Lunar Network

Neural Networks - Assignment 3

May 16, 2018

Abstract

Neural Network implementation gets more and more developed while high complexity of data types becomes the norm nowadays. Many different approaches have been implemented in the field of Deep-Learning. Various network types range from simple one layer perceptron and multilayer convolutional networks, to recurrent networks and deep Q-learning networks (DQN). Some of these approaches, and especially the latter, triggered the present analysis, thus it is centered on the concept of reinforcement learning implemented on a game named Lunar Lander. The following sheet is an endeavor of addressing some ideas and experimentations about the implementation of training two algorithms on the Lunar Lander game.

1 Game: OpenAI LunarLander-v2

Lunar-Lander concept belongs to the supervised learning field and is provided for training and experimentation through OpenAI library. The aim of the specific game is to land the spaceship on its landing pad by navigating it through the space environment. However, the peculiarity of this kind of problems is the absence of labels and data in the forms in which we are used to. The spaceship appears from above and has to land at zero speed on the landing pad (which is centered at coordinates (x,y) = (0,0)). This can be achieved by firing the engines of the spaceship (given that we are provided with infinite fuel). Firing the main engine yields a reward of -0.3 per time step; the other engines can be fired for free. Moving from the top of the screen to the landing pad yields a reward of 100 to 140 points. If the spaceship is landed(either inside or outside the landing pad), the corresponding reward is 100 points. If the spaceship gets crashed at any set of coordinates, the corresponding reward is -100. Moreover, additional bonuses apply for each leg that is on the ground. The environment is considered solved if you get a total reward of at least 200.

Therefore, the agent acts upon observations formed from the environment, and depending on the result of its action it receives the corresponding reward. The possible actions are four: do nothing, fire left orientation engine, fire right orientation engine and fire main engine. Each time the agent chooses an action it gains a negative or positive reward measured in points provided by the environment. So, it implies that the reward increases, as the agent chooses right actions, which efficiently lead it in the landing pad. Finally, the episode ends when the agent lands, crashes or flies out of the space environment.

Each one of the states generated by the environment are represented by a 1×8 vector.

$$State = (x, y, \frac{\partial x}{\partial t}, \frac{\partial y}{\partial t}, \theta, \frac{20\partial\theta}{\partial t}, \iota_{leg1}, \iota_{leg2}) \tag{1}$$

In the above vector, x, y values represent the coordinates of each phase of the agent and the $\frac{\partial x}{\partial t}, \frac{\partial y}{\partial t}$ represents the velocity of each coordinate respectively, in the specific dimensions. θ represents the angle of the spaceship and $\frac{20\partial\theta}{\partial t}$ is the angular velocity multiplied by 20. The two final elements are Booleans which indicate whether the first or the second leg of the spaceship touches the ground. So each time-stamp of the game is represented by a vector-state, like the above, and a reward which is earned by the agent's action. According to all these information, taken from Lunar-Lander, the goal is to build an algorithm that leads the agent to choose the right sequence of actions in order to gain more rewards, and land the spaceship at the landing pad with zero speed.

2 Naive Approach

The first approach that was implemented, was based on the idea of a simple neural network which classifies the observations(state) to an action, guided from the previous rewards. By including rewards inside the

loss function, the network tries to optimize and predict the optimal actions. This is a quite naive method called reward-guided-loss-function.

2.1 Network Architecture

As stated above, the first method concerned a simple multilayer, fully connected, network used for classification. The initial goal of this approach, was to inspect whether the agent can be trained to win the game with a pretty simple multilayer neural network, based on reward-guided loss function. Consequently, this means that the network is trying to optimize a loss function which contains the rewards, in a way that the weights are trained in order to predict actions that produce better rewards. *Lunar-Lander* game does not consist of a dataset thus the data are created from the observations collected from the agent (on the fly). In other words, the input(observation-state) of the network consists of an eight-length vector that represents the values which are observed from the *environment* and generated by the selected action (section 1).

