Deep Reinforcement Learning

ASSIGNMENT 1

Dor Shmuel - 312179971 | Yuval Cohen - 314978198 | 23/11/2022

Section 1 – Tabular Q Learning

1.1

While value and policy iteration are efficient approaches in dynamic programming, their processes require the evaluation of all available actions and states. Therefore, those methods cannot be implemented under complex environments with many states, even when the true dynamics are known. Moreover, as model-based methods, they require the knowledge of the transition probability distribution $p(s'|s,a)$ (based on the Bellman equation) to update the value function, which is lacking or too complicated to calculate in an unknowable dynamics environment.

1.2

For scenarios where there is no prior knowledge about environment dynamics i.e. the transition probability matrix is missing, model-free algorithms are aimed to directly estimate the expected reward function $q(s,a)$ by observing state-action pairs, instead of modeling the true "physics "of the environment. Since those methods are based on sampling strategies, they can infer from a sufficient number of observations, without sampling every available state and action. A model-free method has advantages over a model-based method that tries to construct an accurate model of an environment.

1.3

The difference between Q-learning and SARSA algorithms lies in the distinction between on-policy and off-policy. Using on-policy methods, we attempt to evaluate and improve the policy that is used to make decisions and interact with the environment. Off-policy methods are attempt to improve a policy other than the one used to generate the data it infers on. Sarsa algorithm is an on-policy method that updates the q-function based on

the current policy, whereas Q-learning is an off-policy method that updates q based on greedy actions. Therefore, Q-learning will find the optimal solution, even at high risk.

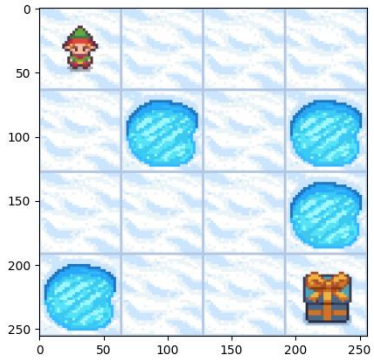
1.4

The ϵ-greedy algorithm aims to balance exploration and exploitation by randomly acting under each approach. Exploitation chooses the action that maximizes reward, which may lead to sub-optimal actions. Exploration allows an agent to improve its current knowledge about each action, hopefully leading to long-term benefits. By maintaining the ability to explore forever, under the law of large numbers, ϵ-greedy (epsilon refers to the probability of choosing to explore) ensures that we will find the optimal policy (by exploring all available actions and exploiting them).

1.5 Q-LEARNING ALGORITHM

Frozen lake Environment

- "**FrozenLake_v1**" is a virtual environment that confers the ability to play and simulate different episodes of the popular game "Frozen Lake"



- As it described in the official documentation of the gym library, Frozen lake game involves crossing a frozen lake from Start(S) to Goal(G) without falling into any Holes(H) by walking over the Frozen(F) lake.
- Since the frozen lake has a slippery nature, the agent may not always move in the intended direction, i.e. the environment will apply the chosen action with a probability to operate, while there is also a probability to commit different action.
- Both scenarios of slippery nature will be examined.
- **Actions**: {0: move left, 1: move down, 2: move right, 3: move up}
- **States**: for a given NxN table game, there are N^2 available states, where s(t), the current state at given time stamp t, can be numerated from 0 to $N^2 - 1$.
- **Rewards:** equal to 1 when reaching the goal, otherwise 0.

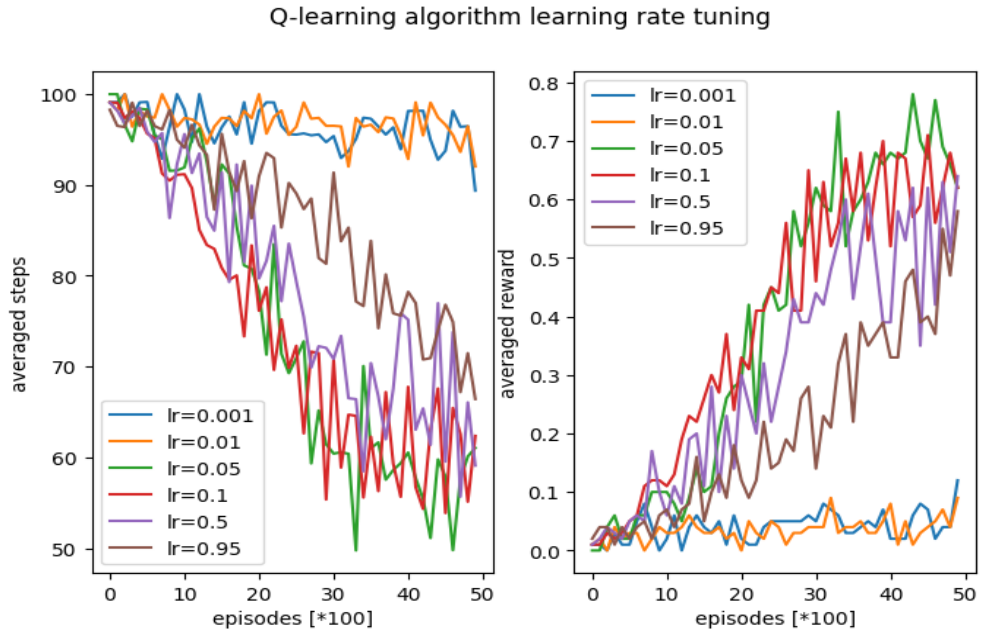
Q-Learning algorithm – probabilistic settings

"frozen_lake = FrozenLakeGame("Stochastic")"

In the following, we implemented the Q-learning algorithm over the discussed environment, allowing our machine to optimally act over the game, reaching an overall amount of 100 steps over 5000 episodes. Actions have been sampled using a ϵ-greedy algorithm with a decaying factor.

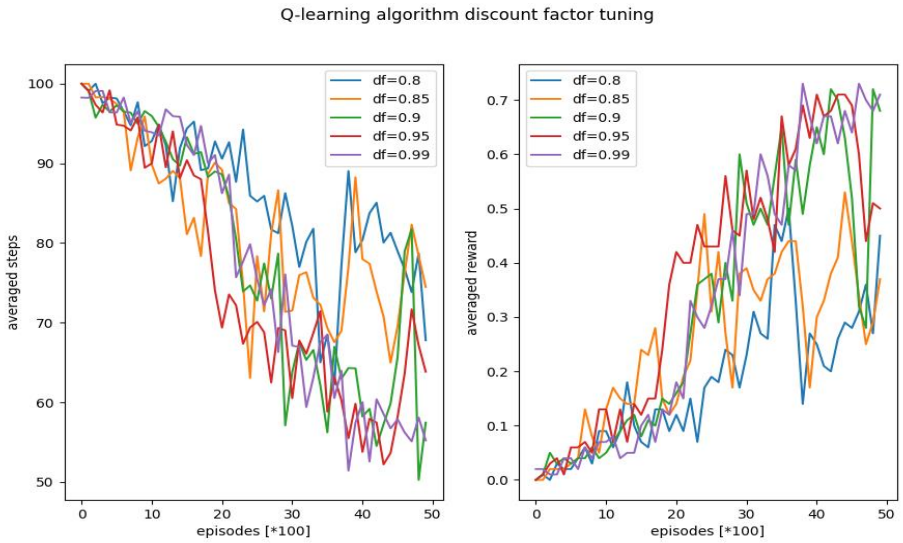
For that purpose we have committed tuning for the following hyper-parameters:

1. **Learning rate tuning**:
   we can notice that by setting the learning rate to **lr=0.05**, after 5k episodes, the algorithm is able to achieve an average reward (over 100 consecutive episodes) of ~**0.8**, reaching that goal by performing **50-60** steps.



