Assignment 2: Reinforcement Learning 2021
Function Approximation

Deadline April 2nd 2021

1 Introduction

Reinforcement Learning (RL) is a machine learning paradigm where an agent learns the optimal action for a given task through repeated interaction with a dynamic environment, which either rewards or punishes the agent's behavior. This paradigm is widely known as operant conditioning within behavioral psychology and is a type of associative learning process through which the strength or occurrence of a behavior is modified by reinforcement or punishment [1]. In both RL and operant conditioning, initially random actions are shaped into goal oriented purposeful behavior with the help of supervision signals (more commonly referred to as reward signals), which evaluate an agents behavior during training. Learning occurs through interaction between agent and the environment, where the supervision signal is made available in-directly in the shape of rewards or punishments by the environment. As such RL can be regarded as semi-supervised learning [3]. Informally the agent's goal is to maximize the total amount of reward it receives. In order to achieve this goal, the agent should then not maximize the immediate reward, but maximize the cumulative long term reward [2].

There are two common methods called 1) action-value methods and 2) policy gradient methods, which are used to help the agent pick an action that maximizes its cumulative reward. Action-value based methods estimate a value function, which is a function of states (or of state–action pairs) that compute the expected return of future rewards given an agent's action in a given state. The reward an agent can expect in the future depends on what actions are chosen. Since rewards and actions depend on each other, a value function is usually defined with respect to particular ways of acting, called a policy. A policy is just a way to describe each possible action in a given state in terms of probabilities of selecting it. Policy gradient methods on the other hand, do not require a value function, but instead directly parameterize the policy. They seek to directly optimize and update the policy to find an action which maximizes the cumulative reward.

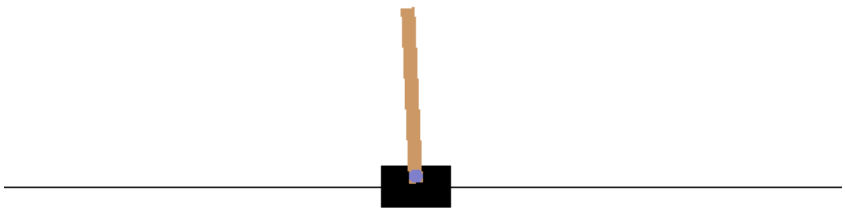


Figure 1: The cart-pole environment used for this report.

In details, since selecting a sequence of specific actions is then the result of excluding other possible actions [6], the optimization problem is not only dependent on the given reward r, but also on the set of possible actions a chosen within a policy π under a given state s, each having their own distribution. The agent starts at an initial state $s_0 \sim P(s_0)$ and repeatedly samples an action a_t from a policy $\pi_\theta(a_t \mid s_t)$. For each action a_t, the agent receives a reward $r_t \sim P(r_t \mid s_t, a_t)$ from the environment, then transitions to the next state s_{t+1} with respect to the Markov Decision Process (MDP) $P(s_{t+1} \mid a_t, s_t)$. This generates different trajectories of rewards, actions and states $(s_0, a_0, r_0, s_1, a_1, \ldots)$ with each trajectory having a varied spread

of reward. If an action-value method is used, actions which return a high value function v_π that should lead to a higher cumulative reward are prioritized when sampling an action a_t from a policy $\pi_\theta(a_t \mid s_t)$. That is, the action a_t in state s_t under policy π, which has the highest expected cumulative reward $q_\pi(s, a)$, is prioritized.

In this paper, we will investigate the potential of both action-value based methods and a policy gradient methods. To do so, we provide the implementation details of three well known methods within both frameworks. The first two being action-value based methods and the last method being a policy gradient method. These are tabular Q-learning, deep Q-learning using a Deep Q-Network (DQN) and Monte Carlo Policy Gradient (MCPG). They will be applied to the control of a cart-pole system. The objective in this task is to apply instantaneous forces to a cart moving in 1 dimension (left, right), along a track, to keep a pole hinged to the cart from falling over. A failure is said to occur if the pole falls past a given angle with the vertical axis, or if the cart runs off the track.

2 Methodology

2.1 Simulation

To simulate the cart-pole system, an available environment by OpenAI, Gym is used [4][5]. An example of the cart-pole environment can be seen in Figure 1. The environment consists of a black colored rectangular cart and a vertical brown colored pole attached to the cart. Cart and pole are attached via pivot joint, that allows only rotary movement around a single axis. The cart can move left or right. The problem is to prevent the vertical pole from falling, by moving the cart. The observation vector for the cart-pole system is a vector x containing four elements {cart position, cart velocity, pole angle, pole angular velocity}. The action has two possible inputs: left (0) and right (1). During training, an episode terminates under three conditions; either 1) if the pole angle reaches an angle that is larger than $\pm 15^\circ$ from the vertical axis, or 2) if the cart position is further than ± 2.4 units from the centre, or 3) if the episode length is greater than 200 in CartPole-v0, or greater than 500 in CartPole-v1. For every time-step taken without collapse, the agent receives a reward of 1. The task is considered solved if the agent reaches a cumulative reward of 195 out of 200 in CartPole-v0, or 475 out of 500 in CartPole-v1. Both versions are used throughout the paper, but it is clear from each figure which environment was used.

2.2 Implementation

2.2.1 Q-Learning

Q-learning is an off-policy reinforcement learning algorithm used to quantify the expected discounted future rewards an agent can obtain by choosing an action a_t in a given state s_t at time t [7]. The expected future discounted rewards are quantified using a Q-value function:

$$Q(s_t, a_t) = \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \mid (s_t, a_t)\right] \tag{1}$$

In the equation above the future rewards are represented as R_i for $i = t, t+1, \ldots, t+n$ and the reward discount factor as γ. In its basic form, given a sample of states, actions, rewards and next states denoted (s, a, r, s), Q-learning seeks to improve expected value approximation solutions using the following updates of equation:

$$Q(s, a) \leftarrow Q(s, a) + \alpha\left(r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right) \tag{2}$$

Since Q-learning assumes a discrete observation space in domains with large state spaces (i.e. continuous environments), these environments need to be 1) discretized so that similar state spaces fall into the same category or 2) algorithms need to represent the Q-function using a parameterized function approximator like a neural network. A neural network then parameterizes the Q-value function with its weights θ. Since the cart-pole environment is continuous, continuous state values are discretized directly and approximated using a neural network. As such we provide a tabular Q-learning implementation in which the state space is discretized and a Deep Q-Network (DQN) to approximate the Q-value function $Q(s, a)$ without discretization.

