Deep Reinforcement Learning for End-to-End autonomous driving

Research Paper
MSc Business Analytics
Vrije Universiteit Amsterdam

Touati, J. - 2540273,

Supervisors:
Slik, J.
Van Goor-Balk, H.J.M.

July 29, 2019

Abstract

Quite some research has been devoted to the field of reinforcement learning, however, applying this field to autonomous driving still requires extensive research. This results directly from a safety concern of applying them in real life. Reinforcement learning approaches have recently achieved superhuman performance in games such as Go, Chess, and classic video games. However, these algorithms are sensitive to perturbations in the environment and often struggle or catastrophically fail when applied to environments that differ from where they were trained. During this research, multiple reinforcement learning techniques are used in order to learn an agent to drive through a racing environment. Here, the main goal is to understand which reinforcement learning algorithms are viable for this task. In addition, evaluating, comparing, and interpreting the algorithms are central topics. The results show that the deep SARSA algorithm with ϵ-greedy exploration achieves the best performance within the racing environment.

Table of Contents

1 Introduction

People have been experimenting with the idea of self-driving vehicles since the 1920s. However, the first truly autonomous vehicles appeared in the 1980s when two mobile robots were capable of driving the vehicle down a road in continuous motion [1]. This was a big step forward in the goal of creating a fully autonomous vehicle, but there was still a long way to go. It wouldn't be until 2013 for the first tests of autonomous vehicles on the public road, where a robotic vehicle succeeded in navigating through roundabouts, traffic lights, pedestrian crossings, and other common hazards [2]. Now, almost 100 years later than the first experiments, we often see vehicles of the Tesla brand in the streets. These vehicles have the functionality of driving fully autonomous, but due to country specific legislation it is not allowed to use it yet. This results directly from an understandable safety concern. It is therefore important that the techniques used for autonomous driving are tested extensively.

One might think that driving autonomously is purely a luxury functionality. However, the main reasons for investigating techniques that allow vehicles to drive without human interference are far more useful. Firstly, driving without human interference will not leave any room for human error. Assuming that any autonomous driving technique can reach human level performance, this would mean that the number of traffic casualties would decrease. So, from a safety perspective alone it would be valuable to investigate these techniques. In addition, autonomous driving could lower costs in public transport and transport of goods, since a human driver is redundant. Finally, vehicles that don't require human interference also don't require the mechanics to control the car such as a steering wheel or the gas pedal. This could lower the cost of production of vehicles and therefore the price of acquiring them. From all this, the importance rises to investigate techniques that allow vehicles to drive autonomously.

To describe the problem at hand, first consider the term *"end-to-end autonomous driving"*. This is defined as the task of driving by a self-contained system that uses sensory input, such as an image frame from a front-facing camera, to take actions necessary for driving, such as accelerating, braking, or the angle of the steering wheel [3]. This self-contained system will be referred to as the agent. The agent has to learn its actions from data rather than manual instructions, mainly due to the complexity of developing manual instructions for such a system. This problem can be divided into two tasks, namely a recognition task and a modelling/learning task.

The recognition task consists of identifying components of the environment such as the lane, pedestrians, roadwork, and other vehicles. Nowadays, this task has been studied extensively where Deep Learning algorithms and in particular Convolutional Neural Networks have been proven to reach human level capabilities for object detection and recognition [4][5]. In addition, this task has been replicated in lane and vehicle detection for autonomous driving [6]. Given this extensive progress in the recognition task, it seems to be more valuable to investigate the second task. Therefore, the focus of this paper will be the modelling/learning task.

Using the output of the recognition task, the driving agent can observe the surrounding environment. Based on this, the agent has to decide what driving actions to take for successful navigation. It is difficult to pose this task as a supervised learning problem, as there is a strong interaction with the environment including other vehicles, pedestrians, and roadworks. Reinforcement learning techniques seem better candidates for this problem as they can be applied in situations where the environment is a non-stationary one [7]. In addition, supervised learning techniques require training examples from a human driver. It seems impossible to encounter all possible traffic situations in these training examples, especially situations in which accidents happen. Therefore, only reinforcement learning techniques will be considered for the modelling/learning task. A more detailed description about this task will be given in the next section.

The goal of this research is to understand which reinforcement learning algorithms are viable for the modelling/learning task. Specifically, deep reinforcement learning in combination with on-policy and off-policy algorithms will be considered. An OpenAI Gym environment (Figure 1) [8] will be used to apply these algorithms.

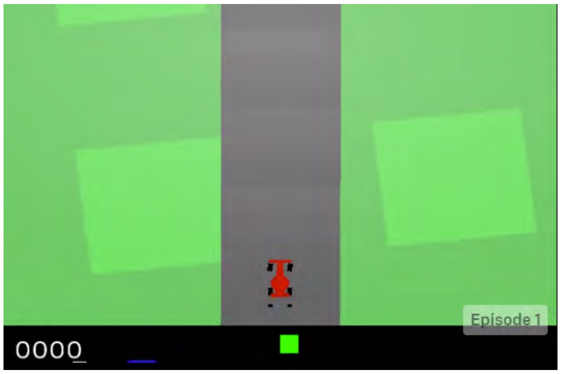

Fig. 1. CarRacing-v0 environment of OpenAI Gym

Related literature will be discussed in the next section, in order to investigate which reinforcement learning techniques are used for the problem. In Section 3, the methods that are implemented in the CarRacing-v0 environment will be explained in detail. Next, the experiments and the implementation will be discussed and the results of the experiments will be shown in Section 5. Finally, the results will be discussed and a conclusion will be drawn.