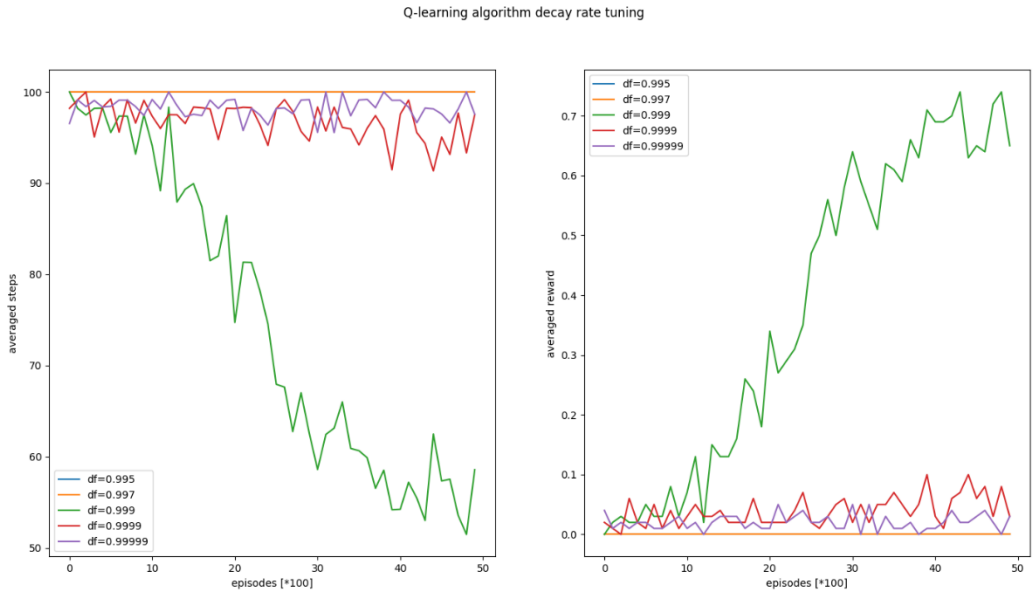
2. **Discount factor tuning:**

   The optimal discount factor which achieves the performances described above is set to be **df = 0.99**. we can figure out that since "Frozen Lake" with a 4x4 table is a relatively simple and short game, better results are achieved as the discount factor is closer to 1 (as it is in finite setting).



3. Epsilon decay factor tuning:

By setting the initial ϵ value to 0.99, we can notice the optimal results (and the only one to converge to solution) are given by taking decay rate = 0.999. this could happen since the chosen factor value is the only one that combined auth the other tuned hyper-parameters, or that this rate is the one that perfectly synchronizes the trade-off between exploration and exploitation, yielding good performances.

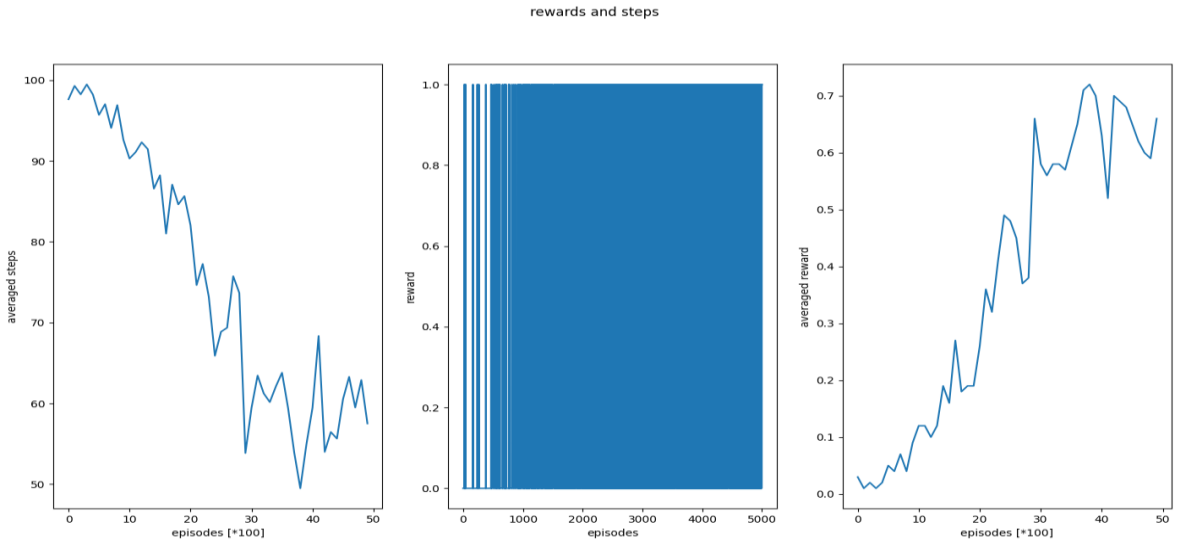


Optimal hyper-parameters:

Learning rate	Discount factor	Decaying rate	Epsilon
0.05	0.99	0.999	0.9

Probabilistic settings performances:

Under those chosen parameters, we simulate the frozen lake environment for 5k episodes, resulting the following **plots**:



We can notice that the curve of rewards for a given episode is fluctuate (even though, that the reward in the initial episodes is more tends to 0), due to the probabilistic nature of the environment (the desired action will be applied in the probability of $\frac{1}{3}$). The maximal averaged award is about ~**0.75**, whereas the minimal averaged step for reaching the goal is about ~**50-60** steps.

We can also notice that using decaying $\epsilon - greedy$, yields that the agent starts with high exploration and continue into progressed exploitation.

Q-tables

Q-value lookup table after 500 episodes

	LEFT	DOWN	RIGHT	UP
0	0.0004	0.0007	0.0005	0.0004
1	0.0003	0.0004	0.0005	0.001
2	0.002	0.0009	0.0009	0.0003
3	0.0	0.0001	0.0	0.0001
4	0.0009	0.0004	0.0003	0.0002
5	0.0	0.0	0.0	0.0
6	0.0047	0.0019	0.0004	0.0001
7	0.0	0.0	0.0	0.0
8	0.0002	0.002	0.0005	0.0012
9	0.001	0.0024	0.0076	0.0027
10	0.0351	0.0088	0.0145	0.0007
11	0.0	0.0	0.0	0.0
12	0.0	0.0	0.0	0.0
13	0.0024	0.0089	0.0	0.0196
14	0.0107	0.1693	0.0017	0.0401
15	0.0	0.0	0.0	0.0

Q-value lookup table after 2000 episodes

	LEFT	DOWN	RIGHT	UP
0	0.2487	0.27	0.2295	0.2212
1	0.1335	0.144	0.1307	0.2251
2	0.1998	0.1371	0.1401	0.1028
3	0.0912	0.0314	0.0065	0.0137
4	0.3151	0.2425	0.1967	0.133
5	0.0	0.0	0.0	0.0
6	0.1343	0.1176	0.1959	0.0359
7	0.0	0.0	0.0	0.0
8	0.1556	0.195	0.1509	0.393
9	0.179	0.4684	0.2205	0.1531
10	0.4825	0.2609	0.1997	0.1048
11	0.0	0.0	0.0	0.0
12	0.0	0.0	0.0	0.0
13	0.1076	0.1481	0.5829	0.2134
14	0.2713	0.7646	0.5128	0.3222
15	0.0	0.0	0.0	0.0

