**Section 1 – Tabular Q learning** (25%)

1. Why methods such as Value-Iteration **cannot** be implemented in such environments? Write down the main problem. (3%)
2. Methods such as value iteration require us to assess the probability of the next state and a reward given the current state and an action, for example on Atari game given the current screen frame and an action we need to formulate the probability of the reward and the next state, for all possible pairs, which is complicated.

2. How do model-free methods resolve the problem you wrote in previous question? Explain shortly. (2%)

1. Model free methods don’t rely on a model and so don’t have to approximate the real or a good behavior, instead it holds a parametric data structure that describes a behavior, filling the parameters and constantly rechecking and correcting/adjusting until the parameters describe a behavior that is considered “good”.

3.What is the main difference between SARSA and Q-learning algorithms? Explain shortly the meaning of this difference. (3%)

1. in both we chose an action according to Q and take this action, the difference between these two algorithms is that in SARSA we update the Q value according to the best next action (next state best action according to Q) and on SARSA I update the Q according to the Q value of the next real chosen action . SARSA is more stable because the cost is updated in more stochastic manner while in Q-learning the updating is done according to a ”better” value because it is with the higher probability and so is faster, but may be wrong so may affect the result more dramatically along the way (and will be less forgiving to other parameters).
2. Why is it better than acting greedily (choosing an action with )? (2%)
3. When choosing a random action we explore an action that the model wouldn’t have chosen, this action can be better or worse than what the model would do. In case that its better it will enable the algorithm to learn a new better action to the state, if its worse it will worsen the result and may make the model worse. Therefore allowing random action every now and then is good if it doesn’t damage more than it helps. It will also introduce scenarios that the network wouldn’t ar rarely see otherwise. On the training process at the beginning the model is not good so there is more room for exploration, so that there is a greater chance that to random action will perform better then on a trained model and accordingly , so decaying epsilon is applied.

Frozen lake

Hyper parameters:

lr\_alpha = 0.3  
df\_gamma = 0.95  
epsilon = 1.0, epsilon\_decay\_rate = 0.0005

#(epsilon = max(epsilon - epsilon\_decay\_rate, 0)

nof\_episodes = 5000  
max\_nof\_steps = 100  
init\_state\_y = 0  
init\_state\_x = 0  
goal\_state\_y = 3  
goal\_state\_x = 3

Q-value table

After 500 steps:

[[[6.34099088e-03 1.99979637e-03 1.97620075e-03 3.64242355e-05]

[6.07108066e-03 0.00000000e+00 4.82660456e-02 0.00000000e+00]

[2.26668123e-03 1.73646584e-02 0.00000000e+00 0.00000000e+00]

[0.00000000e+00 6.94473750e-03 0.00000000e+00 0.00000000e+00]]

[[5.70560152e-03 3.02491103e-03 6.18149214e-05 5.63217214e-04]

[4.32724305e-03 0.00000000e+00 0.00000000e+00 0.00000000e+00]

[2.55839697e-03 0.00000000e+00 0.00000000e+00 0.00000000e+00]

[0.00000000e+00 0.00000000e+00 3.71790075e-01 0.00000000e+00]]

[[4.76333612e-03 1.55906350e-03 5.27181451e-03 2.28116482e-05]

[2.64067146e-03 0.00000000e+00 2.12347163e-04 0.00000000e+00]

[2.20415093e-03 1.70572500e-02 8.91765000e-02 0.00000000e+00]

[0.00000000e+00 0.00000000e+00 3.00000000e-01 0.00000000e+00]]

[[4.55777192e-03 3.71523205e-03 6.97577680e-05 1.03809071e-05]

[5.00136703e-03 0.00000000e+00 0.00000000e+00 0.00000000e+00]

[1.17156414e-02 2.62228590e-02 0.00000000e+00 0.00000000e+00]

[0.00000000e+00 6.67947375e-02 3.00000000e-01 0.00000000e+00]]]

After 2000 steps:

[[[0.25668736 0.03038334 0.17058078 0.01729755]

[0.31864263 0. 0.00995209 0. ]

[0.06366358 0.11474326 0.48746962 0. ]

[0. 0.19903911 0.31571787 0. ]]

[[0.0488604 0.02239691 0.02329278 0.01719424]

[0.05519101 0. 0.01031591 0. ]

[0.06296122 0.57457047 0.03345 0. ]

[0. 0.19536027 0.79671865 0. ]]

[[0.04944036 0.02646941 0.0248446 0.01433839]

[0.04830692 0. 0.16586938 0. ]

[0.06799078 0.14652726 0.10749834 0. ]

[0. 0.67711403 0.33262665 0. ]]

[[0.04925716 0.2012679 0.02470558 0.01558466]

[0.04636642 0. 0.00930234 0. ]

[0.46946358 0.10813315 0.08348861 0. ]

[0. 0.19599206 0.35157995 0. ]]]

[[[0.25668736 0.03038334 0.17058078 0.01729755]

[0.31864263 0. 0.00995209 0. ]

[0.06366358 0.11474326 0.48746962 0. ]

[0. 0.19903911 0.31571787 0. ]]

[[0.0488604 0.02239691 0.02329278 0.01719424]

[0.05519101 0. 0.01031591 0. ]

[0.06296122 0.57457047 0.03345 0. ]

[0. 0.19536027 0.79671865 0. ]]

[[0.04944036 0.02646941 0.0248446 0.01433839]

[0.04830692 0. 0.16586938 0. ]

[0.06799078 0.14652726 0.10749834 0. ]

[0. 0.67711403 0.33262665 0. ]]

