Capstone Project - Deep Reinforcement Learning

Deep reinforcement learning has generated a lot of public interest lately, most notably when DeepMind's AlphaGo defeated a professional Human Go player in 2016. In fact, deep reinforcement learning might very likely be a key ingredient for general artificial in the future. In this project, we will use two of the more common deep reinforcement learning algorithms to solve the prototypical example of problems for reinforcement learning: the pole–balancing problem.

1. Definition

1.1. Project Overview

For this project, I will use two algorithms of deep reinforcement learning — namely deep q–learning and policy gradients — to solve the problem of *pole-balancing*, as it was described by [8]. The first attempt to solve this problem with regular reinforcement learning methods was made in 1968 by Michi and Chambers, who used their reinforcement learning controller called BOXES. *" [T]hey applied it to the task of learning to balance a pole hinged to a movable cart on the basis of a failure signal occurring only when the pole fell or the cart reached the end of the track. This task was adapted from the earlier work of Widrow and Smith (1964) who used supervised learning methods, ..."* [8]. The problem is the following: *" The objective here is to apply forces to a cart moving along a track so as 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 from vertical or if the cart runs off the track. The pole is reset to vertical after each failure."*

The recent survey [1] gives a great overview of the field of deep reinforcement learning. In general, reinforcement learning is a framework for experience–driven autonomous learning. Prior to the advent of deep learning, reinforcement learning approaches lacked scalability and were limited to rather low–dimensional problems. Deep learning has helped reinforcement learning to scale to decision–making problems that were previously intractable. Two of the most notable successes of deep reinforcement learning have been

1. the development of an algorithm that could learn to play a range of ATARI 2600 video games at a superhuman level, just by looking at the pixels ([5], [4])
2. the development of a hybrid deep reinforcement learning system that defeated a human world champion in Go [7]

Following theses successes, deep reinforcement

learning algorithms have been applied to a wide range of problems, such as robotics, indoor navigation, computer animation and natural language processing. I think it is safe to say that deep reinforcement learning will be a major ingredient in future AI systems and — at least for me — it is the overall most interesting application of machine learning to date.

The data for this capstone project (which in a reinforcement learning problem corresponds to the environment) comes from OpenAI. In April 2016, OpenAI has released Gym, a toolkit for developing and comparing reinforcement learning algorithms. With Gym, everyone can write their own reinforcement learning algorithms and test it in a number of different environments without the need to build the environment oneself. This is an ideal platform for this project.

We will solve OpenAI's implementation of the pole-balancing environment mentioned above. We will use one algorithm from each of the two different main approaches to solving reinforcement learning problems, i.e., an algorithm based on policy iteration and an algorithm based on policy search and perform some hyperparameter tuning for these agents. As [5] have already stated: hyperparameter tuning for reinforcement learning (and especially for deep reinforcement learning) is computationally quite expensive. This is why we have also only touched on the subject later on.

1.2. Problem Statement

OpenAI's description of the pole–balancing problem is the following:

"A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center." [`https://gym.openai.com/envs/CartPole-v0`]

For this project, we will use two reinforcement learning algorithms to solve the problem: deep q–learning from the family of value iteration methods (see [5]) and the REINFORCE algorithm (see [10]) from the family of policy search methods.

The machine learning pipeline is rather short and straightforward for the cartpole problem due to its relative simplicity:

- **Obtaining data**: the clean data is provided by the OpenAI gym library in an easy–to–read format.
- **Pre-processing**: the cartpole problem is so simple that no pre-processing is necessary. This would be different if we used plain reinforcement learning, i.e., reinforcement learning without a neural network as function approximator. In that case, we would need to make sure that the continuous state–space of the problem is discretized appropriately, such that a solution on a computer is possible. With deep reinforcement learning, we use the data exactly as it is given to us by the OpenAI environment.
- **Training**: in reinforcement learning the agents learn to solve a given problem by trial–and–error learning. This means that a given agent repeatedly tries out possible actions in the environment and observes the rewards it obtains. Broadly speaking, the training part of our pipeline can thus be subdivided into:
  - perform a step in the environment
  - observe the state transition and rewards
  - update itself according to the algorithm used
- **Model Evaluation**: the evaluation of the model is done on–the–fly. Since the agent is updated on each training step it will also tend to perform better on each training step, meaning that one can continually measure its average reward. Once it achieves a satisfactory average reward, the training procedure can be cancelled.
- **Result**: the result of the training will be an agent with a good neural network approximation to its q–function (for the deep q–learning algorithm) or an agent with a good neural network approximation to its policy (for the REINFORCE agent).

1.3. Metrics

The metrics to use for this project are rather straightforward since OpenAI explicitly states when the cartpole-v0 environment is considered solved:

"CartPole-v0 defines "solving" as getting average reward of 195.0 over 100 consecutive trials." [`https://gym.openai.com/envs/CartPole-v0`]

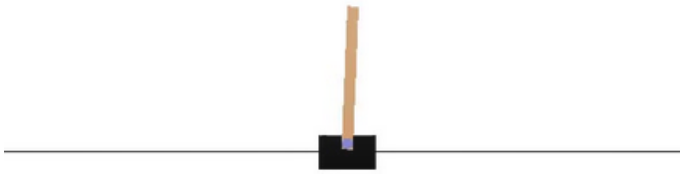


Figure 1: *The cartpole-v0 environment from OpenAI.*

We will also choose this metric to measure success: the reward per episode, averaged over 100 episodes, must exceed 195. One reason for using the averaged reward as a measure of success is that the approximated policy — which is used in training to determine the actions to take — is not the optimal policy and when small changes are applied to the approximated policy, this can lead to large changes in the distribution of states the policy visits. If we simply took the best reward achieved as our metric, this would lead to a quite random metric as long as our policy is still far away from the optimal policy.

2. Analysis

2.1. Data Exploration

We do not have a dataset for this problem, our "data" is the Markov decision process (MDP) to solve. Therefore, let us explore the MDP for the cartpole problem in this section. In general, an MDP is defined as the tuple $(\mathcal{S}, \mathcal{A}, P(s'|s,a), R(s,s'), \gamma)$, where

- $\mathcal{S}$ is a set of states
- $\mathcal{A}$ is a set of actions
- $P(s'|s,a)$ is the probability that taking action a in state s will lead to state s' (the model of the environment)
- $R(s,s')$ is the immediate reward received after transitioning from state s to state s'
- $\gamma \in [0,1]$ is the discount factor

Let us investigate each of these for the cartpole problem.

2.1.1. The State Space $\mathcal{S}$

The state space is 4-dimensional and continuous. The 4 components of the state vector s are:

$$s_1 = \text{horizontal position of the cart}$$
$$s_2 = \text{velocity of the cart}$$
$$s_3 = \text{angle of the pole}$$
$$s_4 = \text{angular velocity of the pole}$$

Every component of s is from a subset of $\mathbb{R}$. Looking at the source file of the environment on `https://github.com/openai/gym/blob/master/gym/envs/classic_control/cartpole.py`, we see that the following thresholds for the state space apply:

$$s_1 \in [-4.8, 4.8]$$
$$s_2 \in [-3.4 \cdot 10^{38}, 3.4 \cdot 10^{38}]$$
$$s_3 \in [-0.419, 0.419]$$
$$s_4 \in [-3.4 \cdot 10^{38}, 3.4 \cdot 10^{38}]$$

When s_1 or s_2 exceed their limits the episode is set to done. This makes sense because if s_1 exceeds its limits, the cartpole is not onscreen any more and if s_3 exceeds its limits, the angle of the pole is such that it cannot be balanced any more.

2.1.2. The Action Space $\mathcal{S}$

The action space is 1-dimensional and discrete, with 2 possible values for the action, i.e., $\mathcal{A} = \{0, 1\}$, where

$$a = 0 \equiv \text{cart is pushed to the left,}$$
$$a = 1 \equiv \text{cart is pushed to the right.}$$

2.1.3. The Environment Model $P(s'|s,a)$

The environment model is a physical model of the cartpole problem. We are not dealing with state transition probabilities here but with a classical control problem. This means that we have a deterministic model of the environment: the laws of physics. For cartpole-v0, the equations of motion are discretized with a very simple Euler forward method and the state s' is calculated according to the discretized equations based on the previous state s and a force that depends on whether the cart is pushed to the left (i.e., action 0 was chosen) or to the right (i.e., action 1 was chosen).

2.1.4. The Reward Structure $R(s,s')$

The reward structure for this MPD is very simple: for every step, the agent receives a reward of -1 until

the episode is done (i.e., until either s_1 or s_3 exceed the limits.)

2.1.5. The Discount Factor γ

γ is a hyperparameter for our models. Its influence on the performance of the agents is discussed in sections 4.1.2 and 4.2.2.

2.2. Exploratory Visualization

We cannot really perform exploratory visualization for this task because we have no data to visualize. What we can do is to visualize the state space, which I have done in figure 2.

It must suffice to show what the environment looks, which is shown in figure 1.

2.3. Algorithms and Techniques

To solve the cartpole environment, I will use two algorithms from the realm of reinforcement learning. In general, the algorithms in reinforcement learning can be categorized in two families: *value iteration* and *policy search* methods. I will use one algorithm from each family so solve the cartpole environment: deep q–learning [5] as an example from the value iteration family and policy gradients from the family of policy search methods. I will give a short introduction into both methods but this will be very superficial. The interested reader should check the quasi–standard reference [8], which is an excellent introductory presentation of the subject.

2.3.1. Deep Q–Learning

This section on deep q–learning is influenced by [5], [8], Tambet Matiisen's blog post, Keon's blog post and Ferdinand Muetsch's improvements thereof.

In 2013, [5] proposed an end-to-end reinforcement learning algorithm, which allowed an agent to learn to play several Atari 2600 games by just "looking at the screen", i.e., by getting pixels as input data. Recent advances in deep learning in general and computer vision in particular have made it possible to use a convolutional neural network to directly use the frames as inputs. Prior to their work, most successful reinforcement learning algorithms that use high–dimensional sensory inputs have relied on hand–crafted features combined with linear value functions or policy representations.

In the following section, I will first give a very short overview of general q–learning after which I will describe the deep q–learning approach used by [4].

Summary of Q–Learning As already stated, for an introduction to reinforcement learning in general, [8] is the de-facto standard reading. Here, we will start directly with q–learning, which is a so-called off-policy temporal difference control algorithm that was proposed by [9]. The q–function — also called action-value function — is often used when investigating Markov Decision Processes (MDPs). $q_\pi(s, a)$ represents the expected return when starting from the state s, taking the action a and thereafter following the policy π.