Q-value lookup table after 5000 episodes

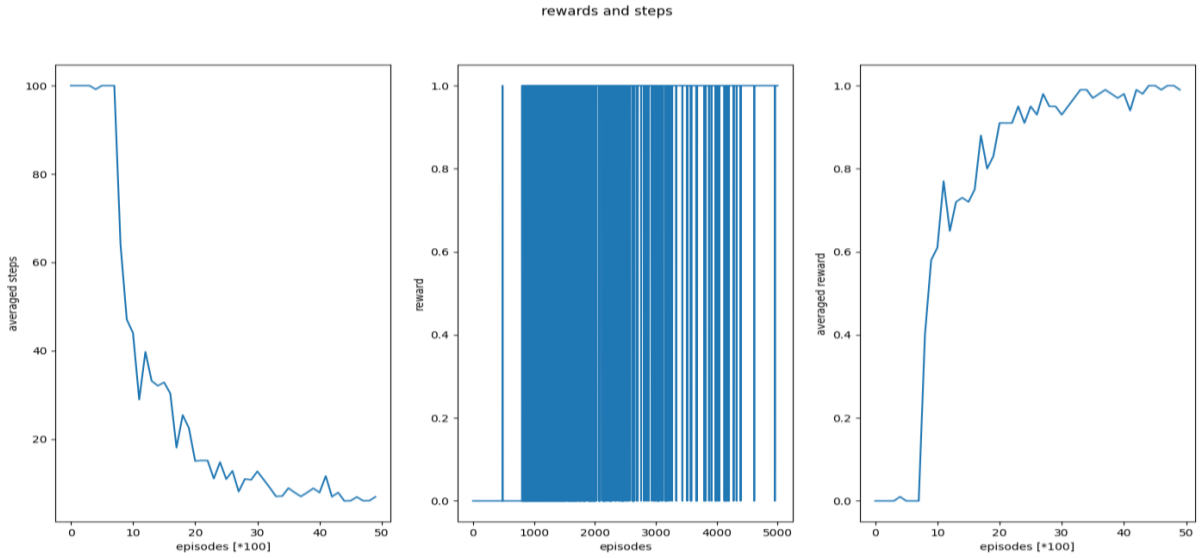
	LEFT	DOWN	RIGHT	UP
0	0.4976	0.4742	0.4576	0.466
1	0.2394	0.207	0.1826	0.4288
2	0.3703	0.2064	0.2013	0.1765
3	0.1393	0.0314	0.0109	0.0169
4	0.5208	0.4288	0.3187	0.2875
5	0.0	0.0	0.0	0.0
6	0.2091	0.1664	0.3101	0.056
7	0.0	0.0	0.0	0.0
8	0.3746	0.3314	0.412	0.5617
9	0.309	0.6054	0.4575	0.3939
10	0.5881	0.4624	0.4037	0.2885
11	0.0	0.0	0.0	0.0
12	0.0	0.0	0.0	0.0
13	0.3781	0.5346	0.652	0.3984
14	0.6537	0.8379	0.7074	0.6642
15	0.0	0.0	0.0	0.0

1. First, we can observe that since states {5, 7, 11, 12} are defined as traps in the initial configuration of the environment and state 15 is the destination, the discrete condition of termination aborts the game, resulting in zero Q-Value, i.e. the value to commit some action where some of those states are the current one, is equal to 0. This is holds since the 500 initial episodes.
2. Since states {11,12} are traps, reaching the goal state {15} is available only when action is taken from state {14}. Hence, and as we can notice from the above Q-tables, maximal Q values are present in state 14(after 500 episodes, not all the values in the 14 state line are maximal, but after 2000 episodes it does happen).
3. Moreover, even though we expect that the maximal Q-value would be given for taking action {2:right} from state 14, we can observe that higher Q-value is presented for taking the {1:down} action. This is a direct result of the stochastic nature of the environment (the agent will move in the intended direction with probability of $\frac{1}{3}$ else will move in either vertical direction with equal probability of $\frac{1}{3}$ in both directions). In that case, tasking the down action from state 14 will just keep us afar from the destination:
    a. Taking the **right** action will lead us to:
    state 15 w.p of $\frac{1}{3}$, stay in 14 w.p of w.p of $\frac{1}{3}$ and move to state **10** w.p of $\frac{1}{3}$.
    b. Taking the down action will lead us to:
    state 15 w.p of $\frac{1}{3}$, stay in 14 w.p of w.p of $\frac{1}{3}$ and move to state **13** w.p of $\frac{1}{3}$.

    The only difference between the actions is the option to be on state 10 or state 13. Since state 13 has a **higher** Q-value for reaching state 14, and is safer than state 10 when moving toward the goal (taking the right from state 10 has more probability to fall into trap in comparison to taking right action from state 13, which has zero chance to fall), the down action has higher value than the right one, when current state is 14.

Deterministic settings performances:

Under the same hyper-parameters chosen for the probabilistic settings, we simulate the frozen lake environment for 5k episodes, resulting in the following **plots**:



We can notice that the curve of rewards for a given episode is way less fluctuating, whereas for the last episodes it is almost always 1.

The maximal averaged award and steps are close to ~0 over the first 1k episodes, due to the high exploration rate induced by the ϵ-greedy algorithm. In comparison to the probabilistic case, the average reward increased sharply, reaching almost ~1, and the average number of steps is way lower, attending to the optimal number of steps as the slope over the 4-5k episodes became flat.

Q-tables

Q-value lookup table after 500 episodes

	LEFT	DOWN	RIGHT	UP
0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0
5	0.0	0.0	0.0	0.0
6	0.0	0.0	0.0	0.0
7	0.0	0.0	0.0	0.0
8	0.0	0.0	0.0	0.0
9	0.0	0.0	0.0	0.0
10	0.0	0.0	0.0	0.0
11	0.0	0.0	0.0	0.0
12	0.0	0.0	0.0	0.0
13	0.0	0.0	0.0	0.0
14	0.0	0.0	0.05	0.0
15	0.0	0.0	0.0	0.0

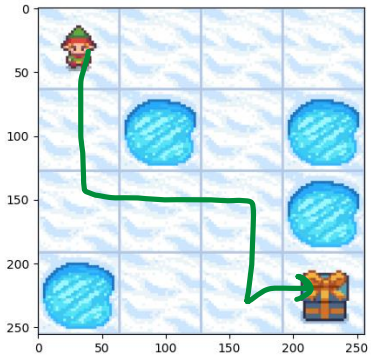
Q-value lookup table after 2000 episodes

	LEFT	DOWN	RIGHT	UP
0	0.9182	0.951	0.7833	0.8562
1	0.9053	0.0	0.0217	0.1939
2	0.187	0.0	0.0	0.0003
3	0.0033	0.0	0.0	0.0
4	0.9158	0.9606	0.0	0.8853
5	0.0	0.0	0.0	0.0
6	0.0	0.8413	0.0	0.0099
7	0.0	0.0	0.0	0.0
8	0.9107	0.0	0.9703	0.884
9	0.8917	0.8836	0.9801	0.0
10	0.8958	0.99	0.0	0.5853
11	0.0	0.0	0.0	0.0
12	0.0	0.0	0.0	0.0
13	0.0	0.3419	0.9818	0.29
14	0.7941	0.9421	1.0	0.886
15	0.0	0.0	0.0	0.0

Q-value lookup table after 5000 episodes

	LEFT	DOWN	RIGHT	UP
0	0.9365	0.951	0.8826	0.9241
1	0.9329	0.0	0.0217	0.2644
2	0.2228	0.0	0.0	0.0003
3	0.0033	0.0	0.0	0.0
4	0.9448	0.9606	0.0	0.93
5	0.0	0.0	0.0	0.0
6	0.0	0.9558	0.0	0.0187
7	0.0	0.0	0.0	0.0
8	0.9445	0.0	0.9703	0.9304
9	0.9458	0.9651	0.9801	0.0
10	0.9559	0.99	0.0	0.8624
11	0.0	0.0	0.0	0.0
12	0.0	0.0	0.0	0.0
13	0.0	0.3419	0.9897	0.324
14	0.9321	0.9792	1.0	0.9566
15	0.0	0.0	0.0	0.0

1. Under deterministic settings, maximal Q-value is given for taking action {2:right} from state 14, as expected $Q(right, s{:}\,14) = 1$.
2. The policy thar described in the last Q-table, produce the optimal path to the target, with the possible minimal length. (there are more similar optimal paths).