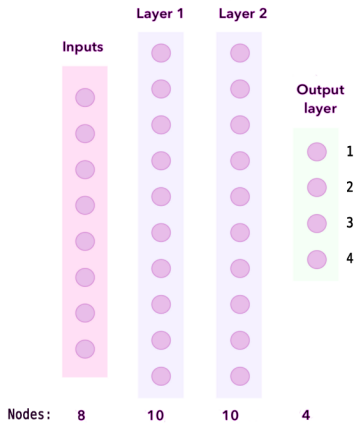


Figure 1: *Network Architecture*

The architecture of the network contains two fully connected hidden layers with 10 nodes each and an output layer with four nodes, which refer to the four possible actions. Both hidden layers are activated through relu function, thus it is able to "deactivate" nodes with negative values by setting them equal to zero, allowing for better decision making. As for the output layer, it is activated via softmax function that produces a probability density distribution which leads to the chosen action.

2.2 Loss Function

Landing the agent is accomplished through a neural network whose loss function is based on the rewards that the agent receives at each time point in a whole episode. Data are collected on-the-fly for each episode and therefore, are stored for the training phase after some pre-processing. The idea is to minimize the loss function which is combined with a discounted version of the rewards. We define a reward, r, as discounted using the formula:

$$disc_i = \gamma \cdot disc + reward_i \tag{2}$$

$$normalization : \frac{disc_i - \mu_{disc}}{\sqrt{\sigma^2_{disc}}} \tag{3}$$

Note that, the above equations take place in an iterative procedure. The initial value of *disc* is 0. So, after the data have been collected and stored at the end of an episode, the rewards are preprocessed using the above procedure. Afterwards, the vector with the truncated rewards has to be normalized in equation (3), in order to reduce variance and gain better results in the loss function.

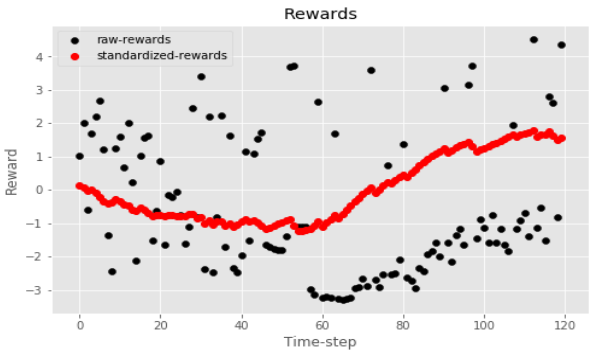


Figure 2: *Raw-rewards Vs standardized rewards*

The above plot visualizes the raw-rewards taken from the environment, in 120 time-steps versus the discounted rewards. It can easily be observed, that by implementing the previous equations, the reward variance is reduced and a smoother distribution is achieved. The implementation of this procedure is inspired from the idea of Advantage Functions, which are in fact improved versions of loss functions that yield more accurate results. Therefore, the loss function should reflect the cost of each "wrong" answer in terms of the corresponding reward. So, taking into consideration the previous statements, we constructed a custom loss function which was plugged into the *Keras* model. Hence, the update of the model parameters is accomplished in a way that the actions that lead to better rewards are more encouraged, in comparison to the ones that lead to a negative reward. The equations below, attempt to define the categorical-crossentropy loss function in order to give an idea of the whole procedure.

$$softmax_j = \frac{e^{a_j}}{\sum_{i=1}^{N} e_k^a} \tag{4}$$

$$crossentropy = -\sum_i y_i log(p_i) \tag{5}$$

Where, p_i in *crossentropy*, is defined from equation (4) and represents the probabilities from the network output through softmax function. By multiplying the crossentropy result with the standardized rewards, obtained from equation (3), and calculate the mean value afterwards, we manage to form the reward guided loss function. Taking into account that $0 \leq p_i \leq 1$, it implies that $log(p_i)$ will always be negative and so, the *Crossentropy* outcome will always be positive.

Algorithm 1: **Loss Function Pseudocode**

```
First preprocess and normalization of discounted rewards
Collect observations and rewards from episode
disc = 0
discounted_rewards = empty_list()
for reward in episode_rewards do
   disc = gamma*disc + reward
   Append: disc in discounted_rewards
M = mean(discounted_rewards)
V = var(discounted_rewards)
Output = (discounted_rewards - M)/V          ▷ Normalized discounted rewards
Then include them in LossFunction
Network_Loss = Categorical_Crossentropy      ▷ Keras-Categorical_Cross_Entropy
temp = Network_Loss * Output
loss = mean(temp)
```