If we have access to the optimal q–function q^*, it is easy to choose the best action in a given state: you simply have to pick the action that gives you the highest q–value, i.e.,

$$a^*(s) = \operatorname*{argmax}_a q^*(s, a), \tag{1}$$

where $a^*(s)$ represents the optimal action to take when being in state s. Of course, we do not have access to the optimal q–function when starting out, so the problem we have to solve first: how do we obtain the optimal q–function? Q–learning is defined by the update rule

$$q(s_t, a_t) \leftarrow q(s_t, a_t) + \alpha \left[r_t + \gamma \max_a q(s_{t+1}, a) - q(s_t, a_t) \right] \tag{2}$$

For q–learning, the learned action–value function q directly approximates the optimal action–value function q^*, independent of the policy being followed.

In terms of storing the q–function on the computer, the easiest approximation is a table with states as rows and actions as columns. This gives us algorithm 1, Table q–learning.

Deep Q–Learning Approximating q with a table is OK for low–dimensional states. For high–dimensional ones — as e.g. when learning to play ATARI games from raw pixels — this approach is impossible, however. Instead, in deep q–learning, the q–function is approximated with a neural network:

$$q^*(s, a) \approx q(s, a; \theta), \tag{3}$$

where θ represents the parameters of the function approximation (i.e., the weights of the network).

The increase in computing power over the last decades and the ubiquitous availabiliy of data have

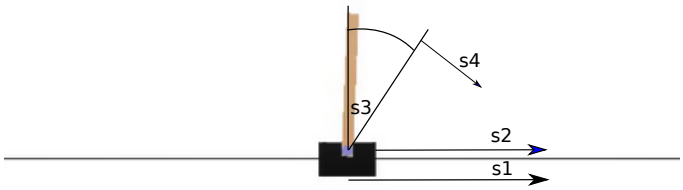


Figure 2: *The statespace of cartpole-v0. $s1$ is the horizontal position of the cart, $s2$ is the velocity of the cart, $s3$ is the angle of the pole and $s4$ is the angular velocity of the cart.*

Algorithm 1 Table Q-learning

initialize $q[\text{num\_states}, \text{num\_actions}]$
observe initial state s_t
repeat
    select and carry out an action a_t
    observe reward r_t and the new state s_{t+1}
    $q[s_t, a_t] \leftarrow q[s_t, a_t] + \alpha(r_t + \gamma \cdot \max_{a_{t+1}} q[s_{t+1}, a_{t+1}] - q[s_t, a_t])$
until terminated

made it possible for deep learning to achieve mind-boggling results in many tasks that rely on high–dimensional inputs, as e.g. vision and speech. When [5] proposed deep q–learning, they capitalized on that and used deep neural networks as function approximators to the q–function.

However, there are some challenges to overcome when using deep learning methods for reinforcement learning. Most deep learning methods

1. require large amounts of hand–labelled training data,
2. assume data to be independent,
3. assume a fixed underlying distribution of the data.

In reinforcement learning, on the other hand,

1. one must be able to learn from a scalar reward signal, which is sparse, noisy and delayed,
2. one typically encounters sequences of highly correlated states,
3. the data distribution changes as the algorithm learns new behaviours.

Points 2 and 3 basically mean that the data is not i.i.d. A key ingredient to deal with this non-i.i.d.-ness of data is the experience replay mechanism, which we will describe further below.

Network Architecture There are a number of ways to set up the network architecture. Usually, one would assume that when approximating a function with two arguments, $q(s, a)$, a network architecture that reflects these two arguments would be appropriate, i.e., using a network with two inputs and one output. However, when calculating $\max_a q(s, a)$, one has to perform a feedforward-pass through the network for each possible action in the action-space (which can be quite large). A better solution — and the one used by [5] — is to only have the state s as an input to the network and to have it output the q–value for each state in the action-space. For our case, the cartpole, this means that the network takes 4 inputs (horizontal position, velocity, angle of the pole, angular velocity of the pole) and yields 2 outputs (value of the state if going left, value of the state if going right).

Training The training of the network's weights is achieved via regression. To get an understanding of the process, imagine the experience tuple (s_t, a_t, r_t, s_{t+1}). We know that the best possible return in the current state s is given by the Bellman equation. This will be our target. The target has be the same as the prediction from the neural network for the current state, $q(s_t, a_t; \theta)$. This means that we have to minimize the loss function

$$L = \frac{1}{2}\Big[\underbrace{r_t + \gamma \max_{a_{t+1}} q(s_{t+1}, a_{t+1}; \theta)}_{\text{target}} - \underbrace{q(s_t, a_t; \theta)}_{\text{prediction}}\Big]^2.$$

The left term inside the parenthesis is the rhs of the Bellman equation, whereas the right term is the prediction by the network. To make the network learn, the following steps are performed:

1. do a feedforward pass for the current state s_t to obtain $q(s_t, a_t; \theta)$ for all actions a_t (and thus getting a value for the "prediction" part of the loss above)
2. do a feedforward pass for the next state s_{t+1} to obtain $q(s_{t+1}, a_{t+1})$ for all actions a_{t+1} and calculate the maximum over all q–values $\max_{a_{t+1}} q(s_{t+1}, a_{t+1}; \theta)$
3. for the performed action a_{t+1}, set the q–value target to $r_t + \gamma \max_{a_{t+1}} q(s_{t+1}, a_{t+1}; \theta)$, thus getting the "target" part of the loss above
4. for all other actions, set the q–value target to $q(s_t, a_t; \theta)$, obtained in step 1
5. backpropagate to update the network weights

Experience Replay As already mentioned earlier, the experience replay mechanism serves to deal with non-i.i.d. data that occurs in reinforcement learning. It works by storing the agent's experiences at each timestep into a replay memory:

for all timesteps t :

$$e_t = (s_t, a_t, r_t, s_{t+1}) \rightarrow \mathcal{D} = \{e_1, ..., e_N\}$$

When applying the q–learning updates, we apply them to samples of experiences, drawn at random from the replay memory $\mathcal{D}$. This randomization breaks the correlations between consecutive samples and therefore reduces the variance of the updates and smooths the training distribution over many past behaviours (see [5] for details).

The resulting deep q–learning algorithm with experience replay is shown in algorithm 2.

2.3.2. Policy Gradients

Instead of using methods based on state–value functions and action–value functions — as in deep q–learning for example — one can also directly learn a parametrized policy without using any value function. That is the basis of policy gradient methods. Note what while a value function might still be used to learn the policy parameter, it is *not* required for action–selection any more. According to [8], a policy gradient method is an instance of a method that learns a parametrized policy. As in the previous section, we will denote the parameter of the policy by θ. With this notation, the probability that action a is taken at time t, given that the environment is in state s at time t with parameter θ is written as

$$\pi(a|s, \theta) = \Pr\{A_t = a | S_t = s, \theta_t = \theta\} \qquad (4)$$

Policy–based methods have a few advantages over action–value methods:

- for some action selection mechanisms, it is possible that the approximate policy can approach a deterministic policy, whereas with ϵ-greedy action selection, there is always an ϵ probability of selecting a random action,
- policy approximating methods are able to find stochastic optimal policies, whereas action–value methods have no natural way to do that,
- the choice of policy parametrization is sometimes a good way of injecting prior knowledge about the desired form of the policy into the reinforcement learning learning system,
- the choice of policy parametrization is sometimes a good way of injecting prior knowledge about the desired form of the policy into the reinforcement learning system.

Policy gradient methods optimize the parameters of a policy by gradient ascent. For this, some performance measure $J(\theta)$ is chosen and the policy parameter θ is learned based on the gradient of $J(\theta)$ with respect to θ. The methods seek to maximize the performance measure, such that their updates approximate gradient ascent in J:

$$\theta_{t+1} = \theta_t + \widehat{\nabla J(\theta_t)}, \qquad (5)$$

where $\widehat{\nabla J(\theta_t)}$ is a stochastic estimate whose expectation approximates the gradient of the performance measure with respect to θ_t. Using the REINFORCE rule (see [10]) to get an explicit expression for the estimate $\widehat{\nabla J(\theta_t)}$ this can be expressed as

$$\theta_{t+1} = \theta_t + \alpha G_t \frac{\nabla_\theta \pi(A_t|S_t, \theta_t)}{\pi(A_t|S_t, \theta_t)}, \qquad (6)$$

where G_t denotes the return (i.e., the cumulative discounted reward) from time t. This means that each increment is proportional to the product of a return G_t and a vector: the gradient of the probability of taking the action actually taken divided by the probability of taking that action. This vector points into the direction in parameter space that most increases the probability of repeating the action A_t on future visits to the state S_t. The update increases the parameter vector in this direction proportional to the return, and inversely proportional to the action probability.

Since we do deep reinforcement learning here, we approximate the policy with a neural network.

Algorithm 2 Deep Q-learning with Experience Replay from [5]

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode = 1, M **do**
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ **do**
        with probability ϵ select a random action a_t
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        execute action a_t in emulator and observe reward r_t and image x_{t+1}
        set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        sample random minibatch of transitions $(\phi_t, a_t, r_t, \phi_{t+1})$ from $\mathcal{D}$
        set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
    end for
end for

The Neural Network Policy The ideas of this section mainly come from [2], from which we have also taken the idea for the parametrization.

We want to approximate $\pi(a|s)$. Again, one could assume that the inputs of the approximating neural network will be the state s and the action a and that the output of the network will be the probability for this given state-action pair. But as in the case of deep q–learning above, the input to the network will only be a state *without* an action. In the section about q–learning, the output of our network was the q–value for each action. Here, it will be the probability of taking the action for each action. But since the state–space consists of 2 elements only, we further simplify this by saying that the network will only output the probability of *one* of the actions, A_{left}, while the probability of the other possible action A_{right} will simply be calculated by

$$P(A_{\text{right}}) = 1 - P(A_{\text{left}}). \tag{7}$$

Having the probabilities for both actions available, we now select an action randomly according to the estimated probabilities. The reason for picking the action randomly instead of choosing the action with the highest probability is that this approach automatically lets the agent find the right balance between exploring new actions and exploiting the actions that are known to work well.

After initializing the policy with random weights, it doesn't achieve great performance, of course. For that, the correct policy has to be learned, which is achieved with the REINFORCE method described above.