Running the script:
In order to run the above simulation, run the script **Q_learning_frozen_lake.py**.

- In order to change the nature of the environment, set " Deterministic" or "Stochastic" inside the FrozenLakeGame class in the code.

```
frozen_lake = FrozenLakeGame("Deterministic")
```

Section 2 – Deep Q-Learning

2.1 – EXPERIENCE REPLAY

Since the experience reply deque comprised from various of consecutive samples (organized as episodes), sampling batches in a deterministic fashion will generate datasets that are suffering from the strong correlation between samples, influenced by the order of which they were committed. Sampling those experiences in a stochastic fashion, significantly mitigates those correlations and the future relations between upcoming samples, increasing the stabilization of the training procedure.

2.2 - USE AN OLDER SET OF WEIGHTS TO COMPUTE THE TARGETS

Conventional Q-Learning algorithm starts with an initial estimate of the Q function agent (Agent knows nothing about the environment in the beginning) and than update it at each iteration, using the same estimation mechanism Q(s,a).

Such update change in the network $Q(s, a)$ will cause an immediate change in the estimation of $Q(s', a')$, because we use the same parameters for the estimation and the Q-targe, allowing to noise from the initial estimation to generate large positive biases in the updating procedure, causing to the "moving target" effect.

To make training more stable and avoid those biases, two separate networks are integrated, when one network is trained and the other, which used to estimate the target ,is frequently updated. The predicted Q values of the target network, are used to backpropagate through and train the main Q-network. The idea is that using the target network's Q values to train the main Q-network will improve the stability of the training.

2.3 – IMPLEMENTATION OF DEEP Q-LEARNING

In the following, we will describe the relevant information for the implementation of DQN algorithm.

Cart Pole Environment

- virtual environment that confers the ability to play and simulate different episodes of the infinite cart-pole problem.As it described in the official documentation a pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum is placed upright on the cart and the goal is to balance the pole by applying forces in the left and right direction on the cart.
- **Actions**: {0: move left, 1: move right}
- **States**: The observation is a ndarray with shape (4,) with the values corresponding to the following positions and velocities.
- **Rewards:** 1 for each step taken.

Architectures

1. Architecture with 3 hidden layers:

```
class ThreeLayersModel(tf.keras.Model):
    def __init__(self, input_dim, output_dim):
        super().__init__()
        self.input_layer    = Dense(16, input_shape=input_dim, activation='relu')
        self.hidden_1       = Dense(64, activation='relu')
        self.hidden_2       = Dense(256, activation='relu')
        self.hidden_3       = Dense(64, activation='relu')
        self.output_layer   = Dense(output_dim, activation='linear')
```

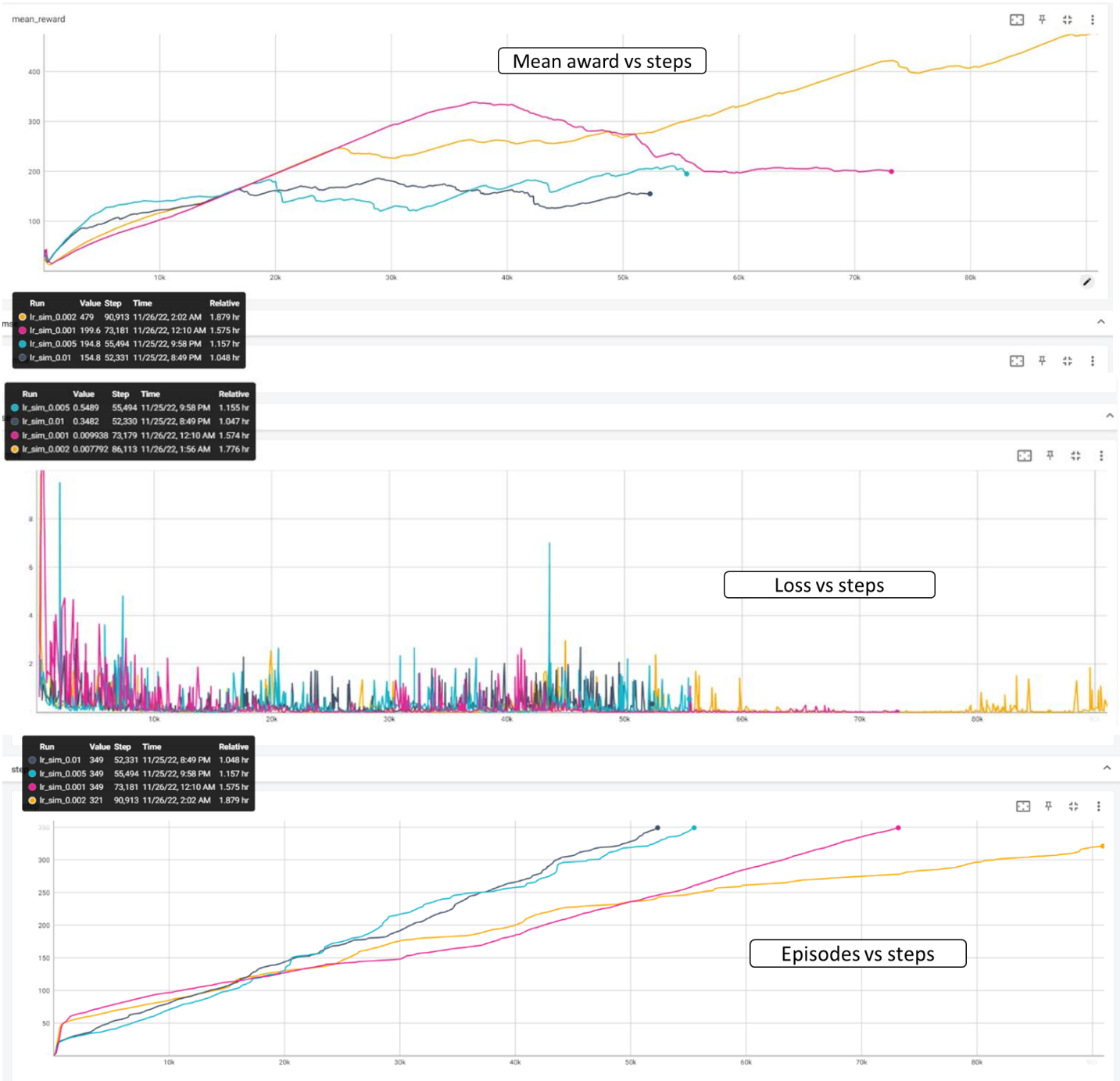
2. Architecture with 5 hidden layers:

```
class FiveLayersModel(tf.keras.Model):
    def __init__(self, input_dim, output_dim):
        super().__init__()
        self.input_layer    = Dense(16, input_shape=input_dim, activation='relu')
        self.hidden_1       = Dense(64, activation='relu')
        self.hidden_2       = Dense(128, activation='relu')
        self.hidden_3       = Dense(256, activation='relu')
        self.hidden_4       = Dense(128, activation='relu')
        self.hidden_5       = Dense(64, activation='relu')
        self.output_layer   = Dense(output_dim, activation='linear')
```

