Learning Based Control HW4: Learning-Based Control - Gym

Submitted By: Ashutosh Gupta
OSU ID: 934533517

I worked on this assignment on my own. All the work below along with the attached code is my own work. To clarify some doubts I worked with Joshua Cook.

Contents

1 Q-Learning

1.1 Problem details

Q-learning is a Reinforcement Learning (RL) value based policy that learns Q-values of the state-action pairs (s,a) using current state, current action, reward, next state and maximum of Q-value for next state. So Q-learning uses (s,a,r,s',max_a) tuples to learn in an off-policy setting for a discrete environment. The update rule is described in Equation 1. Where α is the learning rate and γ is the discount factor.

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha * [r(s_t,a_t) + \gamma * max_a Q(s_{t+1},a) - Q(s_t,a_t)] \tag{1}$$

Q-learning works well only in environments that have finite discrete observation and action space. So for the implementing Q-learning on the Cartpole-v1 gym environment that has continuous observation space, we will need to discretize the space. So we bin the continuous observation space of the cart pole environment in fixed number of bins and then digitize the observation when learning. This enables us to have finite size of the Q-table and implement a basic Q-learning on the problem. The cart pole has an observation space of 4 variables, namely cart position, cart velocity, pole angle, pole angular velocity. We dicretize each of the observation into number of bins η_{bins}. For the velocity theoretical observation range is $(-inf,inf)$, but the agent never does reach very high velocities so we practically clip the velocities observation space to $[-5.0,5.0]$. The action space of the environment is discrete defined as $[0,1]$, where 0 means push left and 1 means push right.

1.2 Implementation

In our implementation for the cart pole problem, we zero initialize the Q-table. The policy chooses an action from a list of actions by the ε-greedy algorithm. Policy samples a random action with probability of ε or chooses the action with the highest Q-value at the current state. The exploration factor (ε) is decayed at every episode by a decay factor (β), until a minimum (ε_{min}). Each episode is run for a given maximum time steps or until the episode is either terminated or truncated. The agent receives a termination when pole angle or cart position is out of safety limits defined by gym. The agent receives a truncation if it is able to balance the pole for 500 time steps. Below are the parameters for the implementation

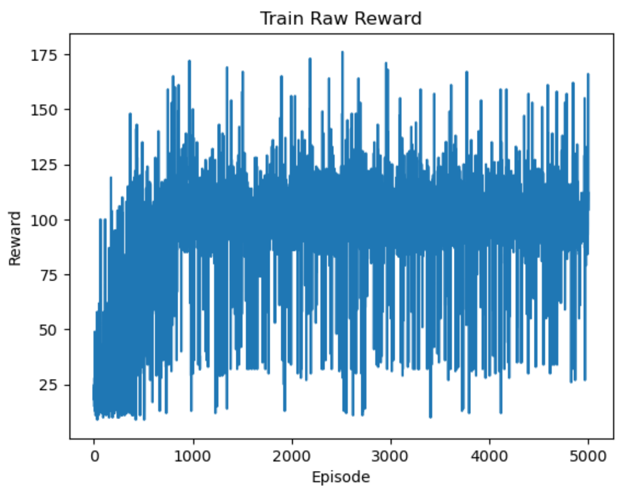
- Episodes: $e = 5000$
- Max Time steps: $t = 500$
- Number of bins: $\eta_{bins} = 6$
- Learning rate: $\alpha = 0.001$
- Discount factor: $\gamma = 0.9$
- Exploration rate: $\varepsilon = 1.0$
- Exploration Decay rate: $\beta = 0.995$
- Minimum Exploration rate: $\varepsilon_{min} = 0.01$

After training we also test our Q-table by running the algorithm with $\varepsilon, \alpha = 0$ for 100 testing episodes.

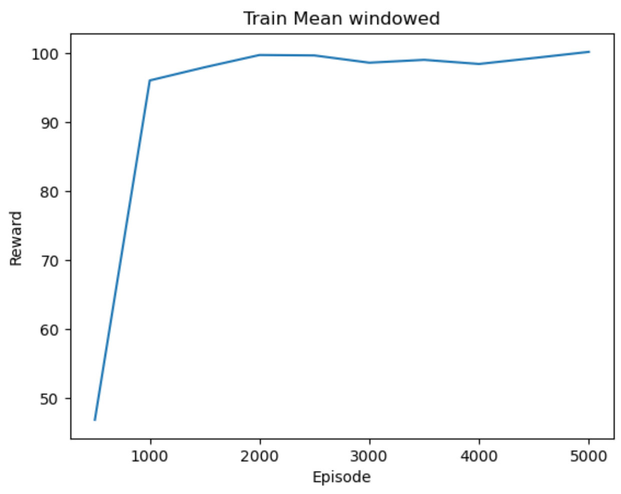
1.3 Results

The learning curve - rewards against episodes is plotted in Figure 1a. A windowed mean reward over the episodes is shown in Figure 1b, the rewards are averaged at every 100 episodes. Rolling window average reward is shown in Figure 1c, the rewards are averaged in a rolling window that increases by 100 episodes. Similar plots for the testing run are shown in Figure 2a, Figure 2b and Figure 2c. Note that it seems the testing mean reward goes down over episodes but it is a small decrease if you notice the y-limits of the plot.

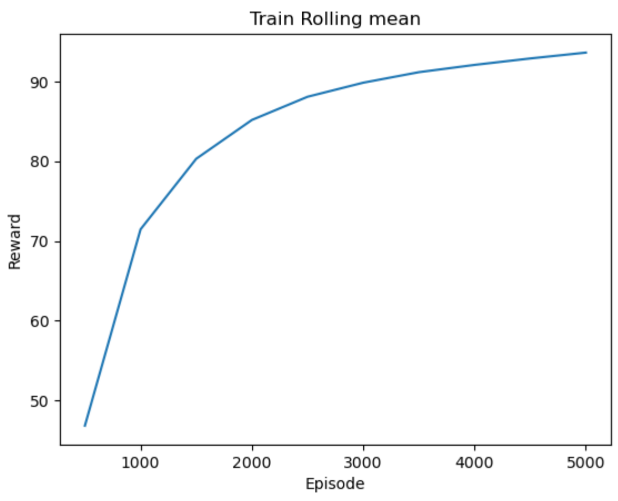
The learning does converge to good enough policy but it takes really long. The Q-learning implementation is also very sensitive and requires extensive fine tuning of parameters.



(a) Reward vs Episodes for Q-learning



(b) Windowed mean reward



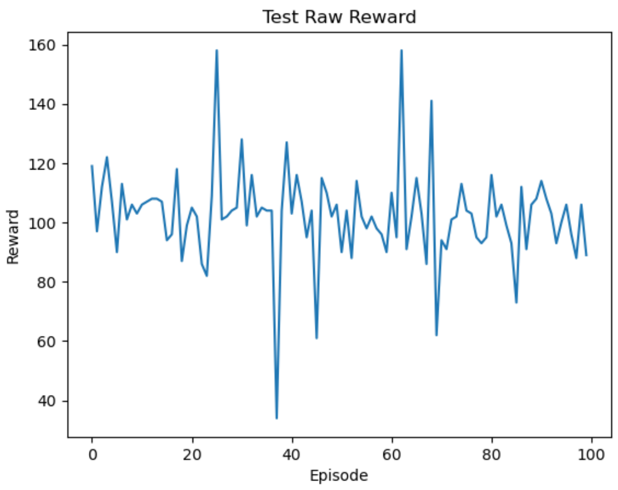
(c) Rolling window mean reward

Figure 1: Training curve for Q-learning

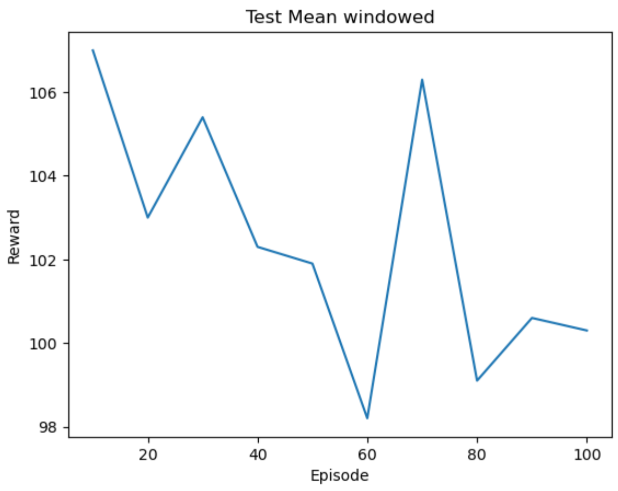
2 Evolutionary Neural Network

2.1 Problem details

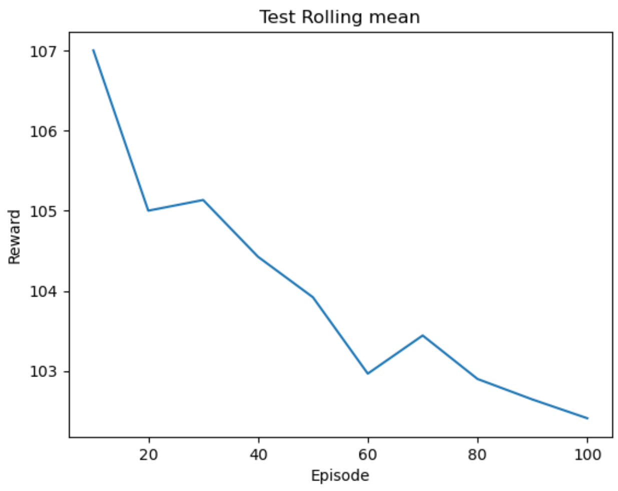
Now for the same cart pole gym environment, we implement an evolutionary algorithm to evolve a neural network. We implement the evolutionary algorithm to learn the weights and biases of a simple one-layer neural network. The single layer neural network maps the observations to actions with just one linear layer followed by softmax probability computation. The input to the network are the 4 observations and the output of the network is the probability of taking either of the two actions. The action with highest probability is



(a) Reward vs Episodes for Q-learning



(b) Windowed mean reward



(c) Rolling window mean reward

Figure 2: Testing curve for Q-learning

chosen as output. So in this problem, the number of weights are $(4*2=8)$ and number biases are just 2, making it in total just 10 network parameters for the algorithm to evolve. The network equation is shown in Equation 2, with dimensions of the matrix denoted in subscripts.

$$Y_{2x1} = W_{2x4} * X_{2x1} + B_{2x1} \tag{2}$$

$$W = \begin{bmatrix} w_{ij} \end{bmatrix}_{2x4} \tag{3}$$

$$B = \begin{bmatrix} b_0 & b_1 \end{bmatrix}_{2x1} \tag{4}$$

We evaluate each network for one episode to get the total reward of the network over the episode. Each episode is run for a given maximum time steps or until the episode is either terminated or truncated. The agent receives a termination when pole angle or cart position is out of safety limits defined by gym. The agent receives a truncation if it is able to balance the pole for 500 time steps. We also give a high positive reward of $+100$ if the agent is truncated, meaning that it balanced the pole for 500 time steps. So the maximum possible reward of a network is 600.

2.2 Implementation

In our implementation of evolutionary algorithm to evolve a neural network, we start with an initial population (P_t) and evaluate each network. Population is initialized by random uniform sampling the parameters of each network $[W,B] \sim U(-1,1)$. We then select a percentage of top performing networks from P_t and generate a mutated population (M_t) by our mutation operator. The mutation operator multiplies each network parameter by a normal gaussian noise, so new parameters are given by Equation 5.

$$[W',B'] = [W,B] * \mathcal{N}(\mu,\sigma) \tag{5}$$

We then get a combined population $(C_t = P_t + M_t)$, and we select the top performing networks for the next generation (P_{t+1}). The parameters used for the implementation are listed below

- Generations: $e = 1000$
- Maximum Time steps: $t = 500$
- Population size: $p = 100$
- Mutation percent: $m = 0.25$, basically 25 agents
- Mean for Gaussian: $\mu = 1.0$
- Standard deviation for Gaussian: $\sigma = 0.001$

2.3 Results

The reward of every agent in every generation (episodes) is plotted in Figure 3a and reward for the best agent in each generation is shown in Figure 3b. The mean and standard deviation of reward over all agents in the population at each generation is shown in Figure 3c and Figure 3d. As it is seen the algorithm is able to converge to the best possible agent really well, as discussed earlier the maximum reward possible for the agent is 600.

Since here we are learning a linear mapping of continuous observations to actions, sort of implementing a linear controller very similar to PID, the network implementation performs much better than Q-learning. For the cart pole problem a linear controller is more than sufficient to learn to control the pole.