2.3 Training process and Results

Finally, after defining the loss function and the term of discounted rewards, the network will be implemented in an iterative procedure of 5000 episodes. The term *episode* reflects the time period of a single game in which the agent navigates through space, collecting observations from the environment, until the two legs touch the ground or spaceship crashes. It is important to distinguish the episode from the way the agent collects data. In a single episode, the agent collects n observations which will be used in the training process after the end of each episode. This way, since there are 5000 episodes, we will have the same number of training procedures. This is exactly why these kind of problems are interesting, thus the training sets are generated on the fly and there is no prior knowledge or data. It has to be mentioned, that the collected data from each episode are playing the role of the train_set. The network input will be the observations and the target the corresponding actions. The peculiarity of the train_set is, that each episode produces different number of observations, so in the end of each episode, the network will be trained with different sizes of training sets.

As stated in section 1, the goal is to land on the moon without crashing, which in terms of rewards is translated as: reach 200 points at least in one episode. After some episodes, approximately 400, the agent manages to land on the moon and keeps on training, until it reaches a point where it finally lands at most 2 times, on average. Intuitively we know, that if we repeat the process many times, with the final weights plugged in the next set of episodes, the winning rate of the agent will increase. As it is expected, this increase will be accomplished with extreme sluggishness. The following steps describe the training algorithm for each episode.

Algorithm 2: **Training-Algorithm**

```
Get the initial values (observations) of the environment
Feed-forward the network and predict the action
Store the reward and the new_observation for the predicted action
Repeat steps 2-4 until the episode ends
Train the network with the collected-episode-data using the custom loss function
Play the next episode and follow the same procedure for each episode
```

All in all, the above process was used to investigate whether the spaceship could be trained to learn how to fly and consequently how to land itself. In other words, it should be reaching a point where eventually wins the game by earning the reward of 200 in at least one episode. Remember that at each training phase, the input of the network is the observations which were collected, (1×8 state-vector section 1) and the target_y is the corresponding actions which were predicted from the feed-forward step of the network inside each episode. The table below, represents the instances where the agent won for each Game-set(5000 episodes).

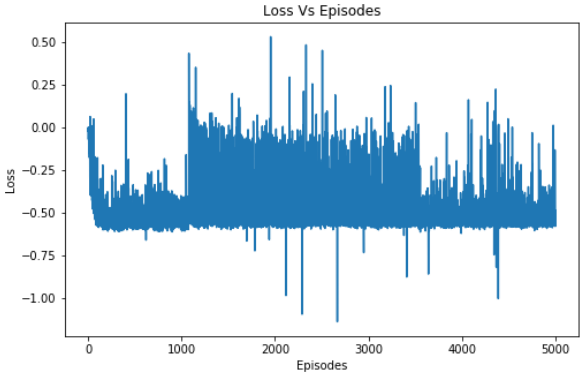


Figure 3: *Loss per Episode*

Game-set	Episode-won
1	0
2	0
3	2
4	4
5	3
6	1
7	0
8	1
9	3
10	2

Table 1: *Winning times per Game-set*

2.4 Conclusions

As it is observed from table 1 and figure 3, our approach is not stable and the agent does not win the game more than 2 times. Given these results and this specific simulation, of ten game-sets with 5000 episodes each, the agent achieves to win on average 1.6 times. Moreover, the loss value per episode(train) fluctuates a lot and it does not appear to show a growing tendency to win. This occurs due to the implementation of a pretty naive approach, based on a policy-gradient network. Hence, using a reward-guided-loss-function, the agent is pushed to take actions in order to optimize loss function. The whole algorithm leverages on a stochastic process of choosing the next action at random, using the softmax-output probabilities. So, the agent chooses an action using a semi-biased random choice, according to the four softmax probabilities.

2.5 Experimentation

The first try was to experiment with different network configurations, such as various activation functions, different optimizers and various number of layers and nodes. After much experimentation, it was decided that the optimal configuration is the one described in 2.1, thus it yields the maximum number of winning times. Regarding the optimizer, we concluded that *Adam* produced the best results. Note that, *Adam* (adaptive moment estimation) optimizer is a deviation of the classic *Stochastic-Gradient-Descent* combined with *Adaptive-Gradient-Algorithm*(AdaGrad) and *Root-Mean-Square-Propagation*(RMSProp). So, *Adam* leverages on both RMSProp and AdaGrad optimizers and calculates the exponential moving average of the gradient and the squared gradient, too. This way, *Adam* achieves to converge fast and provide robust and efficient results. For more information about *Adam* follow the link

In addition, we made an attempt to include some extra algorithms, like ϵ-*greedy* in order to boost the naive model. *Epsilon-greedy* is a stochastic algorithm which provides the agent with the opportunity to decide completely at random, allowing for more exploration. However, by adding this approach, the model showed slight improvement on the one hand but more instability on the other, due to the stochastic nature of ϵ-*greedy*. That led us in the decision to not include it in the algorithm. The algorithm of ϵ-*greedy* will be explained thoroughly in the next section.

