Deep Reinforcement Learning

Assignment 2

Dor Shmuel - 312179971 | Yuval Cohen - 314978198 | 14/12/2022

# Section 1 – Monte Carlo Policy Gradient (REINFORCE)

## – Advantage

Since REINFORCE algorithm uses sampling as a derivative of Monte-Carlo algorithm, it might be unstable and fluctuate, yielding slow convergence and high variance (due to its empirical nature). Using the advantage which is the approximation of the expected actual return and not varying with the action taken function, we receive a baseline for reducing the gradient update size that mitigates the high variance. The advantage resembles the award that could be given by taking specific action in each state. Since high variance can be fairly large in different directions, the advantage function is used to normalize the rewards actions that outperform the average for the state.

## – prerequisite condition for baseline reduction

The prerequisite condition for the equation to hold is that the baseline function doesn’t vary with the action i.e. doesn’t depend on the action.

Prove:

1. From the expectation definition:
2. By writing the explicit derivative of , the expectation can be written as:
3. Since and and are continuous in [0, 1], we can interchange integration and differentiation, hence:
4. Since the integral of the p.d.f is given by , we get that:
5. Finally, the expectation is given by:

## 1.3 – Implementation of Reinforce with baseline

In the following, we will describe the relevant information for the implementation of REINFORCE with baseline algorithm in the cart-pole v1 environment (as detailed in assignment 1).

**Changes from policy\_gradients.py**

1. The actor: Policy network architecture – without any changes.
2. The critic: Value network architecture as described below.
3. Hyper-parameters: changed due to tuning.

**Architecture of baseline**

1. Architecture of Value Network:

class ValueNetwork:

    def \_\_init\_\_(self, state\_size, learning\_rate, name='state\_value\_network'):

        self.state\_size = state\_size

        self.learning\_rate = learning\_rate

        with tf.variable\_scope(name):

            self.state = tf.placeholder(tf.float32, [None, self.state\_size], name="state")

            self.A\_t = tf.placeholder(tf.float32, name="discounted\_advantage")

            self.R\_t = tf.placeholder(tf.float32, name="total\_rewards")

            tf2\_initializer = tf.keras.initializers.glorot\_normal(seed=0)

            self.W1 = tf.get\_variable("W1", [self.state\_size, 8], initializer=tf2\_initializer)

            self.b1 = tf.get\_variable("b1", [8], initializer=tf2\_initializer)

            self.W2 = tf.get\_variable("W2", [8, 1], initializer=tf2\_initializer)