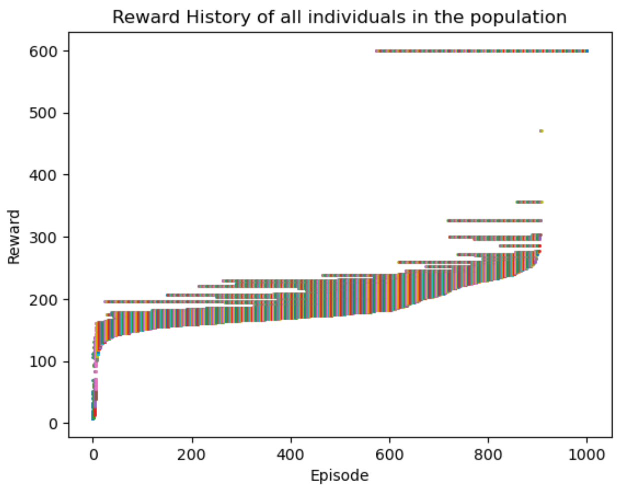
3 Extra Credit: Deep Q-Network

3.1 Problem details

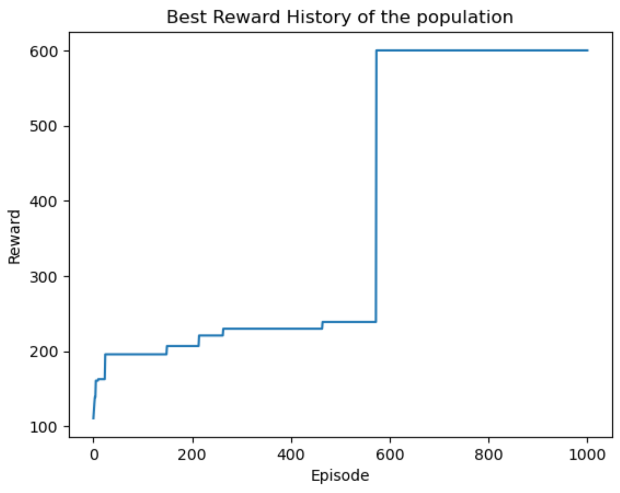
We implement a Deep Q-Network learning on the cart pole gym environment. Here we learn a network that maps the continuous observation to values foe each of the discrete action. We select the action with highest value from the model prediction. During weight update, since we don't have ground truth values we use temporal difference learning to set the target value for the prediction and then compute the gradients. We will use just the reward value when there was a truncation. Algorithm 1 explains the target value selection.

3.2 Implementation

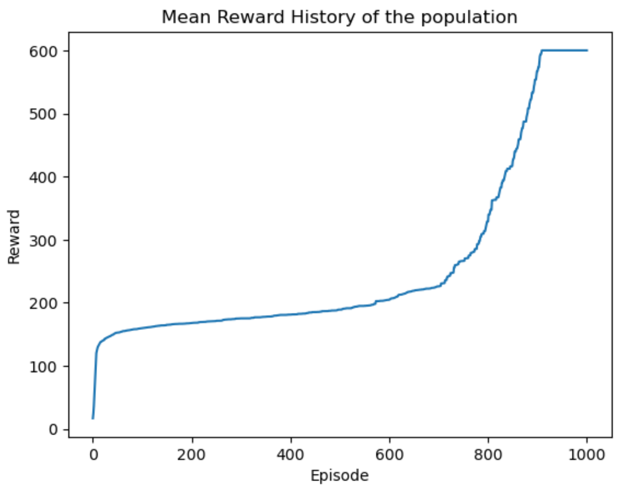
In our implementation for the Deep Q-Network learning for the cart pole problem, we learn a network with one linear hidden layer of size h. The loss function for the network is mean squared error and we use Adam optimizer to update weights. Each episode is run for a given maximum time steps or until the episode is either terminated or truncated. The agent receives a termination when pole angle or cart position is out of safety



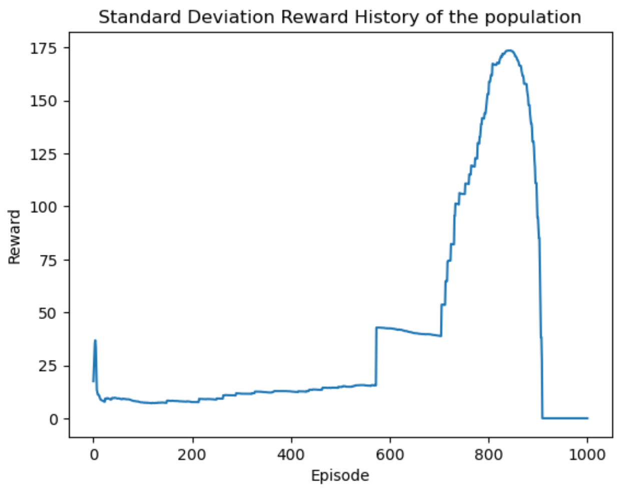
(a) Reward for All agents



(b) Best Reward of each generation



(c) Mean reward of each generation



(d) Standard deviation of reward in each generation

Figure 3: Evolutionary neural network training curves

limits defined by gym. The agent receives a truncation if it is able to balance the pole for 500 time steps. At every time step we store the environment transitions in a replay buffer. After every episode we sample a batch (of size s) from this replay buffer to be used for weight update.

Below are the parameters for the implementation

- Episodes: $e = 1000$
- Max Time steps: $t = 500$
- Sample size: $s = 32$
- Hidden layer size: $h = 8$
- Learning rate: $\alpha = 0.01$

Algorithm 1 Deep Q-Network

```
  if $truncated$ then
      $target = r_t$
  else
      $target = r_t + \gamma * max(NN(s_{t+1}))$
  end if
```

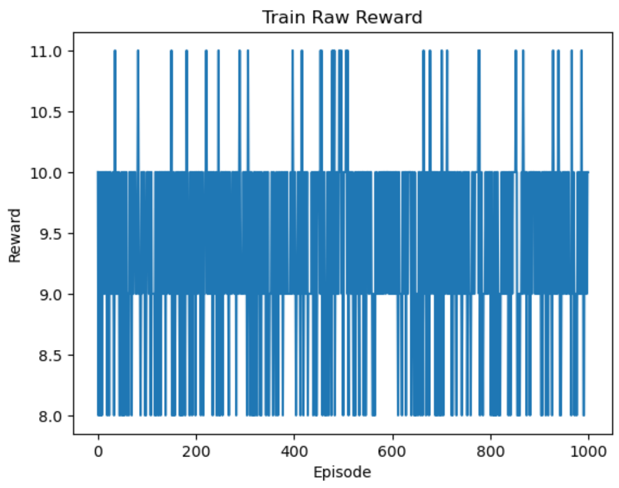
- Discount factor: $\gamma = 0.9$

After training we also test our Network by running the algorithm with $\alpha = 0$ for 100 testing episodes.

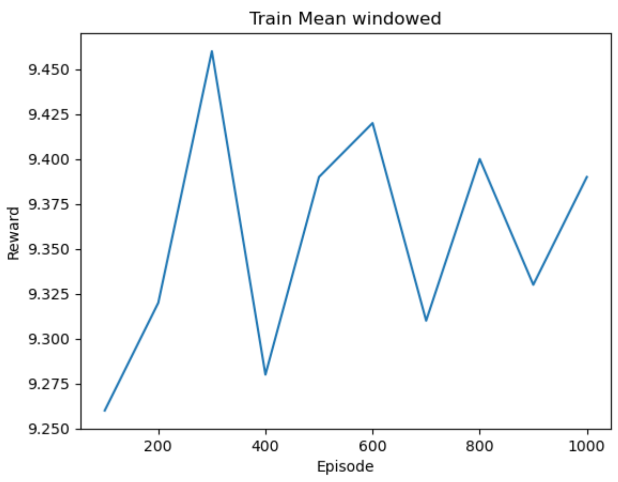
3.3 Results

The learning curve - rewards against episodes is plotted in Figure 4a. A windowed mean reward over the episodes is shown in Figure 4b, the rewards are averaged at every 100 episodes. Rolling window average reward is shown in Figure 4c, the rewards are averaged in a rolling window that increases by 100 episodes. Similar plots for the testing run are shown in Figure 5a, Figure 5b and Figure 5c. Note that it seems the testing mean reward goes down over episodes but it is a small decrease if you notice the y-limits of the plot.

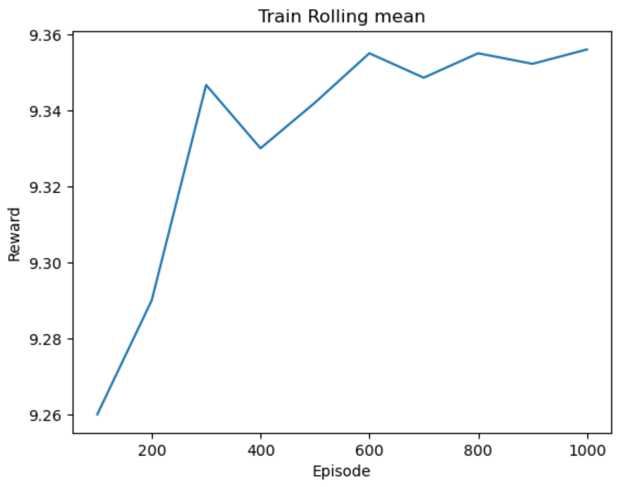
The learning seems to converge to local minimum, with not much improvement to the rewards. This could a consequence of lack of exploration caused by a simple feed forward network. The evolutionary neural network worked the best as it encouraged more exploration of parameters and so it was able to learn the linear mapping much better. In this case even with a hidden layer, by simple gradient update there isn't much exploration for network to learn.



(a) Reward vs Episodes for Deep Q-Network

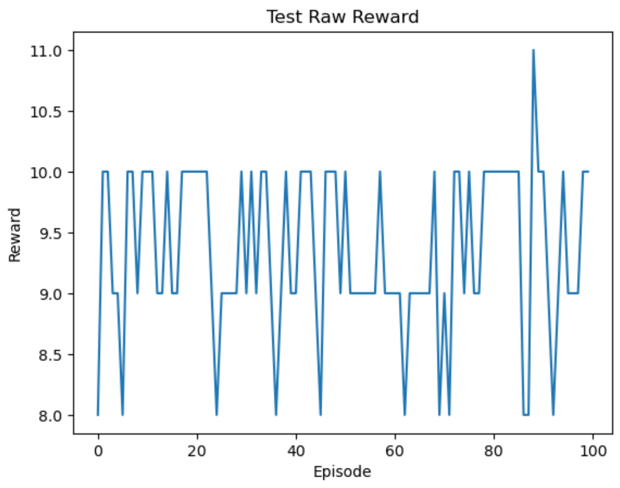


(b) Windowed mean reward

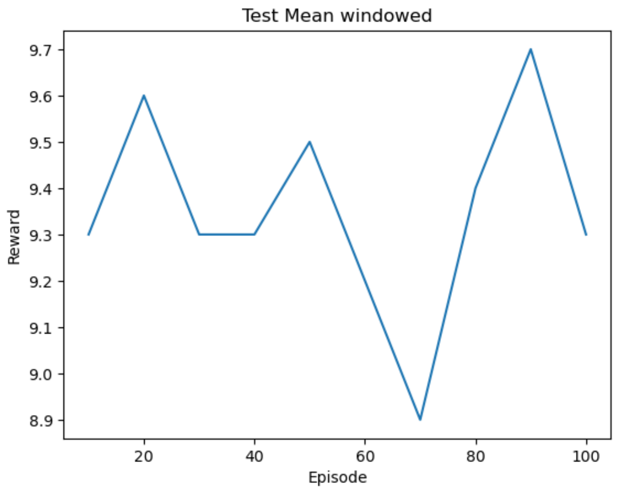


(c) Rolling window mean reward

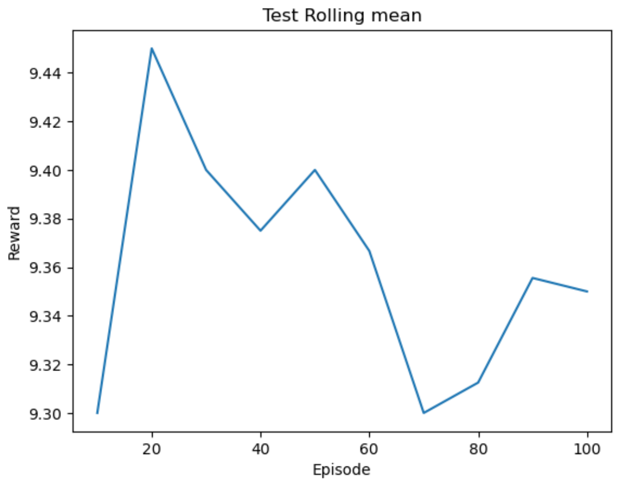
Figure 4: Training curve for Depp Q-Network



