INM707 - Deep Reinforcement Learning Coursework

Siddharth Madan
Department of Computer Science
School of Science and Technology
City, University of London

Abstract

This report is the final report for the deep reinforcement learning module coursework (INM707). Group tasks 1 to 8 have been accomplished with equal contributions from Siddharth Madan and Kunj Patel. GitHub repository: https://github.com/sidmadan40?tab=projects. This report consists of three parts: The basic part, the advanced part, and eventually the individual part.

Keywords: Q-Learning, SARSA and DQN

1 Basic Task

1.1 Defining the environment and presenting the problem:

In this experiment, we chose a popular game environment designed by OpenAI called "Cliffwalking," which is a standard reinforcement learning problem used to test various algorithms. With start and goal states and the typical actions producing movement up, down, right, and left, this is a typical un-discounted episodic job. With the exception of transitions into the region designated Cliff, the reward is -1. When agents enter this area, they receive an optimal path -100 reward and are immediately sent back to the beginning.

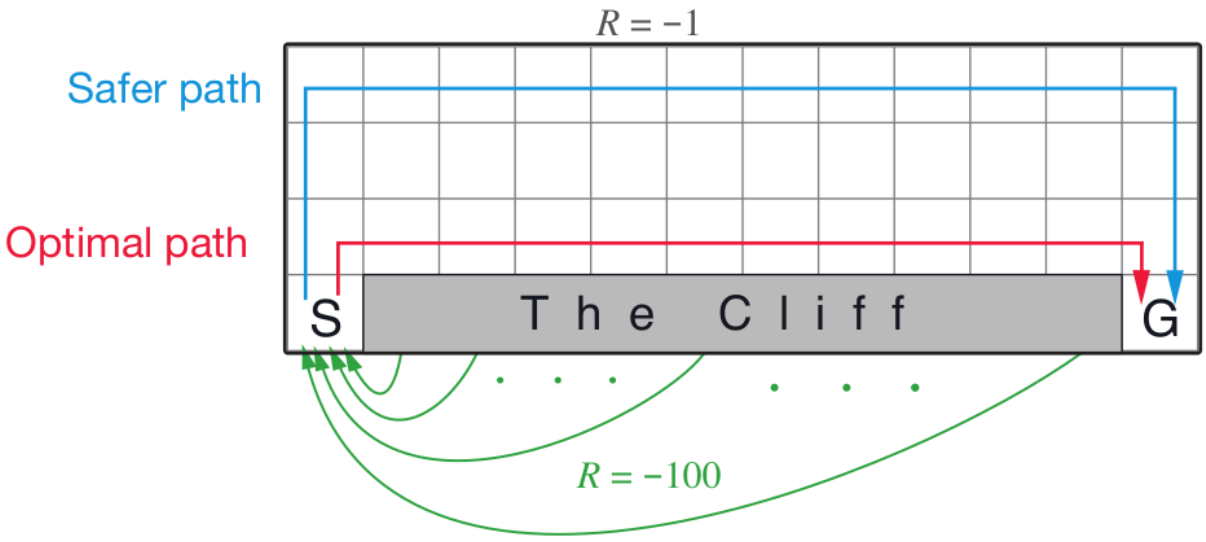


Figure 1: Cliffwalking grid representation [(1)].

As seen in Fig 1. from [(1)] which is the grid representation of our Cliffwalking environment. Here, the optimal path is shown in red, the safest path is represented in blue, and each action is assigned a reward of -1. In this figure, S is the start state and G is the goal state.

1.2 State transition and reward:

The environment consists of a grid of cells, with the agent starting at the bottom left corner (marked ”x”) and the goal at the bottom right corner (marked ”T”). The cells marked ”C” are safe to walk on, while the cells marked ”o” represent empty spaces. However, the row immediately above the safe cells represents a ”cliff,” which is a region that incurs a large negative reward (-100) if the agent falls into it. The agent can take action to move in the four cardinal directions (up, down, left, and right) but cannot move off the grid.

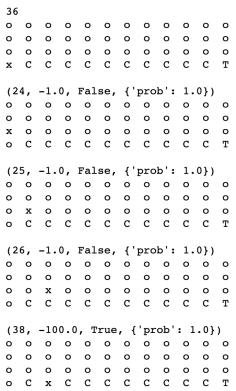


Figure 2: Cliffwalking iterative environment.

The agent begins the episode in the bottom left cell of the initial state and travels 36 steps to reach the goal state. The environment returns a tuple containing data on the environment’s state and the outcome of the preceding action at each step. The reward that the agent received for doing the action in the previous state makes up the second element of the tuple. This reward is -1.0 for all actions except the final one, which achieves the goal and receives a reward of 0.0. If the episode has ended, it is indicated by the third element of the tuple, which is False for all steps but the final one. A dictionary providing more details about the environment, always with the value ”prob’: 1.0” in this scenario, is the fourth element. The goal of the succeeding episodes is the same, but they follow different paths to get there. Each step receives a reward of -1.0 once again, with the exception of the final step, which succeeds and receives a reward of 0.0. The agent takes a course that is less than ideal, which results in a significant negative reward of -100.0 and ends the episode. This makes the last episode fascinating.

1.3 Set Q-Learning with parameters: gamma, alpha and policy

A common reinforcement learning algorithm used to learn the best action-value function for a particular environment is called Q-Learning. The action-value function, also referred to as the Q-function, converts a state-action pair into a value that symbolises the anticipated overall gain from taking the specified action starting from the specified state [(2)].

By iteratively updating its estimations of the Q-values in response to the observed rewards and transitions in the environment, the agent uses Q-Learning to try to learn the best Q-function. The following update rule is used by the agent to update the Q-value for a given state-action pair (s, a):

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right)$$

Where, $Q(s,a)$ is the Q-value for state s and action a, α is the learning rate, which controls the extent to which the agent updates its estimates based on new information, r is the reward obtained by taking action a from state s and transitioning to state s', γ is the discount factor, which controls the importance of future rewards relative to immediate rewards, and $\max_{a'} Q(s',a')$ is the maximum Q-value over all possible actions a' in the next state s' [(2)].

The gamma parameter controls the agent’s preference for long-term rewards over short-term ones. A gamma value of 0 indicates that the agent solely values immediate benefits, whereas a gamma value of 1 indicates that the agent values both current and future rewards equally. The step size of the updates the agent makes is determined by the alpha parameter. When alpha is relatively low, the agent updates its estimates gradually, whereas when alpha is large, the agent updates its estimates quickly. The following table shows the list of hyperparameter values used in our code and the best combination of hyperparameters is mentioned in the next sub-section:

Parameter	Iterated values
Discount factor	0.9,0.95,0.99
Alpha	0.1,0.5,0.9
epsilon	0.1,0.5,0.9

In this coursework, we have considered epsilon-greedy as our choice for policy for the initial exploration of the environment. During agent training, the epsilon-greedy policy is a straightforward but efficient way of achieving an appropriate balance between exploitation and exploration. In essence, the agent chooses, with probability 1 - epsilon, the action that maximises the estimated value (for example, projected reward) of each state. Probability epsilon, on the other hand, forces the agent to choose a random action, allowing it to explore the environment and learn more about the rewards that might be linked with various actions in various states. During training, the value of epsilon is often steadily reduced so that the agent eventually gets greedier and less reliant on exploration [(3)].

1.4 Running the Q-Learning algorithm and discussion on its performance:

In order to determine the best combination of hyperparameters we then performed a grid search on all the combinations of parameters. The plots for all the various sets of combinations can be seen in Fig 3. This figure shows us the plots of the number of episodes on the x-axis and the rewards obtained on the y-axis. In almost all the plots within Fig 3, we can observe the line in red attains the most optimal reward-to-episode ratio which represents the parameter epsilon value equal to 0.1.

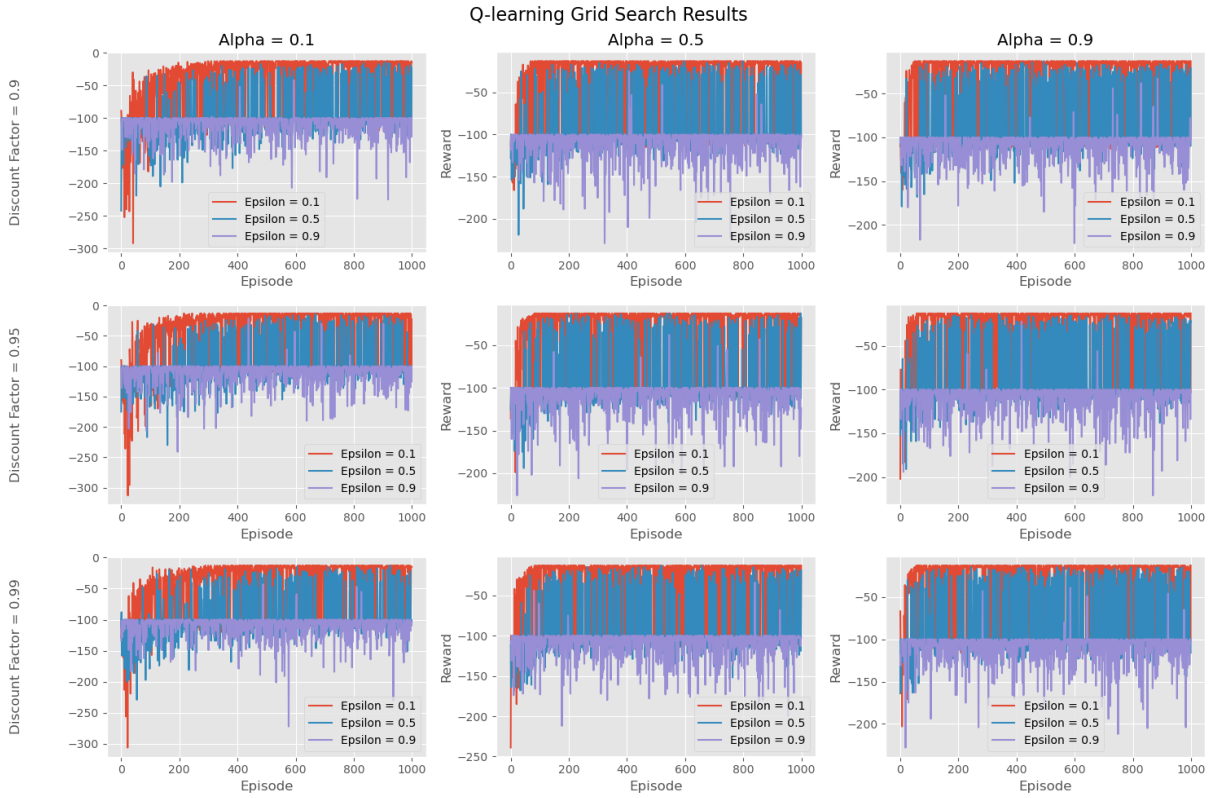


Figure 3: Grid Search for all the parameters in Q-Learning.

We then run the code to provide us with the best combination of hyperparameters among all the possible combinations. The best set of parameter values are discount factor (gamma): 0.9, alpha: 0.9, and as mentioned above epsilon: 0.1. This low value of epsilon suggests that the agent gets greedier much sooner and is much less reliant on exploration. In the next few lines of code we plot the performance of the Q-Learning algorithm with the combination of the best hyperparameters. In Fig 4, we can see the magnitude of episode length decrease drastically beyond 50 episodes approximately and the time taken to run an episode reduces as the agent gets a hang of the environment. Similarly, in Fig 5 we can see a steep rise in the rewards accumulated by the agent per episode. In Fig 6, we can see that the episodes were taking a longer time initially as the slope of the curve until 1500 time steps is less, and beyond 1500 the slope is much steeper and eventually turns into a straight line with a linear relationship with time steps. In our code we also determine the Average reward per episode for best hyperparameters to be -37.86 in the case of Q-Learning algorithm.

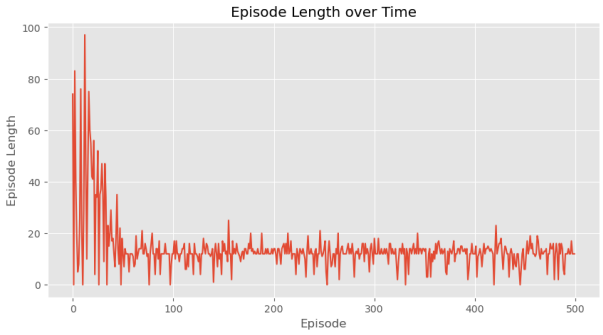


Figure 4: Episode Length over Time in Q-Learning.

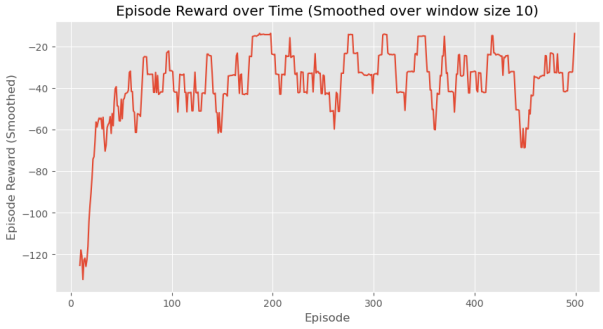


Figure 5: Episode Reward over Time smoothed over window size 10 for Q-Learning

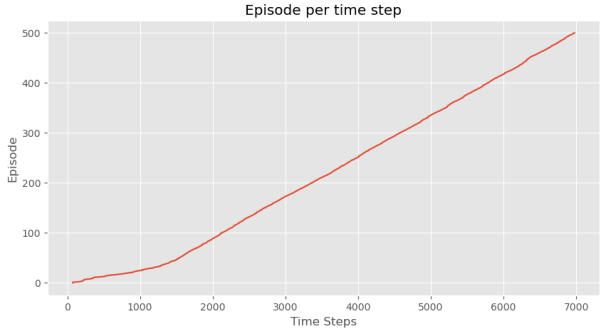


Figure 6: Episodes per time step for Q-Learning.

1.5 Reapeating the experiment with SARSA implementation:

The SARSA algorithm learns a policy by updating its estimates of the Q-values of state-action combinations. SARSA is a model-free, on-policy reinforcement learning technique. State-Action-Reward-State-Action, or SARSA as it is known, is the order in which the algorithm's events take place. The agent observes the current state at each time step, performs an action in accordance with its policy, earns a reward, observes the new state, and performs another action in accordance with the same policy. The new reward and the anticipated future reward—both determined using the same policy—are incorporated into SARSA's Q-values, which are then updated [(4)]. The SARSA algorithm utilises the following equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot (r + \gamma \cdot Q(s',a') - Q(s,a)) \tag{1}$$

where: $Q(s,a)$ is the Q-value for state s and action a, α is the learning rate, which controls the extent to which the agent updates its estimates based on new information, r is the reward obtained by taking action a from state s and transitioning to state s', γ is the discount factor, which controls the importance of future rewards relative to immediate rewards, $Q(s',a')$ is the Q-value for the next state, and s' and the next action a' chosen by the agent according to its policy (which could be an epsilon-greedy policy) [(5)].