2 Related Literature

The modelling/learning task has been examined in multiple studies in the past few years. The techniques that seem most promising from these studies will be considered in this section. Most of these techniques are based on the standard reinforcement learning framework.

The standard reinforcement learning framework was formulated in [9] as a model to provide the best policy an agent can follow (best action to take in a given state), such that the total accumulated rewards are maximized when the agent follows that policy from the current and until a terminal state is reached. This standard reinforcement learning structure can be seen in Figure 2.

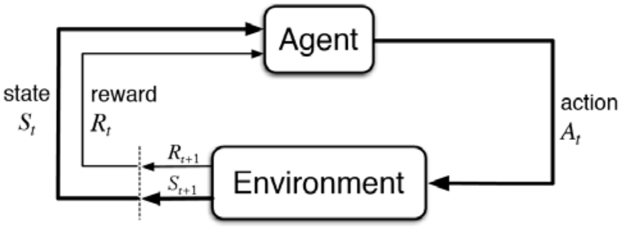

Fig. 2. Standard reinforcement learning framework

Based on the current state, the agent has to decide which action to take. In this decision the agent has to balance its choice between exploitation and exploration. In other words, does the agent want to exploit its current knowledge of the environment and take the action that seems best, or does the agent want to explore a different action in order to see if there is more reward to gain. Palm & Tokie state that balancing the ratio between exploration and exploitation is one of the most challenging tasks in reinforcement learning with a great impact on the agent's learning performance [10]. There is no holy grail when it comes to exploration mechanisms and it is therefore important to investigate multiple techniques. Well known and widely used exploration mechanisms are ϵ-greedy exploration and softmax exploration [11] [12]. Tijsma et al. shows that softmax exploration outperforms ϵ-greedy exploration in their experimental setting, although it is harder to tune its temperature parameter [13].

The standard reinforcement learning setting uses an action value function $Q(s, a)$ to quantify how much reward an action will yield in a certain state. In this case, the Q-values (action values) can be represented as a table where the rows are actions and the columns are states. An entry in this table then corresponds to the Q-value of an action a in a state s. This standard setting can be applied to problems where the space of possible actions and the space of possible states are discrete [14]. However, when the state space and the action space are continuous it becomes harder, since both spaces become infinitely large. The recognition task uses high-dimensional sensory input like an image frame and has to transform this to a state of the environment. This makes the problem harder, since the state space increases significantly. Deep reinforcement learning can solve this problem by using a function approximator, in order to represent the action value function. Neural networks are widely used as function approximators [15]. The neural network approximates the Q-values in a state and it solves the problem of dealing with a continuous state space. The term 'deep' reinforcement learning refers to a large number of hidden nodes and layers within the neural network. A continuous action space can easily be transformed to a discrete action space by defining possible values for the actions.

Deep Q Networks (DQN's) refer to the deep neural networks that approximate the Q-values and they have been applied in several autonomous driving settings [14] [16]. Well known reinforcement learning algorithms such as Q-learning and SARSA can be combined with these deep learning techniques. However, Mnih et al. state that it was previously thought that the combination of simple online reinforcement learning algorithms with deep neural networks was fundamentally unstable [15]. Instead, a variety of solutions have been proposed to stabilize the algorithm [17] [12].

Most of these approaches share the common idea that the sequence of observed data encountered by an agent is non-stationary and the updates are therefore strongly correlated. The issue here is that the policy assumes that the observed data is i.i.d. (independent, identically distributed), but this is not the case when the data is obtained sequentially. By storing the agent's data in an experience replay buffer, the data can be batched or randomly sampled from different timesteps [11]. Randomizing the observed data and feeding it to the agent in a different order breaks the correlations and therefore reduces the variance of the updates [12]. Due to the randomization of the experience updates, each data sample is potentially used in many weight updates and this allows for greater data efficiency, which is another advantage.

Deep reinforcement learning algorithms based on experience replay have achieved unprecedented success in challenging domains such as the Atari games [12]. However, experience replay does have several drawbacks. Firstly, experience replay requires more memory and computation time per real interaction. This is caused by updating the weights of the model more often than without an experience replay buffer. Furthermore, when using experience replay to update the weights of a model, it is necessary to learn off-policy, because the current weights are different to those used to generate the data sample. An on-policy method cannot update the current weights with a data sample that was generated with different model weights, which motivates the choice of an off-policy method such as Q-learning.

