RL Homework assignment 2

Joost 't Hart (s1691570) and Joshiwa van Marrewijk (164491)

2 April 2020

1 Introduction

In this homework assignment, we want to build an AI based on function approximations that can overcome the nearly exponential growth of the search parameters when increasing the search depth, as shown in the previous assignment.

In this assignment (Plaat, 2021), we need to implement three function approximation algorithms to try and overcome the limitations of tree-search algorithms. The first being a Tabular Q-learning method, the second a Deep Q-learning method, and the third a Monte-Carlo Policy Gradient (REINFORCE) algorithm.

In this project, we work with the CartPole environment from OpenAI Gym (Brockman et al., 2016; Barto et al., 1983). This environment consists of a pole attached by an un-actuated joint to a cart, which moves along a frictionless track. An agent can apply a force of +1 or -1 to the cart. The goal for the agent is to prevent the pole from falling. Every time step that the pole remains upright the AI gets a reward. The episode is terminated when the maximum reward is reached, when the pole moves 15 degrees from horizontal or when the cart moves more than 2.4 units from the center. The observations provided by the environment are the position of the cart, the velocity of the cart, the angle of the pole, and the angular velocity of the pole. All models are trained in the V0 environment. This environment has a maximum reward of 200. This environment defines "solving" as getting a reward of at least 195 for 100 consecutive episodes. The environment is shown in Figure 1

The rest of the assignment is structured as follows. First, we provide a theoretical background and experimental set-up (Chapter 2) of each implementation such that necessary information is given to fully understand the results (Chapter 3) of each method, respectively. The assignment concludes with a general discussion (Chapter 4), a conclusion that summarizes our findings (Chapter 5), and a section that describes where our code can be found (Chapter 6).

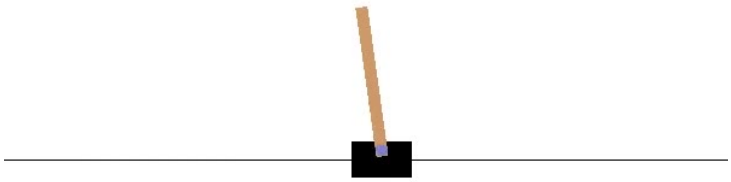


Figure 1: An Illustration of the CartPole environment. This Figure is adopted from AG Barto & Anderson (2021).

2 Theory

The main purpose of this section is to introduce an explanation of the techniques used to implement all three algorithms. Per method, it describes the mathematical descriptions we implemented in PYTHON and a brief description of its meaning, derivations, and resemblance with the other methods. Furthermore, we provided explanations on particular design choices (such as discretizing the environment in section 2.1).

2.1 Tabular Q-Learning

We start with a naive approach using a Q-value table. During the learning phase, the Q-values for different states will be stored in a look up table. The CartPole environment has a continuous observation space. To store the observed states in a lookup table, the observation space has to be discretized. During simulations, the model will lookup the current state in the table and select the move with the highest Q-value. Discretization is a subtle task. For instance, if the bins are too small, the learning will never converge. On the other hand, if the bins are too large, the model is not able to distinguish different states. The discretization of the observation space is done using linearly spaced bins with a fixed with. More details on the discretization can be found in Chapter 4 and Table 1.

The Q-value for a given observed state is calculated using the Bellman's equation (Plaat, 2020), which is given by:

$$Q(s,a) = (1-\lambda)Q(s,a) + \lambda\left(r + \gamma \max\left[Q(s',a')\right]\right), \tag{1}$$

where $Q(s,a)$ is the Q-value for action a at state s, r is the reward for taking action a at state s, λ is a learning rate, γ a discount factor and $\max\left[Q(s',a')\right]$ is the maximum Q-value of the next state. At the start of training, all actions are equally likely. Using equation 1, the Q values of all states in the table are attractively updated. Two important parameters in equation 1 are λ and γ. The learning rate, λ, controls how much the Q-values are updated by new observations. The discount factor, γ, discounts future rewards.

As in most reinforcement, the exploration-exploitation trade-off has to be balanced for the training of the model (Plaat, 2020). Here, the ϵ-greedy approach is used. At the end of each simulation, ϵ is multiplied by a factor δ_ϵ, which is smaller than 1, to increase exploitation and decrease exploration.

