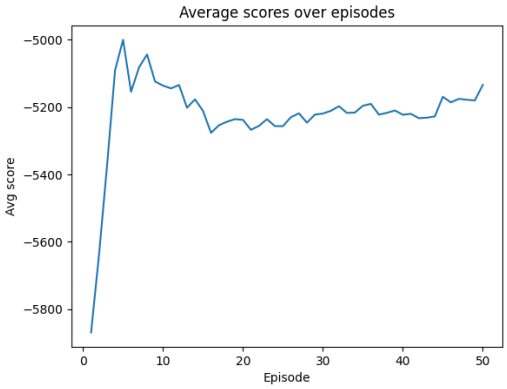
Figure 13a is not skewed from the unexpected punishment. Moreover, Figure 13a shows the average score beginning to balance by episode 10×10^3, with a potential to increase if further training was provided.

In Figure 13b there were some episodes where the score peaked. Visually inspecting the agent while training, these were episodes where the agent followed the waypoints well.

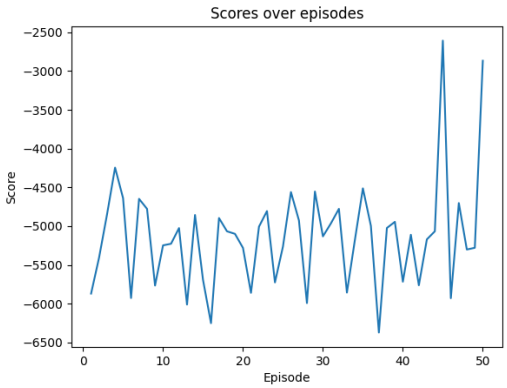
The loss in Figure 13c is similar to the camera-based approach, meaning that it did not converge. The model's performance would have likely improved if there had been enough time to tune the hyperparameters.

When evaluating approach 3.3 on the straight route for 50 episodes using the model trained for 20×10^3 episodes, Figure 14c shows that the agent did not manage to reach the first waypoint during any episode. Nonetheless, the agent did attempt different actions, demonstrated by the different scores per episode in Figure 14a and Figure 14b.

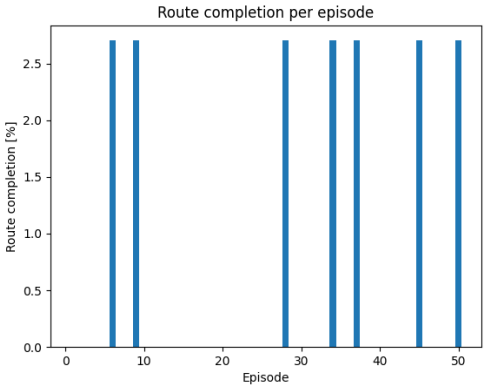
Seemingly, the agent has not learned to follow the waypoints well enough during the 20×10^3 episode training. Thus, the model most likely needs to be optimized by tuning the hyperparameters.



(a) Average score

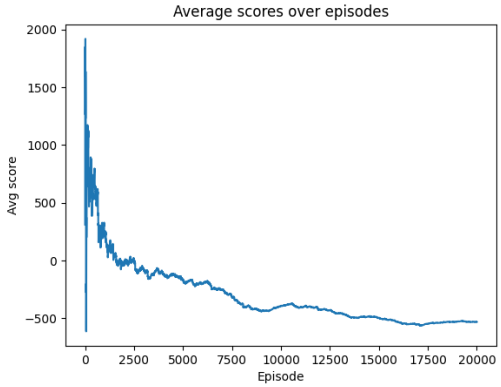


(b) Score

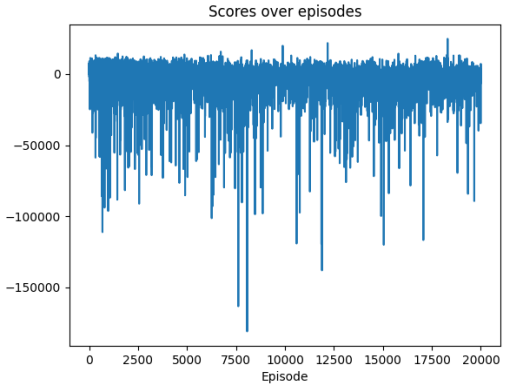


(c) Route completion per episode

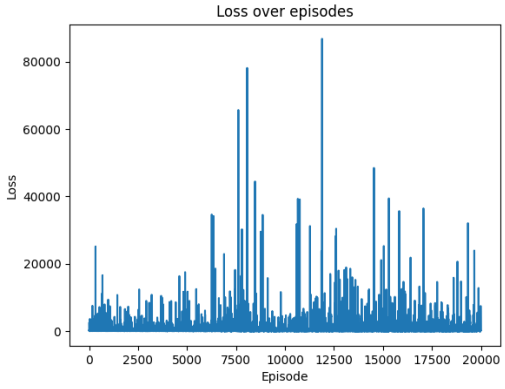
Figure 12: Results from 50 episodes evaluation of the model trained for 20×10^3 episodes using the camera-free approach, 3.2 in Table 3. Batch size 512, learning rate 1×10^{-5}.



