REPORT FOR A REINFORCEMENT LEARNING AGENT ON CART POLE

ARTIFICIAL INTELLIGENCE & MACHINE LEARNING 2022/2023

SAPIENZA
UNIVERSITÀ DI ROMA

Project presented by:

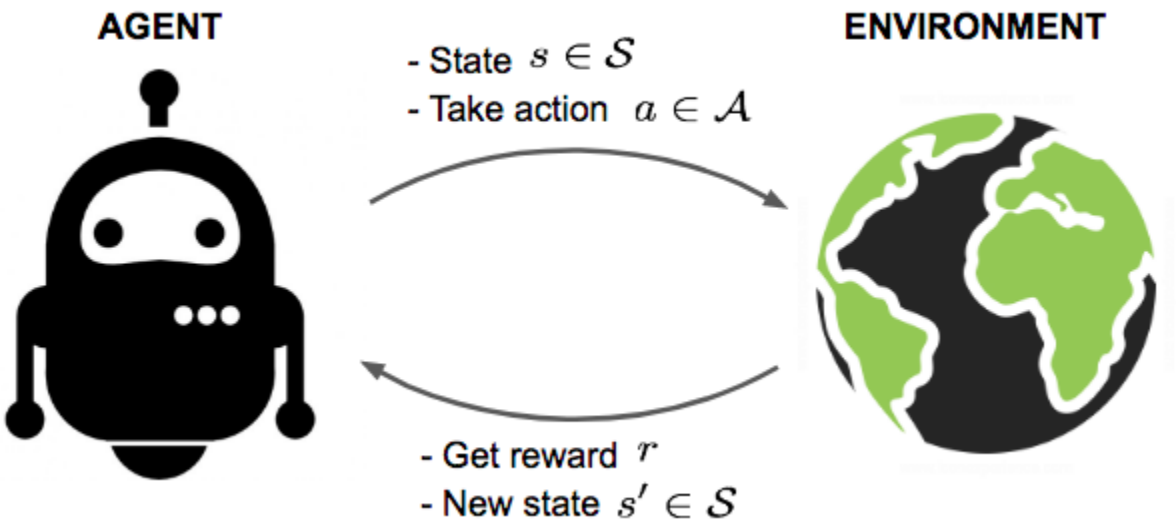
Leonardo Maria Carrozzo - Matricola 2088934
Eugenio Facciolo - Matricola 2065516

INTRODUCTION

Our machine learning project aims to tackle the classic Cart Pole Problem using reinforcement learning techniques. In this problem, a pole is attached to a cart, and the objective is to keep the pole balanced atop the cart by applying forces in either the left or right direction. The primary challenge is to teach an agent to learn an effective policy for this task, where the goal is to maximize the time the pole remains upright.

WHAT IS REINFORCEMENT LEARNING?

Reinforcement learning is a method for solving problems involving continuous decision-making. It is a method employed for discovering an agent's behavioral policy that enables the agent to perceive the current state of the environment and choose the action that maximizes the cumulative reward. The agent interacts with the environment and can execute multiple actions in order to determine the optimal policy by obtaining reward feedback on each action. Two common approaches to reinforcement learning are the Q-table method and the Q-function approximation method.



The Q-table method is a straightforward and tabular representation of the Q-values for each state-action pair in the environment. The Q-value represents the expected cumulative reward an agent can obtain by taking a particular action in a specific state. Initially, the Q-table is empty, and the agent explores the environment to fill it up.

During the training, the agent takes actions based on an exploration-exploitation strategy, such strategy is called epsilon-greedy algorithm. It chooses either the action with the highest Q-value (exploitation) or a random action (exploration) with a certain probability. After each action, the agent updates the Q-value in the Q-table based on the observed reward and the maximum Q-value of the next state. This update follows the Bellman equation, which aims to optimize the Q-values iteratively and convergence is guaranteed only if every possible state-action pair is visited infinitely often.

However, the Q-table method has limitations when the environment becomes large or continuous, as it requires storing and updating values for every state-action pair. This leads us to the Q-function approximation method.

The Q-function approximation method is an approach that uses a function, typically a parametric model, to approximate the Q-values instead of storing them in a tabular form. This function takes the state-action pair as input and outputs an estimation of the Q-value. One common approach for the Q-function approximation is using a neural network as the function approximator, resulting in a technique called deep Q-networks (DQNs).

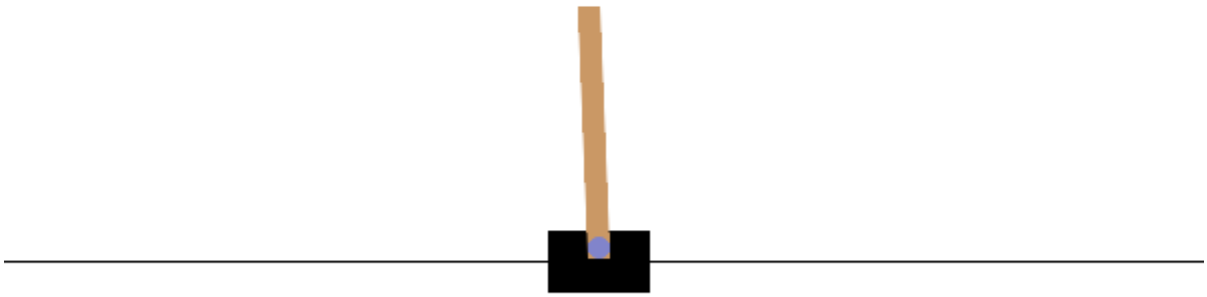
In the Q-function approximation, the agent still follows a similar exploration-exploitation strategy as in the Q-table method, but instead of updating individual Q-values, it trains the neural network to approximate the Q-values based on a loss function. The loss function is defined using the Bellman equation and the observed rewards.

The advantage of the Q-function approximation is that it can handle large and continuous state spaces more efficiently. However, it introduces challenges such as convergence issues and the need for careful selection of the network architecture and training strategies to ensure stability and effective learning.

Overall, both the Q-table method and the Q-function approximation are techniques used in reinforcement learning to enable agents to learn and make decisions in an environment. The Q-table method is suitable for small and discrete environments, while the Q-function approximation is more suitable for larger and continuous environments.

THE CART POLE ENVIRONMENT

The cart pole environment is part of the Classic Control environments of Gymnasium which contains general information about the environment. In particular we have loads of information such as:



- Action Space:
    - 0: Push cart to the left
    - 1: Push cart to the right

Note: The velocity that is reduced or increased by the applied force is not fixed and it depends on the angle the pole is pointing. The center of gravity of the pole varies the amount of energy needed to move the cart underneath it.

- Observation Space (also known as State Space):
    - Cart: The cart has two values: position and velocity.
        - The first one has a minimum value of -4.8 and a maximum value of 4.8.
        - The second one has a minimum value of -Inf and a maximum value

of Inf.
    - Pole: The pole has two values: angle and angular velocity.
        - The first one has a minimum value of ~ -0.418 rad (-24°) and a maximum value of ~ 0.418 rad (24°).
        - The second one has a minimum value of -Inf and a maximum value of Inf.
- Rewards:
    - Since the **goal** is to keep the pole upright for as long as possible, a reward of +1 for every step taken, including the termination step, is allotted.

- Starting State:
    - All observations are assigned a uniformly random value in (-0.05, 0.05).
- Episode End:
    - The episode ends if any one of the following occurs:
        - Termination: Pole Angle is greater than ±12°
        - Termination: Cart Position is greater than ±2.4 (center of the cart reaches the edge of the display)
        - Truncation: Episode length is greater than 500

In this case the only Reinforcement learning algorithm that can be applied to train the agent to solve the Cart Pole Problem is the Q-function approximation to find the optimal policy that maximizes the cumulative rewards. We can't use the Q-table approach because we have continuous values and it would require too much memory to store them all.

SOLUTION ADOPTED

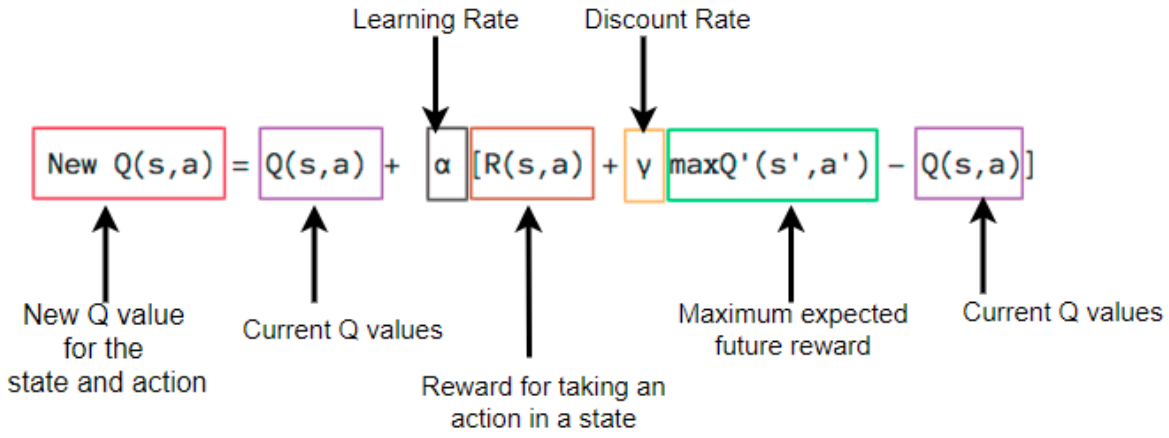
In the following section, we will describe all the algorithms and approaches adopted in order to solve the problem and successfully implement the learning.

LEARNING ALGORITHM (Q-learning)

The Q-learning algorithm is a model-free reinforcement learning algorithm used to find the optimal action-selection policy for an agent in a Markov decision process (MDP). Unlike, S.A.R.S.A., which is an on-policy algorithm. Q-learning is an off-policy algorithm, meaning it learns the optimal policy while exploring and following a different policy (epsilon-greedy). It aims to approximate the optimal values of the Q-function for a given environment.

The key steps of the Q-learning algorithm are as follows:

1. Initialize the Q-values: Initialize the Q-values for all state-action pairs arbitrarily to zero
2. Observe the current state S: The agent observes the current state it is in within the environment
3. Choose an action A: Select an action A based on a policy. In our case the epsilon-greedy policy, meaning there's a balance between exploration and exploitation
4. Execute the chosen action A and observe the next state S' and the reward R: The agent takes the chosen action, transitions to the next state S', and receives a reward R from the environment
5. Update the Q-value for the current state-action pair: Q-values are updated using the following formula:



Where:

- Q(S, A) is the Q-value for the current state-action pair (S, A)
- α (alpha) is the learning rate, which controls how much the Q-values are updated after each step
- R is the observed reward
- γ (gamma) is the discount factor that determines the importance of future rewards
- max(Q(S', a)) is the maximum Q-value over all possible actions in the next state S'

Repeat steps 2-5 until convergence or for a predetermined number of episodes.

REPLAY BUFFER

In reinforcement learning, a replay buffer is a data structure that is commonly used in algorithms such as Q-learning. The replay buffer stores the experience variables (state, action, reward, next state) observed during the agents' interaction with the environment.

We used it as it follows:

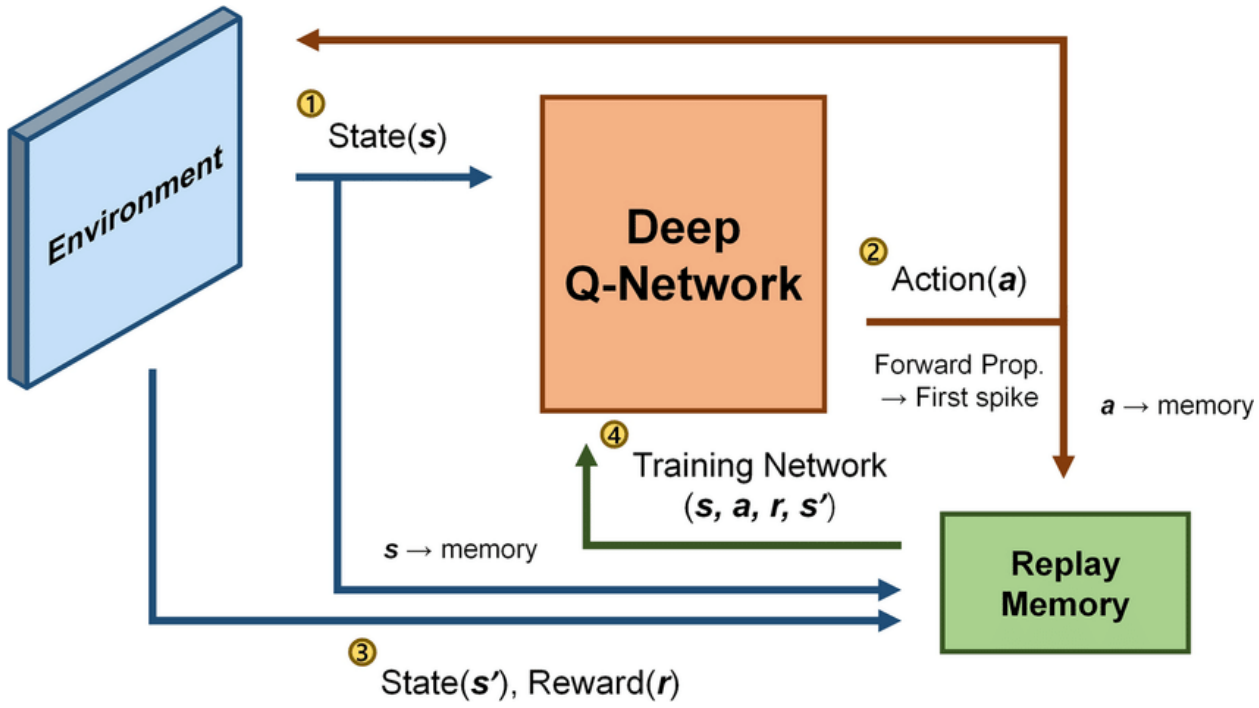
- Initialize an empty replay buffer with a fixed capacity (in particular we used a data structure called deque, which is a stack-like data structure that pops out the oldest values when the maximum capacity is reached, while storing the new ones)
- During the agents' interaction with the environment, we store the

transition (state, action, reward, next state) in the replay buffer
- Finally, when we want to update the values of the Q-function, we sample a batch of transitions from the replay buffer and we use it for updating the Q-function itself

The replay buffer helps to break the temporal correlation between consecutive transitions, making the learning process more stable and efficient. By randomly sampling transitions from the replay buffer, the agent can learn from a diverse set of experiences, reducing the bias that can arise from the sequential correlations. The replay buffer also enables the agent to learn from the past experiences multiple times, as the transitions are stored and reused during the training process. For this reason, it is useful for improving the learning stability and efficiency.

Q-FUNCTION APPROXIMATION

As we said before, Q-function Approximation is an algorithm used to find the optimal action-selection policy by approximating the Q-function rather than explicitly storing it in a lookup table. Instead of maintaining a Q-table where the rows represent states and the columns represent actions, the Q-function approximation algorithm employs a function approximator, such as a neural network, to map states to their corresponding Q-values. The parameters of the function approximator are learned through an iterative process using experiences collected from interacting with the environment. The Q-learning algorithm with Q-function approximation follows those steps:



In our solution, we have create a class, which name is Agent, that works as follows:

We have a bunch of hyper parameters, such as:

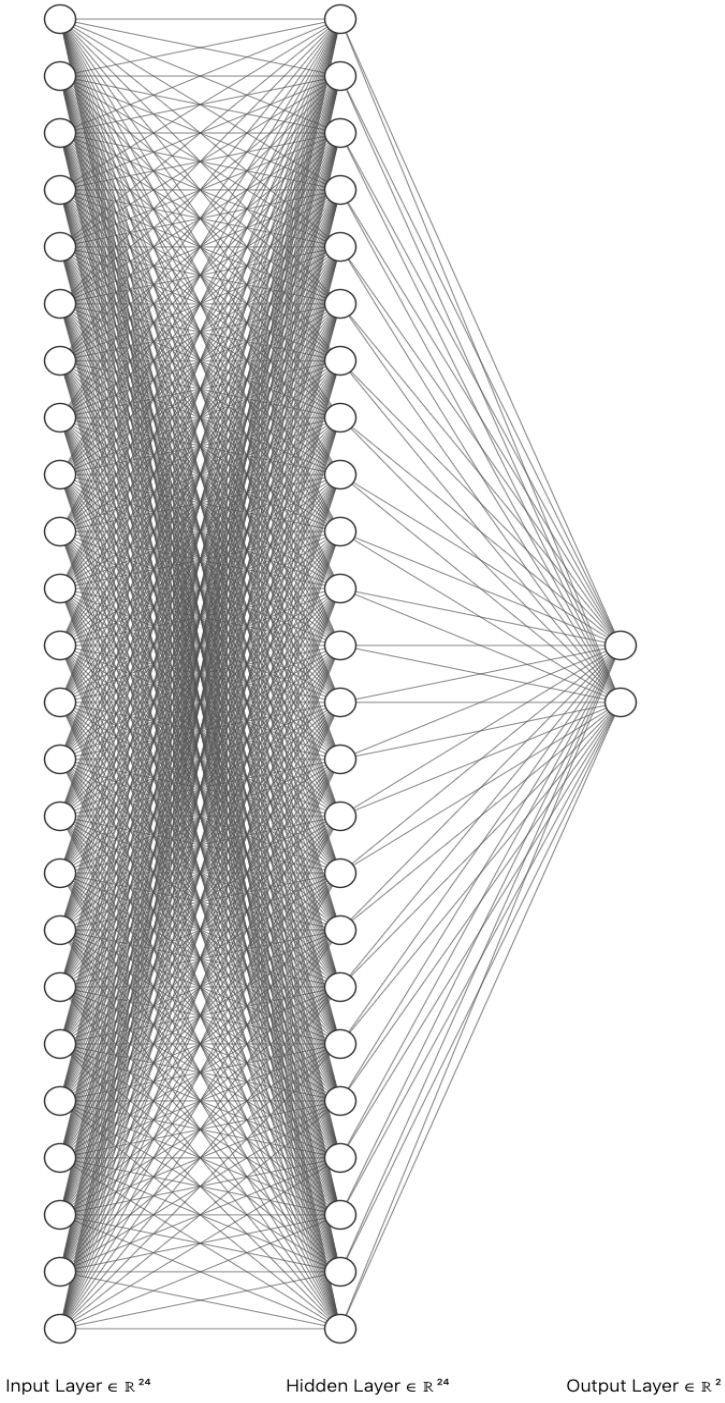
```
train_episodes = 500

test_episodes = 100

max_steps = 600

state_size = env.observation_space.shape[0]

action_size = env.action_space.n

action_space = env.action_space

batch_size = 20
```