Several hyperparameters can be optimized in the Tabular Q-learning algorithm. When implementing the model, ad-hoc testing is performed, to get quasi-optimal hyperparameters. After that, a grid search is performed on the parameters: ϵ, the minimum ϵ, δ_ϵ, λ, γ and the four bin sizes for the discretization. Grid search is a very inefficient way of tuning hyperparameters. Therefore, we are not able to estimate the optimal hyperparameters at a high resolution.

2.2 Deep Q-learning

Next to Tabular Q-learning, Deep Q-learning (DQL) has been implemented to solve the CartPole problem. DQL overcomes the discretization problem of the TQL algorithm by approximating the Q-value function using a Neural Network (NN). This NN effectively functions as a look-up table (i.e. it has as input a state and as output the Q-values), but, it takes the continuous states as input.

The NN has to be trained supervised. Therefore, an estimate of the true Q-value is needed, for every state-action combination. This estimate of the true Q-value is calculated using a modified version of Bellman's equation:

$$\hat{Q}(s,a) = r + \gamma \max\left[Q(s',a')\right], \tag{2}$$

which is equivalent to equation 1 with $\lambda = 1$. If a state is a terminal state, the second term is set to zero, since there is no next move, and $\hat{Q}(s,a)$ is simply equal to r.

While simulating the environment, consecutive states are strongly correlated. One solution the has been implemented to solve this problem is an experience buffer. Every observed state is saved in a buffer during training. when the buffer has reached a minimal size, a sample of n observations are sampled from the buffer. The NN is trained for one epoch on this batch of data. The buffer has a finite maximal size. When the buffer is full and a new state has to be added, the oldest state in the buffer will be removed. The maximal and minimal buffer size is set to 10^3 and 10^6, respectively, for all experiments

(Zhang & Sutton, 2017). Since DQL uses ϵ-greedy exploration, the buffer will also have on- and off-policy realizations.

Before the training starts, the experience buffer is filled with random realizations, which we refer to as the burn-in phase. ϵ will not be updated during the burn-in phase. After this burn-in phase, at every step, during every episode, one batch is taken from the buffer and the NN is trained for one epoch on this batch. Different implementations of training with an experience buffer exist. E.g. the model could also be trained after every episode, instead of during the episodes. Here has been chosen for the latter. Using this implementation, all observed stated have a high probability of ending up in a training batch. When the model is trained after each episode, a lot of new experience is added to the buffer, which results in more experiences being removed. However, the implementation used here is less computationally efficient.

A second improvement of the DQL algorithm is infrequent weight updates. This method works by using two separate NNs, one network is used to update the Q-value, and one to calculate the targets of the update of the Q-value. Every x epochs, the Q-value network weights are copied to the target network. The target network increases stability during the training.

The NNs used in the experiments are (deep) feed forward networks. The networks are fully connected and contain at least an input layer, one hidden layer, and an output layer. Before training, the neurons are randomly initialized according to a normal distribution with a mean of 0 and a standard deviation of 0.05. The activation function of the hidden layers is the Rectified Linear Unit (ReLu) for all experiments. The loss function that is optimized by the NN is the mean squared error (MSE).

The hyperparameters that are investigated in this project are the number of hidden nodes (which includes a varying number of layers), the learning rate of the NN, the discount factor, γ, the update frequency of the target model, and the batch size for training the model.

2.3 Monte-Carlo Policy Gradients

For the third and final evaluation, we implemented a Monte-Carlo Policy Gradient (also called REINFORCE) method. Similar to the DQL, the REINFORCE uses a continuous input environment. However, there are major differences between the two methods.

REINFORCE, uses an explicit policy method while the previously discussed methods are implicit. An explicit policy approaches its action taking process as a probability distribution ($\pi_\theta(s) = p_\theta(a|s)$, with π_θ the policy) over the possible actions by predicting the parameters (θ) that describe this distribution (Moerland, 2021). Therefore, one can directly sample an action given the current state from this distribution when we want to go the next while maximizing the reward.

These explicit policies are different from implicit policies, as those policies are not defined by characteristic parameters, but are focused on finding a maximum Q-value given a certain state. Thus, implicit policies are for instance the ϵ-greedy action. Implicit policies are fast for discrete spaces as one can simply look up a maximum Q-value in their table (Moerland (2021), and section 2.1) but become computationally inefficient for continuous action space. Then, the greedy policy becomes an optimization problem that can be time-consuming to compute or needs to be approximated with for instance a NN (section 2.2).