(a) Average score

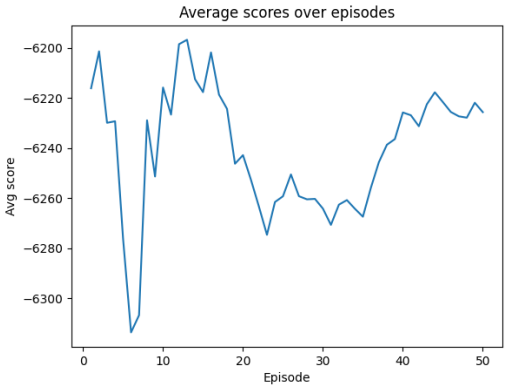


(b) Score

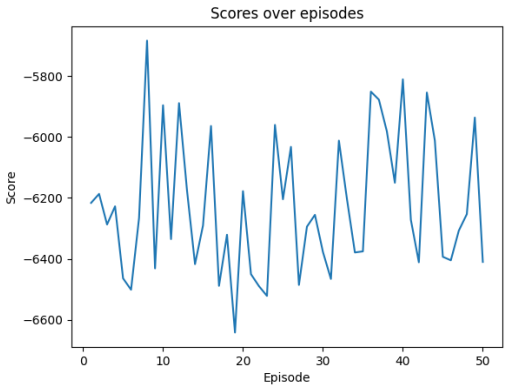


(c) MSE loss

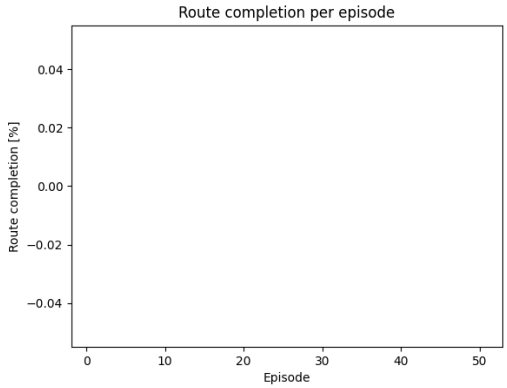
Figure 13: Results from 20×10^3 episodes training using the camera-free approach, 3.3 in Table 3. Batch size 512, learning rate 1×10^{-5}.



(a) Average score



(b) Score



(c) Route completion per episode

Figure 14: Results from 50 episodes evaluation of the model trained for 20×10^3 episodes using the camera-free approach, 3.3 in Table 3. Batch size 512, learning rate 1×10^{-5}.

4.3 Further Discussions

The image resolution for the camera-based approach was chosen to be 128x128, because it was the most efficient image resolution for the computer used in this project. Both 84x84 and 256x256 were tried. The 256x256 resolution yielded more information, but at the same time increased the required computational resources. The larger resolution caused the agent to sometimes not register when it reached a waypoint, which is detrimental to the training process. The smaller resolution of the 84x84 image gave no additional impact on the performance of the training compared to 128x128. However, the smaller resolution contains less information and so if the performance of 128x128 and 84x84 is satisfactory, then the larger resolution is preferable.

The goal of implementing the camera-free approaches was to compare them to the similar, successful, approach in [16]. Unfortunately, in [16] it is not disclosed how the neural network was configured or how the hyperparameters were set. Furthermore, there are differences between how the agent is made aware of the waypoints. In this project, the agent received the relative angle between itself and the next waypoints, which was updated each step. In [16], a transformation matrix and the x-coordinates of the waypoints with respect to the agent, was given to the agent to find the waypoints. Hence, it is not feasible to directly compare the approaches. However, it is clear from [16] that this type of approach can be trained to successfully follow waypoints within 20×10^3 training episodes, given the right hyperparameters and neural network configuration.