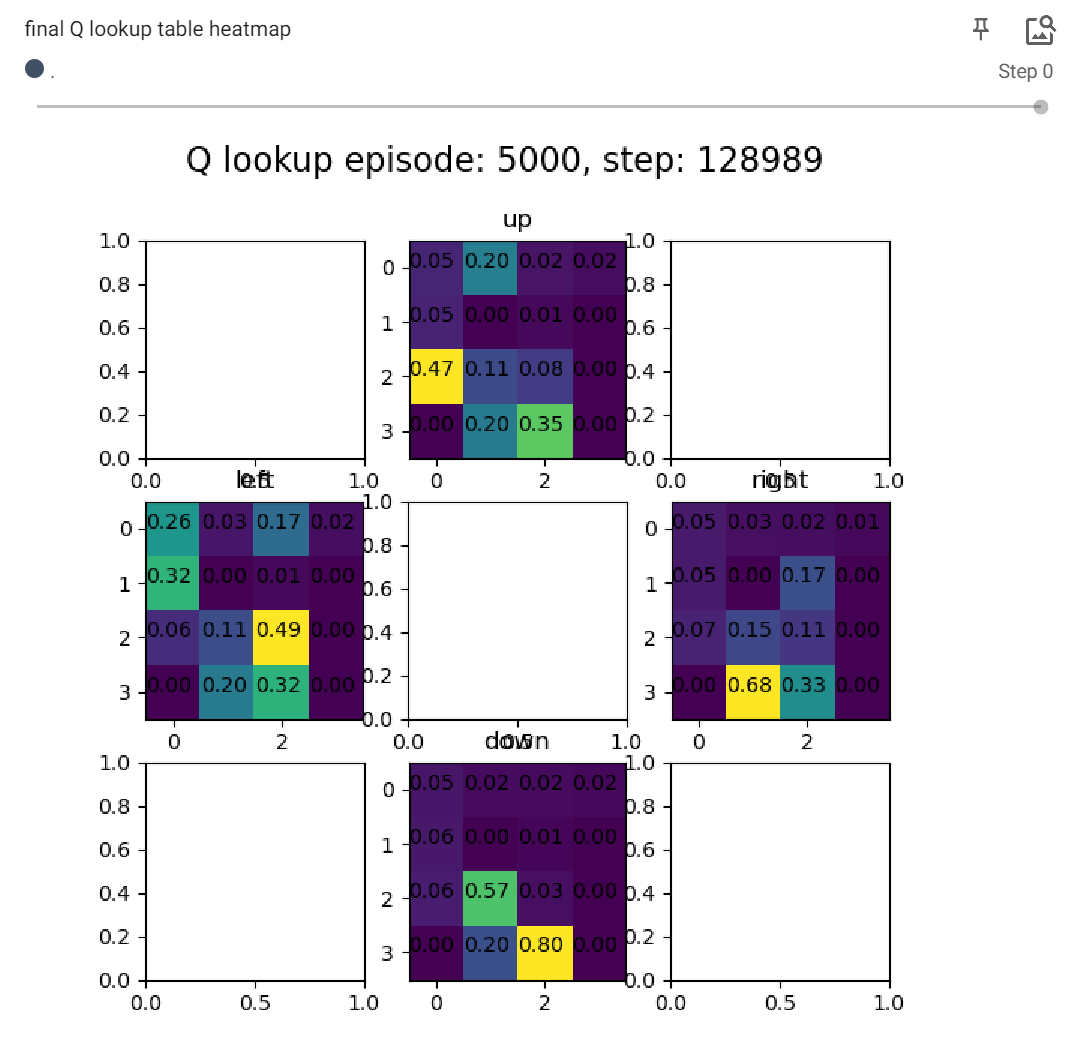
[[0.04925716 0.2012679 0.02470558 0.01558466]

[0.04636642 0. 0.00930234 0. ]

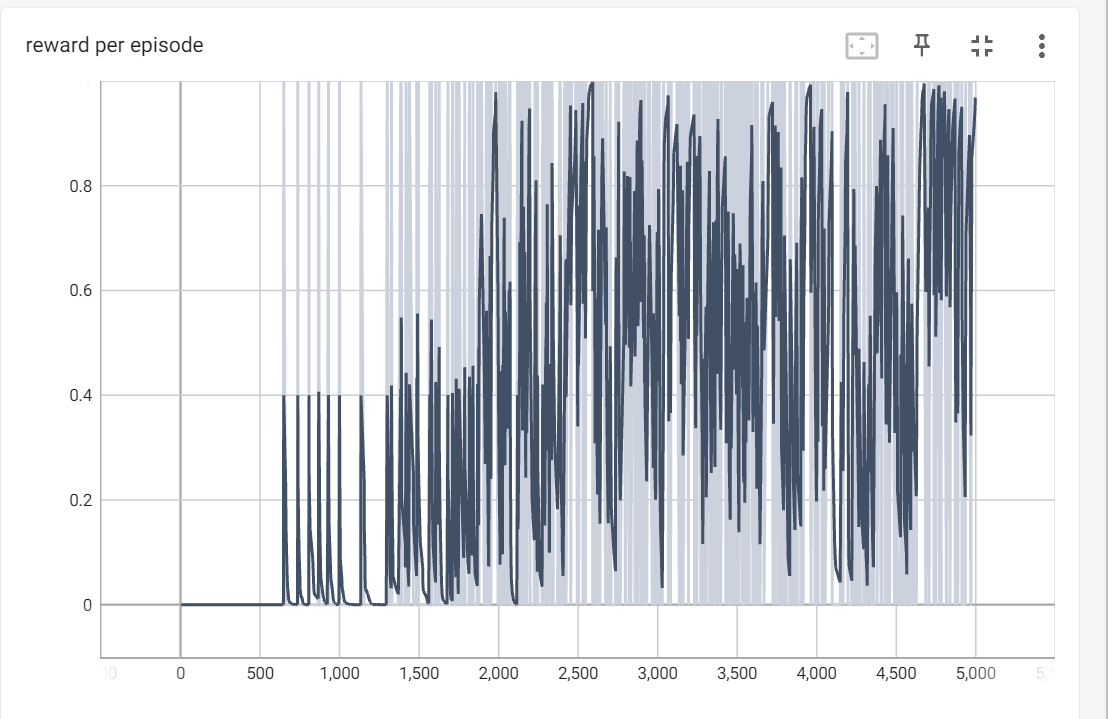
[0.46946358 0.10813315 0.08348861 0. ]