Thus, with implicit policies the AI, intuitively speaking, learns the parameters that describe this probability distribution to make the proper action that maximizes the total reward. Thereby, it is a policy-based reinforcement learning method instead of the value-based approaches described in the previous methods.

The learning process of the REINFORCE algorithm is as follows. given a trace

$$h_t = \begin{Bmatrix} s_t, & s_{t+1}, & \dots, & s_{t+n}, & s_{t+n+1} \\ a_t, & a_{t+1}, & \dots, & a_{t+n}, & a_{t+n+1} \\ r_t, & r_{t+1}, & \dots, & r_{t+n}, & r_{t+n+1} \end{Bmatrix}, \tag{3}$$

with $s, a,$ and r the state, action, and reward at episode t, respectively and n the maximum trace length (Plaat, 2021), we can get a total cumulative reward by

$$R = \sum_{i=0}^{n} \gamma^i r_{t+i}, \tag{4}$$

with γ the discount factor. Therefore, at a state s_t we want to maximize the expected total reward, written as (Plaat, 2021; Moerland, 2021):

$$J(\theta) = E_{h_0 \sim p_\theta(h_0)} \left[R(h_0) \right], \tag{5}$$

with E the expectation symbol.

Then, using a gradient descent approach, the policy, $p_\theta(a|s))$ that minimizes $-J(\theta)$ is estimated. In this implementation (the reason why the algorithm is called a Monte-Carlo), we run a trace (comparable to a roll-out from the previous assignment) until it terminates (noted as $n \to \infty$) by sampling for each run a move from the policy. Such an algorithm is called a stochastic method.

To learn the policy that maximizes the reward with a gradient descent approach, we need to obtain the gradient $\nabla_\theta J(\theta)$. This gradient (a gradient over an expectation) in Reinforcement learning is known as the REINFORCE estimator and written as

$$\nabla_\theta J(\theta) = \nabla_\theta E_{h_0 \sim p_\theta(h_0)} \left[R(h_0) \right] = E_{h_0 \sim p_\theta(h_0)} \left[\sum_{t=0}^{n} R(h_t) \nabla_\theta \log p_\theta(a_t|s_t) \right], \tag{6}$$

with $p_\theta(a|s)$ the policy and a single trace. The full derivation can be found in Moerland (2021). Then the expectation is obtained by averaging over M traces. Therefore, the policy gradient becomes (Moerland, 2021):

$$\nabla_\theta J(\theta) \approx \frac{1}{M} \sum_{i=1}^{M} \left[\sum_{t=0}^{n} R(h_t^i) \nabla_\theta \log p_\theta(a_t|s_t) \right]. \tag{7}$$

Luckily, this gradient can easily be implemented with automatic differentiation software such as Tensorflow. We update the policy after each trace is terminated (after a cumulative reward of 200, or a failed attempt) and take a running mean over 100 traces. Therefore, the loss function (per trace) used for the automatic differentiation software reduces to the part within the brackets:

$$L \equiv -\sum_{t=0}^{n} R(h_t^i) \log p_\theta(a_t|s_t), \tag{8}$$

which is known as the categorical cross-entropy. This function is used to obtain the maximum likelihood estimation, by minimizing the cross-entropy between the data distribution, $R(h_t^i)$, and the model distribution, $p_\theta(a_t|s_t)$ (Moerland, 2021).

At last, we need to discuss how we implemented the parameterization of the policy. As described in the introduction (section 1), the policy outputs a prediction on the likelihood of the cart needing to move left or right to increase its reward. Then, we assume that the parameters describing this distribution can be approximated with a neural network in which we used a Softmax activation function in the output layer. With the Softmax activation in the output layer, the output of the NN becomes a probability of taking an action. Thus training this network results in learning the proper policy not learning the maximum Q-value given a state s.

Also, the NN used in this experiment is a (deep) feed forward network and is fully connected. The initialization and activation function of the input and hidden layers of this network is similar to that of the DQN. The hyperparameters for this model are the discount factor γ, the learning rate (implemented in the gradient descent step) λ, the number of hidden layers, and the implementation of reward normalization which is done by subtracting the mean of the discounted returns in the trace in Eq. 4 and divide the residue by its standard deviation. This is supposed to improve the training stability of the network (Moerland, 2021).