2.2.2 Tabular Q-Learning

To find the best action to take given the agent's current state, tabular Q-learning uses a table, called the Q-table, to store and remember the Q-values for each state-action pair $Q(s, a)$. To discretize the cart-pole environment, the ranges of the four elements of the environment were divided into multiple bins. The number of bins was varied to find the best implementation. The results of this can be read in Section 3.1. The Q-table was initialized with random values between 0 and 1 drawn from a uniform distribution. This was done to prevent that the first actions that are done are always the same. To give an idea: with three bins for each element in the environment, there are $3 \times 3 \times 3 \times 3 = 81$ rows in the Q-table, because there is a row in the Q-table for each different combination of the four element bins. This shows that an increase in number of bins rapidly increases the size of the Q-table and therefore slows down all processes. In a realistic situation, this is disadvantageous, since the faster the agent can determine the next action, the more control the agent has over the environment. In the simulated environment, this is not an issue, however, since the environment freezes after every step. The main issue why we do not want to increase the number of bins is that this would slow down the learning process significantly, since having a larger Q-table means more states have to be explored in order to converge to a well-working model. For this reason, the number of bins was kept at low integer values.

Besides that, we hypothesize, in line with [12], that some state parameters are less important than others. For example, cart position and velocity are not as important as pole angle or velocity. Ideally, you want the cart to be around the zero-point on the cart position axis, however, it is not as important to keep the cart on this point, as keeping the pole straight up, with a low angular velocity. For this reason, the amount of bins for the pole angle and angular velocity is expected to be higher than the amount of bins for the cart position and velocity.

Furthermore, to discretize the state parameter values of cart velocity and pole angular velocity, which have (nearly) infinite domains, ranging from (large integer values representing) -∞ to ∞, we decided to cut off the parameter value space at a certain threshold, above which all states with higher velocities are put into the same bin. This means that a threshold for cart velocity of 1 unit/s would discretize a state containing cart velocity of -50 units/s to the same bin as -2 units/s. We were not concerned too much with finding the perfect values for these thresholds, since we argue that the velocity is a less important parameter compared to the position and angle. The thresholds were put at ± 0.5 units/s for cart velocity and ± 50 deg/s for pole velocity. These are the same values as were used in [17].

Multiple values for amount of bins of each state parameter were experimented with. Furthermore, once a good combination of bins was found, experiments with respect to the learning schedule were performed. Four different learning schedules were tried out: decreasing ϵ, a large constant value for ϵ, a medium constant value for ϵ and small constant value for ϵ were all combined with a decreasing learning rate of the optimizer α. In Figure 2, the decreasing parameters α and ϵ are plotted as a function of the number of episodes. For the large, medium and small ϵ, the values $1, 0.5$ and 0.1 were used respectively. The results of the experiments with learning schedules are presented in Section 3.1.

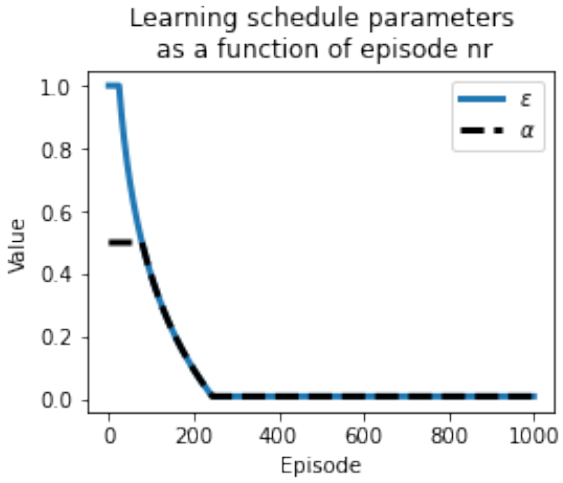


Figure 2: α and ϵ parameters of learning schedule "decreasing ϵ".

2.2.3 Deep Q-Learning

To maintain a continuous action space we represent the Q-function using a parameterized function approximator that is a deep neural network, which is parameterized by weights θ. In this case, the value-function estimate is updated as follows:

$$\theta \leftarrow \theta + \alpha \left(r + \gamma \max_{a'} Q\left(s', a'; \theta\right) - Q(s, a; \theta) \right) \nabla_\theta Q(s, a; \theta) \tag{3}$$

The DQN used employs two additional techniques, namely experience replay and the use of a separate target network, to stabilize learning and improve performance as suggested my Minh et al. [16]. Experience replay stores samples or experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ in a buffer, randomly samples a mini-batch, and performs the above update over that mini-batch. Since a random sample is drawn from the buffer, temporal correlations between observations are broken between consecutive samples, which can otherwise negatively effect the neural network's gradient descent. This means instead of running Q-learning on state action pairs as they occur during simulation, the replay buffer stores the previously discovered information in a table as quintuples. The target network is a copy of the estimated value function that is held fixed for some time to serve as a stable target for some number of steps. To be precise, it is a separate copy θ^- of the weight vector θ, which is used to create a temporal gap between the target action-value function and the action-value function that updates continually. This makes divergence less likely, since it adds a delay between the time that Q-values are updated and the time that the target Q_T values are updated, therefore adding stability.

These two techniques combat the inherent instability of the Q-value distribution and mitigate a phenomenon known as catastrophic forgetfulness in which an agent's performance drastically drops after a relatively high value is reached. To illustrate the improvement of these techniques, we conduct a simple experiment where we turn the experience replay on and off, illustrating the difference when both strategies are used and otherwise not used. All (except this auxiliary) experiments are conducted using experience replay in combination with a target network. The observation vector x is used as input and the network returns a Q-value for the possible actions the agent can take (left or right motion), where the action corresponding to the maximum Q-value is then chosen [8].