[0. 0.19599206 0.35157995 0. ]]]

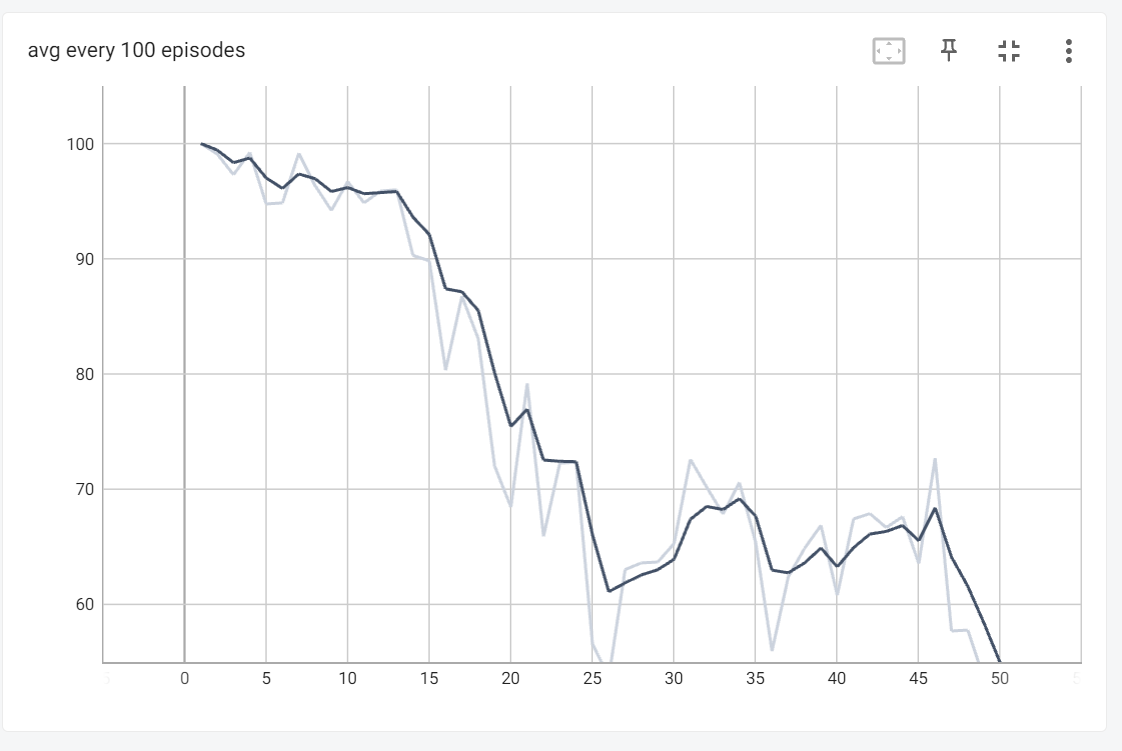
Final Q table:



Plot of the reward per episode:



average number of steps to the goal over last 100 episodes:



**Section 2 – Deep Q-learning**

1. Why do we sample in **random** order? (3%)

Each sample in the training batch associates a state and the outputs regardless of the history, consecutive states however are correlated between themselves because the state didn’t change a lot between them so they will be quite close, if we would use the sets for training on the same order in which the sets were received it will be as is we input the same state over and over again, and also in time it will be the same state that slowly changes, this will cause overfitting and will make the network forget old states.

1. How does this improve the model? (2%)

Every training batch changes the training network a little bit, the change will effect the output to states too. If we would use the same network as the value and target every batch will change the network and the loss of the next batch will make the change of the previous batch irrelevant because the output (target) that it was calculated by is not relevant, in that way we aggregate corrections for the weights from many scenarios and apply them together.

Hyper-parameters

LEARNING\_RATE = 0.007; if not do\_train: LEARNING\_RATE = 0

NUM\_OF\_EPISODES = 5000

NUM\_OF\_STEPS = 10000; if not do\_train: NUM\_OF\_STEPS = 500

REPLAY\_LIMIT = 20000

epsilon\_decay = 0.999

min\_epsilon = 0.01; if not do\_train: min\_epsilon = 0

SAMPLE\_SIZE = 32

GAMMA = 0.95

C = 200

5 layers network

Layer (type) Output Shape Param #

=================================================================

input\_20 (InputLayer) [(None, 4)] 0

dense\_94 (Dense) (None, 256) 1280

dense\_95 (Dense) (None, 128) 32896

dense\_96 (Dense) (None, 64) 8256

dense\_97 (Dense) (None, 32) 2080

dense\_98 (Dense) (None, 16) 528

dense\_99 (Dense) (None, 2) 34

=================================================================

Total params: 45,074

Trainable params: 45,074

Non-trainable params: 0

Training:

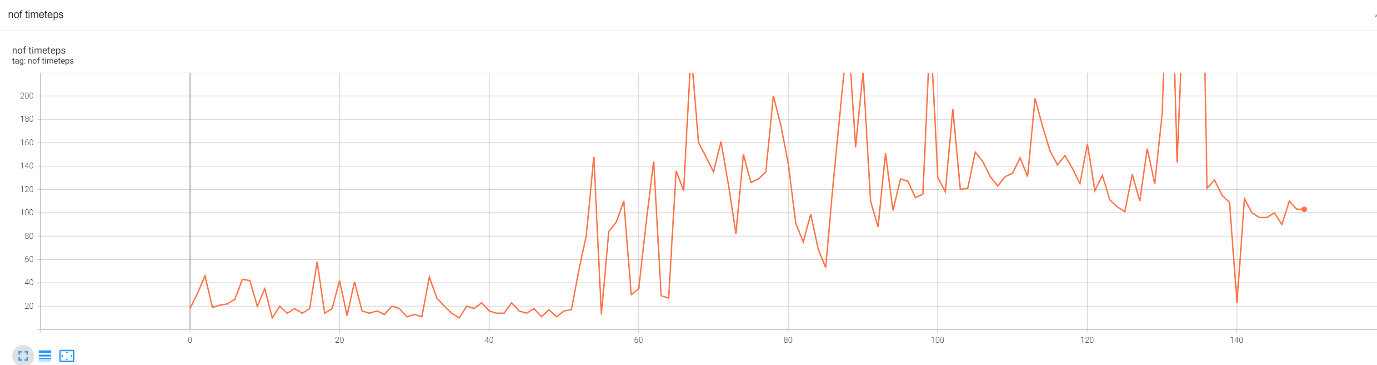
picked episode saved on episode 134 after 700 timesteps (episode reached 714 steps)