3 Results

3.1 Tabular Q-Learning

The implemented TQL model is optimized using a simple grid-search. Models with different hyperparameter combinations are trained for 2×10^4 episodes. Before defining the grid, different sets of

hyperparameters are tested ad-hoc to find out what the relevant part of hyperparameter space to explore. The problem with grid-search is that it is very computationally expensive. Therefore, only two values have been tested for all hyperparameters. Table 1 shows the tested hyperparameters. This grid contains 512 different combinations of hyperparameters.

Several combinations of hyperparameters were able to "solve" the CartPole environment (i.e. reaching a reward of at least 195 for 100 consecutive episodes). The best model is defined as the model that has the most rolling mean rewards of 200 for a window size of 100 episodes. The hyperparameters yielding the best model are indicated in bold font in Table 1. The rolling mean of the training reward of the best performing model is shown in figure 2a.

To further test the best-trained model, the model is tested on the CartPole-v1 environment. The difference between the v0 and v1 environment is that the maximal achievable reward in the v1 environment is 500, instead of 200 in the v0 environment. The best performing model is applied to the CartPole-V1 environment for 100 episodes. From these 100 episodes, the mean and the standard deviation are calculated. The result of this test is shown in Table 3. Noteworthy about this test is that the minimal reward of all episodes is 232, which is higher than the maximum reward of the CartPole-v0 environment.

Table 1: Evaluated Hyperparameters for the Tabular Q-Learning method. Variables are as in equation 1.

Hyperparameters	Values
Initial Exploitation Fraction (ϵ)	[0.5, **1**]
Minimal Initial Exploitation Fraction	[$\mathbf{10^{-2}}$, 0.5]
Exploitation Fraction Decay (δ_ϵ)	[0.99, **0.999**]
Learning Rate (λ)	[10^{-2}, $\mathbf{10^{-1}}$]
Discount Factor (γ)	[0.9, **0.99**]
Cart Position Grid Size	[**0.48**, 0.048]
Cart Velocity Grid Size	[**0.4**, 0.2]
Pole Angle Grid Size	[0.0418, **0.0209**]
Pole Angular Velocity Grid Size	[**0.4**, 0.2]

Note: *The hyperparameters in bold font are the hyperparameters yielding the best results.*

3.2 Deep Q-Learning

Just as for the TQL model, a grid-search is performed to optimize the hyperparameters of the DQL model. Models with different hyperparameter combinations are trained for 150 episodes. Note, that the NN is trained after every move during the episodes, resulting in more training cycles than episodes. Again, the resolution of the grid-search is low due to the computational intensity of the training. The grid is given in Table 2. This grid contains 648 different combinations of hyperparameters. The most important hyperparameter is the number of hidden nodes. $\{i\}$ means that the model contains 4 input nodes i hidden nodes and 2 output nodes. $\{i, j\}$ is a model with 4 input nodes and two hidden layers, with i and j nodes, and 2 output nodes. A target model update frequency of i indicates that every i episode the target model is updated with the weights of the training model.

The models are only trained for 150 episodes due to the computational expense of the training. The NN is trained after every action. Therefore, training the DQL model for one episode takes significantly longer than training the TQL model for one episode. The best DQL model is defined as the model that reached the most rolling mean rewards of 200 for a window size of 10. Again, the hyperparameters yielding the best results are given shown in bold font in Table 2 and the rolling mean of the training reward of the best performing model is shown in figure 2b.

With the above experiments, the models are not yet tested to solve the CartPole environment. Therefore, similar to the TQL model, the best performing DQL network is simulated for 100 episodes on the CartPole-v0 environment. The performance of the best model is always perfect in the v0 environment. The best model is also tested on the v1 environment, with 500 steps. This test also resulted in a perfect score for all episodes (see Table 3). To further test the model performance, it is tested for a maximum reward of 10^6. Again, the model reached a perfect score on every iteration.

Table 2: Evaluated Hyperparameters for the Deep Q-Learning method.