2.2.4 DQN Hyperparameter Tuning

Fine-tuning a neural network is an ongoing research field, since the complexity of tuning a neural network is extrapolated due to the vast number of hyperparameters to be tweaked. Furthermore, due to the large computational requirement to test neural models, techniques such as grid search or cross validation are infeasible. Therefore to determine an appropriate neural model, we relied on 1) literature research and 2) a step-wise building approach, where we update our base model based on experimental results step-wise. That is, first we experiment on simpler models that only vary with respect to one hyperparameter. Their performance is compared. The model with the best performance is then chosen for the next experiment. The winning model is then used for the next set of experiments and updated in turn if performance should improve. As such, we start with a general model, as described, moving towards a more specific and fined tuned one with each step, updating our previous model.

As a literature guideline, we used the DQN architecture for an Atari game as described by Plaat [9]. This network consists of 5 layers and uses convolutional layers. Since the complexity of the cart-pole environment is much lower, we concluded that a more parsimonious neural network with fewer nodes and fewer hidden layers would suffice. As such we started by examining three different types of architectures as shown in Table 1 with respect to their performance. Then, we experiment with different activation functions such as relu, elu, sigmoid and tanh. Lastly, we experimented with regularization, such as dropout layers and L1 (lasso) and L2 (ridge) to see whether a more parsimonious model can be obtained. Even though drop-out layers are technically not regarded as regularizers, they can achieve the same effect as the L1 regularizer [21], since they aim to randomly drop out a neural unit [11], therefore we included it in the regularization section. To recap, regularizers penalize unusually large values in the weights vector θ by adding a penalty term to the loss function. This reduces the relative influence of the weights θ and can stabilize training by artificially introducing a bias to the model, thus having a chance to improve performance. The lasso (L1) regularizer can shrink coefficient or values inside the weight matrix towards zero, effectively removing them [21], and creating a more simple, parsimonious model.

The ridge (L2) regularizer shrinks them towards zero (never equal to), therefore only reducing the relative influence [22].

To stabilize training, layer normalization [10], which standardizes the entire output of the layer, accelerates training and improves the performance of the deep network, has already been added in each experiment. Due to the inherent noisy distribution of the Q-value function, we use Adam optimizer [15] with a learning rate of 0.001. Adam is a replacement optimization algorithm for stochastic gradient descent (SGD). Adam provides an optimization algorithm that can handle sparse gradients on noisy problems, which seemed most appropriate for the cart-pole problem. The temperature curiosity parameter ϵ is decreasing as the number of episodes increase by a factor of 0.999 floored at 0.01. To break the temporal dependence of consecutive actions, we randomly sample mini-batch of size 32 from the replay buffer. The maximum number of epochs is set to 200 with an early stopping criterion, if the agent solves the task in 25 successive episodes with a score larger than or equal to 195.

Table 1: Comparison of Model Architectures

Layer	Nodes	Activation
Input	4	-
Dense	24	Relu
Output	2	Linear

a) Architecture 1

Layer	Nodes	Activation
Input	4	-
Dense	24	Relu
Dense	24	Relu
Output	2	Linear

b) Architecture 2

Layer	Nodes	Activation
Input	4	-
Dense	24	Relu
Dense	24	Relu
Dense	24	Relu
Output	2	Linear

c) Architecture 3

2.2.5 Monte Carlo Policy Gradient

A third method to solve the cart-pole environment is called Monte Carlo policy gradient (MCPG). The policy gradient method aims to optimize the policy directly [13]. A neural network was created to represent the policy. The implementation of this algorithm was based on [13]. The architecture and parameters of the implemented policy network were experimented with, the results can be seen in Section 3.3.

To optimize the policy, the expected return from the start state has to be maximized. This expected return is formulated as

$$J(\theta) = \mathbb{E}_{h_0 \sim p_\theta(h_0)}[R(h_0)]. \tag{4}$$

In this equation, $R(h_0)$ is the total reward in trace h_0, where trace h_0 is $\{s_0, a_0, r_0, s_1, \ldots, a_n, r_n, s_{n+1}\}$. The total reward of a trace is represented with

$$R(h_t) = \sum_{i=0}^{n} \gamma^i r_{t+i}. \tag{5}$$

To optimize the expected return, gradient descent is used [14]. An estimate of the gradient of Equation 4 is stated in Equation 6. This equation was used to formulate the loss function of the policy network.

$$\nabla_\theta J(\theta) = \mathbb{E}_{h_0 \sim p_\theta(h_0)}[\sum_{t=0}^{n} \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot R(h_t)] \tag{6}$$

For the third solution to the cart-pole environment, a policy network was implemented. The hyperparameters for this network were optimized by running the MCPG algorithm multiple times, with different hyperparameters.

The hyperparameters that were tuned are: the number of hidden layers, the number of hidden nodes in the hidden layers, the discount factor γ and the optimizer's learning rate α. Every network we trained has one or two hidden layers, the number of hidden nodes of whom is always equal. The number of hidden nodes H was varied, with $H \in \{16, 32, 64, 128, 256, 512\}$. The discount factor γ was also experimented with, with $\gamma \in \{0.85, 0.9, 0.95, 0.99\}$. The last parameter that was varied was the learning rate α, with $\alpha \in \{0.001, 0.002, 0.003, 0.004, 0.005, 0.006, 0.007, 0.008\}$.

This resulted in a total of $2 \times 4 \times 6 \times 8 = 384$ runs of the Monte Carlo policy gradient algorithm. All the models were run for 1000 episodes. The results of this are presented in Section 3.3.

The Monte Carlo policy gradient algorithm is a policy method. It directly optimizes the policy. The biggest difference between policy methods and for example DQN or tabular Q-learning is that the latter two do not directly optimize the policy, but learn the Q-value first, and later obtain the policy based on the learned value [13]. The Monte Carlo policy gradient algorithm is an on-policy method, meaning that the policy which is evaluated is the same policy as is being used to select actions [18]. On the other hand, tabular Q-learning and DQN are off-policy methods, the policy they evaluate is not the same as the one that is used for choosing actions. There are some disadvantages of the Monte Carlo policy gradient algorithm. Firstly, it converges slower than other methods because policy gradients converge step by step, which can take some time. Secondly, a lot of the time, the algorithm converges to a local optimum instead of the global optimum [19]. The last disadvantage is that policy methods mostly need more interaction with the environment than for example DQN or tabular Q-learning [13].