Training the Policy We use the same variant of the REINFORCE algorithm as [2]:

1. Let the neural network policy play the game several times and at each step compute the gradients that would make the chosen action even more likely, but don't apply the gradients yet.
2. Once you have run several episodes, compute each action's score.
3. a) If an action's score is positive, it means that the action was good and you want to apply the gradients computed earlier to make the action even more likely to be chosen in the future.
   b) If an action's score is negative, it means that the action was bad and you want to apply the opposite gradients to make this action slightly less likely in the future.
4. Compute the mean of all the resulting gradient vectors and use it to perform a gradient ascent step.

For details on the training procedure, please check [2] or the accompanying Python code.

2.4. Benchmark

As benchmark, we can use the scores on the OpenAI website. There are 4 reproducable algorithms available on the website (the ones that say **writeup**):

- a deep q–learning agent by n1try (85 episodes before solve)

- a deep q–learning agent by mbalunovic (306 episodes before solve)
- a deep q–learning agent by ruippeixotog (933 episodes before solve)
- another deep q–learning agent by ruippeixotog (961 episodes before solve)

We will try to beat them or at least come close to them in terms of episodes before solve.

3. Methodology

3.1. Data Preprocessing

This section is not applicable to our project because we do not have to preprocess any data. The environment is given by OpenAI and ready to be used at once.

3.2. Implementation

For the implementation I have used Python 3.6 in concert with keras for deep q–learning and TensorFlow for policy gradients. The project code has been developed in an object-oriented manner such that the deep-q–learning agent and the policy-gradient agents are separate classes. The advantage of this is that they can be instantiated with different hyperparameters quite easily and can be compared with each other and with other OpenAI gym participants. The implementation of an existing algorithm is a rather straightforward process. Especially q–learning is not particularly difficult to implement and the neural network implementation with keras is also very easy. For policy gradients, it was a bit more difficult to understand the underlying algorithm and the implementation by [2] helped significantly to implement my own version of it (which in effect is the same implementation in an object oriented way). A more detailed description of the implementation is shown in appendix B.

3.3. Refinement

The refinement of the hyperparameters and network architecture was done while performing the experiments and can be found in the accompanying Jupyer notebook.

4. Results

The detailed results of the experiments can be found in the accompanying Jupyter notebook. I will summarize the results and visualize the most important findings concerning hyperparameter tuning and general performance.

4.1. Model Evaluation and Validation — Deep Q–Learning Agents

We started out our experiments with the deep q-network agents (DQN). As hyperparameters, we have used pretty much what [5] have used, with the exception of the replay memory capacity, the neural network architecture and the replay start size. The cartpole problem is much more lower-dimensional than the visual input from the Atari games, so we get away with a significantly simpler function approximator, compared to the CNN used by DeepMind.

4.1.1. Adjusting The Network Architecture

First, we have investigated the influence of the network architecture. For that, we have started out with the reference agent dqn1 and a network architecture of [10], i.e., with 1 hidden layer of 10 nodes. The result can be seen in figure 4a.

This choice of hyperparameters clearly does not make sense for this setting. The agent has very bad performance and does not reach the goal within 3000 episodes.

By varying the number of hidden layers and the number of hidden nodes, we have been able to find a neural network architecture that is more fitting for the problem at hand. The results of these experiments can be seen in figures 4a - 4e. Of these 5 hyperparameter settings, dqn4 achieves the best performance, which we have arbitrarily defined as reaching a high average score and low variance.

4.1.2. Adjusting the Discount Rate

We have used dqn4 as the new reference agent and have reduced the discount rate successivly from 0.99 to 0.75. The results can be seen in figures 5a - 5f.

The agent dqn7 clearly wins against dqn4 and dqn7 in terms of variance across different trials. The difference in performance between dqn7 and dqn8 is not quite as clear. dqn7 seems to be better in terms of the numbers of episodes needed to reach an average score of 195. Comparing the results with dqn9, it seems that, on average, dqn9 achieves higher scores faster. E.g., at episode 1000, dqn8 has a score of around 70, while dqn9 has 100. While the best trial of dqn10 was able to reach the goal of 195 in just around 1500 episodes, we prefer the low–variance results of dqn8

and dqn9, of which we have taken dqn9 as the new reference agent.

4.1.3. Adjusting the Learning Rate

The results for changing the learning rate α can be found in figures 6a - 6c.

It seems that higher learning rates as used by dqn11 lead to a faster increase of the score, while lower learning rates as used by dqn12 lead to slower learning. This is consistent with previous experience. The optimal thing to do would be to lower the learning rate of dqn11 once it reaches around 150 points at episode 1000. This would go to far for this project, however. We have not found that decreasing the learing rate leads to improved performance, so we will still use dqn9 as currently best-performing model.

4.1.4. Adjusting the Exploration Rate Decay

The exploration rate decay determines how fast the ϵ-greedy exploration algorithm reaches the lowest exploration rate of $\epsilon = 0.1$. The reference agent dqn9 has an exploration rate decay of 0.999. We have decreased this rate to 0.995 (dqn13) and 0.990 (dqn14). The results can be found in figures 9a - 9c.

It seems that lowering the exploration rate decay does not yield better performance. In retrospect, this makes sense since lowering the exploration rate decay to 0.99 means that the exploration rate will have reached the minimum of 0.1 at episode 229 already (since $0.99^{229} \approx 0.1$), which explains the high variance in the susequent episodes: the q–function is still far from the optimal one but exploration has already almost ceased.

4.1.5. Adjusting the Replay Start Size

The replay start size is the number of episodes the agent performs before it starts to learn. Generating these episodes is rather cheap (compared to learning with the neural net). Results can be found in figures 8a - 8c. dqn16 combines relatively smooth learning with a quite low variance. But keep in mind that we only have 5 samples here, so the results have to be taken with a grain of salt. In theory, the more episodes you simulate before starting to sample out of these, the more uncorrelated the samples should be. And for this environment, running 1500 episodes without learning takes about 1.5sec, so we can safely use a dqn with a higher setting for the replay start size, without sacrificing any performance. This is different for other environments, where simulating takes a long time, but for this very simple problem, it does not matter. dqn 16 will be taken as our new reference agent.

4.1.6. Adjusting the Replay Memory Size

The last batch of experiments we performed for the deep q–learning agent is the variation of the size of the replay memory. We have chosen a size of 10000 for our reference agent. We have performed experiments with sizes of 1000 (dqn17) and 100000 (dqn18). The results can be found in figures 9a - 9c. The performance of dqn17 seems quite erratic. Whereas dqn18 shows an almost strictly monotonous increase of average score, dqn17 shows a sharp kink around episode 1500. This kink is also present for dqn16, but it is much less pronounced. In terms of episodes needed to obtain a high average score, the picture is rather mixed. While dqn17 reaches a score of 100 already at about 900 episodes, dqn16 needs about 1200 and dqn18 needs about 1300 episodes to achieve the same score. 150 points, on the other hand, are achieved after about 1400 episodes by dqn16, after about 1500 episodes by dqn17 and after about 1600 episodes by dqn18. Because of this faster learning behaviour, we have decided to choose dqn16 as our best–performing agent based on deep q–learning.

Next, we will now investigate the results of the policy gradient based agents.

4.2. Model Evaluation and Validation — Policy Gradient Based Agents

In this section, we will take a look at the hyperparameter tuning we have performed for the policy gradient based agents. As with DQNs, we will start by an investigation into the impact of the neural network architecture.

4.2.1. Adjusting The Network Architecture

We have started out with the reference agent pga1 and a network architecture of [4], i.e., with 1 hidden layer of 4 nodes. Since the policy is easier to approximate than the q–function for DQNs, we can start with a simpler network use a simpler network to approximate the policy. The result for this reference pga is shown in figure 10a. Compared to deep q–learning, the learning is much smoother. Also, for the best trial of pga1, the goal of 195 points was reached after 1300 episodes already. However, there is also a very high variance in the results. We will now check if a change to the network architecture can deal with this issue.

The results for different architecture can be found in figures 10b - 10e. What is seen there is that for more hidden nodes, the variance decreases significantly. Increasing the number of nodes and the number of hidden layers increases both the performance of the agent and the variance across trials. The best agent of pga5 reached a score of 195 after 600 episodes already, which no other agent has achieved so far. In terms of variance, it also beats the other 2 agents shown. pga4 learns faster than pga2, but there is some crawling going on near the top, which is not present in the other agents. For pga4, there is already some variance present when starting out, whereas pga2 and pga5 are almost variance-free under 250 episodes. We will choose pga5 as our new reference agent because even the worst trial has achieved a score of 150 at 600 iterations. Next, we will vary the discount rate.

4.2.2. Adjusting the Discount Rate

The reference agent pga5 has a discount rate of 0.95. We have both increased it to 0.99 and decreased it to 0.80. The results are seen in figures 11a - 11c. The agents with a discount rate of 0.95 (pga5) and 0.80 (pga7) show very similar performance, whereas the agent with a discount rate of 0.99 (pga6) shows a much higher variance between trials towards higher episodes. We keep pga5 as our benchmark model and will now investigate the influcene of the number of simulated episodes per gradient update.

4.2.3. Adjusting the Number of Episodes per Gradient Update

Our current reference agent pga5 performs 10 simulations before updating the gradients and the policy. This is the same number used in [2]. We will now see if adjusting this number yields any benefits for the performance of the agent. First, we have increased the number to 20 and decreased it to 5 simulated episodes per update. The results can be seen in figures 12b and 12c, respectively. pga9 clearly outperforms pga5 and pga8 here. It has a much lower variance than pga8 and learns faster than pga5.

In the subsequent experiments, we have reduced the number of simulated episodes per update up to only one episode per update, the results of which are found in figures 12d and 12e. We see that a further decrease in the number of episodes per update does not yield any increase in performance. pga9 still is the best-performing agent. We will now turn towards our last hyperparameter to tune: the learning rate.