3 Methods

3.1 Reinforcement Learning

In the standard reinforcement learning setting, an agent interacts with an environment over a number of discrete time steps. At each time step t, the agent observes a state s_t and selects an action a_t from some set of possible actions according to its policy π. In return, the agent observes the next state s_{t+1} and receives a reward r_t. An episode continues until the agent reaches a terminal state after which a new episode starts. The total accumulated return from time t is given by:

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

Where $\gamma \in (0, 1]$ is the discount factor. The goal of the agent is to maximize the expected return from each state s_t. To represent every action in all states, the action value function is defined as:

$$Q^{\pi}(s, a) = \mathbb{E}[R_t | s_t = s, a]. \tag{2}$$

The action value function is the expected return for selecting action a in state s while following π. The maximum action value for state s and action a achievable by any policy is given by $Q^*(s, a) = \max(Q^{\pi}(s, a))$. For a more detailed overview of reinforcement learning, please refer to the textbook of Sutton and Barto [18].

3.1.1 Q-Learning

A value-based model free method for reinforcement learning is Q-learning. Q-learning aims to find the optimal policy. With Q-learning, a number of episodes or trials are simulated and the Q-values are updated by taking the outcomes of these episodes into account. These outcomes consist of the obtained reward r for taking action a in state s and observing the next state s'. The Q-values are updated with the following update rule:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'}(Q(s', a')) - Q(s, a) \right]. \tag{3}$$

Where α is the learning rate and the part between brackets is referred to as the TD error. In addition, $\max_{a'}(Q(s', a'))$ is the maximum Q-value of the next state s' and corresponding next action a'. The update of the Q-values assume that we follow a greedy policy in the next state, so it does not take into account its own exploration mechanism. Q-Learning is therefore an off-policy algorithm.

A disadvantage for off-policy algorithms in general is that they have a higher per-sample variance than on-policy algorithms. As a result, off-policy algorithms may suffer from problems converging, especially in combination with neural networks.

However, off-policy algorithms are more risk-averse in the sense that when there is a risk of a large negative reward close to the optimal path, off-policy algorithms will tend to trigger that reward whilst exploring. On-policy algorithms would tend to avoid this reward and this may result in a higher expected reward for off-policy algorithms than for on-policy algorithms. This idea is illustrated in Figure 3.

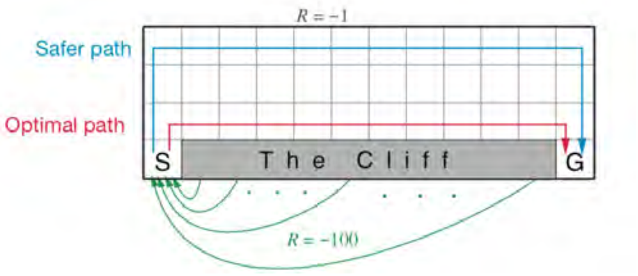

Fig. 3. Cliff walking: On-policy - Safer path, Off-policy - Optimal path

3.1.2 SARSA

In contrast to Q-Learning, SARSA learns the Q-values of the executed policy and does not approximate the optimal policy. SARSA is therefore an on-policy algorithm. This difference can be seen in the update rule for the Q-values:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma Q(s', a') - Q(s,a) \right]. \tag{4}$$

In this equation, we can see that the Q-value of the next state s' and the next executed action a' is used instead of the maximum Q-value of the next state with corresponding action. SARSA is therefore more conservative, as can be seen in Figure 3. SARSA will take the safer path, since it takes actions into account where the agent falls off the cliff. Q-learning updates the Q-values with the maximum Q-value which does not include actions that end up falling off the cliff. This is of course a quite simple example and it has be tested to see how this applies to more complex environments.

3.2 Exploration Mechanism

To balance the choice between exploration and exploitation, softmax and ϵ-greedy exploration will be considered for the implementation in the Carracing-v0 environment.

3.2.1 ϵ-greedy

ϵ-greedy exploration selects a random action with a probability ϵ. With probability $1 - \epsilon$ it exploits its current knowledge and takes the best action, i.e., $a^* \leftarrow \arg\max_a Q(s, a)$.

3.2.2 Softmax

Under softmax exploration, the probability of selecting an action in state s is linked to its Q-value estimate. Higher Q-values get rewarded with a higher probability of being selected. The greedy action is still given the highest selection probability, however, all other probabilities are ranked and weighted according to their Q-values. Softmax computes the probability of an action in a certain state in the following way:

$$\pi(a|s) = \frac{e^{\frac{Q(s,a)}{\tau}}}{\sum_{a'} e^{\frac{Q(s,a')}{\tau}}}. \tag{5}$$

Where $\tau > 0$ is a computational temperature. A high value for τ causes the probabilities of selecting the actions to be nearly equal. A low value causes a greater difference in the selection probabilities. Where a value of zero for τ results in taking the greedy action always.

3.3 Deep Reinforcement Learning

As explained in Section 2, deep reinforcement learning can be applied in situations where the state space is continuous. The Q values can be represented using a function approximator in a deep reinforcement learning setting. Neural networks are widely used function approximators, where the number of layers and the number of hidden nodes are often high. Most settings where these neural networks are used as function approximators solve the problem of dealing with high-dimensional sensory data. The neural networks can handle this type of input such as an image frame from a front-facing camera. The Deep Q Network (DQN) will be discussed in this section.