3 Results

3.1 Tabular Q-Learning

The most important parameters to be tuned for tabular Q-learning are the amount of bins each parameter's possible value is split up in. In order to find optimal values for these, a general learning schedule was stuck to, that applies to all experiments. This learning schedule is described in Section 2.2.2. The reason for sticking with this learning schedule is that it was found through experimentation which is described later in this section. The experiment to find the optimal number of bins was performed using a simple for-loop over all possible combinations of bin numbers for each parameter ranging from 1 through 6 bins. All agents then performed 250 episodes, and the resulting total reward was plotted as a function of episode number, similar to the right graph in Figure 3. In almost all cases, the model performed poorly, due to a bad choice of number of bins. There are several reasons that could explain a bad choice:

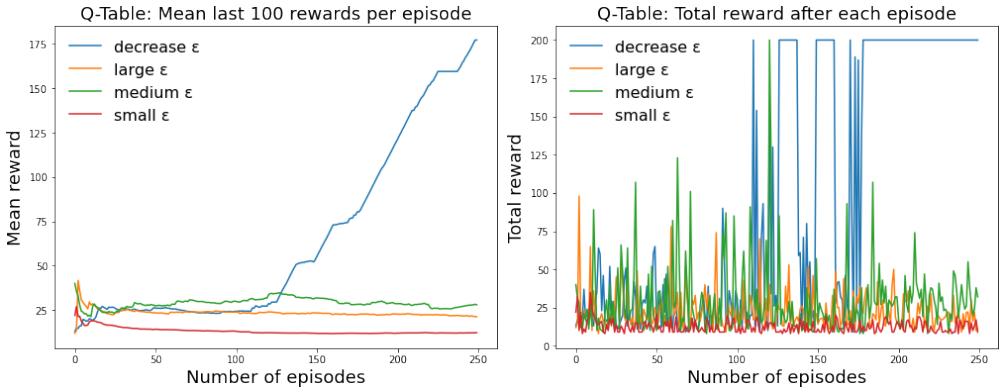
- A too small number of bins results in too little information about the cart-pole system, resulting in a too general Q-table, which causes the agent to make poor choices in very specific cases.
- A too large number of bins results in the Q-table being too large, causing the agent to learn the proper actions in each state too slowly.
- Some parameters are not as important as others, so having a poor choice of 'parameter importance' for some parameters results in one of the first two points.

After reviewing all graphs by a quick scan over the graphs from the experiment described above, it was quickly found that our hypothesis was confirmed: the pole angle is the most important parameter, because it requires the most amount of bins. The pole angular velocity is the second most important parameter, whereas the cart position and velocity do not influence the results very much, since only low values give a working model within the 250 episode run. The agent with the smallest amount of bins that consistently reached a good score had parameter bin numbers {cart position, cart velocity, pole angle, pole velocity}: [1,1,6,3]. The agent's score as a function of episode is shown as the blue line in the right graph of Figure 3.

Another interesting finding is that the parity of the parameters is sometimes of importance, which makes sense intuitively. If an even amount of bins is used, e.g. for pole angle, the positive and negative values for this parameter are always separated into separate bins. This causes the agent to know that if the pole angle is negative (so that the pole is pointing to the left), it should (most often) move the cart to the left. If an odd amount of bins is used, however, one of the bins is centered on zero, causing the model to have some ambiguity of values close to zero. In some cases, this matters, and the agent becomes 'clueless' for small angles. On the other hand, if the pole angle is small, it probably doesn't matter a lot what side the cart moves to, as in either case it will most likely increase the pole angle.

After finding the bins that worked best for the cart-pole environment, various learning schedules were applied, as was explained in Section 2.2.2. In Figure 3 the mean of the last 100 rewards per episode on the left, and the total reward after each episode on the right are shown for the four learning schedules that were applied to tabular Q-learning with [1,1,6,3] bins. For the three

learning schedules with a constant ϵ, the results are not very good. In both graphs in Figure 3, the orange, green and red lines stay low. The blue line however, goes upwards in the graph on the left, and reaches the maximal score of 200 in the figure on the right. This means that the learning schedule, in combination with the bins [1,1,6,3], works very well, and it solves the cart-pole environment in approximately 170 episodes. The reason why the 'decrease ϵ' learning schedule works well is because both the learning rate α and the exploration rate ϵ start out high and decrease over time. This makes sense because the algorithm will start with more exploration to get out of local optima and in the end, the exploration rate is low to prevent the algorithm to jump out of the global optimum. The learning schedules which have a constant exploration rate ϵ do not work well because a too large ϵ causes the algorithm to be very noisy and a too low value for ϵ will cause a model to get stuck in a local optimum.



(a) Mean reward.

(b) Total reward.

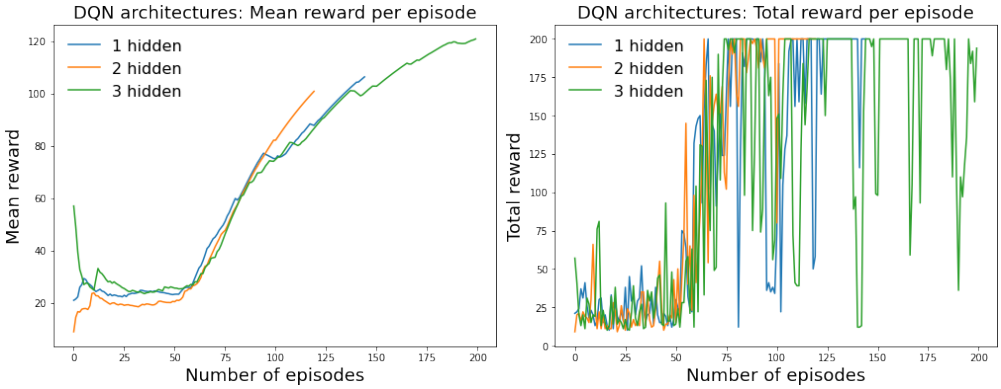
Figure 3: The mean of last 100 reward per episode with tabular Q-learning [left] and the total reward after each episode with tabular Q-learning [right].

