Assignment 2: Solve Bounded Knapsack problem using deep reinforcement learning framework

1BM120

COMPUTATIONAL INTELLIGENCE

Group 3:

Name	Student ID
J. de Boer	1496166
J.J. Sassen	1018639
N.C.M. van de Ven	0948317
M.H.M. van der Meijden	1239639
E. Azoum	0860047

1st June 2021

Contents

1 Environment **1**

2 Learning algorithm **1**
2.1 Choice of learning algorithm 1
2.2 Benefits of using a Deep Q Network 2
2.3 Model Architecture 2

3 Hyper-parameters **3**
3.1 Hyper-parameter Epsilon 3
3.2 Hyper-parameter for Learning Rate 3
3.3 Hyper-parameter for the Discount Factor 4
3.4 Added hyper-parameters 4

4 Agent's training **5**

A Appendix **6**
A.1 Additional Parameters 9

References **10**

1 Environment

The Bounded Knapsack problem has an environment of 200 different items. For every item, its weight, value and limit are known. Since the knapsack can only carry a specified amount of weight (i.e. the maximum capacity of the Knapsack), an agent needs to make choices on which items to pick. The goal of the Bounded Knapsack problem is to optimize the value of the items in the knapsack while still satisfying the knapsack's capacity constraint. The items for this specific problem are imported from the OR_GYM library, and their characteristics are stored in three different vectors. Every vector represents an item's weight, value and limit for which all indices in the separate vectors indicate the same item.
Prior to starting the problem analysis, it is interesting to determine whether all items have approximately the same ratio between value and weight. When looking at figure 1 the environment consists out of a population of uniformly distributed data points which makes it suitable for our problem approach. Furthermore, only one outlier can be detected with a relatively high weight in contrast to its value. It is highly unlikely that this item will be chosen using the deep reinforcement learning method.

