Malachi Eberly
Assignment 7 Report

1. RL Algorithm

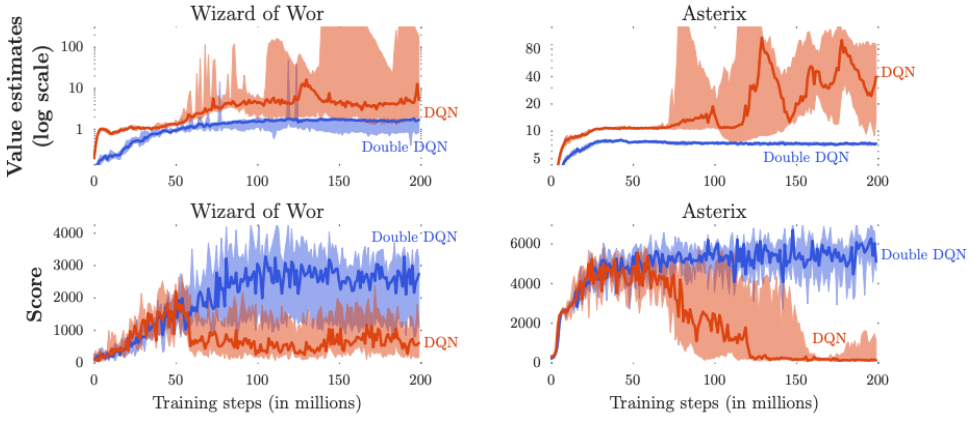
   This code uses a Double Deep Q-Network (DDQN) algorithm to learn how to play Super Mario Bros. Going through the code, I saw many references to Q-values, so I knew it was related to Q-Learning. I also saw that in neural.py, there were two copies of a neural network: the online network and the target network. These are both used in agent.td_target(); the online network is used to pick the best action, but the target network is used to evaluate its Q-value. This is a key adjustment from regular DQN.

2. Algorithm Research

   DDQN was introduced in 2015 by researchers at Google DeepMind. It's an extension of regular DQN, which was introduced in 2013 by DeepMind Technologies before DeepMind was purchased by Google. DDQN was developed to improve the performance of previous deep reinforcement learning approaches, especially DQN, and it was evaluated using Atari 2600 games in a similar manner. Its significance lies in the increase of performance it presents in many games and other applications by addressing overestimation.

3. Algorithm Explanation

   The main purpose of the algorithm is to improve the performance of DQN by accounting for the problem of overestimation. Because the same neural network is responsible for choosing an action and evaluating its value, the network may begin to give very high Q-Values, which for some tasks doesn't cause issues, but if the overestimations are not uniform or concentrated around valuable states, performance can suffer. The addition of a separate neural network to evaluate the actions addresses this problem and improves performance. This is demonstrated in the graphs below from the original DDQN paper; The estimated values from DQN are much higher, and the performance in DDQN is significantly improved, especially when the number of training steps grows large.



DDQN starts by initializing two neural networks with the same architecture: the online network and the target network. The online network is continually updated, and the target network is periodically updated to match the online network. Through training, a replay buffer is saved, which contains states, actions, and rewards that the agent experiences. It uses a model-free, off-policy approach to solve the given problem. DDQN has many strengths; it's reliable and stable, it's applicable to many complex problems, and it's able to be easily integrated with other enhancements. Its weaknesses include its computational complexity and the sensitivity of the hyperparameters. Small changes in network structure or learning parameters can have major impacts on the agent's ability to learn.

When training, the following update rule is used:

$$Y_t^{\text{DoubleDQN}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\operatorname{argmax}}\, Q(S_{t+1}, a; \boldsymbol{\theta}_t), \boldsymbol{\theta}_t^-)$$

Where θ is the online network and θ^- is the target network. Most of the structure is the same as the update rule for DQN; the new Q-value is chosen using the next reward, the discount factor, the next state, and the next best action. The next best action is still chosen with θ within argmax(), but when evaluating that action from the next state S_{t+1}, θ^- is used instead.

4. Code Analysis

I decided to look at this code for solving CartPole with DDQN: https://github.com/amirmirzaei79/CartPole-DQN-And-DDQN/blob/master/Train_DDQN.py

The hyperparameters are initialized at the beginning:

- Episode limit = 100
- Target update delay = 2 (Target network updated every 2 episodes)
- Learning rate = 1e-4 = 0.0001
- Epsilon start = 1
- Epsilon end = 0.1
- Epsilon decay = 0.9 / 2.5e3 = 0.00036
- Discount factor = 0.99
- Experience replay length = 10000

The learning loop uses optimize_model() to improve:

- The training batch size is set to the minimum between 100 and the current length of the replay buffer
- The training sample is selected randomly from the replay buffer
- The current state Q-value estimates, next state Q-value estimates, and next actions are defined from the neural networks
- For the number of training samples sample:
    - Retrieve the next action
    - Use the update rule for the estimated Q-value
- Fit the data

The reward mechanism for CartPole works by giving the agent a +1 reward for every time step of the episode, attempting to maximize the number of time steps. The episode ends if the pole is more than 15 degrees away from vertical, or if the cart moves more than 2.4 units away from the center.

The update rule follows the equation in section 3: the estimated Q-value for the current state and action is found using the training sample's related next state, reward, current Q-value estimate, and the discount factor.

This optimizer is used in the episode training loop, which is called in the main function. The score and reward of each episode is recorded and tested each time. The model is saved if a new best score is achieved.