4.2.4. Adjusting the Learning Rate

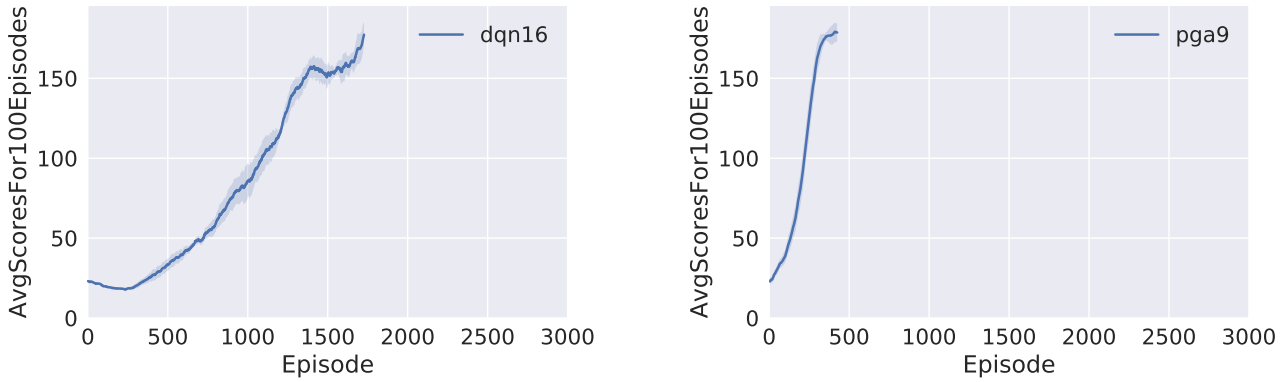
Our new reference agent pga9 used a learning rate of 0.01. We have compared the performance to agents using a learning rate of 0.02 (pga12) and 0.005 (pga13). The results are shown in figures 13a - 13c. The performance of pga9 and pga12 are quite close. pga13 has a higher variance and reaches the goal at a later stage, so it can be discarded. For pga9 and pga12, I think that pga9 has the best overall performance due to the slightly lower variance near the maximum score.

4.3. Justification

The justification for choosing the hyperparameters was done while presenting the results in the previous section. In this section, we will compare the results of our optimized models to the benchmark models that have been proposed in section 2.4:

- a deep q–learning agent by n1try (85 episodes before solve)
- a deep q–learning agent by mbalunovic (306 episodes before solve)
- a deep q–learning agent by ruippeixotog (933 episodes before solve)
- another deep q–learning agent by ruippeixotog (961 episodes before solve)

Our two best–performing models have been dqn16 with an average number of episodes before solve of 2056.8 and pga9 with an average number of episodes before solve of 688.0. Since every benchmark algorithm on the OpenAI website is a q–learning agent, it makes sense to compare the result of our q–learning agent dqn16 with these. 2056.8 is much worse than even the agent by ruippeixotog with 961 episodes before solve. However, I am not entirely certain if these numbers are comparable. E.g., solving the environment after 85 episodes is impossible if the averaged reward across 100 episodes is taken as a measure of success here. All of the commentators for this agent mention that the environment was never solved under 1000 episodes, so I have no idea how the number 85 came about. In retrospect, the benchmark I have chosen (the scores on the website) might have been bad, considering that nobody could not replicate them.



(a) *The best–performing DQN agent.*

(b) *Learning rate: 0.020*

Figure 3: *Comparison of the best–performing DQN and PG agents.*

5. Conclusion

5.1. Free-Form Visualization

We will compare the performance of the best–performing deep q–learning agent and the best–performing policy gradient agent in figure 3. It's easy to see the superior performance of the used policy gradient algorithm over the used deep q–learning algorithm. For this task, the policy gradient approach seems to be more suited. Another form of visualization would be to actually show the agent in action (i.e., show a video of the agents' performances for different agents and after different episodes). However, there are many videos of this kind available online and we would get no further insight here by recording the same thing.

5.2. Reflection

This was a fun project! I am still impressed by what reinforcement learning can achieve — with the small caveat: if we give it enough time to train. Training takes a significant amount of work and computing power. I still think that with ever–rising computing power in the future, we will see much more successes of reinforcement learning based agents. One day, one might nostalgically remember the good old times when deep reinforcement learning was mostly used to play video games. To summarize the project, I will describe the ML pipeline again that was implemented here:

- **Obtaining data**: the clean data was provided by the OpenAI gym library in an easy–to–read format
- **Pre-processing**: the cartpole problem is so simple that no pre-processing was necessary. The neural network takes care of the feature extraction
- **Training**:
  - perform a step in the environment
  - observe the state transition and rewards
  - update itself (q–value or policy) according to the algorithm used
- **Model Evaluation**: the evaluation of the model was done on–the–fly by measuring the average reward. Once a satisfactory average reward was achieved, the training procedure could be cancelled.
- **Result**: the result of the training is an agent with a good neural network approximation to its q–function (for the deep q–learning algorithm) or an agent with a good neural network approximation to its policy (for the REINFORCE agent).

5.3. Improvement

In a machine learning project, there is always room for improvement. The field advances in such a rapid way that it's quite hard to always use cutting–edge methods. This means that one way of improvement could be to use the newest methods of deep reinforcement learning. For deep q–learning this means that one could, e.g., incorporate the same improvements that [3] have used. For policy search this means that one could use Proximal Policy Optimization Algorithms as proprosed by [6], for example.

Another way to improve results in machine learning is via hyperparameter tuning. Due to the considerable computing costs, this was only done in a quite

superficial manner. It would be possible to use a structured grid–search or randomized–search to find better hyperparameters than we have done here.

However, one should keep in mind that the purpose of this project was not to beat today's best methods to solve the balancing-pole problem, but rather to understand and implement two common deep reinforcement learning algorithms, one from the policy iteration and one from the policy search family. Although I have proposed the currently best–performing method on OpenAI as a benchmark, I do not know how they have achieved this score. Maybe they have done some pre–training. With the methods and hyperparameters used here, I do not think that this score can be reached.

References

[1] Kai Arulkumaran et al. "Deep Reinforcement Learning: A Brief Survey". In: *IEEE Signal Processing Magazine* 34.6 (2017), pp. 26–38.

[2] Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems.* " O'Reilly Media, Inc.", 2017.

[3] Matteo Hessel et al. "Rainbow: Combining Improvements in Deep Reinforcement Learning". In: *arXiv preprint arXiv:1710.02298* (2017).

[4] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), p. 529.

[5] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[6] John Schulman et al. "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347* (2017).

[7] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *nature* 529.7587 (2016), pp. 484–489.

[8] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* Vol. 1. 1. MIT press Cambridge, 1998.

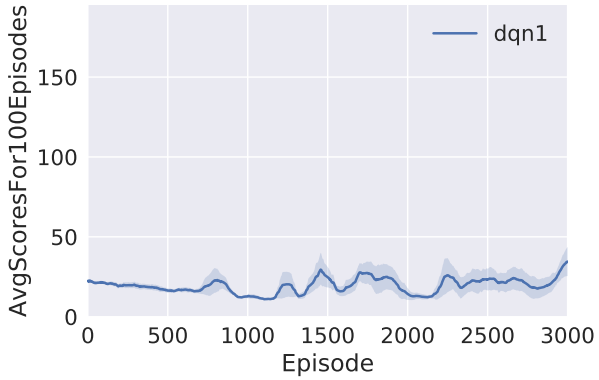
[9] Christopher JCH Watkins and Peter Dayan. "Q-learning". In: *Machine learning* 8.3-4 (1992), pp. 279–292.

[10] Ronald J Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Reinforcement Learning.* Springer, 1992, pp. 5–32.

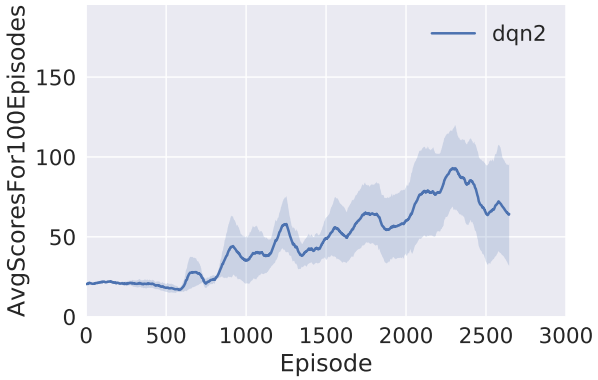
Appendices

A. Figures

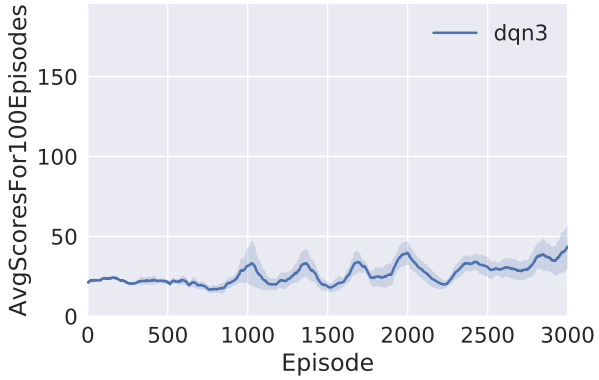
A.1. Deep Q–Learning



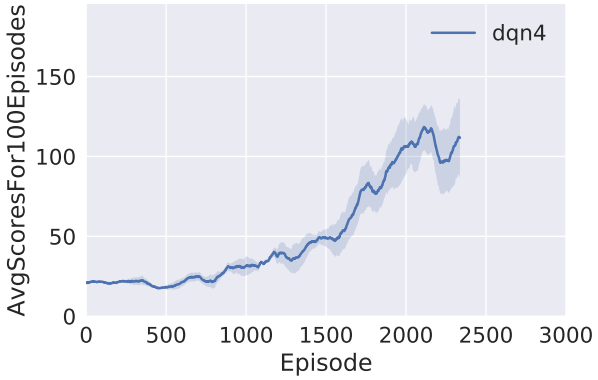
(a) *The reference DQN agent. NN architecture: [10]*



