Machine Learning - Report

Giuseppe Prisco 1895709

Environment description

The environment I chose for developing the project is the Cart Pole from Gymnasium (https://gymnasium.farama.org/environments/classic_control/cart_pole/).

The environment consists of a moving cart with a pole attached with an un-actuated joint to it. The pole is placed upright on top of the cart and the goal is to keep the pendulum balanced by applying forces in the left and right directions on the cart.

Action Space

The action space of the environment consists of the following two actions:

- move the cart to the left
- move the cart to the right

The possible values for these actions are 0 for moving the cart to the left and 1 for moving the cart to the right.
Therefore the action space consists of a discrete space with possible values in {0,1}.

Observation Space

The observation space consists of four dimensions, in the following order:

- cart position, with values in (-4.8, 4.8)
- cart velocity, with values in (-∞, +∞)
- pole angle, with values in (-24°, +24°)
- pole angular velocity, with values in (-∞, +∞)

Reward

The reward assigned to each step is +1 since the goal is to keep the pole balanced as much as possible. The maximum obtainable reward before an episode ends is 500.

Episode Starting State

Upon restarting the environment with env.reset(), all the observation's values are uniformly assigned a value in (-0.05, +0.05).

Episode Conclusion

An episode ends if either the termination conditions are met or the truncation condition is met:

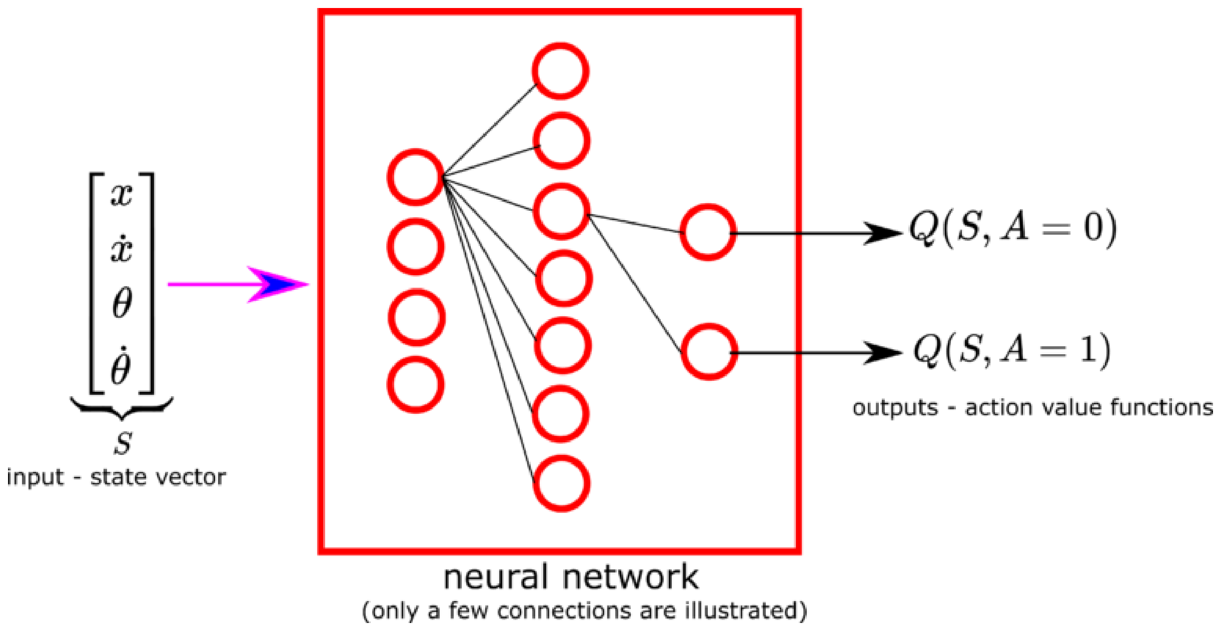
- termination: the pole angle reaches values outside (-12°, +12°)
- termination: the cart position reaches values outside (-2.4, +2.4), meaning that the center of the cart reaches the edge of the display
- truncation: the reward obtained during the episode is 500

Solutions Adopted

In order to solve the environment with Q-learning, we can generally employ two known possibilities: the first is by using a Q-table to store action value functions and the other is to directly approximate the action value function through the use of a neural network.