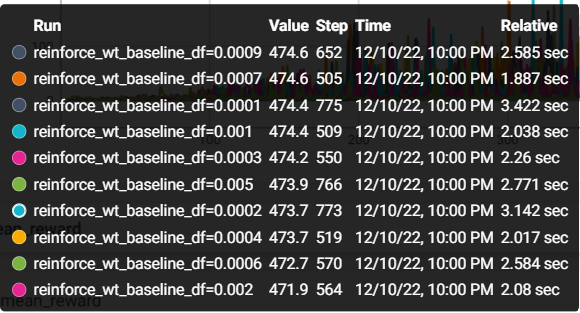
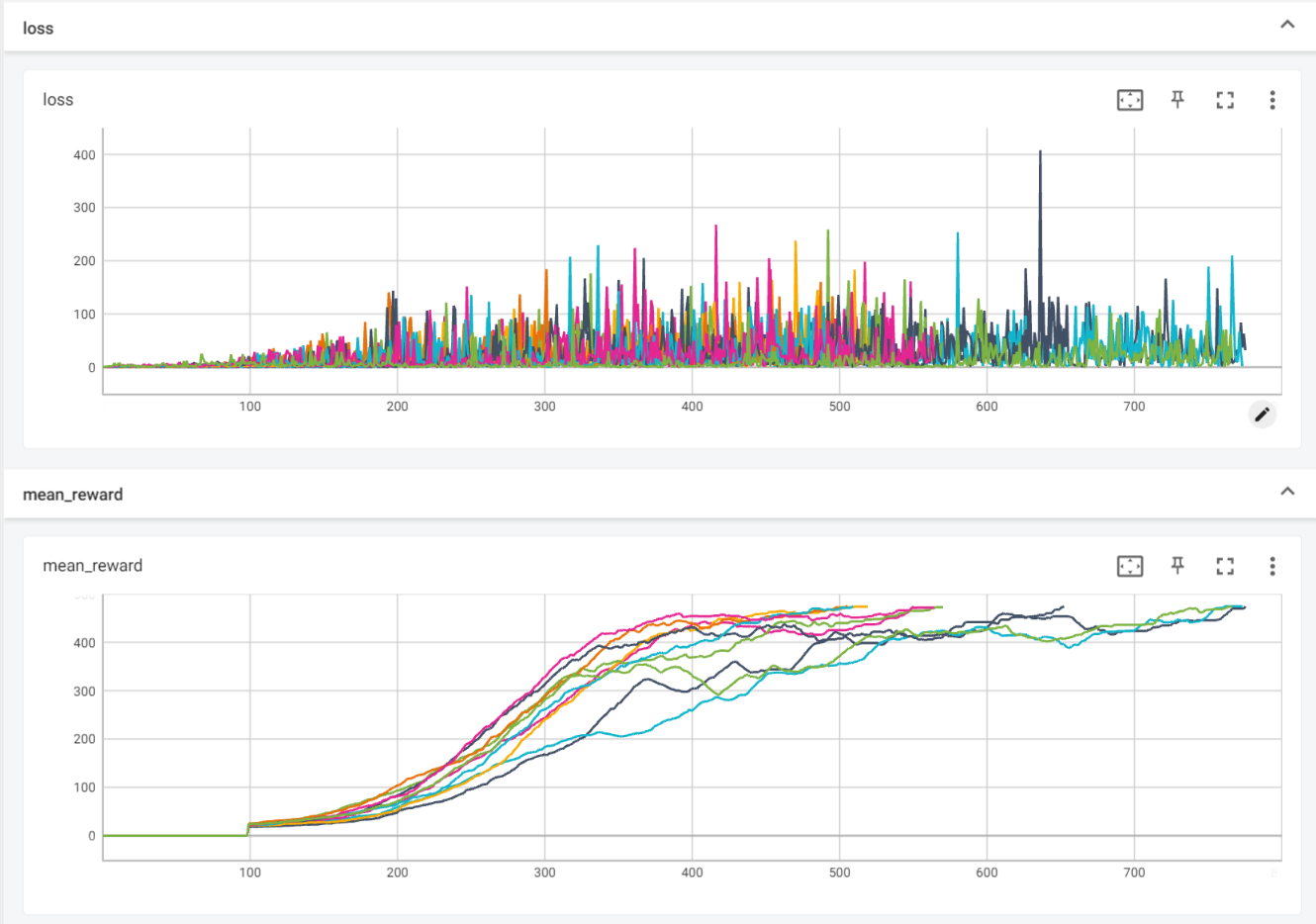
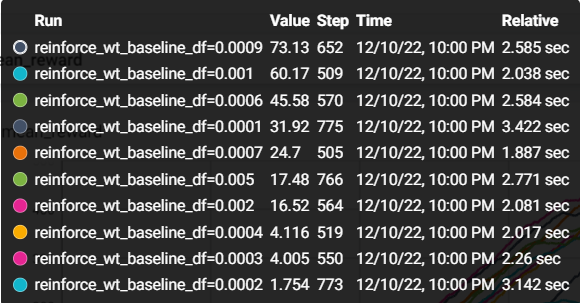
            self.b2 = tf.get\_variable("b2", [1], initializer=tf2\_initializer)

**Hyper-parameters Tuning**

Actor-critic algorithm is implemented over the discussed environment, in order to reach a mean reward of 475 points for 100 consecutive episodes. For optimal tuning we set the same seed through all the tuning procedure.