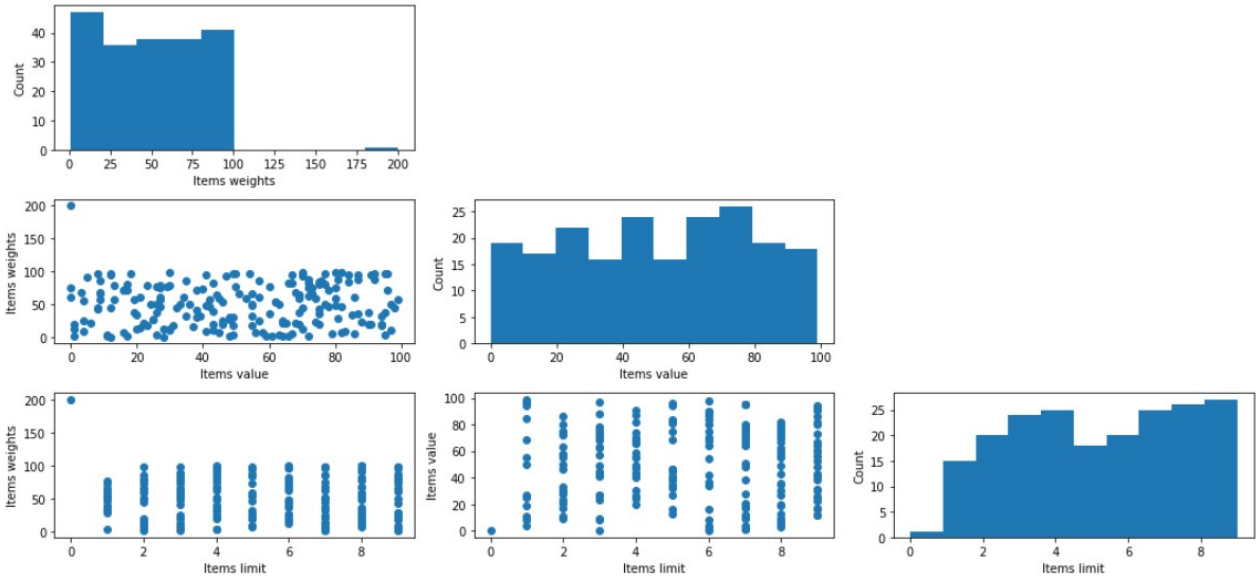

Figure 1: Overview of relations between item characteristics

2 Learning algorithm

2.1 Choice of learning algorithm

In order to solve the Bounded Knapsack problem, the choice was made to use a Deep Q Network (DQN). This algorithm came out on top after extensive research on the following deep reinforcement learning algorithms: Deep Q Network (DQN), REINFORCE, Proximal Policy Optimization (PPO) and Deep Deterministic Policy Gradient (DDPG). Based on the amount of available information on DQN and REINFORCE, these two algorithms were selected for further research. While doing research on both algorithms it became clear that DQN is mostly used for value-based solutions and REINFORCE focuses more on a policy-gradient based approach. The benefit of using a value-based approach is that it constantly updates its winning move after each iteration. With regards to the Knapsack problem, this implies that after choosing one item it recalculates the best configuration, while a policy-based approach would update after filling the whole knapsack.

2.2 Benefits of using a Deep Q Network

Deep Q Learning continues on the basis of normal Q-Learning with respect to gaining rewards based on an action following a policy π. However, the actions in Q-Learning are determined based on a Q-table whereas the actions of a DQN are determined in its neural network. Furthermore, the feature of "experienced replay" and the "epsilon greedy policy" are important properties of a DQN. The experienced replay stores the current state, action, reward and next state in a separate buffer. Subsequently, it combines multiple stored steps into one learning step. This approach secures that the updates of the neural network will be more stable, and it prevents outliers from having a big impact on the network. However, in order to prevent the network from being stuck in a local optima when looping over the same values in the replay buffer, it is crucial for the model to explore different options. The Epsilon Greedy Policy helps the neural network to start exploring multiple options and afterwards slowly incorporate these best findings. This process is called "exploration" and happens more frequently towards the end of the training.

2.3 Model Architecture

Figure 2 shows the architecture of the neural network. As can be seen, the model consists of four layers: one input layer, two hidden layers and one output layer. The input layer consists of 200 neurons, each representing the input of all 200 items. This layer is followed by two input layers with respectively 600 and 400 neurons. While designing the neural network, the idea was developed that the first hidden layer represents whether an item should be placed in the knapsack or that it should hold off for a moment. Subsequently, the second hidden layer decides when to incorporate the given item in the knapsack. In the last layer, the output layer, the amount of neurons was set to 200. Each neuron represents an item of the environment and the one with the highest Q-value will be picked as the next item to be put in the knapsack. However, in order to train the network the weights need to be adjusted based on the processed data. The loss function helps adjusting these weights by minimizing the prediction error of the neural network. As loss function the Mean Squared Error (MSE) method is used which implies that bigger errors are relatively harder punished. Finally, as activation function the choice was made to incorporate "relu" which implies that neurons will be activated if the input value is greater than 0. As a result less neurons will be activated and therefore this activation function makes the neural network more efficient and reduces running time.

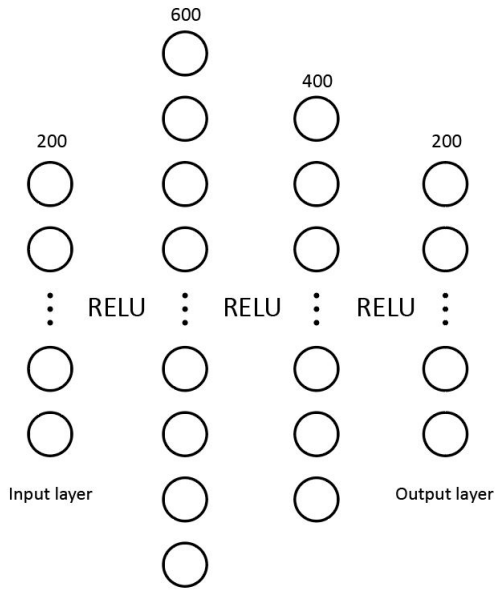

Figure 2: Architecture of the Neural Network

3 Hyper-parameters

In this section, the determination of the hyper-parameters of the algorithm is discussed by giving a brief explanation about each of the parameters. Moreover, a measurement of different parameter values is performed in order to compare the impact on the model's performance. For a start, representative parameter values were chosen from sources [1], [2] and by looking at the parameters of several examples. The results for the algorithm are displayed using several plots.