3 Q-learning

Taking into account the low results and the moderate progress of the previous approach, the next step was to extend the structure and the logic of the model into more reinforcement learning fields. Similarly to the previous section, the aim is to obtain the optimal action decision policy which leads the agent to land on the moon and hit a reward of 200, but in a more stable way. The whole idea behind this policy is based on reinforcement learning, thus we seek for the sequence of actions formed from the environment, which will maximize the rewards of the game. This process is improving through trial and error by a stochastic path of actions at time points t.

3.1 Q-Learning Basics

The Q-learning algorithm attempts to learn the value of being in a given state, s, and take a specific action at that state. This means that when the agent is located on a current position, it chooses an action and travels to the next. In the new state s', the agent has to select the action with the highest value and iteratively update the following equation:

$$Q(a, s) = (1 - \alpha) \cdot reward(a, s) + \alpha \cdot \gamma \cdot max(Q(a', s')) \tag{6}$$

The above equation is the well-known *Bellman-Equation* used for Q-Learning, where $reward(a, s)$ defines the reward gained from a state s and an action a, γ defines the decay rate and is usually set to a value very close to 1(e.g. γ=0.99). We can think of $Q(a, s)$ as the target value, (y_{target}), which will be the label values of our network, while $max(Q(a', s'))$ defines the maximum value of the four possible actions, predicted by the feed-forward phase for the next-state. The Q-learning process will be plugged into the network-training process in order to progressively update the y_{target} values of the deep-Q-network with the ones that are generated from equation (6). By using this algorithm inside the training process, we iteratively "push" the network to predict actions that retrieve better rewards by optimizing the loss-function which is described extensively in section 4.2. More interested readers can follow this link for more information about Q-learning algorithm or learn about it with an easy example.

3.2 ϵ-greedy Algorithm

The *epsilon-greedy* algorithm is a simple and general approach which controls and manipulates the parameter of randomness. In this experiment, *ϵ-greedy* will lead the *AI-Agent* to explore its environment and choose many actions completely at random. The algorithm is used in our approach in the following way:

Algorithm 3: **ϵ-greedy Pseudocode**

```
Initialize epsilon = 1 ; epsilon_min = 0.01 ; discount = 0.995
for Each time the agent has to choose an action do
   if random[0,1] <= epsilon then        ▷ random[0,1] : random float value in range[0,1]
      action = random_choice(possible actions)
for Each time the episode ends do
   if epsilon >= epsilon_min then
      epsilon = epsilon*discount
```

The above algorithm enriches the whole simulation with controllable randomness, by giving a chance to the agent to choose either a random, or a trained decision for the current action. We control the exploration and exploitation mode of the agent by tuning the ϵ and *discount* values. For further details follow the link.

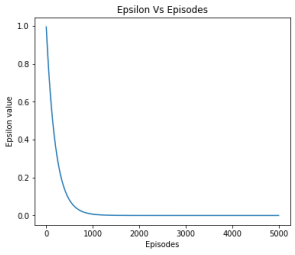


Figure 4: *ϵ=1, dc=0.995*

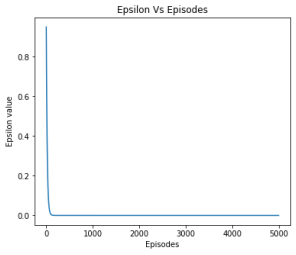
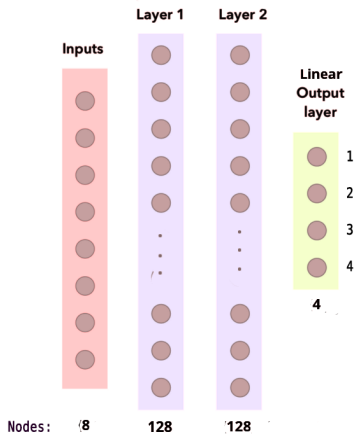


Figure 5: *ϵ=1, dc=0.95*

4 Model with Q-learning approach

4.1 Architecture of the Network

Regarding the network, it consists of 2 hidden layers, with 128 nodes each, activated through relu function again. Moreover, the output layer, contains 4 nodes one for each action a_t, and this time it is activated through linear function, $f(x) = \alpha \cdot x$, instead of softmax, hence it creates an output with just numbers.