Unfortunately, the lane markers in CARLA did not seem 100% consistent. In Figure 15, the agent vehicle is seen crossing a solid lane marker without the episode ending. CARLA included an example code that, among several features, allowed the user to take manual control of a vehicle and alerted the user when a lane marker was crossed. By using this code it could be confirmed that the area in Figure 15, next to the intersection, had solid lane markers that were not recognized by the lane invasion sensor. This made the training of the agent more difficult when it was sometimes not punished for crossing these lane markers.

Figure 15: Frame capture of the agent vehicle crossing a solid lane marker without being flagged by the lane invasion sensor.

5 Conclusion & Future Work

In this project, a DQN algorithm has been implemented using Python in order to train a model to drive a vehicle in a city environment by following waypoints. The environment and driving physics were provided by the autonomous driving research simulator CARLA.

Unfortunately, the method and approaches considered during this project were not successful in training an agent to drive a car in a city environment. Before implementing other vehicles and pedestrians, the first goal was to train the agent to drive a short route in an empty city, without crossing the solid lane markings. This first challenge already proved difficult for the DQN. Due to the time cost of both training and optimizing the hyperparameters, the agent was unsuccessful in learning how to follow a route in the amount of training episodes provided.

Furthermore, because the camera-based approach was not successful, approaches 3.2 and 3.3 in Table 3 were implemented, based on the similar approach in [16]. The goal of the latter approaches, was to learn to only follow waypoints, ignoring traffic rules. Likewise, the model was also unsuccessful in learning to only follow waypoints using these approaches. However, it is proved in [16] that it is possible to train a model to accomplish this task, given an optimized neural network, optimized hyperparameters, and enough training.

To improve upon the work executed in this project, training the agent for more episodes would most probably result in a better model. In [16], where a similar problem is solved using a DQN and CNN, they trained for 120×10^3 episodes. This

would take at least 15 days with the approach in this project. A solution could be to run training for several agents simultaneously to accelerate the learning. However, this would require a more powerful system than the one used in this project, see system specification in Section 3.

Furthermore, it could benefit the training of the camera-based approach if it was trained on the random route design, similarly to the camera-free approach, instead of the 4-route design. This would create several different scenarios for the agent to learn from and decrease the familiarity of the 4-route design.

There is room to optimize the reward function with better fitting values for the rewards and punishments. Especially, for approach 3.2, where the agent would receive a huge punishment for driving off the map. Either driving off the map needs to be prevented, or the punishment decreased.

In the camera-based approach, instead of letting the agent always see all the waypoints, an improvement could be to only show the next or the two/three next waypoints. As of now, if the route is a curve, the agent can basically see all waypoints, which could result in confusion. Furthermore, waypoints farther from the camera still appear as large as waypoints closer to the camera, which could also add to confusion. All waypoints are drawn when starting the simulation using a debug function. In the present version of CARLA, there is no way to remove drawn waypoints without restarting the simulation, which would result in even longer training times. Alternatively, the drawn waypoints can be set to only last a set time. For example, the first three waypoints can be drawn for 5 seconds, then the next three for 5 seconds, but this also poses a problem. The best would be for the CARLA team to implement a removal function for debug elements, otherwise another way to create waypoints is needed.

As shown in Figure 15, the lane markers were not always flagged by the lane invasion sensor. Upon further inspection, this seemed to be the case when there was an intersection nearby. Therefore, the training stage would benefit from ensuring that the lane markers the agent can cross are actually being flagged by the lane invasion sensor. Alternatively, implementing another method to identify when a lane marker is being crossed, or when the agent leaves the lane.

Unfortunately, a large part of the project consisted of troubleshooting a CARLA Unreal Engine 4 crashing issue in CARLA version 0.9.14, in which the program required to run the training would crash randomly. This severely hindered the ability to make progress on the solution of the considered problem. However, a downgrade to version 0.9.13 did not have this issue and therefore for future work, it is recommended to use this CARLA version or trying CARLA on a better system, another supported OS, i.e. Linux, or finding another simulator.

The largest improvement to the model would be to optimize the neural network configuration, as well as experiment with the hyperparameters to find the best suitable values for the considered problem.

References

[1] L. R. Soenksen, T. Kassis, S. T. Conover *et al.*, 'Using deep learning for dermatologist-level detection of suspicious pigmented skin lesions from wide-field images,' *Science Translational Medicine*, vol. 13, no. 581, eabb3652, 2021. DOI: 10.1126/scitranslmed.abb3652.