3.1 Hyper-parameter Epsilon

The hyper-parameter epsilon, ϵ, signifies the ratio between exploration and exploitation. Here, ϵ denotes the probability of exploration, and $1 - \epsilon$ the probability of exploitation. The reason for using an Epsilon-greedy policy instead of simply using a greedy policy is mainly based on the issues related to finding the local optimum. While a greedy algorithm can quickly find a solution, it might not be the optimal solution. By exploring the state-space, this can be prevented. The exploration is enacted by choosing a random action.

Because a solution needs to be reached eventually, it would be unwise to keep exploring endlessly. This approach would not result in a solution. Thus, a decay rate is introduced. By using decay rates, it is possible to make epsilon smaller after each iteration, resulting in a larger probability of the greedy algorithm being used. This effect would mean that in the start, a lot of exploring takes place, while exploitation is done mostly at the end. An example of using this method is shown in the Appendix

Now, for the given algorithm, the epsilon with the best decay parameter needs to be found. This process is performed by comparing the average results over the last 100 episodes, with different decay rates. The results are shown in the Appendix, Figure 6.

As can be inferred from the graph, a decay rate of 0.01 yields the best results. Since the problem is considered to be solved at an average reward of 1600, we looked at the settings that require the least number of episodes to reach this reward value. Using a decay rate of 0.01, the minimum reward value to solve the problem is reached the quickest. The lower decay rates all need more episodes to reach an average reward of 1600. Thus, it can be concluded that for this Knapsack problem, fairly limited exploration is necessary to find a solution.

3.2 Hyper-parameter for Learning Rate

The hyper-parameter for the Learning Rate, η, signifies the rate at which the weights in the network are updated, effectively being the rate at which the network learns. The formulation is shown in Equation 1, where w denotes the new weights, and w_e is the estimated weight error.

$$w = \eta w_e \tag{1}$$

The purpose of using the learning rate η is related to divergence. An example is shown in the Appendix. In short, larger learning rates do not converge, while small learning rates only converge slowly.
Now, for the given algorithm, the best learning rate needs to be found. This goal is accomplished by measuring different learning rates. Usually, Learning Rates are about 0.1 [1]. Thus, learning rates of 0.05, 0.1, 0.2 and 0.4 were chosen to show the impact of the learning rates. The results are shown in Figure 8 in the Appendix.

From Figure 8, when looking at the algorithm reaching the target value of 1600 first, it can be concluded that no clear distinction between given learning rates can be made. Because after reaching 1600 average reward the learning rate of 0.4 increases the most, is chosen that this rate will be used as the best parameter.

3.3 Hyper-parameter for the Discount Factor

The hyper-parameter for the Discount Factor, γ, signifies the long term reward falloff. This factor generally controls how much future reward is valued against intermediate reward. An example is shown in the Appendix.

For the given algorithm, the best Discount Factor needs to be found. This task is performed by measuring different Discount Factors. Usually, Discount Factors are between 0.9 and 0.99. [2] Thus, the values of 0.99, 0.95, 0.9 and 0.8 were chosen to compare the impact. The results are shown in Figure 10 in the Appendix.

It is shown that a discount rate of 0.9 and 0.8 both perform slightly better than the values of 0.99 and 0.95. Because the discount rate of 0.9 reached the goal of 1600 faster, it is determined that this discount rate is the best parameter value.

3.4 Added hyper-parameters

After some testing with the algorithm using the previously given hyper-parameters, it was concluded that the optimum was still hard to find. In order to improve the algorithm, an extra function was added. This extra function uses two hyper-parameters, namely a high-percentage and a reward factor. This is displayed further in Figure 3