In the case of the Cart Pole environment, even if it has only 4 dimensions for its observation space, the table storing the Q value functions for different states becomes extremely large and difficult to accurately estimate.
A possible solution is to use a neural network and train it in order to obtain a function to estimate the action value function for a given state.
In our case we would take as input the observation state composed of the 4 values and produce in output 2 values, corresponding to the two possible actions of the environment (action = 0 for pushing the cart left and action = 1 for pushing the cart right).

A simple illustration of this process is shown in the following figure:


$\underbrace{\begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{bmatrix}}_{S}$
input - state vector
$Q(S, A = 0)$
$Q(S, A = 1)$
outputs - action value functions
neural network
(only a few connections are illustrated)

The goal is to train the neural network and use it to predict and greedily select the best action available.
The training of the network is done by the use of an experience buffer, which keeps track of the past experience when executing an action A in the state S. In particular, the experience buffer is composed of tuples with the form (S, A, R, S', Term, Trunc), where S is the current observation state, A is the action executed in that state, R is the reward obtained by executing the action A in state S, S' is the new observation state obtained by executing action A in state S and Term and Trunc are boolean variables denoting if S' reached its terminating or truncating condition respectively.

When training the network we only use a subset of the tuples (called batch) instead of the whole experience buffer by randomly sampling a batch of tuples from the full experience data. The reasoning is that near tuples may be highly correlated between each other, therefore bringing little new significant information and slowing the learning process.

The training rule used for the Q-learning is the following:

- if S' is a terminal state then: $\widehat{Q}(S, A) = R$
- if S' in NOT a terminal state then: $\widehat{Q}(S, A) = R + \gamma \cdot max_{A'}\widehat{Q}(S', A')$

where R is the immediate reward obtained by executing action A in state S and S' is the next observation state. The value $max_{A'}\widehat{Q}(S', A')$ is computed by choosing as input the batch containing the next observation states and producing the predictions for A = 0 and A = 1 and selecting the highest between these two quantities.

The agent generally chooses an action A to be executed based on two strategies:

- Exploitation: the agent selects the action A that maximizes the current approximation for the Q value, that is that maximizes $\widehat{Q}(S, A)$
- Exploration: the agent randomly chooses an action A among all the available actions

The epsilon-greedy approach initializes a variable epsilon (ε) in range [0, 1] and the agent will choose a random action with probability ε and the best action with probability 1 - ε.
Generally we choose an high ε at the start and decrease it over time so that we promote exploration first and exploitation later.
Among the strategies for decreasing ε the most common are a linear decay, an exponential decay or a discrete interval decay.

During the experiments for training the agent, I employed an epsilon-greedy approach by initializing it at ε = 1.0 and trying both the exponential and the linear decay over time.

The full algorithm used to train the Cart Pole agent can be summarized in the following steps:

- Initialization
    - Setup the agent
    - Creation of the neural network
    - Instantiation of the environment
    - Create a log to save relevant data obtained during the execution of episodes
- Loop over the training episodes
    - Reset the environment
    - Loop until the episode ends
        - Pick an action (with ε-greedy approach)
        - Execute the selected action
        - Update the experience buffer
        - Set the current observation as the next observation
    - Update the ε (following either exponential or linear decay)
    - Train the neural network
        - Select a random batch from the experience buffer
        - Create the current and next observation batch from the random batch
        - Predict the target values
        - Compute $\widehat{Q}(S, A)$ (depending on if S’ is a terminal state or not)
        - Fit the neural network with the current batch and the predicted Q values
    - Update the log with obtained data
- Save the model

In order to test the agent after a number of training episodes finishes, we execute the remaining part of the algorithm:

- Instantiate the environment in render mode (used to visualize the agent performing actions in the environment)
- Load the saved model
- Loop over the test episodes
    - Reset the environment
    - Loop until the episode ends
        - Pick the best action
        - Execute the selected action
    - Print the average score obtained from the tests performed so far

The log which records relevant data obtained during the execution of the episodes is in turn used to plot the data and save the graphs as images to compare the performance of the training with different hyperparameters.