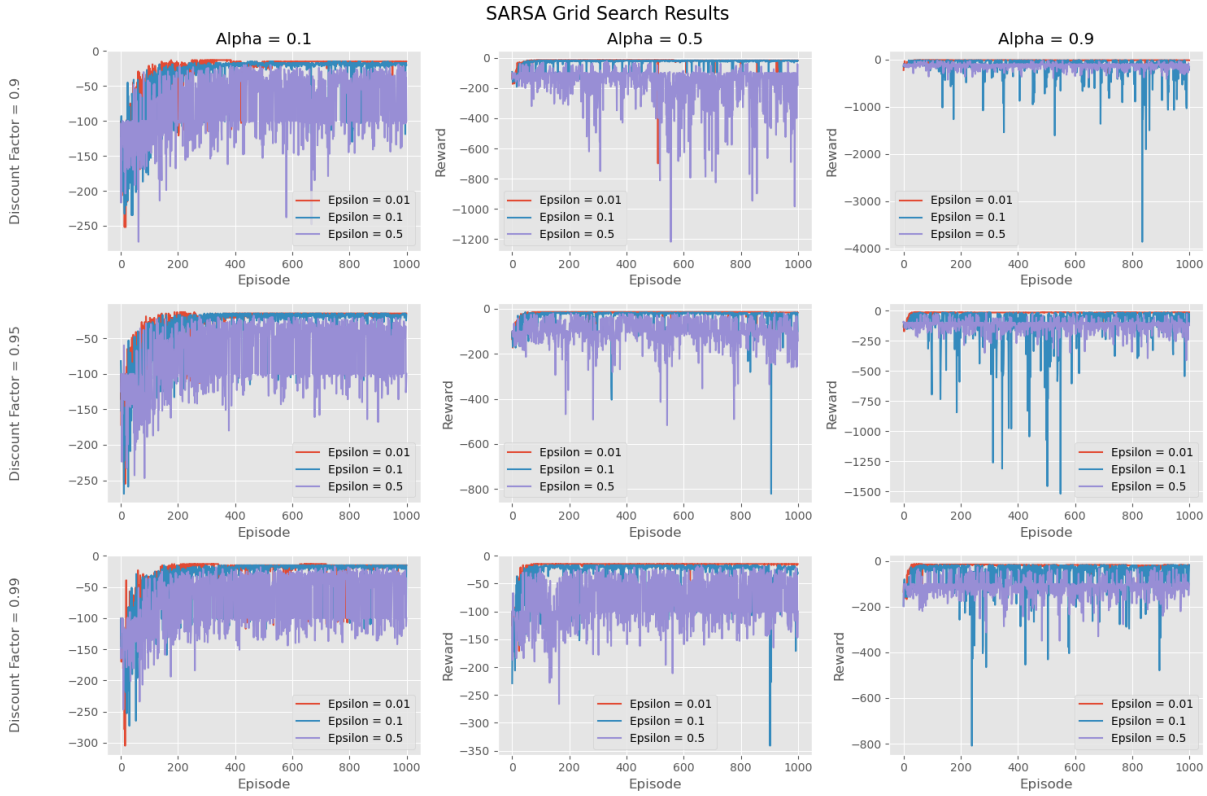


Figure 7: Grid Search for all the parameters in SARSA.

We then conducted a grid search on all the parameter combinations to identify the ideal combination of hyperparameters. Fig. 7 displays the graphs for each of the several sets of combinations. The plots of the number of episodes are shown on the x-axis, and the rewards that were earned are shown on the y-axis in this graph. Almost all of the plots in Fig. 7 show that the red line, which represents the parameter epsilon with a value of 0,01, achieves the best reward-to-episode ratio.

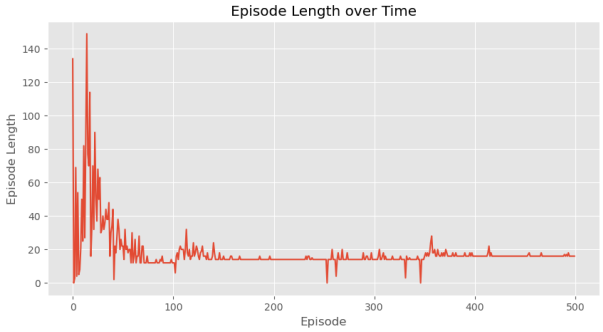


Figure 8: Episode Length over Time in SARSA.

The code is then executed to give us the greatest possible combination of hyperparameters out of all those that might be used. Discount factor (gamma) = 0.99, alpha = 0.5, and epsilon = 0.01 are the ideal values for the parameters. This low value of epsilon implies that the agent becomes less dependent on exploration and becomes more greedy much sooner. We plot the SARSA algorithm's performance using the ideal hyperparameter combination in the following few lines of code.

As the agent becomes more familiar with the environment, we can see in Fig. 8 that the magnitude of episode duration rapidly decreases beyond 40 episodes, roughly, and that the time required to execute one episode decreases. Similar to Fig. 8, we can witness a sharp increase in the agent's incentives per episode. According to Fig. 10, the episodes initially took longer to complete since the slope of the curve remained smaller until 1500 time steps, after which it became more steeper and eventually became a straight line with a linear relationship to steps. Additionally, we calculate in our algorithm that the average reward for each episode for the optimal hyperparameters

to be -18.587 in the case of the SARSA algorithm.

Figure 9: Episode Reward over Time smoothed over window size 10 for SARSA

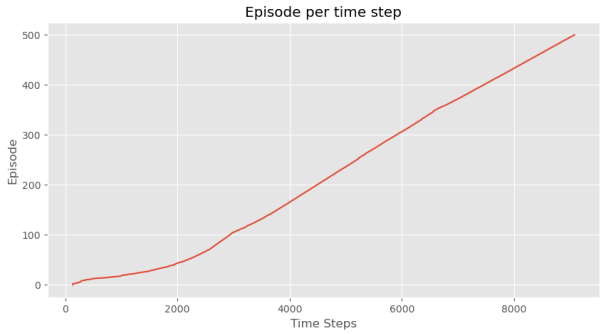


Figure 10: Episodes per time step for SARSA.

1.6 Quantitative and Qualitative Analysis:

In SARSA, the agent implements the same action-choice strategy while updating its Q-values to choose the best course of action. In other words, the agent considers its exploration strategy and estimates the significance of the current policy. In contrast, Q-learning uses the greedy policy in order to update the Q-values in relation to the estimated Q-values. In other words, even if the agent does not adhere to this policy during learning, Q-learning presupposes that the best policy is the one that chooses actions that maximise the Q-value of the subsequent state [(2)].

In all the above plots in both the cases of Q-Learning and SARSA algorithms, it is quite evident that the performance of the SARSA algorithm is significantly better than that of the Q-Learning algorithm. In the grid plots themselves, it can be observed that the rewards per episode are scattered all over every individual subplot for the Q-Learning algorithm, while on the contrary, all the subplots in the grid search Fig 7 for SARSA algorithm have an upward trend suggesting that the latter is performing better. Moreover, as mentioned in the previous sections the average rewards with the best combination of hyperparameters in both cases are -37.86 for the case of the Q-Learning algorithm and -18.587 for the case of the SARSA algorithm, suggesting that the SARSA algorithm has accumulated greater rewards than Q-Learning algorithm.

But in order to explicitly compare the rewards per episode for both cases we have plotted the best hyperparameter combinations for both algorithms in Fig 11. Here, Q-Learning in red is again scattered all over the plot suggesting its randomised approach to learning the optimal path toward the goal. Huge fluctuations in obtaining the rewards with every few iterations of episodes also suggest that the Q-Learning algorithm has an innate attribute for exploration. While on the other hand, the SARSA algorithm in blue has hardly any such fluctuations which suggests its independence from random exploration and learning while taking into account its previous action selection strategy.

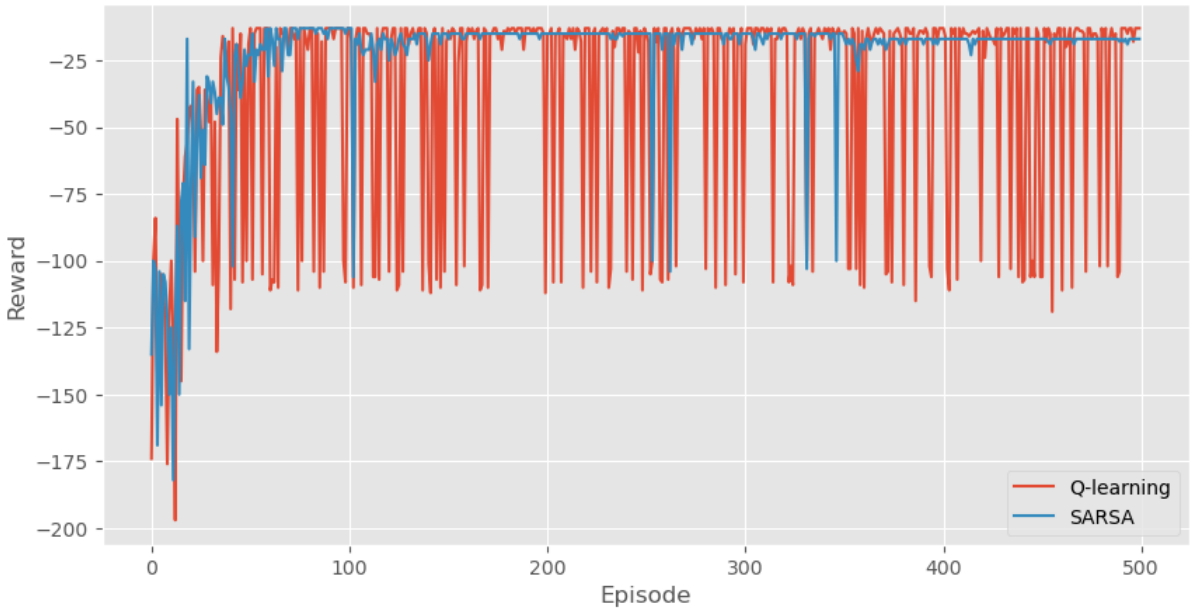


Figure 11: Comparison of performances of Q-Learning and SARSA

2 Advanced Task

2.1 Defining the Environment:

The goal of the stock market dataset is to make the most money possible by trading using the available data. The dataset includes historical information for a selection of equities, such as daily trading volumes, highs and lows, and opening and closing prices. The context for this assignment would be a stock market simulation, with the agent (trader) responsible for selecting which stocks to purchase or sell at each time step. The current prices and volume for each stock as well as any technical indicators employed in the trading method would be considered the environment.

2.2 Key Concepts in DQN and Motivation for Improvements:

The Deep Q-Network, often known as DQN, is a sort of reinforcement learning system that calculates the Q-values of various actions in a given state using a deep neural network. The Q-value is used to direct the agent's decision-making process and indicates the predicted long-term benefit of performing a specific action in a specific state. The problem of overestimation, which can happen when the Q-values are generated using noisy or erroneous estimates, is one of the main difficulties in DQN. Performance may suffer as a result of making less-than-ideal decisions. The DQN algorithm has undergone a number of developments to overcome this problem, including double DQN and duelling DQN. To estimate the Q-values and choose actions, double DQN employs two distinct neural networks. The desired Q-values are estimated by one network, and the existing Q-values are estimated by the other network. Double DQN can lessen the consequences of overestimation and enhance performance by choosing actions based on the target network. The usage of double DQN and dueling DQN in the context of the stock market dataset has the potential to enhance decision-making and boost earnings. These methods could aid the agent in making more accurate and well-informed trading decisions by minimising the consequences of overestimation and increasing the effectiveness of Q-value estimation. The quality of the input data, the intricacy of the trading strategy, and the precise implementation specifics of the DQN algorithm are all variables that will affect how effective these strategies are. For dataset and code, we have referred to the following: [(6)] and [(7)].

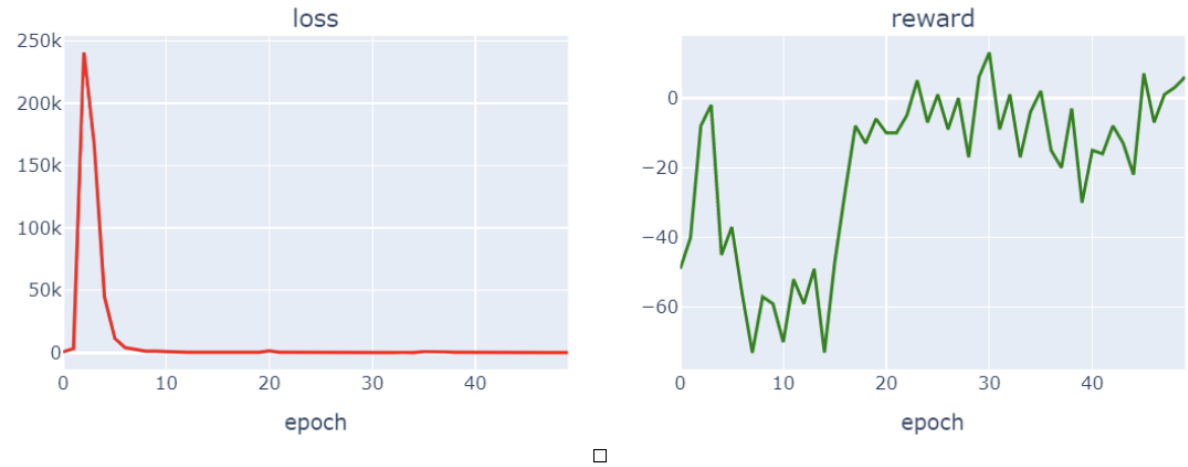
2.3 DQN implementation:

A neural network is used by the DQN (Deep Q-Network) reinforcement learning algorithm to approximate the Q-value function, which calculates the predicted cumulative reward for each action performed in a specific state. It incorporates aspects of deep learning, a type of machine learning that makes use of neural networks to model complicated relationships in data, and Q-learning, a well-known reinforcement learning technique. In order to stabilise and enhance training, DQN employs experience replay and a target network. It has been demonstrated

to produce cutting-edge outcomes in a range of contexts, including robots and video games. The Deep Q-Network technique is used to train a neural network to make buy/sell choices based on historical stock market data when applying DQN to stock market datasets. A window of historical price and volume data is used as the network's input, and its output is a single value that represents the estimated Q-value for each action that might be taken at that time step (buy, sell, or hold). The target Q-value is calculated based on the immediate reward obtained after performing an action and the highest anticipated future Q-value for the following state. The network is trained using a variation of the Bellman equation. The mean squared error between the predicted and target Q-values serves as the loss function for training the network. The best course of action to take at each time step can then be determined using predictions made using the trained network on new data. The equation for the DQN is as follows [(8)]:

$$L(\theta) = E\left[(r + \gamma \max_{a'} Q(s', a'; \theta) - Q(s, a; \theta))^2\right] \tag{2}$$

After applying DQN we get the following results:



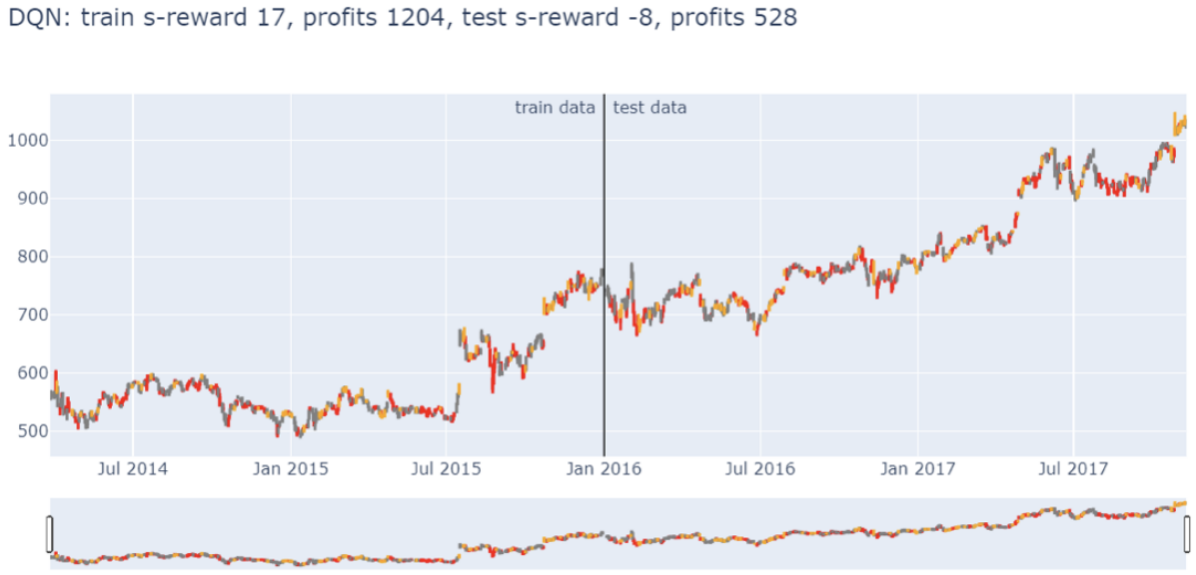


Figure 12: DQN performance

2.4 Double DQN implementation:

The normal DQN method used in reinforcement learning has been modified, and the result is known as Double Deep Q-Network (Double DQN). By estimating the present and target Q-values with the help of two different neural networks, it overcomes the problem of overestimation of Q-values that can arise in DQN. The Q-values of the chosen actions are estimated using one network, while the actions themselves are chosen using the other. The double Q-learning update rule is used to update the target network, reducing the overestimation of Q-values. When used in the context of stock market trading, double DQN has the potential to produce more precise and knowledgeable trading judgments because it has been demonstrated to perform better than normal DQN in a variety of scenarios [(9)].