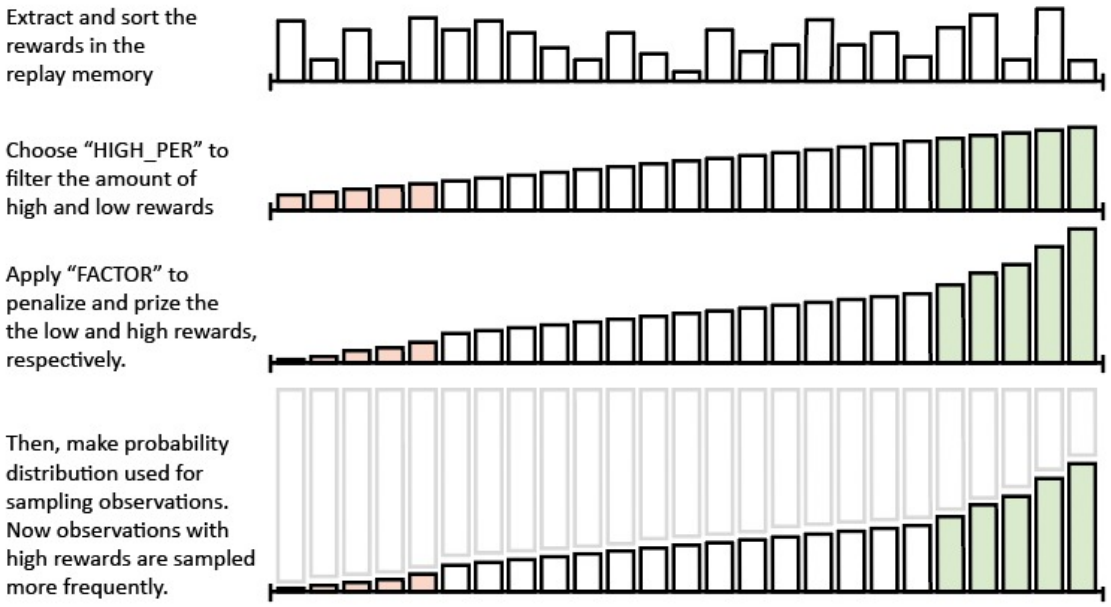

Figure 3: An example of the behaviour using the high-percentages and a factor

In order to find the best values for given hyper-parameters, these were also tested. The results are shown in Figure 11 in the Appendix. The best option, reaching the target of 1600 first, is with hyper-parameters high percentage = 20 and factor = 1.5. This option will be seen as the best of the four options. It can be concluded however that there is no option that performs significantly worse.

After testing the different options for all hyper-parameters can be concluded that the optimal within the experiments are equal to a decay rate of 0.01, learning rate of 0.4, discount factor of 0.9 and the added parameters of 20 and 1.5 for the high percentage and factor respectively.

4 Agent's training

The tests in the previous section highlighted the impact of the decay rate, learning rate, and discount factor on the model's performance, which resulted in parameters of 0.01, 0.4, and 0.9 respectively. Using this configuration, we were able to improve the agent's training, which resulted in the plot as shown in Figure 4. After 452 episodes, the algorithm surpasses an average accumulated reward of 1600 (marked in orange), meaning that the network yields satisfactory solutions to the Knapsack problem after approximately 350 episodes already. Moreover, the maximum average accumulated reward is reached towards the end and equals an average of 2317.69. This value is close to the maximum of 2600. Since the acquired value of 2317.69 is an accumulated average of the last 100 episodes, it can be inferred that the individual episodes towards the very end have reached values in proximity to the maximum height of 2600, and, thus, the agent's training has been maximized.