3.2 Deep Q-Learning

3.2.1 Number of Hidden Layers

The performance of the DQN under a varying number of hidden layers as indicated in Table 1 is shown in Figure 4. Since the total reward graphs are noisy (see Figure 4b), the average reward (see Figure 4a) is graphed as well. From panel 4a we observe that using two hidden layers results in the best performance in terms of mean reward and reaching the early convergence criterion with less number of episodes. A network with two hidden layers reaches the early stopping criteria at about 125 episodes with the highest mean reward. A network using one hidden layer also is able to meet the early stopping criterion, however about 25 episodes later, ranked second in terms of mean reward. A network utilizing three hidden layers is however not able to reach the criterion after 200 episodes.

Assuming that solving the cart-pole environment for 25 successive episodes is a good indication of the network having converged, using two hidden layers converges fastest, while one hidden layer converges later on and three hidden layers does not. It would seem that a too shallow network with only one hidden layer cannot capture the complexity of the cart-pole game as quickly as a network with two hidden layers. A network with three hidden layers, on the other hand, would likely be able to capture the complexity. It would however require more time (episodes) to converge. This is to be expected since the number of weights to be tuned is much larger, thus requiring more time to arrive at a good solution. Overall a network with 2 hidden layers learns more efficiently per episode than a network with one or three hidden layers. As such, we adapt a network with two hidden layers and 24 nodes as shown in Table 1b for the following experiments.



(a) Mean reward.

(b) Total reward.

Figure 4: Experiments regarding a various number of hidden layers. The left figure shows the mean reward per episode as a function of episode number, the right figure shows the total reward for each episode.

3.2.2 Replay Buffer and Target Network On/Off

The auxiliary experiment to illustrate the benefit of using a replay buffer in combination with a target network can be seen in Figure 4. When both of these are turned off, the performance degrades. The agent immediately forgets after achieving a relatively large reward of around 80, returning to much lower reward values, illustrating catastrophic forgetfulness. As such, all DQN experiments are run using a replay buffer and target network.

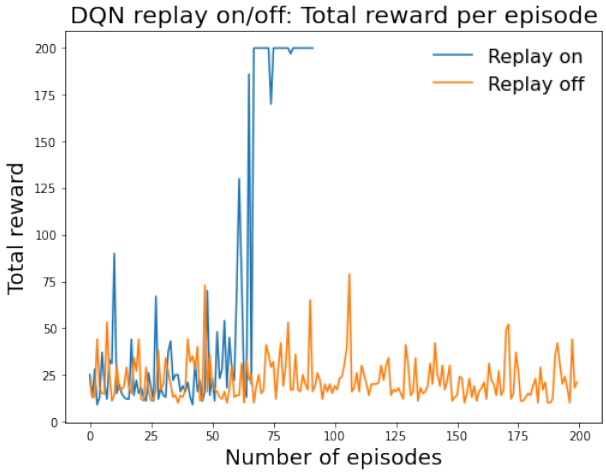


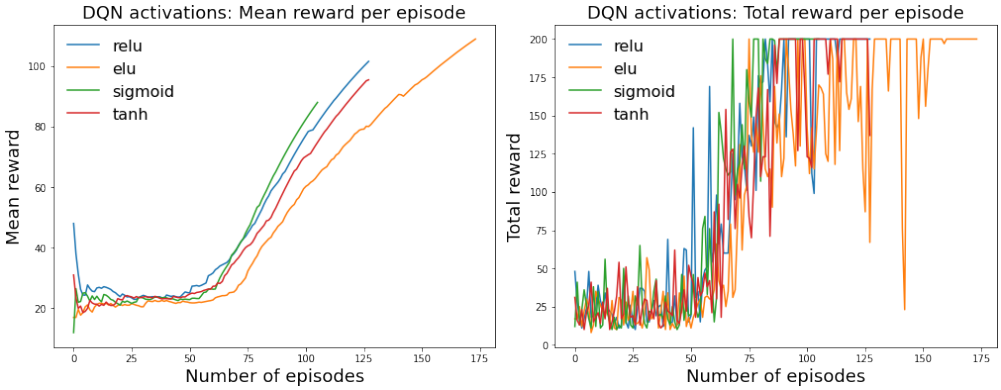
Figure 4: Total reward when replay buffer and target network are turned off and on.

3.2.3 Activation Functions

The effect of different activation functions on the performance of the neural network is shown in Figure 5. In Figure 5a we observe that the sigmoid activation and relu activation are quite close in performance, outperforming both tanh and elu. However, sigmoid outperforms relu with respect to reward. Sigmoid requires around a 100 episodes to reach the early stopping criterion and convergence, while relu reaches the criterion and converges at around 125 with a lower mean reward. Tanh is a close competitor to relu, reaching early stopping and convergence earlier, however with a lower score than relu. Elu also reaches the early stopping criterion at around 175 episodes, with a lower performance than the previous three. As such the activation functions for the cart-pole problem from most suitable to least suitable are sigmoid, relu, tanh and elu. The order is also indicative of which activation function leads to a network that learns most efficiently from the observations per episode. This is surprising, since relu reduces the likelihood of vanishing gradients and is computationally more efficient than sigmoid as it picks $\max(0, x)$ and does not need to perform an expensive exponential operation. However, it is unbound at x therefore unsaturated and may over-fit. The sigmoid function on the other hand squashes values

onto the $[0, 1]$ interval [23]. Hence, the derivative becomes small during gradient descent and vanishing gradients become more recurrent.

Given that sigmoid outperforms relu, it would suggest that smaller derivatives are suitable and vanishing gradients are less of a problem, especially since the depth of the network is rather shallow. Furthermore, the Adam optimizer makes working with sparse gradients [15] as in the case when using sigmoid more convenient, mitigating the adverse effect sigmoid may have on vanishing gradients. Since sigmoid outperforms the other activation functions, we replace the previous relu activation by a sigmoid for the following experiments. The new network then has two hidden layers utilizing a sigmoid activation instead of relu, 24 nodes per layer and a linear output activation.