(a) Reward vs Episodes for Deep Q-Network



(b) Windowed mean reward



(c) Rolling window mean reward

Figure 5: Testing curve for Deep Q-Network

A Learning algorithm code

A.1 Q-Learning

```
import numpy as np
import gymnasium as gym
import matplotlib.pyplot as plt


class QAgent():
    """Q-learning agent for Gymnasium environments. Discretizes the continuous
    ↪  state space.

    Args:
        env: Gymnasium environment. Must have a discrete action space.
        episodes: Number of episodes to train. Defaults to 1000.
        max_time_steps: Number of time steps per episode. Defaults to 500.
        test_episodes: Number of episodes to test. Defaults to 100.
        num_bins: Number of bins to discretize the state space. Defaults to 10.
```

```python
        mean_reward_window: Window size for the rolling mean. Defaults to 10.
        alpha: Learning rate. Defaults to 0.1.
        epsilon: Exploration rate. Defaults to 1.0.
        epsilon_decay: Decay rate for epsilon. Defaults to 0.99.
        epsilon_min: Minimum value for epsilon. Defaults to 0.01.
        gamma: Discount factor. Defaults to 0.9.
    """

    def __init__(self, env, episodes:int=1000, max_time_steps:int=500,
     ↪  test_episodes:int=100, num_bins:int=10,
                 mean_reward_window:int=10, alpha:float=0.1, epsilon:float=1.0,
                  ↪  epsilon_decay:float=0.99, epsilon_min:float=0.01,
                  ↪  gamma:float=0.9):
        """Initializes the Q-Learning agent."""

        # Check if the environment has a discrete action space.
        assert isinstance(env.action_space, gym.spaces.Discrete), "Environment has
         ↪  no discrete action space."
        # Check if the environment has a continuous observation space.
        assert isinstance(env.observation_space, gym.spaces.Box), "Environment has
         ↪  no continuous observation space."

        # Define the agent's attributes here.
        self.env = env
        self.episodes = episodes
        self.max_time_steps = max_time_steps
        self.test_episodes = test_episodes
        self.num_bins = num_bins
        self.mean_reward_window = mean_reward_window
        self.alpha = alpha
        self.epsilon = epsilon
        self.epsilon_decay = epsilon_decay
        self.epsilon_min = epsilon_min
        self.gamma = gamma
        self.training = True

        # Extract the number of actions and observations from the environment.
        self.num_actions = env.action_space.n
        self.num_observations = env.observation_space.shape[0]

        # Exrtact the observation space bounds from the environment.
        self.observation_space_low = env.observation_space.low
        self.observation_space_high = env.observation_space.high

        # If the bounds are not finite, approximate only the infinite ones.
        # In gymnasium, infinite bounds are represented as large floats and not
        ↪   np.inf.
        # This is why we check for values larger than 1e3. And approximate them
        ↪   with -5.0 and 5.0.
        self.observation_space_low[self.observation_space_low < -1e3] = -5.0
        self.observation_space_high[self.observation_space_high > 1e3] = 5.0

        # Initialize the bins for each observation dimension.
        self.bins = np.linspace(self.observation_space_low,
        ↪   self.observation_space_high, self.num_bins).T

        # Initialize the Q-table with zeros.
        self.Q = np.zeros(([self.num_bins] * self.num_observations) +
        ↪   [self.num_actions])

        # Initialize the reward history.
        self.reward_train = []
        self.reward_test = []
        self.reward_train_rolling = [] # Rolling mean over every 1/10 of the train
        ↪   episodes.
        self.reward_test_rolling = [] # Rolling mean over every 1/10 of the test
        ↪   episodes.
        self.reward_train_mean = [] # Mean over every 1/10 of the train episodes.
        self.reward_test_mean = [] # Mean over every 1/10 of the test episodes.

        self.train_x_axis = []
        self.test_x_axis = []

    def set_training(self, training:bool=True):
        """Sets the training flag."""

        self.training = training

        # If training false, set epsilon and alpha to 0.0.
        if not training:
            self.epsilon = 0.0
            self.alpha = 0.0

    def get_discrete_observation(self, observation):
        """Discretizes the continuous observation space.

        Args:
            observation: Observation from the environment.

        Returns:
            Discretized observation index.
        """

        # Initialize the discrete observation.
        discrete_observation = np.zeros(self.num_observations, dtype=np.int64)

        # Digitize each observation dimension.
        for i in range(self.num_observations):
            discrete_observation[i] = np.digitize(observation[i], self.bins[i]) - 1

        return tuple(discrete_observation)

    def get_action(self, observation):
        """Returns the action to take given the observation. Implements
        ↪  epsilon-greedy policy with Q-table.

        Args:
            observation: Observation from the environment.

        Returns:
            Action to take.
        """

        # Get the discrete observation.
        sd = self.get_discrete_observation(observation)

        # With probability epsilon, take a random action.
        if np.random.rand() < self.epsilon:
            action = self.env.action_space.sample()
        # Otherwise, take the best action.
        else:
            action = np.argmax(self.Q[sd])

        return action

    def update(self, observation, action, reward, next_observation):
        """Updates the Q-table."""

        # Get the discrete observations.
        sd = self.get_discrete_observation(observation)
        sd_prime = self.get_discrete_observation(next_observation)

        # Get the Q-value for the current state and action.
        Q_sa = self.Q[sd][action]

        # Get the maximum Q-value for the next state.
        max_Q_s_prime = np.max(self.Q[sd_prime])
        # Update the Q-value for the current state and action.
        Q_sa = Q_sa + self.alpha * (reward + self.gamma * max_Q_s_prime - Q_sa)

        # Update the Q-table.
        self.Q[sd][action] = Q_sa

    def run(self, verbose:bool=False):
        """Train and test the Q-Learning agent.

        Args:
            verbose (bool, optional): whether to print progress. Defaults to
             ↪  False.
        """

        for episode in range(self.episodes + self.test_episodes):

            # Reset the environment.
            observation,_ = self.env.reset()
            episode_reward = 0

            # Run the episode for max_time_steps or until the episode is done.
            for _ in range(self.max_time_steps):

                # Get the action to take.
                action = self.get_action(observation)

                # Take the action and get the next observation and reward.
                next_observation, reward, terminated, truncated,_ =
                 ↪  self.env.step(action)

                # Update the Q-table.
                self.update(observation, action, reward, next_observation)

                # Update episode reward.
                episode_reward += reward

                # Update the observation.
                observation = next_observation

                # If terminated, break the loop.
                if terminated:
                    break

                # If truncated, break the loop and give positive reward.
                if truncated:
```

```
            episode_reward += 100.0
            break

    # Update the reward history.
    if self.training:
        self.reward_train.append(episode_reward)
    else:
        self.reward_test.append(episode_reward)

    # Decay epsilon.
    if self.epsilon > self.epsilon_min:
        self.epsilon *= self.epsilon_decay
    else:
        self.epsilon = self.epsilon_min

    # If verbose, print progress for training episodes.
    if verbose and (episode % (self.episodes // 10) == 0 or episode ==
     ↪  self.episodes - 1) and self.training:
        print(f"Episode: {episode+1}/{self.episodes} | Mean Reward until
         ↪  now: {np.mean(self.reward_train):.2f}")

    # Set the training flag to false after training.
    if episode == self.episodes - 1:
        self.set_training(False)
        if verbose and self.test_episodes > 0:
            print("Testing...")

    # Print progress for testing episodes at the end.
    if verbose and episode == self.episodes + self.test_episodes - 1 and
     ↪  self.test_episodes > 0:
        print(f"Mean Reward in Test: {np.mean(self.reward_test):.2f}")

# Compute the rolling mean and mean over every mean reward window of the
 ↪  episodes for training episodes.
for i in range(0, self.episodes, self.episodes // self.mean_reward_window):
    self.reward_train_rolling.append(np.mean(self.reward_train[0:i +
     ↪  self.episodes // self.mean_reward_window]))
    self.reward_train_mean.append(np.mean(self.reward_train[i:i +
     ↪  self.episodes // self.mean_reward_window]))
    self.train_x_axis.append(i + self.episodes // self.mean_reward_window)

# Compute the rolling mean and mean over every mean reward window of the
 ↪  episodes for testing episodes.
for i in range(0, self.test_episodes, self.test_episodes //
 ↪  self.mean_reward_window):
```

```
            self.reward_test_rolling.append(np.mean(self.reward_test[0:i +
             ↪   self.test_episodes // self.mean_reward_window]))
            self.reward_test_mean.append(np.mean(self.reward_test[i:i +
             ↪   self.test_episodes // self.mean_reward_window]))
            self.test_x_axis.append(i + self.test_episodes //
             ↪   self.mean_reward_window)

    def plot_results(self):
        """Plots the reward history."""

        # Training rewards
        plt.figure()
        plt.plot(self.reward_train)
        plt.title("Train Raw Reward")
        plt.xlabel("Episode")
        plt.ylabel("Reward")

        plt.figure()
        plt.plot(self.train_x_axis, self.reward_train_mean)
        plt.title("Train Mean windowed")
        plt.xlabel("Episode")
        plt.ylabel("Reward")

        plt.figure()
        plt.plot(self.train_x_axis, self.reward_train_rolling)
        plt.title("Train Rolling mean")
        plt.xlabel("Episode")
        plt.ylabel("Reward")

        # Testing rewards
        plt.figure()
        plt.plot(self.reward_test)
        plt.title("Test Raw Reward")
        plt.xlabel("Episode")
        plt.ylabel("Reward")

        plt.figure()
        plt.plot(self.test_x_axis, self.reward_test_mean)
        plt.title("Test Mean windowed")
        plt.xlabel("Episode")
        plt.ylabel("Reward")

        plt.figure()
        plt.plot(self.test_x_axis, self.reward_test_rolling)
        plt.title("Test Rolling mean")
        plt.xlabel("Episode")
    plt.ylabel("Reward")
```