The choice of this activation function is not at random, thus linear activation produces a format in which the highest number indicates the chosen action which is needed for the definition of the loss function. This output format will be used in the mean squared error loss function, in a way that it will be explained later on. In Figure 6, the architecture of the network is visualized to create an idea of how the input is processed through the network, leading to the desired output.

Figure 6: *Network of Q-learning model Architecture*

4.2 Loss Function

For the definition of the Mean Squared Error loss function, it is crucial to list some terms from Q-learning, mentioned in section 3.1, which will be plugged in the function and adjust its interpretation to the needs of the Lunar Lander game. First, we list the following parameters defined in equation (6), for further explanation:

- decay rate: γ
- learning rate: α

Given a state s_t, an action a_t and a reward r_t at a time point t, we have a number of parameters that has significant contribution in the model. Decay rate γ is a digit ranging from 0 to 1, and its role is to manipulate the way that the agent considers the future rewards. If γ is closer to zero, the agent will tend to consider only immediate rewards. If γ is closer to one, the agent will consider future rewards with greater weight. In addition, a more familiar parameter is α, the q-learning rate which stands for the amount of speed that the agent forgets previous states. The equation below, describes the network loss function.

$$loss = (Y_{target} - \hat{Y}_{predicted})^2 \tag{7}$$

$$loss = [\underbrace{(1-\alpha) \cdot r + \alpha \cdot \gamma \cdot \max_{a'}(Q(s', a'))}_{target} - \underbrace{\hat{Q}(s, a)}_{prediction}]^2 \tag{8}$$

It can easily be observed that the above equation is the standard mean-squared-error, where $\hat{Q}(s, a)$ is the predicted value of the network. The loss function above, is used for regression problems and it generally measures the averaged squared distance between the target value and the predicted outcome. In this task, concerning the **target** part in equation (8), it is translated as the $(1 - \alpha)\%$ of the current state's reward r, plus the $\alpha\%$ of the discounted maximum reward of the next state s' given the chosen action a'. The purpose of this term, is to represent a combination of two weighted rewards. On the one hand, the weighted current reward, and on the other hand the weighted maximum future reward. Regarding the **prediction** part, $\hat{Q}(s, a)$ represents the predicted reward-value from the network output. It contains four different values, each one $\in \mathbb{R}$, and the maximum one indicates the preferred action. Combining all the above, the aim is to reduce the deviation of the predicted values from the potential rewards of the current and the next state. This way, we are confident that the outcome of the network leads to the optimal sequence of actions by training the weights to this direction.

4.3 Training process and Results

The training procedure begins with the same number of episodes as in the naive model, that is 5000. The iterating process is more or less similar to the previous model, except that in this case the term of *batch-size* is introduced in the algorithm as well as the ϵ-*greedy*. One of the differences here, is that the training of the agent contains random sampling from the previously collected observations, actions and rewards(agent-history). As it follows, it is crucial to determine the sample size which is translated to batch size in the neural network theory. We chose a batch size of 16 which will remain stable in the whole training procedure. So,at the end of each episode the network will be trained 16 (*batch-size*) times iteratively inside the *mini-batch* phase, taking as input an observation and as target the output of equation (6) in 3.1. Note that mini-batch technique is memory-preserving and is simultaneously helpful for the network to accomplish better results.

Another value which played a significant role in our experiment is ϵ. According to section 3.2, ϵ is the value which controls the equilibrium between exploration and exploitation in stochastic tasks like this one. More precisely, the value of ϵ is the number which controls whether the actions will be at random (agent in exploration mode) or actions will be predicted by the trained-network (exploitation mode). In other words, the value of ϵ is the bias towards the one mode to the other. Figures 4 & 5 visualize the space of the exploration-opportunity the agent has, but also provide information about the impact of the discount factor which controls the rate of ϵ-value reduction. Therefore, it is preferable to set the agent in exploration mode, in order to collect many different observations by choosing actions at random. As the training escalates, the agent is set to exploitation mode using the previous observations. After some training, when the network is ready to produce robust outcome, it is recommended to set ϵ to a very small value, thus we are confident with the trained network. So, we chose an initial value of $\epsilon = 1$, keeping in mind that it keeps on truncating through the training process, in order to gradually reduce the space of randomness-opportunity. So, in the end, the training process returns the outcome of the network as well as the mean value of the loss function through the episodes, reminding a bit of a Monte Carlo approach.