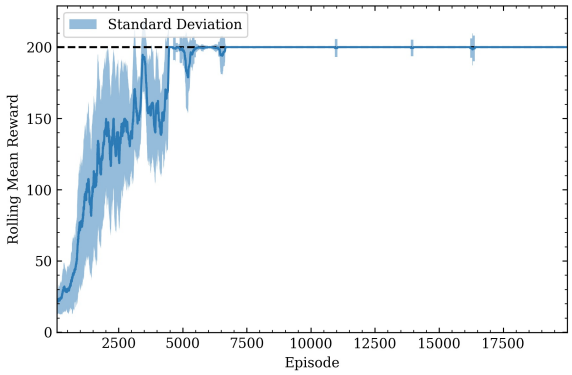
Hyperparameters	Values
Number of Hidden Nodes	[{4}, {8}, {4,4}, {8,8}, **{64}**, {64, 64}]
Learning Rate	[10^{-3}, $\mathbf{10^{-2}}$, 10^{-1}]
Discount Factor	[**0.9**, 0.95 0.99]
Target Model Update Frequency	[**1**, 5, 10, 15]
Batch Size	[**32**, 64, 128]

Note: *The hyperparameters in bold font are the hyperparameters yielding the best results.*

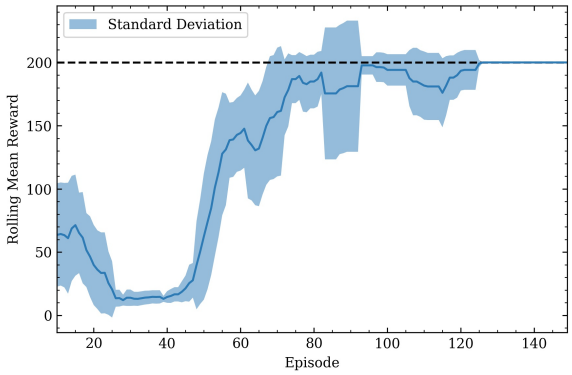
Table 3: Performance of best Tabular Q-Learning and Deep Q-Learning models on Cartpole-v1 environment. The models are simulated 100 times. The used model hyperparameters are indicated in bold font tables 1 and 2b.

Model	Mean Reward	Standard Deviation
TQL	411	101
DQL	500	$0^{\dagger}$

Notes: † *All 100 runs with the DQL model yielded a reward of 500.*



(a) The learning process for the best performing Tabular Q-learning method



(b) The learning process for the best performing Deep Q-learning method

3.3 Monte-Carlo Policy Gradients

Also, for the Monte-Carlo Policy Gradients (MCPGs) we implemented a grid search, with hyperparameters (discussed in section 2.3) shown in Table 4. Each training was done for 2000 episodes in the CartPole-v0 environment, in order to test the mean reward and the stability of the model. As the implementation is stochastic, we expect the stability of the MCPG to be worse than that of the value-based implementations. In Table 4 we selected in bold the hyperparameters that obtained the most rolling means rewards of 200. However, in 2000 episodes, we observed a large variance of the running cumulative reward for this policy model. Therefore, the hyperparameter combination with the highest total mean in the 2000 episodes (with hyperparameters: Learning Rate= $0.01, \gamma = 0.99$, Normalized = True, and Number of Hidden Layers = 0) was not the same as the one with the most running mean rewards of 200. Therefore, we retrained both hyperparameter values with 10^4 episodes each. These results can be found in Figures 3a & 3b.

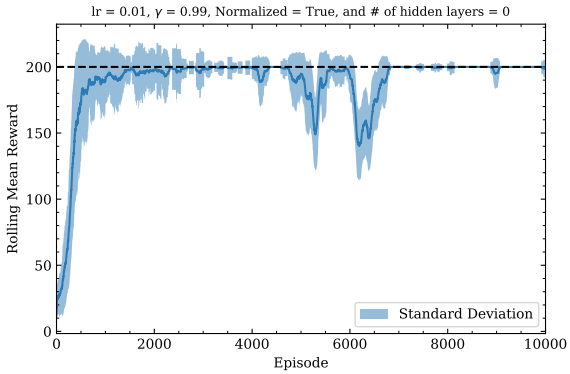
Regarding the hyperparameter statistics, Table 5 shows the ratio over the averaged cumulative reward from all runs while keeping a single hyperparameter constant. For example, in the first row, we took the mean over all cumulative rewards for simulations with the hyperparameter $\gamma = 0.95$ divided by the mean over all simulations with hyperparameter $\gamma = 0.99$. We observed that on average the discount value $\gamma = 0.99$ performed better than the lower value. From this table, we learn that higher discount values and Reward Normalization are the most effective tools in obtaining high cumulative rewards.