Plus other parameters defined in the class' constructor:

```
self.experience = deque(maxlen=100_000)

self.learning_rate = 0.001

self.epsilon = 1

self.max_eps = 1

self.min_eps = 0.01

self.eps_decay = 0.01/3

self.gamma = 0.9

self.state_size = state_size

self.action_size = action_size

self.action_space = action_space
```

- train_episodes: Maximum number of episodes used during the training.
- test_episodes: Maximum number of episodes used during the test/evaluation of the neural network.
- max_steps: Maximum number of steps that can be performed during the training/testing. Basically it's the maximum number of actions that can be performed during a single episode.
- state_size: The size of the state object. It will be 4 in our case.
- action_size: The size of the action object. It will be 2 in our case since we can perform only 2 actions: move left or move right.
- action_space: It will give us Discrete(2) since we can only perform 2 actions, as we have said before.
- batch_size: Size of the batch used for the training. It specifies the number or experiences sampled from the replay buffer at each training iteration.
- self.experience: Our data structure in which we will store the experience for the replay buffer. It has a fixed max length of 100.000 elements. When it reaches the maximum of it, and when we insert a new element in it, the oldest one will be popped off.
- learning_rate: Rate that controls how quickly our neural network updates its weights (how quickly it learns).
- self.epsilon: Exploration rate variable, this value will be changed as the algorithm progresses according to a decay strategy.
- self.max_eps: The maximum exploration rate (epsilon) used in the epsilon-greedy exploration algorithm. It represents the probability of selecting a random action during the exploration phase.
- self.min_eps: The minimum exploration rate (epsilon).
- self.eps_decay: The rate by which the exploration rate (epsilon) decreases its value over time. It determines how quickly the algorithm transitions from exploration to exploitation.
- self.gamma: The discount factor used in the Q-learning algorithm. It determines the weight given to future rewards compared to the current rewards.

For the architecture of our neural network we have tried a bunch of different configurations, the one that have given us the best results is the following:



To do so, we have create a method for our agent:

```
def build_model(self):

   model = Sequential()

   model.add(Dense(24, input_dim=self.state_size, activation='relu'))

   model.add(Dense(24, activation='relu'))

   model.add(Dense(self.action_size, activation='linear'))

   model.compile(loss='mse',
optimizer=Adam(learning_rate=self.learning_rate))

   return model
```

In particular, this method creates a neural network model that approximates the Q-function for a given environment, in our case the cart pole environment. First we initialize our model as a sequential model, which represents a linear stack of layers. Each layer in the model transforms the input data to produce meaningful outputs. In the case of the Q-function approximation, we use dense layers, also known as fully connected layers. Those are added to the model one by one by using the add method of the model. These layers consist of multiple neurons (units) that are fully connected to the neurons (units) in the previous layer. Fully connected neurons means that each neuron is connected to every other neuron of the previous layer, one by one (as it's clearly understandable by looking at the above picture that displays our neural networks' architecture). This connectivity lets the neural network produce complex relationships between inputs and outputs. Also, we have used the ReLU (Rectified Linear Unit) as activation function inside our dense layers, which determines non-linear behavior of the neurons and introduces non-linearity in our model. In the last layer we have used a linear activation function. After we have added all the layers to the model, we compile it using the compile method of the model, giving some arguments to it. In particular loss and optimizer arguments are given. The loss function measures the discrepancy between the predicted Q-values and the target Q-values. In our case, we have used the Mean Squared Error (MSE) loss function. The optimizer is responsible for updating the models' weights during the training process to minimize the loss. In particular, we have used the Adam optimizer, which is known for its efficiency and effectiveness.

Now we dive into how we train the neural network:

```
# Create an instance of our agent's class
agent = Agent(state_size, action_size, action_space)


# Train our model
for episode in range(train_episodes):
   state, _ = env.reset()
   state = np.reshape(state, [1, state_size])


   for step in range(max_steps):
       action = agent.action(state)
       new_state, reward, terminated, _, _ = env.step(action)
       new_state = np.reshape(new_state, [1, state_size])
       agent.add_experience(new_state, reward, terminated, state,
action)
       state = new_state


       if terminated:
           break


   if len(agent.experience) > batch_size:
       agent.replay(batch_size)
```

First we define an instance of our agent. Then we have a for loop that goes from 0 to the number of maximum training episodes we set, in our case 500. For each episode, we reset the state of the environment through the reset method of the environment. It's essential because it allows the agent to interact with the environment. Within every episode our agent selects an action given its current state and the exploration-exploitation strategy determined by the epsilon-greedy algorithm. To do so, we use the action method of our agent.

This method balances exploration and exploitation. It randomly selects an action from the action space with a certain probability, given by the epsilon, for the exploration, but it exploits the learned Q-function approximation to choose the action with the highest predicted Q-value when the random values is less than the epsilons' value.

```
def action(self, state):
   if np.random.rand() < self.epsilon:
       return self.action_space.sample()
   return np.argmax(self.model.predict(state)[0])
```

After that we execute it by using the step method of the environment. This method returns us the new state, the reward and a boolean (terminated) that is true if the pole has fallen or false if not. We, then, add experience to our agent, providing the new state, the reward, the terminated boolean, the state and the action. Those information are stored in the replay buffer that we have defined in our constructor under the experience property. The replay buffer has a fixed length, it cannot exceed it.

Then the current state is updated with the new state obtained before. If the episode is terminated we break our loop and exit the inner loop that loops for every step. Also, if there are enough experiences in the replay buffer, we call the replay method of the agent. In this way we train our model, updating the Q-function approximation using a batch of experiences.

```
def replay(self, batch_size):

   minibatch = random.sample(self.experience, batch_size)

   for new_state, reward, terminated, state, action in minibatch:

       target = reward

       if not terminated:

           target += self.gamma *
np.max(self.model.predict(new_state)[0])

       target_function = self.model.predict(state)

       target_function[0][action] = target

       self.model.fit(state, target_function, epochs=1, verbose=0)



   if self.epsilon > self.min_eps:

       self.epsilon = (self.max_eps - self.min_eps) *
np.exp(-self.eps_decay * episode) + self.min_eps
```

This method is responsible for training the Q-function approximation model using a batch of experiences obtained from the replay buffer. Those are the steps that are taken in it:

- Batch sampling: A random batch of experiences is sampled from the replay buffer (experience buffer)
- Q-value composition: For each experience in the batch, the Q-value for the current action is obtained by predicting the Q-values for the current state. We do this by first assigning the reward of the single experience to the Q-value. Then if the episode is not terminated we predict the Q-value using the Q-learning rule:
    - We add to the target Q-value, the cumulative reward given by the Bellman equation.
        - We add the discount factor and we multiply it with the maximum value of the prediction (the probability of each action) on the new state
    - We make a prediction on the current state as target function

    - Then we assign the updated Q-value to the corresponding action in the predicted actions of the current state (target_function = self.model.predict(state))
- Model training: The model is trained using the state (input) and the target function (output) from the batch using the model.fit(state, target_function, epochs=1, verbose=0) method. The training process aims to minimize the discrepancy between the predicted Q-values and the target Q-values using the Mean Squared Error (MSE) loss function.
- Epsilon update: The epsilon value is updated according to the epsilon decay formula. It gradually decreases the epsilon value exponentially as the number of the current episode increases.

Q-FUNCTION APPROXIMATION PERFORMANCE TESTS

In order to improve the performance of the learning process, the fine-tuning of the hyper parameters used to calculate the Q-function is necessary.

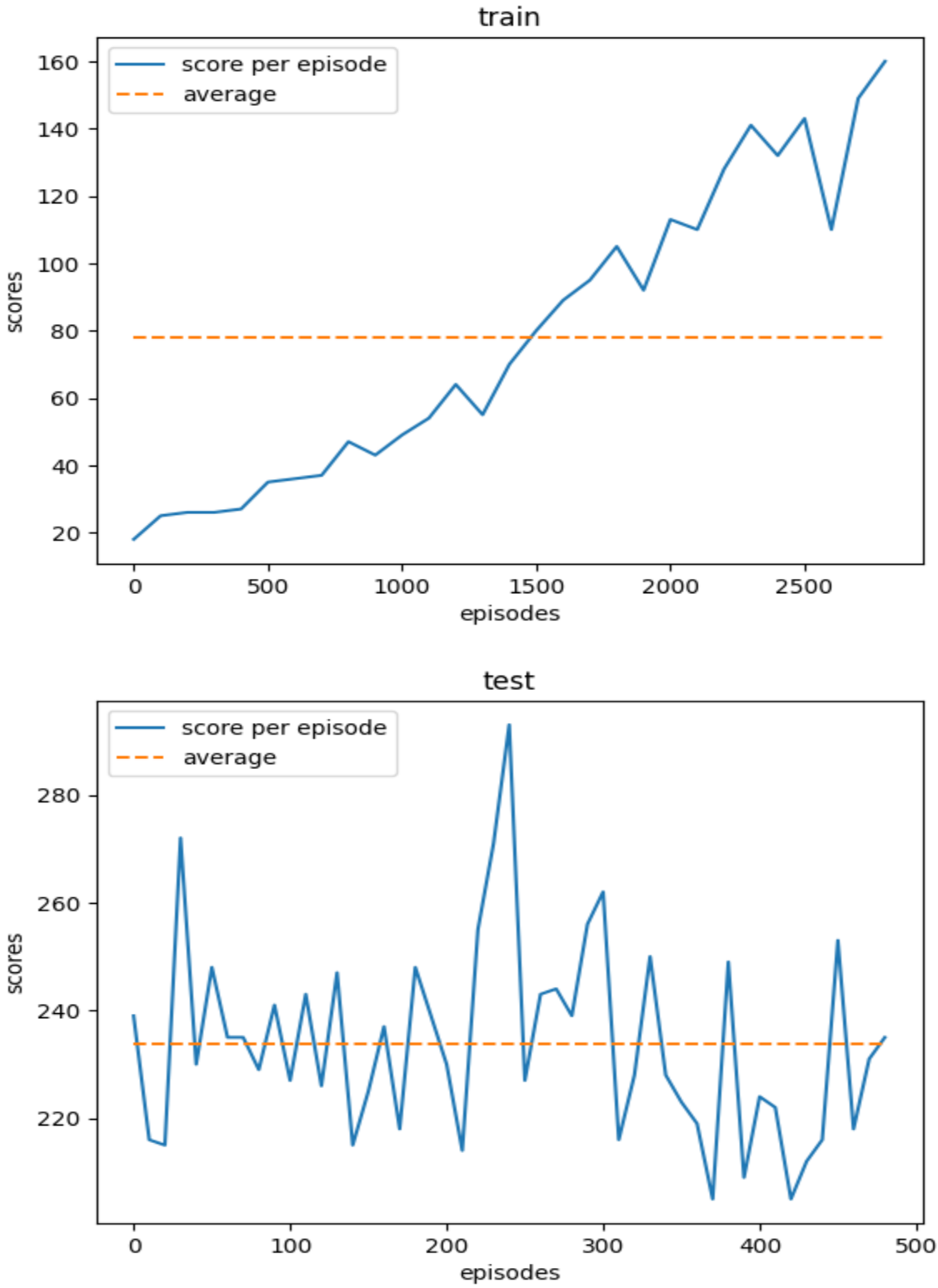
We have tried a bunch of these and conducted some experiments. In particular, we have tested the following configurations:

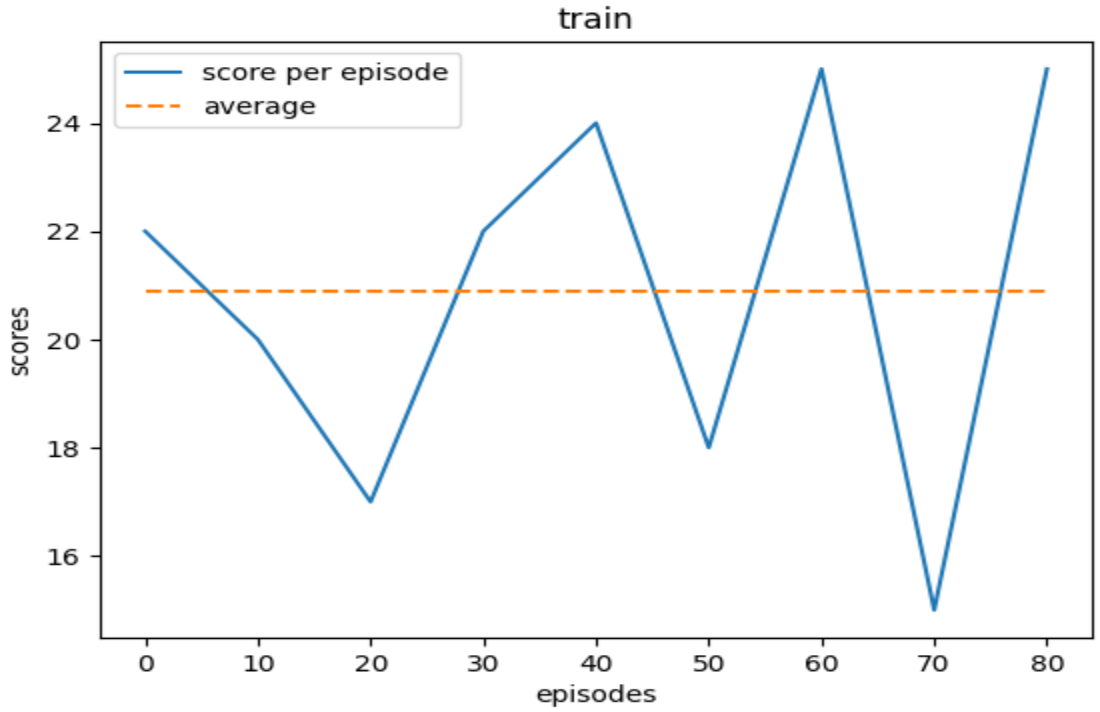
1. train_episodes = 3000, batch_size = 32, deque maxlen = 10.000, learning_rate = 0.001, epsilon_decay = 0.001/3, gamma = 0.9
2. train_episodes = 3000, batch_size = 32, deque maxlen = 3.000, learning_rate = 0.001, epsilon_decay = 0.001/3, gamma = 0.95
3. train_episodes = 500, batch_size = 32, deque maxlen = 10.000, learning_rate = 0.001, epsilon_decay = 0.001/3, gamma = 0.95
4. train_episodes = 500, batch_size = 32, deque maxlen = 10.000, learning_rate = 0.001, epsilon_decay = 0.001/3, gamma = 0.9
5. train_episodes = 500, batch_size = 32, deque maxlen = 2.000, learning_rate = 0.001, epsilon_decay = 0.001/3, gamma = 0.9
6. train_episodes = 500, batch_size = 32, deque maxlen = 10.000, learning_rate = 0.001, epsilon_decay = 0.01/3, gamma = 0.9
7. train_episodes = 500, batch_size = 32, deque maxlen = 100.000, learning_rate = 0.001, epsilon_decay = 0.01/3, gamma = 0.9
8. train_episodes = 500, batch_size = 32, deque maxlen = 1.000.000, learning_rate = 0.001, epsilon_decay = 0.01/3, gamma = 0.9
9. train_episodes = 500, batch_size = 32, deque maxlen = 50.000, learning_rate = 0.001, epsilon_decay = 0.01/3, gamma = 0.9