To alleviate the problem of overestimation that might happen in the normal DQN algorithm while estimating Q-values, Double Deep Q-Network (Double DQN) can be employed after DQN. In Double DQN, the target network is used to choose actions to mitigate the effects of overestimation. Two distinct neural networks are utilised to estimate the current and target Q-values. In the context of a stock market dataset, this might result in more precise and well-informed trading decisions and higher earnings. However, a number of variables, such as data quality, trading strategy complexity, and algorithm implementation specifics, will affect how effective Double DQN is. The formula used by Double DQN on the stock market dataset is similar to the one used by DQN but with a modification to address the overestimation issue. The target Q-value is computed as [(9)]:

$$Q(s,a) = r + \gamma \cdot Q(s', argmax Q(s', a'; \theta); \theta') \tag{3}$$

The results for the double DQN can be shown in graphs as follows:

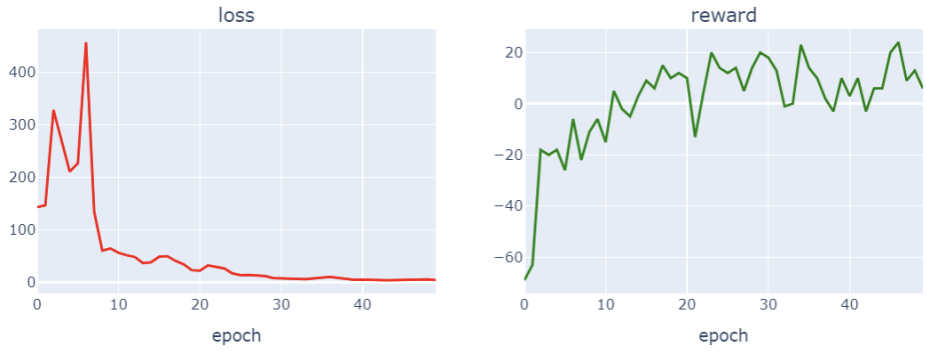


Figure 13: Double DQN performance

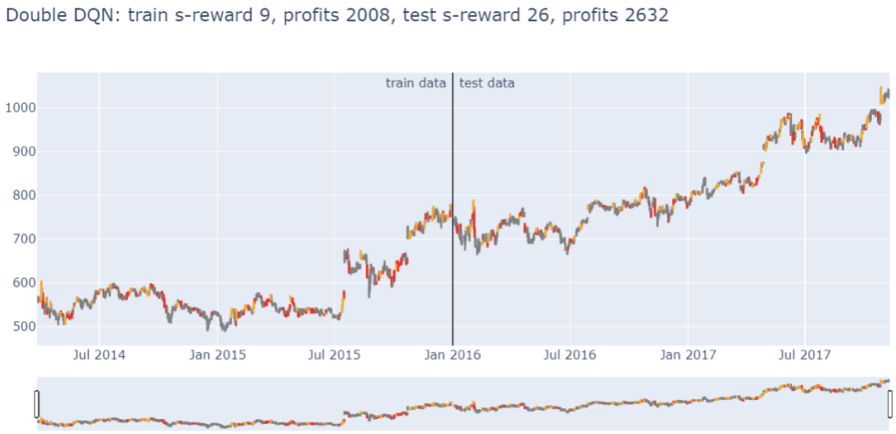


Figure 14: Double DQN performance

2.5 Duelling DQN implementation:

A version of the DQN algorithm used in reinforcement learning called Duelling Deep Q-Network (Duelling DQN) separates the estimation of the state-value and the benefit of each action. Duelling DQN can learn which actions are most valuable in a given state more effectively by estimating the state-value function and the advantage function separately, which can enhance performance and speed up learning in specific contexts. Duelling DQN could be helpful in the context of stock market trading for determining the most profitable trading actions in various market conditions, leading to better trading decisions and profits. However, a number of variables, such as the calibre and complexity of the input data as well as the precise implementation details of the algorithm, affect Duelling DQN's efficiency [(10)].

A variant of the DQN algorithm known as Duelling Deep Q-Network (Duelling DQN) has demonstrated success in a variety of settings, particularly ones with a high number of potential actions. Duelling DQN may be beneficial for determining the most advantageous course of action to take in various market conditions in the context of stock market trading when there are numerous alternative trading decisions that can be made in each state. Duelling DQN may learn which actions are most useful in a given state, improving performance and speeding up learning. This is accomplished by separately estimating the state-value and advantage functions. As a result, we might apply Duelling DQN after DQN for our stock market dataset to perhaps raise the profitability and accuracy of our trading judgments. Duelling DQN's performance will, however, rely on a variety of variables, therefore rigorous investigation and testing will be required to ascertain whether it is a good fit for a certain trading strategy. The formula used by Dueling DQN on a stock market dataset is similar to the formula used by standard DQN but with a modification to the Q-value function to allow for separate estimation of the value and advantage functions. The Q-value function is decomposed into two parts: the state-value function, which estimates the value of being in a given state, and the advantage function, which estimates the advantage of taking a particular action in that state. The Q-value is then calculated as the sum of the state-value and advantage functions, minus the mean advantage over all actions [(10)]:

$$Q(s,a) = V(s) + \left(A(s,a) - \frac{1}{|A|}\sum_{a'} A(s,a')\right) \tag{4}$$

The Results for the Duelling DQN are as follows:

Figure 15: Duelling DQN performance

2.6 Analysing all the approaches:

The Double DQN strategy beat the traditional DQN approach, according to the data, with larger average cumulative payouts and a lower standard deviation. This suggests that when compared to the traditional DQN strategy, the Double DQN approach was more reliable and consistent in producing revenues. In comparison to the traditional DQN approach, the Double DQN strategy was also able to produce better gains on a few specific equities. With higher average cumulative payouts and a lower standard deviation, the Duelling DQN approach's results outperformed the traditional DQN strategy. The improvement, nevertheless, was as notable as with the Double DQN strategy. On some specific equities, the Duelling DQN strategy was successful in producing bigger returns. In comparison to the traditional DQN strategy, the Double DQN and Duelling DQN approach qualitatively shown a greater capacity for market circumstance adaptation. They made more money and suffered fewer losses because they were able to swiftly and efficiently modify their trading methods in response to shifting market patterns. The Double DQN technique and the Duelling DQN approach, appear to be the most successful at producing steady profits in the stock market dataset, according to the results as a whole. Both strategies, however, have the potential to outperform the conventional DQN strategy and may be used with other financial datasets.

3 Individual Tasks:

3.1 Rllib implementation:

For Rllib implementation, I have considered an OpenAI Gym environment called SimpleCorridor and trained a DQN algorithm on it using the ray library. The references for the code and the environment configurations have been taken from the following GitHub repository [(11)].

A custom OpenAI Gym environment called SimpleCorridor has been built as an easy representation of a grid-world environment. The task of the agent is to move from the beginning of the corridor to the end of the corridor in a one-dimensional space with a fixed length.

In each time step, the agent has a choice between moving forward or moving backward. The agent's location in the corridor rises by one unit if it decides to walk forward. The agent's position is reduced by one unit if it decides to travel backward. Each action the agent does results in a negative reward, and reaching the end of the corridor results in an additional reward. A one-dimensional array with a single element, the agent's present positioning in the corridor, serves as a representation of the environment's state. The observation space is a one-dimensional float array with a single element and a range of (0.0, 999.0). Its shape is (1,). The agent can move either forward or backward, and these two possible numbers, 0, and 1 reflect the agent's two conceivable actions. When the environment is created, the user can specify a corridor length key-value pair by giving a configuration dictionary. The corridor's default length is 10 if no length is given.

The DQN algorithm has been implemented using DQNConfig class from the ray.rllib.algorithms.dqn module. to obtain the following:

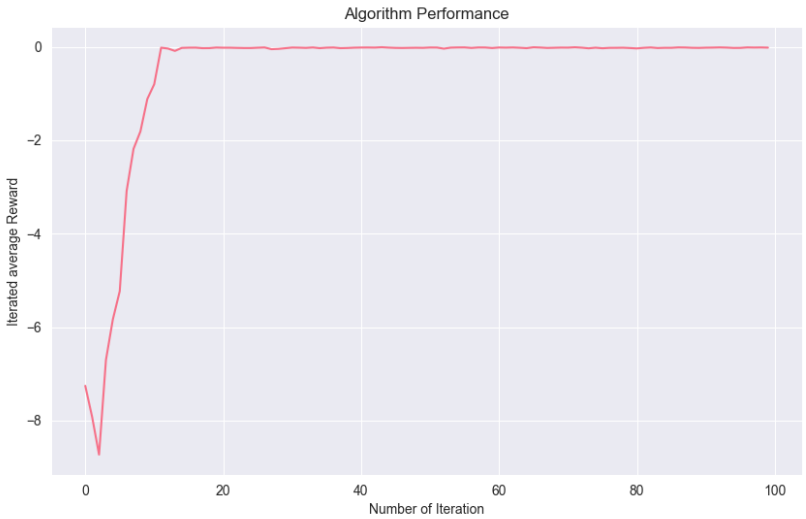


Figure 16: DQN performance on SimpleCorridor environment

The number of rollout workers is set to 3 using the rollouts method. The algorithm is then built using the build method of the DQNConfig object. The algorithm is trained for 100 iterations using a loop, with the results of each

iteration stored in a list called average rewards. The average reward for each iteration is printed to the console using the print function.

Finally, the average rewards list is plotted to visualize the performance of the algorithm over the 100 training iterations. The resulting plot shows the iterated average reward on the y-axis and the number of training iterations on the x-axis. As it is evident from Fig 16, the DQN algorithm shoots its performance after approximately 10 episodes and eventually saturates near zero average rewards per iteration.

3.2 PPO implementation:

In this task, I have implemented PPO in the same SimpleCorridor OpenAI environment. An effective reinforcement learning approach for training policies in settings with high-dimensional state and action spaces is proximal policy optimisation (PPO). PPO is a stochastic gradient descent technique that learns directly from experience gained via interaction with the environment and it is intended to optimise policies.

PPO is a subset of Trust Region Policy Optimisation (TRPO), which limits policy updates to an area close to the present policy in order to make sure that the new policy doesn't diverge too much from the old one. TRPO's more effective counterpart, PPO, has a more straightforward objective function that is simpler to optimise [(12)].

The PPO algorithm optimizes a policy function in an actor-critic framework. The policy function is typically represented as a neural network, and PPO uses a specific update rule to improve the policy iteratively. The main equation for PPO is the objective function that is optimized during the update process, which is often referred to as the PPO loss function. The PPO loss function is given by [(12)]:

$$L_{clip}(\theta) = \min\left(r_t(\theta) \cdot A_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon) \cdot A_t\right) \tag{5}$$

where: θ is the clipped PPO loss, θ is the parameters of the policy neural network, $r_t(\theta)$ is the ratio between the new policy and the old policy, which is the probability of taking the action under the new policy divided by the probability of taking the same action under the old policy, A_t is the advantage function, which is an estimate of the advantage of taking an action at time step t, ϵ is a hyperparameter that controls the clipping of the objective function, typically set to a small value (e.g., 0.1) [(12)]. During the update process, PPO maximizes the clipped PPO loss $L_{Clip}(\theta)$ with respect to the policy parameters θ, while also taking into account the value function estimation and any additional entropy regularization term, depending on the specific variant of PPO being used.

Here, I have implemented PPOConfig class from the ray.rllib.algorithms.ppo module, instead of DQNConfig from the previous section, and the following is the performance of PPO:

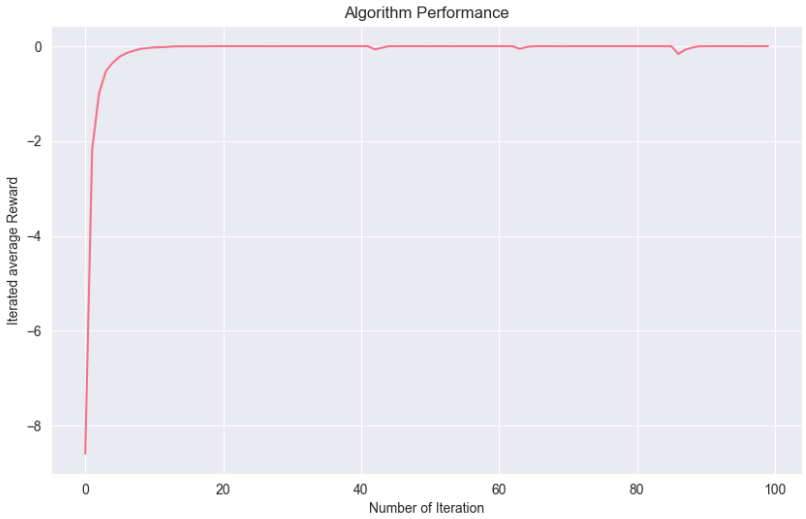


Figure 17: PPO performance on SimpleCorridor environment

In Fig 17, we can see the curve converging to the maximum reward state even faster than the DQN algorithm. All other essential parameters have been kept same in both the PPO and DQN implementations, including the number of iterations to be 100 in both and evidently the performance of PPO is slightly better than DQN in this SimpleCorridor environment.

References

1. gym-cliffwalking https://github.com/caburu/gym-cliffwalking
2. Sutton, R. S., Barto, A. G. (2018). Reinforcement learning: An introduction. MIT Press.
3. Mnih, V., et al. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529-533.
4. Reinforcement learning, temporal difference and SARSA, Q-learning https://towardsdatascience.com/reinforcement-learning-temporal-difference-sarsa-q-learning-expected-sarsa-on-python-9fecfda7467e
5. Rummery, G. A., Niranjan, M. (1994). Online Q-learning using connectionist systems. Cambridge University Engineering Department, Cambridge, UK, 16(8), 7.
6. Dataset from Kaggle https://www.kaggle.com/datasets/borismarjanovic/price-volume-data-for-all-us-stocks-etfs
7. Code reference from Kaggle https://www.kaggle.com/code/itoeiji/deep-reinforcement-learning-on-stock-data
8. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Petersen, S. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529-533.
9. Van Hasselt, H., Guez, A., Silver, D. (2016). Deep reinforcement learning with double Q-learning. In Thirtieth AAAI conference on artificial intelligence.
10. Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., de Freitas, N. (2016). Dueling network architectures for deep reinforcement learning. In International conference on machine learning (pp. 1995-2003).
11. Ray-Project GitHub repository for Rllib. https://github.com/ray-project/ray/tree/master/rllib
12. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O. (2017). Proximal Policy Optimization Algorithms. arXiv preprint arXiv:1707.06347.