Architecture of the Implementation

Numerous tests have been performed in order to train the agent and several models have been derived with different configurations. Among the hyperparameters characterizing a model we have:

- n_episodes: the number of episodes to train the agent
- start_epsilon: the value assigned to ε at the start of the training
- final_epsilon: the minimum value that ε can reach
- epsilon_decay: the decay factor applied to reduce ε over time (exponentially or linearly)
- discount_factor: the discount factor used to attenuate the predicted future rewards
- EXP_MAX_SIZE: the maximum size of the experience buffer
- BATCH_SIZE: the size of the batch used for training the neural network

For the neural network I chose to use two hidden layers with the Rectified Linear Units (ReLU) as the activation functions and a linear output layer.
The loss function used for this environment is the Mean Squared Error (mse), the final algorithm employed for training is Adam (although also RMSprop has been tested) and the final number of hidden units for the first layer is 64 while for the second is 32. Of course, the output layer consists of only two values, the ones corresponding to the predictions performed on the two different actions.

The agent is composed of an ordinary class which simply assigns some agent-related parameters to it.

The storing of performance metrics during the execution of the episodes is done with a csv file which is later opened and inspected to construct the graphs corresponding to the total execution time for each episode and to track the epsilon and cumulative reward over time.

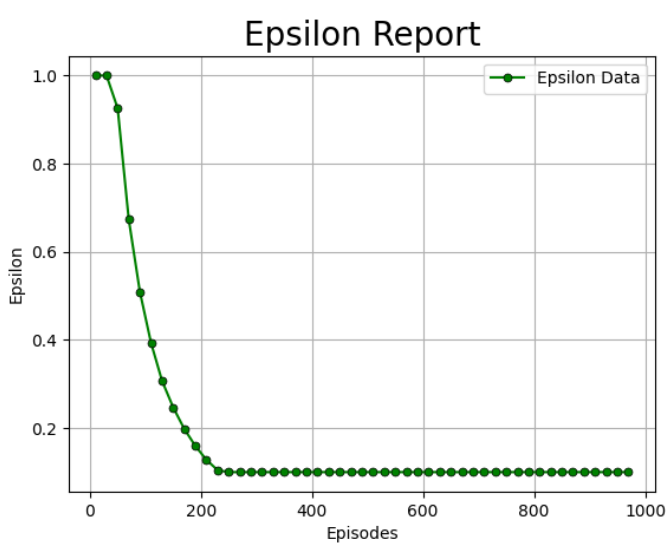
The experience buffer consists of a queue data structure which is populated by appending each time the tuple containing the current state S, the action A executed, the reward R obtained, the next state S' reached and the terminating Term and truncating Trunc conditions. When the length of the experience buffer reaches its maximum length EXP_MAX_SIZE, the oldest entry is replaced by the newest entry, so that we always have the last EXP_MAX_SIZE tuples in the data structure.

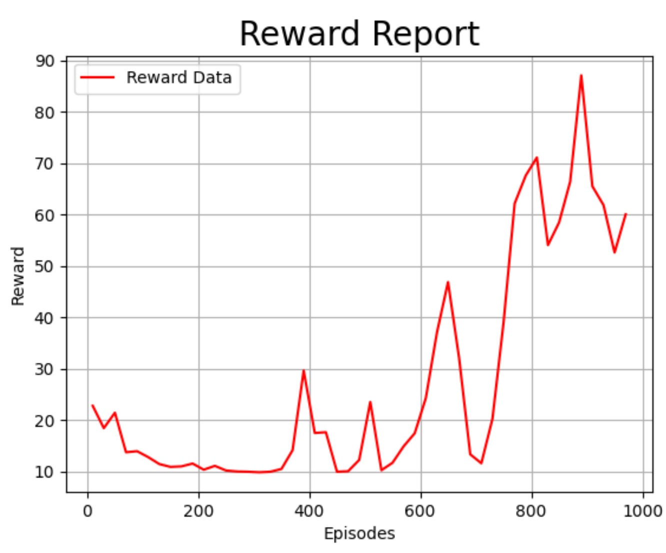
In this final section I summarize some of the results obtained with different configurations and decide on the final hyperparameters chosen.
To make the graphs less noisy, I took the average of the performances every 20 episodes and derived a visually improved graph.