Hyper-parameters Tuning

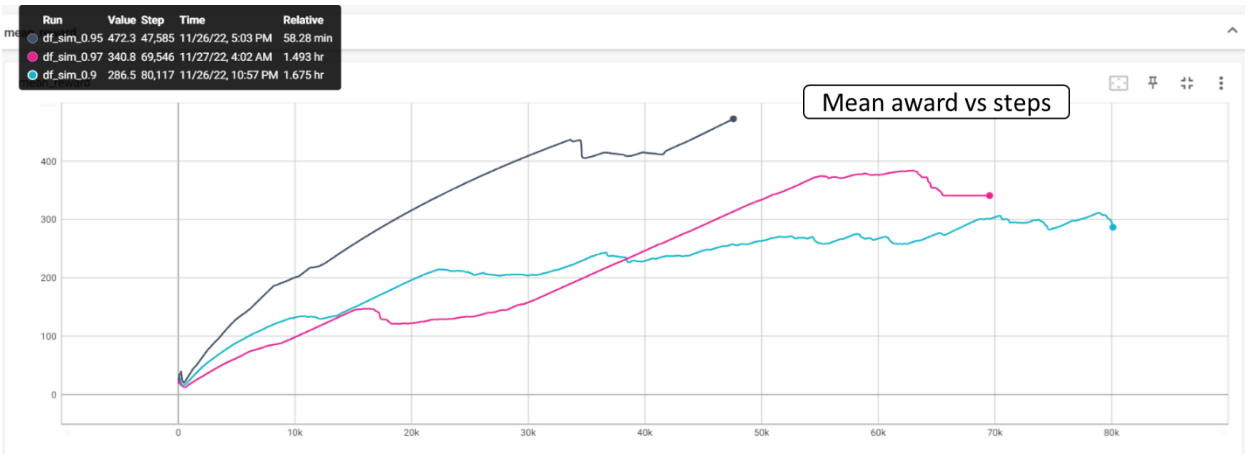
Deep Q-learning algorithm is implemented over the discussed environment, by running 350 episodes in order to reach a mean reward of 475 points for 100 consecutive episodes. As before, actions have been sampled using a ϵ-greedy algorithm with a decaying factor. We managed to apply comprehensive tuning we doubled the maximal number of steps per episodes.
Resulting from the hyper-parameters tuning, the three most affective are the following:

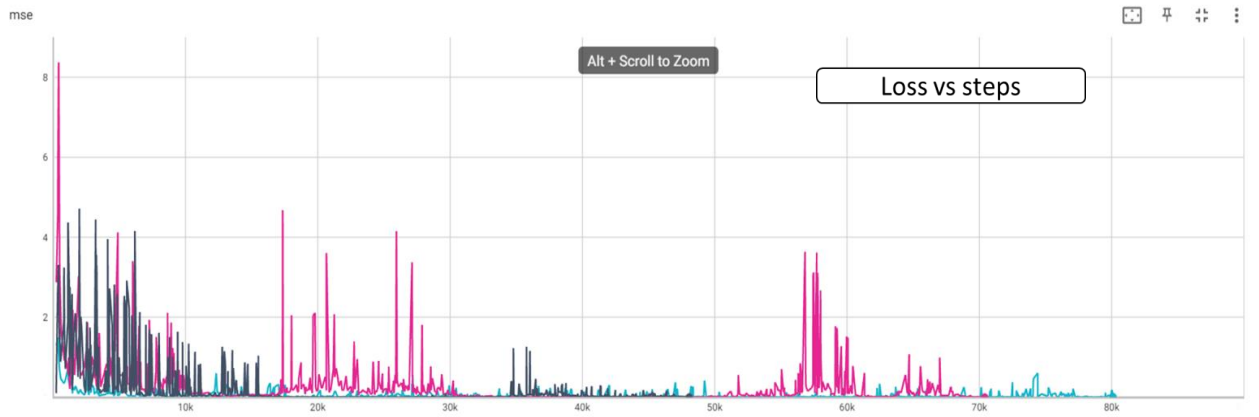
1. **Most effective - learning rate tuning**:
   we can notice that by setting the same seed, only for learning rate **lr=0.002** the agent converged into the desired goal (mean award = 479) after 91K steps which are 321 episodes. The models with the other learning rates are not converged at all. We can observe that the loss curves are fluctuating, where just before convergence the model with lr=0.02 achieves the minimal loss out of all curves.

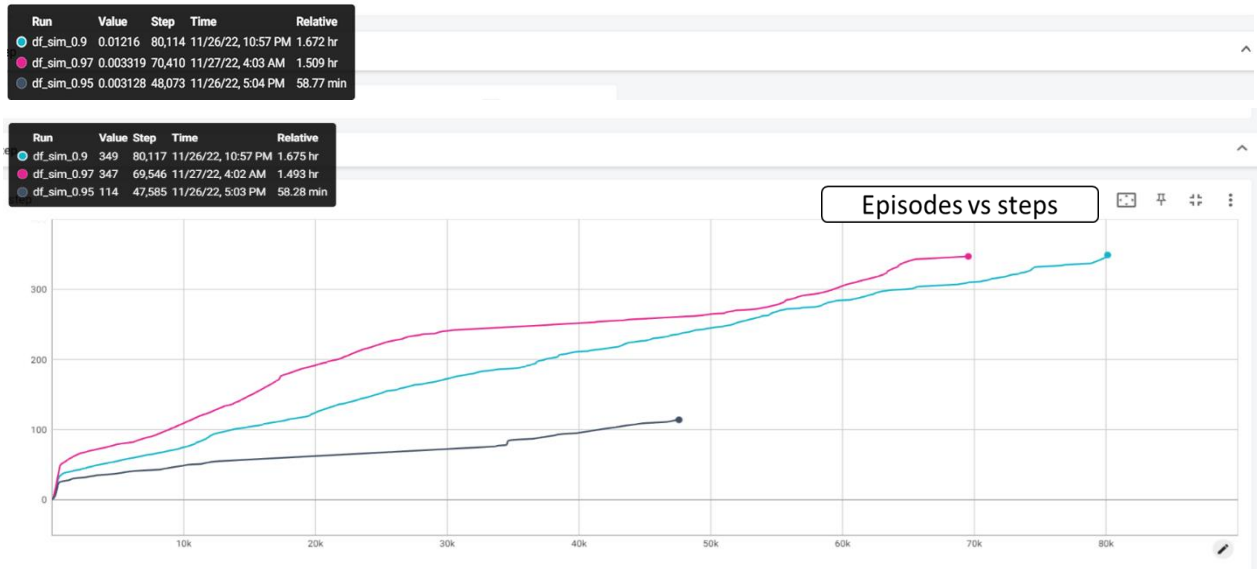


2. **Discount factor tuning:**

   The optimal discount is given by **df = 0.95**, after 114 episodes! again we can notice that for different decaying rates model models not converged to desired mean award. converged.