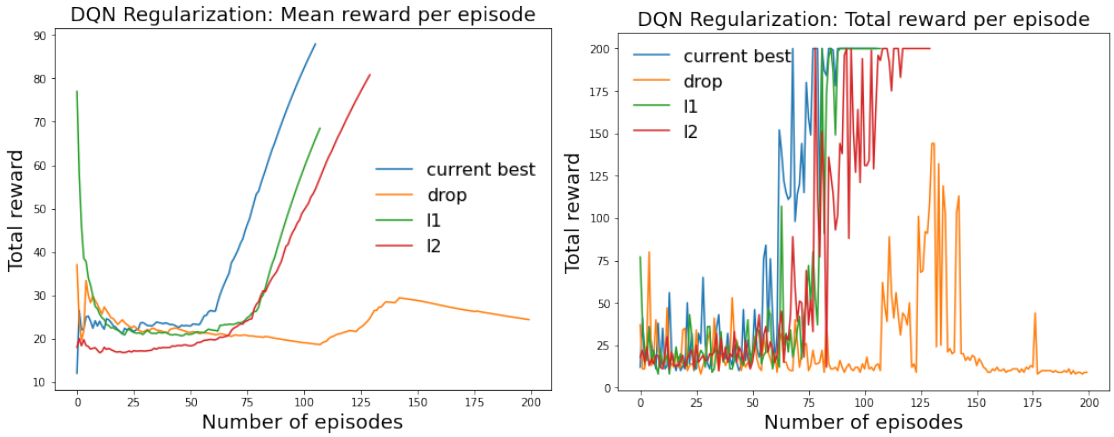
(a) Mean reward.

(b) Total reward.

Figure 5: Experiments regarding different activation functions for the hidden layers. The left figure shows the mean reward per episode, the right figure shows the total reward per episode.

3.2.4 Regularization

The effect of different regularization techniques to obtain a more stable and potentially more parsimonious model are shown in Figure 6. From Figure 6b, we observe that a dropout layer with a dropout rate of 0.1 effectively degrades performance in terms of total reward obtained as compared to a L1 (lasso) or L2 (ridge) penalty. Mean reward as shown in 6a is also much lower compared to the other networks. This indicates that dropping a hidden node, seriously degrades performance, indicating that a more parsimonious neural network with fewer nodes may not exist. To further test whether training can be stabilized and a potentially more parsimonious model can be obtained, the effect of a L1 (lasso) penalty and L2 (ridge) penalty, both initialized with a factor of $\lambda = 0.1$ is examined. Performance degeneracy occurred for both L1 and L2, where the less strict L2 regularization outperforms the stricter L1. The regularization methods are however not able to outperform our current best model. It would seem that the penalty factor λ is too high. However, the performance of L2 penalty, would indicate, that using 24 nodes may be optimal already. As such, we do not update the neural network and arrive at a current best model: a model with two hidden layers, sigmoid activation function and 24 nodes per layer with a linear output activation.



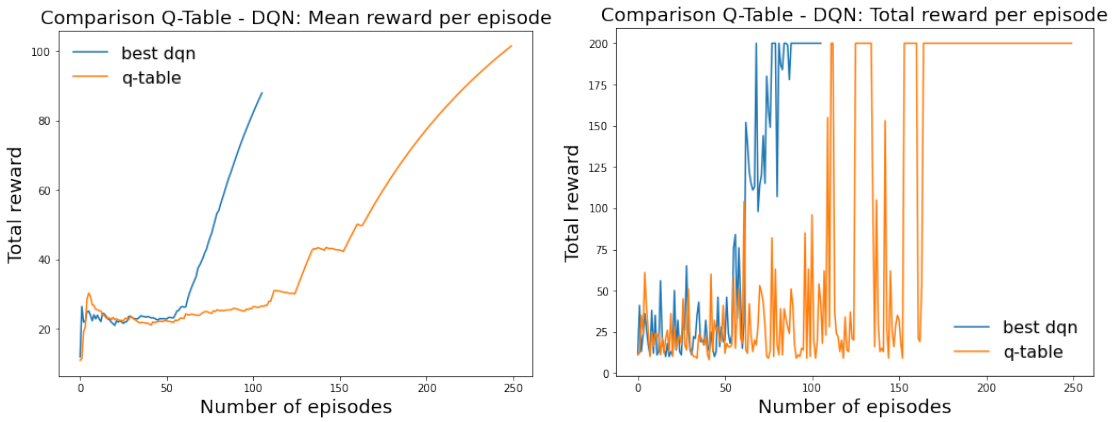
(a) Mean reward.

(b) Total reward.

Figure 6: Experiments regarding a different forms or regularization to obtain a more stable or parsimonious network. The left figure shows the mean reward per episode, the right figure shows the total reward per episode.

3.2.5 Model Comparison: Q-Table - DQN

The model comparison between tabular Q-learning and the final DQN can be seen in Figure 7. Visibly, the DQN outperforms tabular Q-learning, learning much more efficiently per episode. DQN consecutively solves the environment after around 110 episodes, while tabular Q-learning achieves a similar performance after 150 episodes.



(a) Mean reward.

(b) Total reward.

Figure 7: Model Comparison between Tabular Q-Learning and our optimised DQN.

3.3 Monte Carlo Policy Gradient

In this section, the results of the experiments with the MCPG algorithm are displayed. Most of the MCPG models are very inconsistent, which means that if they are run twice that the output will not be the same. Because of time constraints, all models were only run once for 1000 episodes. It could be that some of the models do not perform well in our results, but when they would be run again, they could have a better performance. This is a result of the stochasticity of MCPG, and an unfortunate shortcoming of the experiment.

3.3.1 Hyperparameter tuning

The four hyperparameters that were tuned for the Monte Carlo policy gradient algorithm were the number of hidden layers, the number of hidden nodes in each layer, the discount factor γ and the optimizer's learning rate α. Figure 8 shows the results. In the two graphs in the Figure, all models that achieved a running mean score over the last 50 episodes of at least 480 (these are the good models) are presented. In the left graph, the models that have one hidden layer are

shown, and in the right graph, the models with two hidden layers are shown. In both graphs, the horizontal axis represents the discount factor γ and the vertical axis represents the learning rate α. The size of the markers show the number of hidden nodes in the hidden layers.