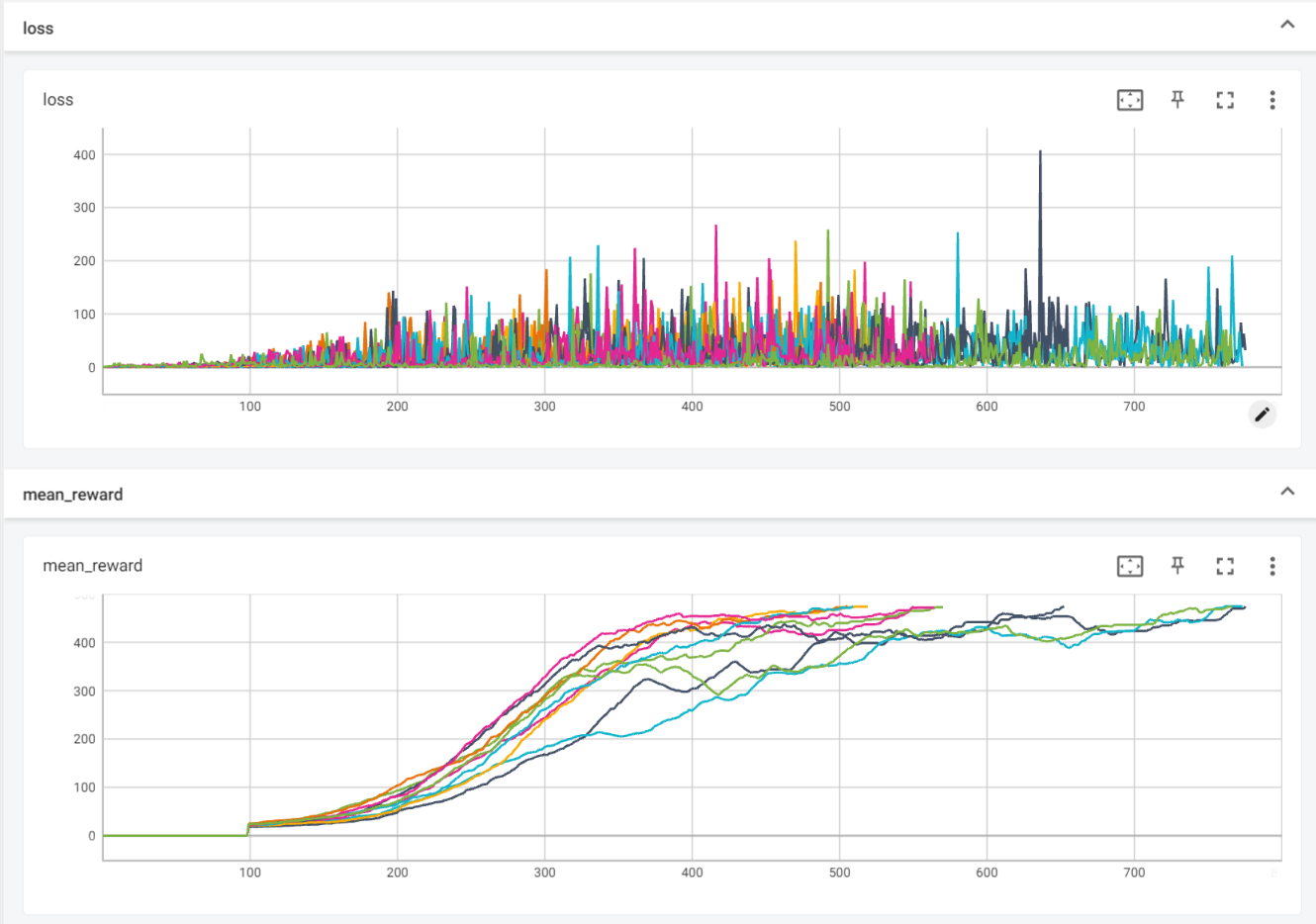
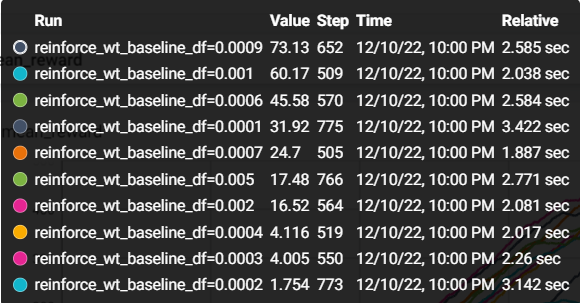
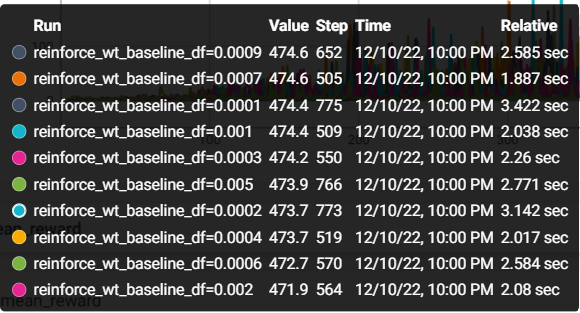
1. **learning rate of baseline - value network tuning**:

reference values for tuning: policy learning rate: 0.0005, discount factor: 0.99.  
for learning rate **lr=0.0007** the agent converged into the desired goal (mean award = 475) after **505** episodes.