Furthermore, we noticed that hyperparameter values with the averaged highest total reward over each simulation also obtained a higher standard deviation over the cumulative rewards. However, combinations

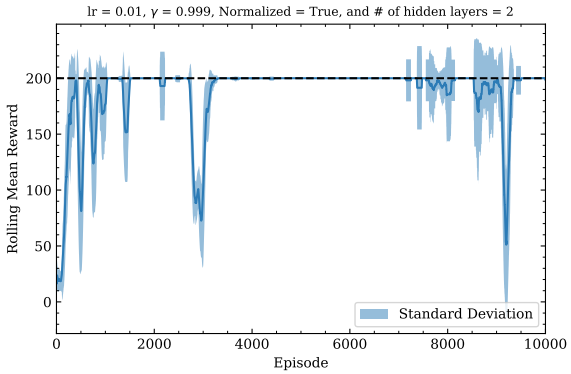
Table 4: Evaluated Hyperparameters for the Monte-Carlo Policy Gradients method

Hyperparameters	Values
Discount factor (γ)	[0.95, 0.99, **0.999**]
Learning Rate	[**0.01**, 0.001]
Number of Hidden Layers	[0, **2**]
Reward Normalization	[**True**, False]

Notes: *The values in bold combine the hyperparameters of the policy that produced the most running mean rewards of 200*



(a) Monte-Carlo Policy Gradients: Highest mean cumulative reward within 2000 episodes with 10000 episodes of training time



(b) Monte-Carlo Policy Gradients: Highest amount of successful runs in 2000 episodes with 10000 episodes of training time

Table 5: The ratio over the averaged mean from all hyperparameter combinations while keeping a single hyperparameter constant†.

Hyperparameters	Values	Averaged Mean
Discount Factor	$\gamma = 0.95/0.99$	0.72
	$\gamma = 0.95/0.999$	0.75
	$\gamma = 0.99/0.999$	1.04
Learning Rate	$\lambda = 0.01/0.001$	1.47
Number of Hidden Layers	$n_{\text{hidden}} = 0/2$	0.78
Reward Normalization	Norm = False /True	0.67

Notes: † *For example in the first row, we took the mean over all cumulative rewards for simulations with the hyperparameter* $\gamma = 0.95$ *divided by the mean over all simulations with hyperparameter* $\gamma = 0.99$.

that did not converge to a proper policy had a consistent running mean reward of $\sim 30 - 50$ resulting in a low scatter between their values. These non-successful simulations were mostly built without reward normalization.

4 Discussion

Tabular Q-Learning

Starting with the TQL model, the algorithm proved to be very sensitive to the grid size. With a too large grid, the model is not able to generalize enough while with a too-small grid the model is not able to learn enough to solve the CartPole-v0 problem. An intuitive interpretation of this is to imagine a too large angular velocity grid size. Say the pole passes through 0° at a given angular velocity that, due to the discretization, is interpreted as zero. The model is in this case not able to respond to transitioning states in time. Furthermore, the training of the model is very sensitive to the ϵ decay factor. If it is too

small, the model is not able to explore enough of the observation space before it converges. However, the best TQL model can solve the CartPole-v0 environment, while it proved unstable in the CartPole-v1 environment.

The reason is that the algorithm is not perfectly optimized. As stated before, the grid-search is performed with low resolution. A deeper look into hyperparameter space is needed to better optimize the performance of the model. The most interesting hyperparameters that can be investigated more, are these discretization grids. In this project, only linear bins of equal size are tested. It will be interesting to see how the model performs with different bin sizes and spacings. For instance, The model might perform better with more bins near zero for the angle and angular velocity of the pole and less at higher values.

A strong point of the TQL-model is its speed. Compared to the other models discussed in the report, the TQL model makes faster prediction sand trains faster.

Deep Q-Learning

The DQL models take a longer time to train, relative to the TQL models. Therefore, grid-search is even more difficult to perform with a high resolution, than it is for TQL. However, the performance of the DQL model is outstanding. With only 150 episodes of training, a very stable model has been trained. The best performing model is a simple, shallow feed forward network with only a single layer of 64 hidden nodes. This proofs that convolutions are not necessary to solve this simple environment to arbitrary performance. Further improvements of the hyperparameters might speed up the training of the network. We did not have the computational resources to test this.