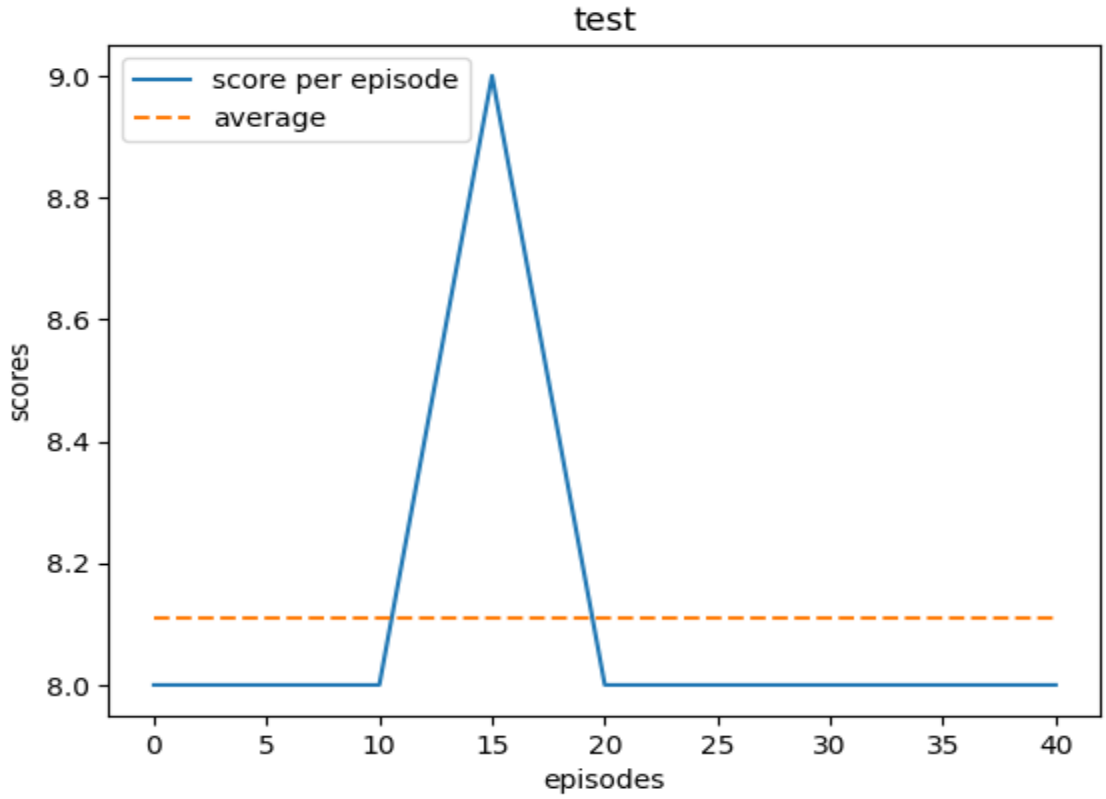
10. train_episodes = 500, batch_size = 32, deque maxlen = 200.000, learning_rate = 0.001, epsilon_decay = 0.01/3, gamma = 0.9
11. train_episodes = 500, batch_size = 20, deque maxlen = 100.000, learning_rate = 0.001, epsilon_decay = 0.01/3, gamma = 0.9
12. train_episodes = 500, batch_size = 32, deque maxlen = 100.000, learning_rate = 0.001, epsilon_decay = 0.01/3, gamma = 0.9
13. train_episodes = 500, batch_size = 20, deque maxlen = 100.000, learning_rate = 0.001, epsilon_decay = 0.01/3, gamma = 0.95
14. train_episodes = 500, batch_size = 20, deque maxlen = 100.000, learning_rate = 0.0001, epsilon_decay = 0.01/3, gamma = 0.9
15. train_episodes = 500, batch_size = 20, deque maxlen = 100.000, learning_rate = 0.001, epsilon_decay = 0.01, gamma = 0.9
16. train_episodes = 500, batch_size = 20, deque maxlen = 100.000, learning_rate = 0.001, epsilon_decay = 0.001/3, gamma = 0.9
17. train_episodes = 500, batch_size = 20, deque maxlen = 100.000, learning_rate = 0.001, epsilon_decay = 0.01/3, gamma = 0.9
18. train_episodes = 500, batch_size = 20, deque maxlen = 100.000, learning_rate = 0.001, epsilon_decay = 0.01/3, gamma = 0.8
19. train_episodes = 500, batch_size = 20, deque maxlen = 100.000, learning_rate = 0.001, epsilon_decay = 0.01/6, gamma = 0.9

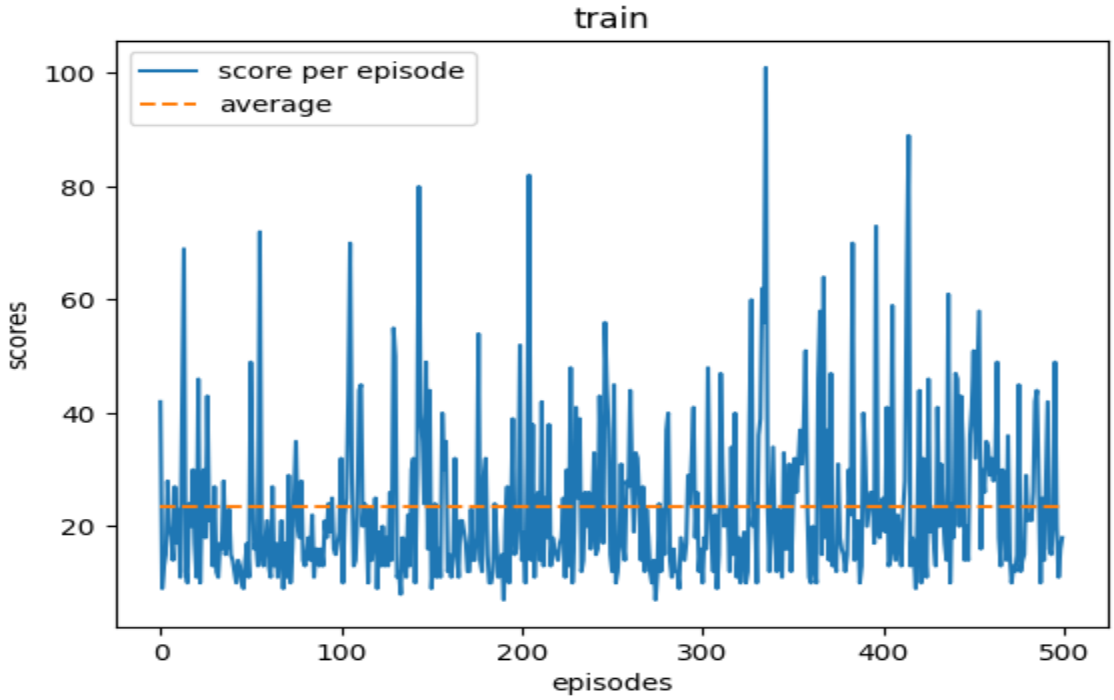
For each configuration, we have plotted the cumulative reward over the number of episodes for both the training and the testing. The best configuration we have found is: train_episodes = 500, batch_size = 20, deque maxlen = 100.000, learning_rate = 0.001, epsilon_decay = 0.01/3, gamma = 0.9. In particular, this configuration achieved an average of 490 points over 100 episodes in the testing (evaluation) of the neural network performance.

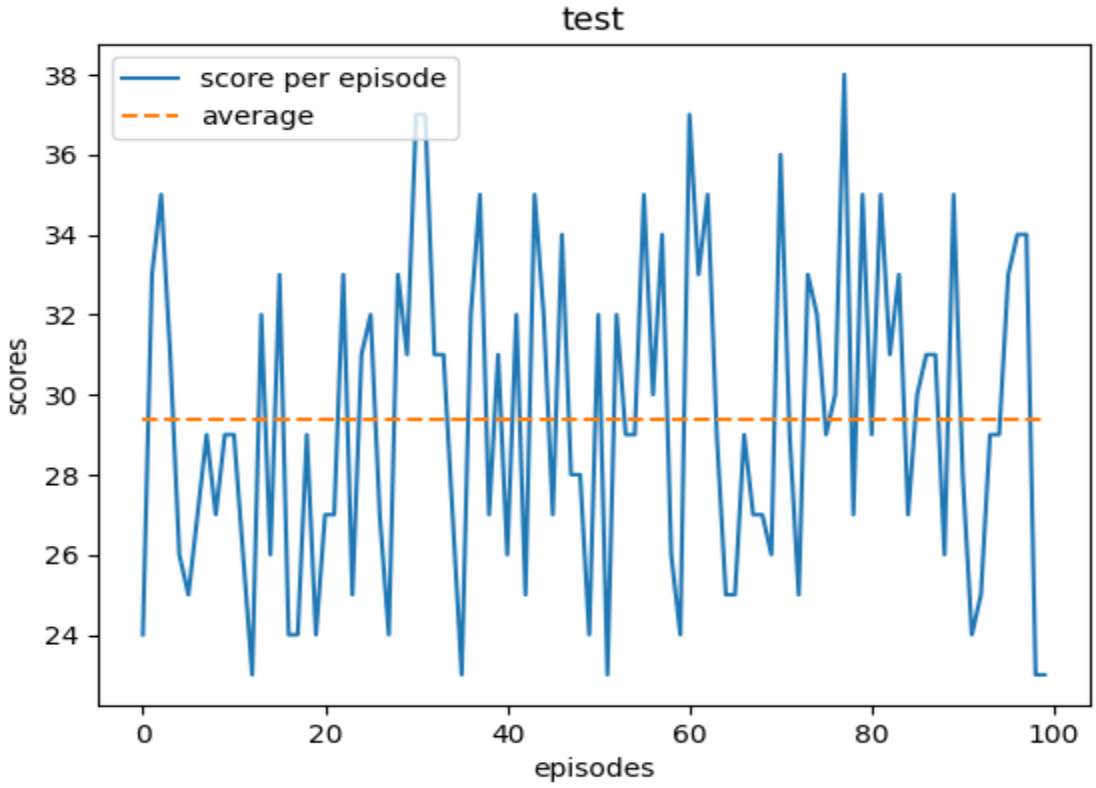
1


train
score per episode
average
scores
episodes
0
500
1000
1500
2000
2500
20
40
60
80
100
120
140
160
test
score per episode
average
scores
episodes
0
100
200
300
400
500
220
240
260
280

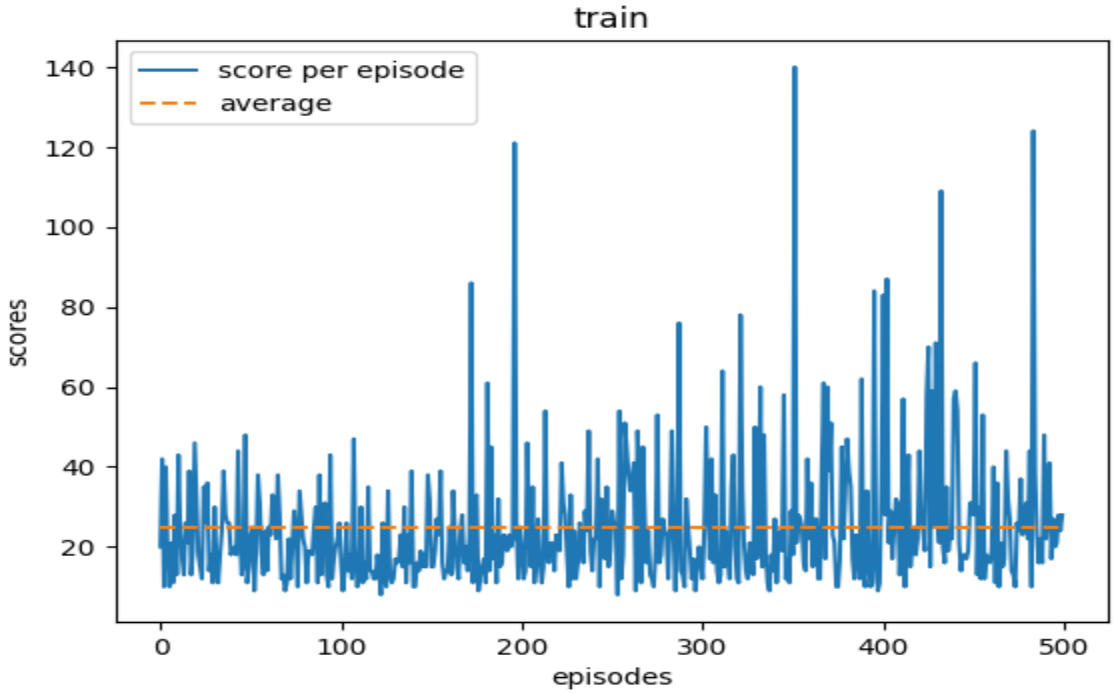

train
score per episode
average
scores
episodes
24
22
20
18
16
0
10
20
30
40
50
60
70
80

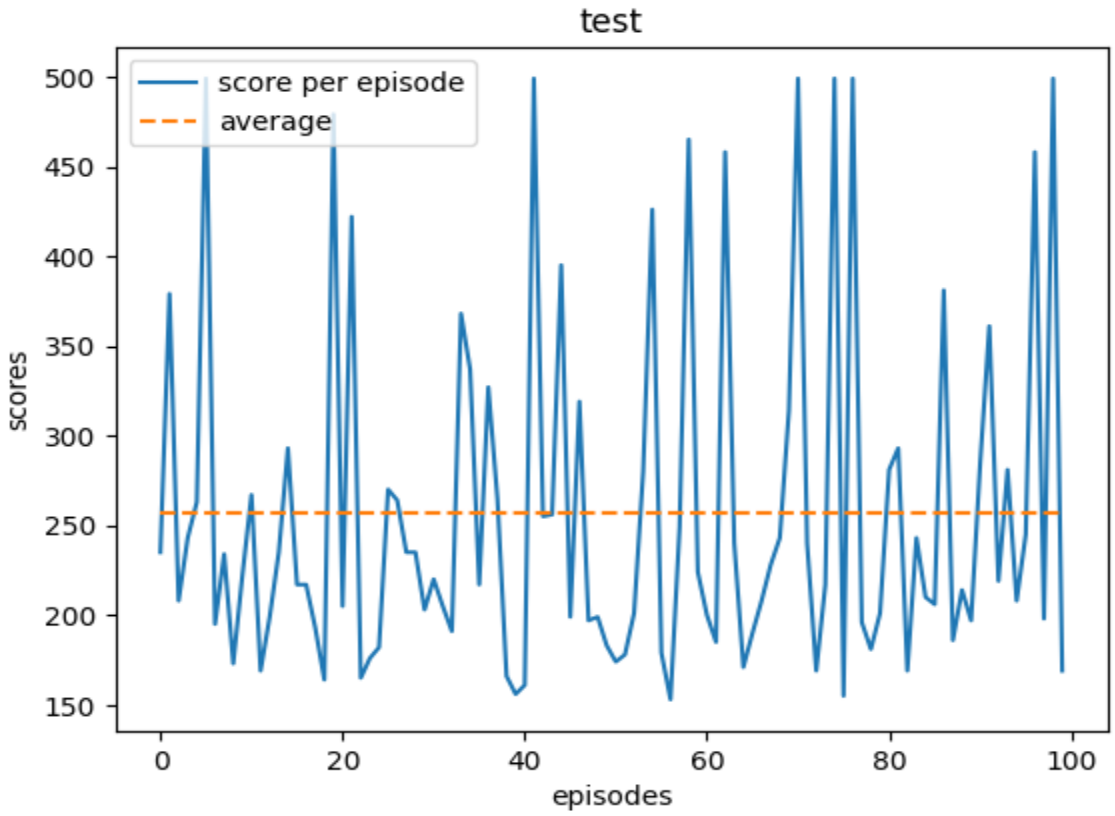

test
score per episode
average
scores
episodes
9.0
8.8
8.6
8.4
8.2
8.0
0
5
10
15
20
25
30
35
40

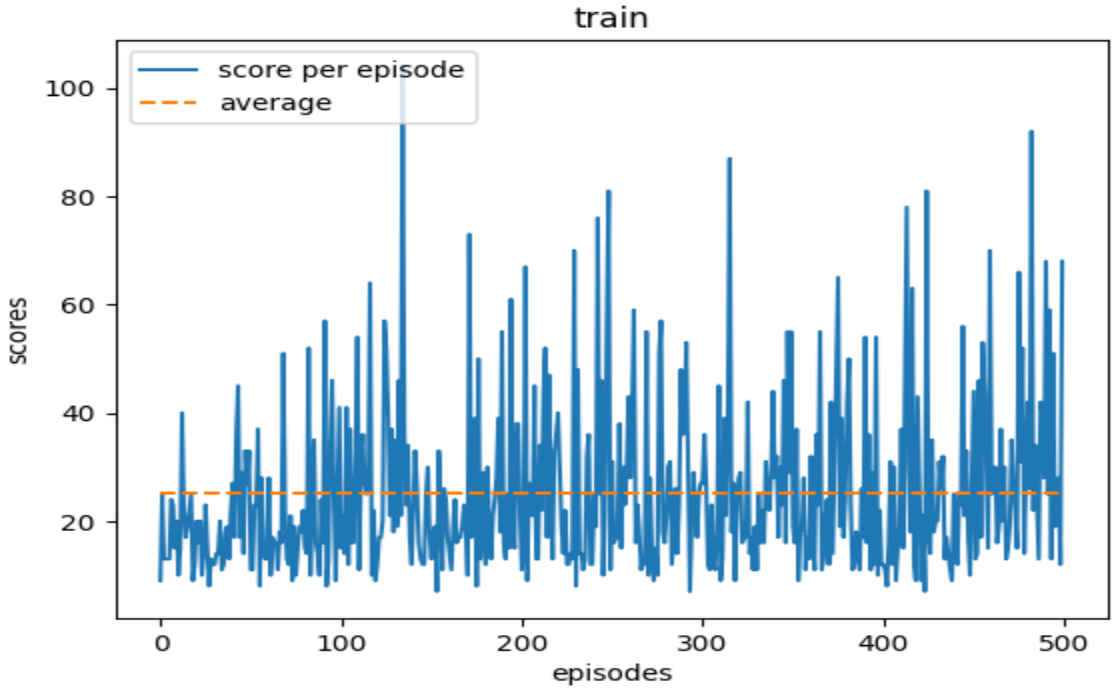

train
score per episode
average
scores
episodes
0
100
200
300
400
500
20
40
60
80
100