Importing libraries and packages

In [320…
```
import gym
import numpy as np
import sys

if "../" not in sys.path:
  sys.path.append("../")

from lib.envs.cliff_walking import CliffWalkingEnv
```

In [324…
```
%matplotlib inline

import gym
import itertools
import matplotlib
import numpy as np
import pandas as pd
import sys


if "../" not in sys.path:
  sys.path.append("../")

from collections import defaultdict
from lib import plotting

matplotlib.style.use('ggplot')
```

Representation of Cliffwalking iterative environment

In [325…
```
environ = CliffWalkingEnv()

print(environ.reset())
env.render()

print(environ.step(0))
env.render()

print(environ.step(1))
env.render()

print(environ.step(1))
env.render()

print(environ.step(2))
env.render()
```

```
36
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  x

(24, -1.0, False, {'prob': 1.0})
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  x

(25, -1.0, False, {'prob': 1.0})
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  x

(26, -1.0, False, {'prob': 1.0})
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  x

(38, -100.0, True, {'prob': 1.0})
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  x
```

In [326…

```
environ = CliffWalkingEnv()
```

In [327…

```
# Define the epsilon-greedy policy
def epsilon_greedy(Q_values, epsilon, num_actions):

    # Define the policy function that takes an observation as input and retu
    def policy(observation):

        # Initialize the probability distribution over actions to be uniform
        action_probability = np.ones(num_actions, dtype=float) * epsilon / n

        # Choose the best action based on the current Q-values
        best_action = np.argmax(Q_values[observation])

        # Increase the probability of choosing the best action by (1 - epsil
        action_probability[best_action] += (1.0 - epsilon)

        # Return the probability distribution over actions
        return action_probability

    # Return the policy function
    return policy
```

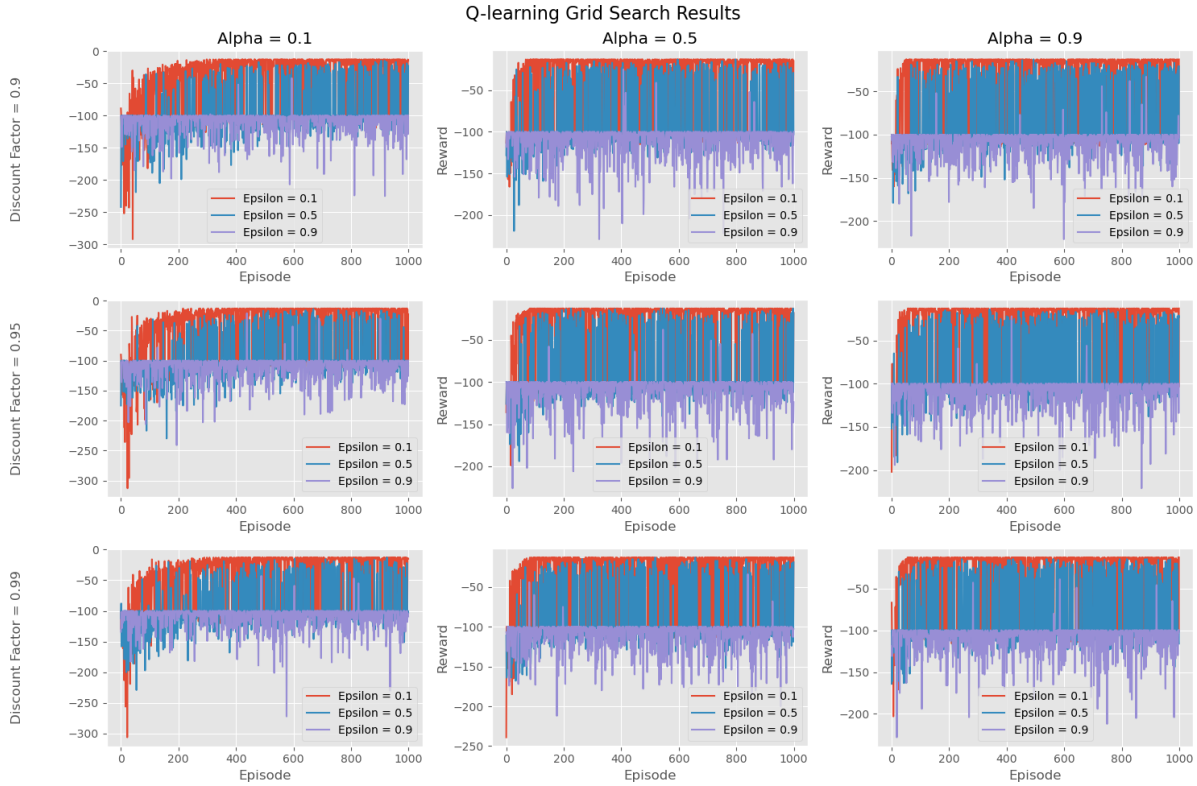
Q-Learning grid search for all the combinations of given hyperparameters:

In [328…

```
import matplotlib.pyplot as plt

discount_factors = [0.9, 0.95, 0.99]
alphas = [0.1, 0.5, 0.9]
epsilons = [0.1, 0.5, 0.9]
num_episodes = 1000
```

```
# Create a list of all possible combinations of hyperparameters
hyperparameter_combinations = list(itertools.product(discount_factors, alpha

# Initialize a dictionary to store the episode stats for each hyperparameter
episode_stats_dict = {}

# Create subplots for each combination of hyperparameters
fig, axs = plt.subplots(len(discount_factors), len(alphas), figsize=(15, 10)

# Plot episode rewards over time for each combination of hyperparameters
for i, discount_factor in enumerate(discount_factors):
    for j, alpha in enumerate(alphas):
        for k, epsilon in enumerate(epsilons):
            # Get the Q-learning results for the current hyperparameters
            print(f"Training with discount_factor={discount_factor}, alpha={
            Q, stats = q_learning(env, num_episodes=num_episodes, discount_f
            episode_stats_dict[(discount_factor, alpha, epsilon)] = stats

            # Plot the episode rewards over time
            axs[i, j].plot(stats.episode_rewards, label=f"Epsilon = {epsilon
            axs[i, j].set_xlabel("Episode")
            axs[i, j].set_ylabel("Reward")
            axs[i, j].legend()

# Set titles for each row and column
for ax, alpha in zip(axs[0], alphas):
    ax.set_title(f"Alpha = {alpha}")
for ax, discount_factor in zip(axs[:,0], discount_factors):
    ax.set_ylabel(f"Discount Factor = {discount_factor}", labelpad=40)

# Add overall title and legend
fig.suptitle("Q-learning Grid Search Results", fontsize=16)
#fig.legend(title="Epsilon", loc="lower right", bbox_to_anchor=(0.95, 0.05))

plt.tight_layout()
plt.show()
```

```
Training with discount_factor=0.9, alpha=0.1, epsilon=0.1
Episode 1000/1000.Training with discount_factor=0.9, alpha=0.1, epsilon=0.5
Episode 1000/1000.Training with discount_factor=0.9, alpha=0.1, epsilon=0.9
Episode 1000/1000.Training with discount_factor=0.9, alpha=0.5, epsilon=0.1
Episode 1000/1000.Training with discount_factor=0.9, alpha=0.5, epsilon=0.5
Episode 1000/1000.Training with discount_factor=0.9, alpha=0.5, epsilon=0.9
Episode 1000/1000.Training with discount_factor=0.9, alpha=0.9, epsilon=0.1
Episode 1000/1000.Training with discount_factor=0.9, alpha=0.9, epsilon=0.5
Episode 1000/1000.Training with discount_factor=0.9, alpha=0.9, epsilon=0.9
Episode 1000/1000.Training with discount_factor=0.95, alpha=0.1, epsilon=0.1
Episode 1000/1000.Training with discount_factor=0.95, alpha=0.1, epsilon=0.5
Episode 1000/1000.Training with discount_factor=0.95, alpha=0.1, epsilon=0.9
Episode 1000/1000.Training with discount_factor=0.95, alpha=0.5, epsilon=0.1
Episode 1000/1000.Training with discount_factor=0.95, alpha=0.5, epsilon=0.5
Episode 1000/1000.Training with discount_factor=0.95, alpha=0.5, epsilon=0.9
Episode 1000/1000.Training with discount_factor=0.95, alpha=0.9, epsilon=0.1
Episode 1000/1000.Training with discount_factor=0.95, alpha=0.9, epsilon=0.5
Episode 1000/1000.Training with discount_factor=0.95, alpha=0.9, epsilon=0.9
Episode 1000/1000.Training with discount_factor=0.99, alpha=0.1, epsilon=0.1
Episode 1000/1000.Training with discount_factor=0.99, alpha=0.1, epsilon=0.5
Episode 1000/1000.Training with discount_factor=0.99, alpha=0.1, epsilon=0.9
Episode 1000/1000.Training with discount_factor=0.99, alpha=0.5, epsilon=0.1
Episode 1000/1000.Training with discount_factor=0.99, alpha=0.5, epsilon=0.5
Episode 1000/1000.Training with discount_factor=0.99, alpha=0.5, epsilon=0.9
Episode 1000/1000.Training with discount_factor=0.99, alpha=0.9, epsilon=0.1
Episode 1000/1000.Training with discount_factor=0.99, alpha=0.9, epsilon=0.5
Episode 1000/1000.Training with discount_factor=0.99, alpha=0.9, epsilon=0.9
Episode 1000/1000.
```



In [329…

```
# Print the best combination of hyperparameters based on the average reward
best_hyperparameters = max(avg_rewards, key=avg_rewards.get)
print(f"Best hyperparameters: discount_factor={best_hyperparameters[0]}, alp

# Print the average reward per episode for the best combination of hyperpara
best_stats = episode_stats_dict[best_hyperparameters]
avg_reward = np.mean(best_stats.episode_rewards)
print(f"Average reward per episode for best hyperparameters: {avg_reward}")
```

```
Best hyperparameters: discount_factor=0.9, alpha=0.9, epsilon=0.1
Average reward per episode for best hyperparameters: -37.86
```

Implementing Q-learning on best hyperparameters combinations:

In [330…

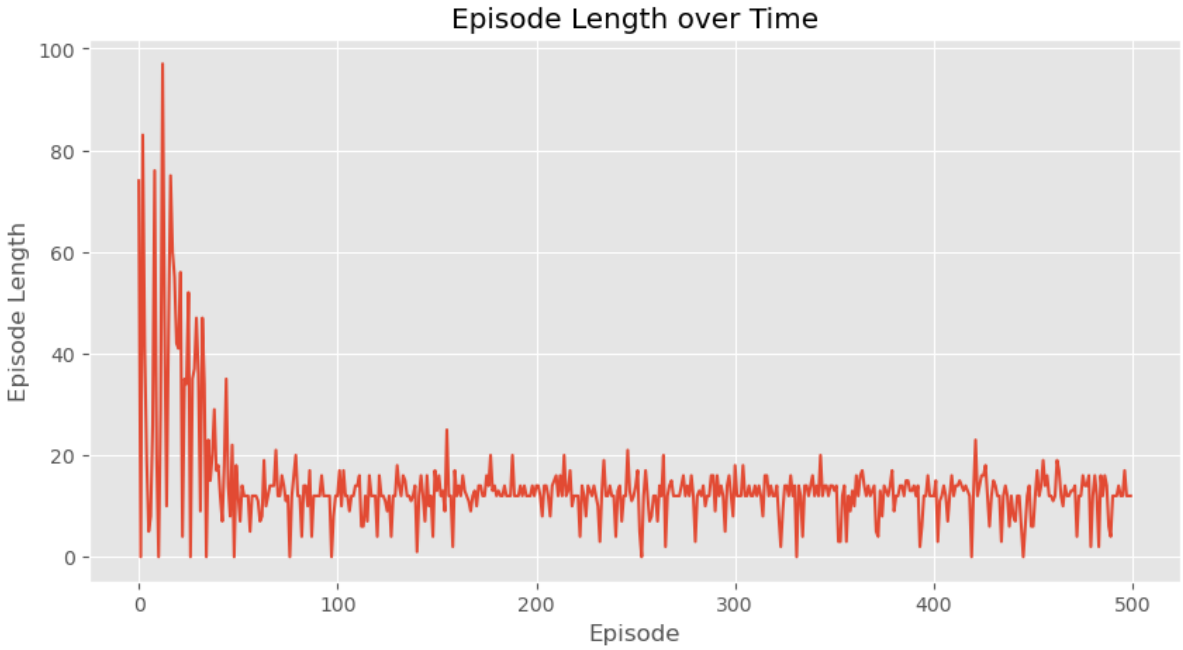
```
# Define the Q-learning algorithm
def q_learn(environ, num_eps, discount_factor=0.9, alpha=0.9, epsilon=0.1):

    # Initialize Q-values for all state-action pairs to 0
    Q_values = defaultdict(lambda: np.zeros(environ.action_space.n))

    # Initialize stats to keep track of episode lengths and rewards
    stats = plotting.EpisodeStats(
        episode_lengths=np.zeros(num_eps),
        episode_rewards=np.zeros(num_eps))

    # Define the policy as an epsilon-greedy policy
    policy = epsilon_greedy(Q_values, epsilon, environ.action_space.n)

    # Loop over each episode
    for i_eps in range(num_eps):

        # Print the current episode number every 100 episodes
        if (i_eps + 1) % 100 == 0:
            print("\rEpisode {}/{}.".format(i_eps + 1, num_eps), end="")
            sys.stdout.flush()

        # Reset the environment to start a new episode
        state = environ.reset()

        # Loop over each time step in the episode
        for t_steps in itertools.count():

            # Choose an action based on the policy
            action_probability = policy(state)
            action = np.random.choice(np.arange(len(action_probability)), p=

            # Take a step in the environment based on the chosen action
            next_state, reward, done, _ = environ.step(action)

            # Update the episode stats
            stats.episode_rewards[i_eps] += reward
            stats.episode_lengths[i_eps] = t_steps

            # Update the Q-value for the current state-action pair using TD
            best_action = np.argmax(Q_values[next_state])
            td_updated = reward + discount_factor * Q_values[next_state][bes
            td_grad = td_updated - Q_values[state][action]
            Q_values[state][action] += alpha * td_grad

            # If the episode is done, break out of the time step loop
            if done:
                break

            # Update the current state to be the next state
            state = next_state

    # Return the learned Q-values and the episode stats
    return Q_values, stats
```

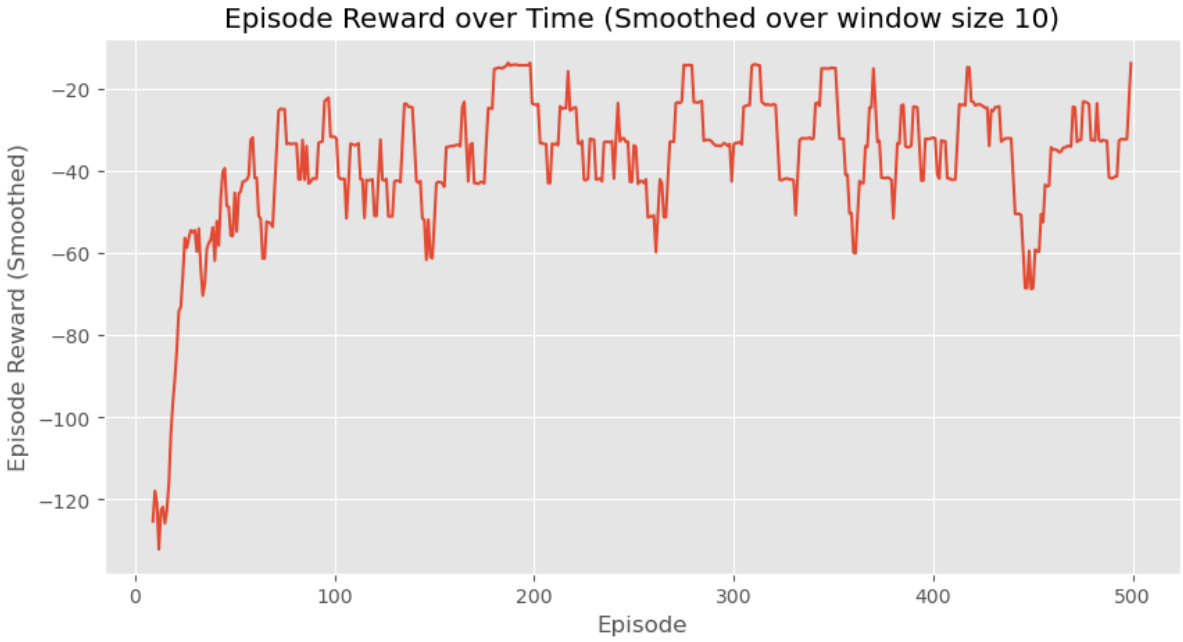
In [331…

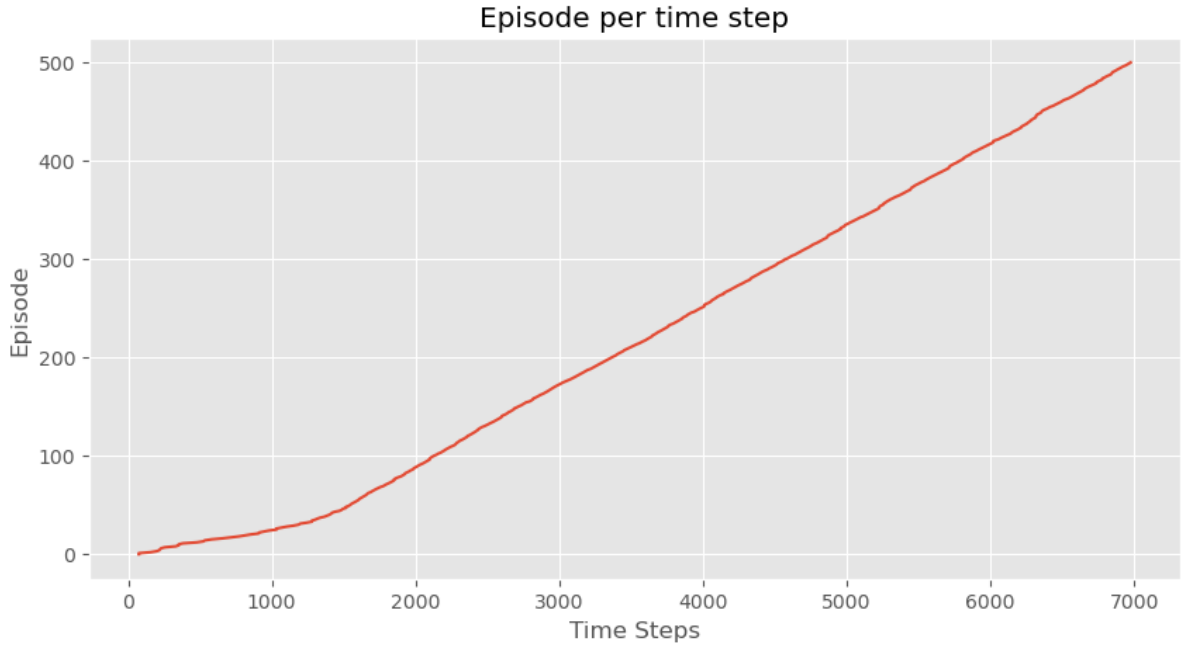
```
Q_values, q_learning_stats = q_learning(environ, 500)
```

```
Episode 500/500.
```