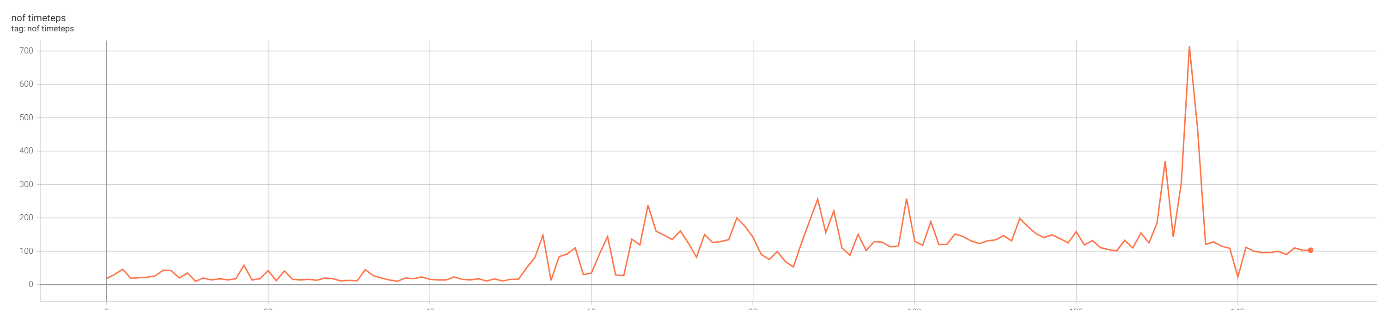
Loss:

Chart, bar chart, line chart, histogram

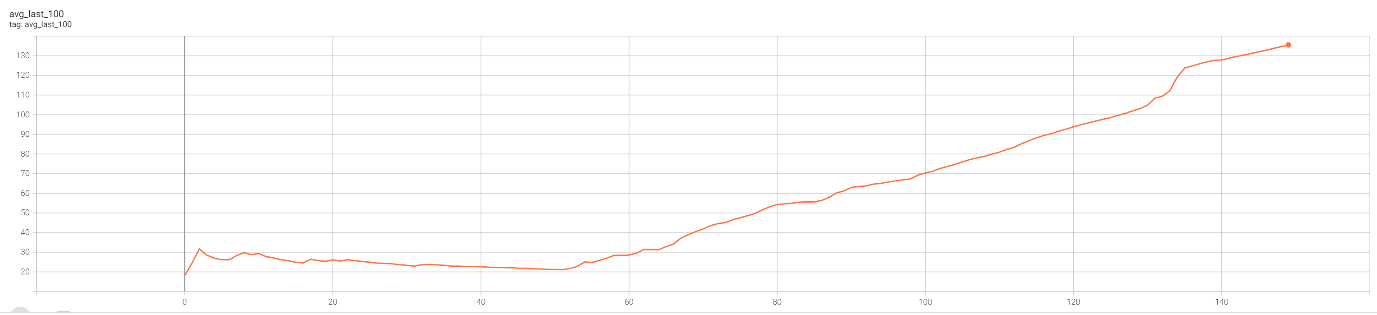
Description automatically generated

last step index (reward):

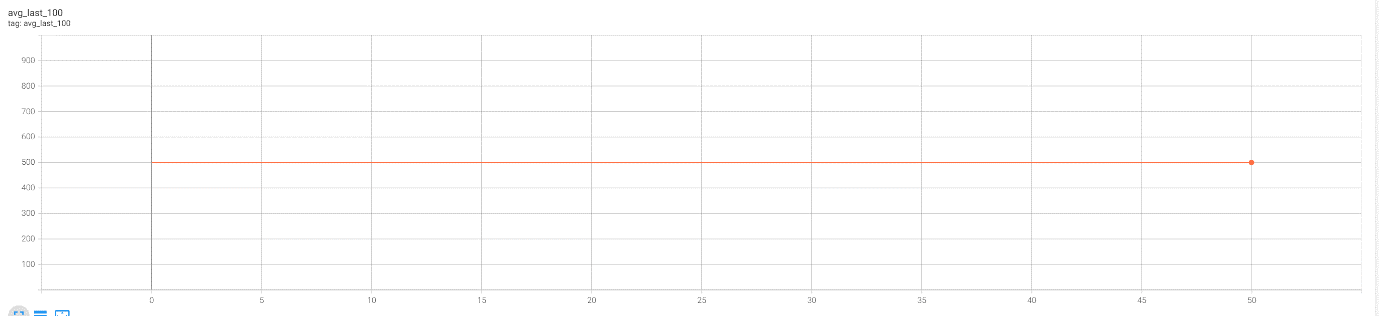


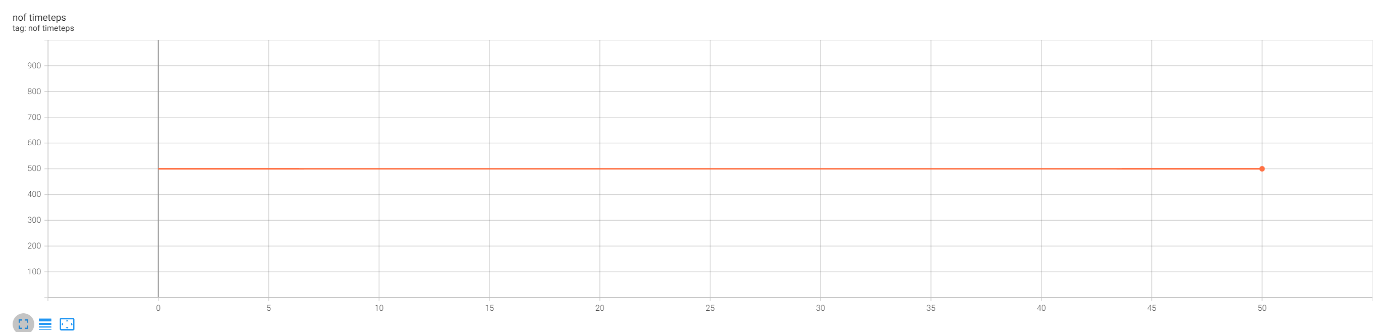
Zoomed out:

Average last 100:



Testing:

average last 100 (all 500):

last step index (reward) (all 500):

3 layers network

Layer (type) Output Shape Param #

=================================================================

input\_4 (InputLayer) [(None, 4)] 0

dense\_12 (Dense) (None, 512) 2560

dense\_13 (Dense) (None, 256) 131328

dense\_14 (Dense) (None, 64) 16448

dense\_15 (Dense) (None, 2) 130

=================================================================

Total params: 150,466

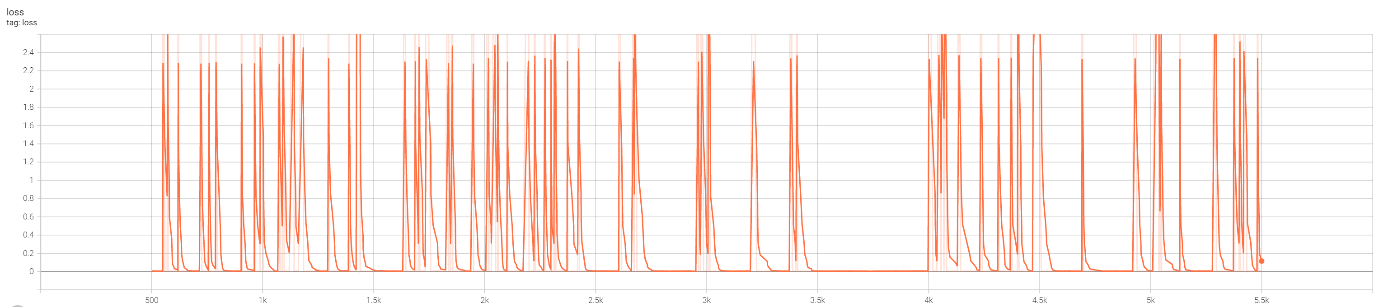
Trainable params: 150,466

Non-trainable params: 0

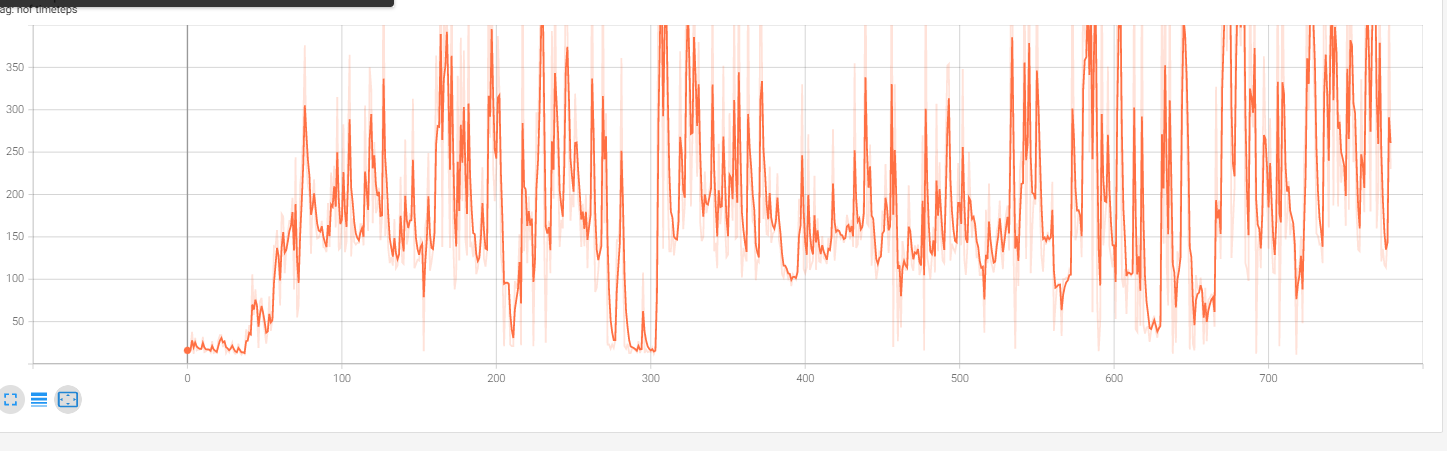
Training:

Saved weights of episode 768 of 500 steps (limited to 500 on training)