3.3.1 Deep Q Networks

To elaborate on how the Deep Q Network works, let us first define $Q(s, a, \theta)$ as the approximate Q-function with parameters θ. The parameters θ are the weights of the neural network. The updates to θ can be applied by a variety of reinforcement learning algorithms, such as SARSA. The solution then lies in finding the best parameter settings θ. As with any other neural network, the objective is to iteratively minimize a sequence of loss functions. In this research the objective shall be to minimize the mean squared error (MSE) of the Q-values. The loss function for an update of one-step SARSA is then defined as:

$$L_i(\theta_i) = \mathbb{E}\left(r + \gamma Q(s', a', \theta_{i-1}) - Q(s, a, \theta_i)\right)^2. \tag{6}$$

Where s' is the state after s and a' is the action after a. Furthermore, θ_{i-1} refers to the previous weights of the neural network. This is an example of one iteration of one-step SARSA, since it updates the Q-value towards the one-step return $\gamma Q(s', a', \theta_{i-1})$. The Deep SARSA algorithm is shown in Algorithm 1 and the Deep Q-learning algorithm is shown in Algorithm 2.

Algorithm 1 Deep SARSA with ϵ-greedy action selection

```
Initialize action-value function Q with random weights
for episode = 1, ..., M do
    Initialise state $s_1$
    $a_1 \leftarrow$ select action based on $\epsilon$-greedy exploration
    for $t = 1, ..., T$ do
        Simulate action $a_t$ and observe reward $r_t$ and next state $s_{t+1}$
        $a_{t+1} \leftarrow$ select action based on $\epsilon$-greedy exploration
        $\text{Set } y_i = \begin{cases} r_i & \text{for terminal } s_{t+1} \\ r_i + \gamma Q(s_{t+1}, a_{t+1}, \theta_i) & \text{for non-terminal } s_{t+1} \end{cases}$
        Perform a gradient descent step on $(y_i - Q(s_t, a_t, \theta))$ with respect to the
        network parameters $\theta$
        Set $s_t \leftarrow s_{t+1}$, $a_t \leftarrow a_{t+1}$
    end for
end for
```

Algorithm 2 Deep Q-learning with ϵ-greedy action selection

```
Initialize action-value function Q with random weights
for episode = 1, ..., M do
    Initialise state $s_1$
    for $t = 1, ..., T$ do
        With probability $\epsilon$ select a random action $a_t$
        $a_t \leftarrow$ select action based on $\epsilon$-greedy exploration
        $\text{Set } y_i = \begin{cases} r_i & \text{for terminal } s_{t+1} \\ r_i + \gamma \text{max}_{a_{t+1}} Q(s_{t+1}, a_{t+1}, \theta_i) & \text{for non-terminal } s_{t+1} \end{cases}$
        Perform a gradient descent step on $(y_i - Q(s_t, a_t, \theta))$ with respect to the
        network parameters $\theta$
        Set $s_t \leftarrow s_{t+1}$
    end for
end for
```

In both algorithms, y_i is referred to as the target and is used to perform a gradient descent step in order to update the network parameters.

3.4 Experience Replay

Deep Q Networks can be unstable and experience problems with converging. Also, the algorithms mentioned above are not data efficient. They use every data sample or experience sample only once. Experience replay tackles these problems by saving and reusing samples. It stores each (s, a, s', r)-tuple in a database D and uses these later to update the Q-function. Instead of updating the Q-function based on the most recent transition, we sample uniformly from D, and use these past experiences to perform the gradient descent step. Once every T steps, the data tuples in the database are used to update the parameters of the Q-function. T is referred to as the trajectory length and determines how often the database is accessed for updating. So, during gathering T data tuples, the approximate Q-function and therefore the greedy policy remain constant. There are two ways of updating the Q-function once T data tuples have been gathered and they are shown in Algorithm 2 and Algorithm 3.

Algorithm 3 Experience Replay with Samples

```
for n = 1, ..., N do
    Retrieve a random $(s, a, s', r)$-tuple from D using a uniform distribution
    Update Q-function using the random sample
end for
```

Algorithm 4 Experience Replay with Trajectories

```
for l = 1, ..., L do
    Retrieve a random trajectory from D using a uniform distribution
    for t = 1, T do
        update Q-function using the trajectory sample
    end for
end for
```

Experience replay with samples simply updates the approximate Q-function with N different random samples from the database D. N is usually set to be the number of iterations in an episode. Experience replay with trajectories retrieves a random trajectory from D and then updates the approximate Q-function with the T data tuples in this trajectory. This is done in the same order as how the data tuples were gathered. This process is then repeated L times, where L is usually set to be the number of trajectories in the one episode.

In practice, the database D also has a capacity on how many data tuples it can store. When the capacity is left out this will most likely cause memory issues or in the best case increase the computation time drastically.

4 Experiment

4.1 Evaluation

In order to evaluate and compare the algorithms, multiple metrics will be used. First, the running average over the past 100 episodes will be computed during learning. This gives an idea on how consistent the agent is and if the agent is learning. This running average is computed on the in-game reward score, which will be explained in Section 4.3.1. Furthermore, the largest running average that the agent achieved will be used as the main evaluation metric. Also, this is similar to the main evaluation metric by Open AI Gym for the Carracing-v0 environment. In addition, the standard deviation over the interval of the largest running average will be included. These results will be compared to the results that other studies [19] achieved on this environment to validate them. Also, the total accumulated reward will be computed to see how the methods perform over the whole period of learning.