A major disadvantage, as stated before, is the computation time of the DQL model. Compared to the best performing TQL model, one prediction, for the best performing DQL model, takes ~ 1.6 times as long. In a well-behaved gym environment, this is not a problem. The DQL model performs orders of magnitude better in terms of achieved reward. In real-world applications, however, the delay imposed by the computation time can have negative effects.

Monte-Carlo Policy Gradients

Finally, we need to discuss the performance of the REINFORCE method. First to note is that the training time for the policy optimization was slower than the previous methods. For every epoch with the MCPG implementation, we needed to sample an action from the policy until the environment terminated. This took longer when the model was performing appropriately as more roll-outs were necessary. Therefore, we also did not test the MCPG on the CartPole v1 with a maximum reward of 500 as it would be computationally to expensive. It took ~ 24 hours to train both models shown in Figures 3a & 3b.

Furthermore, to evaluate the stability of the training we had to run many episodes. Therefore, the resolution of the hyperparameter grid search was very low. Future directions on improving the stability of the policy training would be to run a more specific hyperparameter search, focused on the discount factor and the number of hidden layers, which had the lowest global variance of the training data within 2000 episodes.

Regarding the training stability, we can look at Figures 3a & 3b. Especially Figure 3b is interesting as after a period of ~ 3000 episodes maximizing the cumulative reward, we observe a sudden decrease in its stability. We believe this to be the cause of the stochastic nature of the REINFORCE algorithm originated by sampling an action from the policy instead of using an ϵ-greedy approach as done with implicit policies.

We believe, the stochastic method in this environment is at a disadvantage due to the low complexity of the CartPole, due to its small state and action space and low randomness as only the initial force push is random. The MCPG-method would be better applicable if the complexity of the environment increases, causing it difficult to discretize the inputs and hard to approximate the Q-values with a NN causing the decision process to become too slow for real-world implementations, or when the environment inhibits more randomness.

5 Conclusion

In this assignment, we are asked to implement three function approximation algorithms to the gym environment CartPole v0. The three algorithms we experimented with were: Tabular Q-learning (TQL), Deep Q-learning (DQL), and Monte-Carlo Policy Gradients (MCPG). We evaluated the training of all implementations based on their learning speed, stability, and success (defined as the amount of running mean rewards of 200 within a certain window of episodes). In total, we tested 18 different hyperparameters resulting in over a thousand different evaluated models.

With grid searches, we found the specific hyperparameters that maximized the results per algorithm. Chapter 3 shows that the TQL implementation had the fastest training speed of all three methods, while the DQN converged within the least amount of episodes. However, this is due to its specific implementation, see section 2.1. The MCPG needed the most episodes to illustrate its training performance and stability.

Furthermore, we found that the DQN-method was the most stable implementation, obtaining consecutive rewards of 500 over 10^6 episodes in CartPole v1. The MCPG was the least stable implementation as with its training it showed the highest variance of its running mean reward when compared with the other methods. This is observed when comparing the training history (Figure 2a, 2b, 3a, and 3b) of all best-performing models.

We believe this is caused by the stochastic implementation of the MCPG method and the low complexity of the CartPole environment. Evaluating these three implementations on environments with more randomness, such as the game Pac-Man, and higher complexities, such as real-time robotics with continuous input spaces, might result in different performances of the three methods.

6 Code

All code used to produce the results in this project can be found at `https://github.com/joosthart/cartpole`. The experiments can be run with a single command. The best obtained results can easily be tested and displayed using the minimal CLI. Acknowledgments, to blogposts that helpd us during the development of the code, are given in the `README.md`.

References

AG Barto R. S., Anderson C., 1983 (accessed 29-03-2021), CartPole, `https://gym.openai.com/envs/CartPole-v1/`

Barto A. G., Sutton R. S., Anderson C. W., 1983, IEEE transactions on systems, man, and cybernetics, pp 834–846

Brockman G., Cheung V., Pettersson L., Schneider J., Schulman J., Tang J., Zaremba W., 2016, OpenAI Gym, `http://arxiv.org/abs/1606.01540`

Moerland T., 2021, Lecture notes: Continuous Markov Decision Process and Policy Search

Plaat A., 2020, Learning to Play: Reinforcement Learning and Games. Springer Nature

Plaat A., 2021, Leiden University

Zhang S., Sutton R. S., 2017, arXiv e-prints, p. arXiv:1712.01275