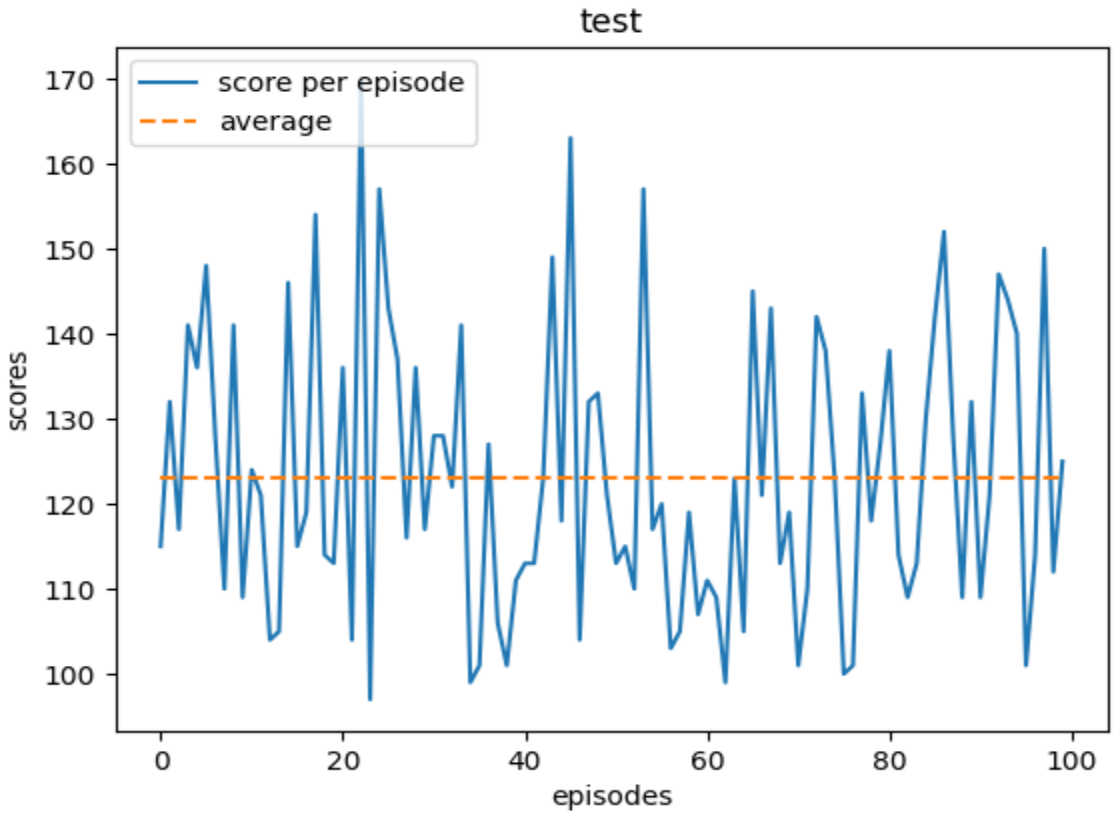

test
score per episode
average
scores
episodes
0
20
40
60
80
100
24
26
28
30
32
34
36
38

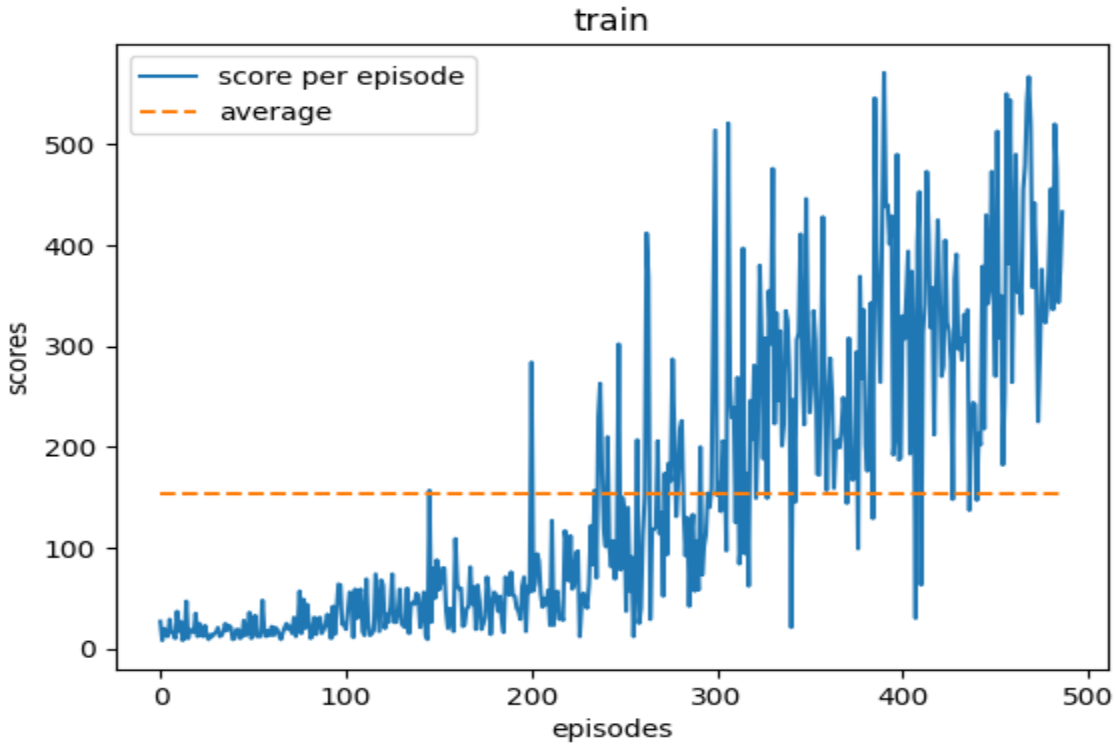
4

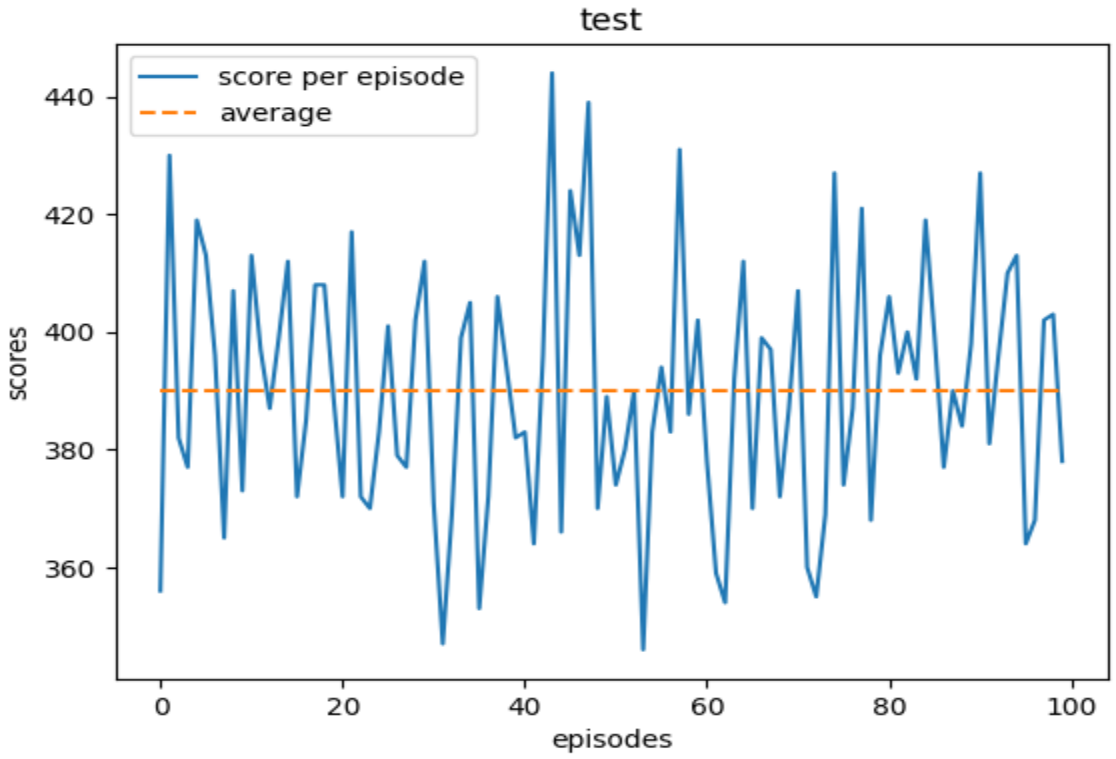

train
score per episode
average
scores
episodes
0
100
200
300
400
500
20
40
60
80
100
120
140

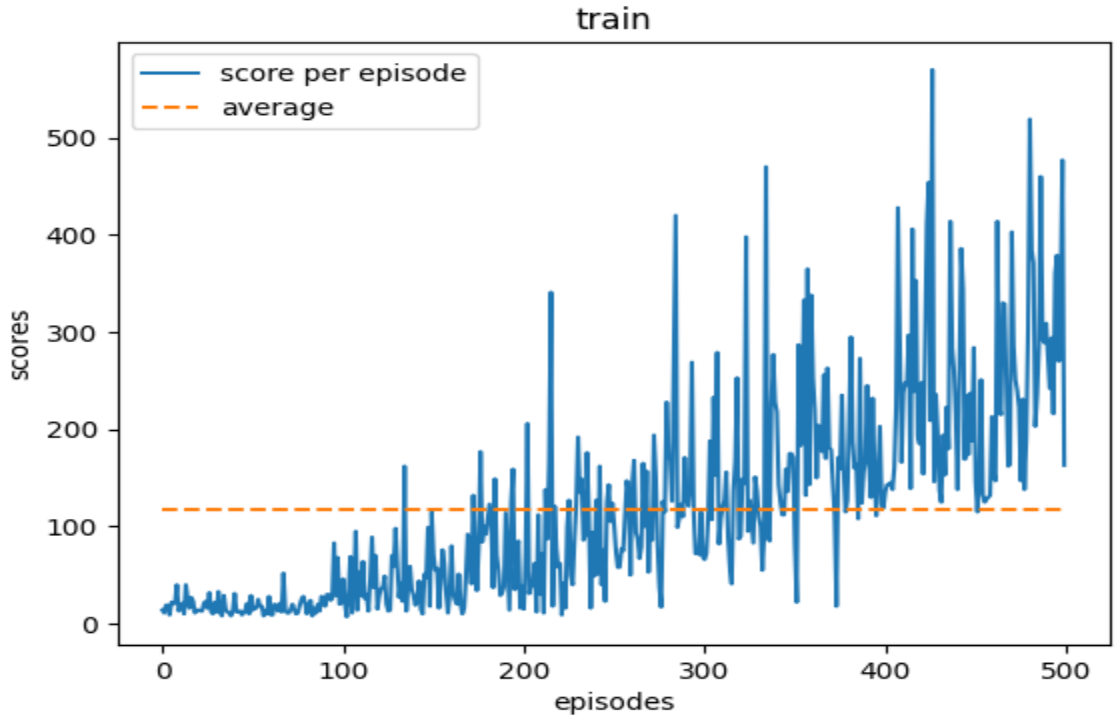

test
score per episode
average
scores
episodes
0
20
40
60
80
100
150
200
250
300
350
400
450
500

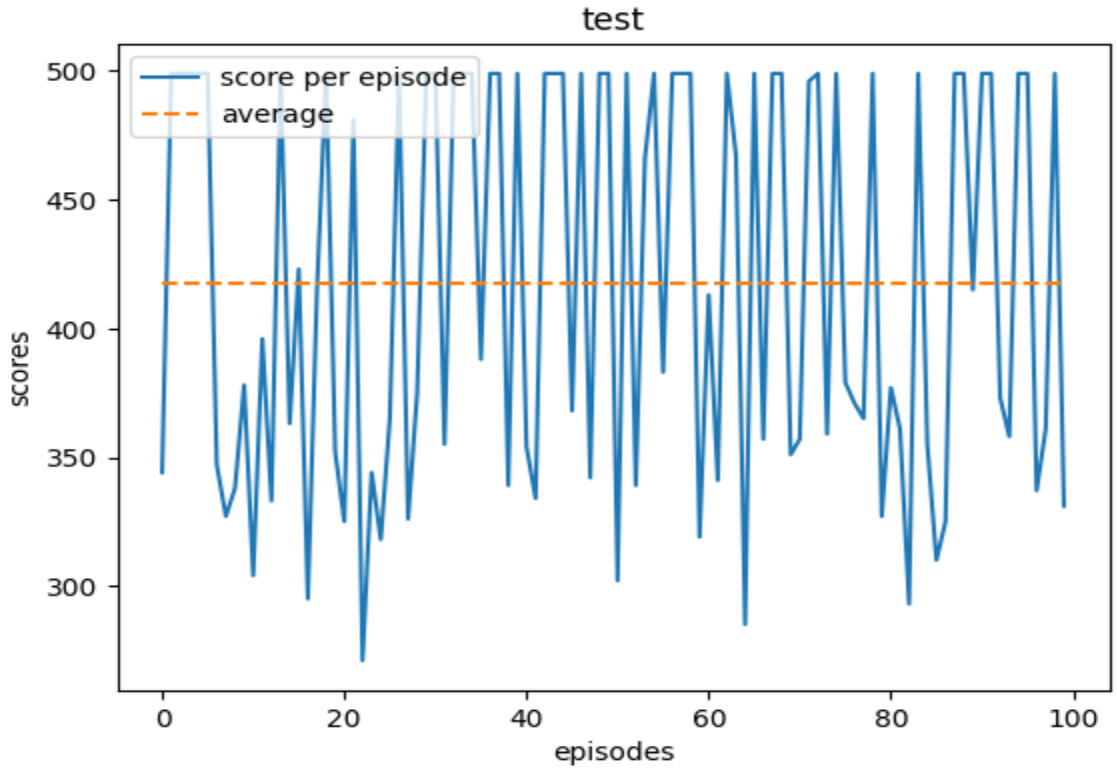

train
score per episode
average
scores
episodes
0
100
200
300
400
500
20
40
60
80
100

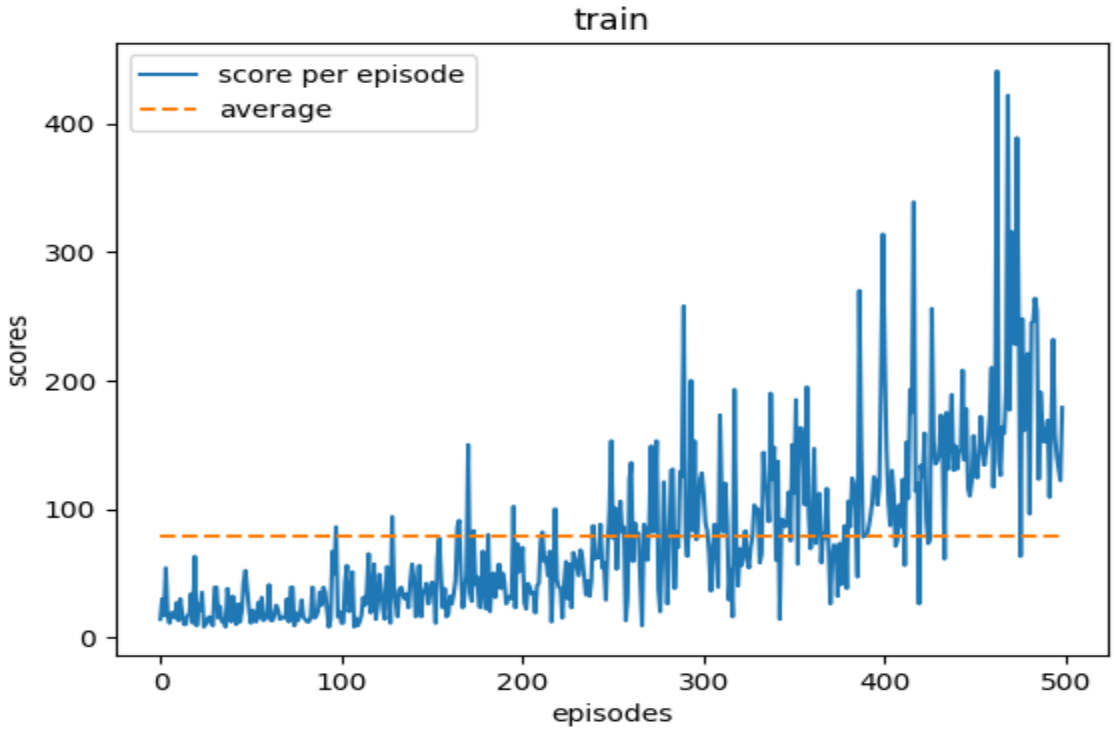

test
score per episode
average
scores
episodes
0
20
40
60
80
100
100
110
120
130
140
150
160
170

