CSE 546 Assignment 2 - Deep Q-Networks
Checkpoint 2

Ijaj Ahmed
Department of Computer Science
University at Buffalo, SUNY
Buffalo, NY 14260
ijajahme@buffalo.edu

Mohammed Nasheed Yasin
Department of Linguistics
University at Buffalo, SUNY
Buffalo, NY 14260
m44@buffalo.edu

Abstract

This report presents our experiments on a self-created `GridWorld` environment and two additional environments, `CartPole-v1` and `LunarLander-v2`. These environments were selected from the `ClassicalControl` and `Box2D` collections in the Gymnasium library [2]. We applied DQN and Double DQN to solve these environments and conducted a case study on the outcomes.

Environment Description

1 CartPole-v1

In the cart-pole problem version described by Barto, Sutton, and Anderson [1] a pole is connected to a cart through a joint that cannot be moved. The cart can move on a track without any friction. The pole is positioned upright on the cart and the objective is to keep the pole balanced by applying forces to the cart in either the left or right direction. This environment has been visualized in Figure 1.

1.1 Action Space

The action is an ndarray of shape (1,) that can take on values 0, 1 to indicate the direction in which the cart is pushed with a fixed force.

- 0: The cart is pushed to the left
- 1: The cart is pushed to the right

The velocity that is either decreased or increased by the force applied is not constant and depends on the angle at which the pole is pointing. The center of gravity of the pole affects the amount of energy required to move the cart beneath it.

1.2 Observation Space

The observation is an ndarray of shape (4,) where the values represent the positions and velocities described next.

Num	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	-Inf	Inf
2	Pole Angle	$\sim$ -0.418 rad (-24°)	$\sim$ 0.418 rad (24°)
3	Pole Angular Velocity	-Inf	Inf

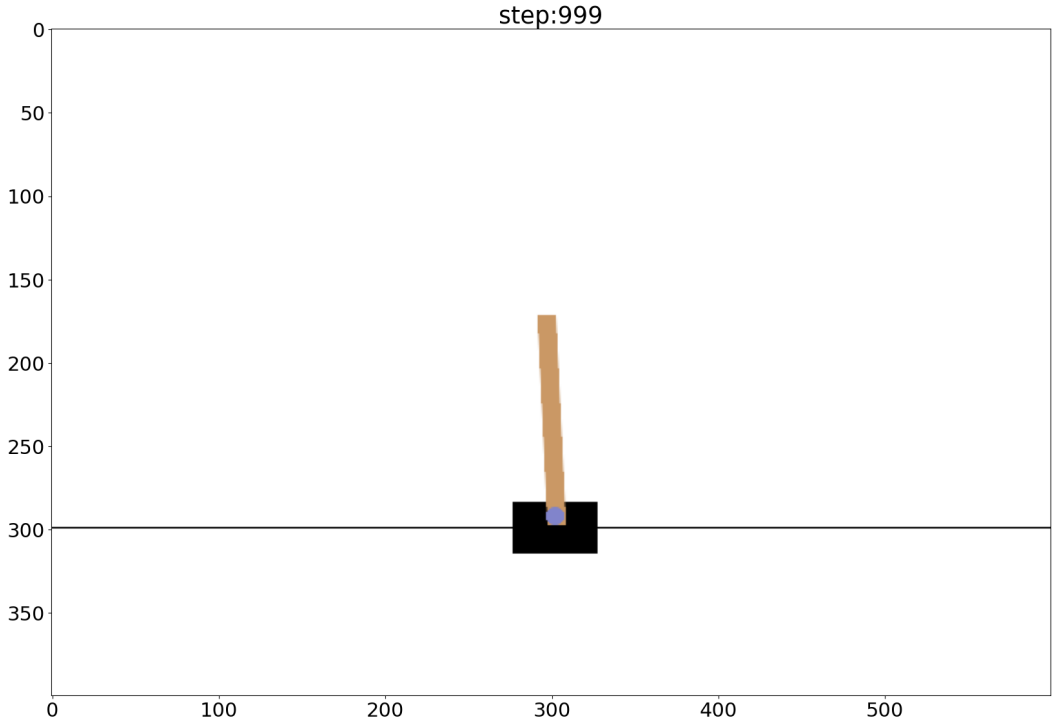

Figure 1: A frame from CartPole-v1

Although the ranges above indicate the possible values for each element in the observation space, they do not reflect the allowed values of the state space in an ongoing episode. Specifically:

- The cart x-position (index 0) can be take values between `(-4.8, 4.8)`, but the episode terminates if the cart leaves the `(-2.4, 2.4)` range.
- The pole angle can be observed between `(-.418, .418)` radians or `(±24°)`, but the episode terminates if the pole angle is not in the range `(-.2095, .2095)` or `(±12°)`

1.3 Rewards

The objective of the task is to keep the pole upright for as long as possible. To encourage this behavior, a reward of +1 is given for every step taken, including the final step when the episode terminates. In version 1 of the task, the threshold for achieving a successful outcome is set at 475.

1.4 Start State

All observations are assigned a uniformly random value in `(-0.05, 0.05)`

1.5 Episode End

The episode terminates under these conditions:

1. Termination: Pole Angle is greater than `±12°`
2. Termination: Cart Position is greater than `±2.4` (center of the cart reaches the edge of the display)
3. Truncation: Episode length is greater than `500`

2 LunarLander-v2

This environment represents a classic problem of optimizing rocket trajectory. Based on Pontryagin's maximum principle, the optimal approach is to either fire the engine at full throttle or turn it off completely. As a result, this environment has discrete actions: the engine is either on or off.

Two versions of the environment are available: discrete and continuous. In our work we have used the discrete version. The landing pad is always located at coordinates `(0,0)`, which are represented by the first two numbers in the state vector. It is possible to land outside of the landing pad. Since fuel is unlimited, an agent can learn to fly and land successfully on its first attempt.

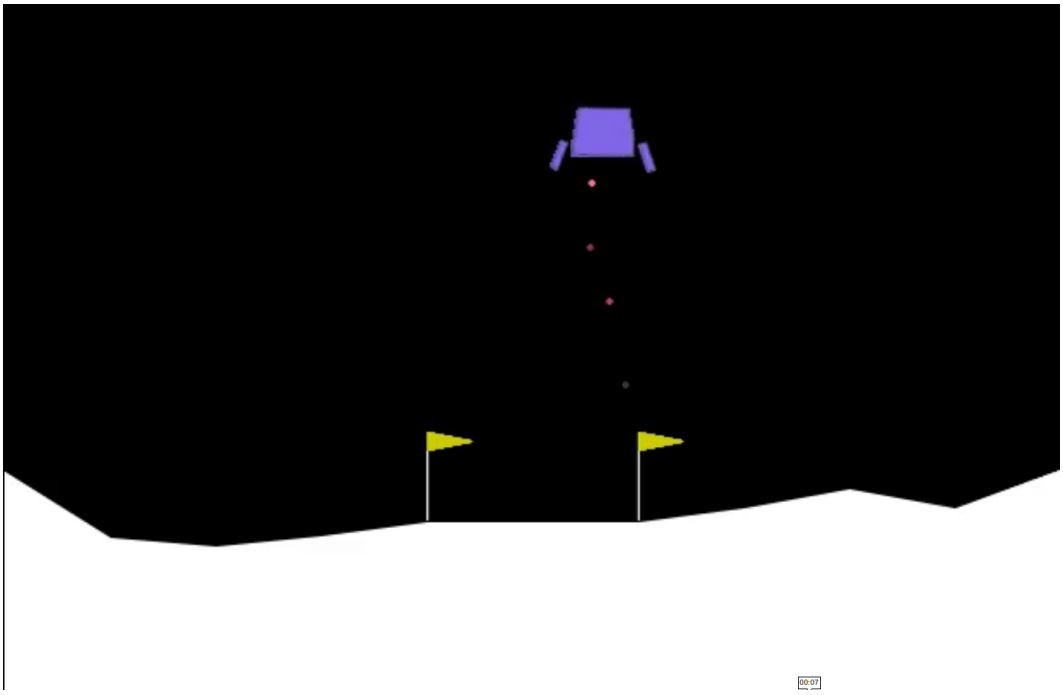

Figure 2: A frame from LunarLander-v2

2.1 Observation Space

The state of the environment is represented by an 8-dimensional vector that includes the `x` and `y` coordinates of the lander, its linear velocities in `x` and `y`, its angle and angular velocity, and two boolean values indicating whether each leg is in contact with the ground.