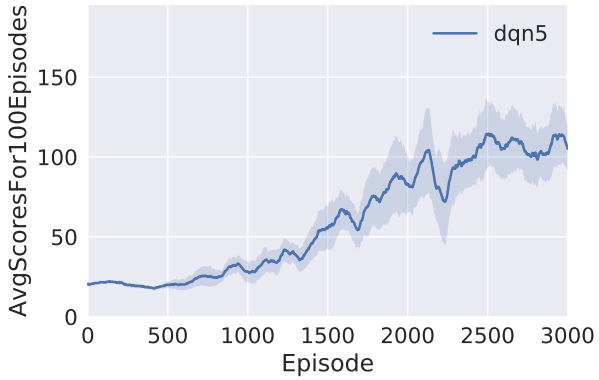
(b) *NN architecture: [10, 10]*



(c) *NN architecture [10, 20]*

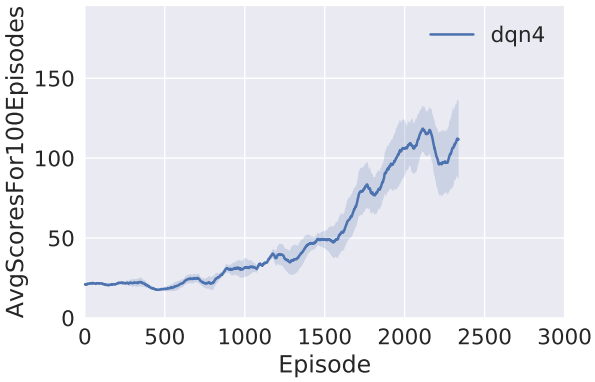


(d) *NN architecture: [20, 10]*

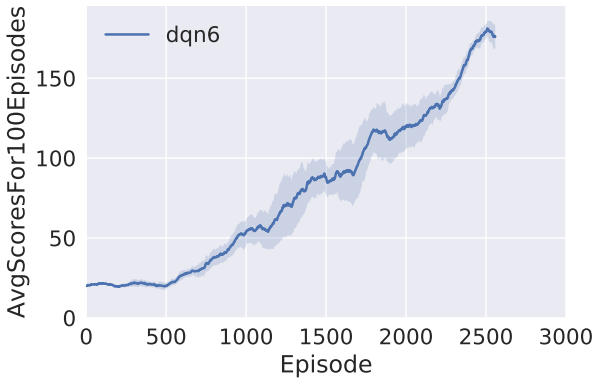


(e) *NN architecture: [30, 20]*

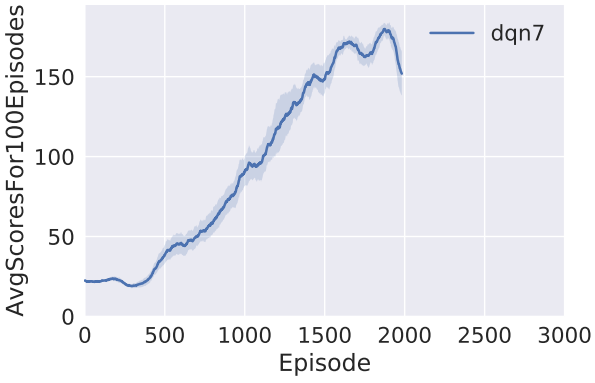
Figure 4: *These plots show the impact of changing the neural network architecture. Regarding the nomenclature: the reference agent has 1 hidden layer with 10 nodes, denoted by [10]. The agent in figure 2b has 2 hidden layers with 10 nodes each, denoted by [10, 10].*



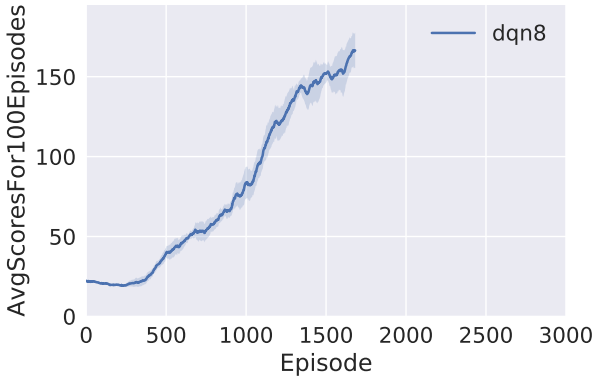
(a) *The new reference DQN agent. Discount rate: 0.99*



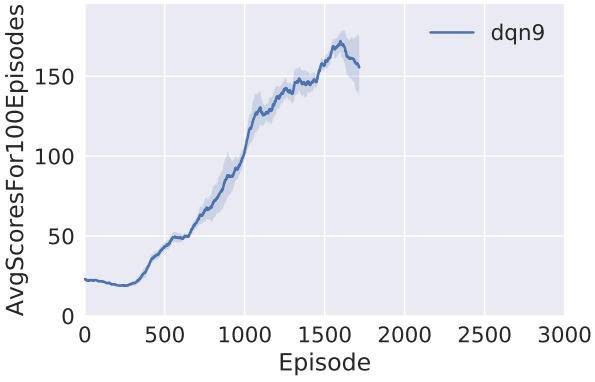
(b) *Discount rate: 0.95*



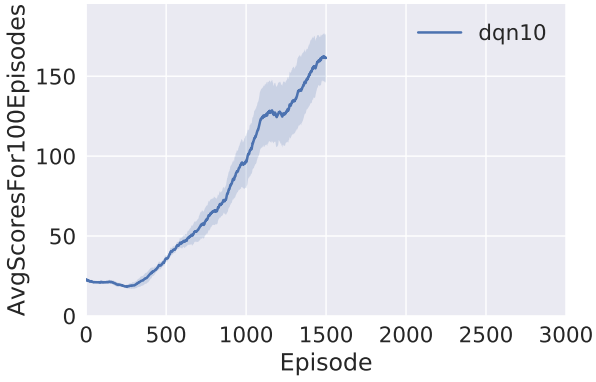
(c) *Discount rate: 0.90*



(d) *Discount rate: 0.85*

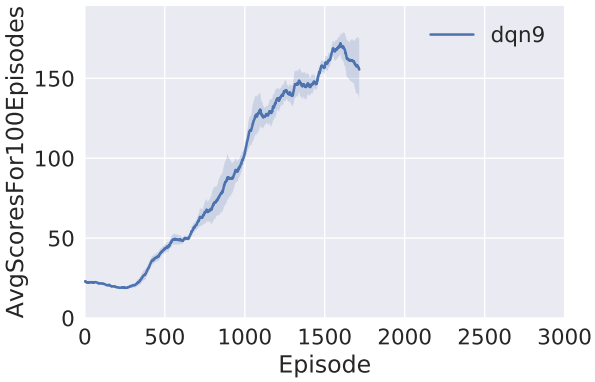


(e) *Discount rate: 0.80*

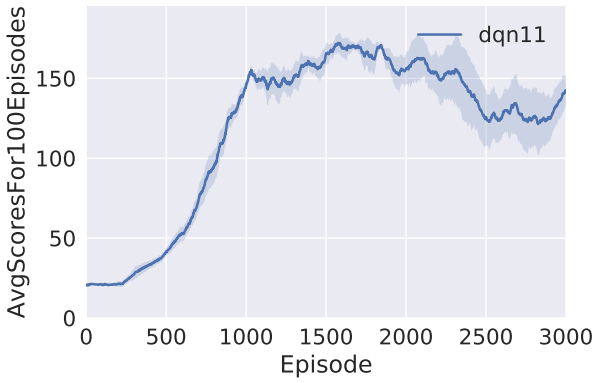


(f) *Discount rate: 0.75*

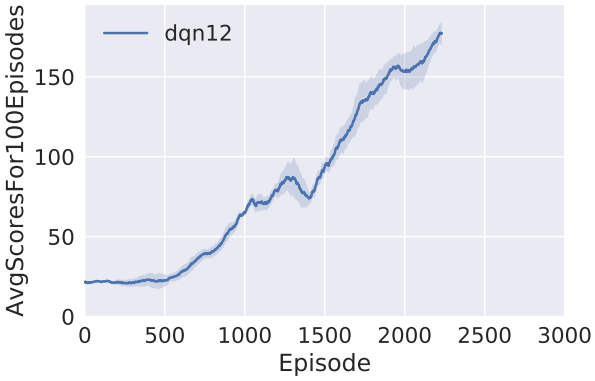
Figure 5: *These plots show the impact of changing the discount rate.*



(a) *The new reference DQN agent. Learning rate: 0.00025*

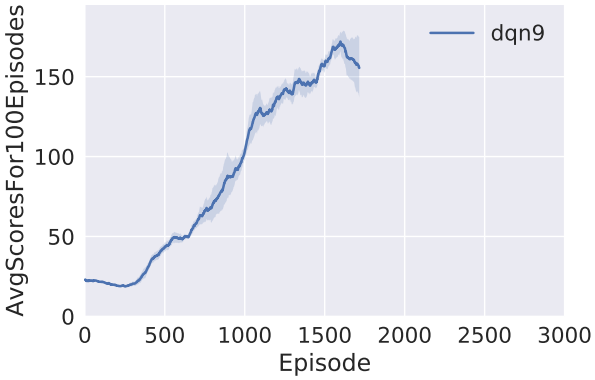


(b) *Learning rate: 0.00050*

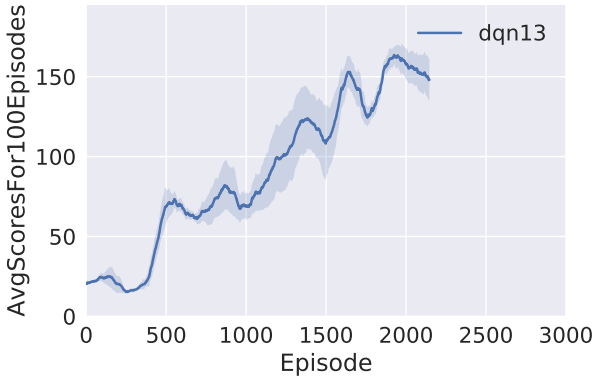


(c) *Learning rate: 0.000125*

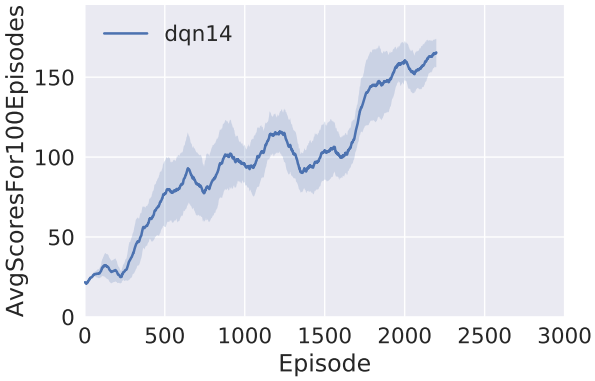
Figure 6: *These plots show the impact of changing the learning rate.*