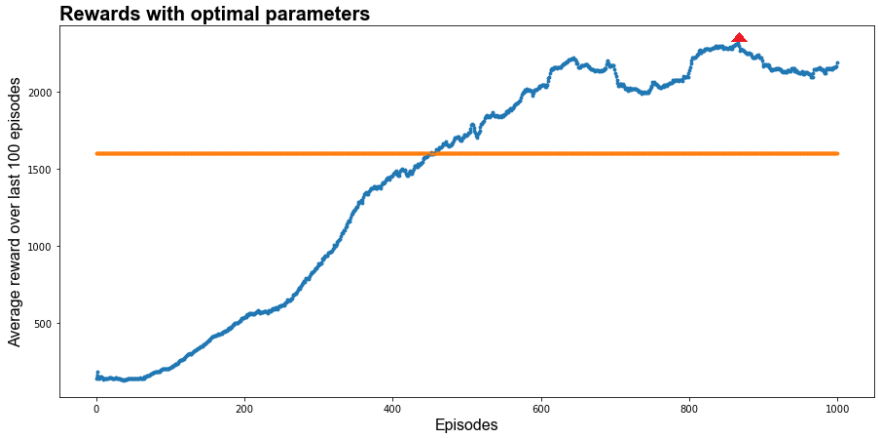

Figure 4: Number of episodes vs average accumulated reward

The DQN has proven to be an efficient and remarkably industrious algorithm for solving the Bounded Knapsack problem. In just over 1000 episodes, acceptable and high quality solutions were produced. The maximum solution is set at 2600, and it was found that the highest reward yielded by the network was partly dependent on the seed set at the beginning. However, as evidenced by previous tests, finding the right parameter values is paramount for successful training of the agent. Reducing the decay rate forces the agent to decrease exploration and thus find higher rewards in significantly quicker succession. The learning rate and discount factor are presently influential to a smaller degree, but applying the right values helps finding the minimum solution more easily. All in all, with the right configuration, the Knapsack problem can be solved within an hour with the use of a DQN.

A Appendix

Parameter for Decay Rate

An example of a method of decay is shown in Figure 5 Here, the starting epsilon is not completely one, and the final epsilon is not completely zero. The falloff from steps 0-600 is determined by the decay rate. Note how the magnitude of the decay rate is dependent on the amount of steps that need to be taken. A lot of steps mean that the decay rate can be smaller. While this method is linear, the method used in the report is exponential.