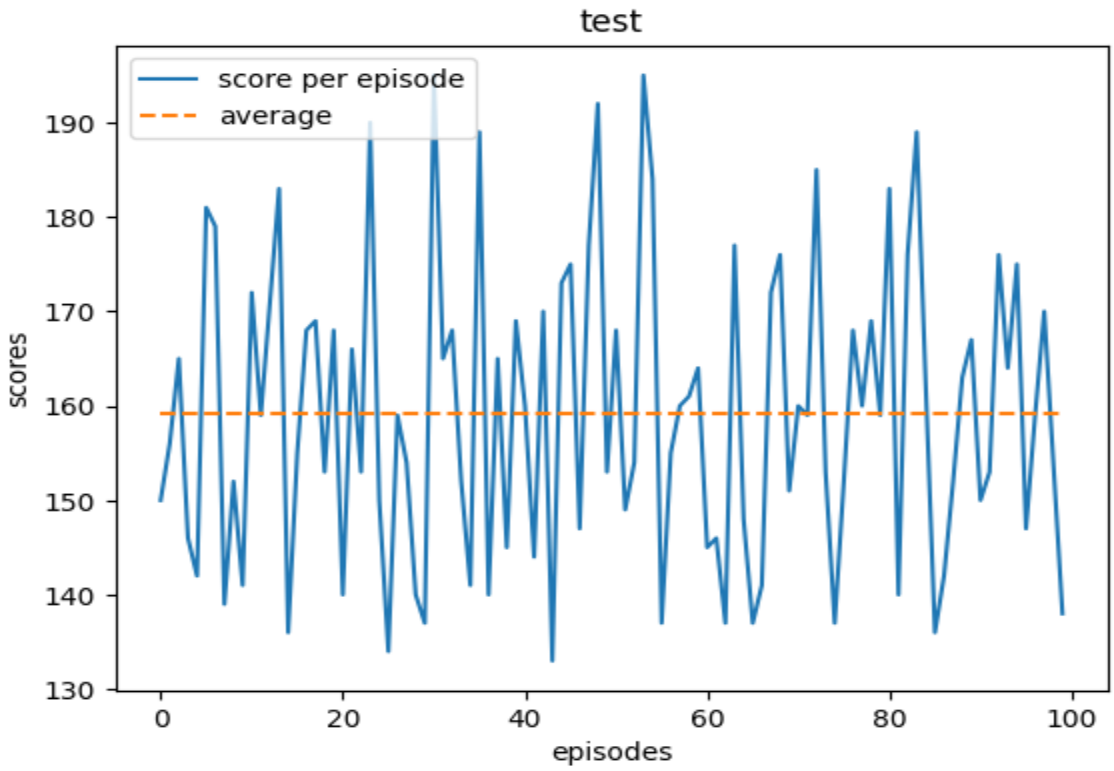

train
score per episode
average
scores
episodes
0
100
200
300
400
500

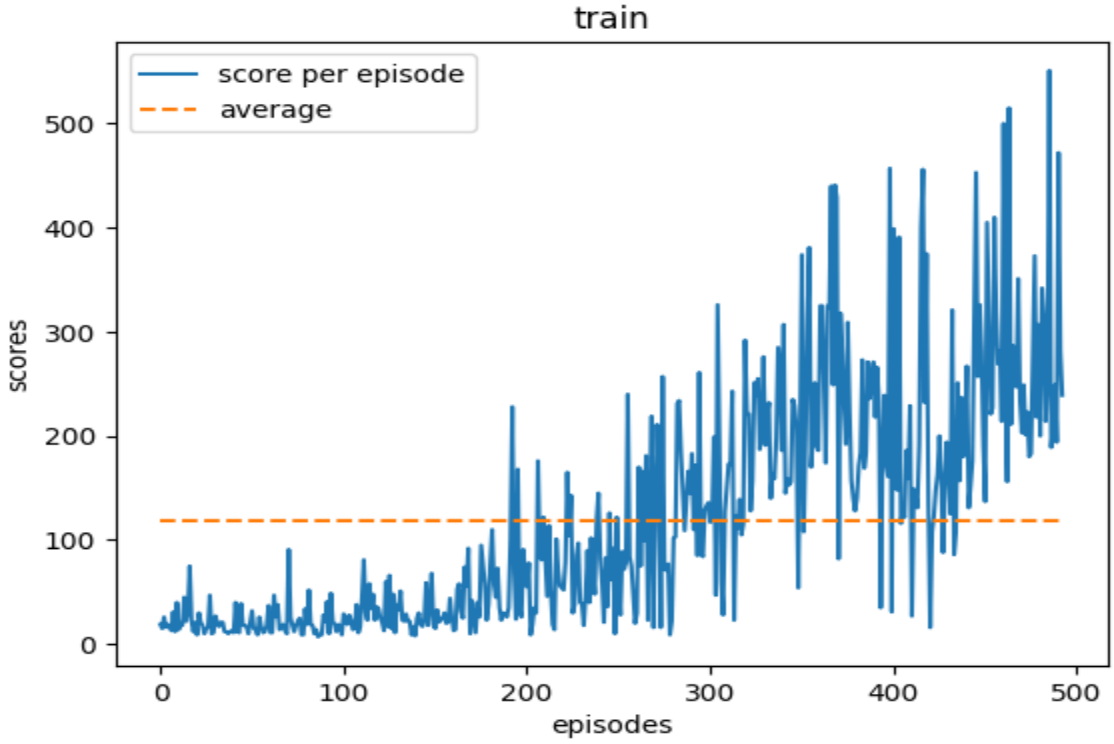

test
score per episode
average
scores
episodes
360
380
400
420
440
0
20
40
60
80
100

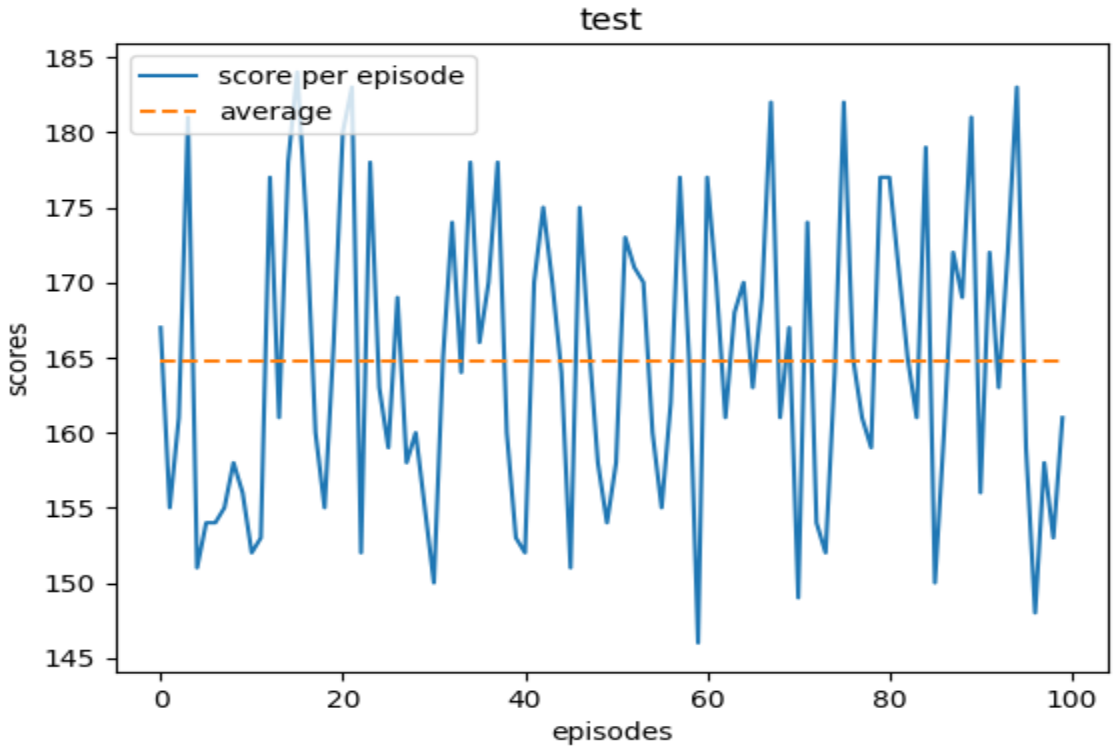

train
score per episode
average
scores
episodes
0
100
200
300
400
500

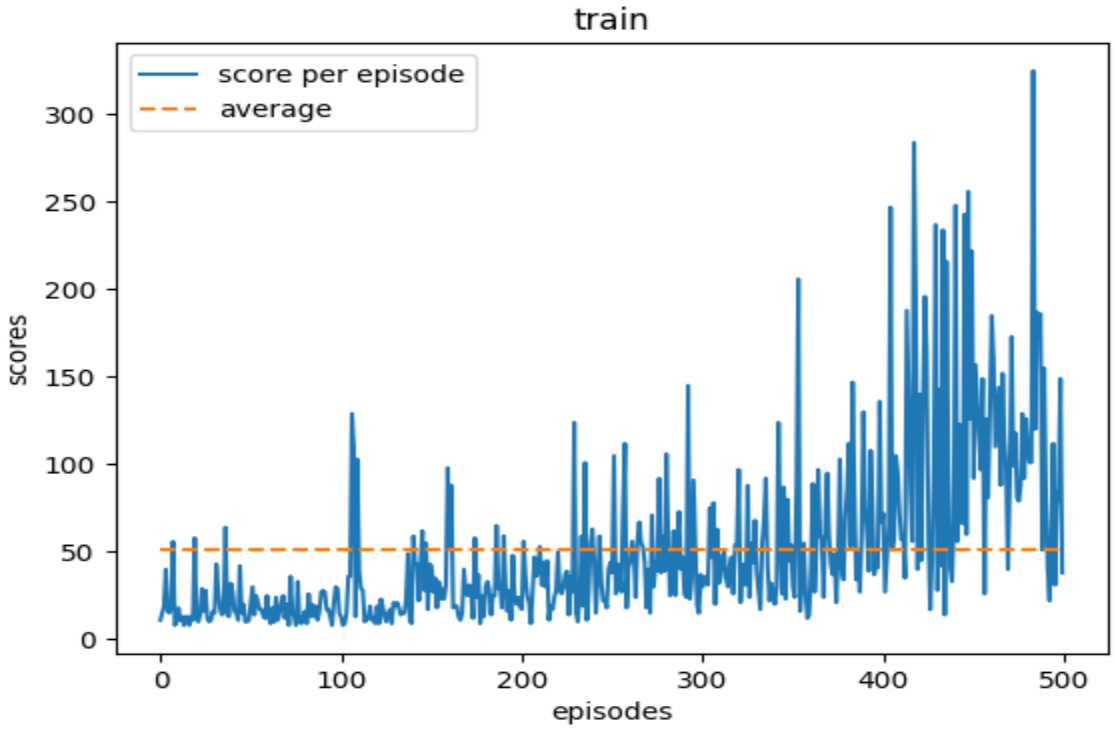

test
score per episode
average
scores
episodes
300
350
400
450
500
0
20
40
60
80
100

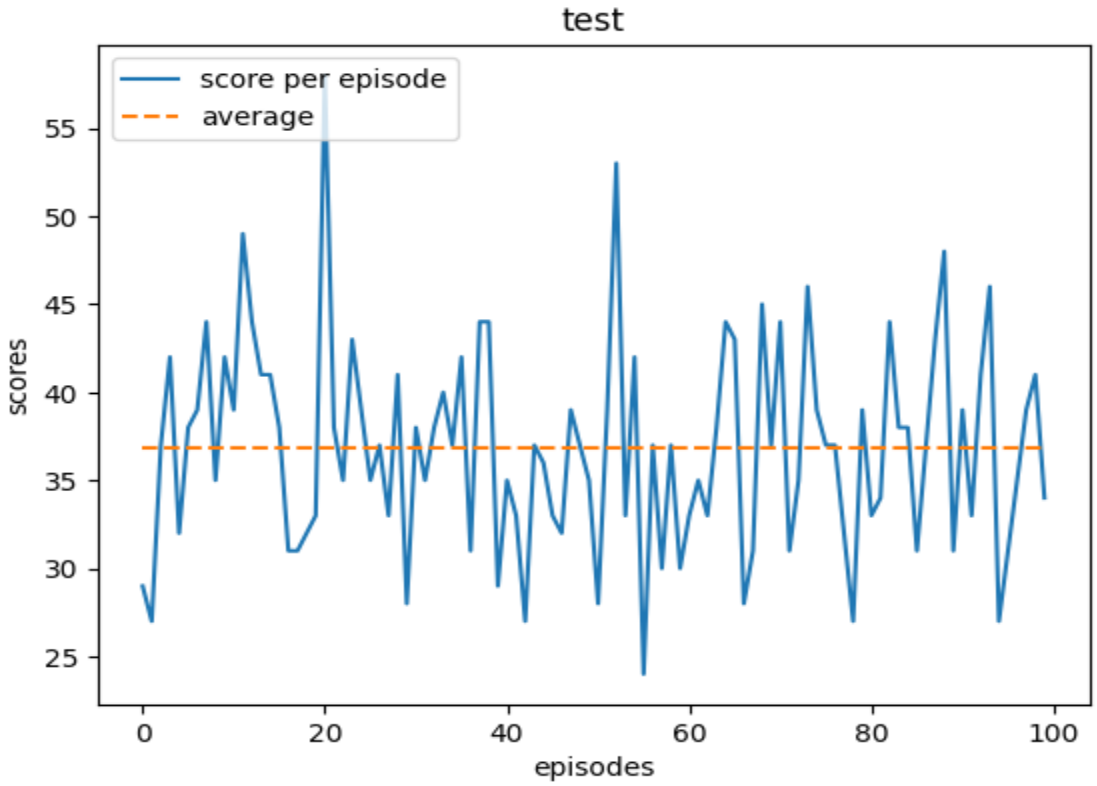

train
score per episode
average
scores
episodes
0
100
200
300
400
500

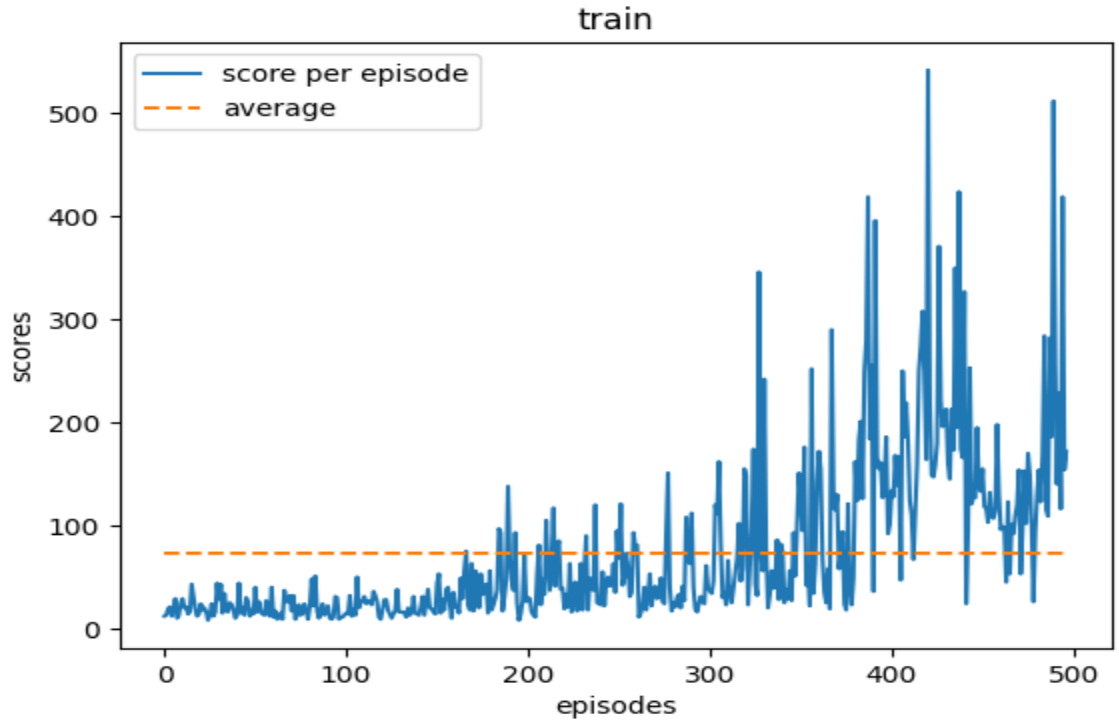

test
score per episode
average
scores
episodes
130
140
150
160
170
180
190
0
20
40
60
80
100