A.2 Evolutionary Neural Network

```
import numpy as np
import gymnasium as gym
import matplotlib.pyplot as plt

# NN related imports
import torch


class EvolveNNAgent():
    """Evolutionary genetic algorithm Neural Network agent for Gymnasium
    ↪ environments.

    Args:
        env: Gymnasium environment. Must have a discrete action space.
        episodes: Number of episodes to train. Defaults to 1000.
        max_time_steps: Maximum number of time steps per episode. Defaults to 500.
        population_size: Number of individuals in the population. Defaults to 100.
        mutation_percent: Percentage of individuals to mutate. Defaults to 0.5.
        mean: Mean of the normal distribution to sample the noise for mutation.
        ↪ Defaults to 1.0.
        std: Standard deviation of the normal distribution to sample the noise for
        ↪ mutation. Defaults to 0.001.
    """

    def __init__(self, env, episodes:int=1000, max_time_steps:int=500,
    ↪ population_size:int=100,
                 mutation_percent:float=0.5, mean:float=1.0, std:float=0.001):
        """Initializes the Evolutionary genetic algorithm Neural Network agent."""

        # Check if the environment has a discrete action space.
        assert isinstance(env.action_space, gym.spaces.Discrete), "Environment has
        ↪ no discrete action space."
        # Check if the environment has a continuous observation space.
        assert isinstance(env.observation_space, gym.spaces.Box), "Environment has
        ↪ no continuous observation space."

        # Define the agent's attributes here.
        self.env = env
        self.episodes = episodes
        self.max_time_steps = max_time_steps
        self.population_size = population_size
        self.mutation_percent = mutation_percent
        self.mean = mean
        self.std = std

        # Extract the number of actions and observations from the environment.
        self.num_actions = env.action_space.n
        self.num_observations = env.observation_space.shape[0]

        # Number of parameters in the neural network.
        self.num_weights = self.num_observations * self.num_actions
        self.num_biases = self.num_actions
        self.num_params = self.num_weights + self.num_biases

        # Initialize the reward history.
        self.reward_history = np.zeros((self.episodes + 1, self.population_size))

        # Initalize the population.
        self.population = None

    def model_init(self, params:np.ndarray):
        """Initializes the neural network model.

        Args:
            params: Parameters of the neural network model. Shape: (num_params,).

        Returns:
            model: Neural network PyTorch model.
        """

        # Define the model architecture
        # Model has a linear layer with input size of num_observations and output
        ↪   size of num_actions
        # Basically linearly maps the observation to the action
        # There is a softmax layer at the end to convert the output to a
        ↪   probability distribution
        model = torch.nn.Sequential(
            torch.nn.Linear(self.num_observations, self.num_actions, bias=True),
            torch.nn.Softmax(dim=0)
        )

        # Extract the weights and biases from the parameters.
        weights = params[:self.num_weights].reshape(self.num_actions,
        ↪   self.num_observations)
        biases = params[self.num_weights:].reshape(self.num_actions)

        # Set the weights and biases of the model.
        model[0].weight.data = torch.from_numpy(weights).float()
        model[0].bias.data = torch.from_numpy(biases).float()
        return model

    def population_init(self):
        """Initializes the initial neural network model population using random
         ↪  parameters.

        Returns:
            population: List of neural network PyTorch models.
        """

        # Initialize the population.
        population = []

        # Iterate over all individuals in the population.
        for i in range(self.population_size):
            # Initialize the individual's parameters with random values.
            individual_params = np.random.uniform(-1.0, 1.0, self.num_params)

            # Initialize the neural network model with the current individual's
             ↪  parameters.
            model = self.model_init(individual_params)

            # Add the model to the population.
            population.append(model)

        return population

    def get_action(self, observation, model):
        """Returns the action to take given an observation. Implements
         ↪  epsilon-greedy exploration with model prediction.

        Args:
            observation: Observation from the environment.
            model: Neural network PyTorch model.

        Returns:
            action: Action to take.
        """

        # Convert the observation to a PyTorch tensor.
        observation = torch.from_numpy(observation).float()

        # Get the action with the highest probability from the model.
        action = torch.argmax(model(observation)).item()
        return action

    def get_model_reward(self, model):
        """Returns the reward of the model over one episode.

        Args:
            model: Neural network PyTorch model.

        Returns:
            reward: Reward for the model over one episode.
        """

        # Reset the environment.
        observation,_ = self.env.reset()
        episode_reward = 0.0

        # Run the episode for a maximum of max_time_steps or until the episode is
        ↪   done.
        for _ in range(self.max_time_steps):

            # Get the action to take.
            action = self.get_action(observation, model)

            # Take the action.
            next_observation, reward, terminated, truncated,_ =
            ↪   self.env.step(action)

            # Update the total reward.
            episode_reward += reward

            # Update the observation.
            observation = next_observation

            # If terminated, break the loop.
            if terminated:
                break

            # If truncated, break the loop and give positive reward.
            if truncated:
                episode_reward += 100.0
                break

        return episode_reward

    def get_population_reward(self, population):
        """Returns the reward of the population of models over one episode.
```

```python
        Args:
            population: List of neural network PyTorch models.

        Returns:
            rewards: Numpy array of rewards for the population over one episode.
        """

        # Initialize the rewards.
        rewards = np.zeros(len(population))

        # Iterate over all individuals in the population.
        for i in range(len(population)):
            # Get the reward of the current individual.
            rewards[i] = self.get_model_reward(population[i])

        return rewards

    def mutate_model(self, model):
        """Mutates the model by adding random noise to its parameters.

        Args:
            model: Neural network PyTorch model.

        Returns:
            model_mutated: Mutated neural network PyTorch model.
        """

        # Extract the model parameters.
        weights = model[0].weight.data.numpy().reshape(self.num_weights)
        biases = model[0].bias.data.numpy().reshape(self.num_biases)

        # Add random noise to the parameters.
        weights *= np.random.normal(self.mean, self.std, self.num_weights)
        biases *= np.random.normal(self.mean, self.std, self.num_biases)

        # Concatenate the weights and biases.
        params = np.concatenate((weights, biases))

        # Initialize the mutated model.
        model_mutated = self.model_init(params)

        return model_mutated

    def run(self, verbose:bool=False):
        """Train the Evolutionary genetic algorithm Neural Network agent.
```

```
Args:
    verbose (bool, optional): whether to print progress. Defaults to
    ↪   False.
"""

# Initialize the population.
self.population = self.population_init()

# Get the reward of the population.
rewards = self.get_population_reward(self.population)

# Store the initial reward history.
self.reward_history[0] = rewards

# Iterate over all episodes.
for episode in range(1, self.episodes + 1):

    # Get the indices of the top individuals in the population.
    top_indices = np.argsort(rewards)[-int(self.mutation_percent *
    ↪   self.population_size):]

    # Get the top individuals in the population.
    top_individuals = [self.population[i] for i in top_indices]

    # Mutate the top individuals.
    mutated_individuals = [self.mutate_model(model) for model in
    ↪   top_individuals]

    # Get rewards of the mutated individuals.
    mutated_rewards = self.get_population_reward(mutated_individuals)

    # Get buffer total population and mutated individuals.
    population_buffer = self.population + mutated_individuals # list of
    ↪   models
    rewards_buffer = np.concatenate((rewards, mutated_rewards)) # numpy
    ↪   array of rewards

    # Select the next generation.
    next_generation_indices =
    ↪   np.argsort(rewards_buffer)[-self.population_size:]
    self.population = [population_buffer[i] for i in
    ↪   next_generation_indices] # list of models
    rewards = rewards_buffer[next_generation_indices] # numpy array of
    ↪   rewards
```

```
            # Store the reward history.
            self.reward_history[episode] = rewards

            # If verbose, print progress.
            if verbose and (episode % (self.episodes // 10) == 0 or episode ==
             ↪  self.episodes):
                print(f"Episode: {episode}/{self.episodes} | Mean Reward for
                 ↪  episode: {np.mean(rewards)}")

    def plot_results(self):
        """Plots the reward history."""

        # Plot all the reward in each episode.
        plt.figure("Reward History All Individuals")
        for i in range(len(self.reward_history)):
            plt.scatter(np.repeat(i, len(self.reward_history[i])),
             ↪  self.reward_history[i], s=1)
        plt.xlabel("Episode")
        plt.ylabel("Reward")
        plt.title("Reward History of all individuals in the population")

        # Plot the mean reward in each episode.
        plt.figure("Reward History Mean")
        plt.plot(np.mean(self.reward_history, axis=1))
        plt.xlabel("Episode")
        plt.ylabel("Reward")
        plt.title("Mean Reward History of the population")

        # Plot the best reward in each episode.
        plt.figure("Reward History Best")
        plt.plot(np.max(self.reward_history, axis=1))
        plt.xlabel("Episode")
        plt.ylabel("Reward")
        plt.title("Best Reward History of the population")

        # Plot the standard deviation of the reward in each episode.
        plt.figure("Reward History Std")
        plt.plot(np.std(self.reward_history, axis=1))
        plt.xlabel("Episode")
        plt.ylabel("Reward")
        plt.title("Standard Deviation Reward History of the population")
```

A.3 Deep Q-Network

```
import os
import random
import numpy as np
import gymnasium as gym
import matplotlib.pyplot as plt

# NN related imports
import collections
import torch


class DeepQAgent():
    """Deep Q-learning agent for Gymnasium environments.

    Args:
        env: Gymnasium environment. Must have a discrete action space.
        model_weights: Path to the model weights file. If file does not exist, it
        ↪  will be created. Extensions supported: .pt
        use_prev: Use previous model weights. Defaults to False.
        episodes: Number of episodes to train. Defaults to 1000.
        test_episodes: Number of episodes to test. Defaults to 100.
        max_time_steps: Number of time steps per episode. Defaults to 500.
        sample_size: Sample batch size for replay memory. Defaults to 32.
        hidden_size: Number of neurons in the hidden layer. Defaults to 24.
        mean_reward_window: Window size for computing the mean reward. Defaults to
        ↪  10.
        alpha: Learning rate. Defaults to 0.001.
        gamma: Discount factor. Defaults to 0.9.
    """

    def __init__(self, env, model_weights:str, use_prev:bool=False,
    ↪  episodes:int=1000, test_episodes:int=100, max_time_steps:int=500,
                sample_size:int=32, hidden_size:int=24, mean_reward_window:int=10,
                ↪  alpha:float=0.001, gamma:float=0.9):
        """Initializes the Deep Q-Learning agent."""

        # Check if the environment has a discrete action space.
        assert isinstance(env.action_space, gym.spaces.Discrete), "Environment has
        ↪  no discrete action space."
        # Check if the environment has a continuous observation space.
        assert isinstance(env.observation_space, gym.spaces.Box), "Environment has
        ↪  no continuous observation space."

        # Define the agent's attributes here.
        self.env = env
        self.model_weights = model_weights
        self.use_prev = use_prev
        self.episodes = episodes
        self.test_episodes = test_episodes
        self.max_time_steps = max_time_steps
        self.sample_size = sample_size
        self.hidden_size = hidden_size
        self.mean_reward_window = mean_reward_window
        self.alpha = alpha
        self.gamma = gamma
        self.training = True

        # Memory for experience replay.
        self.memory = collections.deque(maxlen=2000)

        # Extract the number of actions and observations from the environment.
        self.num_actions = env.action_space.n
        self.num_observations = env.observation_space.shape[0]

        # Initialize the neural network model.
        self.model = None
        self.model_loss = None
        self.optimizer = None
        self.model_init()

        # Initialize the reward history.
        self.reward_train = []
        self.reward_test = []
        self.reward_train_rolling = [] # Rolling mean over every 1/10 of the train
         ↪  episodes.
        self.reward_test_rolling = [] # Rolling mean over every 1/10 of the test
         ↪  episodes.
        self.reward_train_mean = [] # Mean over every 1/10 of the train episodes.
        self.reward_test_mean = [] # Mean over every 1/10 of the test episodes.

        self.train_x_axis = []
        self.test_x_axis = []

    def set_training(self, training:bool=True):
        """Sets the training flag for the agent."""

        self.training = training

        # If training false, set alpha to 0.0.
        if not training:
            self.alpha = 0.0

    def model_init(self):
        """Initializes the neural network model."""
        ## Define the model architecture here.
        # The model should take the observation as input and output the Q-values
         ↪  for each action.
        # The model has 1 hidden layer and output layer with linear activation.
        self.model = torch.nn.Sequential(
            torch.nn.Linear(self.num_observations, self.hidden_size),
            torch.nn.Linear(self.hidden_size, self.num_actions),
        )

        ## Define the loss function and optimizer.
        # The loss function is mean squared error.
        self.model_loss = torch.nn.MSELoss()
        # The optimizer is Adam with learning rate alpha.
        self.optimizer = torch.optim.Adam(self.model.parameters(), lr=self.alpha)

        # Load previous model weights if specified and file exists.
        if self.use_prev and os.path.isfile(self.model_weights):
            self.model.load_state_dict(torch.load(self.model_weights))

    def get_action(self, observation):
        """Returns the action to take given an observation using model prediction.

        Args:
            observation: Observation from the environment.

        Returns:
            action: Action to take.
        """

        # Convert the observation to torch tensor.
        observation = torch.from_numpy(observation).float()

        # Get the model prediction for the observation.
        action = torch.argmax(self.model(observation)).item()

        return action

    def learn(self):
        """Replay buffer from experience replay for a batch of samples to update
         ↪  the model weights."""

        # Sample a batch of samples from the memory.
        minibatch = random.sample(list(self.memory), min(len(self.memory),
         ↪  self.sample_size))

        # Iterate through the batch of samples.
        for s, a, r, s_prime, _, truncated in minibatch:

            # Convert the state and next state to torch tensors.
            s = torch.from_numpy(s).float()
            s_prime = torch.from_numpy(s_prime).float()

            # Get the target for the current state and action from the model
             ↪  prediction for the next state.
            target = r if truncated else r + self.gamma *
             ↪  torch.max(self.model(s_prime))

            # Get the model prediction for the current state.
            y_pred = self.model(s)
            y_target = torch.zeros_like(y_pred)
            y_target[a] = target

            # Update the model weights.
            loss = self.model_loss(y_pred, y_target)
            self.optimizer.zero_grad()
            loss.backward()
            self.optimizer.step()

    def run(self, save_model:bool=False, verbose:bool=False):
        """Train the Deep Q-Learning agent.

        Args:
            save_model (bool, optional): whether to save the model weights.
             ↪  Defaults to False.
            verbose (bool, optional): whether to print progress. Defaults to
             ↪  False.
        """

        for episode in range(self.episodes + self.test_episodes):

            # Reset the environment.
            observation,_ = self.env.reset()
            episode_reward = 0

            # Run the episode for max_time_steps or until the episode is done.
            for _ in range(self.max_time_steps):

                # Get the action to take.
                action = self.get_action(observation)

                # Take the action and get the next state, reward, and flags.
                next_observation, reward, terminated, truncated,_ =
                 ↪  self.env.step(action)

                # Add the sample to the replay memory.
                self.memory.append((observation, action, reward, next_observation,
                 ↪  terminated, truncated))

                # Update the observation.
                observation = next_observation

                # Update episode reward.
                episode_reward += reward

                # If terminated, break the loop and give negative reward.
                if terminated:
                    break

                # If truncated, break the loop and give positive reward.
                if truncated:
                    episode_reward += 100.0
                    break

            # Update the model weights.
            if self.training:
                self.learn()

            # Update the reward history.
            if self.training:
                self.reward_train.append(episode_reward)
            else:
                self.reward_test.append(episode_reward)

            # If verbose, print progress for training episodes.
            if verbose and (episode % (self.episodes // 10) == 0 or episode ==
             ↪  self.episodes - 1) and self.training:
                print(f"Episode: {episode+1}/{self.episodes} | Mean Reward until
                 ↪  now: {np.mean(self.reward_train):.2f}")

            # Set the training flag to false after training.
            if episode == self.episodes - 1:
                self.set_training(False)
                if verbose and self.test_episodes > 0:
                    print("Testing...")

            # Print progress for testing episodes at the end.
```

```
            if verbose and episode == self.episodes + self.test_episodes - 1 and
             ↪  self.test_episodes > 0:
                print(f"Mean Reward in Test: {np.mean(self.reward_test):.2f}")

        # Compute the rolling mean and mean over every mean reward window of the
         ↪  episodes for training episodes.
        for i in range(0, self.episodes, self.episodes // self.mean_reward_window):
            self.reward_train_rolling.append(np.mean(self.reward_train[0:i +
             ↪  self.episodes // self.mean_reward_window]))
            self.reward_train_mean.append(np.mean(self.reward_train[i:i +
             ↪  self.episodes // self.mean_reward_window]))
            self.train_x_axis.append(i + self.episodes // self.mean_reward_window)

        # Compute the rolling mean and mean over every mean reward window of the
         ↪  episodes for testing episodes.
        for i in range(0, self.test_episodes, self.test_episodes //
         ↪  self.mean_reward_window):
            self.reward_test_rolling.append(np.mean(self.reward_test[0:i +
             ↪  self.test_episodes // self.mean_reward_window]))
            self.reward_test_mean.append(np.mean(self.reward_test[i:i +
             ↪  self.test_episodes // self.mean_reward_window]))
            self.test_x_axis.append(i + self.test_episodes //
             ↪  self.mean_reward_window)

        # Save the model weights.
        if save_model:
            torch.save(self.model.state_dict(), self.model_weights)

    def plot_results(self):
        """Plots the reward history."""

        # Training rewards
        plt.figure()
        plt.plot(self.reward_train)
        plt.title("Train Raw Reward")
        plt.xlabel("Episode")
        plt.ylabel("Reward")

        plt.figure()
        plt.plot(self.train_x_axis, self.reward_train_mean)
        plt.title("Train Mean windowed")
        plt.xlabel("Episode")
        plt.ylabel("Reward")

        plt.figure()
        plt.plot(self.train_x_axis, self.reward_train_rolling)
        plt.title("Train Rolling mean")
        plt.xlabel("Episode")
        plt.ylabel("Reward")

        # Testing rewards
        plt.figure()
        plt.plot(self.reward_test)
        plt.title("Test Raw Reward")
        plt.xlabel("Episode")
        plt.ylabel("Reward")

        plt.figure()
        plt.plot(self.test_x_axis, self.reward_test_mean)
        plt.title("Test Mean windowed")
        plt.xlabel("Episode")
        plt.ylabel("Reward")

        plt.figure()
        plt.plot(self.test_x_axis, self.reward_test_rolling)
        plt.title("Test Rolling mean")
        plt.xlabel("Episode")
        plt.ylabel("Reward")
```