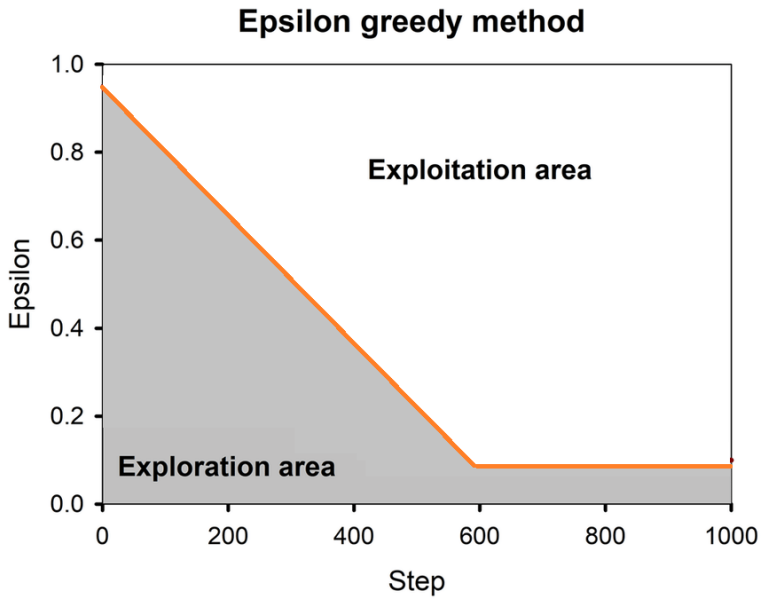

Figure 5: An example exploration versus exploitation space

The results found by testing for different decay rates are displayed in Figure 6

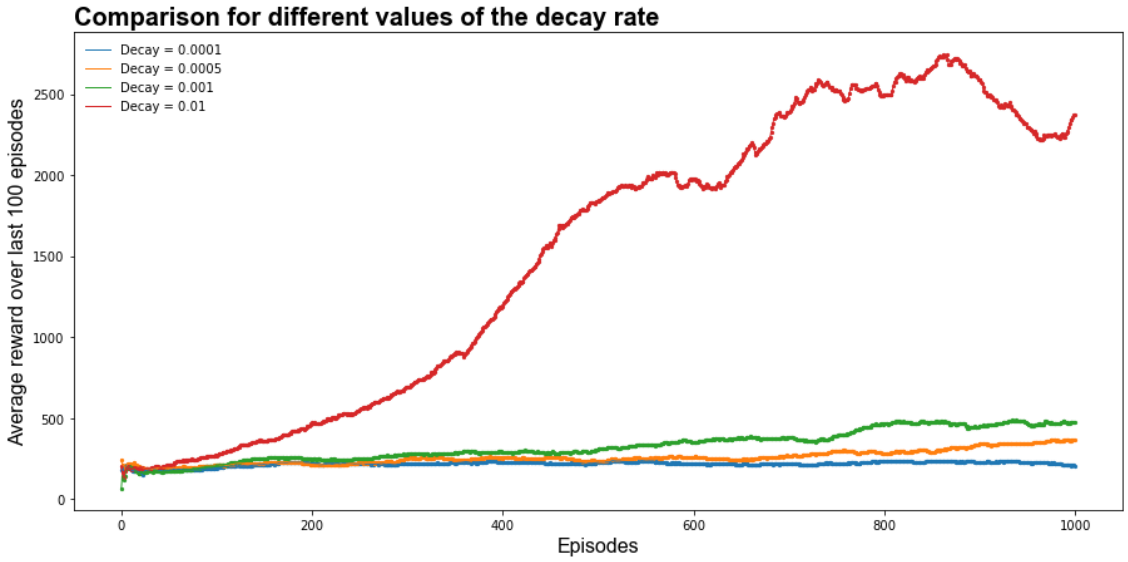

Figure 6: Comparison with decay rates 0.0001, 0.0005, 0.001 and 0.01

Parameter for Learning Rate

An example of divergence and convergence due to learning rate is shown in Figure 7. Here, large learning values diverge. This outcome is mainly due to the method used for deriving the weight error. This method is not completely accurate. For small learning rates, the method does converge. However, the steps to do so will be very large. A good learning rate, therefore, will not diverge, and is still able to find an optimum quickly. The results found by testing for different learning rates are displayed in Figure 8