In the evaluation procedure defined by Open AI Gym, the agent has to race on three different tracks for 100 consecutive races. The average over these 100 races is then computed, together with the standard deviation in order to determine the rank on the leader boards. The goal of this research is not to achieve a high rank in this competition, but to investigate which reinforcement learning algorithms are most viable for driving autonomously through this environment. Therefore, the decision was made to evaluate all algorithms on completely random tracks for each episode instead of three pre-determined tracks. This also makes the task more realistic and gives a better indication on how well the agent is capable of navigating successfully.

4.2 Experimental setup

The final setups of the experiments are shown in Table 1. Experience replay can only be applied to off-policy methods, therefore SARSA is not included with this method. Furthermore, ϵ-greedy exploration is used with experience replay, since it outperformed softmax in the related work.

Table 1. Experiments

Method	Exploration mechanism	ER sampling	ER trajectories
Deep Q-learning	ϵ-greedy exploration	no	no
Deep SARSA	ϵ-greedy exploration	no	no
Deep Q-learning	softmax exploration	no	no
Deep SARSA	softmax exploration	no	no
Deep Q-learning	ϵ-greedy exploration	yes	no
Deep Q-learning	ϵ-greedy exploration	no	yes

The most trivial parameters are shown in the Table 2. All methods that were implemented converged within 1000 episodes, so it seems appropriate to set this parameter to 1000. The discount factor γ was also chosen based on experimentation where a value of 0.99 showed most stable results.

Table 2. Parameter settings

# episodes	γ
1000	0.99

4.2.1 ϵ-greedy

To make this exploration mechanism slightly more sophisticated, the parameter ϵ will decay to zero as the number of episodes goes to ∞. The idea behind this is that as the agent learns more, the need for exploration decreases. Therefore, exploration becomes less important as the number of episodes increases. For the implementation of Deep SARSA and Deep Q-learning the following equation is used to determine ϵ in an episode:

$$\epsilon = 0.02 - (n * 0.00002). \tag{7}$$

For the implementation of deep Q-learning with experience replay the following equation is used:

$$\epsilon = 0.15 - (n * 0.00015). \tag{8}$$

Where n in both equations is the number of episodes. This manner of decaying ϵ ensures that the values for ϵ will decay linearly to zero. Experience replay required a higher ϵ or a higher exploration rate, since it only updates after each T experience samples. So in the first moment of updating, the function approximator is performing terribly which results in a first trajectory that consists of a lot of terrible actions. With a low ϵ, these actions will almost be all the same and this results in updates that only consider this action. In return, this will cause the agent to only select this action for the remainder of the episodes. After experimentation this became clear and a significantly higher exploration rate was needed for consistent positive rewards.

4.2.2 Softmax

For softmax exploration, a similar decaying mechanism was defined for the same reason. A high temperature τ corresponds to more exploration and a low value corresponds to more exploitation. The τ parameter will be decayed in the following manner:

$$\tau = 2 - (n * 0.002). \tag{9}$$

Where n is the number of episode. The value for τ will decay linearly to zero.

4.2.3 Function Approximator

A feed forward neural network is used as a function approximator for all methods. After transforming the high-dimensonal image input to a smaller vector (the state), which will be explained in Section 4.3.3, a neural network is used to approximate the Q-values.

The neural network consists of a fairly simple structure, namely 100 input nodes, a fully connected layer with 512 hidden nodes, and a fully connected output layer with 11 output nodes. The 11 output nodes represent the actions that the agent can take and these will be explained in more detail in Section 4.3.2. The weights are all initialized using the lecun uniform initializer and a ReLu activation function is used between the input layer and the hidden layer. Furthermore, a linear activation function is used between the hidden layer and the output layer in order to have a range of real-valued outputs. The Adam optimizer is used instead of the classical stochastic gradient descent to update the weights of the network. The main reason for using the Adam optimizer is that it is an often used optimizer which achieves good results fast. Finally, the mean squared error is used as the loss function.

4.2.4 Experience Replay

For experience replay there are several parameters that have to be defined. These parameters are shown in the following table:

Table 3. Experience replay parameters

Method	**Capacity**	**T**	**N**	**L**
ER Sampling	10000	50	1000	-
ER Trajectories	10000	50	-	20

In both experiments with experience replay a capacity of 10000 data samples was set on the size of the database. This means that data samples of the past 10 games at most are present in the database. Increasing this capacity resulted in a too large computation time and therefore 10000 seemed appropriate. Furthermore, the trajectory length T was set to 50 in both experiments, which means that the weights of the neural network are updated after every 50 time steps. It is important to find a trajectory length that fits the experimental setting, since this impacts the performance of the algorithm significantly. In this setting, a trajectory length of 50 gave the best results. Next, N was set to 1000 which means that a total of 1000 random samples are drawn from the database and used for updating after every 50 frames or steps. Finally, experience replay with trajectories randomly samples 20 trajectories from the database and updates with all experience samples in these trajectories.