B Experiments code

The pdf generated from the Jupyter notebook is attached below.

experiments

November 23, 2023

[1]:
```python
import matplotlib.pyplot as plt
import gymnasium as gym

from QAgent import QAgent
from EvolveNNAgent import EvolveNNAgent
from DeepQAgent import DeepQAgent
```

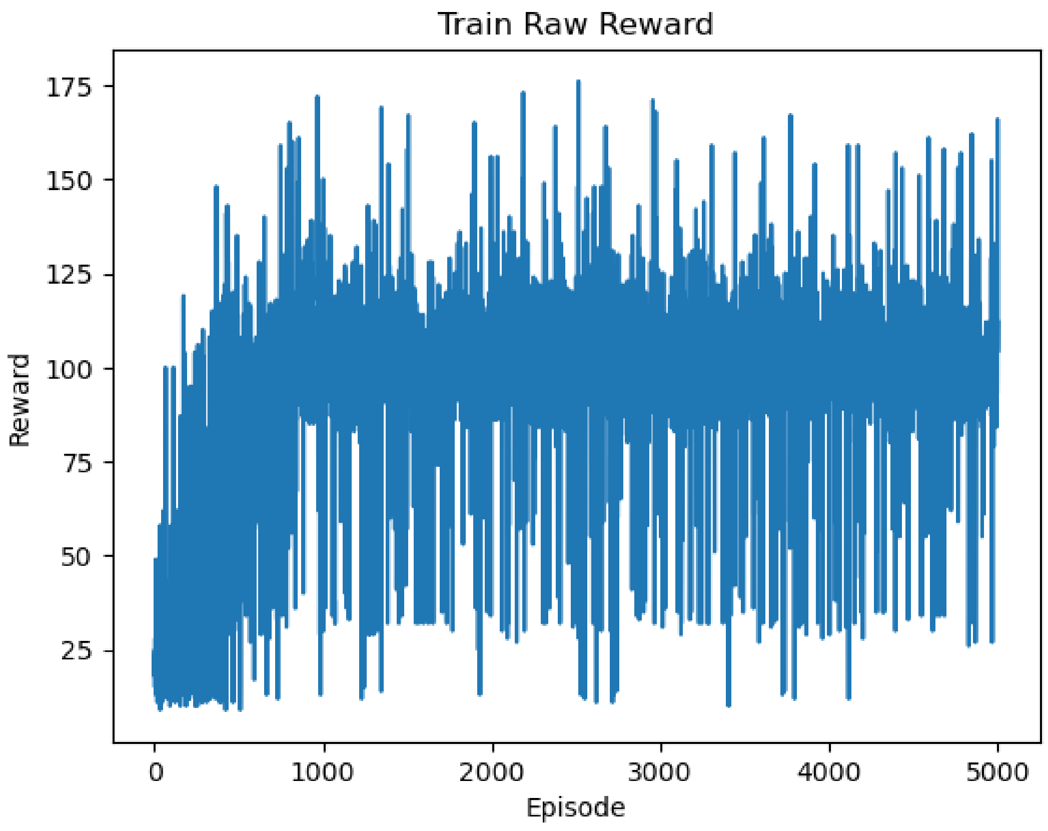
[2]:
```python
# Create the environment CartPole-v1
cartpole_env = gym.make('CartPole-v1')
```

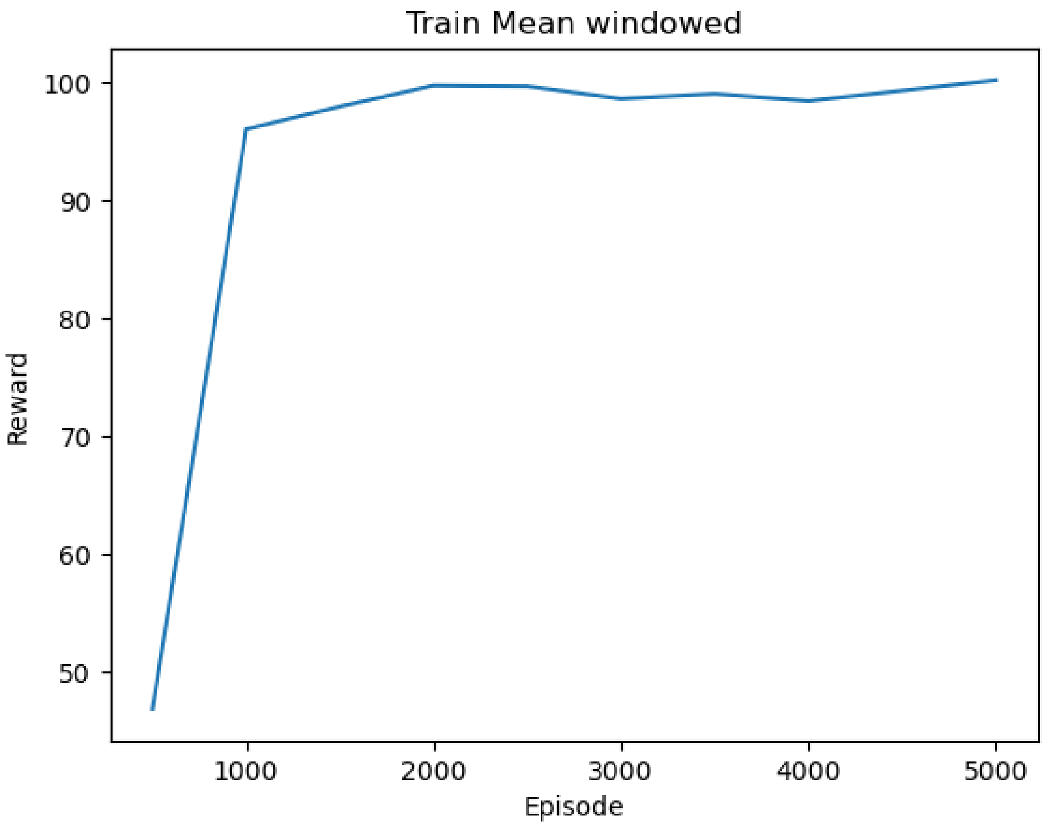
[4]:
```python
## Create the QAgent
# Parameters
episodes = 5000 # Number of episodes to train
max_time_steps = 500 # Maximum number of time steps per episode
test_episodes = 100 # Number of episodes to test the agent
num_bins = 6 # Number of bins to discretize the each observation dimension
mean_reward_window = 10 # Number of episodes to average the reward over␣
 ↪episodes/window size
alpha = 0.001 # Learning rate
epsilon = 1.0 # Initial exploration rate
epsilon_decay = 0.995 # Decay rate of the exploration rate
epsilon_min = 0.01 # Minimum exploration rate
gamma = 0.9 # Discount factor

# Create the agent
q_agent = QAgent(cartpole_env, episodes, max_time_steps, test_episodes,␣
 ↪num_bins, mean_reward_window, alpha, epsilon, epsilon_decay, epsilon_min,␣
 ↪gamma)

# Run the agent training
q_agent.run(verbose=True)
```

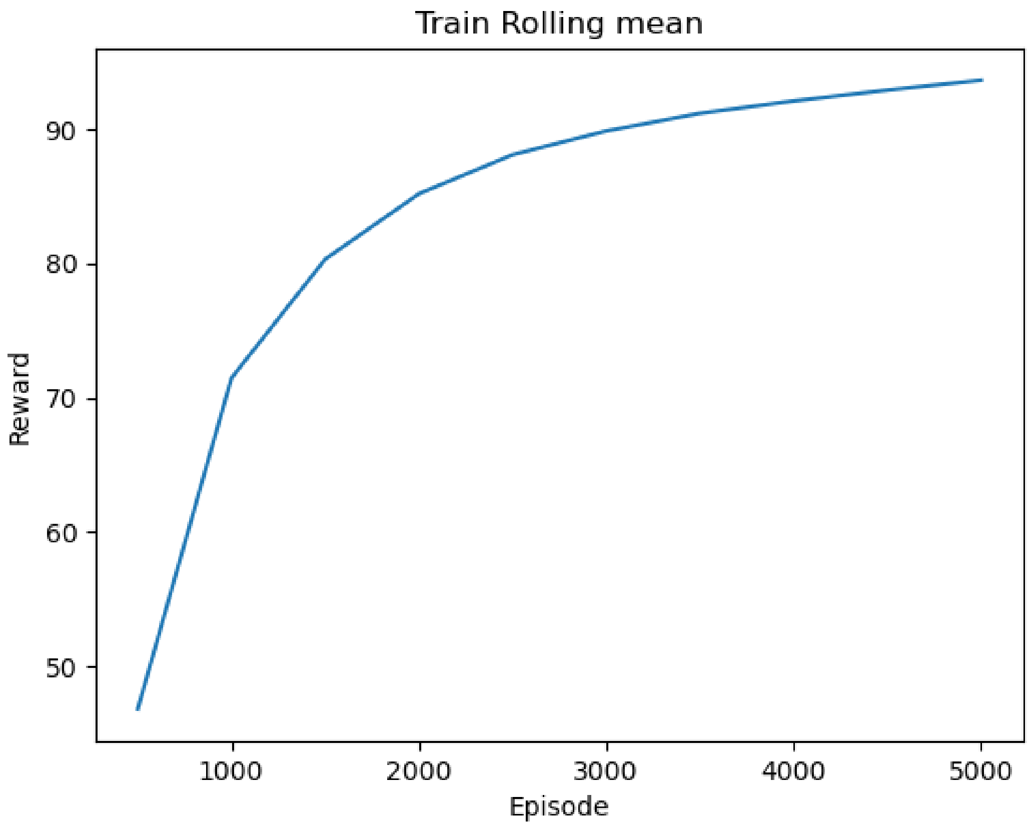
```
Episode: 1/5000 | Mean Reward until now: 23.00
Episode: 501/5000 | Mean Reward until now: 46.93
Episode: 1001/5000 | Mean Reward until now: 71.54
Episode: 1501/5000 | Mean Reward until now: 80.36
Episode: 2001/5000 | Mean Reward until now: 85.19
Episode: 2501/5000 | Mean Reward until now: 88.10
```

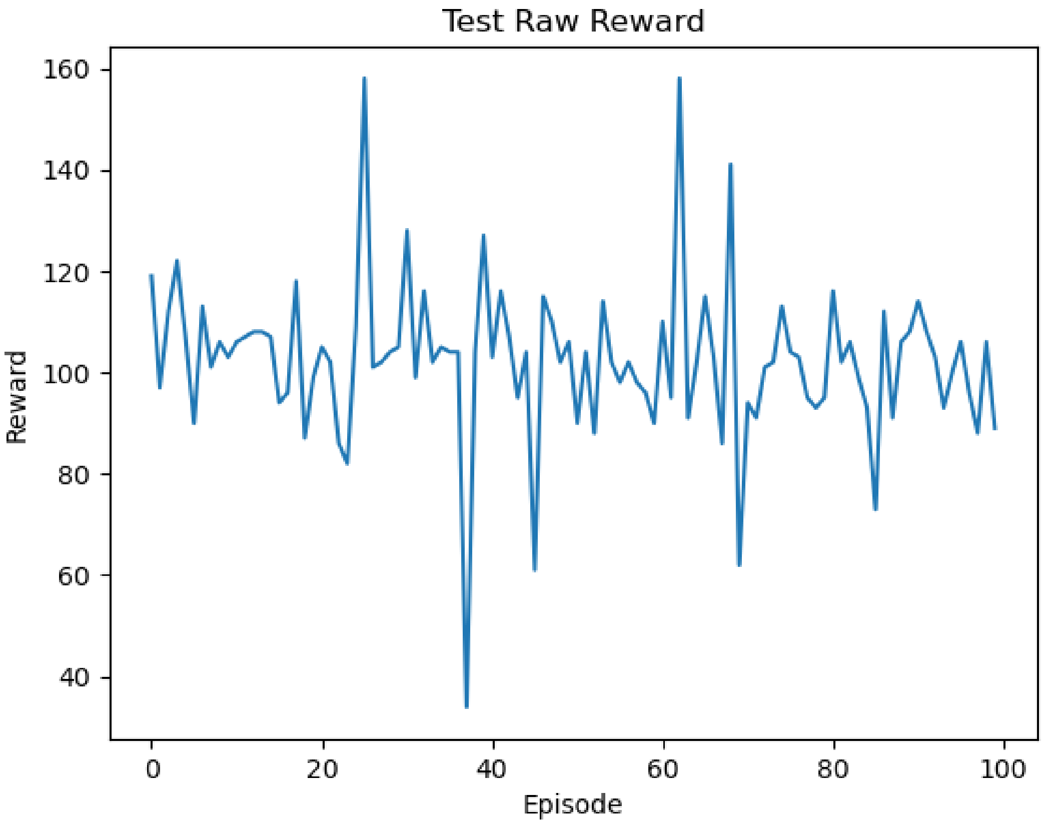
```
Episode: 3001/5000 | Mean Reward until now: 89.86
Episode: 3501/5000 | Mean Reward until now: 91.17
Episode: 4001/5000 | Mean Reward until now: 92.09
Episode: 4501/5000 | Mean Reward until now: 92.90
Episode: 5000/5000 | Mean Reward until now: 93.63
Testing…
Mean Reward in Test: 102.41
```

