Reinforcement Learning - Final Course Project

Yarden Kuperberg (ID. 311155691), Jonathan Ben Dov (ID. 311525448)

Submitted as final project report for the RL course, 2022

1 Introduction

The problem of training an agent to drive a car is widely discussed and regarded as a breakthrough technology. The goal is to train an agent in a simulated environment to understand which actions to take in order to drive correctly and minimize the risk of an accident. The complexity rises from the effect of each action taken on the future actions, for instance a car that chooses to increase speed when it is behind another car.

Our goal and motivation is to use the current virtual environments to train an agent using as simple algorithms as possible - while still using as much of the state of the art algorithms published already as possible.

The advancements in the field of deep learning in recent years have increased the use of theoretical reinforcement learning using deep learning to solve such problems. The most well known examples are the results published by Google's Deep-mind, who published the original DQN paper (Mnih et al., 2015)[1], the famous advancement in training an agent in the game of Go (Silver et al., 2016) and further on (2016))[2] as well as architectural improvements that later came on top of these algorithms such as double DQN (van Hasselt et al., 2016)[3] as well as asynchronous training methods(Mnih et al., 2016)[4] and actor critic methods.

Since training an agent to drive a car is a dangerous environment to do so in the real world, we need to use virtual environments in order to train our agent. We use open-AI's highway-env framework, which simulates a car driving in several environments. We use several known algorithms, as well as changes in them which we found to be interesting and show our results accordingly.

The environment of training the agent consists of a series of images, our **observation space**, of size 4*128*128 which describe the road in which our agent is driving and the three previous steps. Our agent (which is the term we use to describe the driver of the car we are training) is trained in a discrete action space - he has in general (when not in the far left or far right lanes) 5 different options which are out **action space**: increase speed, decrease speed,

[1] https://www.nature.com/articles/nature14236

[2] https://www.nature.com/articles/nature16961

[3] https://proceedings.neurips.cc/paper/2010/file/091d584fced301b442654dd8c23b3fc9-Paper.pdf

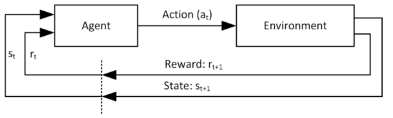
[4] https://www.nature.com/articles/nature16961

go up one lane, go down one lane and stay in the current lane. The environment has other cars driving in them, and an episode (or 'drive') ends when the car crashes into another car. The goal we are optimizing for is for our driver to drive as fast as possible for as many steps as possible without crashing his cars.

The environment used has a reward function as follows:

$$R(s,a) = a\frac{v - v_{\min}}{v_{\max} - v_{\min}} - b\,\text{collision}$$

Where v is the speed the agent at the observation, and a and b are hyper parameters. In order to estimate how well an agent does, it is necessary to increase the reward he gets for every action taken, but also maximize the future rewards the driver can take - i.e. try to see how taking the current action and its reward will affect future rewards. For example, sometimes the driver will prefer to slow down instead of speed up in order to maximize future rewards.



[5]

In part 1, we will show the current best known related work to solve this issue and best known benchmarks. Part 2 will show our general approaches, as well as an in depth description of the algorithms chosen and the experiments we did to change them, as well as the general technical details of the architectures (hyper-parameters, training methods, frameworks and more). Part 3 will show our experimental results, as well as the trade-offs of each result. Finally, we will discuss our results and provide some thoughts for future research.

1.1 Related Works

One of the most well known approaches to teaching an agent to work in a discrete environment is that of the Deepmind article of playing Atari V (Mnih et al,)[6], the algorithm DQN was presented in this article and shown to outperform all previous methods. In (Xiaoyun et al, 2018)[7], the authers describe a DDQN model to solve another discrete environment problem, which is shown to perform better than DDQN. Finally, in (Schaul et al., 2016)[8], another Deepmind paper, the authors devise the idea of prioritized experience replay while performing the training in DQN, and show that this lowers the amount of training steps required and the amount of overfitting.

[5] Presentation 8 slide 63
[6] https://www.cs.toronto.edu/ vmnih/docs/dqn.pdf
[7] https://www.hindawi.com/journals/jr/2018/5781591/
[8] https://arxiv.org/abs/1511.05952?context=cs

2 Solution

2.1 General approach

The type of learning we will use during the training of the agent is **off-policy**, which updates its value function according to data that is obtained from a different policy (or control policy), since we update the control policy (in our case our target network) every few iterations, but still use the historic data which is obtained from the previous policy. This is unlike **on-policy** training, which uses the same value function (or network in our case) to both predict and learn with.

Furthermore, the approaches we used are **model free** - meaning the agent does not have access to a general action-value function of the environment, i.e. a function that can predict ahead the value of each action, unlike **model based** approaches which can predict the reward of each step since they already have one in the firsthand.

Our approach's basics is as follows: For the current observation done, we will calculate the Q-value function result of the its corresponding action and state. We select an action according to a **soft epsilon** approach - in which we the highest Q value is given a chance of 1-epsilon to actually be executed, this is done in order to add an exploration factor to the Agent.