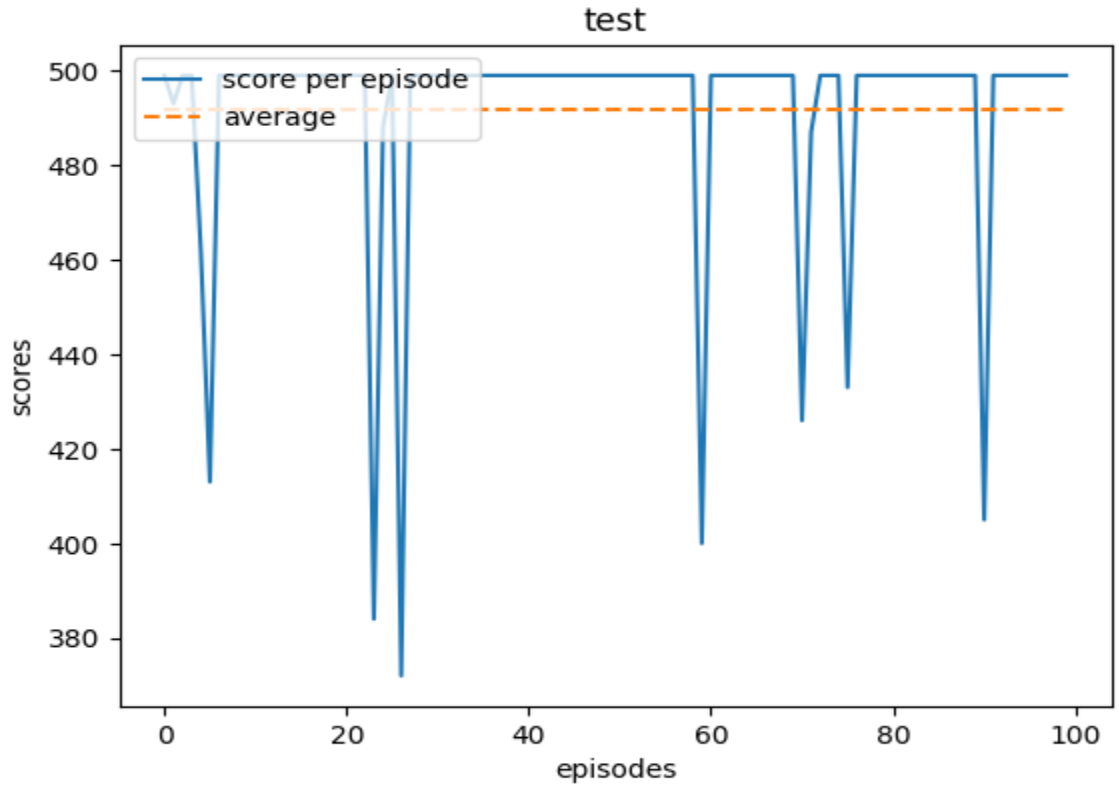

train
score per episode
average
scores
episodes
0
100
200
300
400
500

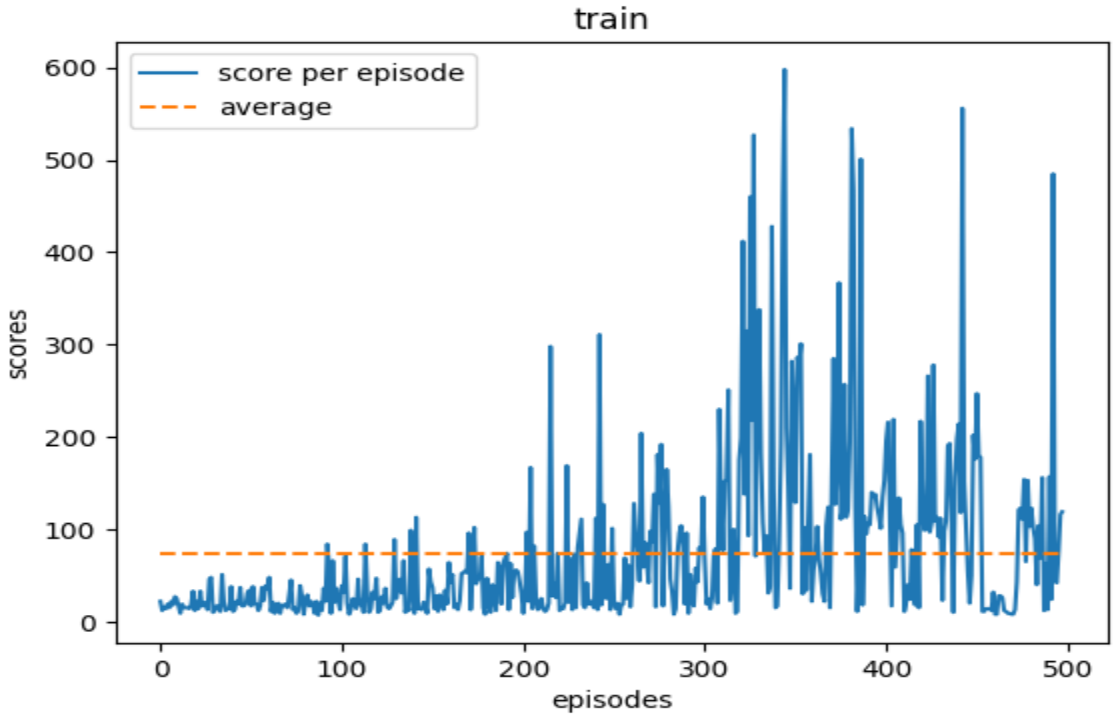

test
score per episode
average
scores
episodes
145
150
155
160
165
170
175
180
185
0
20
40
60
80
100

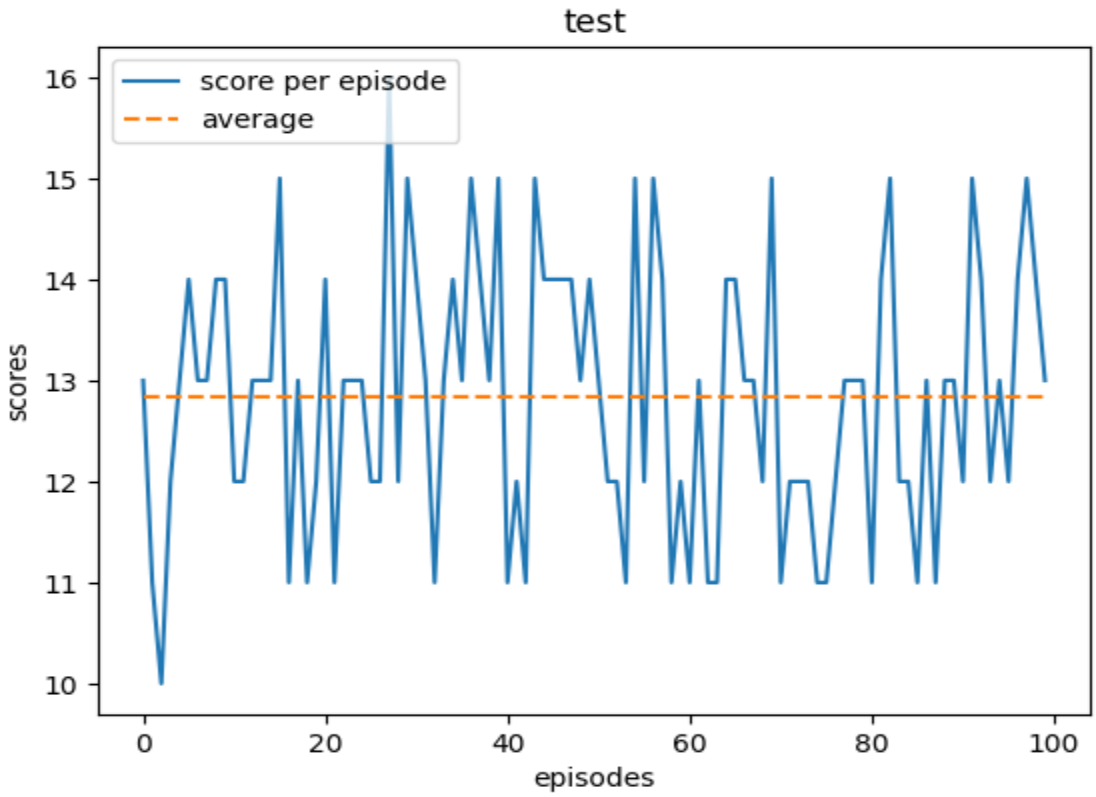

train
score per episode
average
scores
episodes
0
50
100
150
200
250
300
0
100
200
300
400
500

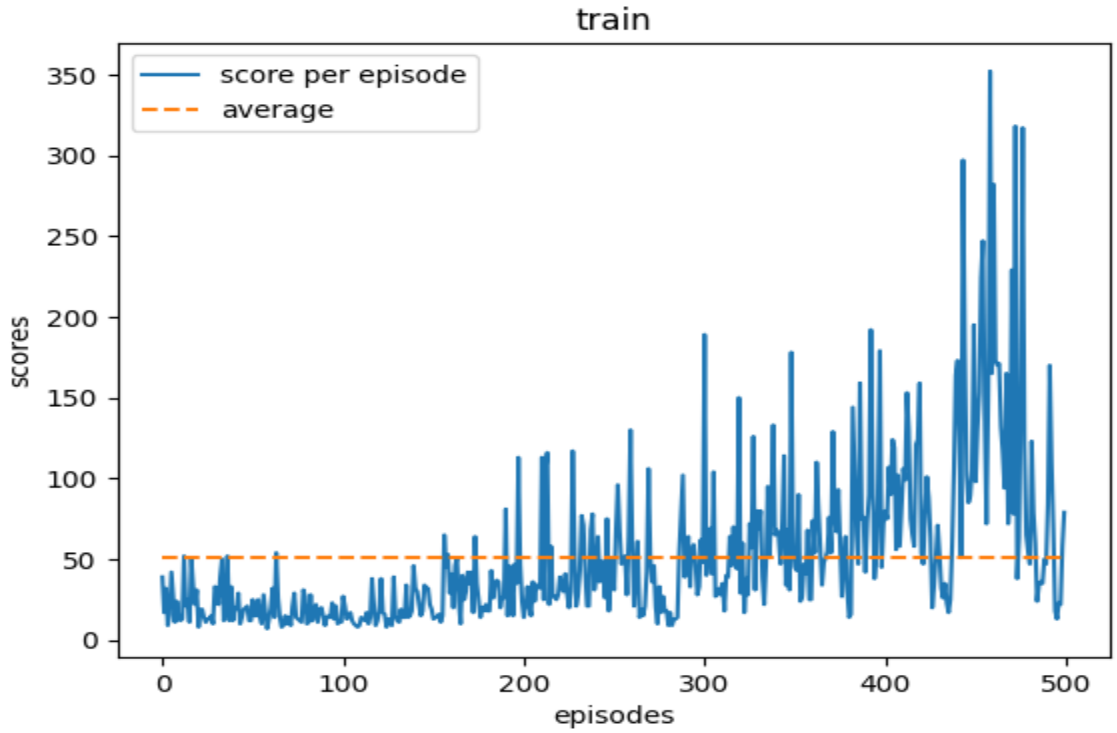

test
score per episode
average
scores
episodes
25
30
35
40
45
50
55
0
20
40
60
80
100

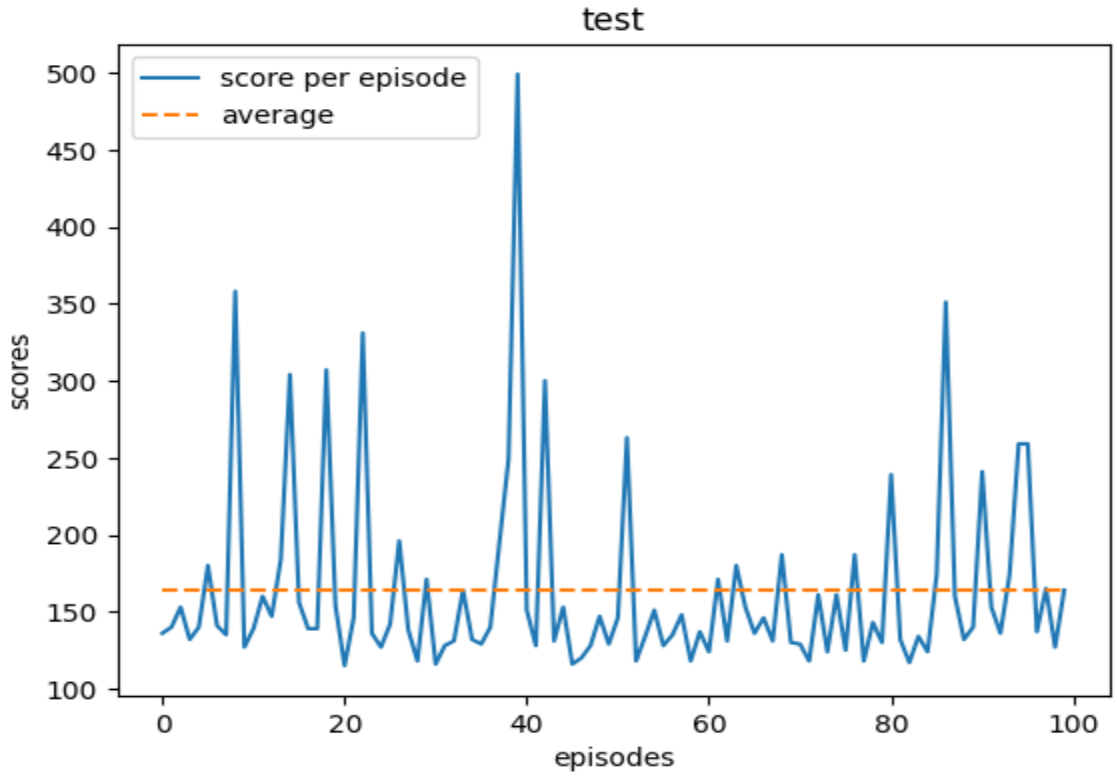

train
score per episode
average
scores
episodes
0
100
200
300
400
500

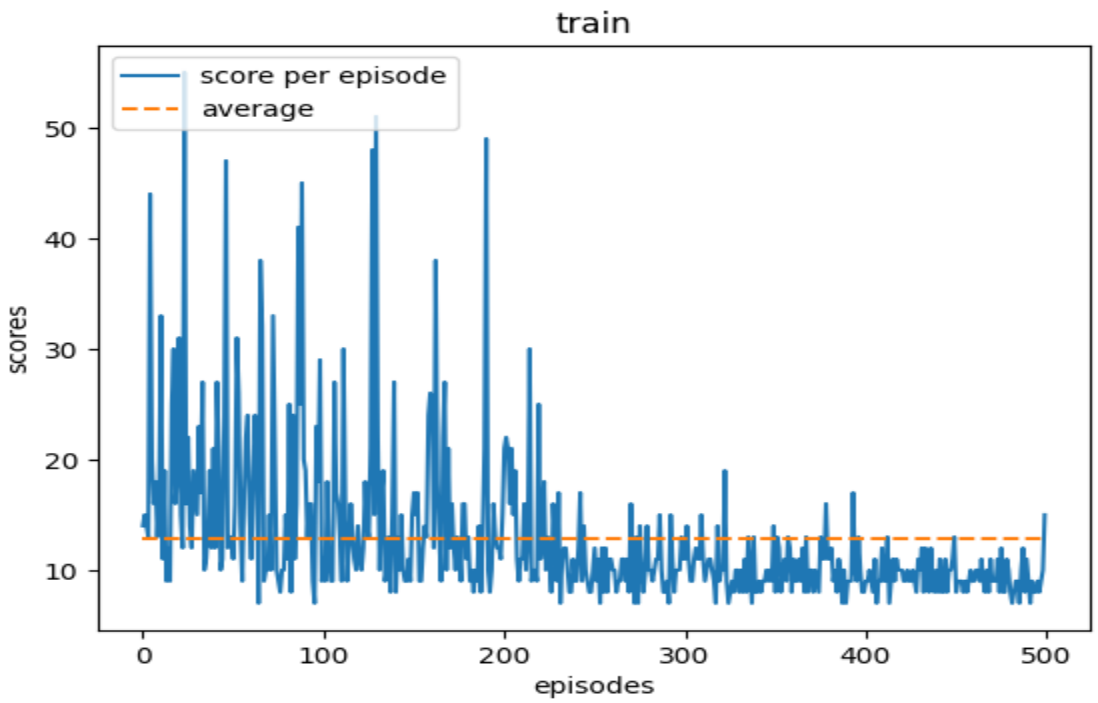

test
score per episode
average
scores
episodes
380
400
420
440
460
480
500
0
20
40
60
80
100

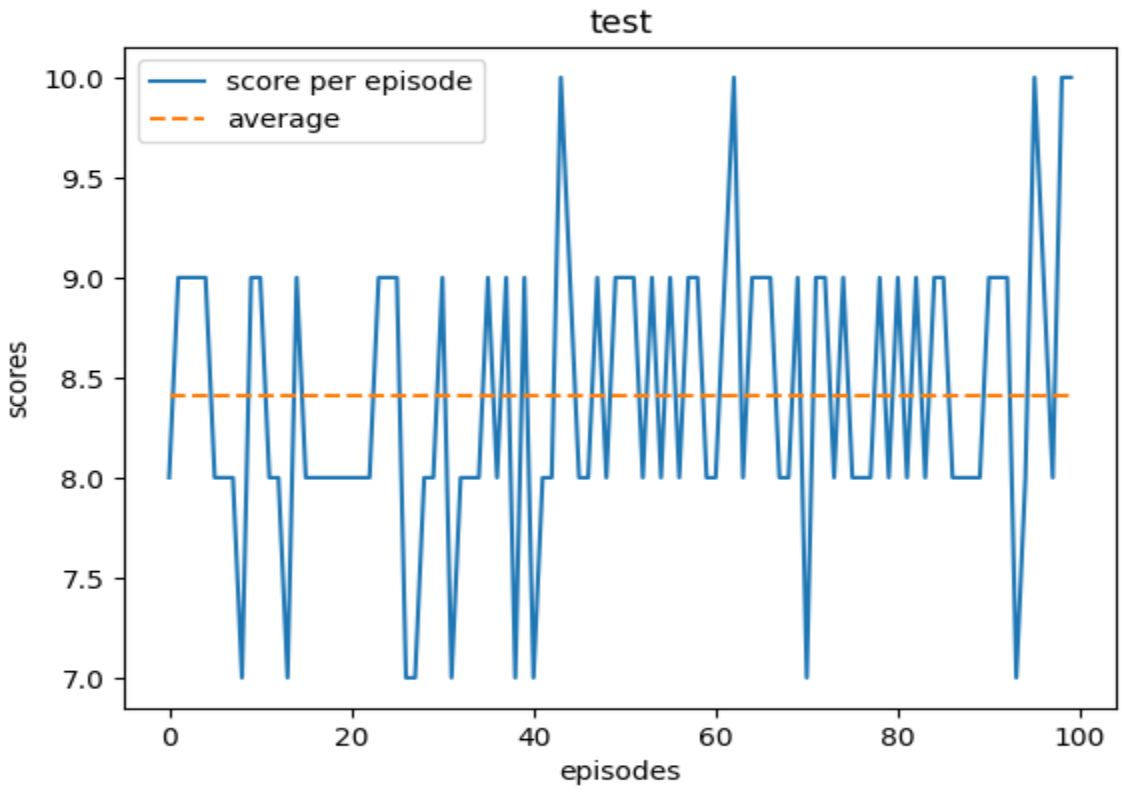

train
score per episode
average
scores
episodes
0
100
200
300
400
500
600

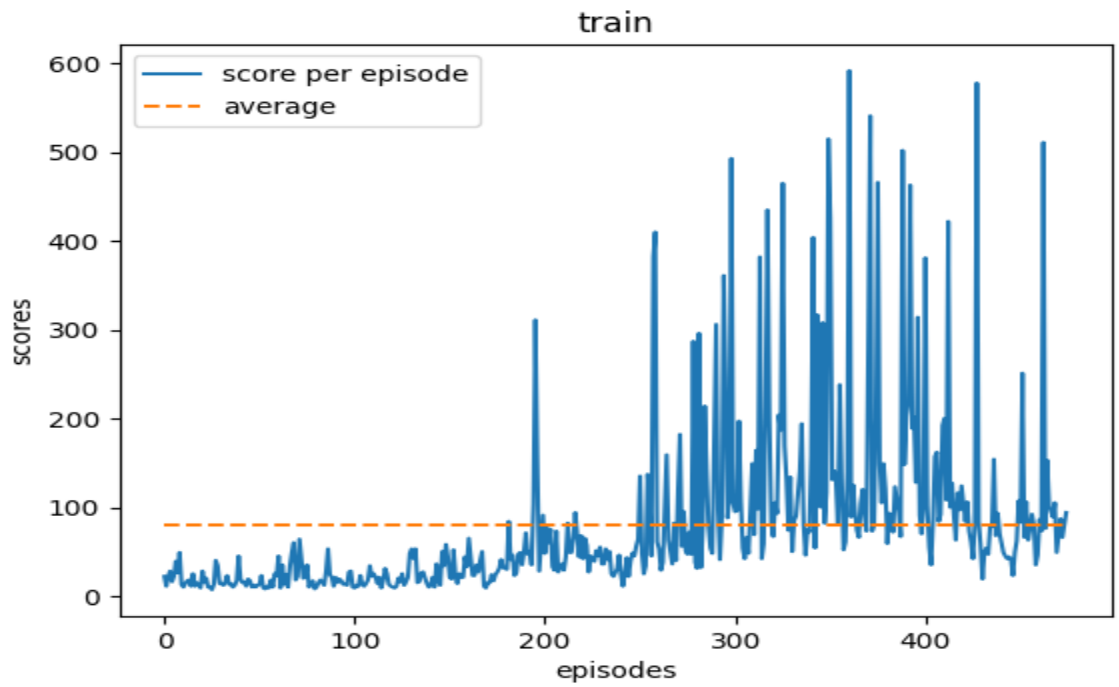

test
score per episode
average
scores
episodes
10
11
12
13
14
15
16
0
20
40
60
80
100