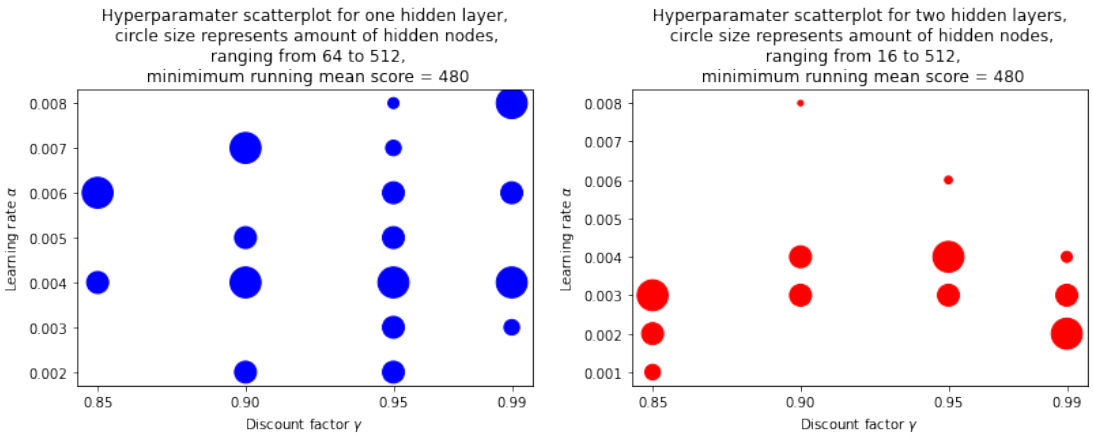


Figure 8: Hyperparameters of the MCPG models that had a minimum running mean score of 480 with [left] one hidden layer, and [right] two hidden layers.

From these two graphs in Figure 8, it can be seen that the models with two hidden layers mostly only achieve a good score when the learning rate is small. On the other hand, models that have only one hidden layer perform well for a big range of learning rates. Furthermore, in the right graph, more small markers can be seen, indicating that the models that performed well with two hidden layers more often contained a smaller number of hidden nodes. The smallest markers, representing 16 and 32 hidden nodes per hidden layer, are not present in the left graph, which means that a model with only 16 or 32 hidden nodes in the single layer it has did not achieve a running mean score above 480.

The models that are not included in Figure 8, have a lower running mean score than 480 at the end of the various runs. This does however not mean that those models are all very bad models. There are multiple reasons why a model does not have a high score at the end of the 1000 episodes. Firstly, models with a low learning rate often were not finished. This could be seen from a running mean score that was still increasing after 1000 episodes. Another reason could be that a model had reached a score of 500, but that the model became worse later on. An example of this can be seen in Figure 9.

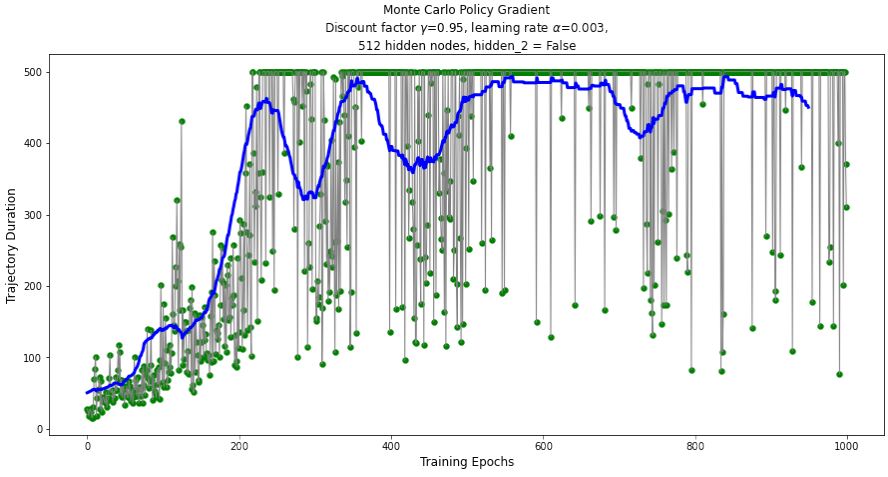


Figure 9: An example of a MCPG model that reached a good score, but decreased in the end.

In a lot of MCPG models, the perfect score of 500 was reached at some point, but later the algorithm was unable to maintain this score consistently. The reason for this is because of the stochastic nature of MCPG, the algorithm has a probability of chosing a different action than intended by the model. These small 'mistakes' can have a snowball effect and can lead to the

algorithm failing [20]. This is why a freeze was implemented as an experiment. This means that when the model reached a score of 500 ten times in a row, the model was frozen. A frozen model does not update anymore, this was implemented to see if the model really was a good model. In Figure 10 an example can be seen where the model is frozen after 437 episodes. It shows that the model is a very good model, as it keeps reaching a perfect score of 500 after the freeze.

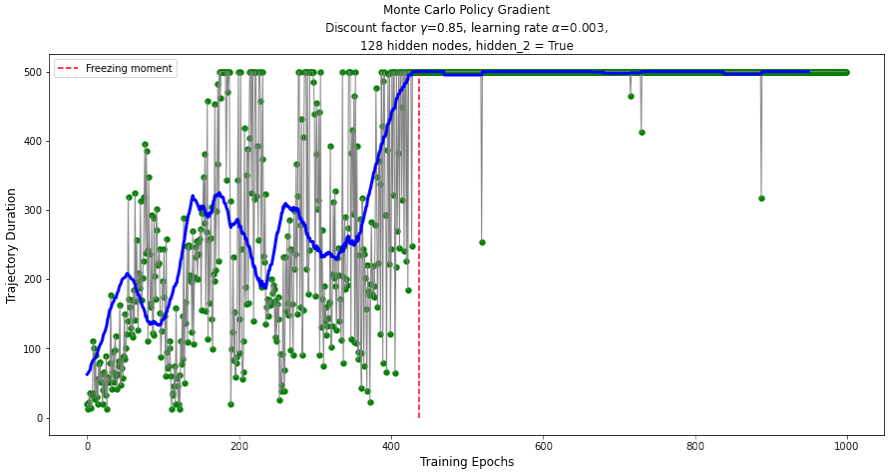


Figure 10: An MCPG model that was frozen after 437 episodes.