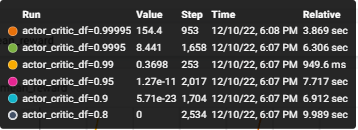
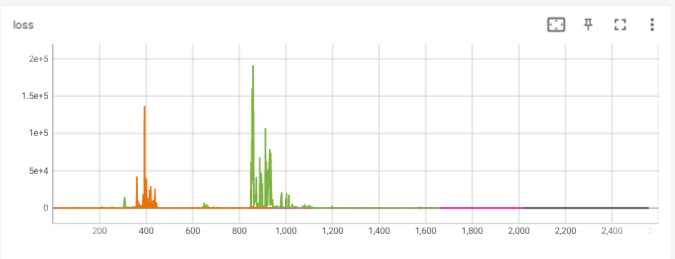
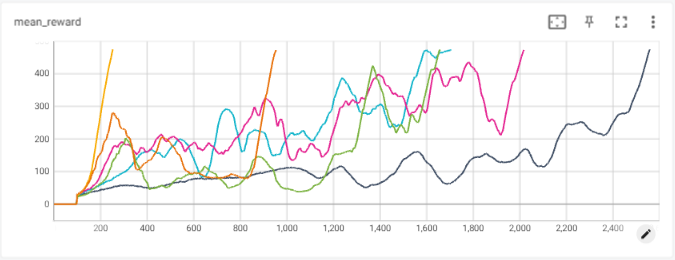
1. **learning rate of policy tuning**:

reference values for tuning: critic learning rate: 0.007, discount factor: 0.99.  
for learning rate **lr=0.0005** the agent converged into the desired goal (mean award = 475) after **505** episodes. (df is lr here)



1. **discount factor**:

reference values for tuning: critic learning rate: 0.007, actor learning rate: 0.0005.  
for discount factor of **df=0.99** the agent converged into the desired goal (mean award = 475) after **253** episodes. We can also notice thar discount factors that greater than 0.99 present wat high loss comparing to the other tested values.



**Optimal hyper-parameters:**

|  |  |  |
| --- | --- | --- |
| Learning rate – policy (actor) | Learning rate – value (critic) | Discount factor |
| 0.0005 | 0.007 | 0.99 |

Under those chosen parameters, we compared the actor-critic, policy gradients with baseline and the original policy gradient algorithms:

**Running the script**:  
In order to run the above simulation, run the script **Q\_learning\_frozen\_lake.py**.

# Section 2 – Advantage Actor-Critic

## 2.1 – Td error Vs Advantage

The TD(0) error is defined as i.e. the difference between current reward plus the estimation of the future rewards since the next state (parametrized value function) and the current estimation of future rewards (resembles the difference between the agent estimation and the reward).

The advantage function estimate is defined as i.e., the difference between the Q-function estimate (the state-action expected rewards approximation) and the value function approximation. Following the bellman equation, the Q-function can be expressed as the expectation of the current reward and the estimation of future rewards . Hence, since both Q-function and the TD(0) error are comprised from the discounted future reward approximation, it can be written as:

As a result, using the TD(0) error for the update process of the actor network is equivalent the using the advantage function approximation.

## 2.2 – actor critic methods

Actor only methods use a parametrized approximator combined with gradient ascent methods in order to learn and constitute an the optimal policy distribution where action will be taken from. Hence, the policy function is playing the rule of the actor. The actor is guided in the action choosing and evaluation by the critic function which rely on the parametrized value function approximation and based on bellman equation. The critic which intended to improve the actor performances and convergence rate is the value function network approximation.

## 2.3 – Implementation of Actor Critic

In the following, we will describe the relevant information for the implementation of actor-critic algorithm in the cart-pole v1 environment (as detailed in assignment 1).

**Changes from policy\_gradients.py**

1. The actor: Policy network architecture – without any changes.
2. The critic: Value network architecture as describes below.
3. Hyper-parameters: changed due tuning.
4. Changes in the form of code – update every step and not every episode.