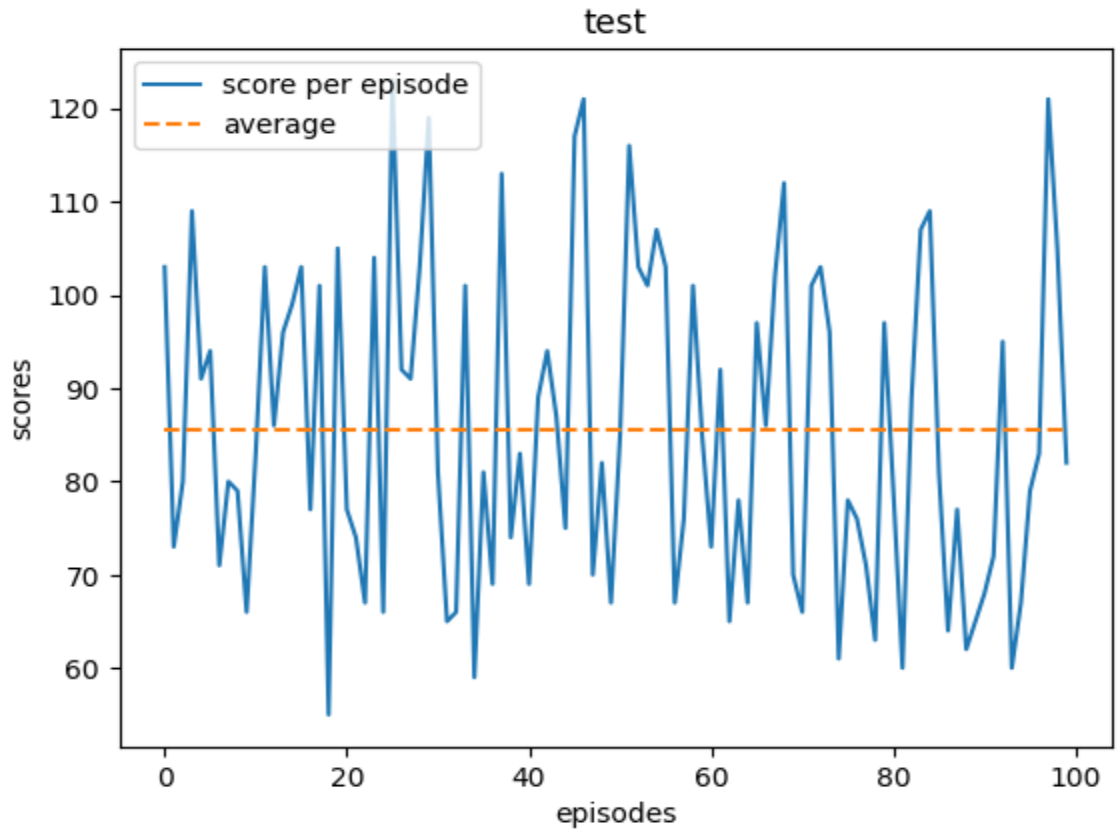

train
score per episode
average
scores
episodes
0
50
100
150
200
250
300
350
0
100
200
300
400
500

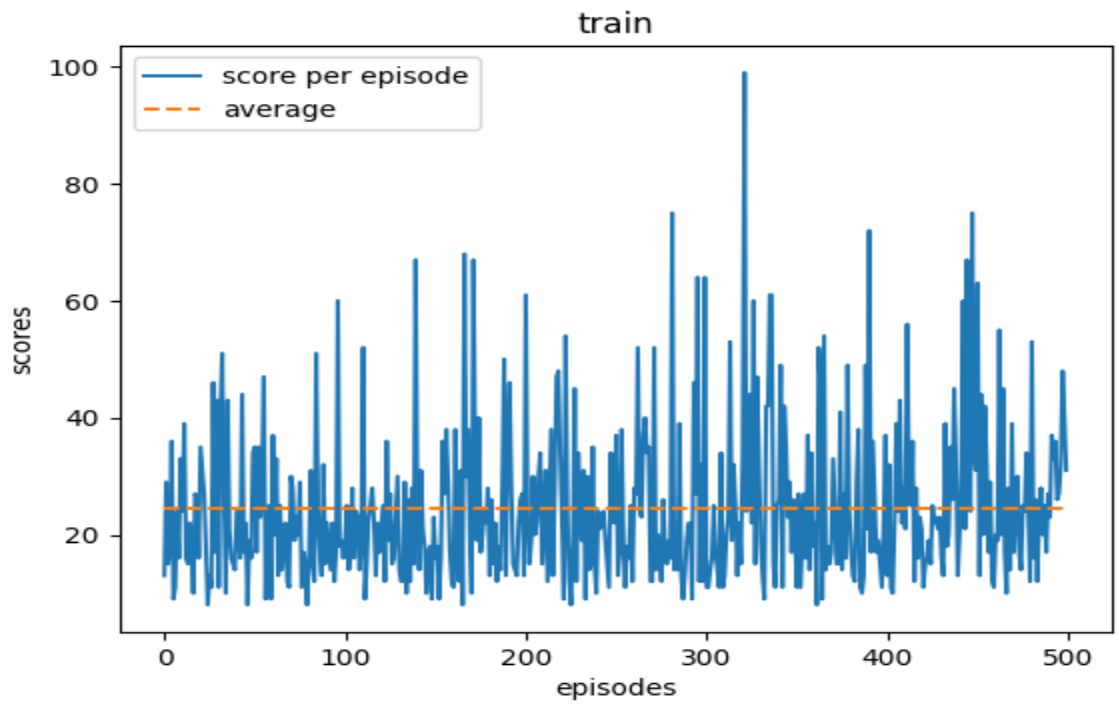

test
score per episode
average
scores
episodes
100
150
200
250
300
350
400
450
500
0
20
40
60
80
100


train
score per episode
average
scores
episodes
0
100
200
300
400
500
10
20
30
40
50

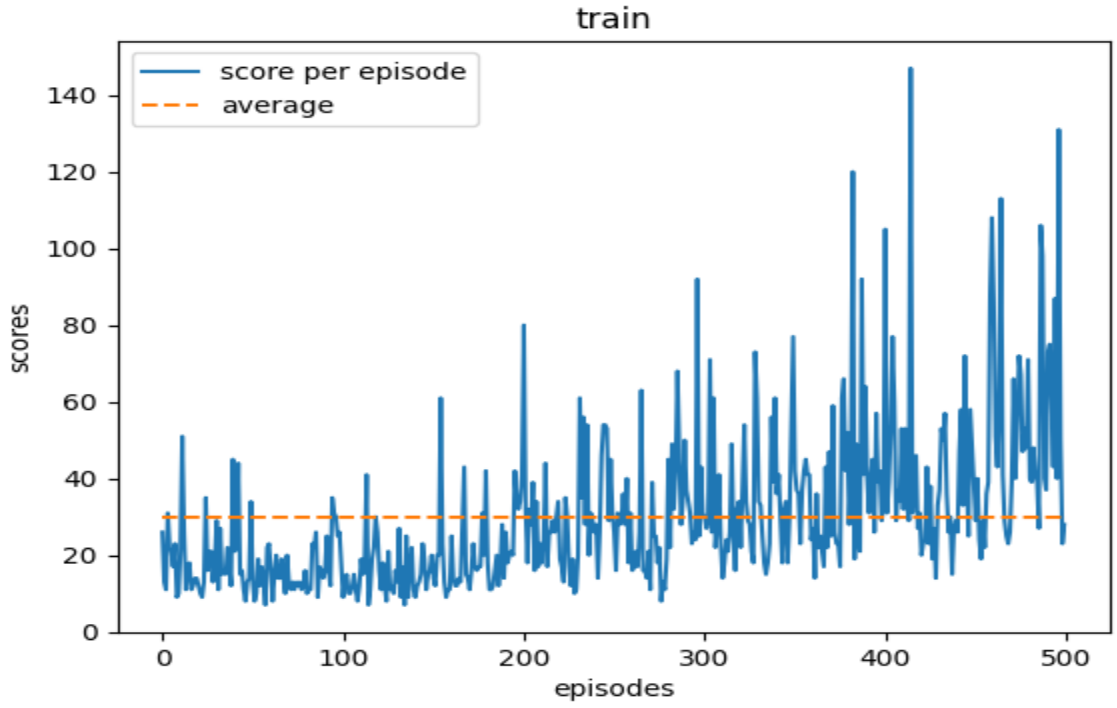

test
score per episode
average
scores
episodes
0
20
40
60
80
100
7.0
7.5
8.0
8.5
9.0
9.5
10.0

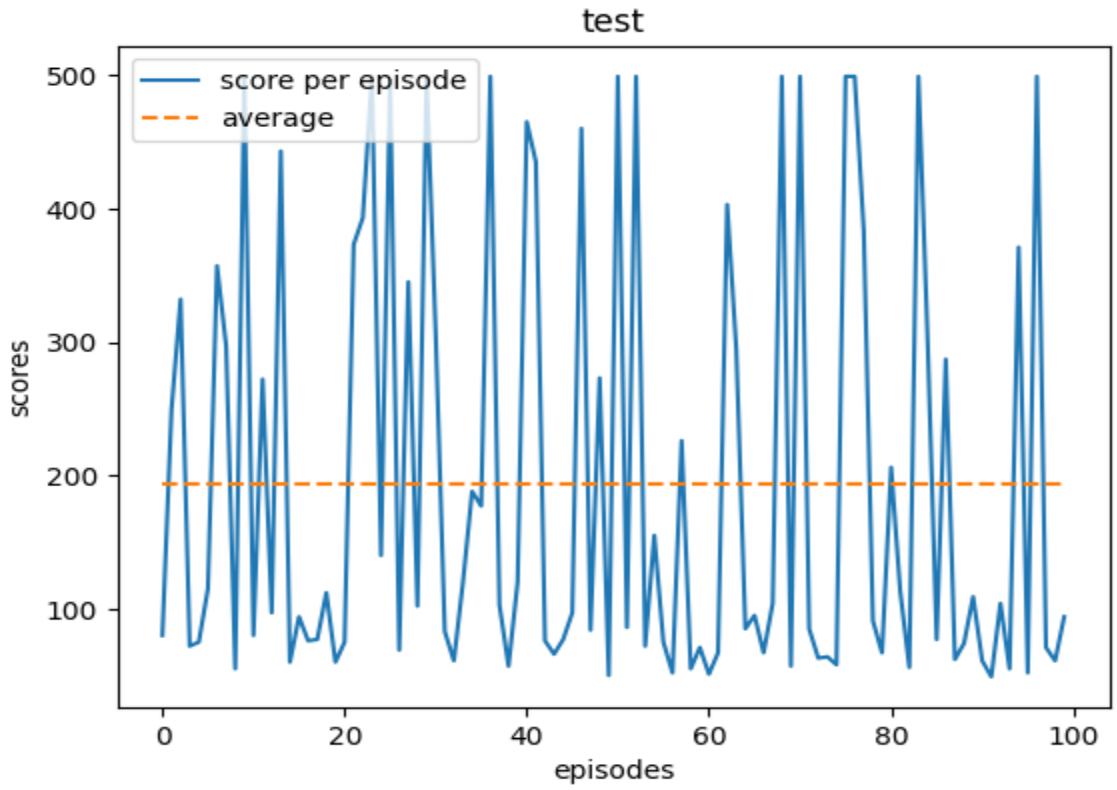

train
score per episode
average
scores
episodes
0
100
200
300
400
500
600

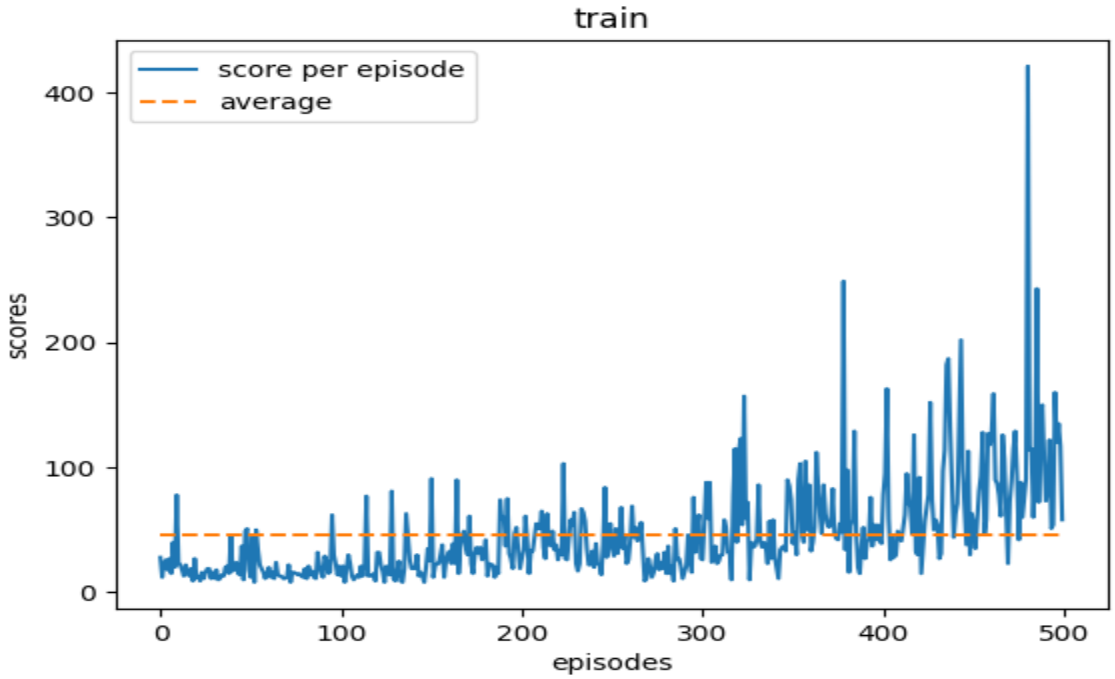

test
score per episode
average
scores
episodes
60
70
80
90
100
110
120
0
20
40
60
80
100

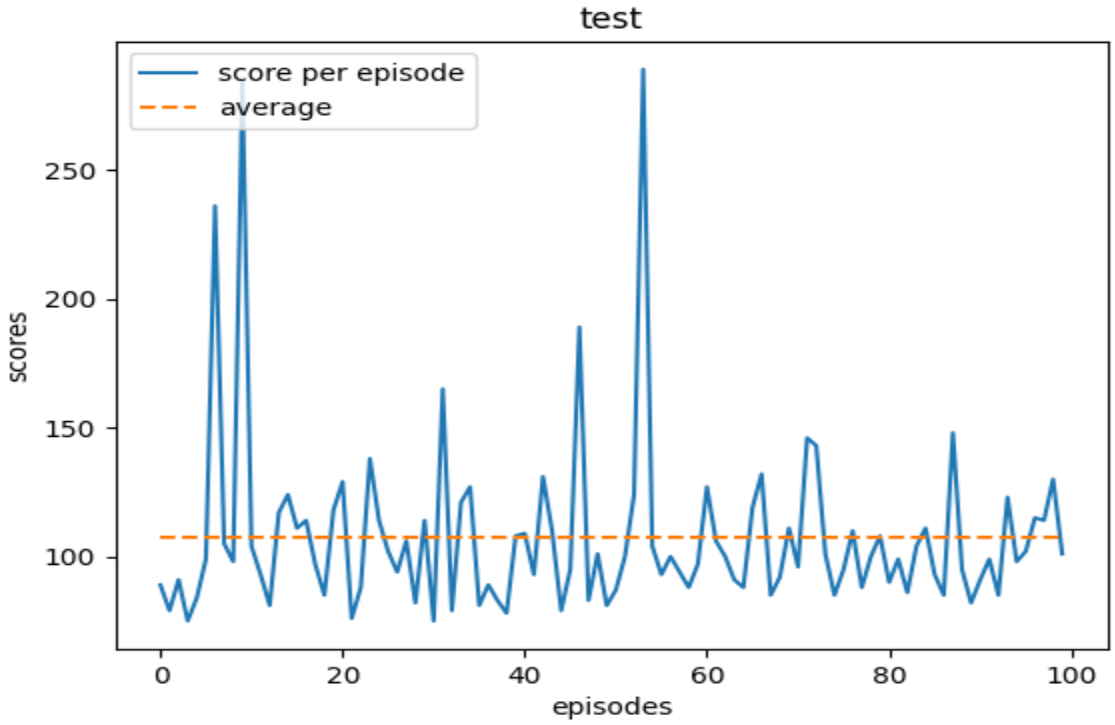

train
score per episode
average
scores
episodes
100
80
60
40
20
0
100
200
300
400
500