In order to keep the report relatively contained, I only included the graphs obtained during the execution of some of the experiments, which were performed in much higher numbers.

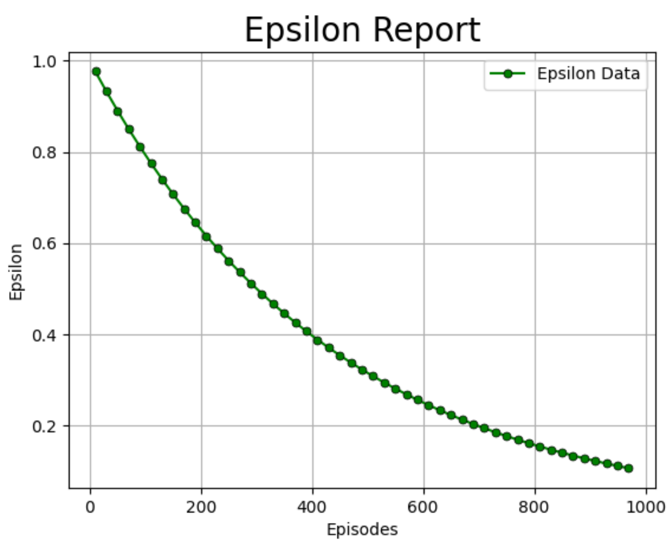
The first set of graphs showed corresponds to the unsuccessful training models derived:

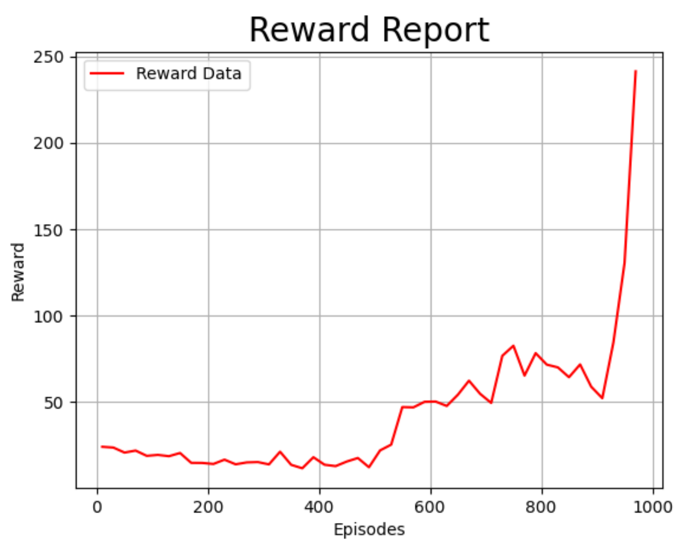
Model 1



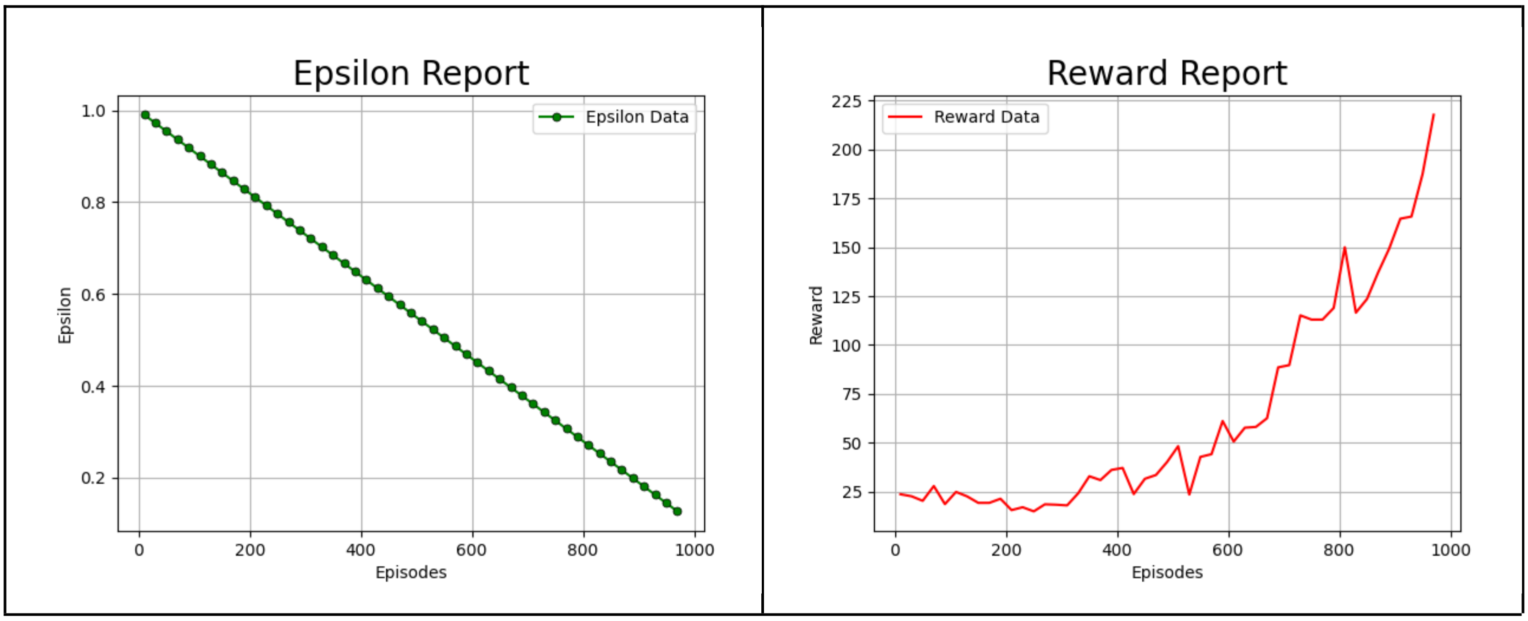


Model 2





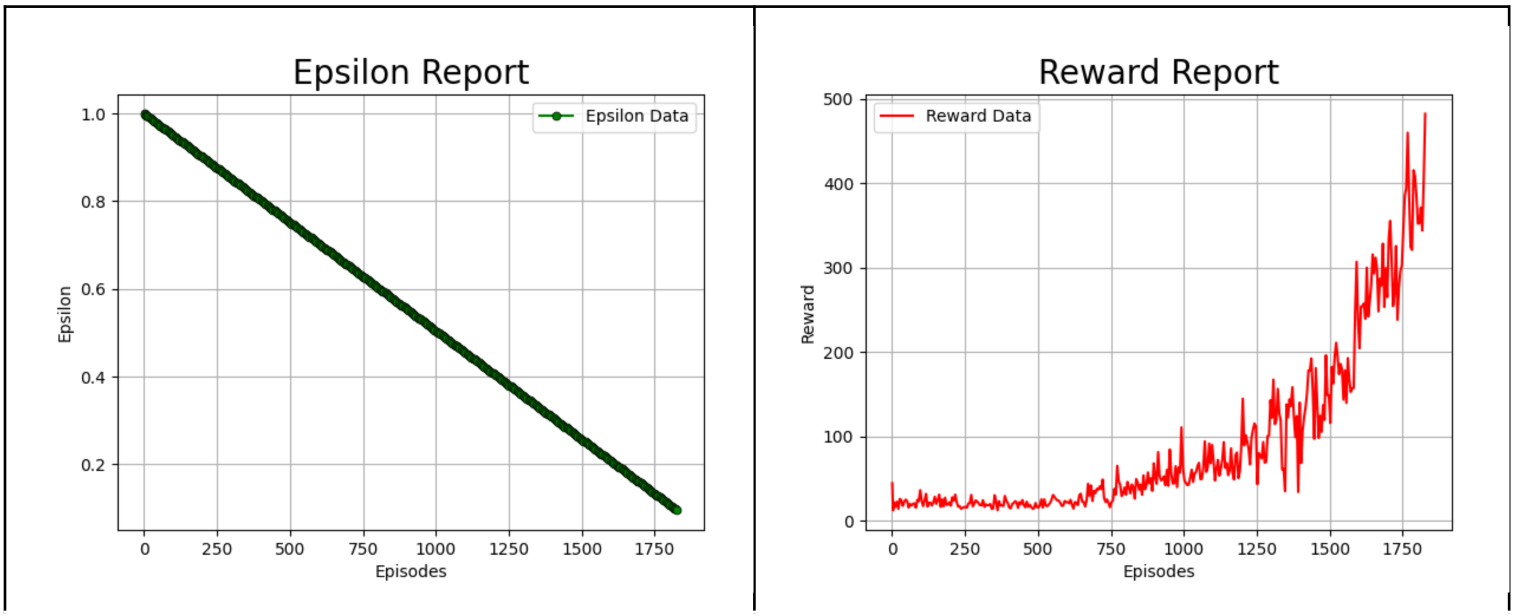
Model 3



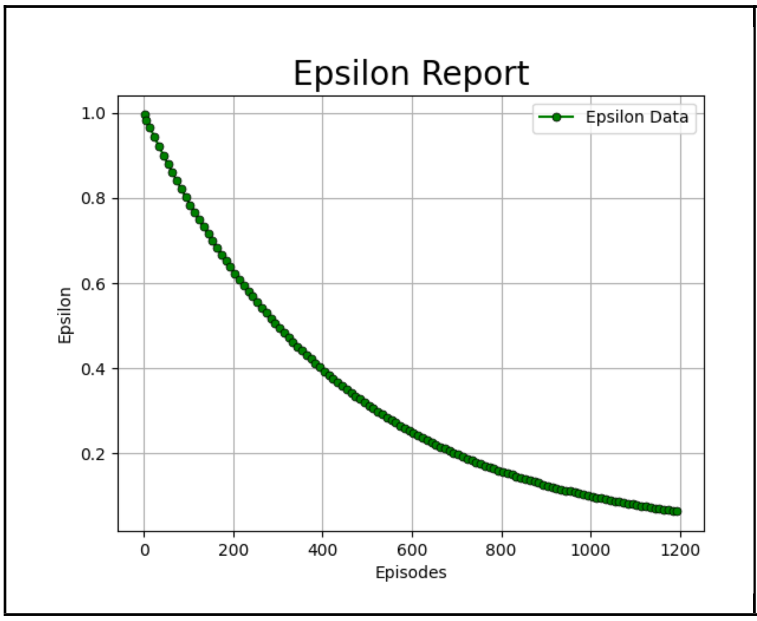
For the previous three models, the number of episodes of training was 1000, which turned out to be too small to obtain successful results and the minimum ε was set to 0.1. In Model 1 the ε decayed at each action taken, rather than at the end of each episode, resulting in a rapid decrease of ε. On the other hand, Model 2 and Model 3 decreased ε at the end of the episode (exponential and linear decayment respectively). Although the reward obtained was much higher than that of Model 1, both 2 and 3 did not collect a cumulative reward higher than 250 (I remind that in order to solve the environment, a score of 500 needs to be achieved).