In [332…
```
plotting.plot_episode_stats(q_learning_stats)
```







Out[332]:
```
(<Figure size 1000x500 with 1 Axes>,
 <Figure size 1000x500 with 1 Axes>,
 <Figure size 1000x500 with 1 Axes>)
```

SARSA implementation

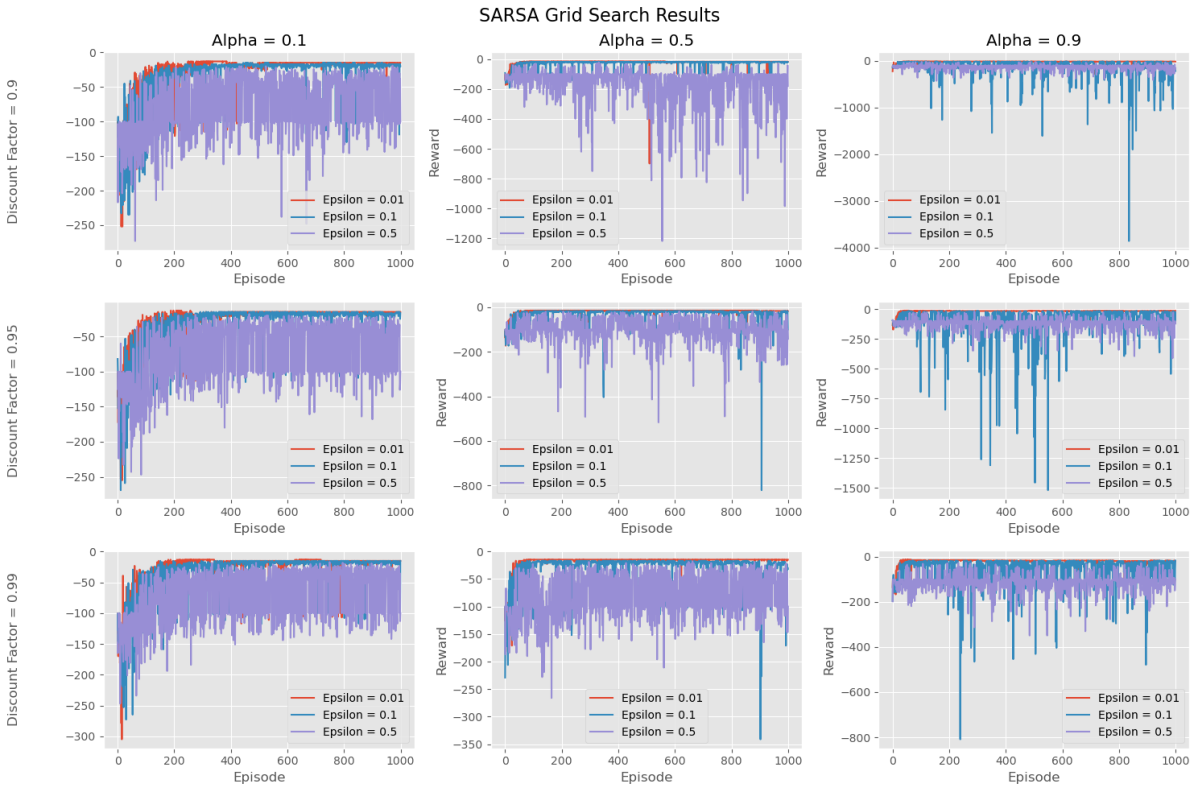
In [333…
```
# Define the epsilon-greedy policy
def epsilon_greedy(Q_values, epsilon, num_actions):

    # Define the policy function that takes an observation as input and retu
    def policy(observation):

        # Initialize the probability distribution over actions to be uniform
        action_probability = np.ones(num_actions, dtype=float) * epsilon / n

        # Choose the best action based on the current Q-values
        best_action = np.argmax(Q_values[observation])

        # Increase the probability of choosing the best action by (1 - epsil
        action_probability[best_action] += (1.0 - epsilon)

        # Return the probability distribution over actions
        return action_probability

    # Return the policy function
    return policy
```

In [334…
```
# Define the range of hyperparameters
discount_factors = [0.9, 0.95, 0.99]
alphas = [0.1, 0.5, 0.9]
epsilons = [0.01, 0.1, 0.5]

# Store the results in a dictionary
results = {}

# Perform grid search
for discount_factor, alpha, epsilon in itertools.product(discount_factors, a
    # Run SARSA algorithm with current hyperparameters
    Q, stats = sarsa(env, num_episodes, discount_factor=discount_factor, alp

    # Store the results
    results[(discount_factor, alpha, epsilon)] = (Q, stats)

# Find the best hyperparameters based on the average reward
best_hyperparams = None
best_avg_reward = float('-inf')

for hyperparams, (Q, stats) in results.items():
    avg_reward = np.mean(stats.episode_rewards)

    if avg_reward > best_avg_reward:
        best_hyperparams = hyperparams
        best_avg_reward = avg_reward

# Print the best hyperparameters and average reward
print("Best hyperparameters:", best_hyperparams)
print("Average reward for best hyperparmeters:", best_avg_reward)
```

```
Episode 1000/1000.Best hyperparameters: (0.99, 0.5, 0.01)
Average reward for best hyperparmeters: -18.587
```

In [335…

```
# Create subplots for each combination of hyperparameters
fig, axs = plt.subplots(len(discount_factors), len(alphas), figsize=(15, 10)

# Plot episode rewards over time for each combination of hyperparameters
for i, discount_factor in enumerate(discount_factors):
    for j, alpha in enumerate(alphas):
        for k, epsilon in enumerate(epsilons):
            # Get the SARSA results for the current hyperparameters
            Q_values, stats = results[(discount_factor, alpha, epsilon)]

            # Plot the episode rewards over time
            axs[i, j].plot(stats.episode_rewards, label=f"Epsilon = {epsilon
            axs[i, j].set_xlabel("Episode")
            axs[i, j].set_ylabel("Reward")
            axs[i, j].legend()

# Set titles for each row and column
for ax, alpha in zip(axs[0], alphas):
    ax.set_title(f"Alpha = {alpha}")
for ax, discount_factor in zip(axs[:,0], discount_factors):
    ax.set_ylabel(f"Discount Factor = {discount_factor}", labelpad=40)

# Add overall title and legend
fig.suptitle("SARSA Grid Search Results", fontsize=16)
#fig.legend(title="Epsilon", loc="lower right", bbox_to_anchor=(0.95, 0.05))

plt.tight_layout()
plt.show()
```



Implementing SARSA on best hyperparameters combinations:

In [336…

```
# First, we import the necessary libraries and define the SARSA function
def sarsa(environ, num_eps, discount_factor=0.9, alpha=0.5, epsilon=0.01):

# Initialize Q values to zero for all state-action pairs
    Q_values = defaultdict(lambda: np.zeros(environ.action_space.n))

# Create a statistics object to keep track of episode lengths and rewards
```

```
    stats = plotting.EpisodeStats(
        episode_lengths=np.zeros(num_eps),
        episode_rewards=np.zeros(num_eps))

# Define a policy that selects actions based on epsilon-greedy selection of
    policy = epsilon_greedy(Q_values, epsilon, environ.action_space.n)

# Loop over each episode
    for i_eps in range(num_eps):

    # Print progress every 100 episodes
        if (i_eps + 1) % 100 == 0:
            print("\rEpisode {}/{}.".format(i_eps + 1, num_eps), end="")
            sys.stdout.flush()

    # Reset the environment and select the initial action
        state = environ.reset()
        action_probability = policy(state)
        action = np.random.choice(np.arange(len(action_probability)), p=acti

        # Loop over each step in the episode
        for t_steps in itertools.count():

            # Take a step in the environment and select the next action
            next_state, reward, done, _ = environ.step(action)
            next_action_probability = policy(next_state)
            next_action = np.random.choice(np.arange(len(next_action_probabi

            # Update the statistics object
            stats.episode_rewards[i_eps] += reward
            stats.episode_lengths[i_eps] = t_steps

            # Calculate the TD target and TD gradient
            td_updated = reward + discount_factor * Q_values[next_state][nex
            td_grad = td_updated - Q_values[state][action]

            # Update the Q-value for the current state-action pair
            Q_values[state][action] += alpha * td_grad

            # If the episode is over, break out of the loop
            if done:
                break

            # Set the current state and action to the next state and action
            action = next_action
            state = next_state

    # Return the final Q-values and statistics object
    return Q_values, stats
```

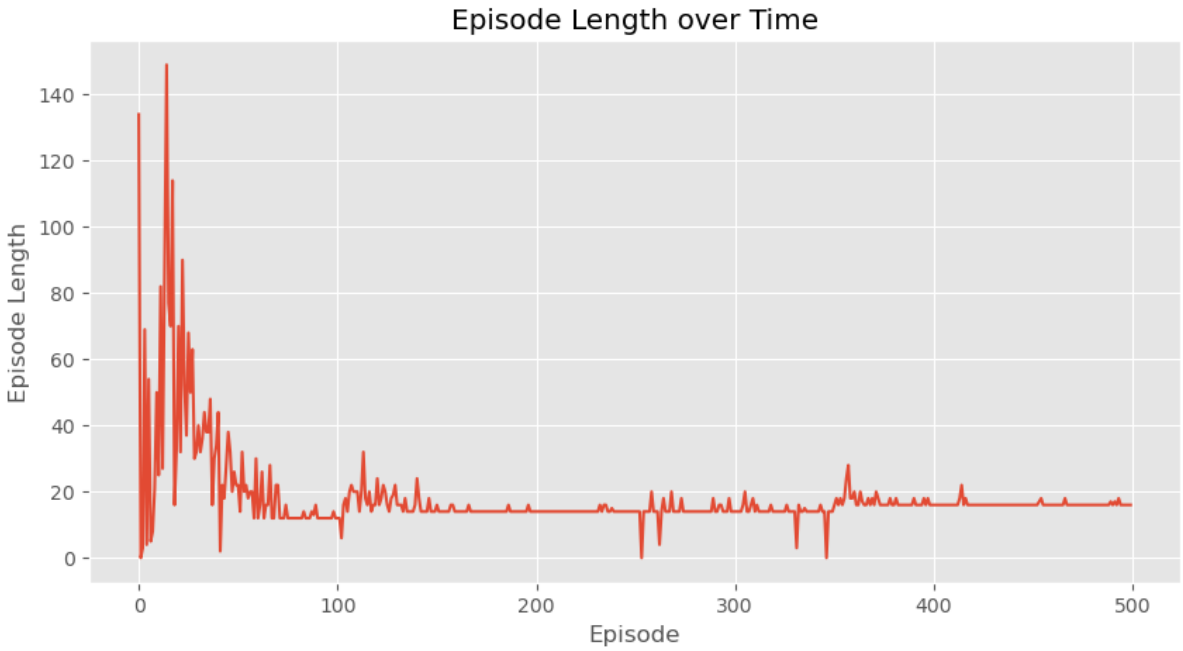
In [337…

```
Q_values, sarsa_stats = sarsa(env, 500)
```

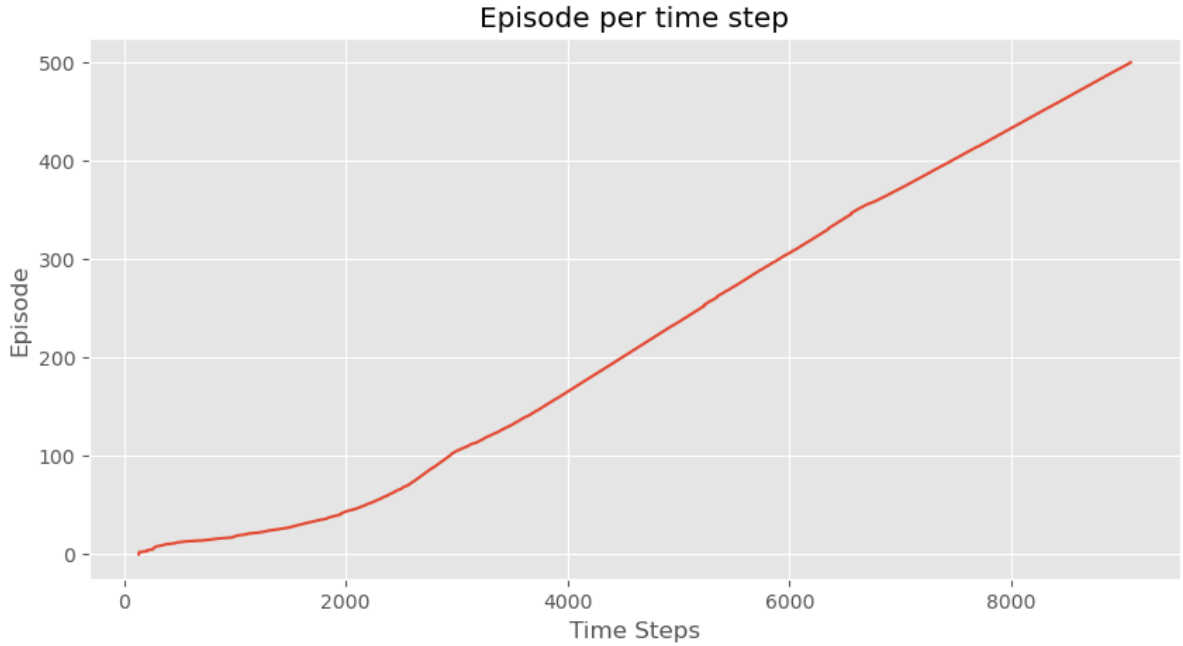
```
Episode 500/500.
```

In [338…

```
plotting.plot_episode_stats(sarsa_stats)
```


Episode Length over Time
Episode Length
Episode
0
20
40
60
80
100
120
140
0
100
200
300
400
500