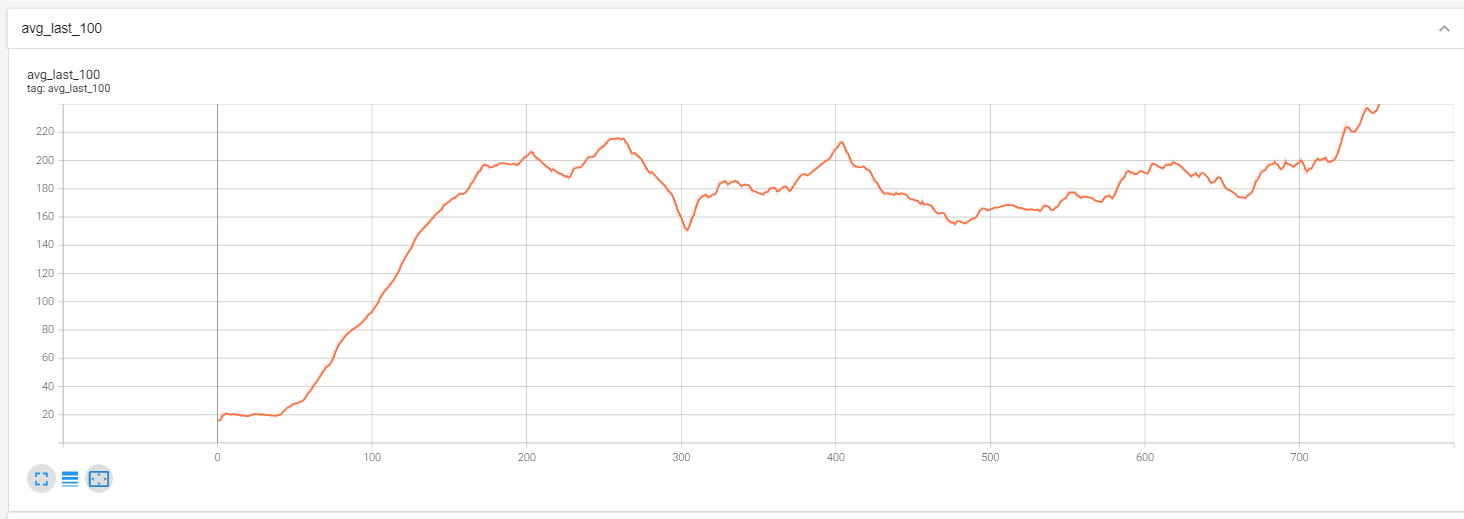
Loss:



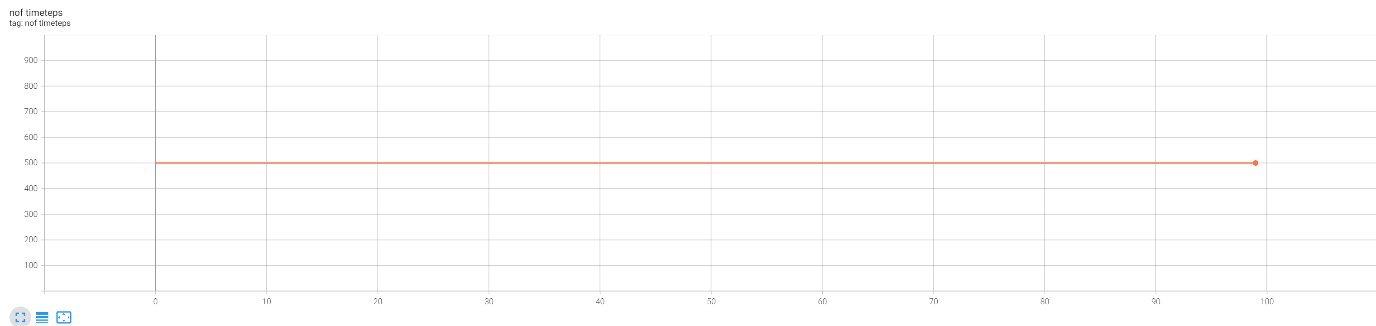
V



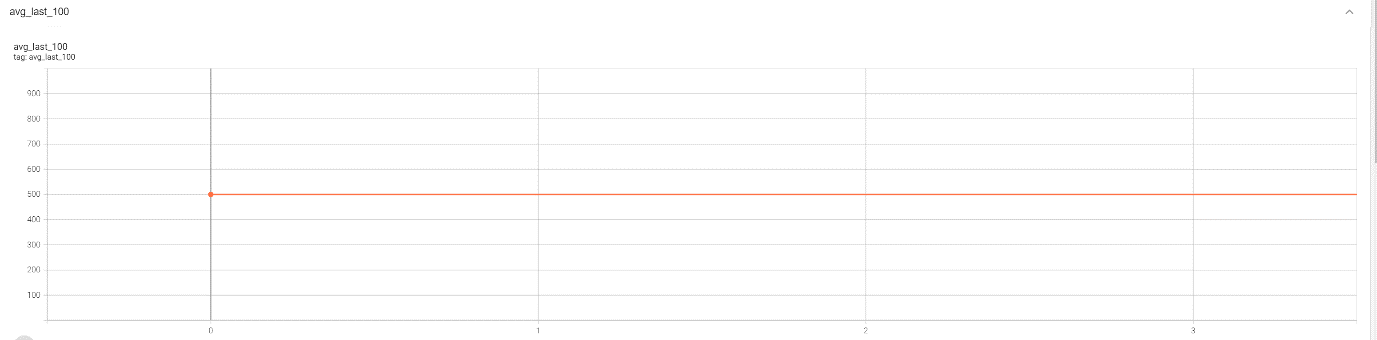
Average last 100:



Testing average last (all 500):



Testing last step index (all 500):



on both setups we used the same parameters (described above) we tried many different layers widths and ended up with the ones above. The shallower network has wider layers and more parameters. On both architectures we received perfect results on the testing.

The parameter C effected the stability of the results C=100 caused the number of timesteps to be unstable. After raising it we got fast results.

The learning rate effected the most, to high - and the results goes up and down, to slow - and it got stuck on low number of steps.

At first, we couldn’t get above a small episode length, therefore we introduced a small change to the experience\_replay\_list appending mechanism, we waited until the episode was over, if it was higher than the last average 100 episodes, we added all the episode, if it wasn’t we added it with probability 0.3.

We also changed the way we save the weights on training; we assumed that the weights of a long run will be better, however when we saved the weights of the longest run, we got poor results. We assumed that the reason is that we saved the weights when the pole dropped - so the last training iterations on the last few steps “ruined” the weights, therefore we saved the weights before the done signal (or done at maximum steps). we saved if there was improvement by 20 steps from the last save in order to reduce the chance that we save the weights when the weights are “ruined” – few steps before the pole actually falls (and in order to save time).

We ran the testing for 100 consecutive episodes above 475 after we got weights that we believe were good, weights of runs well above 500 or when limited to 500, equals 500, and with many “good” runs on close episodes.