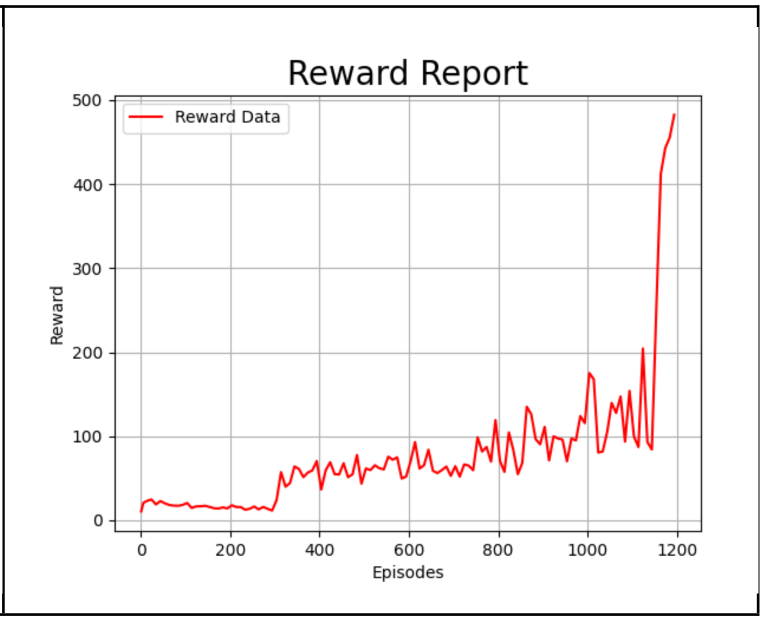
By giving the agent more episodes to train and changing other parameters such as the final minimum epsilon, now set to 0.01, as well as the number of hidden units in the neural network and the maximum size of the experience buffer to 4000, I successfully derived several models which solve the Cart Pole environment:

Model 4

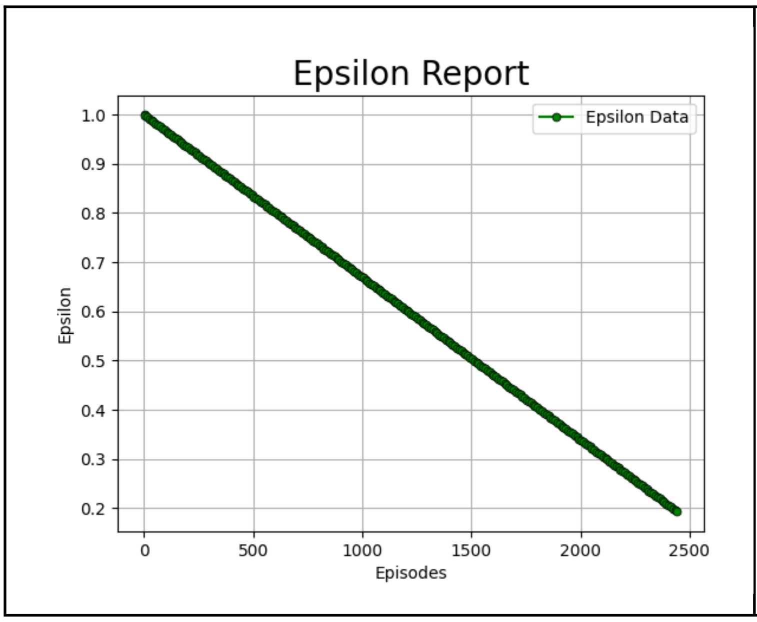


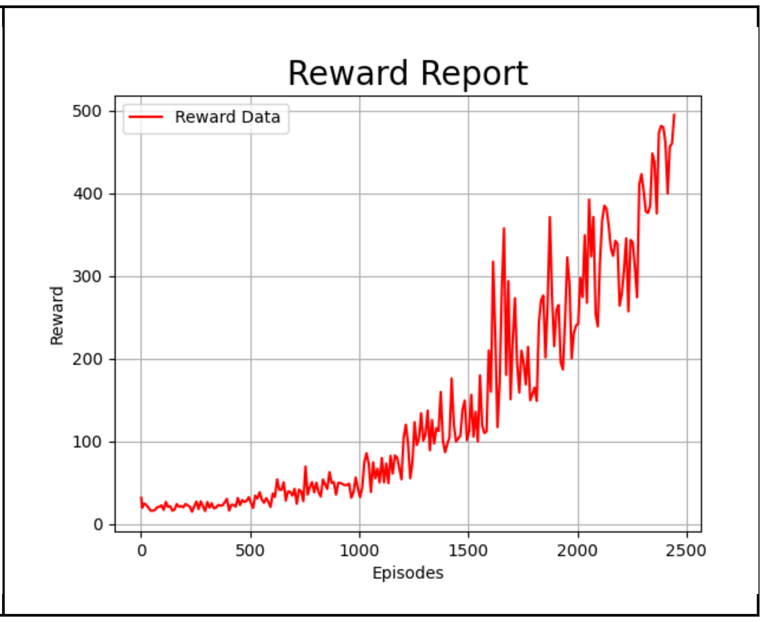
Model 5





Model 6



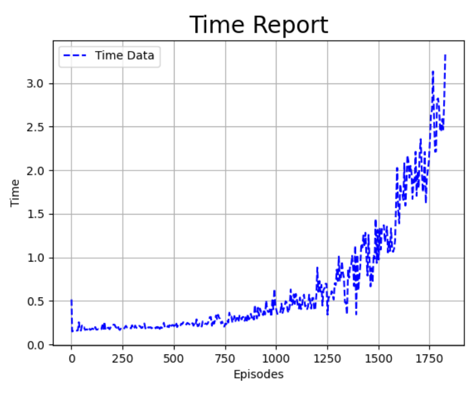


With an increased number of episodes to train on and with the changed parameters, these models were able to successfully pass the tests and thus solved the environment.

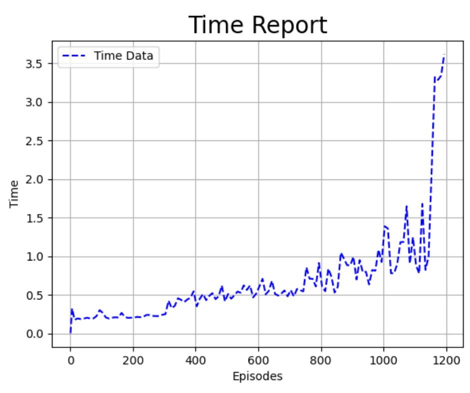
The models derived using a linear decay are much more stable, keeping the pole as straight as possible (that is, with the smallest angle) but the cart tends to follow one of the two directions. On the other hand, the model derived with the exponential decay presents a much more “wobbly” effect, where the pole angles tend to reach higher amplitudes, but the cart stays at the center of the display.

Finally, I also reported the time that the successful models took to complete each episode:

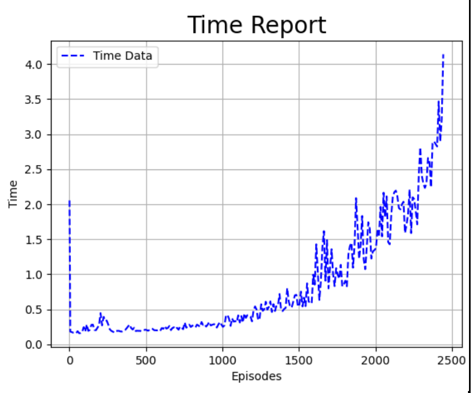
Model 4



Model 5



Model 6



As one can expect, the time to complete an episode depends on the number of actions taken before its end, which in turn reflects the cumulative reward the agent receives.
Therefore the reward and the time graphs are very similar to each other and look pretty much the same.

After many experiments, I decided to stick with the linear decay for the ε as well as adam as the training algorithm, while setting the maximum length for the experience buffer to 4000 and the final ε to 0.01.
These parameters are reflected in the source code accompanying this report.