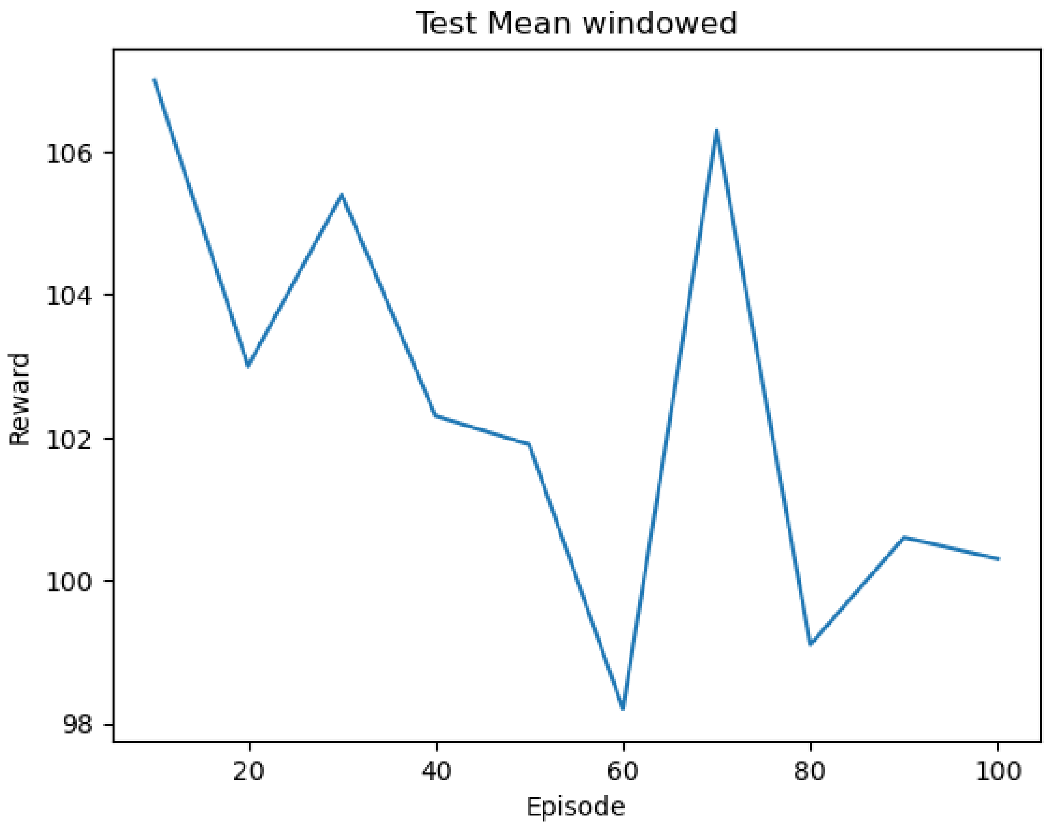
[5]:
```
# Plot the results
q_agent.plot_results()
plt.show()
```

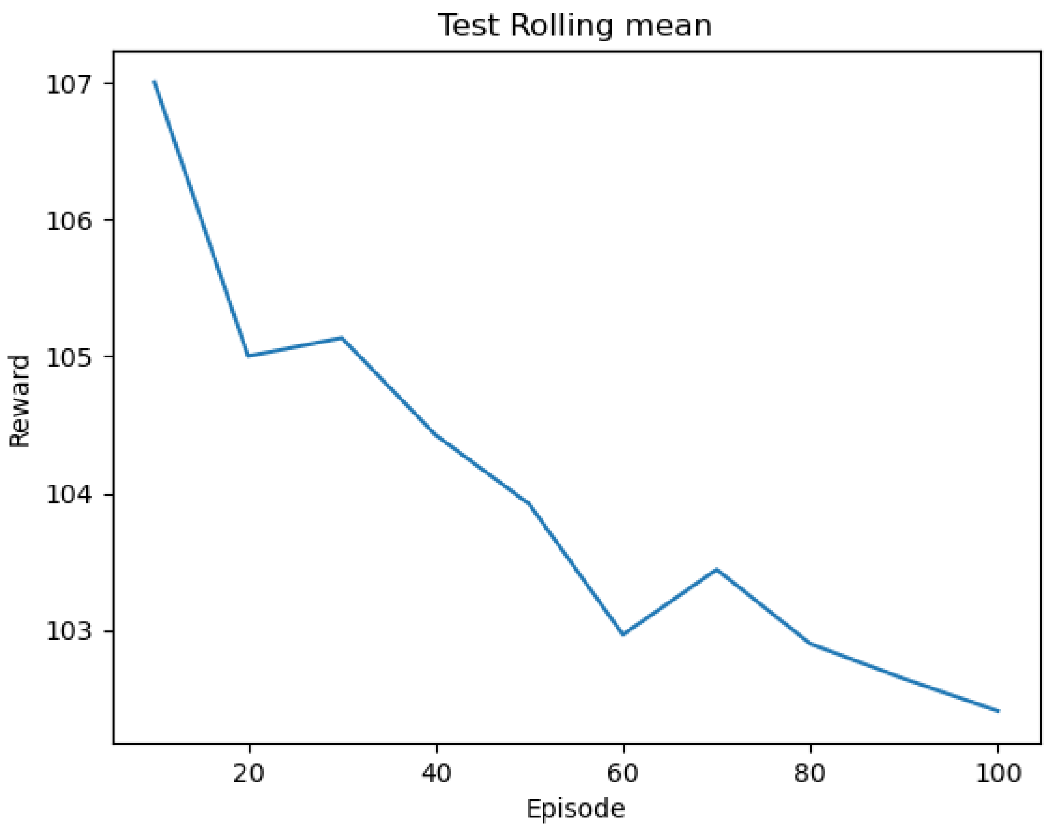



Train Mean windowed
Reward
Episode
100
90
80
70
60
50
1000
2000
3000
4000
5000


Train Rolling mean
Reward
Episode
90
80
70
60
50
1000
2000
3000
4000
5000


Test Raw Reward
Reward
Episode
160
140
120
100
80
60
40
0
20
40
60
80
100


Test Mean windowed
Reward
106
104
102
100
98
Episode
20
40
60
80
100



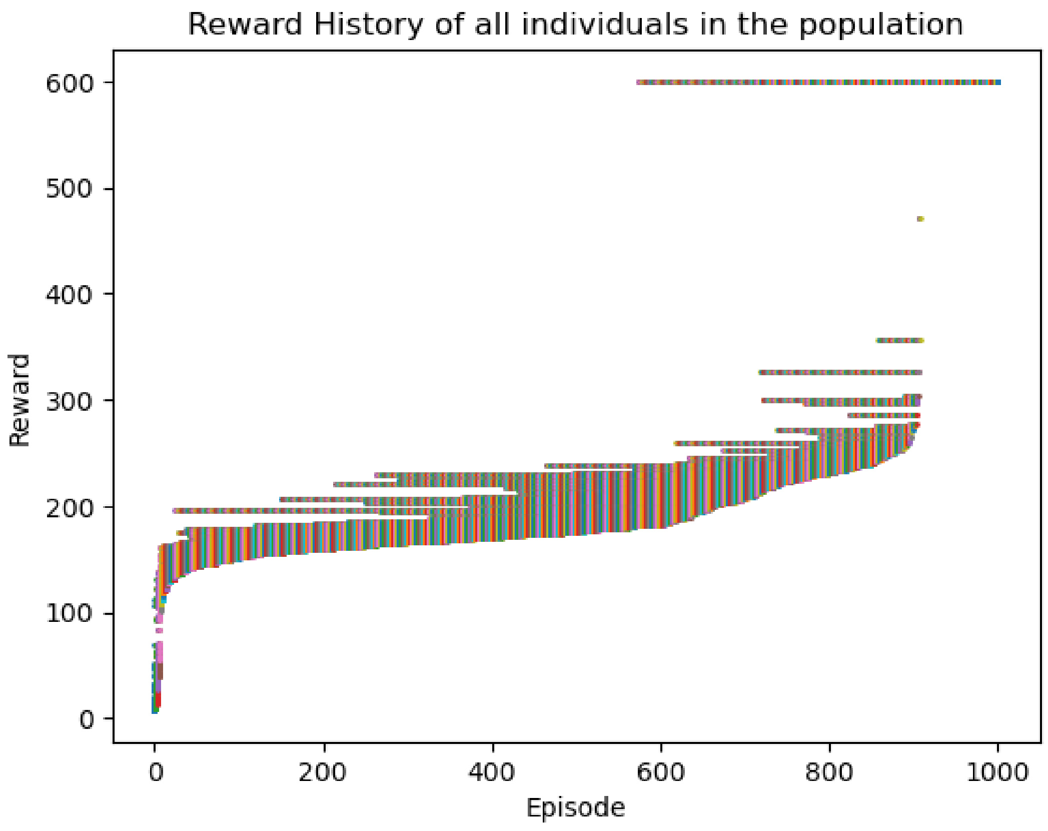
[3]:
```
## Create the EvolveNNAgent
# Parameters
episodes = 1000 # Number of generations to train
max_time_steps = 500 # Maximum number of time steps per episode
population_size = 100 # Number of agents in the population
mutation_percent = 0.25 # Percent of agents to mutate
mean = 1.0 # Mean of the normal distribution used to mutate the weights
std = 0.001 # Standard deviation of the normal distribution used to mutate the␣
 ↪weights

# Create the agent
evolve_nn_agent = EvolveNNAgent(cartpole_env, episodes, max_time_steps,␣
 ↪population_size, mutation_percent, mean, std)
```

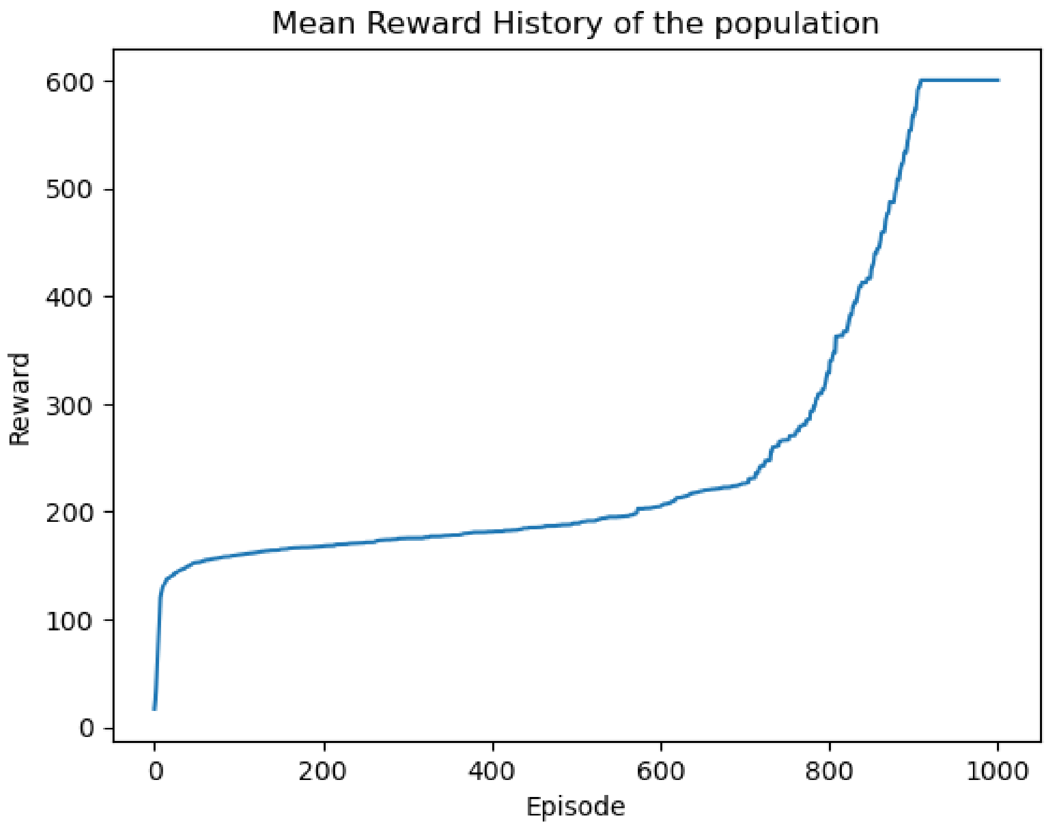
[4]:
```
# Run the agent training
evolve_nn_agent.run(verbose=True)
```