4.3 Implementation

In order to verify the findings of the related literature, multiple frameworks were investigated to implement the methods that are discussed in this section. First, The Open Racing Car Simulator (TORCS) was used, which is a multi platform car racing simulator. It seemed promising, however, all coding has to be done in C++ and I have no experience in this language. I therefore continued searching and found a toolkit for developing and comparing reinforcement learning algorithms, named Open AI Gym. Open AI Gym offers the package 'Gym', which can be used to implement the Carracing-v0 environment in Python. Furthermore, all environments are designed in such a way that only a small amount of time has to be spend on setting up the environments. Rather quickly I was able to start with implementing reinforcement learning techniques.

4.3.1 Carracing-v0

The Carracing-v0 environment, as shown in Figure 4, requires the racing car to navigate successfully over the track, while doing this as fast as possible. The agent has to learn from the image frame input which is the state of the environment and select actions based on this. The state consists of 96 * 96 * 3 pixels, where the 3 accounts for the RGB layers.

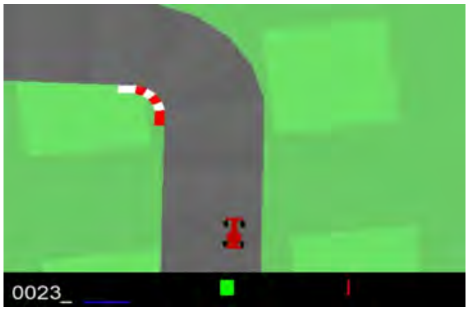

Fig. 4. CarRacing-v0 environment of OpenAI Gym

One episode lasts 1000 frames or until all track tiles are visited. In these 1000 frames or steps that an agent can select an action, the goal is to maximize the total accumulated reward. The reward is -0.1 for every frame as time progresses and +1000/N for every track tile visited. Where N is the total number of tiles in the track. As an example, if you have visited all track tiles in 650 frames, your reward is 1000 - 0.1 * 650 = 935. In the bottom of the window, some indicators are shown, which includes the speed, four ABS sensors, steering wheel position and a gyroscope. At the beginning of every episode a new random track is created, which makes this environment good reflection of reality, where every ride can have a different route.

Normally, to evaluate your agent in this environment, you can choose 3 tracks on which the agent has to race for 100 episodes. The average over these 100 consecutive episodes then becomes the final score. The choice was made to evaluate on 100 consecutive random tracks, since this gives a better reflection of reality.

4.3.2 Discretize action space

In order to transform the continuous action space to a discrete one, 11 possible actions were defined. Four actions are assigned to steering to the left, namely steering to the left with values -1 (fully), -0.75, -0.5, and -0.25. Four actions are assigned to steering to the right with values 1, 0.75, 0.5, and 0.25. Furthermore, one action is doing nothing and one action is assigned to accelerating. Finally, one action is assigned to breaking. To reduce the action space, accelerating and braking cannot be applied at the same time. In addition, steering actions and accelerating/braking cannot be applied at the same time. This makes the driving actions similar to driving actions in a real life racing setting where you also don't accelerate and brake at the same time or accelerate/brake during steering. So, the agent has the possibility to select an action from these 11 actions during each time step.

4.3.3 Preprocessing image input

The image input that consists of 96 * 96 * 3 pixels contains a lot of redundant information for the task of driving through the environment. Therefore multiple preprocessing steps were performed in order to decrease the size of the input. first, The image input was grey-scaled, since the color of the track is irrelevant, which can be seen in Figure 5.

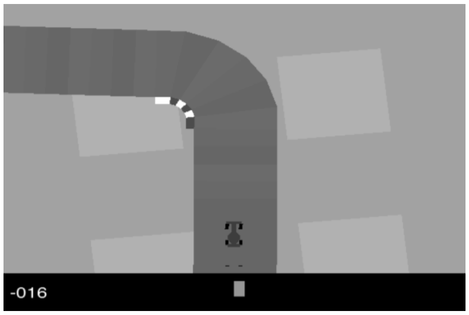

Fig. 5. Gray-scale image representation

The distinction between the track and the grass is still clearly visible. In addition, the curb that indicates the inner side of the turn can be distinguished easily.

This decreases the size of the image input to 96 * 96 pixels. Next, the image is cropped on the sides and the bottom black bar is removed, since this part of the image gives no extra relevant information of the environment. Performing these steps results in a resized image with 84 * 84 pixels.

Finally, the image is resized to even smaller dimensions, resulting in a resized 10 * 10 pixel image. I experimented with the size of the dimensions and found that this reduction gave stable results. Next, the image is transformed to a vector of size 100, which is the input or state during every frame. This input state is used by the neural network to predict the Q-values for all 11 actions.

5 Results

The running averages of the first four experiments are shown in Figure 6. This includes Q-Learning and SARSA with both exploration mechanisms.