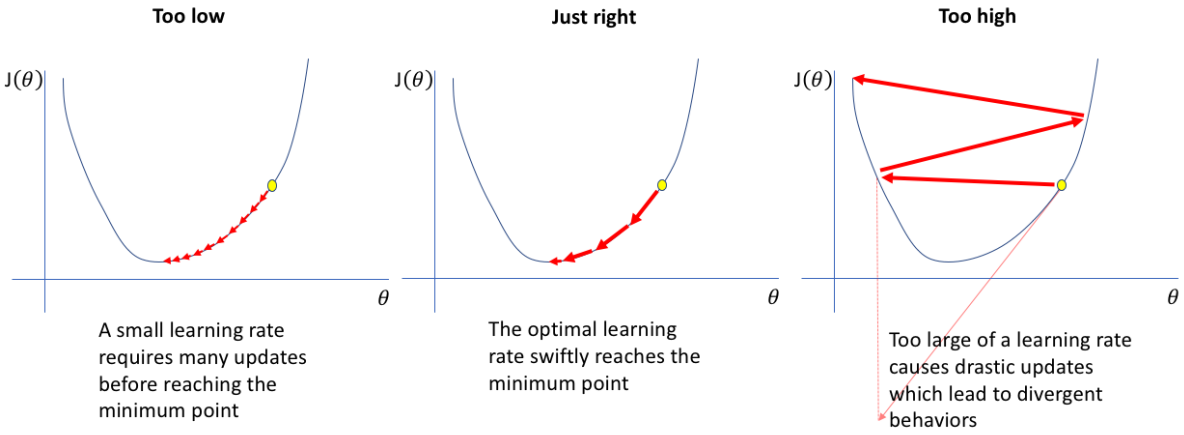

Figure 7: An example of different learning rates used to converge to a solution

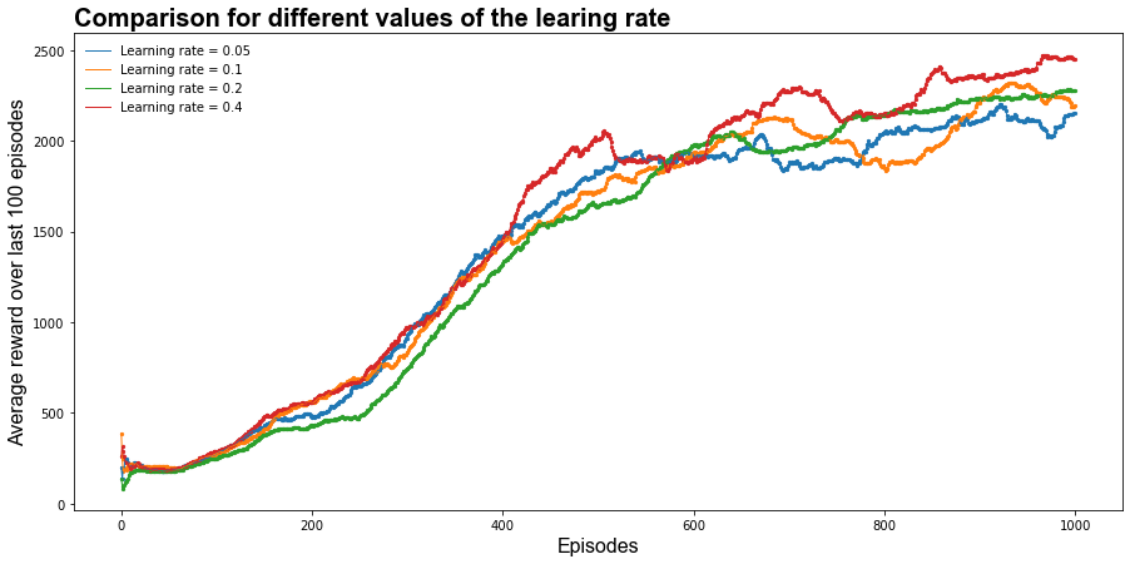

Figure 8: A comparison with different learning rates 0.05, 0.1, 0.2 and 0.4

Parameter for Discount Rate

An example of the effects of the discount rate is shown in Figure 9. Here, the discount factor contributes to the direct and future rewards. The results found by testing for different discount rates are displayed in Figure