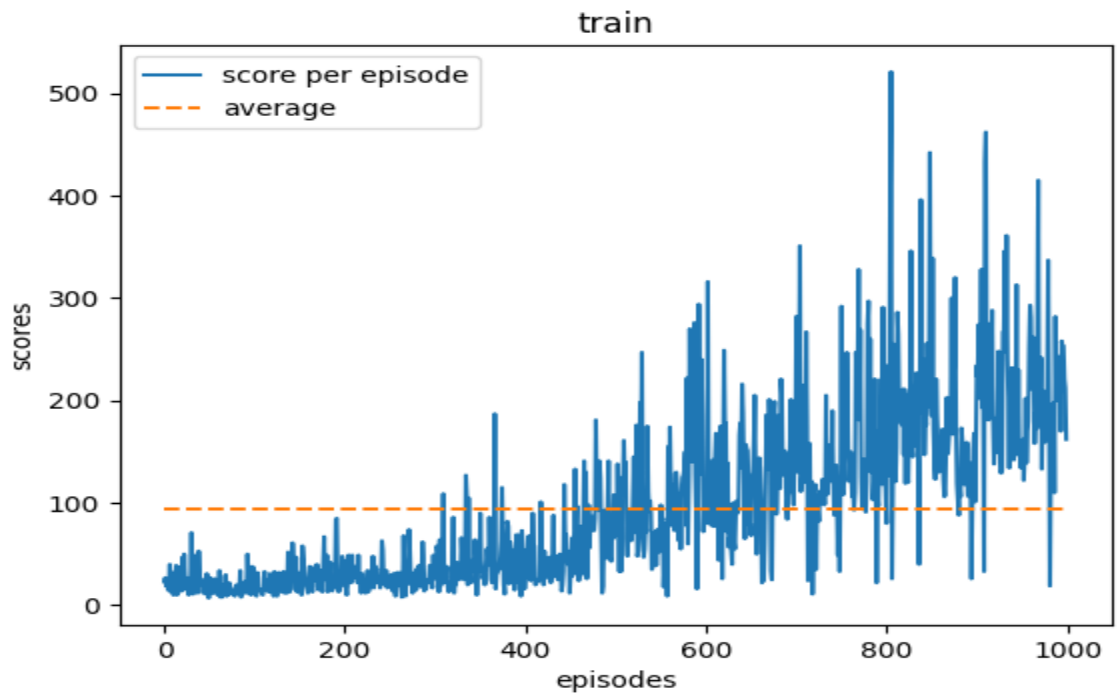

test
score per episode
average
scores
episodes
36
34
32
30
28
26
24
22
20
0
20
40
60
80
100

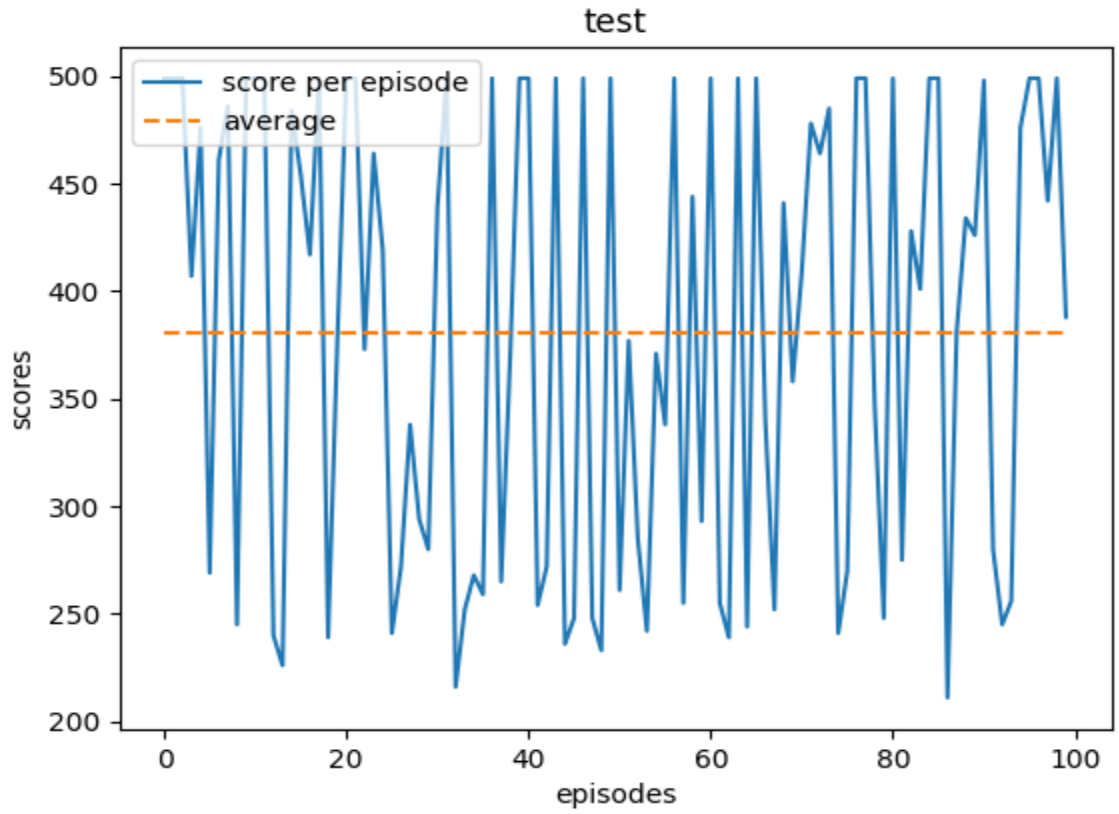

train
score per episode
average
scores
episodes
0
20
40
60
80
100
120
140
0
100
200
300
400
500


test
score per episode
average
scores
episodes
100
200
300
400
500
0
20
40
60
80
100

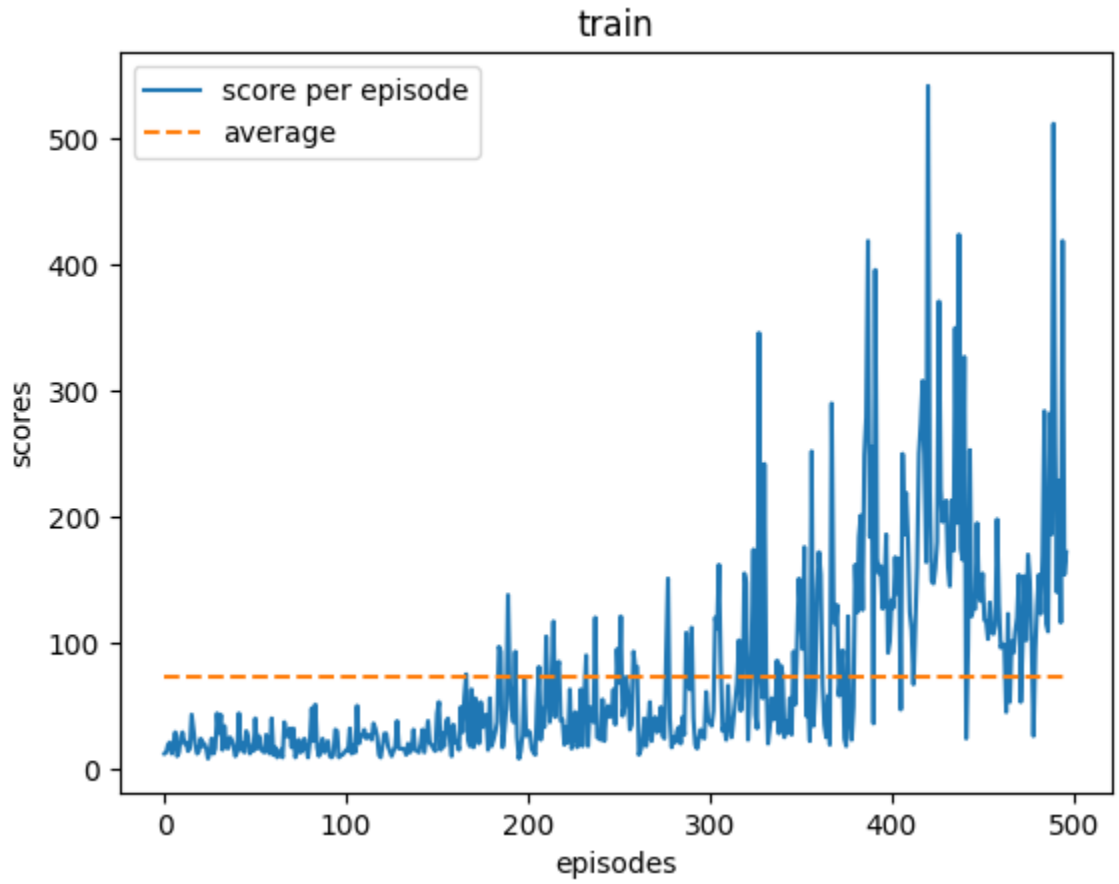

train
score per episode
average
scores
episodes
0
100
200
300
400
500

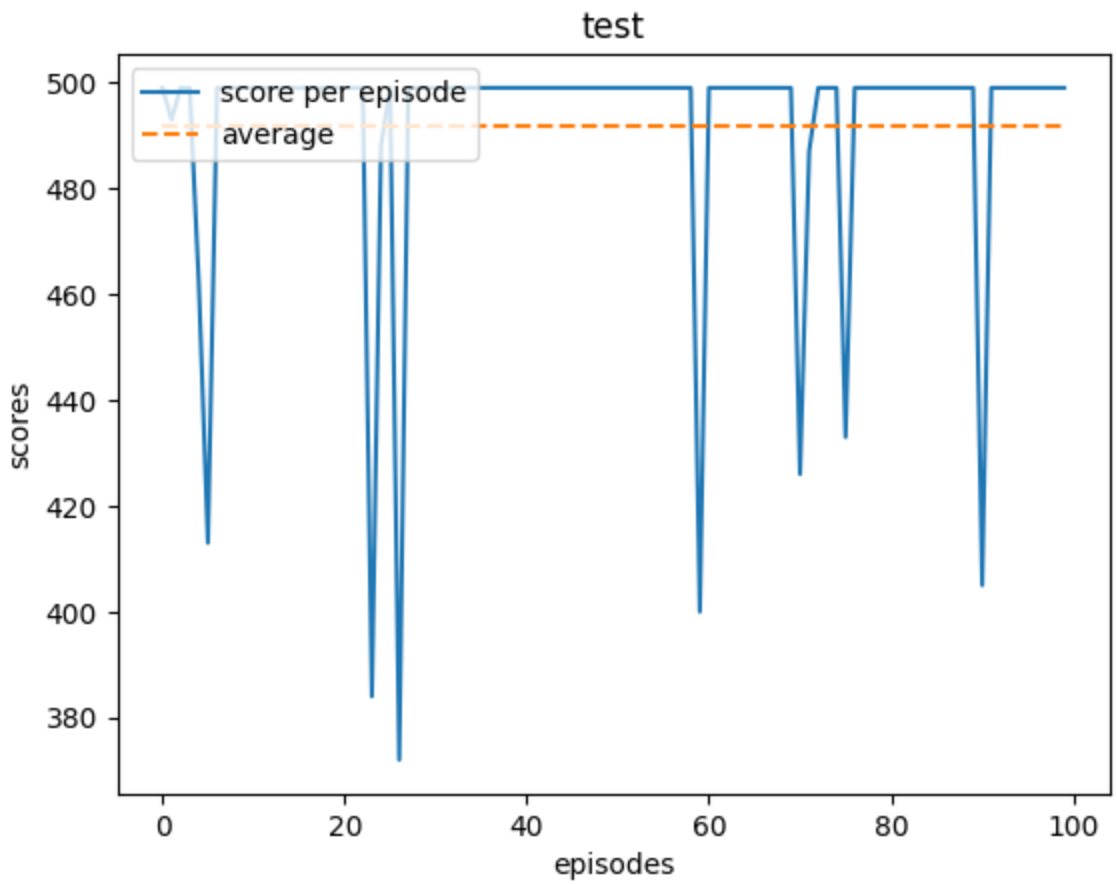

test
score per episode
average
scores
episodes
100
150
200
250
0
20
40
60
80
100

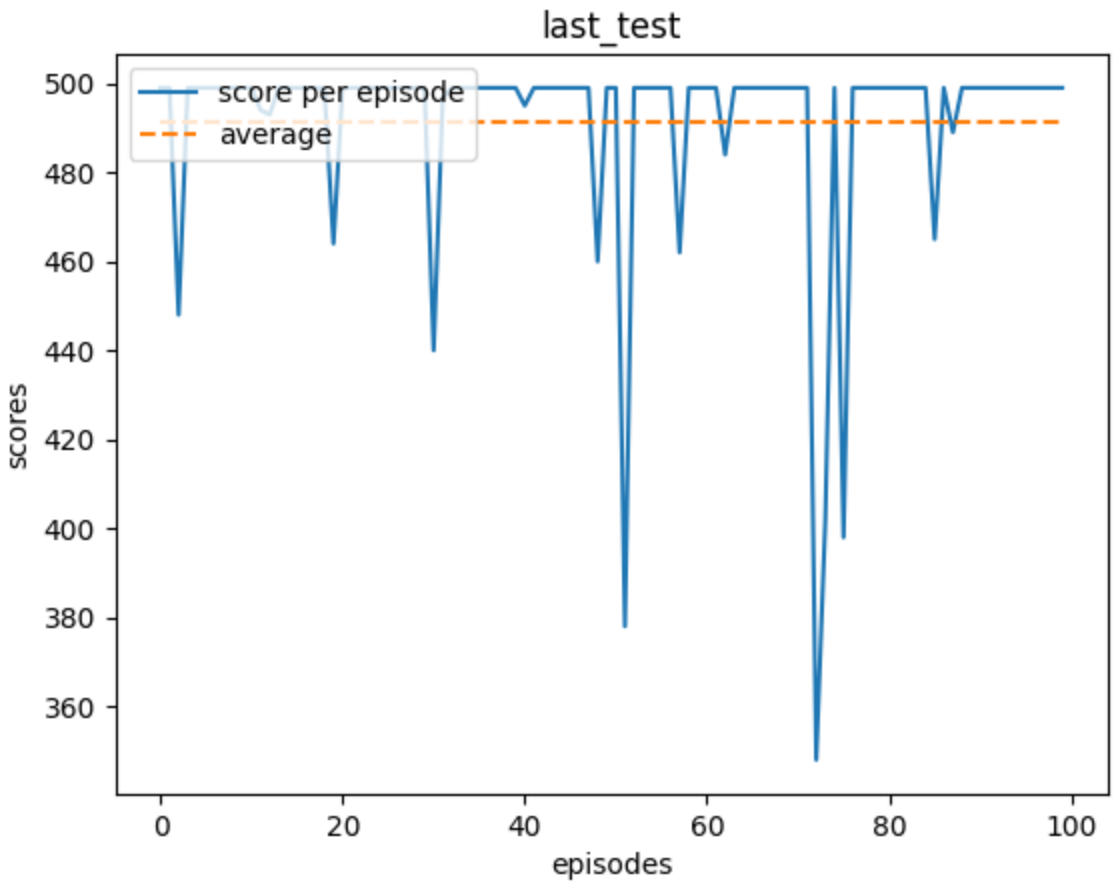

train
score per episode
average
scores
episodes
0
100
200
300
400
500
0
200
400
600
800
1000


test
score per episode
average
scores
episodes
200
250
300
350
400
450
500
0
20
40
60
80
100

We have tried to run again the 11th attempt and we confirmed that it was by far the best configuration among all. In particular, those are the graphs that are self-explanatory:




test
score per episode
average
scores
episodes
500
480
460
440
420
400
380
0
20
40
60
80
100


last_test
score per episode
average
500
480
460
440
420
400
380
360
scores
0
20
40
60
80
100
episodes

CONCLUSIONS

In conclusion, we have successfully implemented a Deep Q-network that managed to learn fast how to balance the pole on the cart. We also noted that we couldn't use the Q-table method since our environment had a continuous state space. If we had used this method, we would have needed a huge amount of memory to store all the state-action pairs in the lookup table. This method is suitable only when the state and action spaces are small and discrete, and the problem is relatively simple. On the other hand, we use a neural network when we deal with large or continuous state and action spaces, and when the problem is more complex.

After all, we have seen how much the choice of the hyper parameters can impact the performance of the network, as well as its architecture. The fine-tuning of those parameters is crucial, but also time consuming. The neural network may suffer from instability during the training, including issues such as overestimation or underestimation of the Q-values.