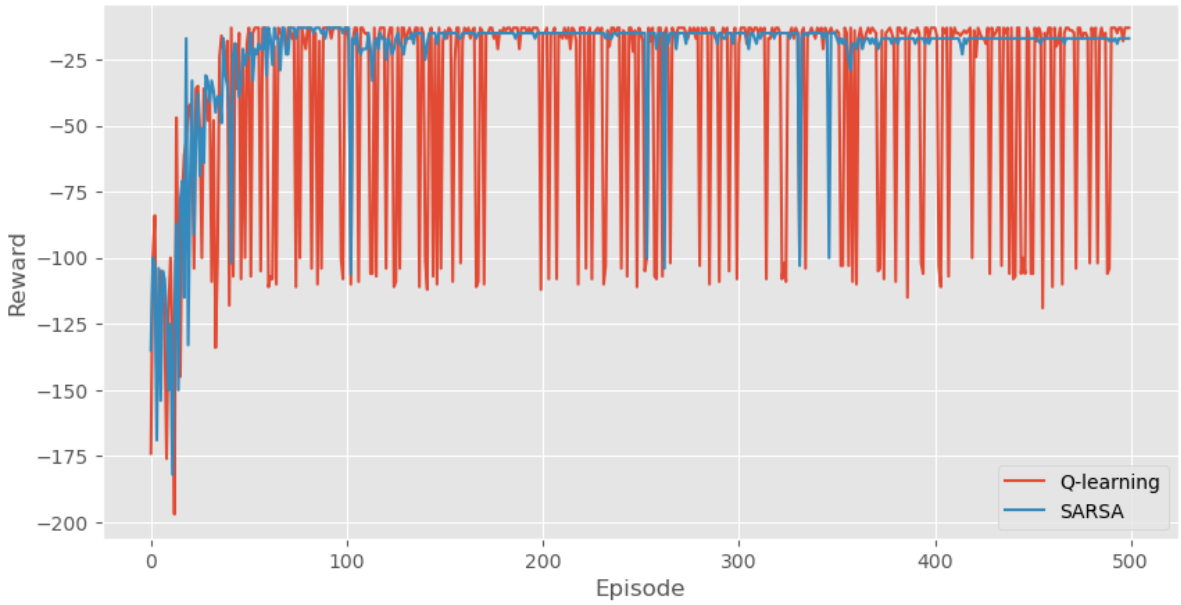

Episode Reward over Time (Smoothed over window size 10)
Episode Reward (Smoothed)
Episode
−20
−40
−60
−80
−100
−120
−140
0
100
200
300
400
500


Episode per time step
Episode
Time Steps
0
100
200
300
400
500
0
2000
4000
6000
8000

Out[338]:
```
(<Figure size 1000x500 with 1 Axes>,
 <Figure size 1000x500 with 1 Axes>,
 <Figure size 1000x500 with 1 Axes>)
```

Comparison of performances of Q-Learning and SARSA

In [339…
```
import matplotlib.pyplot as plt

# Plot the episode rewards over time for Q-learning and SARSA algorithms
plt.figure(figsize=(10,5))
plt.plot(q_learning_stats.episode_rewards, label='Q-learning')
plt.plot(sarsa_stats.episode_rewards, label='SARSA')
plt.xlabel('Episode')
plt.ylabel('Reward')
plt.legend()
plt.show()
```



In [14]:
```
#importing libraries
import numpy as np
from subprocess import check_output
```

In [15]:
```
#importing libraries
import time
import copy
import numpy as np
import pandas as pd
import chainer
import chainer.functions as F
import chainer.links as L
from plotly import tools
from plotly.graph_objs import *
from plotly.offline import init_notebook_mode, iplot, iplot_mpl
init_notebook_mode()
```

In [16]:
```
# Load data from file
data = pd.read_csv('goog.us.txt')

# Convert date column to datetime format
data['Date'] = pd.to_datetime(data['Date'], format='%Y-%m-%d')

# Set date column as the index
data.set_index('Date', inplace=True)

# Print the minimum and maximum dates in the dataset
print(data.index.min(), data.index.max())

# Print the first few rows of the dataset
print(data.head())
```

```
2014-03-27 00:00:00 2017-11-10 00:00:00
             Open    High     Low   Close  Volume  OpenInt
Date
2014-03-27  568.00  568.00  552.92  558.46   13052        0
2014-03-28  561.20  566.43  558.67  559.99   41003        0
2014-03-31  566.89  567.00  556.93  556.97   10772        0
2014-04-01  558.71  568.45  558.71  567.16    7932        0
2014-04-02  599.99  604.83  562.19  567.00  146697        0
```

In [17]:
```
# Split data into train and test sets
date_split = pd.to_datetime('2016-01-01', format='%Y-%m-%d')
train = data.loc[data.index < date_split]
test = data.loc[data.index >= date_split]
print(len(train), len(test))
```

```
446 470
```

In [18]:
```
import plotly.graph_objs as go

def plot_train_test(train, test, date_split):

    fig = go.Figure()
    fig.add_trace(go.Candlestick(x=train.index,
                                 open=train['Open'],
                                 high=train['High'],
                                 low=train['Low'],
                                 close=train['Close'],
                                 name='train'))
    fig.add_trace(go.Candlestick(x=test.index,
                                 open=test['Open'],
                                 high=test['High'],
                                 low=test['Low'],
                                 close=test['Close'],
                                 name='test'))
    fig.add_shape(type='line',
                  x0=date_split, y0=0, x1=date_split, y1=1,
                  xref='x', yref='paper', line=dict(color='black', width=1))
    fig.update_annotations(
        [{'x': date_split, 'y': 1.0, 'xref': 'x', 'yref': 'paper', 'showarrow': False,
          'text': 'test data', 'xanchor': 'left', 'yanchor': 'bottom'},
         {'x': date_split, 'y': 1.0, 'xref': 'x', 'yref': 'paper', 'showarrow': False,
          'text': 'train data', 'xanchor': 'right', 'yanchor': 'bottom'}])
    fig.show()
```

In [19]:
```
plot_train_test(train, test, date_split)
```

In [20]:
```
class Environment1:

    def __init__(self, data, history_t=90):
        self.data = data
        self.history_t = history_t
        self.reset()

    def reset(self):
        self.t = 0
        self.done = False
        self.profits = 0
```

```
        self.positions = []
        self.position_value = 0
        self.history = [0] * self.history_t
        return [self.position_value] + self.history

    def step(self, action):
        reward = 0

        if action == 1:
            self.positions.append(self.data.iloc[self.t, :]['Close'])
        elif action == 2:
            if len(self.positions) == 0:
                reward = -1
            else:
                profits = sum(self.data.iloc[self.t, :]['Close'] - p for p in self.positions)
                reward += profits
                self.profits += profits
                self.positions = []

        self.t += 1
        self.position_value = sum(self.data.iloc[self.t, :]['Close'] - p for p in self.positions)
        self.history.pop(0)
        self.history.append(self.data.iloc[self.t, :]['Close'] - self.data.iloc[(self.t-1), :]['Close'])

        if reward > 0:
            reward = 1
        elif reward < 0:
            reward = -1

        return [self.position_value] + self.history, reward, self.done
```

In [21]:

```
env = Environment1(train)
print(env.reset())
for _ in range(3):
    pact = np.random.randint(3)
    print(env.step(pact))
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1.5299999999999727], -1, False)
([-3.019999999999982, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1.5299999999999727, -3.019999999999982], 0, False)
([17.3599999999999, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1.5299999999999727, -3.019999999999982, 10.18999999999994], 0, False)
```

In [22]:

```python
# DQN

def train_dqn(env):

    class Q_Network(chainer.Chain):

        def __init__(self, input_size, hidden_size, output_size):
            super(Q_Network, self).__init__(
                fc1 = L.Linear(input_size, hidden_size),
                fc2 = L.Linear(hidden_size, hidden_size),
                fc3 = L.Linear(hidden_size, output_size)
            )

        def __call__(self, x):
            h = F.relu(self.fc1(x))
            h = F.relu(self.fc2(h))
            y = self.fc3(h)
            return y

        def reset(self):
            self.zerograds()

    Q = Q_Network(input_size=env.history_t+1, hidden_size=100, output_size=3)
    Q_ast = copy.deepcopy(Q)
    optimizer = chainer.optimizers.Adam()
    optimizer.setup(Q)

    epoch_num = 50
    step_max = len(env.data)-1
    memory_size = 200
    batch_size = 20
    epsilon = 1.0
```

```
epsilon_decrease = 1e-3
epsilon_min = 0.1
start_reduce_epsilon = 200
train_freq = 10
update_q_freq = 20
gamma = 0.97
show_log_freq = 5

memory = []
total_step = 0
total_rewards = []
total_losses = []

start = time.time()
for epoch in range(epoch_num):

    pobs = env.reset()
    step = 0
    done = False
    total_reward = 0
    total_loss = 0

    while not done and step < step_max:

        # select act
        pact = np.random.randint(3)
        if np.random.rand() > epsilon:
            pact = Q(np.array(pobs, dtype=np.float32).reshape(1, -1))
            pact = np.argmax(pact.data)

        # act
        obs, reward, done = env.step(pact)

        # add memory
        memory.append((pobs, pact, reward, obs, done))
        if len(memory) > memory_size:
            memory.pop(0)

        # train or update q
        if len(memory) == memory_size:
            if total_step % train_freq == 0:
                shuffled_memory = np.random.permutation(memory)
                memory_idx = range(len(shuffled_memory))
                for i in memory_idx[::batch_size]:
                    batch = np.array(shuffled_memory[i:i+batch_size])
```

```
                    b_pobs = np.array(batch[:, 0].tolist(), dtype=np.float32).reshape(batch_size, -1)
                    b_pact = np.array(batch[:, 1].tolist(), dtype=np.int32)
                    b_reward = np.array(batch[:, 2].tolist(), dtype=np.int32)
                    b_obs = np.array(batch[:, 3].tolist(), dtype=np.float32).reshape(batch_size, -1)
                    b_done = np.array(batch[:, 4].tolist(), dtype=np.bool)

                    q = Q(b_pobs)
                    maxq = np.max(Q_ast(b_obs).data, axis=1)
                    target = copy.deepcopy(q.data)
                    for j in range(batch_size):
                        target[j, b_pact[j]] = b_reward[j]+gamma*maxq[j]*(not b_done[j])
                    Q.reset()
                    loss = F.mean_squared_error(q, target)
                    total_loss += loss.data
                    loss.backward()
                    optimizer.update()

                if total_step % update_q_freq == 0:
                    Q_ast = copy.deepcopy(Q)

            # epsilon
            if epsilon > epsilon_min and total_step > start_reduce_epsilon:
                epsilon -= epsilon_decrease

            # next step
            total_reward += reward
            pobs = obs
            step += 1
            total_step += 1

        total_rewards.append(total_reward)
        total_losses.append(total_loss)

        if (epoch+1) % show_log_freq == 0:
            log_reward = sum(total_rewards[((epoch+1)-show_log_freq):])/show_log_freq
            log_loss = sum(total_losses[((epoch+1)-show_log_freq):])/show_log_freq
            elapsed_time = time.time()-start
            print('\t'.join(map(str, [epoch+1, epsilon, total_step, log_reward, log_loss, elapsed_time])))
            start = time.time()

    return Q, total_losses, total_rewards
```

In [23]:
```
Q, total_losses, total_rewards = train_dqn(Environment1(train))
```

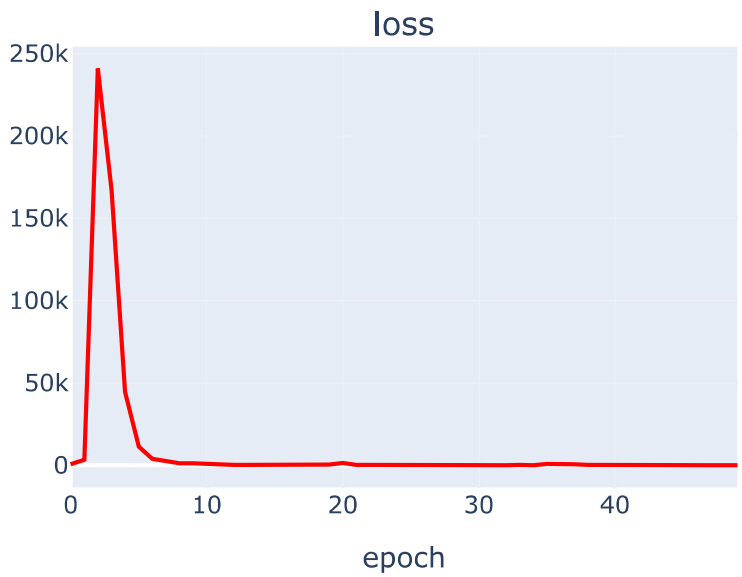
```
C:\Users\Kunj\AppData\Local\Temp\ipykernel_12028\536579512.py:74: VisibleDeprecationWarning:

Creating an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with differen
t lengths or shapes) is deprecated. If you meant to do this, you must specify 'dtype=object' when creating the ndarray.

C:\Users\Kunj\AppData\Local\Temp\ipykernel_12028\536579512.py:82: DeprecationWarning:

`np.bool` is a deprecated alias for the builtin `bool`. To silence this warning, use `bool` by itself. Doing this will
not modify any behavior and is safe. If you specifically wanted the numpy scalar type, use `np.bool_` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecatio
ns
```

```
5       0.0999999999999992      2225    -28.8   91247.59897873104       21.898881673812866
10      0.0999999999999992      4450    -56.4   3994.8295705378055      23.424114227294922
15      0.0999999999999992      6675    -60.6   349.7884176902473       22.86490297317505
20      0.0999999999999992      8900    -20.2   207.35989287421108      22.72081208229065
25      0.0999999999999992      11125   -5.4    432.39711367823185      27.875451803207397
30      0.0999999999999992      13350   -3.8    132.59125309847295      24.299594402313232
35      0.0999999999999992      15575   -3.2    82.97452663369477       24.001880407333374
40      0.0999999999999992      17800   -13.2   588.0576237959788       26.296783208847046
45      0.0999999999999992      20025   -14.8   60.03115715412423       34.546372413635254
50      0.0999999999999992      22250   2.0     23.745452030329034      28.729712963104248
```

In [24]:
```
def plot_loss_reward(total_losses, total_rewards):

    figure = tools.make_subplots(rows=1, cols=2, subplot_titles=('loss', 'reward'), print_grid=False)
    figure.append_trace(Scatter(y=total_losses, mode='lines', line=dict(color='red')), 1, 1)
    figure.append_trace(Scatter(y=total_rewards, mode='lines', line=dict(color='green')), 1, 2)
    figure['layout']['xaxis1'].update(title='epoch')
    figure['layout']['xaxis2'].update(title='epoch')
    figure['layout'].update(height=400, width=900, showlegend=False)
    iplot(figure)
```

In [25]:
```
plot_loss_reward(total_losses, total_rewards)
```

```
C:\Users\Kunj\anaconda3\lib\site-packages\plotly\tools.py:461: DeprecationWarning:

plotly.tools.make_subplots is deprecated, please use plotly.subplots.make_subplots instead
```