**Architectures**

1. Architecture of the actor – Policy Network:

class PolicyNetwork:

    def \_\_init\_\_(self, state\_size, action\_size, learning\_rate, name='policy\_network'):

        self.state\_size = state\_size

        self.action\_size = action\_size

        self.learning\_rate = learning\_rate

        with tf.variable\_scope(name):

            self.state = tf.placeholder(tf.float32, [None, self.state\_size], name="state")

            self.action = tf.placeholder(tf.int32, [self.action\_size], name="action")

            self.A\_t = tf.placeholder(tf.float32, name="discounted\_advantage")

            self.I\_factor = tf.placeholder(tf.float32, name="I\_factor")

            tf2\_initializer = tf.keras.initializers.glorot\_normal(seed=0)

            self.W1 = tf.get\_variable("W1", [self.state\_size, 12], initializer=tf2\_initializer)

            self.b1 = tf.get\_variable("b1", [12], initializer=tf2\_initializer)

            self.W2 = tf.get\_variable("W2", [12, self.action\_size], initializer=tf2\_initializer)

            self.b2 = tf.get\_variable("b2", [self.action\_size], initializer=tf2\_initializer)

1. Architecture of the critic – Value Network:

class ValueNetwork:

    def \_\_init\_\_(self, state\_size, learning\_rate, name='state\_value\_network'):

        self.state\_size = state\_size

        self.learning\_rate = learning\_rate

        with tf.variable\_scope(name):

            self.state = tf.placeholder(tf.float32, [None, self.state\_size], name="state")

            self.A\_t = tf.placeholder(tf.float32, name="discounted\_advantage")

            self.R\_t = tf.placeholder(tf.float32, name="total\_rewards")

            self.I\_factor = tf.placeholder(tf.float32, name="I\_factor")

            tf2\_initializer = tf.keras.initializers.glorot\_normal(seed=0)

            self.W1 = tf.get\_variable("W1", [self.state\_size, 64], initializer=tf2\_initializer)

            self.b1 = tf.get\_variable("b1", [64], initializer=tf2\_initializer)

            self.W2 = tf.get\_variable("W2", [64, 16], initializer=tf2\_initializer)

            self.b2 = tf.get\_variable("b2", [16], initializer=tf2\_initializer)

            self.W3 = tf.get\_variable("W3", [16, 1], initializer=tf2\_initializer)

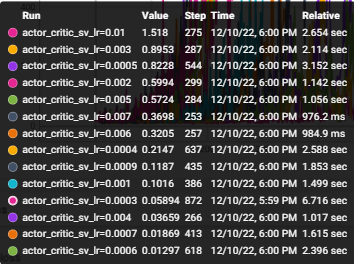
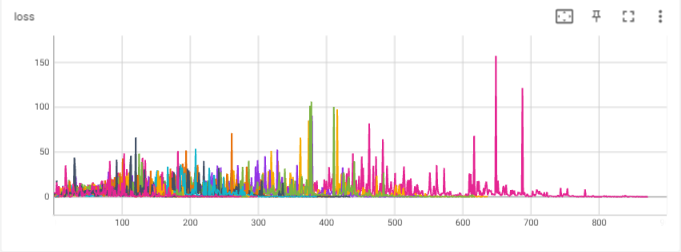
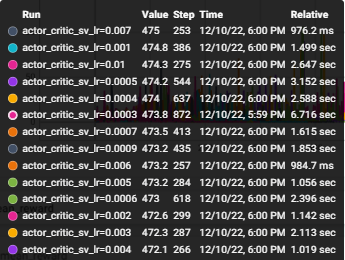
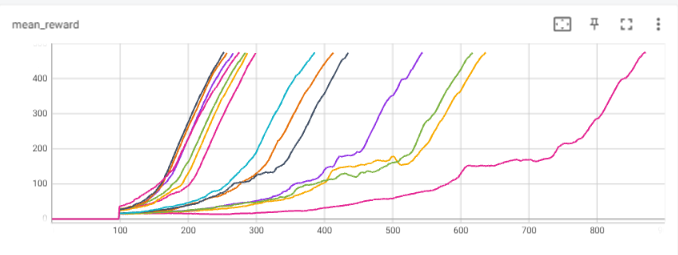
            self.b3 = tf.get\_variable("b3", [1], initializer=tf2\_initializer)

**Hyper-parameters Tuning**

Actor-critic algorithm is implemented over the discussed environment, in order to reach a mean reward of 475 points for 100 consecutive episodes. For optimal tuning we set the same seed through all the tuning procedure.