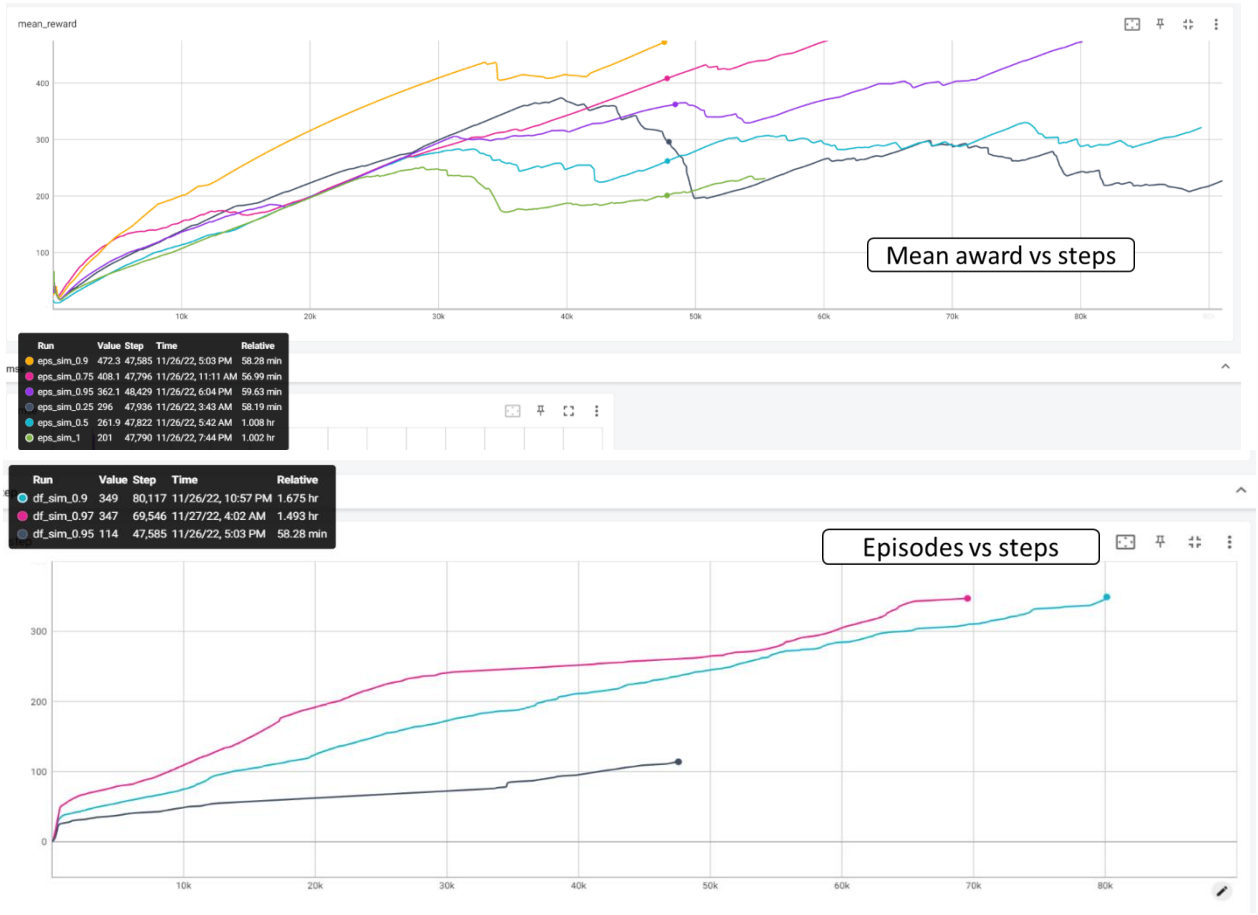






3. Epsilon value tuning:

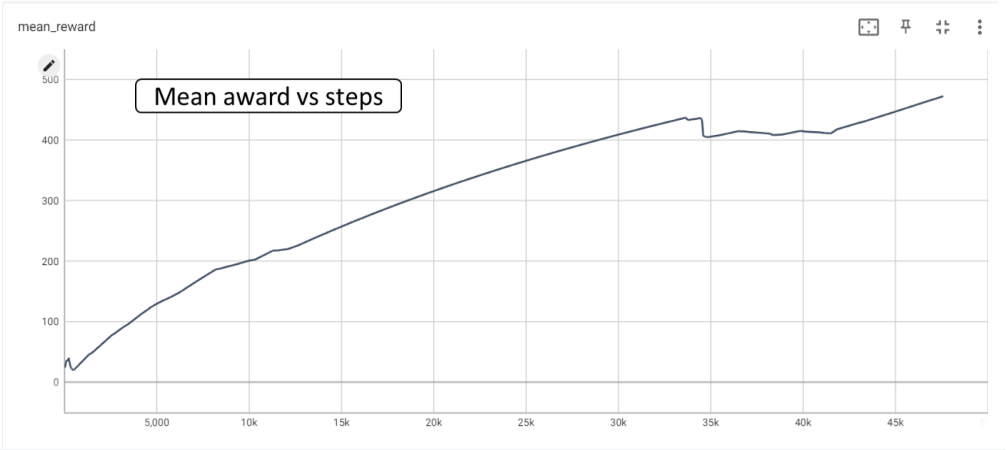
by setting the decaying rate to 0.95, we tested several initial epsilon values. We wish to achieve a high exploration rate at the beginning to increase the reply deque and reduce the correlation of consecutive samples. Even though, reaching fast to the dominant exploitation stage significantly increases the mean award. We can notice that high epsilon values (0.75 and above) are converging into desired goal, while for $\epsilon = 0.25, 0.5$ model didn't converge. Setting the value to 1 produced to much exploration which leads to degrades in performances. Optimal value is given by $\epsilon = 0.95$.

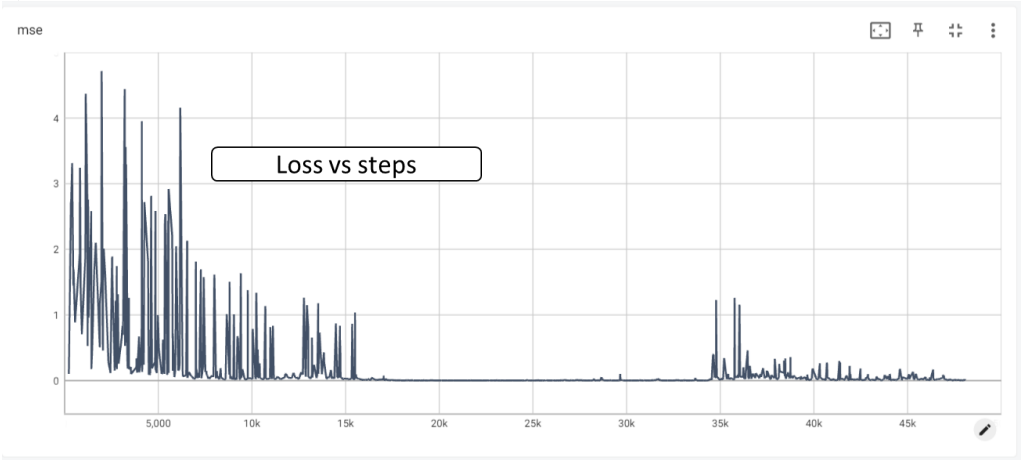


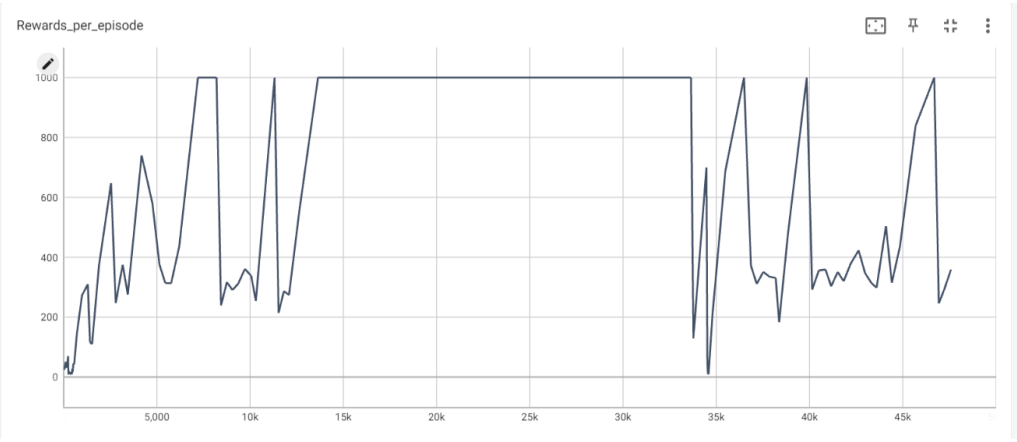
Optimal hyper-parameters:

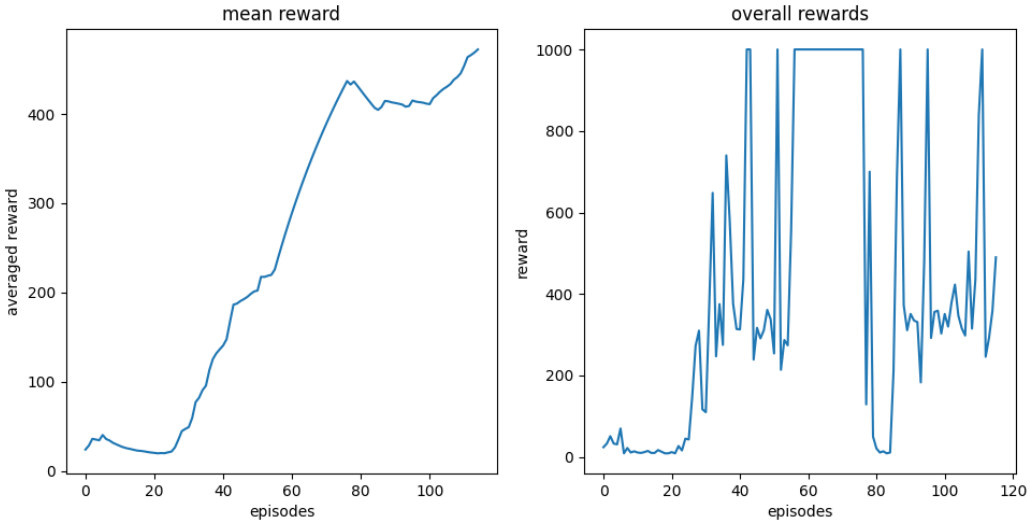
Learning rate	Discount factor	Decaying rate	Epsilon	Update step
0.02	0.95	0.95	0.9	10/50 steps

Under those chosen parameters, we simulate the environment for 350 episodes, resulting the following optimal results:







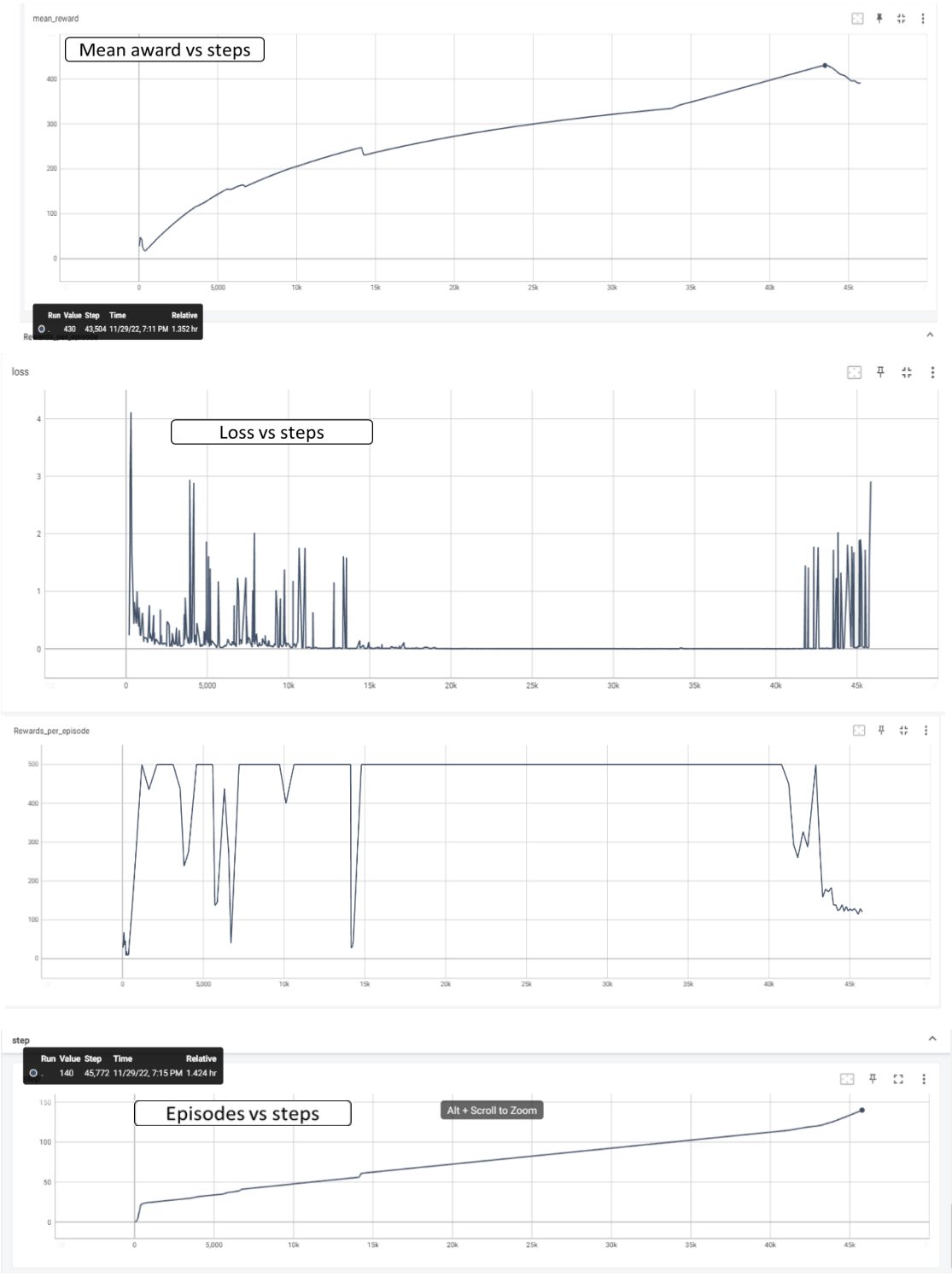


We can observe the following:

1. In the initial episodes, the model achieves low rewards, since it is in the explorations stage. After 20 episodes, we can observe a sharp rise in the received rewards.
2. Even though the limit of the game has been changed to 1000 steps (from time and resource constraints), we can notice that the mean award curve has a very sharp and high slope, which implies on high increasing rate, i.e., the agent seems to apply well-performing policy, and achieve the desired goal.
3. Observing the given loss of the model, we can notice that the loss gradually decreases, while after each update of the weights (every 50 steps), we get a spike in the curve. This can be explained since we are changing the target, which established as the label of the main model training, means we are chasing a moving target.

Simulation with maximal 500 steps in episode:

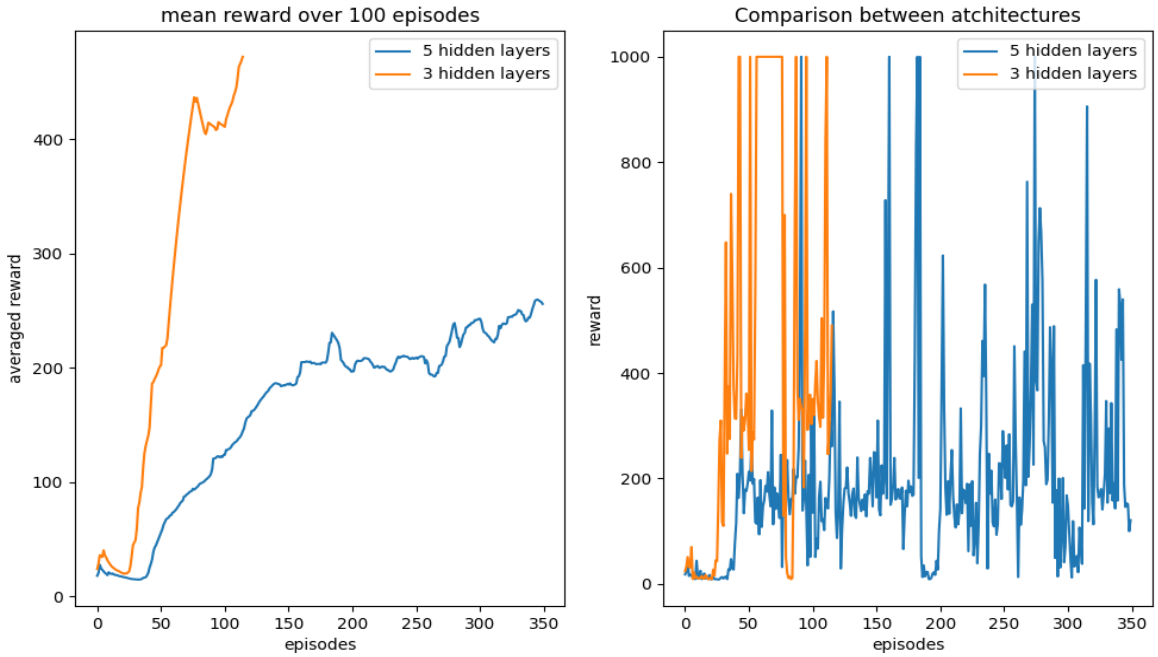
By setting maximal number of 500 steps per episode (as the original setting of the game), we get the following results:



We can notice that the maximal mean award we get is **430** after **114** episodes. while the maximal reward is reached consistently, the instability in the training procedure caused degradation in performances over the few last episodes, as we can also be informed of the loss curve increase.