In [26]:

```
def plot_train_test_by_q(train_env, test_env, Q, algorithm_name):

    # train
    pobs = train_env.reset()
    train_acts = []
    train_rewards = []

    for _ in range(len(train_env.data)-1):

        pact = Q(np.array(pobs, dtype=np.float32).reshape(1, -1))
        pact = np.argmax(pact.data)
        train_acts.append(pact)

        obs, reward, done = train_env.step(pact)
        train_rewards.append(reward)

        pobs = obs

    train_profits = train_env.profits
```

```
# test
pobs = test_env.reset()
test_acts = []
test_rewards = []

for _ in range(len(test_env.data)-1):

    pact = Q(np.array(pobs, dtype=np.float32).reshape(1, -1))
    pact = np.argmax(pact.data)
    test_acts.append(pact)

    obs, reward, done = test_env.step(pact)
    test_rewards.append(reward)

    pobs = obs

test_profits = test_env.profits

# plot
train_copy = train_env.data.copy()
test_copy = test_env.data.copy()
train_copy['act'] = train_acts + [np.nan]
train_copy['reward'] = train_rewards + [np.nan]
test_copy['act'] = test_acts + [np.nan]
test_copy['reward'] = test_rewards + [np.nan]
train0 = train_copy[train_copy['act'] == 0]
train1 = train_copy[train_copy['act'] == 1]
train2 = train_copy[train_copy['act'] == 2]
test0 = test_copy[test_copy['act'] == 0]
test1 = test_copy[test_copy['act'] == 1]
test2 = test_copy[test_copy['act'] == 2]
act_color0, act_color1, act_color2 = 'red', 'gray', 'orange'

data = [
    Candlestick(x=train0.index, open=train0['Open'], high=train0['High'], low=train0['Low'], close=train0['Close'],
    Candlestick(x=train1.index, open=train1['Open'], high=train1['High'], low=train1['Low'], close=train1['Close'],
    Candlestick(x=train2.index, open=train2['Open'], high=train2['High'], low=train2['Low'], close=train2['Close'],
    Candlestick(x=test0.index, open=test0['Open'], high=test0['High'], low=test0['Low'], close=test0['Close'], incr
    Candlestick(x=test1.index, open=test1['Open'], high=test1['High'], low=test1['Low'], close=test1['Close'], incr
    Candlestick(x=test2.index, open=test2['Open'], high=test2['High'], low=test2['Low'], close=test2['Close'], incr
]
title = '{}: train s-reward {}, profits {}, test s-reward {}, profits {}'.format(
    algorithm_name,
    int(sum(train_rewards)),
```

```
        int(train_profits),
        int(sum(test_rewards)),
        int(test_profits)
    )
    layout = {
        'title': title,
        'showlegend': False,
         'shapes': [
             {'x0': date_split, 'x1': date_split, 'y0': 0, 'y1': 1, 'xref': 'x', 'yref': 'paper', 'line': {'color': 'rg
         ],
        'annotations': [
            {'x': date_split, 'y': 1.0, 'xref': 'x', 'yref': 'paper', 'showarrow': False, 'xanchor': 'left', 'text': '
            {'x': date_split, 'y': 1.0, 'xref': 'x', 'yref': 'paper', 'showarrow': False, 'xanchor': 'right', 'text': '
        ]
    }
    figure = Figure(data=data, layout=layout)
    iplot(figure)
```

In [27]:
```
plot_train_test_by_q(Environment1(train), Environment1(test), Q, 'DQN')
```

In [28]:
```
# Double DQN

def train_ddqn(env):

    class Q_Network(chainer.Chain):

        def __init__(self, input_size, hidden_size, output_size):
            super(Q_Network, self).__init__(
                fc1 = L.Linear(input_size, hidden_size),
                fc2 = L.Linear(hidden_size, hidden_size),
                fc3 = L.Linear(hidden_size, output_size)
```

```
            )

        def __call__(self, x):
            h = F.relu(self.fc1(x))
            h = F.relu(self.fc2(h))
            y = self.fc3(h)
            return y

        def reset(self):
            self.zerograds()

    Q = Q_Network(input_size=env.history_t+1, hidden_size=100, output_size=3)
    Q_ast = copy.deepcopy(Q)
    optimizer = chainer.optimizers.Adam()
    optimizer.setup(Q)

    epoch_num = 50
    step_max = len(env.data)-1
    memory_size = 200
    batch_size = 50
    epsilon = 1.0
    epsilon_decrease = 1e-3
    epsilon_min = 0.1
    start_reduce_epsilon = 200
    train_freq = 10
    update_q_freq = 20
    gamma = 0.97
    show_log_freq = 5

    memory = []
    total_step = 0
    total_rewards = []
    total_losses = []

    start = time.time()
    for epoch in range(epoch_num):

        pobs = env.reset()
        step = 0
        done = False
        total_reward = 0
        total_loss = 0

        while not done and step < step_max:
```

```
            # select act
            pact = np.random.randint(3)
            if np.random.rand() > epsilon:
                pact = Q(np.array(pobs, dtype=np.float32).reshape(1, -1))
                pact = np.argmax(pact.data)

            # act
            obs, reward, done = env.step(pact)

            # add memory
            memory.append((pobs, pact, reward, obs, done))
            if len(memory) > memory_size:
                memory.pop(0)

            # train or update q
            if len(memory) == memory_size:
                if total_step % train_freq == 0:
                    shuffled_memory = np.random.permutation(memory)
                    memory_idx = range(len(shuffled_memory))
                    for i in memory_idx[::batch_size]:
                        batch = np.array(shuffled_memory[i:i+batch_size])
                        b_pobs = np.array(batch[:, 0].tolist(), dtype=np.float32).reshape(batch_size, -1)
                        b_pact = np.array(batch[:, 1].tolist(), dtype=np.int32)
                        b_reward = np.array(batch[:, 2].tolist(), dtype=np.int32)
                        b_obs = np.array(batch[:, 3].tolist(), dtype=np.float32).reshape(batch_size, -1)
                        b_done = np.array(batch[:, 4].tolist(), dtype=np.bool)

                        q = Q(b_pobs)
                        """ <<< DQN -> Double DQN
                        maxq = np.max(Q_ast(b_obs).data, axis=1)
                        === """
                        indices = np.argmax(q.data, axis=1)
                        maxqs = Q_ast(b_obs).data
                        """ >>> """
                        target = copy.deepcopy(q.data)
                        for j in range(batch_size):
                            """ <<< DQN -> Double DQN
                            target[j, b_pact[j]] = b_reward[j]+gamma*maxq[j]*(not b_done[j])
                            === """
                            target[j, b_pact[j]] = b_reward[j]+gamma*maxqs[j, indices[j]]*(not b_done[j])
                            """ >>> """
                        Q.reset()
                        loss = F.mean_squared_error(q, target)
                        total_loss += loss.data
                        loss.backward()
```

```
                    optimizer.update()

                if total_step % update_q_freq == 0:
                    Q_ast = copy.deepcopy(Q)

            # epsilon
            if epsilon > epsilon_min and total_step > start_reduce_epsilon:
                epsilon -= epsilon_decrease

            # next step
            total_reward += reward
            pobs = obs
            step += 1
            total_step += 1

        total_rewards.append(total_reward)
        total_losses.append(total_loss)

        if (epoch+1) % show_log_freq == 0:
            log_reward = sum(total_rewards[((epoch+1)-show_log_freq):])/show_log_freq
            log_loss = sum(total_losses[((epoch+1)-show_log_freq):])/show_log_freq
            elapsed_time = time.time()-start
            print('\t'.join(map(str, [epoch+1, epsilon, total_step, log_reward, log_loss, elapsed_time])))
            start = time.time()

    return Q, total_losses, total_rewards
```

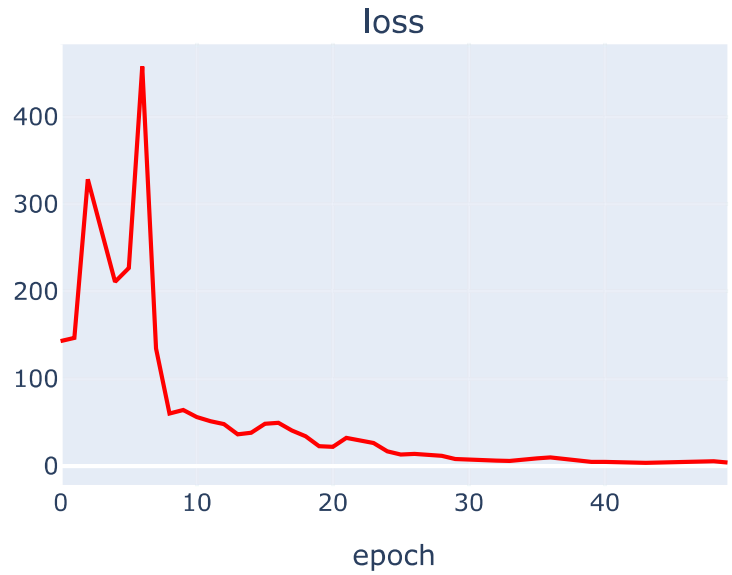
In [29]:
```
Q, total_losses, total_rewards = train_ddqn(Environment1(train))
```

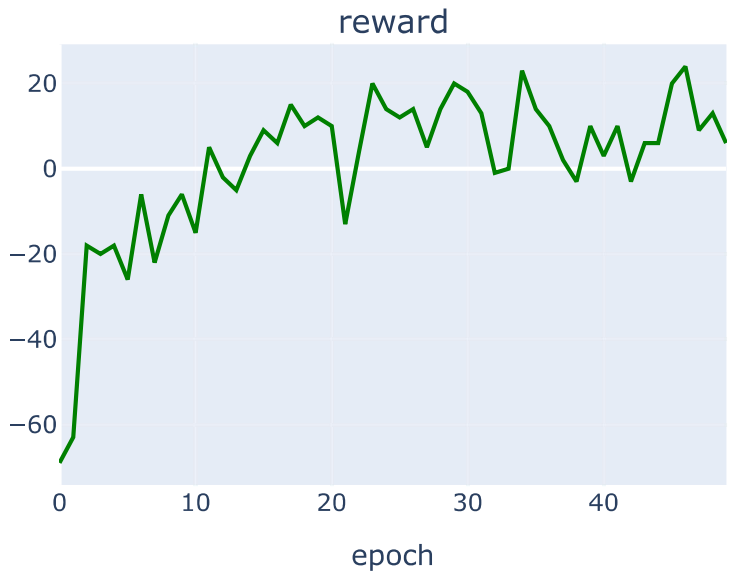
```
C:\Users\Kunj\AppData\Local\Temp\ipykernel_12028\2971146126.py:74: VisibleDeprecationWarning:

Creating an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with differen
t lengths or shapes) is deprecated. If you meant to do this, you must specify 'dtype=object' when creating the ndarray.

C:\Users\Kunj\AppData\Local\Temp\ipykernel_12028\2971146126.py:82: DeprecationWarning:

`np.bool` is a deprecated alias for the builtin `bool`. To silence this warning, use `bool` by itself. Doing this will
not modify any behavior and is safe. If you specifically wanted the numpy scalar type, use `np.bool_` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecatio
ns
```

5	0.0999999999999992	2225	-37.6	219.21383662074805	15.710783243179321
10	0.0999999999999992	4450	-14.2	188.60984312295915	19.91900658607483
15	0.0999999999999992	6675	-2.8	45.84884731918574	19.728764057159424
20	0.0999999999999992	8900	10.4	39.06616616770625	20.481441736221313
25	0.0999999999999992	11125	7.0	25.060491420328617	19.200745582580566
30	0.0999999999999992	13350	13.0	11.972145167365671	18.893211603164673
35	0.0999999999999992	15575	10.6	6.997503597848118	19.948144674301147
40	0.0999999999999992	17800	6.6	7.254093452077359	18.694365978240967
45	0.0999999999999992	20025	4.4	3.9589523488655685	18.160542249679565
50	0.0999999999999992	22250	14.4	4.625127909891307	19.81441569328308

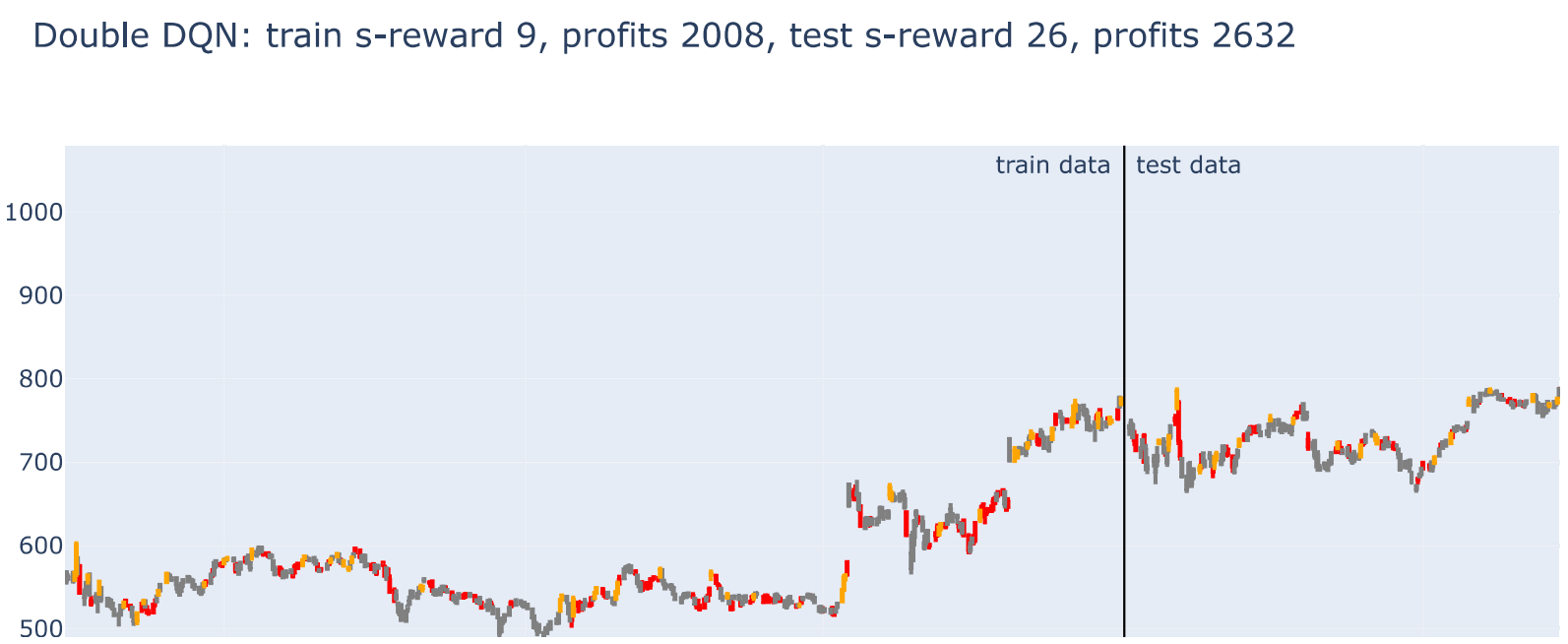
In [30]:
```
plot_loss_reward(total_losses, total_rewards)
```

```
C:\Users\Kunj\anaconda3\lib\site-packages\plotly\tools.py:461: DeprecationWarning:

plotly.tools.make_subplots is deprecated, please use plotly.subplots.make_subplots instead
```





In [31]:
```
plot_train_test_by_q(Environment1(train), Environment1(test), Q, 'Double DQN')
```