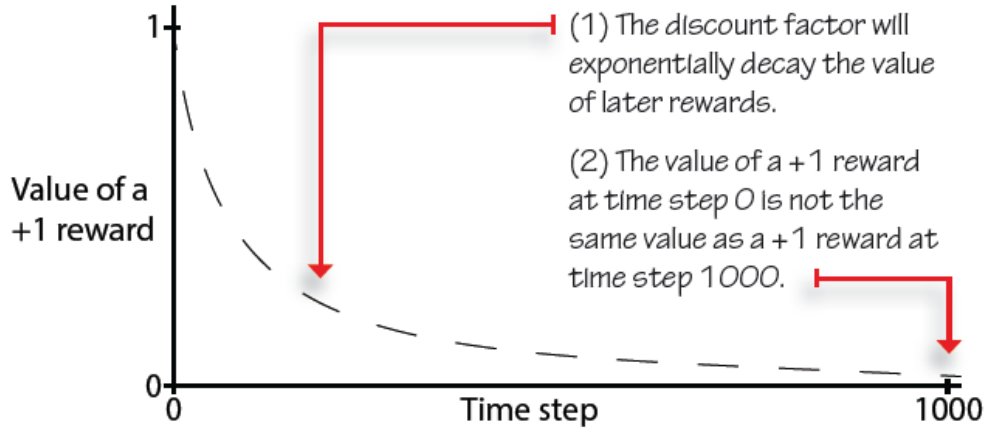

Figure 9: An example of the behaviour with a discount hyperparameter

10

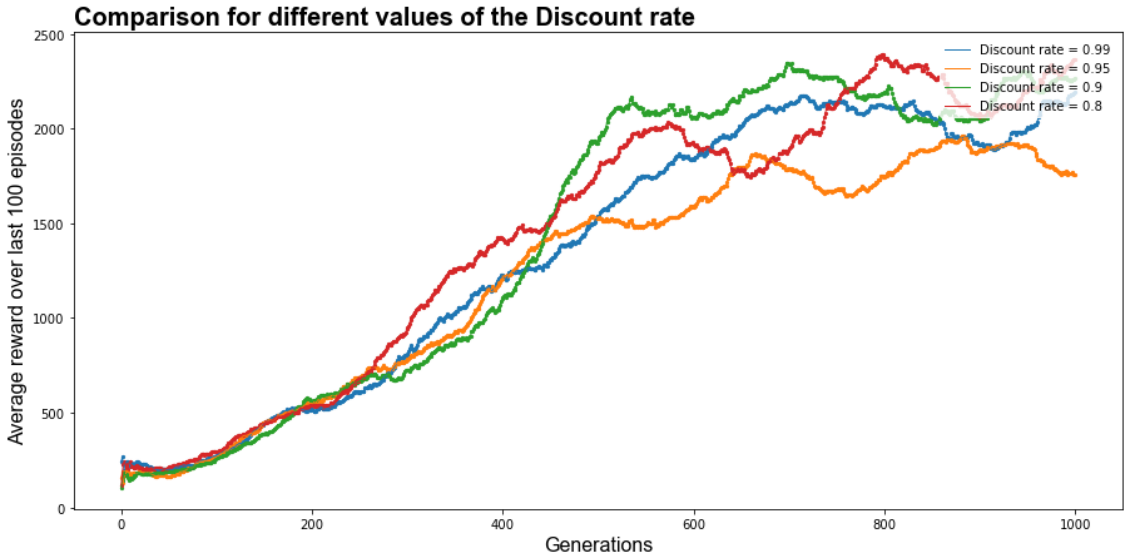

Figure 10: Comparison with a discount factor of 0.99, 0.95, 0.9 and 0.8

A.1 Additional Parameters

The results found by testing for high percentage and added reward are displayed in Figure 11

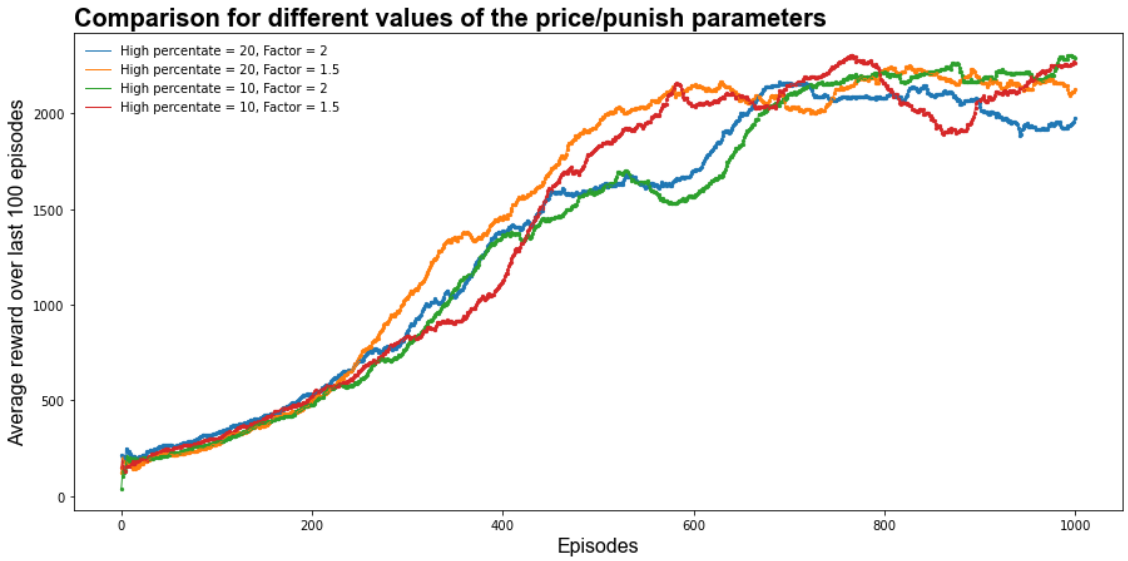

Figure 11: Comparison with (High percentage, reward factor) (20,2), (20,1.5), (10,2) and (10,1.5)

References

Brownlee, J. (n.d.). *How to configure the learning rate when training deep learning neural networks.* Retrieved from https://machinelearningmastery.com/learning-rate-for-deep-learning-neural-networks/

Theiler, S. (n.d.). *Building a powerful dqn in tensorflow 2.0 (explanation & tutorial).* Retrieved from https://medium.com/analytics-vidhya/building-a-powerful-dqn-in-tensorflow-2-0-explanation-tutorial-d48ea8f3177a