2.2 Action Space

Four discrete actions are available in this environment: remain idle, activate the left orientation engine, activate the main engine, or activate the right orientation engine.

2.3 Reward

A reward is given after each step in the environment. The total reward for an episode is calculated by summing the rewards for all steps within that episode. The reward for each step is determined by the following factors:

- The reward increases/decreases as the lander gets closer/further from the landing pad.
- The reward increases/decreases as the lander moves slower/faster.
- The reward decreases as the lander tilts more (angle not horizontal).

- The reward increases by 10 points for each leg in contact with the ground.
- The reward decreases by 0.03 points for each frame a side engine is firing.
- The reward decreases by 0.3 points for each frame the main engine is firing.

An additional reward of -100 or +100 points is given for crashing or landing safely, respectively. An episode is considered solved if it scores at least 200 points.

2.4 Starting State

At the beginning of each episode, the lander is positioned at the top center of the viewport and a random initial force is applied to its center of mass.

2.5 Episode End

An episode terminates if any of the following conditions are met:

1. The lander crashes (its body comes into contact with the moon).
2. The lander moves outside of the viewport (its `x` coordinate is greater than 1).
3. The lander is not awake. According to the Box2D documentation, a body that is not awake does not move or collide with any other body.

3 The Grid-world Environment

There are certain features that are common to both the stocastic and deterministic environments.

Environmental Elements

Figure 3: Environmental elements (from left to right; top to bottom) Agent, Goal, Negative Reward, Reward

The enivironment is a `6x6` grid defined according to the Gym [2] API. With `36` possible positions that the agent can occupy. The goal is to reach the oasis (shown in Figure 3) **after consuming all** the juice (positive reward) within a (configurable) maximum number of time steps. If the agent lands

on a juice tile it is awarded +0.99 and the cactus (negative reward) leads to a -1.0 reward. Once all the juice is consumed, the agent must proceed to the oasis to earn a reward of +1.0 The states in this enviroment are a **combination** of the agent's position and the currently available rewards (positive, negative and goal) on the grid. The Formula 1 gives us the number of possible states for the agent.

$$num_{states} = num_{pos} \sum_{k=0}^{c_{reward}} \binom{c_{reward}}{k} \tag{1}$$

Here num_{pos} refers to the number of grid squares, `36` in our case. c_{reward} refers to the number of positive rewards + the number of negative rewards + 1 (for the goal state). We have `6` negative rewards, `3` positive rewards and one goal. Hence, the num_{states} for us is `36864`.

In each position our agent can take 4 potential actions: 1. Left 2. Right 3. Up 4. Down. Resetting the environment will not change the location or distribution of the rewards and goal state. It only alters the initial state of the agent.

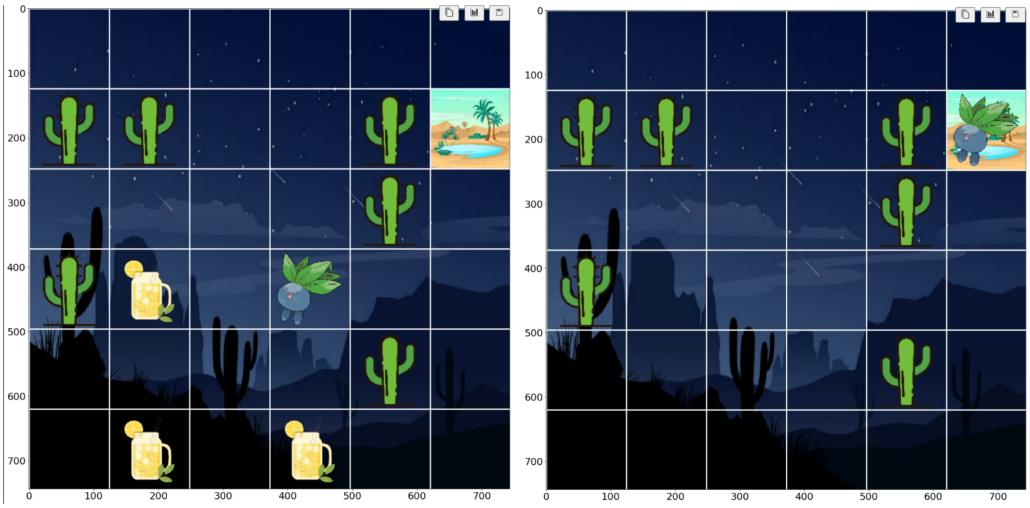

Figure 4: Environment visualizations

3.1 Safety in AI

The following are properties of the environment that ensure valid behavior from the agent:

1. We ensure that the agent consumes *all* the `pos_reward` and reaches the `goal_state` in the fewest number of steps by imposing a penalty of `0.1x` the `max_reward` for every move made.
2. The reward on all squares is consumed once the agent lands in that state, preventing the agent from settling down in a *high-reward* neighborhood.
3. The result of any action (left, right, top, down) are clipped to the min and max values of 0 and `GridSize` (6 in our case) respectively, ensuring that the agent never leaves the environment.
4. If the agent makes a move but remains in the same spot, we impose the *maximum negative reward* (-1.0 in our case). This allows the agent to disincentivize making fruitless moves.
5. We also prevent a *goal rush* (before the agent consumes all the positive rewards) by making the goal square unreachable when there are positive reward squares left. If the agent takes an action to move into a goal square before collecing all the *positive rewards*, they will be kept on the same square, incurring an additional penalty (-1.0 in our case) as detailed in the previous point.

6. The stocasticity of the starting point and limited time steps will nudge the agent to build strategies that accumulate the maximal reward in the shortest time.

4 Discuss the benefits of:

4.1 Using experience replay in DQN and how its size can influence the results

The experience replay in DQN helps to curb the problem of correlation between consecutive input samples to the training network which otherwise causes the network to diverge or make it unstable. The experience relay ensures the samples are i.i.d (independent and identically distributed) by randomizing them among batches by same distribution and are kept independent of each other in the same batch. This avoids the overfitting of some group of consecutive samples as the temporal variation is poor. The experience relay provides the opportunity to train the network by considering a large pool of samples already trained. There is a tradeoff between choosing the buffer size and the final outcome of such choice. Generally a small size buffer helps to learn the environment rapidly at the cost of ignoring the previous experiences. Similarly a larger buffer suffers from slow learning

4.2 Introducing the target network:

The target network helps to avoid oscillation in policy as conventional Q value update changes the current estimate and target value. If the target Q value is impacted by the current update of the Q value, the training chases a non stationary target and does not help much in learning efficiently. WIth the introduction of a target network which is updated with the primary network only after some particular interval helps to train the model effectively.

4.3 Representing the Q function as *q'(s, w)*

This solves the problem of dealing with larger state-action domains or complex networks as the neural network representation can now serve as function approximation. The design matrix now reduces to updating the weight of the network to minimize the loss of the estimate and target network.