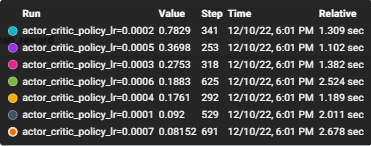
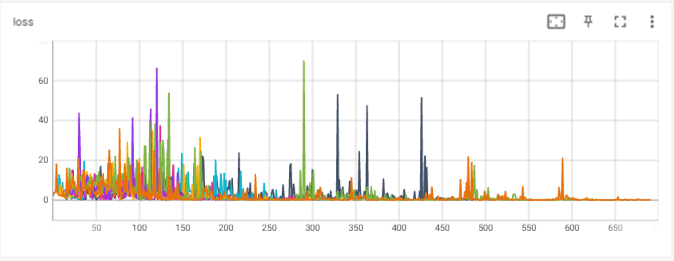
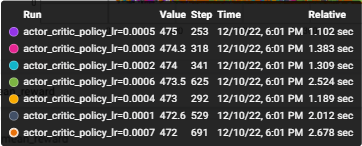
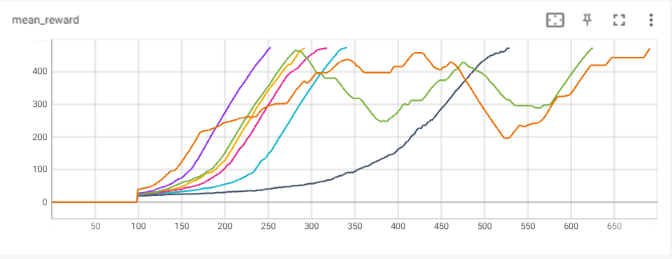
1. **learning rate of critic-value network tuning**:

reference values for tuning: actor learning rate: 0.0005, discount factor: 0.99.  
for learning rate **lr=0.007** the agent converged into the desired goal (mean award = 475) after **253** episodes.



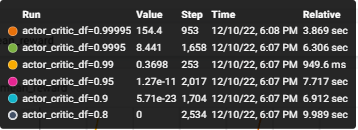
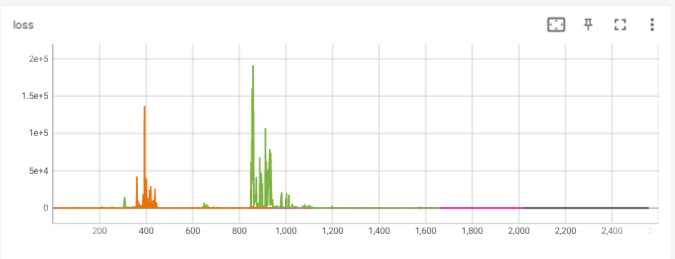
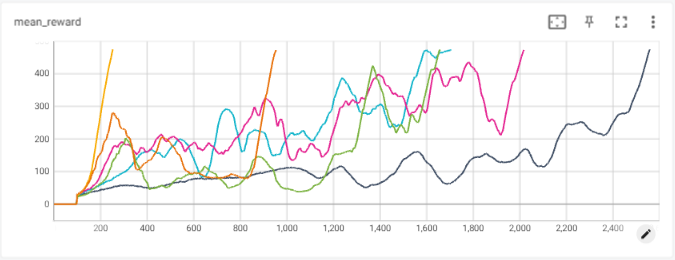
1. **learning rate of actor- policy tuning**:

reference values for tuning: critic learning rate: 0.007, discount factor: 0.99.  
for learning rate **lr=0.0005** the agent converged into the desired goal (mean award = 475) after **253** episodes.