After every episode, we increase the size of our epsilon, in order to add more randomness toward every progress made in the match. This method is called **epsilon decay**.

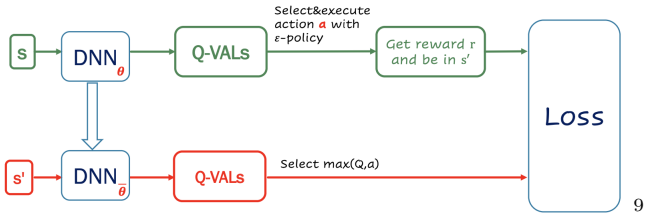
Furthermore, in every few hundred steps taken in an episode, because the amount of data we store is too large, we delete the list of historic observations and begin a new list, but we keep the last actions in every episode, who are the ones that lead to the crash.

As stated, we used three different environments, and in our first environment, we added **stochasticness**. This means that every action that is chosen by our algorithm has a p probability of being chosen, and the rest of the actions that were not chosen have a $\frac{1-p}{n-1}$ chance of being chosen, which is another way of adding stochastic randomness to the action space and thus add to the agents exploration vs. his exploitation of current best actions to take.

The first main algorithms used for our research is **DQN** - This is an algorithm based on (Mnih et al., 2015), in which our agent is trained in the following matter: We initialize two networks - a **target network** and a **Q network**, which we will optimize accordingly.

We maintain a list of the following values in every iteration: the action taken, the reward for this action, the predicted q values for each action (per state), the state and the result state after the action. In each iteration, we will predict the action to be taken by our Q network and perform it, and calculate the difference of the Q value of the action taken with the maximum reward with the Q network with the Q value of the next action with the target network added to its reward, with the hyper parameter discount factor. After a certain

amount of iterations, we will update the target network's weights to be equal to the Q network's weights.



[9]

The second main algorithm we used is **double Q learning - DDQN** - based upon the research presented at (van Hasselt et al., 2016), which presented the problem that the Q values at the beginning are very prone to beginning noisy, and thus the choice of an action with maximum value is not ensured to be the best choice. This leads to learning that is very complicated. The solution presented is to have two different action value functions - Q and Q', in which Q' is by the target network and Q is from the Q network. We find the index of the highest Q-value from the Q model and use that index to obtain the action from the target value as we can see from the figure:

$$y_j = \begin{cases} r & \text{if } s' \text{ is terminal} \\ r + \gamma Q(s', a^{\max}(s'; \theta); \theta^-), & \text{otherwise.} \end{cases}$$

An addition added later on is through the usage of **experience replay** - the choice of the previous mini-batches - In which at every stage we are learning, to choose randomly a mini batch of iterations done, and optimize the network to back-propagate the loss from the differences between the two.

The idea behind this method is that some experiences are correlated to the iterations after them, and if we keep choosing and iterating over them together we will overfit our network on them. By sampling randomly, we smooth the training distribution.

An addition we added to this is the usage of **prioritized experience replay** - in which we give priorities to each iteration according to the difference between the predicted q value and the target network, and provide weights in which to sample them from - an iteration with higher difference will have a higher probability of being sampled in the batch. The idea behind this is that some experiences are more important than others and should be optimized upon more often.

Level of prioritization

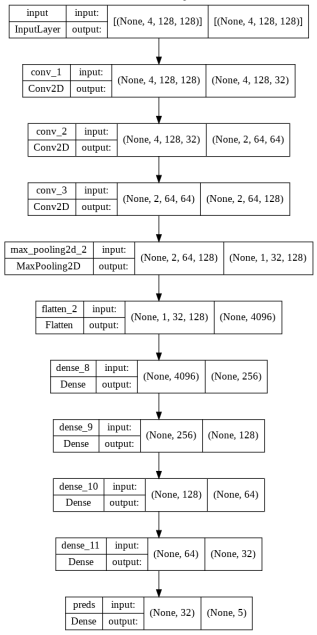
$$P(i) = \frac{p_i^{\alpha}}{\sum_k p_k^{\alpha}}$$

kupi reward optimize

[9] Presentation 8 slide 63

2.2 Design

The network architecture we used for this research is as follows: We pass the batch size x 4 x 128 x 128 images (number of batch amount of 4 images, each image of size 128 x 128) to 3 convolutional layers, after which we added a max pooling regularization layer, flatten and add 4 dense layers, and finally a prediction linear layer.



We used two different loss functions - **mean square error** loss which is widely used in regression tasks, which maximizes the squared differences between the predicted and the target outputs. The problem with this approach is that it gives a high value to edge cases in which the difference is large. Because we are predicting our own output, this means that huge changes in the values will cause huge changes in the network as well. Thus, we also used **Huber loss**, which uses MSE for low values, and mean absolute error for large values, which treats large errors and small errors equally.

$$Huber(a) = \begin{cases} \frac{1}{2}a^2 & \text{for } |a| \leq 1, \\ (|a| - \frac{1}{2}), & \text{otherwise.} \end{cases}$$

Lastly, for the third environment, we tried two different ways: firstly to train one network, and to move between environments every few episodes and train the network.