(a) *The reference DQN agent.* $\epsilon_{decay} = 0.999$

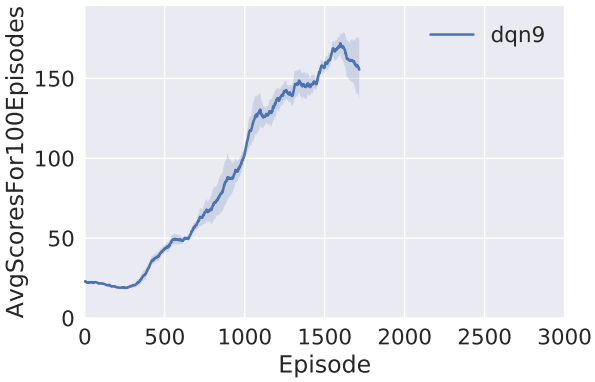


(b) $\epsilon_{decay} = 0.995$

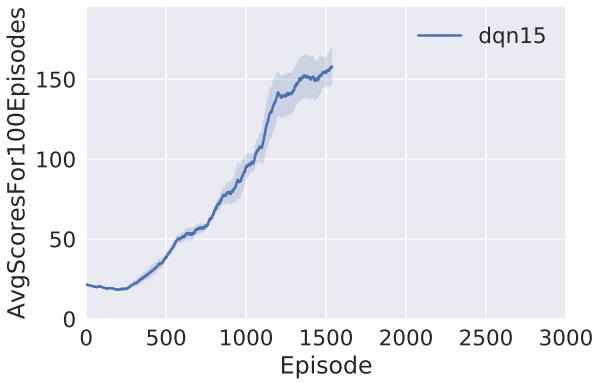


(c) $\epsilon_{decay} = 0.990$

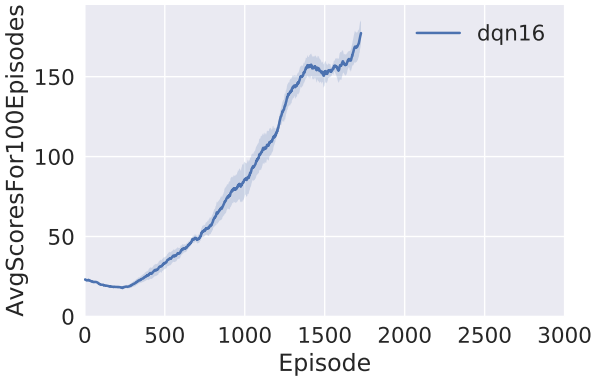
Figure 7: *These plots show the impact of changing the exploration rate decay* ϵ_{decay}.



(a) *The reference DQN agent. Replay memory start size: 32*

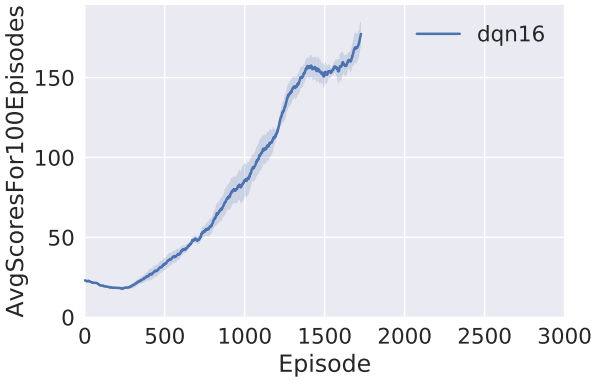


(b) *Replay memory start size: 320*

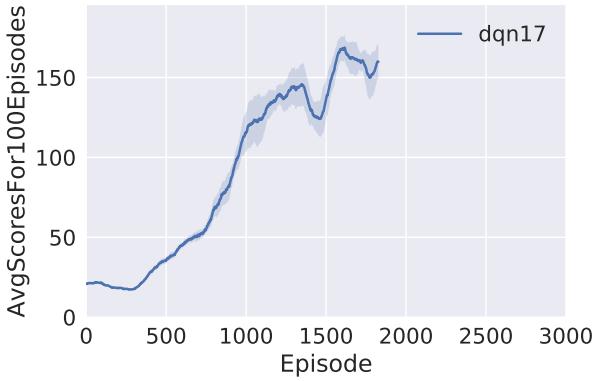


(c) *Replay memory start size: 3200*

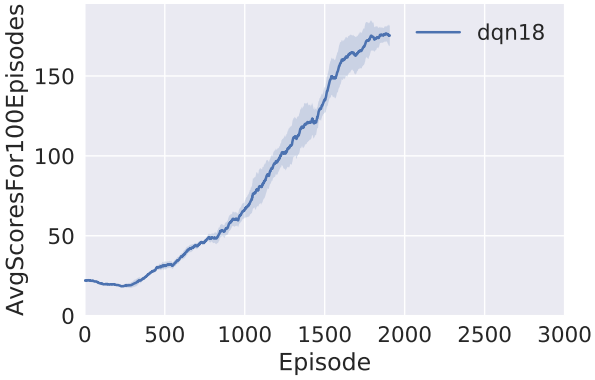
Figure 8: *These plots show the impact of changing the time step the agent starts to learn from its replay memory.*



(a) *The new reference DQN agent. Replay memory size: 10000*



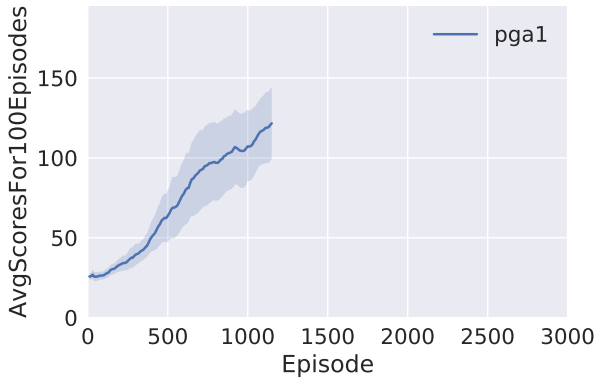
(b) *Replay memory size: 1000*



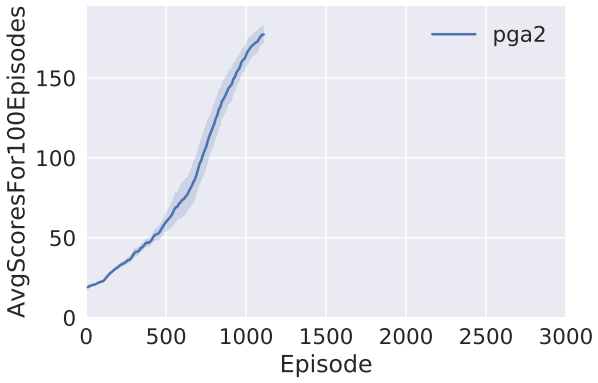
(c) *Replay memory size: 100000*

Figure 9: *These plots show the impact of changing the size of the replay memory.*

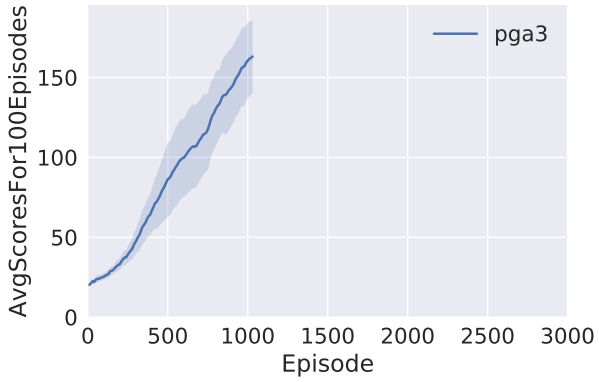
A.2. Policy Gradients



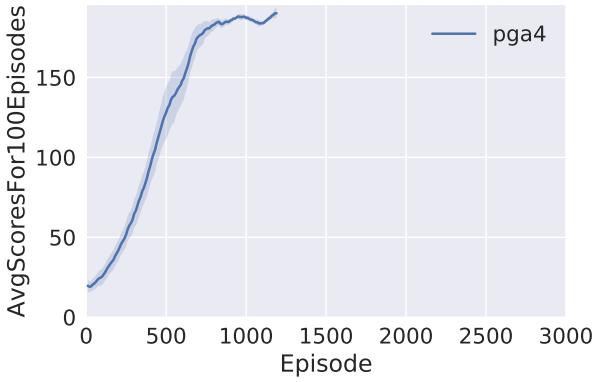
(a) *The reference DQN agent. NN architecture: [4]*



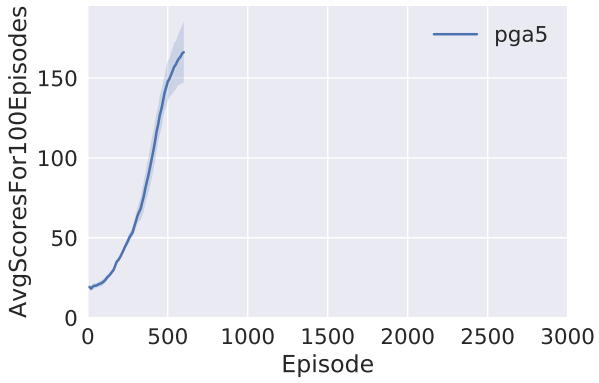
(b) *NN architecture: [10]*



(c) *NN architecture [4, 4]*

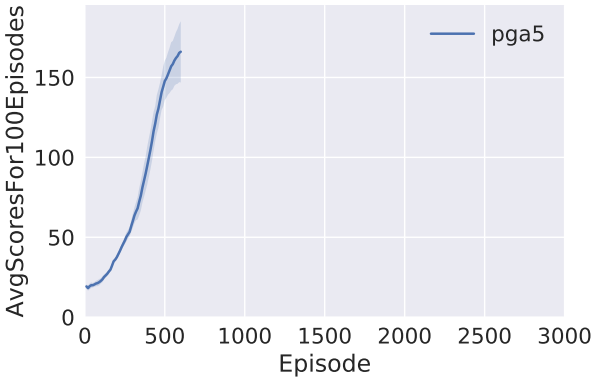


(d) *NN architecture: [20]*

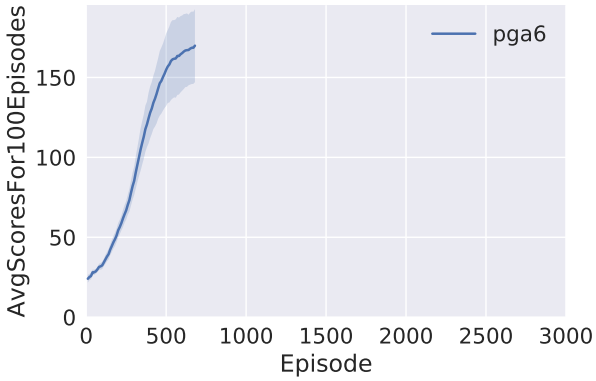


(e) *NN architecture: [10, 10]*

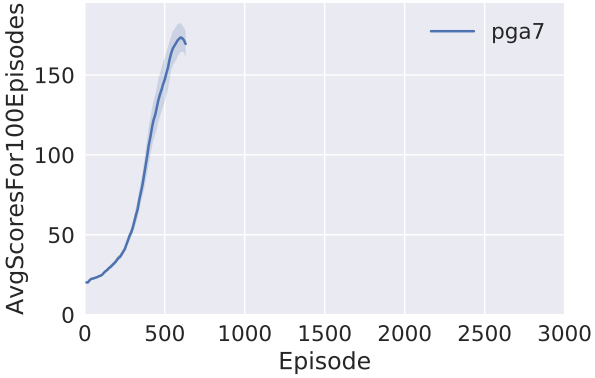
Figure 10: *These plots show the impact of changing the neural network architecture for the policy gradient agents. Regarding the nomenclature: the reference agent has 1 hidden layer with 10 nodes, denoted by [10]. The agent in figure 2b has 2 hidden layers with 10 nodes each, denoted by [10, 10].*