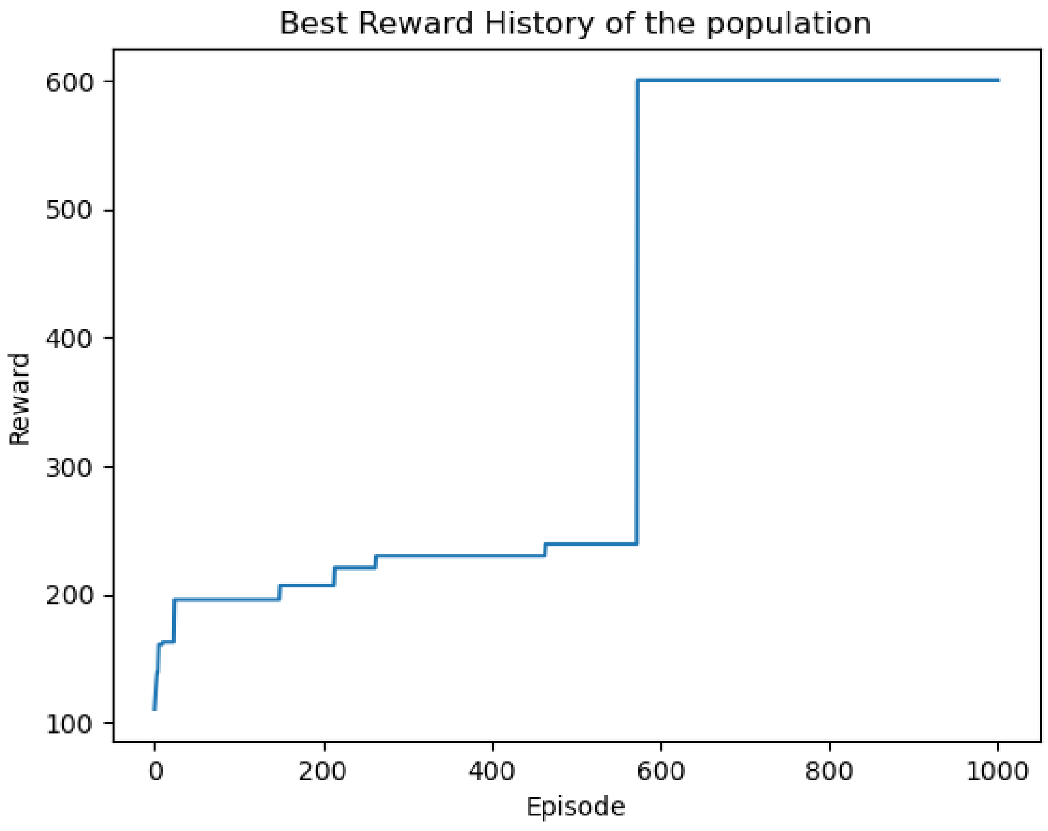
```
Episode: 100/1000 | Mean Reward for episode: 159.79
Episode: 200/1000 | Mean Reward for episode: 168.08
Episode: 300/1000 | Mean Reward for episode: 175.3
```

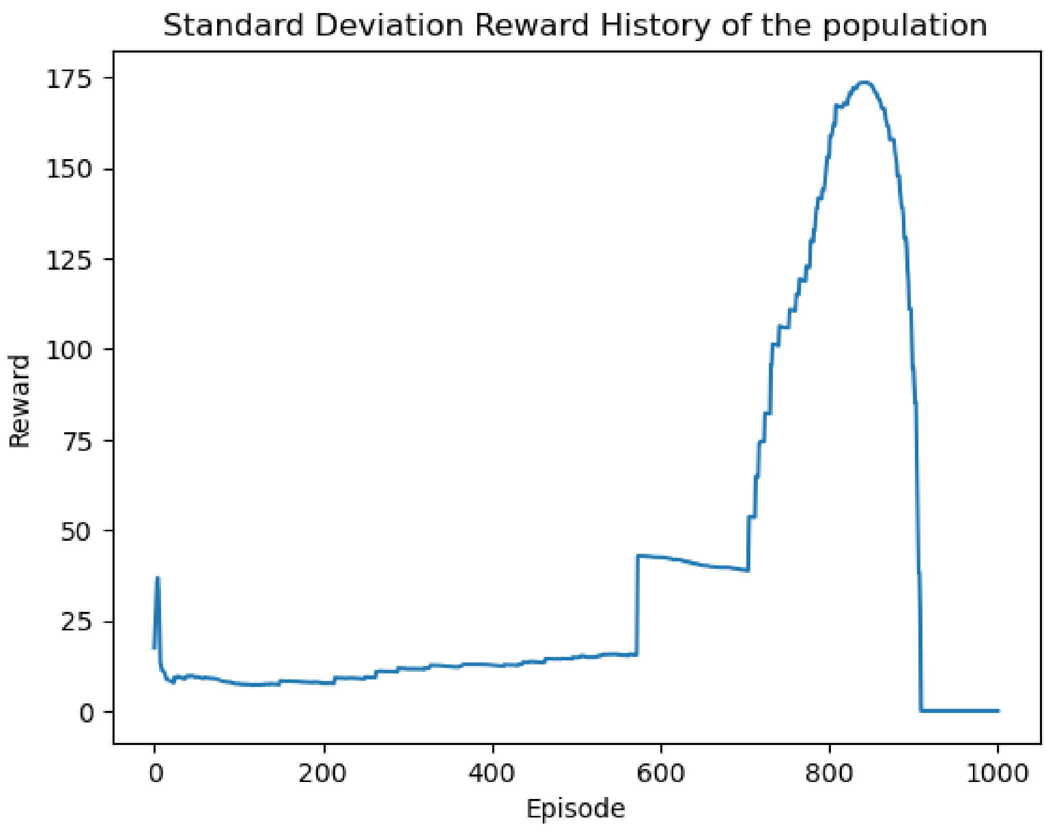
```
Episode: 400/1000 | Mean Reward for episode: 181.28
Episode: 500/1000 | Mean Reward for episode: 189.17
Episode: 600/1000 | Mean Reward for episode: 205.06
Episode: 700/1000 | Mean Reward for episode: 225.96
Episode: 800/1000 | Mean Reward for episode: 328.78
Episode: 900/1000 | Mean Reward for episode: 566.8
Episode: 1000/1000 | Mean Reward for episode: 600.0
```

[5]:
```
# Plot the results
evolve_nn_agent.plot_results()
plt.show()
```




Mean Reward History of the population
Reward
Episode
0
100
200
300
400
500
600
0
200
400
600
800
1000


Best Reward History of the population
Reward
Episode
600
500
400
300
200
100
0
200
400
600
800
1000

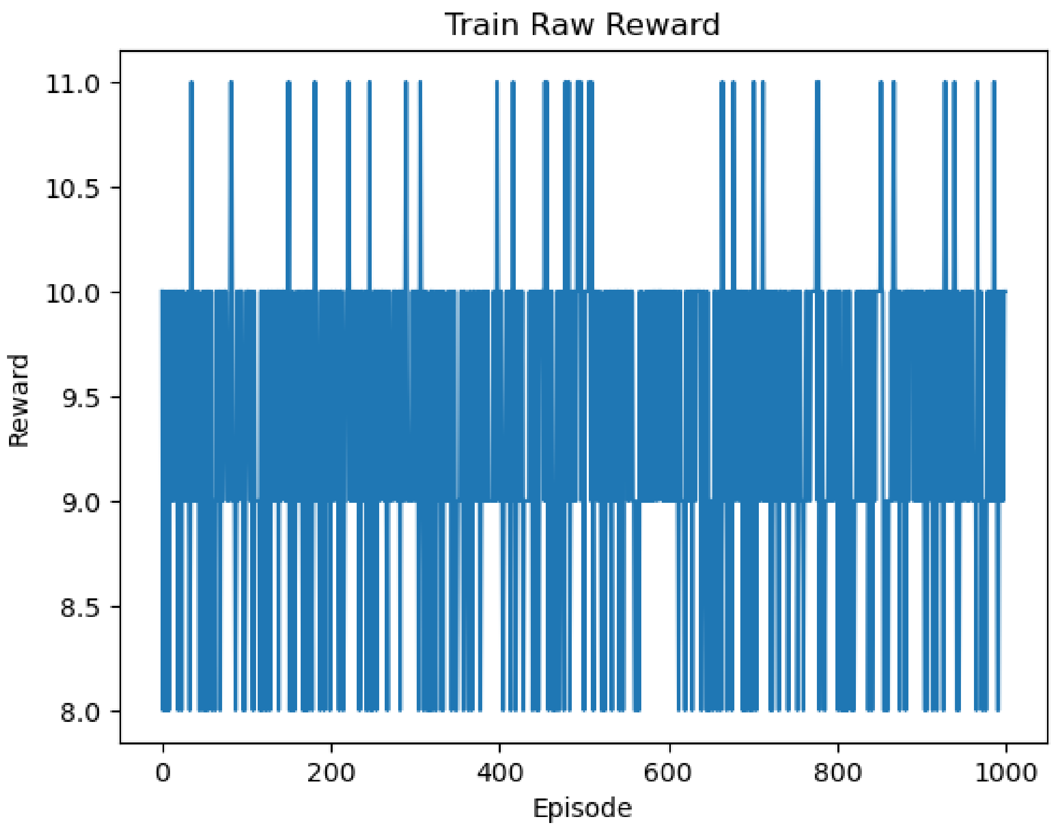


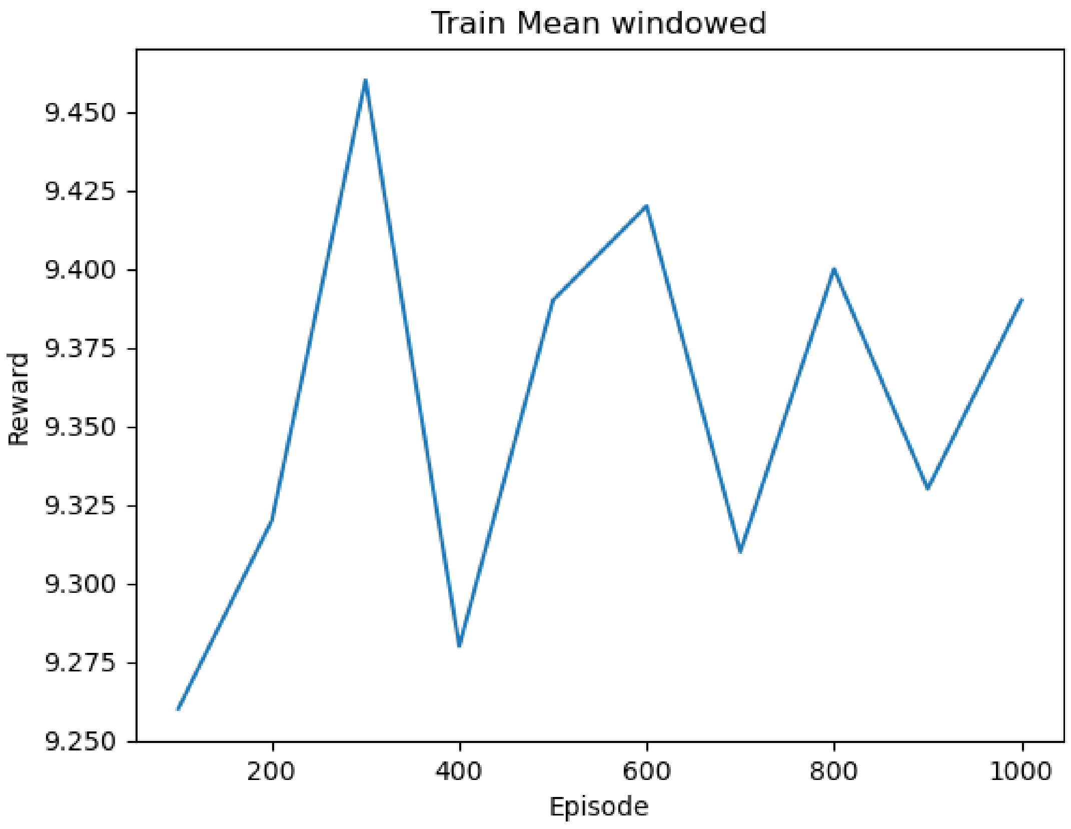
[5]:
```
## Create the DeepQAgent
# Parameters
model_weights = "CartPole-DQN.pt" # Path to the model weights
use_prev = False # Use the previous model weights
episodes = 1000 # Number of episodes to train
test_episodes = 100 # Number of episodes to test
max_time_steps = 500 # Maximum number of time steps per episode
sample_size = 32 # Batch size
hidden_size = 8 # Hidden layer number of neurons
mean_reward_window = 10 # Mean reward window episodes/window size
alpha = 0.01 # Learning rate
gamma = 0.9 # Discount factor

# Create the agent
deep_q_agent = DeepQAgent(cartpole_env, model_weights, use_prev, episodes,␣
 ↪test_episodes, max_time_steps,
                          sample_size, hidden_size, mean_reward_window, alpha,␣
 ↪gamma)

# Run the agent training
```