For each MCPG model that was run, the time that the algorithm took for 1000 episodes was also monitored. However, this is not a good feature of a model because it does not really represent how fast a model really is. The total time depends on how good the model is. When the model is good, the pole stays up for a longer time, so one episode has a longer duration. Whenever the model is not good, the pole won't stay up very long so the episodes take less time. Therefore, the total time of the models was not taken into account while comparing the models.

4 Conclusion

To summarize, three different methods of solving the cart-pole environment were implemented. Tabular Q-learning, which involves a Q-table that stores the best action to perform given a state, deep Q-learning, where a Deep Q-Network (DQN) is used instead of a Q-table, to approximate the Q-values of each action given a state; and Monte Carlo Policy Gradient (MCPG), a method that tries to optimize the policy directly, instead of using the Q-values as an intermediate step.

Tabular Q-learning teaches us that discretizing an environment can have both advantages and disadvantages. The advantage is that it is easy to understand and easy to configure. Disadvantages are that the amount of bins for all state parameters needs to stay small, in order to prevent a too large Q-table and in turn a slow learning process. Too few bins, however, results in a poor model. The optimal amount of bins that we found for each parameter is {cart position, cart velocity, pole angle, pole velocity}: [1,1,6,3], which gives a consistent model after about 150 episodes, if combined with a proper learning schedule.

A DQN on the other hand allows us to parameterize the Q-value function via the weights of a neural network, conveniently avoiding having to manually find a good way to bin and categorizing a continuous action space. This makes a DQN arguably more usable. However, it comes with a larger effort to fine-tune and optimize the larger number of hyperparameters of the network. Extensive experimentation is required to compare different model initializations. Compared to tabular Q-learning a fine tuned neural network will reach a higher mean reward after fewer episodes, solving the cart-pole simulations consistently at around 120 episodes. Therefore, a DQN is able to learn more efficiently per episode. The final network, with two hidden layers, 24 nodes each, a sigmoid activation function for the hidden layers, and a linear output activation function outperforms tabular Q-learning on all examined metrics. As such, if computation time is irrelevant a DQN should be preferred over tabular Q-learning.

For MCPG, hyperparameters that were optimized were: the number of hidden layers, the number of hidden nodes, the learning rate and the discount factor. The main findings from the experiments with all different hyperparameter settings for MCPG are that many hyperparameter combinations give a decent model, however, most models are very inconsistent. Freezing the

model in a good state seems to solve the issue. MCPG has a few disadvantages compared to deep Q-learning: it converges slower and the algorithm also converges to a local optimum more often than deep Q-learning. This can make MCPG a less efficient learner and less usable than DQN and tabular Q-learning, which given proper tuning will consistently converge at an optimal solution. Due to its stochastic nature, MCPG can be unstable, as shown in our case study.

Overall, we find that two hidden layers with 24 nodes for each layer, a sigmoid activation function for the hidden layers and a linear output activation function performs best among all three methods. It uses the information more efficiently per episode, thus learning faster and provides more stable results as compared to its examined competitors.

References

[1] Skinner, B. F. (1938). The behavior of organisms: An experimental analysis. New York, London: D. Appleton-Century Company, Incorporated.

[2] Sutton, R. S., Barto, A. G. (2018). Reinforcement Learning: An Introduction. The MIT Press.

[3] Chapelle, O., Schölkopf, B. Zien, A. (eds.) (2006). Semi-Supervised Learning. The MIT Press.

[4] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W. (2016). Openai gym. ArXiv Preprint ArXiv:1606.01540.

[5] OpenAI Gym. Toolkit for developing and comparing reinforcement learning algorithms. https://gym.openai.com/.

[6] Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based n approximating dynamic programming. Proceedings of the Seventh International Conference on Machine Learning, , 216-224.

[7] Violante, A. (2019). Simple Reinforcement Learning: Q-learning. [Link]

[8] Choudhary, A. A Hands-On Introduction to Deep Q-Learning using OpenAi Gym in Python (2019) [Link]

[9] Plaat, A. Learning to Play: Reinforcement Learning and Games. (2020) ISBN: 978-3-030-59238-7

[10] Ba, J., Kiros, J., Hinton, G.E. (2016). Layer Normalization. ArXiv, abs/1607.06450.

[11] Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I. Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting.. Journal of Machine Learning Research, 15, 1929-1958.

[12] Fakhry, A. (2020). Using Q-Learning for OpenAI's CartPole-v1. [Link]

[13] Lilian Weng https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html (2018)

[14] Torres J. Policy-Gradient Methods. https://towardsdatascience.com/policy-gradient-methods-104c783251e0 (2020)

[15] Kingma, D. P. Ba, J. (2014). Adam: A Method for Stochastic Optimization (cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015)

[16] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D. Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. International Conference on Machine Learning (p./pp. 1928-1937)

[17] JoeSnow7, Using Q-Learning to solve CartPole problem. https://github.com/JoeSnow7/Reinforcement-Learning/blob/master/Cartpole%20Q-learning.ipynb (2020)

[18] Abhishek Suran, On-Policy v/s Off-Policy Learning (2020) https://towardsdatascience.com/on-policy-v-s-off-policy-learning-75089916bc2f

[19] Thomas Simonini, An introduction to Policy Gradients with Cartpole and Doom. (2018) https://www.freecodecamp.org/news/an-introduction-to-policy-gradients-with-cartpole-and-doom-495b5ef2207f/: :text=with%20high%20probability.-,Disadvantages,converge%20slower%2C%20step%20by%20step.

[20] David McNeela, The problem(s) with policy gradient. *https* : *//mcneela.github.io/machine*$_l$*earning*/2019/06/03/*The* − *Problem* − *With* − *Policy* − *Gradient.html*

[21] Tibshirani, Robert. “Regression Shrinkage and Selection Via the Lasso.” Journal of the Royal Statistical Society: Series B (Methodological), vol. 58, no. 1, Jan. 1996, pp. 267–288, 10.1111/j.2517-6161.1996.tb02080.x.

[22] Hoerl, Arthur E., and Robert W. Kennard. “Ridge Regression: Biased Estimation for Nonorthogonal Problems.” Technometrics, vol. 42, no. 1, Feb. 2000, pp. 80–86, 10.1080/00401706.2000.10485983.

[23] Goodfellow, I., Bengio, Y.,, Courville, A. (2016). Deep Learning. MIT Press.