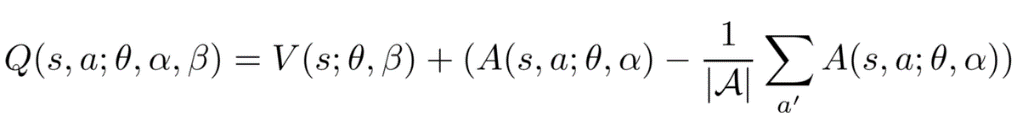
We took the weights and loaded them to a model and ran without training with epsilon=0 until average of at least 475 of at least 100 consecutive episodes. We got perfect score on both architectures after 100 episodes.

**Section 3 – Improved DQN (25%)**

Now we used in Dueling DQN with Prioritized Experience Replay,

Dueling DQN:

The key concept is to decouple the action from the state, so now we have two networks to train one is for the states (which is given by V(s)) and the second is for the action (which is given by A(s,a)). the combination of V(s) and A(s,a) gives the Q function by the following equation:

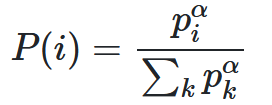


The purpose behind this technique is to consider the importance of choosing action across all states.

Prioritized Experience Replay:

The idea is to “replay” the samples that teach our model the most.

The probability of sampling transition is given by the following equation



where is the priority of transition i, and α is the prioritization factor.

there is two ways to compute :

1. = + ε, is the TD error and ε ensures that every sample has a chance of being sampled.
2. = , is the rank of transition when the replay memory is sorted according to .

1 ≥ α ≥ 0, when α = 0 we have uniform sampling,

when α = 1 we have greedy sampling.

**Hyper-parameters values:**

lr=0.00025

memory\_size = 10000

gamma = 0.95

batch\_size = 32

PER\_a = 0.6 # Hyperparameter that we use to make a tradeoff between taking only exp with high priority and sampling randomly

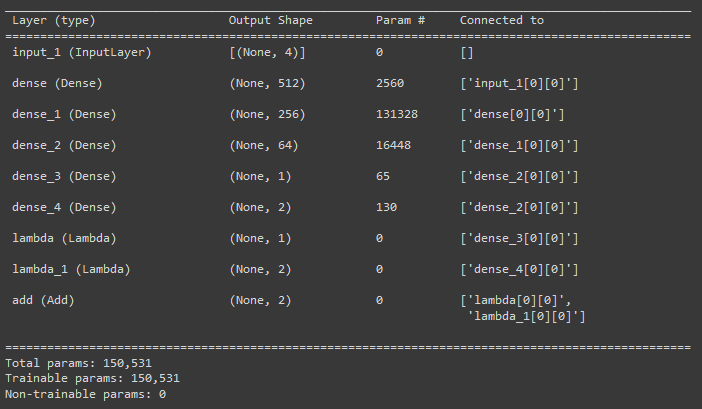
self.env.\_max\_episode\_steps = 4000

epsilon = 1.0 # exploration probability at start

epsilon\_min = 0.01 # minimum exploration probability

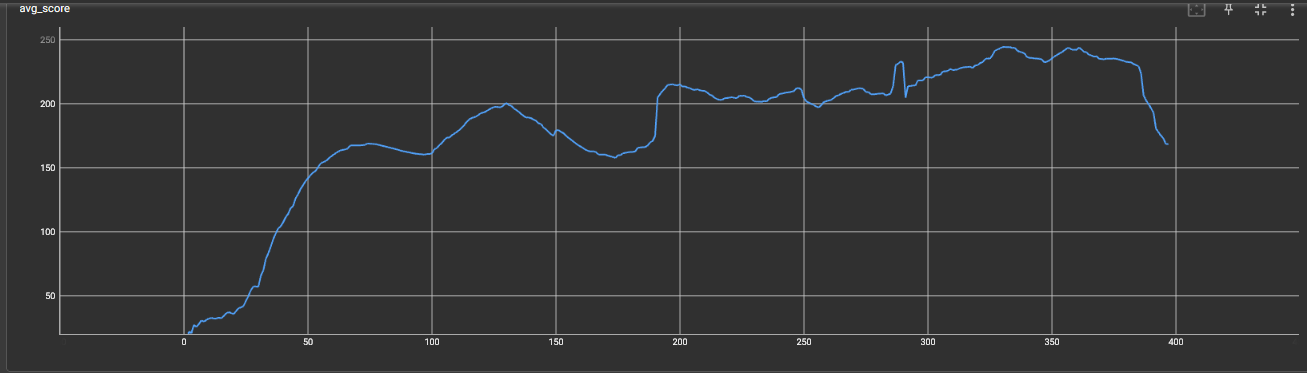
epsilon\_decay = 0.0005 # exponential decay rate for exploration prob