(a) *The new reference DQN agent. Discount rate: 0.95*

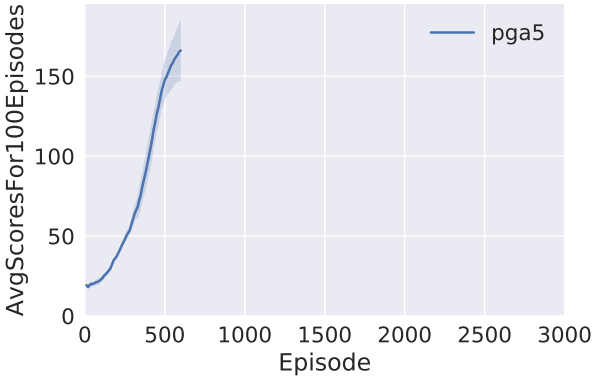


(b) *Discount rate: 0.99*

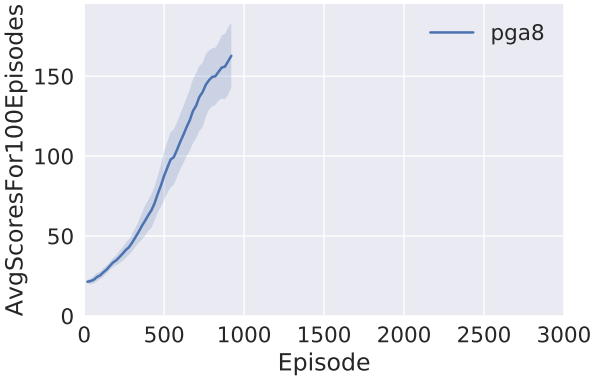


(c) *Discount rate: 0.80*

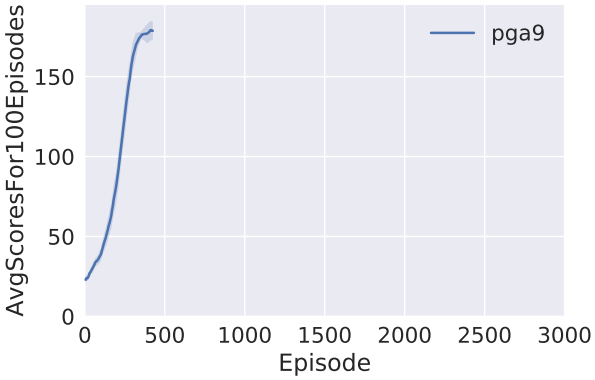
Figure 11: *These plots show the impact of changing the discount rate for the policy gradient agents.*



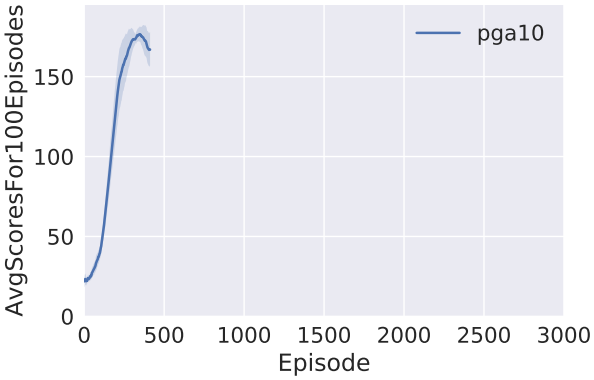
(a) *The reference DQN agent. Number of episodes per update: 10*



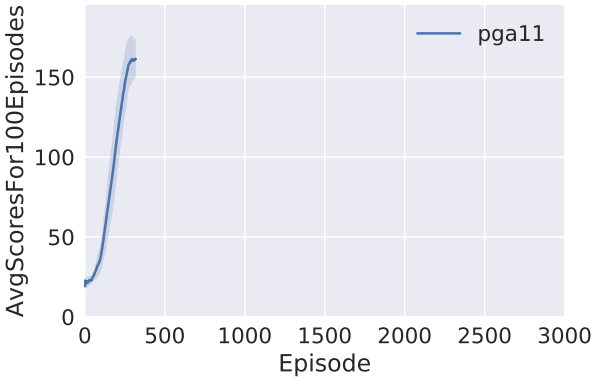
(b) *Number of episodes per update: 20*



(c) *Number of episodes per update: 5*

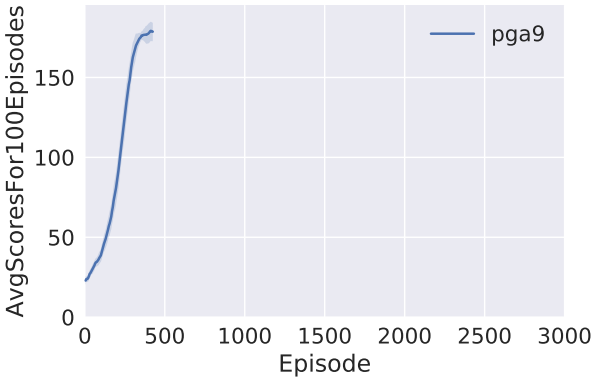


(d) *Number of episodes per update: 2*

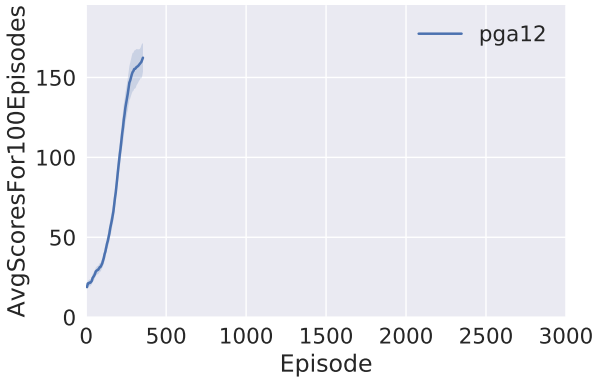


(e) *Number of episodes per update: 1*

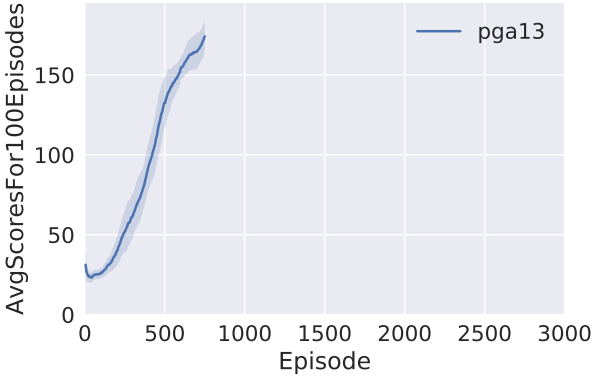
Figure 12: *These plots show the impact of changing the number of simulations before updating the policy.*



(a) *The new reference DQN agent. Learning rate: 0.010*



(b) *Learning rate: 0.020*



(c) *Learning rate: 0.005*

Figure 13: *These plots show the impact of changing the learning rate for the policy gradient agents.*

B. Implementation

B.1. Deep Q-Learning

To document the implementation of the deep q-learning algorithm, I will go through each line of algorithm 2 and describe my implementation thereof.

Line 1 Initialize replay memory $\mathcal{D}$ to capacity N.
A deque data structure was chosen for the replay memory, which acts as a queue here. A FIFO data structure is reasonable for the replay memory, which is supposed to hold the previous N experiences. So we have

```
self.memory = deque(maxlen=self.replay_memory_capacity)
```

where the replay_memory_capacity is a parameter of the agent.

Line 2 Initialize action-value function Q with random weights

In our case, Q is approximated with a neural network. We used keras for the deep q-network because of its simple usage. We have a quite simple neural network here. The weights are initialized with the default Glorot uniform initializer while the activation function is the ReLU. We have implemented the possibility to create a network with one, two or three layers. The final activation function is linear since we are in a regression setting.

```
# layers
if len(self.nn_architecture) == 1:
    model.add(keras.layers.Dense(self.nn_architecture[0],
              input_dim=self.state_size, activation='relu'))
elif len(self.nn_architecture) == 2:
    model.add(keras.layers.Dense(self.nn_architecture[0],
              input_dim=self.state_size, activation='relu'))
    model.add(keras.layers.Dense(self.nn_architecture[1],
                                 activation='relu'))
elif len(self.nn_architecture) == 3:
    model.add(keras.layers.Dense(self.nn_architecture[0],
              input_dim=self.state_size, activation='relu'))
    model.add(keras.layers.Dense(self.nn_architecture[1],
                                 activation='relu'))
    model.add(keras.layers.Dense(self.nn_architecture[2],
                                 activation='relu'))
else:
    raise ValueError("wrong dimension of nn_architecture")

# add the output layer with linear activation function
model.add(keras.layers.Dense(self.action_size, activation='linear'))

# compile the model with the MSE as loss and the Adam optimizer
model.compile(loss="mse",
                    optimizer=keras.optimizers.Adam(lr=self.learning_rate))
```

Line 4 Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$

Since we have no images and input, we do not have a preprocessing step for the image data. The only preprocessing step we do is to reshape the state vector to a row vector.

```
state = self.environment.reset()
state = np.reshape(state, [1, 4])
```

Lines 6 – 7 ϵ-greedy action selection

This is done via the act method:

```
def act(self, state):
    """
    The agent chooses an action based on the current state.

    :param state: the state of the environment
    :return: either a randomly chosen action or the argmax of the Q-network
    """

    # use epsilon-greedy exploration strategy: if a random number between
    # 0 and 1 is smaller equal the exploration_rate hyperparameter,
    # a randomly chosen action
    # is returned.
    if np.random.rand() <= self.exploration_rate:
        return self.environment.action_space.sample()

    # get the q-values for the current state
    all_q_values = self.QNetwork.predict(state)

    # return the action with the highest q-value
    return np.argmax(all_q_values[0])
```