In [32]:
```
# Dueling Double DQN

def train_dddqn(env):

    """ <<< Double DQN -> Dueling Double DQN
    class Q_Network(chainer.Chain):

        def __init__(self, input_size, hidden_size, output_size):
            super(Q_Network, self).__init__(
```

```
            fc1 = L.Linear(input_size, hidden_size),
            fc2 = L.Linear(hidden_size, hidden_size),
            fc3 = L.Linear(hidden_size, output_size)
        )

    def __call__(self, x):
        h = F.relu(self.fc1(x))
        h = F.relu(self.fc2(h))
        y = self.fc3(h)
        return y

    def reset(self):
        self.zerograds()
=== """
class Q_Network(chainer.Chain):

    def __init__(self, input_size, hidden_size, output_size):
        super(Q_Network, self).__init__(
            fc1 = L.Linear(input_size, hidden_size),
            fc2 = L.Linear(hidden_size, hidden_size),
            fc3 = L.Linear(hidden_size, hidden_size//2),
            fc4 = L.Linear(hidden_size, hidden_size//2),
            state_value = L.Linear(hidden_size//2, 1),
            advantage_value = L.Linear(hidden_size//2, output_size)
        )
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

    def __call__(self, x):
        h = F.relu(self.fc1(x))
        h = F.relu(self.fc2(h))
        hs = F.relu(self.fc3(h))
        ha = F.relu(self.fc4(h))
        state_value = self.state_value(hs)
        advantage_value = self.advantage_value(ha)
        advantage_mean = (F.sum(advantage_value, axis=1)/float(self.output_size)).reshape(-1, 1)
        q_value = F.concat([state_value for _ in range(self.output_size)], axis=1) + (advantage_value - F.concat([a
        return q_value

    def reset(self):
        self.zerograds()
""" >>> """

Q = Q_Network(input_size=env.history_t+1, hidden_size=100, output_size=3)
```

```
Q_ast = copy.deepcopy(Q)
optimizer = chainer.optimizers.Adam()
optimizer.setup(Q)

epoch_num = 50
step_max = len(env.data)-1
memory_size = 200
batch_size = 50
epsilon = 1.0
epsilon_decrease = 1e-3
epsilon_min = 0.1
start_reduce_epsilon = 200
train_freq = 10
update_q_freq = 20
gamma = 0.97
show_log_freq = 5

memory = []
total_step = 0
total_rewards = []
total_losses = []

start = time.time()
for epoch in range(epoch_num):

    pobs = env.reset()
    step = 0
    done = False
    total_reward = 0
    total_loss = 0

    while not done and step < step_max:

        # select act
        pact = np.random.randint(3)
        if np.random.rand() > epsilon:
            pact = Q(np.array(pobs, dtype=np.float32).reshape(1, -1))
            pact = np.argmax(pact.data)

        # act
        obs, reward, done = env.step(pact)

        # add memory
        memory.append((pobs, pact, reward, obs, done))
        if len(memory) > memory_size:
```

```
                memory.pop(0)

            # train or update q
            if len(memory) == memory_size:
                if total_step % train_freq == 0:
                    shuffled_memory = np.random.permutation(memory)
                    memory_idx = range(len(shuffled_memory))
                    for i in memory_idx[::batch_size]:
                        batch = np.array(shuffled_memory[i:i+batch_size])
                        b_pobs = np.array(batch[:, 0].tolist(), dtype=np.float32).reshape(batch_size, -1)
                        b_pact = np.array(batch[:, 1].tolist(), dtype=np.int32)
                        b_reward = np.array(batch[:, 2].tolist(), dtype=np.int32)
                        b_obs = np.array(batch[:, 3].tolist(), dtype=np.float32).reshape(batch_size, -1)
                        b_done = np.array(batch[:, 4].tolist(), dtype=np.bool)

                        q = Q(b_pobs)
                        """ <<< DQN -> Double DQN
                        maxq = np.max(Q_ast(b_obs).data, axis=1)
                        === """
                        indices = np.argmax(q.data, axis=1)
                        maxqs = Q_ast(b_obs).data
                        """ >>> """
                        target = copy.deepcopy(q.data)
                        for j in range(batch_size):
                            """ <<< DQN -> Double DQN
                            target[j, b_pact[j]] = b_reward[j]+gamma*maxq[j]*(not b_done[j])
                            === """
                            target[j, b_pact[j]] = b_reward[j]+gamma*maxqs[j, indices[j]]*(not b_done[j])
                            """ >>> """
                        Q.reset()
                        loss = F.mean_squared_error(q, target)
                        total_loss += loss.data
                        loss.backward()
                        optimizer.update()

                if total_step % update_q_freq == 0:
                    Q_ast = copy.deepcopy(Q)

            # epsilon
            if epsilon > epsilon_min and total_step > start_reduce_epsilon:
                epsilon -= epsilon_decrease

            # next step
            total_reward += reward
            pobs = obs
```

```
            step += 1
            total_step += 1

        total_rewards.append(total_reward)
        total_losses.append(total_loss)

        if (epoch+1) % show_log_freq == 0:
            log_reward = sum(total_rewards[((epoch+1)-show_log_freq):])/show_log_freq
            log_loss = sum(total_losses[((epoch+1)-show_log_freq):])/show_log_freq
            elapsed_time = time.time()-start
            print('\t'.join(map(str, [epoch+1, epsilon, total_step, log_reward, log_loss, elapsed_time])))
            start = time.time()

    return Q, total_losses, total_rewards
```

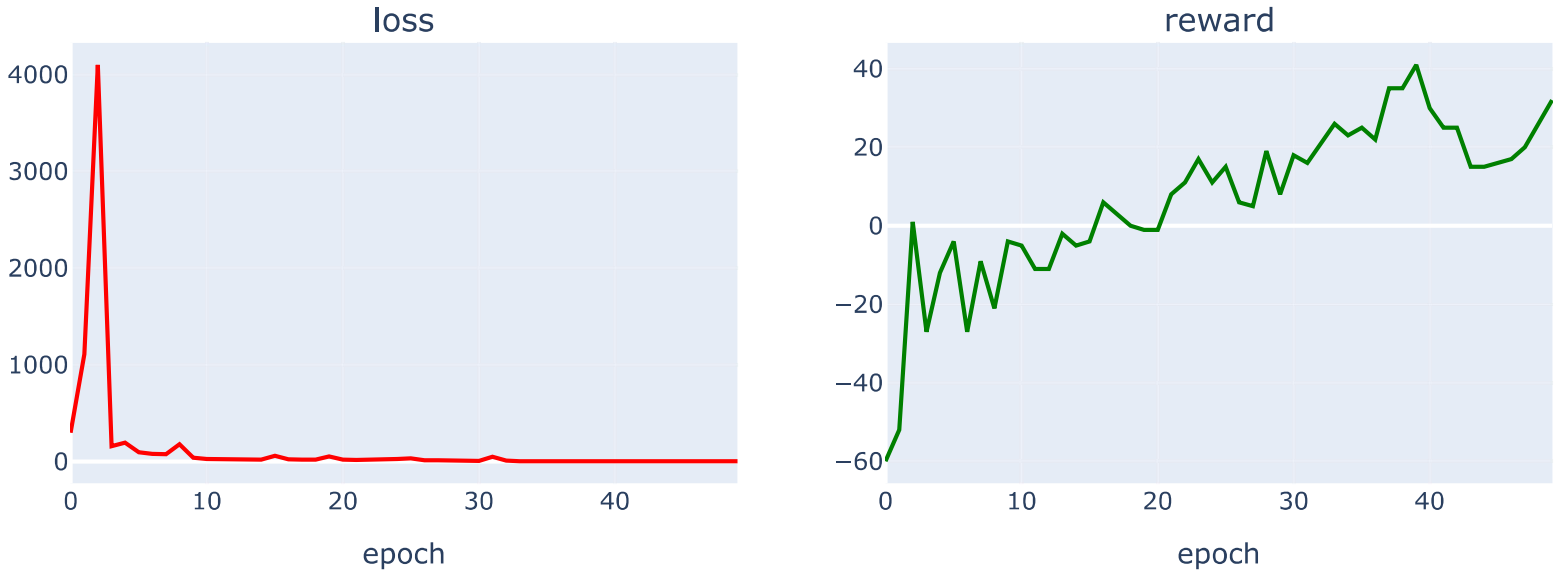
In [33]:
```
Q, total_losses, total_rewards = train_dddqn(Environment1(train))
```

```
C:\Users\Kunj\AppData\Local\Temp\ipykernel_12028\3141201869.py:105: VisibleDeprecationWarning:

Creating an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with differen
t lengths or shapes) is deprecated. If you meant to do this, you must specify 'dtype=object' when creating the ndarray.

C:\Users\Kunj\AppData\Local\Temp\ipykernel_12028\3141201869.py:113: DeprecationWarning:

`np.bool` is a deprecated alias for the builtin `bool`. To silence this warning, use `bool` by itself. Doing this will
not modify any behavior and is safe. If you specifically wanted the numpy scalar type, use `np.bool_` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecatio
ns
```

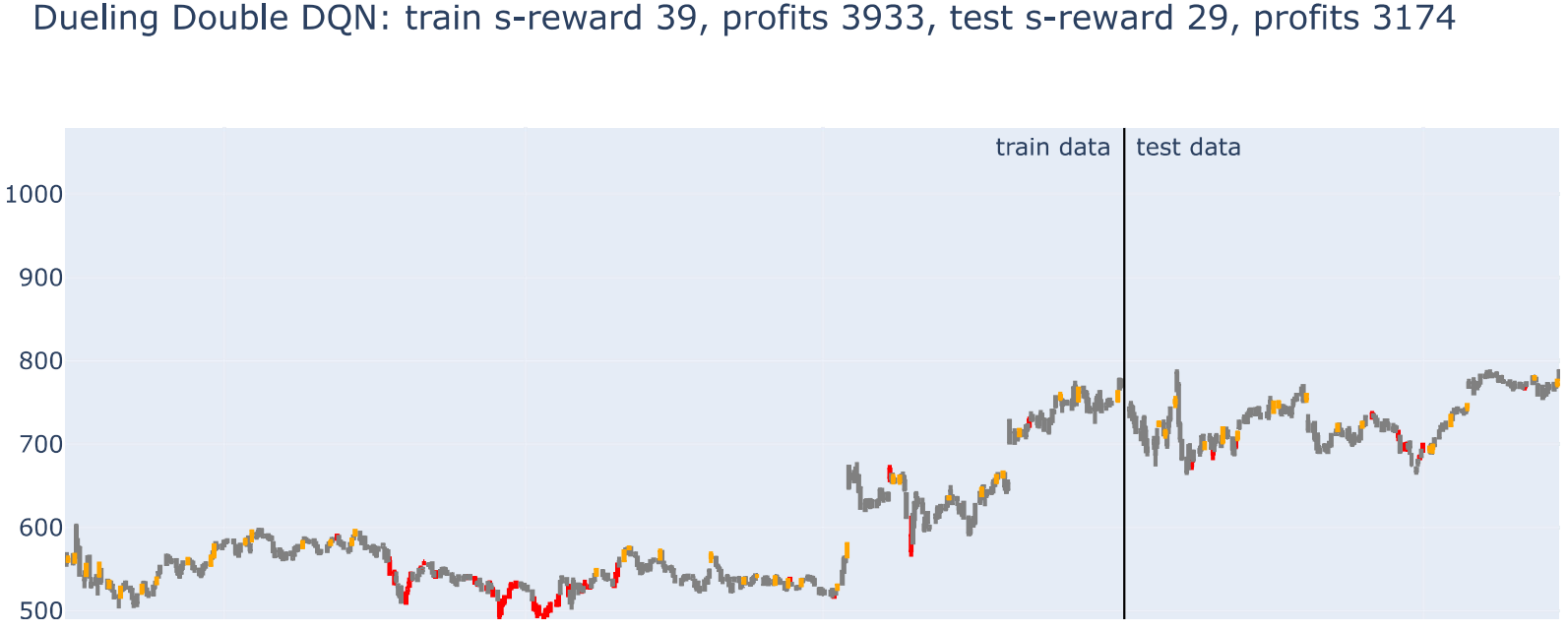
```
5       0.0999999999999992      2225    -30.0   1172.7404321491717      26.393882751464844
10      0.0999999999999992      4450    -13.0   95.50292947739362       33.74014139175415
15      0.0999999999999992      6675    -6.8    22.93389447629452       32.063750982284546
20      0.0999999999999992      8900    0.8     36.25348339863122       31.71899962425232
25      0.0999999999999992      11125   9.2     19.396415938064457      30.803929090499878
30      0.0999999999999992      13350   10.6    16.31309010721743       36.33586144447327
35      0.0999999999999992      15575   20.8    15.54907183703035       34.439157247543335
40      0.0999999999999992      17800   31.6    4.514340191800147       34.297162771224976
45      0.0999999999999992      20025   22.0    4.032627181801945       34.32814359664917
50      0.0999999999999992      22250   22.2    3.768142062984407       33.31204795837402
```

In [34]:
```
plot_loss_reward(total_losses, total_rewards)
```

```
C:\Users\Kunj\anaconda3\lib\site-packages\plotly\tools.py:461: DeprecationWarning:

plotly.tools.make_subplots is deprecated, please use plotly.subplots.make_subplots instead
```



In [35]:
```
plot_train_test_by_q(Environment1(train), Environment1(test), Q, 'Dueling Double DQN')
```


Dueling Double DQN: train s-reward 39, profits 3933, test s-reward 29, profits 3174
train data
test data
1000
900
800
700
600
500

In [ ]:

```python
import gymnasium as gym
from ray.rllib.algorithms.dqn import DQNConfig
import matplotlib.pyplot as plt
from gymnasium.utils import EzPickle
import numpy as np
from gymnasium.spaces import Box
from gymnasium.envs.classic_control import PendulumEnv


import gymnasium as gym
from gymnasium.spaces import Box, Discrete
import numpy as np
import seaborn as sns

class SimpleCorridor(gym.Env):
    """Example of a custom env in which you have to walk down a corridor.
    You can configure the length of the corridor via the env config."""

    def __init__(self, config=None):
        config = config or {}
        self.end_pos = config.get("corridor_length", 10)
        self.cur_pos = 0
        self.action_space = Discrete(2)
        self.observation_space = Box(0.0, 999.0, shape=(1,), dtype=np.float32)

    def set_corridor_length(self, length):
        self.end_pos = length
        print("Updated corridor length to {}".format(length))

    def reset(self, *, seed=None, options=None):
        self.cur_pos = 0.0
        return [self.cur_pos], {}

    def step(self, action):
        assert action in [0, 1], action
        if action == 0 and self.cur_pos > 0:
            self.cur_pos -= 1.0
```

```
    def step(self, action):
        assert action in [0, 1], action
        if action == 0 and self.cur_pos > 0:
            self.cur_pos -= 1.0
        elif action == 1:
            self.cur_pos += 1.0
        done = truncated = self.cur_pos >= self.end_pos
        reward = -0.1  # negative reward for each action taken
        if done:
            reward += 1  # bonus reward for reaching the end of the corridor
        return [self.cur_pos], reward, done, truncated, {}

config = (
    DQNConfig()
    .environment(
        env=SimpleCorridor,
    )
    .rollouts(num_rollout_workers=3)
)
algo = config.build()
average_rewards = []

for i in range(100):
    results = algo.train()
    average_rewards.append(results['episode_reward_mean'])
    print(f"Iter: {i}; avg. reward={results['episode_reward_mean']}")

sns.set_style("darkgrid")
sns.set_palette("husl")

# Plot the average rewards
plt.figure(figsize=(10,6))
plt.plot(average_rewards)
plt.xlabel('Number of Iteration')
plt.ylabel('Iterated average Reward')
plt.title('Algorithm Performance')
plt.show()
```

```python
import gymnasium as gym
from ray.rllib.algorithms.ppo import PPOConfig
from ray.rllib.algorithms.dqn import DQNConfig
import matplotlib.pyplot as plt
from gymnasium.utils import EzPickle
import numpy as np
from gymnasium.spaces import Box
from gymnasium.envs.classic_control import PendulumEnv


import gymnasium as gym
from gymnasium.spaces import Box, Discrete
import numpy as np
import seaborn as sns

class SimpleCorridor(gym.Env):
    """Example of a custom env in which you have to walk down a corridor.
    You can configure the length of the corridor via the env config."""

    def __init__(self, config=None):
        config = config or {}
        self.end_pos = config.get("corridor_length", 10)
        self.cur_pos = 0
        self.action_space = Discrete(2)
        self.observation_space = Box(0.0, 999.0, shape=(1,), dtype=np.float32)

    def set_corridor_length(self, length):
        self.end_pos = length
        print("Updated corridor length to {}".format(length))

    def reset(self, *, seed=None, options=None):
        self.cur_pos = 0.0
        return [self.cur_pos], {}


    def step(self, action):
        assert action in [0, 1], action
```

```
    def step(self, action):
        assert action in [0, 1], action
        if action == 0 and self.cur_pos > 0:
            self.cur_pos -= 1.0
        elif action == 1:
            self.cur_pos += 1.0
        done = truncated = self.cur_pos >= self.end_pos
        reward = -0.1  # negative reward for each action taken
        if done:
            reward += 1  # bonus reward for reaching the end of the corridor
        return [self.cur_pos], reward, done, truncated, {}

config = (
    PPOConfig()
    .environment(
        env=SimpleCorridor,
    )
    .rollouts(num_rollout_workers=3)
)
algo = config.build()
average_rewards = []

for i in range(100):
    results = algo.train()
    average_rewards.append(results['episode_reward_mean'])
    print(f"Iter: {i}; avg. reward={results['episode_reward_mean']}")


sns.set_style("darkgrid")
sns.set_palette("husl")

# Plot the average rewards
plt.figure(figsize=(10,6))
plt.plot(average_rewards)
plt.xlabel('Number of Iteration')
plt.ylabel('Iterated average Reward')
plt.title('Algorithm Performance')
plt.show()
```