Algorithm 4: **Training-Algorithm Pseudocode**

```
Initialize:  alpha=0.1; batch-size=16; gamma=0.99
Get the initial values (observation) of the environment
Feed-forward the network, predict the next action or choose at random according to e-
Store:  observation, new_observation, reward and action
Repeat steps 2-4 until the end of the episode
Take a random sample equal to batch-size from the stored data
for random.sample(stored_data, batch-size) do
   target = (1-alpha)*reward + alpha*gamma*max(predict(new_observation))
   temp = predict(observation)
   temp[action] = target
   train.NN_model.fit(x=observation , y=temp)
```

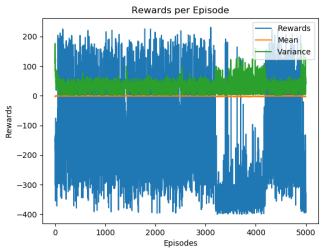


Figure 7: *Sum of rewards game_ 0*

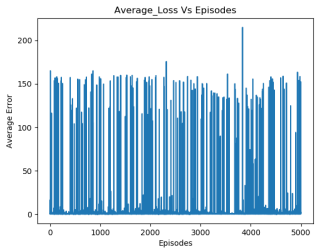


Figure 8: *Average loss game_ 0*

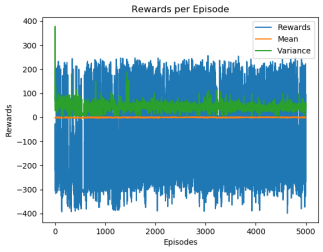


Figure 9: *Sum of rewards final_ game*

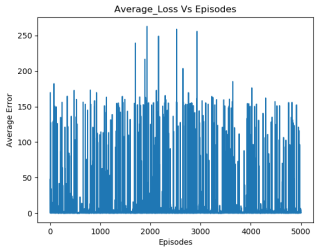


Figure 10: *Average loss final_ game*

The plots on the left side, visualize the total amount of rewards the agent earned in each episode, while on the right side the corresponding average loss of each episode is illustrated. Figures 7 & 8 were produced from the initial training simulation game-set of 5000 episodes without pretrained weights, while 9 & 10 visualize the final game-set. It can easily be observed, that the agent starts with very low winning rate and with high reward-variance per episode, indicated in figure 7. Conversely, figure 9, visualizes the final game-rewards, and is indicative of a lower variance and a stable winning-tendency of the agent. It is also clear that the loss of the final game is on average reduced. That leads to a conclusion that the network, after much training, produces better results on average.

4.4 Conclusions

Tables 2 & 3 illustrate the results of the implementation of this approach on 5 game-sets of 5000 episodes each. The training procedures took place on two different machines, Dell Inspiron(Machine1) and Lenovo Thinkpad(Machine2) using *Keras* and the results are shown in the tables. The evolution of the winning rate is obvious, thus by plugging the aforementioned changes in the algorithm, we obtained the desired results. More specifically, for the first machine the maximum number of wins was 290 while in the second 291. Apart from Q-learning addition in the algorithm, the increase of the wining rates was also achieved by the pretraining procedure which was implemented 4 times in each machine. This procedure is supposed to handle the training weights of the model as a chain-training-process, since the final weights of each game-set will be directly fed to the next one. The results in the tables lead us to conclude, that as the wins of the agent increase, the mean rewards per game-set increase too. This is reasonable, since the agent will receive higher rewards if it chooses the sequences of actions that lead it to the landing pad. Finally, it is observed that in Lenovo machine the number of wins was higher compared to the Dell machine in almost all the pretraining processes. As explained in the next section this occurred due to the fact that the ϵ parameter was tuned differently in each computer.