[2] J.-A. Choi and K. Lim, 'Identifying machine learning techniques for classification of target advertising,' *ICT Express*, vol. 6, no. 3, pp. 175–180, 2020, ISSN: 2405-9595. DOI: https://doi.org/10.1016/j.icte.2020.04.012.

[3] World Health Organization, *Global status report on road safety 2018: Summary*, 2018. [Online]. Available: https://www.who.int/publications/i/item/9789241565684 (visited on 16/02/2023).

[4] B. R. Kiran, I. Sobh, V. Talpaert *et al.*, 'Deep reinforcement learning for autonomous driving: A survey,' *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 6, pp. 4909–4926, 2022. DOI: 10.1109/TITS.2021.3054625.

[5] CB Insights. '40+ corporations working on autonomous vehicles.' (), [Online]. Available: https://www.cbinsights.com/research/autonomous-driverless-vehicles-corporations-list/.

[6] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Mar. 1998.

[7] *Alphago*. [Online]. Available: https://www.deepmind.com/research/highlighted-research/alphago (visited on 24/09/2023).

[8] Y. Li, *Deep reinforcement learning*, 2018. arXiv: 1810.06339 [cs.LG].

[9] K. Arulkumaran, M. P. Deisenroth, M. Brundage *et al.*, 'Deep reinforcement learning: A brief survey,' *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, Nov. 2017. DOI: 10.1109/msp.2017.2743240.

[10] V. Mnih, K. Kavukcuoglu, D. Silver *et al.*, 'Human-level control through deep reinforcement learning,' *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015, ISSN: 1476-4687. DOI: 10.1038/nature14236.

[11] A. Dosovitskiy, G. Ros, F. Codevilla *et al.*, 'CARLA: An open urban driving simulator,' in *Proceedings of the 1st Annual Conference on Robot Learning*, 2017, pp. 1–16.

[12] E. Yurtsever, J. Lambert, A. Carballo *et al.*, 'A survey of autonomous driving: Common practices and emerging technologies,' *IEEE Access*, vol. 8, pp. 58 443–58 469, 2020. DOI: 10.1109/ACCESS.2020.2983149.

[13] M. Bojarski, D. D. Testa, D. Dworakowski *et al.*, *End to end learning for self-driving cars*, 2016. arXiv: 1604.07316 [cs.CV].

[14] F. Codevilla, M. Müller, A. López *et al.*, *End-to-end driving via conditional imitation learning*, 2018. arXiv: 1710.02410 [cs.RO].

[15] Z. Huang, 'Distributed reinforcement learning for autonomous driving,' M.S. thesis, Carnegie Mellon University, Pittsburgh, PA, May 2022.

[16] Ó. Pérez-Gil, R. Barea, E. López-Guillén *et al.*, 'Dqn-based deep reinforcement learning for autonomous driving,' in *Advances in Physical Agents II*, L. M. Bergasa, M. Ocaña, R. Barea *et al.*, Eds., Cham: Springer International Publishing, 2021, pp. 60–76, ISBN: 978-3-030-62579-5.

[17] A. Jain, J. Mao and K. Mohiuddin, 'Artificial neural networks: A tutorial,' *Computer*, vol. 29, no. 3, pp. 31–44, 1996. DOI: 10.1109/2.485891.

[18] Wikipedia, *Artificial neural network*. [Online]. Available: https://en.wikipedia.org/wiki/Artificial_neural_network (visited on 23/05/2023).

[19] K. O'Shea and R. Nash, *An introduction to convolutional neural networks*, 2015. arXiv: 1511.08458 [cs.NE].

[20] Phung and Rhee, 'A high-accuracy model average ensemble of convolutional neural networks for classification of cloud image patches on small datasets,' *Applied Sciences*, vol. 9, p. 4500, Oct. 2019. DOI: 10.3390/app9214500.

[21] V. Mnih, K. Kavukcuoglu, D. Silver *et al.*, *Playing atari with deep reinforcement learning*, 2013. arXiv: 1312.5602 [cs.LG].

[22] R. Liessner, J. Schmitt, A. Dietermann *et al.*, 'Hyperparameter optimization for deep reinforcement learning in vehicle energy management,' Feb. 2019. DOI: 10.5220/0007364701340144.

[23] A. Paszke, S. Gross, F. Massa *et al.*, 'PyTorch: An Imperative Style, High-Performance Deep Learning Library,' in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer *et al.*, Eds., Curran Associates, Inc., 2019, pp. 8024–8035.

[24] S. Gao, S. Paulissen, M. Coletti *et al.*, 'Quantitative evaluation of autonomous driving in carla,' Jul. 2021.