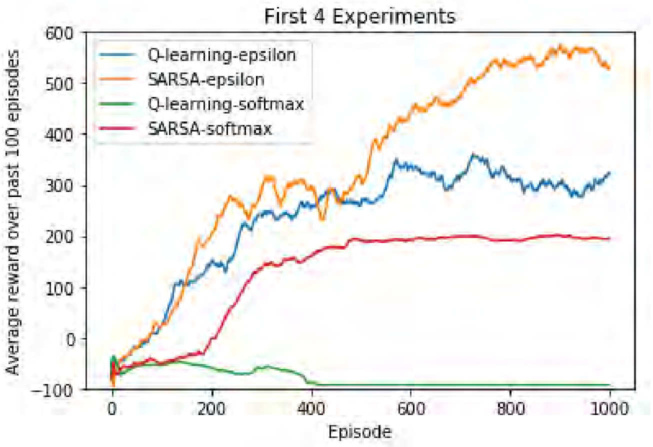

Fig. 6. Running average over 1000 episodes

The running average of Q-learning with ϵ-greedy exploration is shown as a reference in Figure 7. Both experience replay experiments are also shown in this figure.

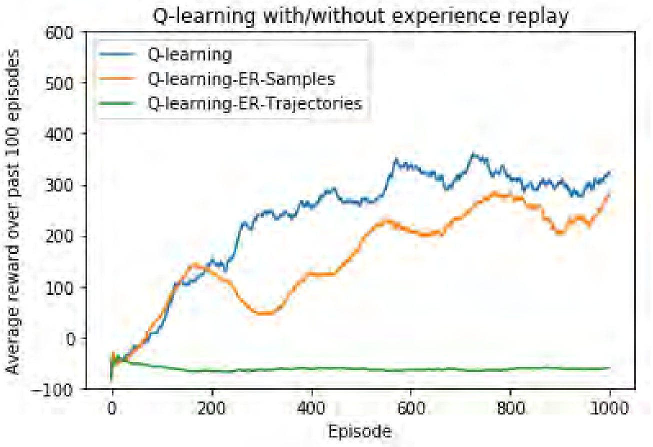

Fig. 7. Running average over 1000 episodes

Furthermore, all other performance metrics are shown in Table 4.

Table 4. Evaluation metrics for all experiments

Method	Max Running avg.	S.D.	total accumulated reward
Deep Q-learning - ϵ-greedy	360.86	195.81	251840.22
Deep SARSA - ϵ-greedy	576.12	201.80	366403.74
Deep Q-learning - softmax	-37.53	22.05	-81864.54
Deep SARSA - softmax	201.09	39.59	140567.60
Deep Q-learning - ER Sampling	284.82	161.03	172835.18
Deep Q-learning - ER Trajectories	-36.39	22.64	-62939.43

Finally, Table 5 is added as a reference. This table contains results that other researches [19] achieved while 'solving' the Carracing-v0 environment

Table 5. Average reward over 100 consecutive episodes achieved in other studies

Method	Average Score
DQN	343 ± 18
DQN + Dropout	893 ± 41
A3C (Continuous)	591 ± 45
A3C (Discrete)	652 ± 10
World Model	906 ± 21
World model with random MDN-RNN	870 ± 120

Note that these results can only be used as an indication, since the evaluation procedure is different from the procedure in this research. The performance of these models is evaluated (after learning) for 100 consecutive episodes on 3 different tracks. During this evaluation, the models are not adjusted. While in this research, the running average over the past 100 episodes is computed during learning. In addition, tracks are randomly generated each episode which makes the task harder, but more realistic. The standard deviation of the other studies can be ignored for these reasons. However, while weighing the different evaluation, the averages can be used to give an indication on how well the algorithms in this research performed compared to the other models.

To give an indication on what an in-game score means in terms of navigation through the environment; the DQN method achieved an average score of 343 and this allowed the agent to navigate through the track in a quivering manner. This method also corresponds to the first experiment in this research (Deep Q-learning - ϵ-greedy). A score of 500 or higher allows the agent to navigate in a more stable manner through the track. Here, the main difference between a score of 500 and 900 is how fast the agent is able to navigate through the track. Also, how often the agent slightly misses corners plays a role for lower scores.

6 Discussion & Conclusion

Figure 6 shows that both deep Q-learning and deep SARSA with ϵ-greedy exploration perform better in terms of running average than both algorithms with softmax exploration. Furthermore, deep SARSA with ϵ-greedy exploration is the best performing algorithm of these experiments. Figure 7 shows that both experience replay experiments do not perform better than deep Q-learning without experience replay. Both experience replay experiments were carried out with ϵ-greedy exploration, since this exploration mechanism performed better in the previous experiments.

Table 4 supports these findings, where deep SARSA achieved the highest maximum running average and the highest total accumulated reward. The standard deviations of most experiments are relatively high compared to those in Table 5. This makes sense, since the running average is computed during learning in this research. In other words, the weights of the function approximator are adjusted during all episodes, while this is not the case for the methods in Table 5. Also, the agent is still exploring actions during all episodes. These reasons account for bigger fluctuations in the score per episode and therefore the standard deviation. Table 5 also shows that deep SARSA with ϵ-greedy exploration achieves almost the same running average as the A3C (continuous method) which uses a more sophisticated function approximator [20].

From these findings, it can be concluded that deep SARSA with ϵ-greedy exploration is best suited for the modelling/learning task in the Carracing-v0 environment.