Lines 8–9 Execute action a_t in emulator and observe reward r_t and image x_{t+1}; set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

This is a very straightforward step with OpenAI gym: we just have to call the environment's step method. Note that we set $s_{t+1} = s_t, a_t, x_{t+1}$ only after storing the transition in the replay memory in the next step

```
next_state, reward, done, _ = self.environment.step(action)
next_state = np.reshape(next_state, [1, 4])
```

Line 10 Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$

```
self.memory.append((state, action, reward, next_state, done))
```

Line 11 Sample random minibatch of transitions $(\phi_t, a_t, r_t, \phi_{t+1})$ from $\mathcal{D}$

```
random_experience = random.sample(self.memory, batch_size)
```

Line 12 – 13 Define the target y_j and perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$

```
for state, action, reward, next_state, done in random_experience:
    # if we have reached the terminal state (i.e., if done is True) we
    # set the target in the loss function to be the reward obtained for
    #  the
    # last transition (i.e., the value that is stored in reward)
```

```
        target = reward
        # if we have not reached the terminal state, the return (i.e., the
        # future discounted reward, defined by the Bellman equation) is
        # used as the target instead
        if not done:
            target = reward + self.discount_rate * np.amax(
                self.QNetwork.predict(next_state)[0])

        target_f = self.QNetwork.predict(state)
        target_f[0][action] = target
        self.QNetwork.fit(state, target_f, epochs=1, verbose=0)
```

B.2. Policy Gradient Agent

To document the implementation of the policy gradient algorithm, I will go through the REINFORCE algorithm by [2] and describe my implementation thereof.

B.2.1. The Construction Phase

A neural network with weights θ was chosen as parametrization for the policy. We have used TensorFlow for the implementation of the Policy Gradient agent. The number of inputs to the NN is the size of the observation space

```
self.numberOfInputs = self.environment.observation_space.shape[0]
```

The output of this policy will the probability of choosing action==left, so we have exactly one output:

```
numberOfOutputs = 1
```

The weights are initialized with unit variance:

```
initializer = tf.contrib.layers.variance_scaling_initializer()
```

We have implemented the possibility to create a network with one, two or three layers:

```
self.X = tf.placeholder(tf.float32, shape=[None, self.numberOfInputs])

# depending on self.nn_architecture, build a NN with 1, 2 or 3
#    hidden layers
if len(self.nn_architecture) == 1:
    last_hidden = tf.layers.dense(
                self.X, self.nn_architecture[0], activation=tf.nn.relu,
                kernel_initializer=initializer)
elif len(self.nn_architecture) == 2:
    hidden1 = tf.layers.dense(
                self.X, self.nn_architecture[0], activation=tf.nn.relu,
                kernel_initializer=initializer)
    last_hidden = tf.layers.dense(
                hidden1, self.nn_architecture[1], activation=tf.nn.relu,
                kernel_initializer=initializer)
elif len(self.nn_architecture) == 3:
    hidden1 = tf.layers.dense(
                self.X, self.nn_architecture[0], activation=tf.nn.relu,
                kernel_initializer=initializer)
    hidden2 = tf.layers.dense(
                hidden1, self.nn_architecture[1], activation=tf.nn.relu,
```

```
                kernel_initializer=initializer)
        last_hidden = tf.layers.dense(
                hidden2, self.nn_architecture[2], activation=tf.nn.relu,
                kernel_initializer=initializer)
    else:
        raise ValueError("Wrong NN architecture specified.")
```

The output will be a probability, which means that the output has to be between 0 and 1. We use the sigmoid function to produce this output.

```
    logits = tf.layers.dense(last_hidden, numberOfOutputs,
                             kernel_initializer=initializer)

    outputs = tf.nn.sigmoid(logits)
```

Action Selection
A random action, based on the estimated probabilities, is selected.

```
    ############ action selection ############
    # select random action based on the estimated probabilities
    # ----------------------------------------------------------------
    probabilitiesForGoingLeftAndRight = tf.concat(
        axis=1,
        values=[outputs, 1 - outputs]
    )

    # call the multinomial function to pick a random action based on the
    # calculated probabilities
    self.action = tf.multinomial(
        tf.log(probabilitiesForGoingLeftAndRight), num_samples=1)
```

Defining the Cost Function We are acting as though the chosen action is the best possible action, so the target probability must be 1.0 if the chosen action is 0 (left) and 0.0 if the chosen action is 1 (right).

```
    y = 1. - tf.to_float(self.action)
```

With the target probability defined we can now define the cost function

```
    cross_entropy = tf.nn.sigmoid_cross_entropy_with_logits(
                        labels=y, logits=logits)
```

Choose an optimizer, set it's learning rate and compute the gradients of the the cost function

```
    optimizer = tf.train.AdamOptimizer(learning_rate=self.learning_rate)
```

The next command returns a a list of (gradient, variable) pairs where 'gradient' is the gradient for 'variable'

```
    gradients_wrt_to_variables = optimizer.compute_gradients(cross_entropy)
```

Put gradients into their own list (they will be changed later)

```
    self.gradients = [gradient for gradient,
                      variable in gradients_wrt_to_variables]
```

During the execution phase, the gradients will be tweaked and reapplied to the optimizer. first, we need a list that can hold the tweaked gradients. In this list, we have to initialize some tensorflow placeholders

```
    self.tweakedGradientPlaceholders = []
    tweakedGradientsAndVariables = []

    for gradient, variable in gradients_wrt_to_variables:
        gradientPlaceholder = tf.placeholder(
```

```
                tf.float32, shape=gradient.get_shape())
        self.tweakedGradientPlaceholders.append(gradientPlaceholder)
        tweakedGradientsAndVariables.append((gradientPlaceholder, variable))
```

For training, we apply these tweaked gradients to the optimizer

```
self.trainingOperation = optimizer.apply_gradients(tweakedGradientsAndVariables)
```

Initialize TF variables and the TF saver.

```
self.init = tf.global_variables_initializer()
self.saver = tf.train.Saver()
```

B.2.2. The Execution Phase

The execution phase is coded in the method `solve`. First, a variable is defined that keeps score of the last 100 episodes and an empty list is defined that is to collect every trajectory:

```
lastHundredScores = deque(maxlen=100)
trajectories = []
```

Now, let the neural network policy play the game several times and at each step, compute the gradients that would make the chosen action even more likely, but don't apply these gradients yet

```
with tf.Session(graph=self.graph) as sess:
    sess.run(tf.global_variables_initializer())

    # attention:
    # the following loop will not loop numberOfEpisodes times, but only
    #     numberOfEpisodes // self.number_of_episodes_per_update times
    for episode in range( (numberOfEpisodes //
                           self.number_of_episodes_per_update) ) :
        print("Starting episode {}/{}.".format(
              (episode * self.number_of_episodes_per_update), numberOfEpisodes))
        # a list of all non-discounted rewards
        all_states_in_one_episode = []
        all_actions_in_one_episode = []
        all_rewards_in_one_episode = []

        # a list of gradients saved at each step of each episode
        all_gradients_in_one_episode = []


        for simulationEpisode in range(self.number_of_episodes_per_update):
            # create list for the rewards of the current simulationEpisode
            rewards_for_emulation_run = []
            actions_for_emulation_run = []
            states_for_emulation_run  = []
            # list of gradients form the current episode
            gradients_for_emulation_run = []

            observation = self.environment.reset()

            # generate 1 trajectory
            for timeStep in range(numberOfTimesteps):
                states_for_emulation_run.append(observation)
                all_states_in_one_episode.append(observation)
```

```
            actionValue, gradientValue = sess.run(
                [self.action, self.gradients],
                feed_dict={self.X: observation.reshape(1,
                                    self.numberOfInputs)}
            )
            observation, reward, done, _ = self.environment.step(
                                        actionValue[0][0])
            actions_for_emulation_run.append(actionValue)
            rewards_for_emulation_run.append(reward)
            gradients_for_emulation_run.append(gradientValue)
            # once the done flag is true, we have one completed trajectory
            #    so we can break the loop and record the
            #    number of timeSteps it took
            #    to generate the episode
            if done:
                lastHundredScores.appendleft(timeStep)
                break

        all_actions_in_one_episode.append(actions_for_emulation_run)
        all_rewards_in_one_episode.append(rewards_for_emulation_run)
        all_gradients_in_one_episode.append(gradients_for_emulation_run)

    trajectory = {"state" : np.array(states_for_emulation_run),
                  "reward": np.array(rewards_for_emulation_run),
                  "action": np.array(actions_for_emulation_run)
    }
    trajectories.append(trajectory)
```

Once you have run several episodes, compute each action's score. If an action's score is positive, it means that the action was good and you want to apply the gradients computed earlier to make the action even more likely to be chosen in the future. If an action's score is negative, it means that the action was bad and you want to apply the opposite gradients to make this action slightly less likely in the future. Compute the mean of all the resulting gradient vectors and use it to perform a gradient ascent step.

```
    # first, the obtained rewards are discounted
    all_rewards_in_one_episode = self.discount_and_normalize_rewards(
        all_rewards_in_one_episode, self.discount_rate)
    feed_dict = {}

    for variableIndex, gradientPlaceholder in enumerate(
            self.tweakedGradientPlaceholders):
        # multiply the gradients by the action scores and compute the mean
        meanGradients = np.mean(
            [reward * all_gradients_in_one_episode[
                            episodeIndex][step][variableIndex]
             for episodeIndex, rewards in enumerate(
                                all_rewards_in_one_episode)
             for step, reward in enumerate(rewards)],
            axis=0
        )
        feed_dict[gradientPlaceholder] = meanGradients

    sess.run(self.trainingOperation, feed_dict=feed_dict)
```

```
currentAverageScore = np.mean(lastHundredScores)


if currentAverageScore >= 195:
    print("----------------------------------------------------------------")
    print("Done. Needed {} episodes to solve the environment.".format(
        ((episode+1) * self.number_of_episodes_per_update)))
    print("----------------------------------------------------------------")
    break

returns = [trajectory["reward"].sum() for trajectory in trajectories]
episode_lengths = [
    len(trajectory["reward"]) for trajectory in trajectories]

# return the current episode,
# which is the episode the environment was solved with
return (episode + 1) * self.number_of_episodes_per_update
```