Pretrain	ϵ	**Wins**	**Mean Reward**
0	1	51	-208.91
1	0.9	135	-204.31
2	0.8	271	-122.15
3	0.7	252	-134.71
4	0.5	290	-131.21

Table 2: *Wins per set Machine1*

Pretrain	ϵ	**Wins**	**Mean Reward**
0	1	51	-208.91
1	0.1	138	-156.54
2	0.1	244	-134.84
3	0.1	291	-125.37
4	0.1	288	-129.05

Table 3: *Wins per set Machine2*

4.5 Experimentation

As stated before, the whole simulation was ran in two different machines, in order to experiment with different configurations. The two machines ran the same network architecture, described in section 4.1 but with different ϵ values. The initial game-set was implemented with $\epsilon = 1$ and $\epsilon_{min} = 0.01$ in both machines. Consequently, for the next simulations, using pretrained-weights from now on, it was decided to allow for limited exploration space ($\epsilon = 0.1$) concerning the first machine, in order to exploit the already pre-trained network. On the contrary, concerning the second machine we allowed for more exploration space, by faintly reducing the ϵ-initial value in each game-set starting from $\epsilon = 1$. After a lot of experimentation regarding the controllable randomness provided by ϵ-*greedy*, we addressed the optimal results in tables: 2 & 3.

The second subject of experimentation, was the different values used for the *Q-learning* algorithm: the q-learning-rate α and the decay rate γ from section 4.2 equation (8). We conclude that the best results were achieved with low learning rate $\alpha = 0.1$ which means that the agent "pays lower attention" at the future rewards and γ value was set to the classic 0.99 q-learning discount.

The last part of the experimentation, revolved around the network architecture. Not only different activation functions, but different number of hidden layers and nodes as well, were implemented and tested. We concluded that the architecture which produced better results was described at section 4.1. It has to be mentioned, that the choice of small number of nodes did not contribute to the predicting ability of our network in actions that would yield high rewards, thus it usually kept leading the agent to crash the spaceship. So, it was decided to use 128 nodes in each hidden layer. In the contrary, adding many hidden layers did not affect significantly the results, leading us to keep only two hidden layers. Finally, different optimization methods were tested as well, but *Adam* optimizer produced again the most efficient results.

5 Final Remarks

All in all, reaching at the end of this analysis, the purpose was to adapt a reinforcement learning problem into the knowledge we have from neural networks. The logic behind the problem was to train an agent, navigating on a space environment, to land on the moon according to the rewards it gets through the selected actions. Hence, as the training procedure unfolds, the weights should be updated in a way that the network produces the actions that lead to the best reward. The key to this update is the definition of loss function in both naive and *Q-learning* approach as well as the definition of some unfamiliar parameters. This way, in the naive approach a custom crossentropy-based loss function was used, while in the second approach we introduced the basic functions of *Q-learning* plugged in a classic mean squared error function. It was speculated that, the first model was more stochastic and more randomized compared to the second, thus the former produced results which were not connected to each other since the actions were chosen at biased-random with probabilities produced from the network. The latter approach differs significantly from the first since we introduced epsilon parameter which tunes the amount of random action-decision making and of course, the *Q-learning* addendum. This way the network yielded better results in terms of number of wins and mean reward values. However, it was less fast than the naive approach due to the complexity of the training process algorithm using $NN.training$ inside the mini-batch phase. Corresponding plots were presented for the results from both models as an additional explanation tool. The results from the pretraining process reassure that the agent has tendency to win as the process keeps on repeating. So, combining the experience from the experimentations in the first approach and the introduction of *Q-learning* algorithm, the training of the agent was successfully accomplished producing increased winning rates.

Lastly, the results of the whole procedure are also presented in a video format, which visualizes the navigation of the agent on its efforts to land, through the space environment. The three different phases of the agent were monitored, and they are available in the next three links:

- **Early trials**
- **Agent learns how to fly**
- **Agent lands itself**

References

[1] *Step by step Q-learning tutorial* : link here

[2] *Felix Yu* : Deep Q Network vs Policy Gradients - An Experiment on VizDoom with Keras

[3] *Patrick Emami* : Deep Deterministic Policy Gradients in TensorFlow

[4] *Sergey Levine* : Policy Gradients Notes

[5] *Volodymyr Mnih et al.*: Human-level control through deep reinforcement learning

[6] *Volodymyr Mnih et al.*: Playing Atari with Deep Reinforcement Learning