1. **discount factor**:

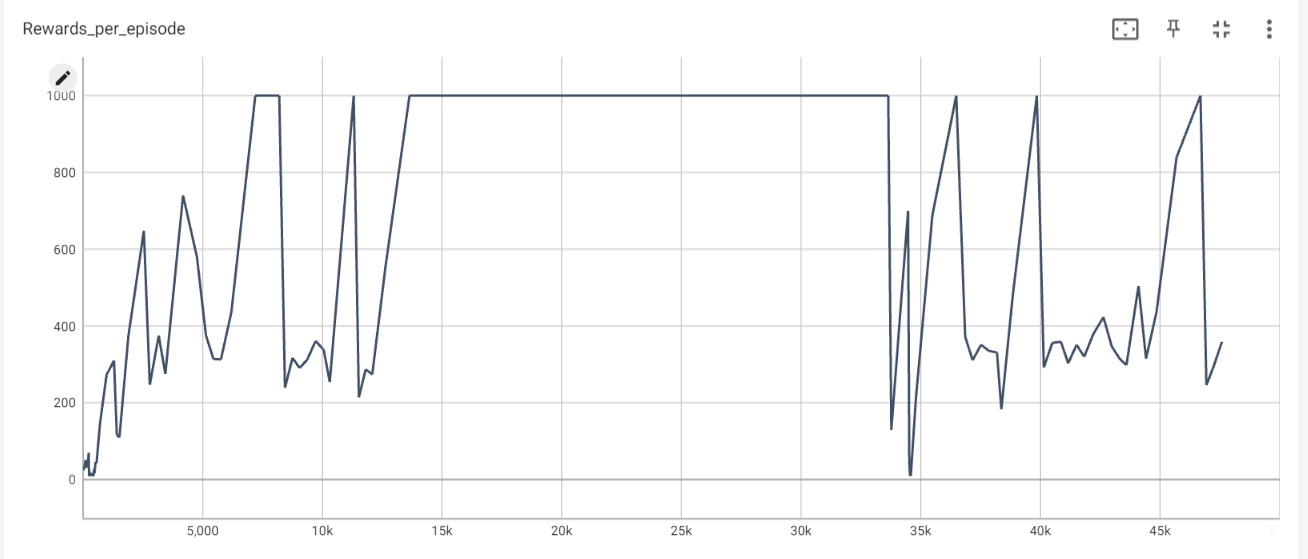
reference values for tuning: critic learning rate: 0.007, actor learning rate: 0.0005.  
for discount factor of **df=0.99** the agent converged into the desired goal (mean award = 475) after **253** episodes. We can also notice thar discount factors that greater than 0.99 present wat high loss comparing to the other tested values.