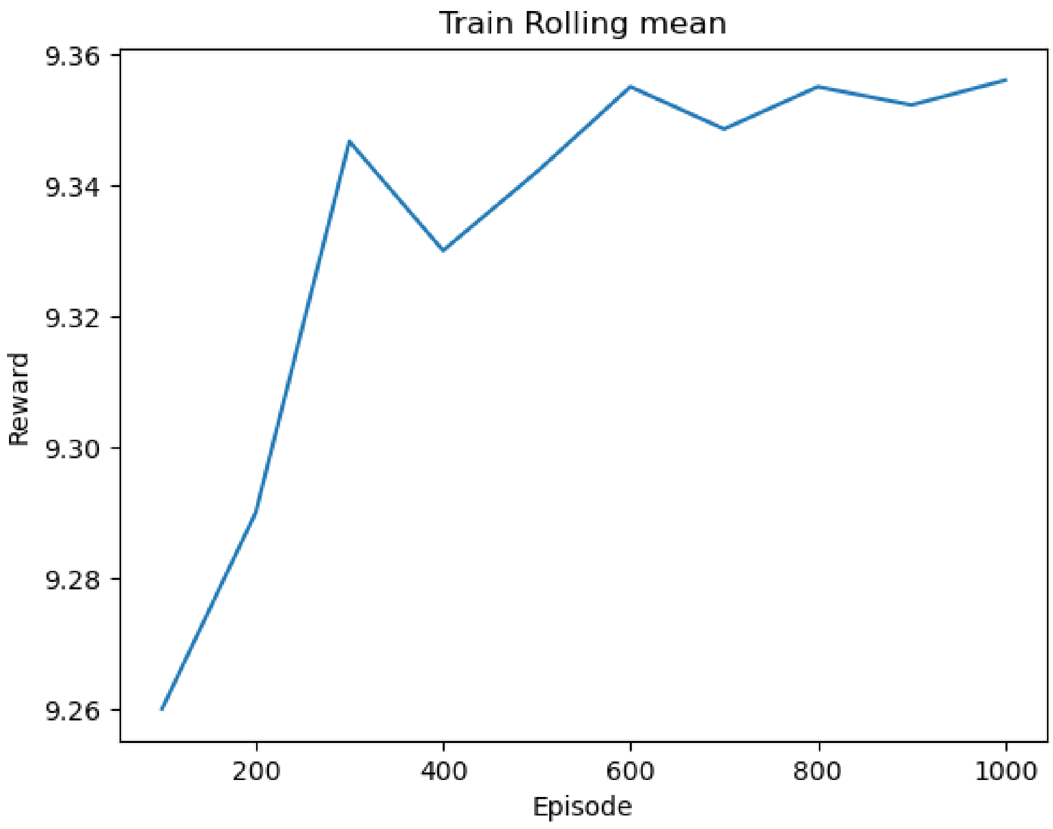
```
deep_q_agent.run(save_model=True, verbose=True)
```

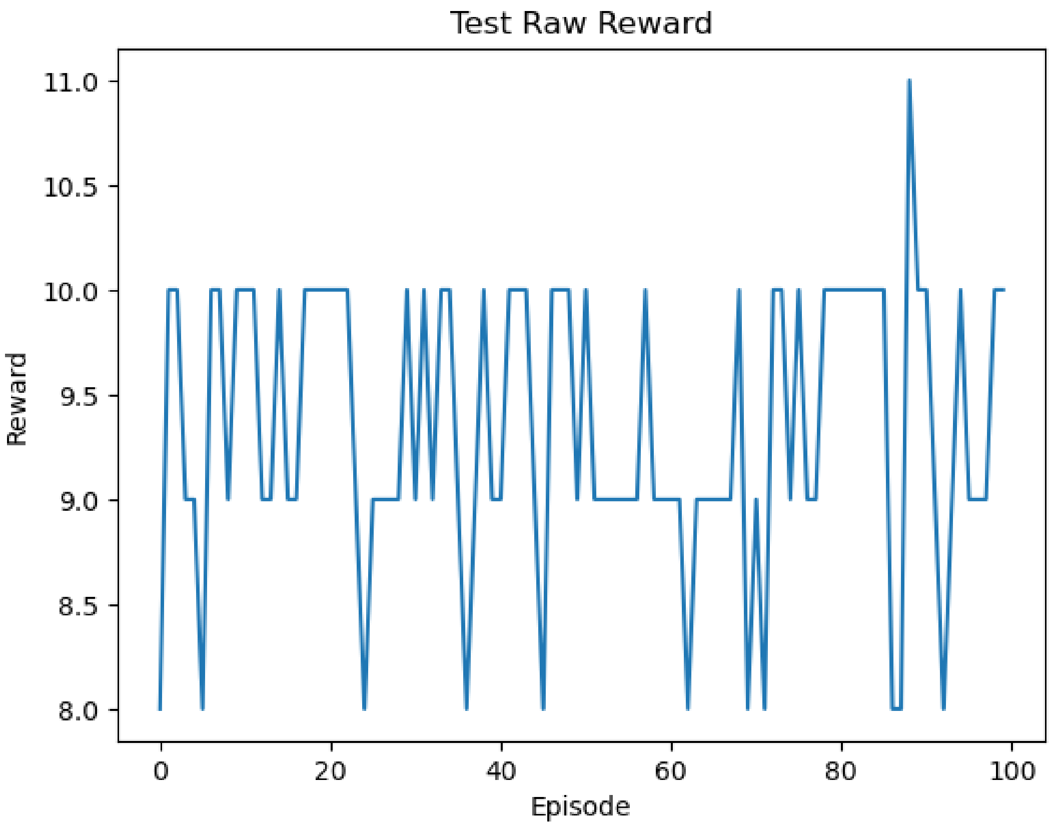
```
Episode: 1/1000 | Mean Reward until now: 10.00
Episode: 101/1000 | Mean Reward until now: 9.27
Episode: 201/1000 | Mean Reward until now: 9.29
Episode: 301/1000 | Mean Reward until now: 9.35
Episode: 401/1000 | Mean Reward until now: 9.33
Episode: 501/1000 | Mean Reward until now: 9.34
Episode: 601/1000 | Mean Reward until now: 9.35
Episode: 701/1000 | Mean Reward until now: 9.35
Episode: 801/1000 | Mean Reward until now: 9.36
Episode: 901/1000 | Mean Reward until now: 9.35
Episode: 1000/1000 | Mean Reward until now: 9.36
Testing…
Mean Reward in Test: 9.35
```

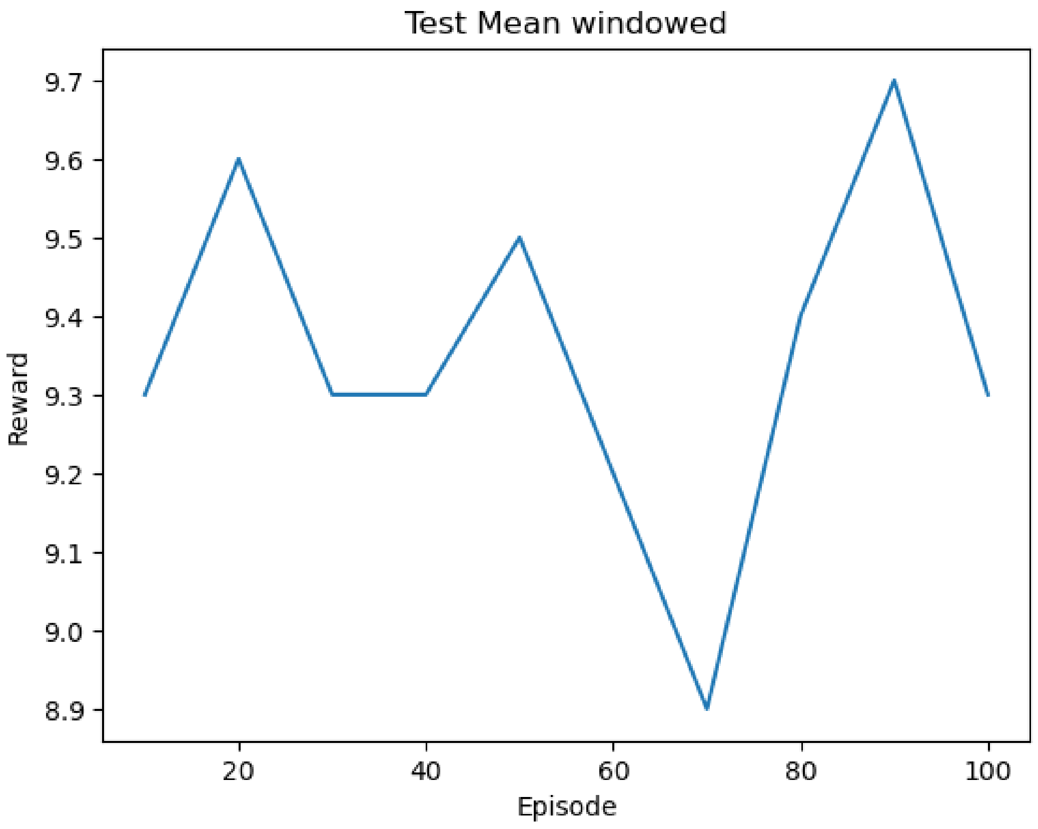
[6]:
```
# Plot the results
deep_q_agent.plot_results()
```

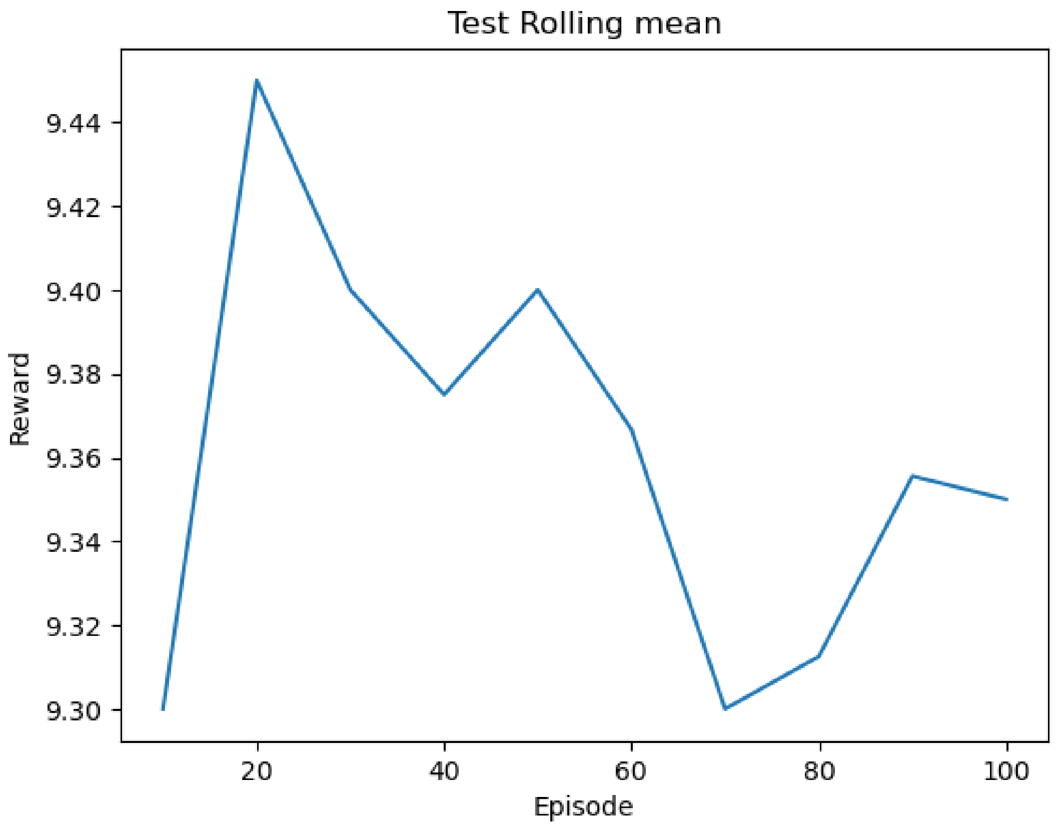



Train Mean windowed
Reward
Episode
9.250
9.275
9.300
9.325
9.350
9.375
9.400
9.425
9.450
200
400
600
800
1000


Train Rolling mean
Reward
Episode
9.36
9.34
9.32
9.30
9.28
9.26
200
400
600
800
1000


Test Raw Reward
Reward
Episode
11.0
10.5
10.0
9.5
9.0
8.5
8.0
0
20
40
60
80
100


Test Mean windowed
Reward
9.7
9.6
9.5
9.4
9.3
9.2
9.1
9.0
8.9
20
40
60
80
100
Episode


Test Rolling mean
Reward
Episode
9.30
9.32
9.34
9.36
9.38
9.40
9.42
9.44
20
40
60
80
100