Furthermore, the reason that softmax exploration performs a lot worse in combination with Q-learning and SARSA is mainly caused by the difficulty of tuning the temperature parameter. Related literature [13] showed that the temperature parameter can be hard to tune and it was found to be true for this environment. Various decaying methods were used, however, only the current method (Equation 9) gave stable results for deep SARSA. From this, it can be concluded that softmax exploration might perform better if the temperature parameter τ can be tuned correctly. However, this might take significantly more time than a relatively simple implementation of ϵ-greedy exploration and a better performance is not guaranteed. Therefore, ϵ-greedy exploration is preferred for this environment.

Finally, experience replay did not improve the performance of the deep Q-learning algorithm. This is most likely caused by the simple function approximator that is used. As stated in Section 1, the focus of this research is the modelling/learning task and not the recognition task. Therefore, the choice was made to use a simple feed forward neural network as the function approximator. This would save time and allow for comparison of the algorithms used for the modelling/learning task. Experience replay is often used for situations where a complex and multi-layered neural network is used as a function approximator.

This would help make the algorithm more stable and decrease the time until convergence. Using the feed forward neural network already resulted in a stable algorithm. In addition, convergence was already achieved by deep Q-learning without experience replay. These reasons contributed to the worse performance of using experience replay with deep Q-learning. Experience replay is therefore preferred when the the performance does not converge or when a complex function approximator is used.

6.1 Limitations

Time restricted this research to some extent, as with any other research. Tuning the parameters of the algorithms was affected most by the limited time available for this research paper. Although all experiments have been executed in time that was available, not all experiments resulted in a positive in-game score. Specifically, softmax exploration requires significantly more tuning of the temperature parameter. So, extra time could have resulted in a better performance of the experiments with softmax exploration.

As mentioned before, the focus of this research is the modelling/learning task and a simple feed forward neural network was therefore used as a function approximator. This also resulted from the limited time available for this research paper. This restricted the research to some extent as well, mainly because the performance of all experiments will likely increase with a better function approximator. Specifically, the experiments that included experience replay would probably have shown more interesting results.

Another aspect arises from the limited available time and the focus of this research. From the recordings of the deep SARSA algorithm with ϵ-greedy exploration it stands out that the agent can successfully drive through the track, however, the agent loses control at a turn when its speed is too high. This is caused by the fact that the agent can not determine its speed from the current state. To solve this, an improvement could be made in the preprocessing steps of the image input. As mentioned before, the bottom of the in-game window consists of some indicators such as the speed. In the preprocessing steps, this bar is removed to reduce the dimensions of the image input. If this speed indicator can be included in the state information, this would help the agent with avoiding to lose control. This is part of the recognition task and was therefore not included in this research, however, future research should definitely consider this information.

References

1. Stentz A. Thorpe C. Moravec H. Whittaker W. Kanade T. Wallace, R. First results in robot road-following. 7, 1985.
2. A. Broggi. Proud car test 2013. *http://vislab.it/proud/*, 2013.
3. Cho K. Zhang, J. Query-efficient imitation learning for end-to-end autonomous driving. 12, 2016.
4. Kendall A. Cipolla R. Badrinarayanan, V. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. 14, 2015.
5. Sutskever I. Hinton G. E. Krizhevsky, A. Imagenet classification with deep convolutional neural networks. advances in neural information processing systems. 9:(1097–1105), 2012.
6. Wang T. Tandon S. Kiske J. Song W. Pazhayampallil J. et al. Huval, B. An empirical evaluation of deep learning on highway driving. 7, 2015.
7. Engel P. M. Basso, E. W. Reinforcement learning in non-stationary continuous time and space scenarios. 10.
8. `https://gym.openai.com/envs/CarRacing-v0/`.
9. R. S. Sutton. Learning to predict by the methods of temporal differences. 36, 1988.
10. Palm G. Tokic, M. Value-difference based exploration: Adaptive control between epsilon-greedy and softmax. 12, 2011.
11. Busoniu L. Babuska R. Adam, S. Experience replay for real-time reinforcement learning control. 12, 2011.
12. Mnih et al. Playing atari with deep reinforcement learning. 9, 2017.
13. Drugan M.M. Wiering M.A. Tijsma, A.D. Comparing exploration strategies for q-learning in random stochastic mazes. 8, 2016.
14. Abdou M. Perot E. Yogamani S. El Sallab, A. End-to-end deep reinforcement learning for lane keeping assist. 9, 2016.
15. Mnih et al. Asynchronous methods for deep reinforcement learning. 10, 2016.
16. Roy D. Srishankar N. Kondapalli, C. Deep reinforcement learning applied to a racing game. 6, 2017.
17. Guez A. Silver D. van Hasselt, H. Deep reinforcement learning with double q-learning. 7, 2015.
18. Barto A. G. Sutton, R. S. *Reinforcement learning: An introduction.* 2nd edition, 2016.
19. Stanley K.O. Risi, S. Deep neuroevolution of recurrent and discrete world models. 7, 2019.
20. Min J. Kim J.H. Jang, S.W. Reinforcement car racing with a3c. 8, 2017.