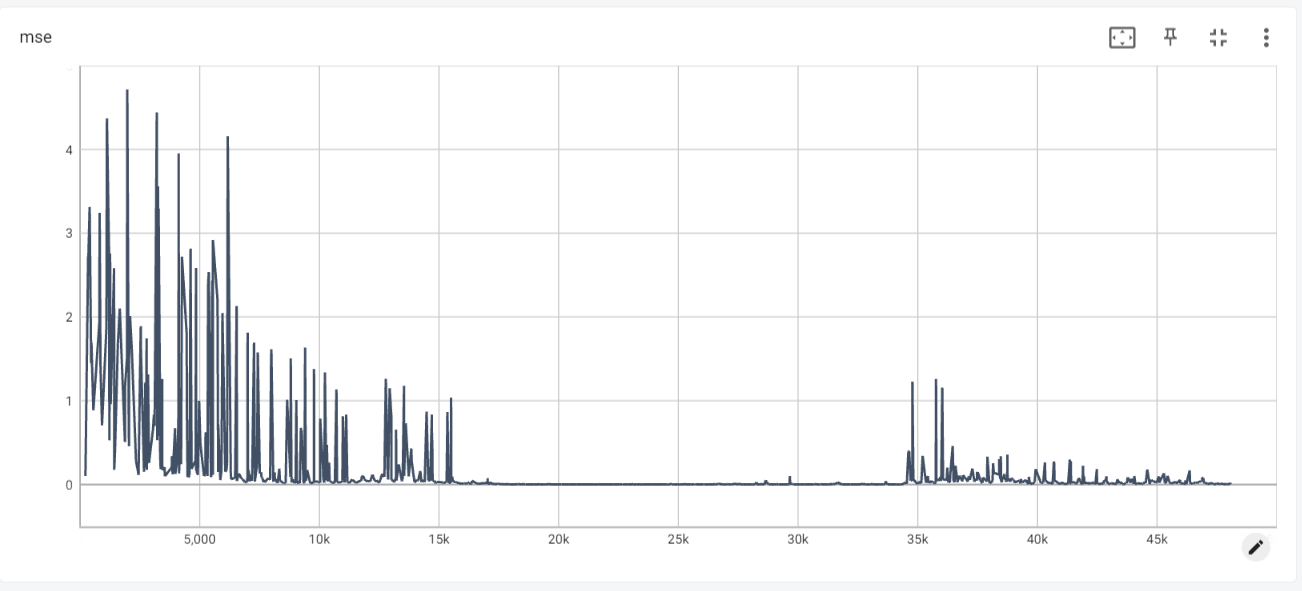
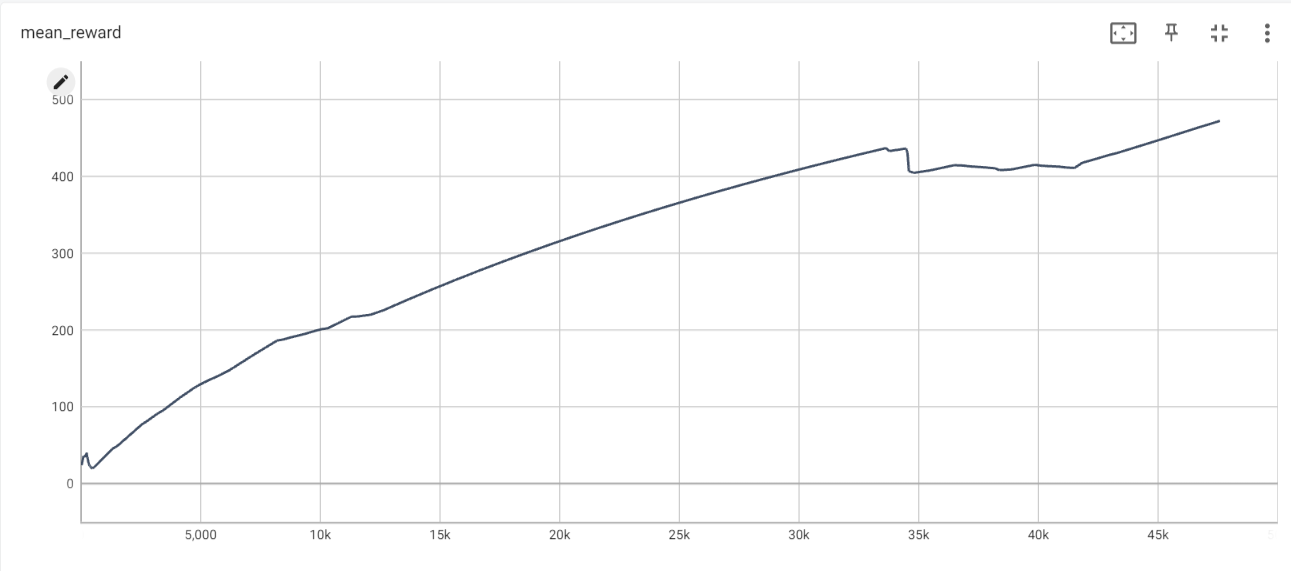


**Optimal hyper-parameters:**

|  |  |  |
| --- | --- | --- |
| Learning rate – policy (actor) | Learning rate – value (critic) | Discount factor |
| 0.0005 | 0.007 | 0.99 |

Under those chosen parameters, we compared the actor-critic, policy gradients with baseline and the original policy gradient algorithms:





We can observe the following:

1. In the manner of **performances**, we can notice that actor-critic algorithm converges way faster than the other algorithms, reaching the desired goal within lower number of episodes. We can also notice that the mean award over 100 consecutive episodes curve more smother than other algorithms curves indicating that the algorithm is more stable. The stability of the actor-critic model is also reflected in the rewards plot, presenting almost constant reward of 500 for last 100 episodes, and moderate rise in results, without aggressive fluctuations comparing to the other algorithms.
2. The actor-critic also hits the goal score after 165 episodes while…
3. Algorithm policy gradients converged after 155.064 seconds,
4. Calculating the time took the algorithms to converge we can notice that again the actor-critic converge much faster (X minutes vs …)..

The improvement of the actor-critic algorithm is obvious since the update of the policy and the value network is done much frequently, in each step and not in each episode end as it is in reinforce algorithms. (Update can be done during episode and not only in the end of it).

**Running the script**:  
To run the above simulation, run the script **actor\_critic.py**.

# Section 4 – References

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