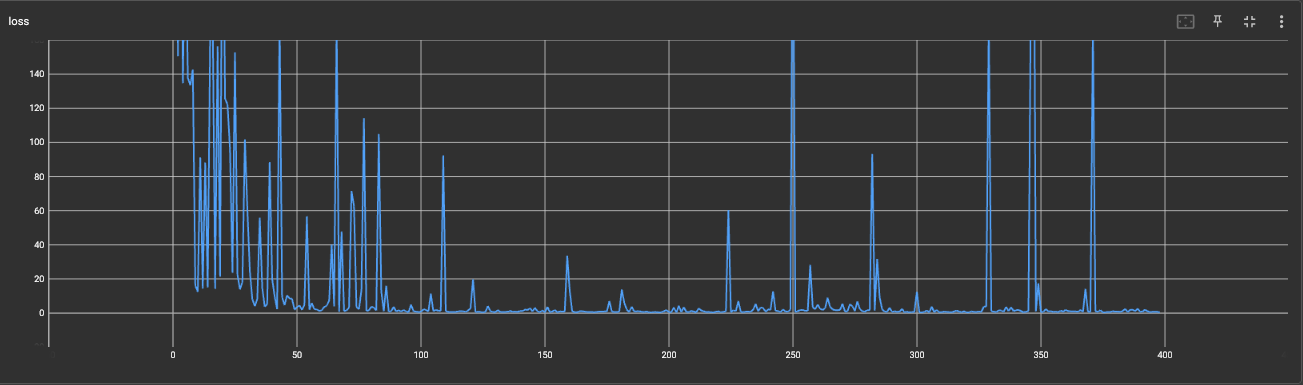
**Neural networks structure:**



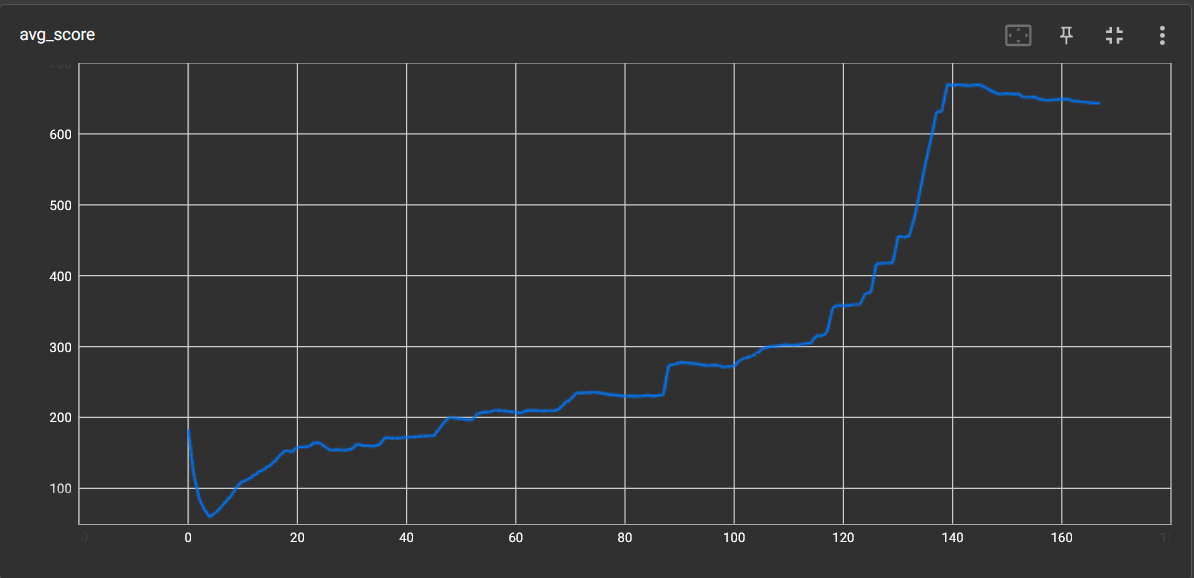
**results:**

We first trained our model in 400 episodes and after that in 140 episodes, In the end we got an average score of 650 over 100 episodes.

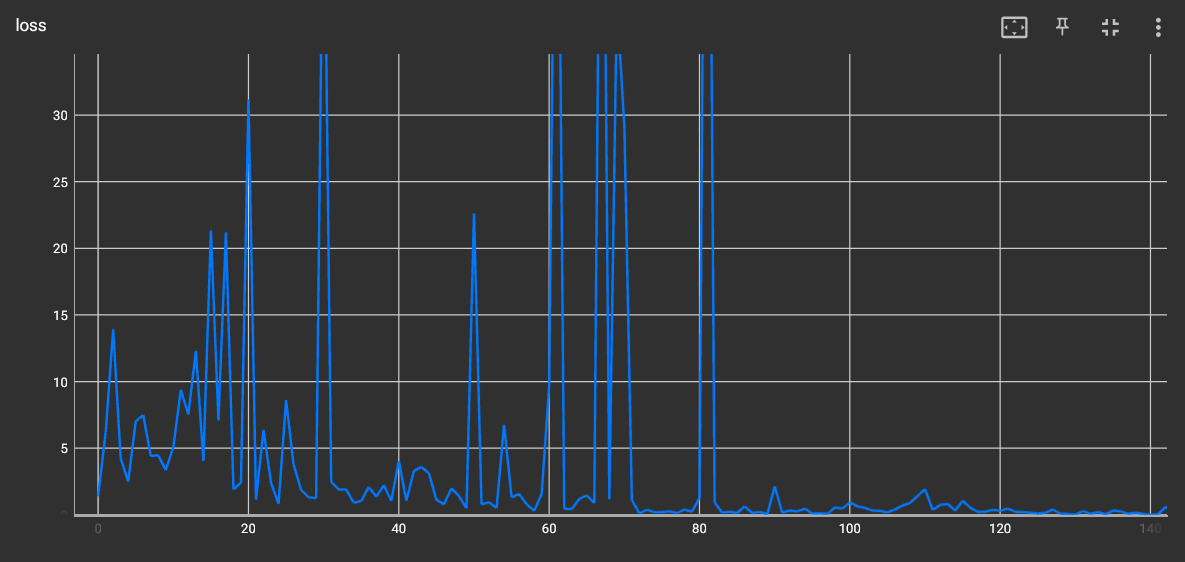




Training:



Loss:



As we can see, we got a score of 475 after 630 episodes.

**Q3 code instruction:**

To start the code you have to take PER.py (which is an external library) and Q3.py codes to the same dir and start Q3.py.

**Code parameters:**

Q3.py include Class DQNAgent that include the operation parameters:

self.env.\_max\_episode\_steps - determines the max number of steps of the episodes by default CartPole-v1 has max episode steps of 500.

self.env.\_max\_score - determines on which score we want to save the NN model.

self.EPISODES - determines how many episodes we want to run on our code.

memory\_size - the number of trajectories that saved in the reply memory

self.gamma - determines our gamma discount.

self.epsilon - exploration probability at start

self.epsilon\_min - minimum exploration probability

self.epsilon\_decay - exponential decay rate for exploration prob

self.batch\_size - number of batch size

self.dueling - use dueling network

self.USE\_PER - use Prioritized Experience Replay

On the main function:

We have a logdir variable which determines the saving path dir, we also have agent.load(‘ ’) when we want to use the initial weights model.

When we want to train the model we type agent.run() and when we want to test the model we type agent.test().