The Second method involved a large master network, and a network for each environment, and at every environment to train the specific network. After every episode, we trained the main network using the gradients of all the other networks. This method is similar to multi threaded training which was presented in class.

3 Experimental results

Our main metrics of success for our experiments were model loss per total steps taken, the amount of total steps taken per episode, and a metrics we developed, which was as follows:

$$(EpisodeRewards)^2 + log(StepsTaken)$$

We want to give high weight to the reward, but take into account the steps the agent did during that episode.

3.1 Environment 1

The First environment was an environment with regular drivers and a discrete stage with maximum 5 different possibilities. The results are as follows:

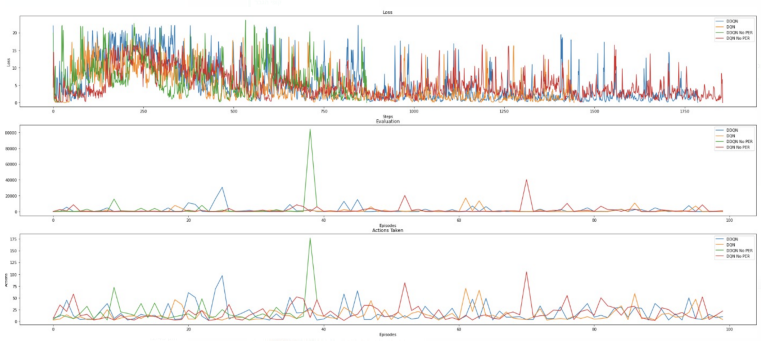
We observed that with our experimentation on environment 1, the DQN model's loss reduced fastest and was minimal amongst the models, and that on average the number of steps it took was highest. This surprised us, but can be explained due to the fact that DQN is prone to overfit on the value function more than DDQN. Furthermore, We saw that using prioritized experience made an exceptional difference while using DDQN, and a relatively small difference in DQN in terms of loss and actions taken. We can see that the DDQN did more steps absolutely among the models and has a higher loss, but seems does more steps.

We can see that our DQN models seems to converge faster model at around 900 steps, compared to the DDQN who diverges after around 1100 steps. This can be explained by the fact that we are now optimizing two value functions, Q and Q', and thus training may be less noisy at first, but will take longer to converge.

To finalize - in environment 1 DQN converges faster and has a lower loss, but DDQN has a higher amount of total actions the models did.

3.2 Environment 2

The second environment included more 'aggressive' drivers and a faster environment, and its results are as follows:

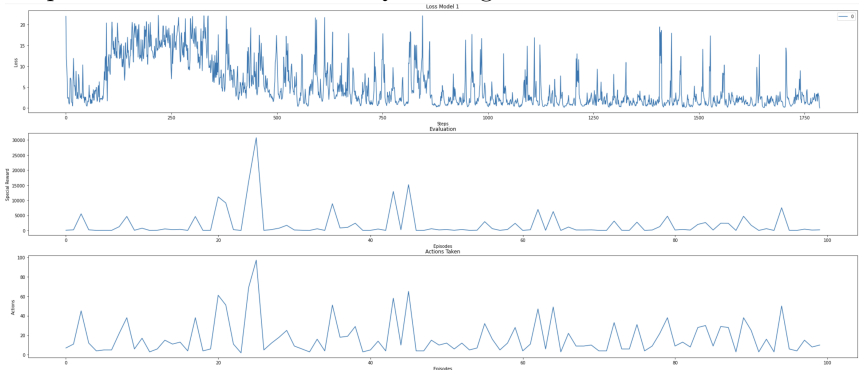


We can see that yet again the DDQN without prioritized experience gave pretty poor results in terms of actions done, but eventually converges. In comparison, it seems that in all metrics (loss, actions done and our metric) the DQN with no prioritized experience seems to be performing best.

Yet again, the DDQN with PER gave very high scores which were close to the highest performer.

3.3 Environment 3

In this environment,for the second method we tried, there is a problem maintaining a time series of loss for every model separately (as each model has a different rate and it is not one single function over time), and thus we presented the results for our first method. We can see that convergence arrives at around 900 steps which is similar to the DQN average.



4 Discussion

We tested a series of combinations with the two main models we chose. We can see that generally, the best architecture in terms of actions performed, loss and our metric, using DDQN with prioritized experience replay and Huber loss was optimal.

This conforms to the research done so far, but it is interesting to see that the regular DQN still outperforms in several scenarios.

It would be interesting to research more advanced Neural Network architectures such as vision transformers, and see the effect on these algorithms, as well as see other papers published implemented.

5 Code

Github:

https://github.com/jonathanbd135/reinforcement_learning_final