Comparison between architectures

We can notice that architecture which contains 3 hidden layers performs better than the extended one. It could probably be since the network which comprised from 5 hidden layers is too complex and tends to overfit.



Running the script:
To run the above simulation, run the script **Deep_Q_CartPole.py**.

Section 3 – Double Deep Q-Learning

3.1 – MOTIVATION

In Deep Q-Learning, the training procedure of the main network requires to implicitly take the maximal overestimated value, introducing a maximization bias in learning. Since it involves bootstrapping where learning estimates from estimates, we are experiencing an unstable learning procedure. Double Deep Q-learning is suggested to tackle this problem, by involving two separate Q-value estimators. Using these independent estimators, we can unbiased Q-value estimates of the actions selected using the opposite estimator. we can thus avoid maximization bias by disentangling our updates from biased estimates.

3.2 – DDQN

Instead of taking the maximal value of the target model estimate to determine the given target, we will use the main network (which is currently being trained) to select the optimal action while the target network (which is not being trained) will evaluate its q-value using the chosen action.[1]

$$\max_a Q_{target}(s', a; \theta^-) \rightarrow Q_{main}(s', argmax_a Q_{target}(s', a); \theta)$$

The weights update is done using Polyak averaging:

$$\theta_{target} = \tau * \theta_{target} + (1 - \tau) * \theta_{main}$$

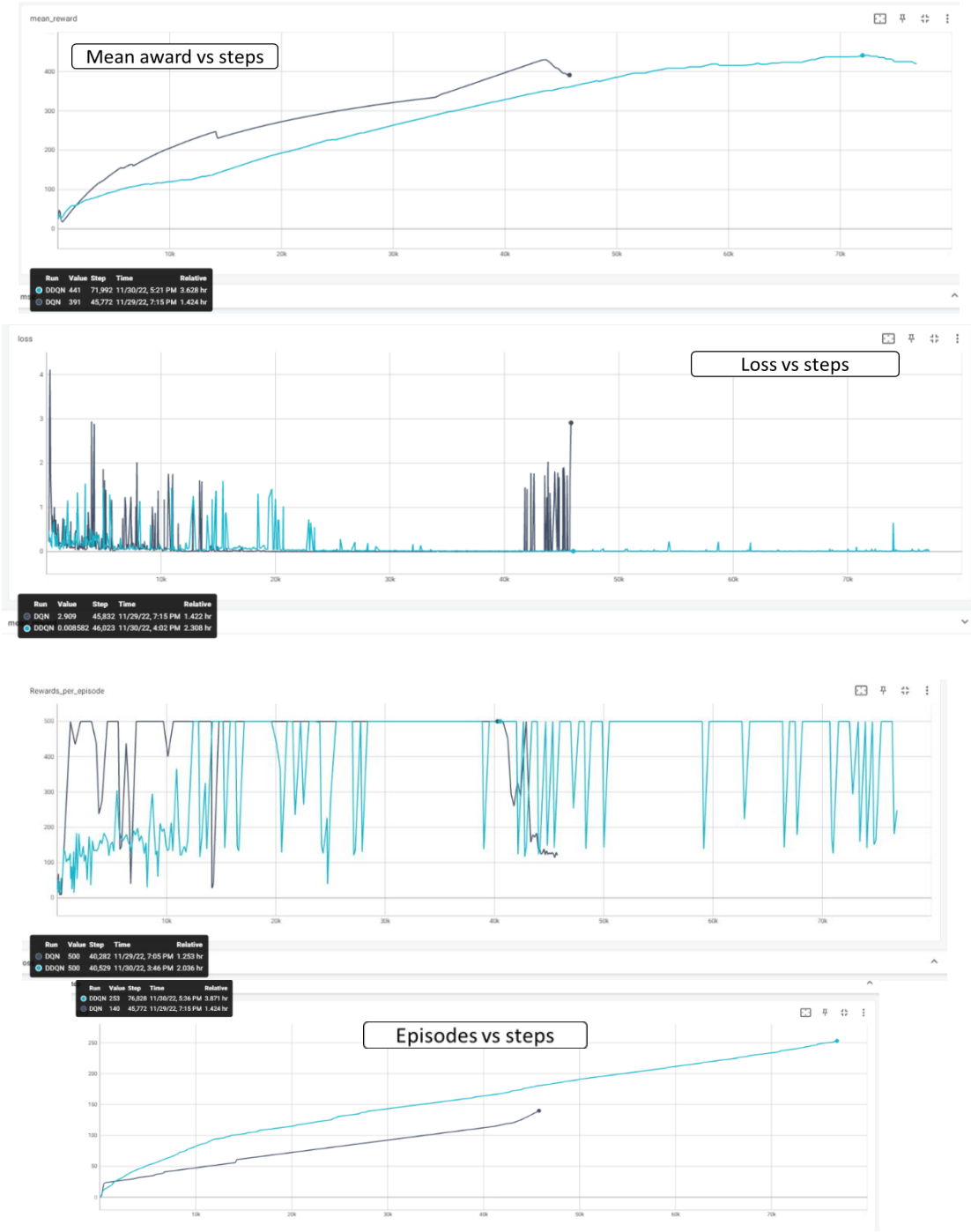
Optimal hyper-parameters:

Learning rate	Discount factor	Decaying rate	Epsilon	Update step	τ
0.002 + lr decay of 0.1 each 50 epochs	0.95	0.95	0.9	5 steps	0.01

We used the same architecture as used in question 2, comprised from 3 hidden layers:

```
class ThreeLayersModel(tf.keras.Model):
    def __init__(self, input_dim, output_dim):
        super().__init__()
        self.input_layer    = Dense(16, input_shape=input_dim, activation='relu')
        self.hidden_1       = Dense(64, activation='relu')
        self.hidden_2       = Dense(256, activation='relu')
        self.hidden_3       = Dense(64, activation='relu')
        self.output_layer   = Dense(output_dim, activation='linear')
```

Under those chosen parameters, we simulate the environment for 350 episodes, and compared it to the results achieved by DQN algorithm:



We can notice the DDQN achieve better performances then DQN (maximal mean award of 441 vs 430), but does it much slower than DQN. DDQN also attains lower loss. and even-though DDQN is noisier than DQN in the initial ~120 episodes, is quite consistent with the noise level added (loss variance is more stable) as more steps are taken, while

DQN perform high and sharp increase in the loss for the last episodes. Both algorithms cant reach the goal of mean award of 475 episodes, but DDQN does performs better.

Running the script:
To run the above simulation, run the script **DoubleDeep_Q_CartPole.py**.

Section 4 – References

[1] Deep Reinforcement Learning with Double Q-learning, 2015, H. van Hasselt, A. Guez, and D. Silver.