5 Show and discuss your results after applying your DQN implementation on the three environments.

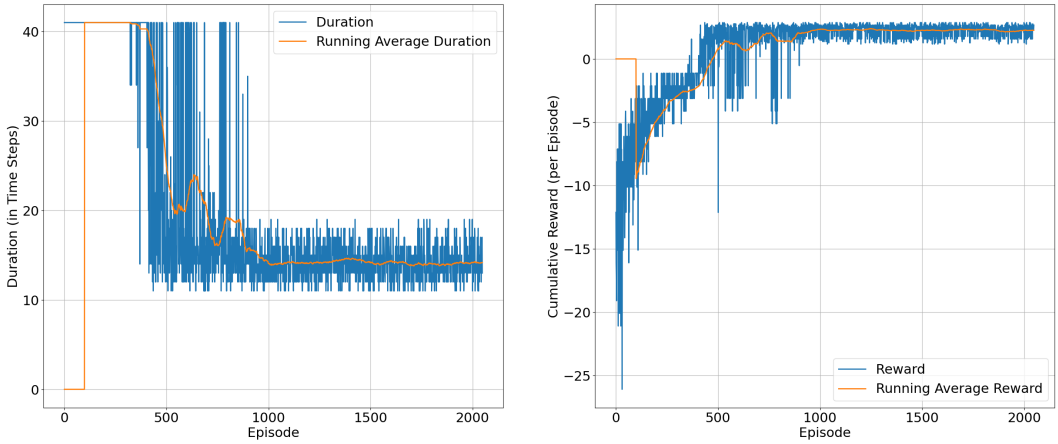

Figure 5: Grid Training

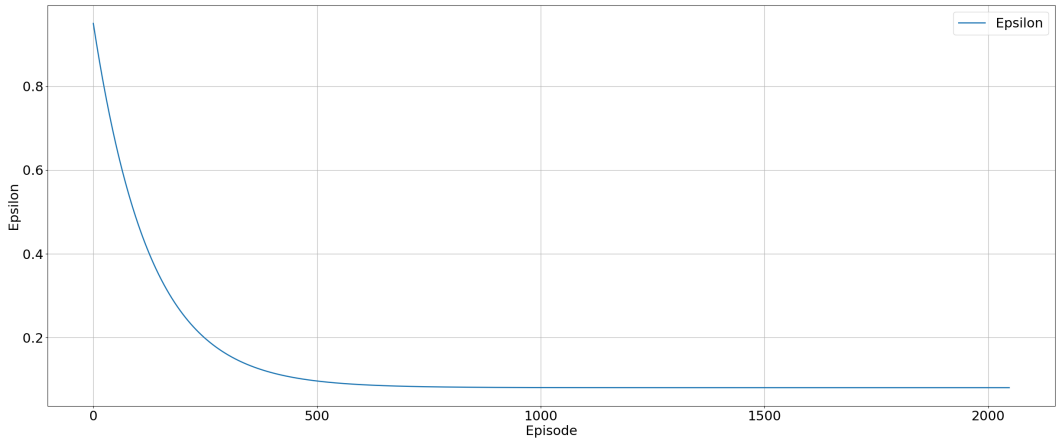

Figure 6: Epsilon Decay – Grid Env

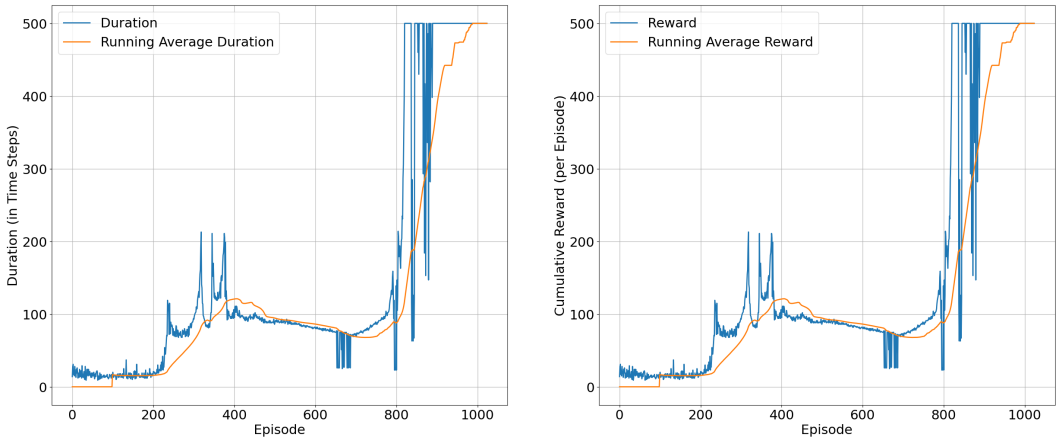

Figure 7: Cart-Pole Training

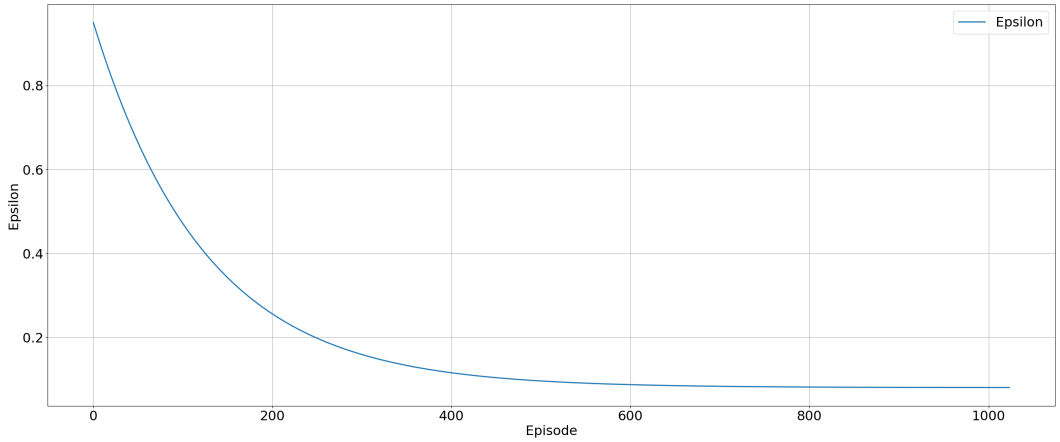

Figure 8: Epsilon Decay – Cart Pole

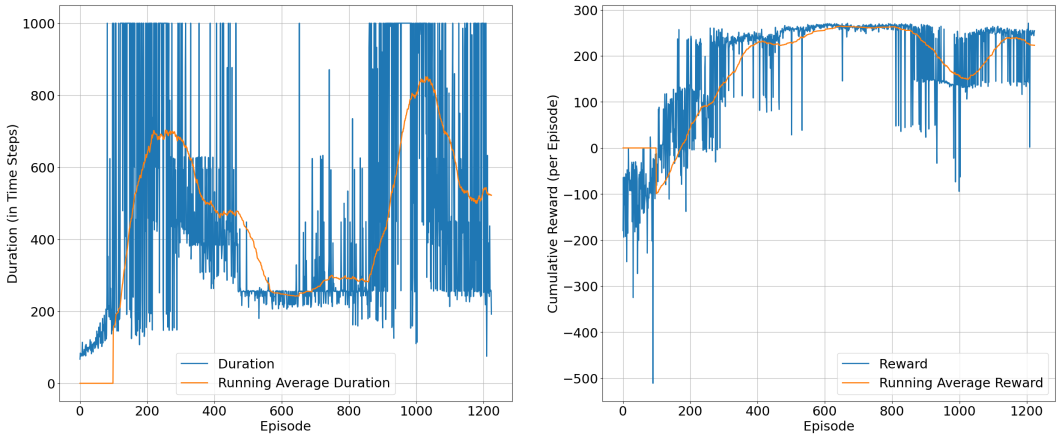

Figure 9: Lunar Lander Training

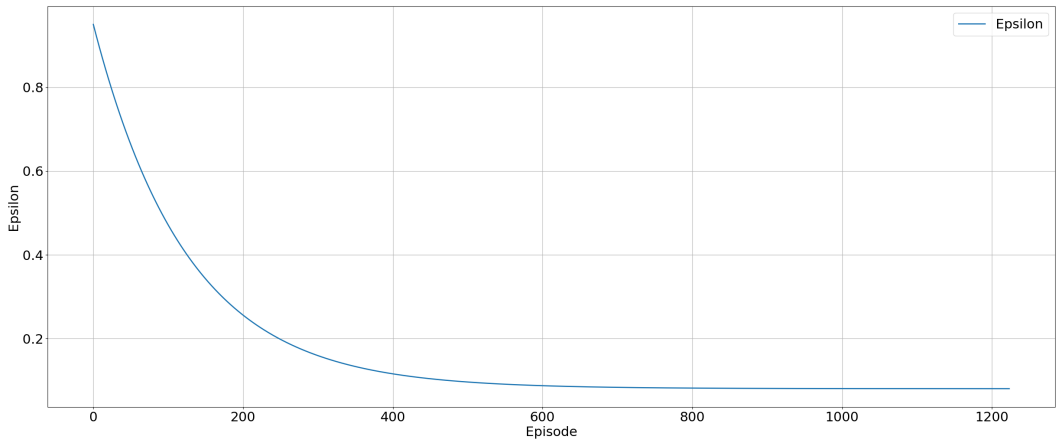

Figure 10: Epsilon Decay – Lunar Lander

We successfully solved all three environments using simple `FeedForward` neural networks. Additionally, we implemented a tau-based Target Network update strategy to stabilize the training process and used an experience replay memory of size 10000. Of the three environments, `CartPole-v1` exhibited the most unstable learning. This instability is due to the fact that a single misstep in `CartPole-v1` can cause the pole to collapse and the episode to end prematurely. The other two environments had more stable learning curves and were successfully solved by the DQN agent.

6 Provide the evaluation results. Run your agent on the three environments for at least 5 episodes, where the agent chooses only greedy actions from the learnt policy.

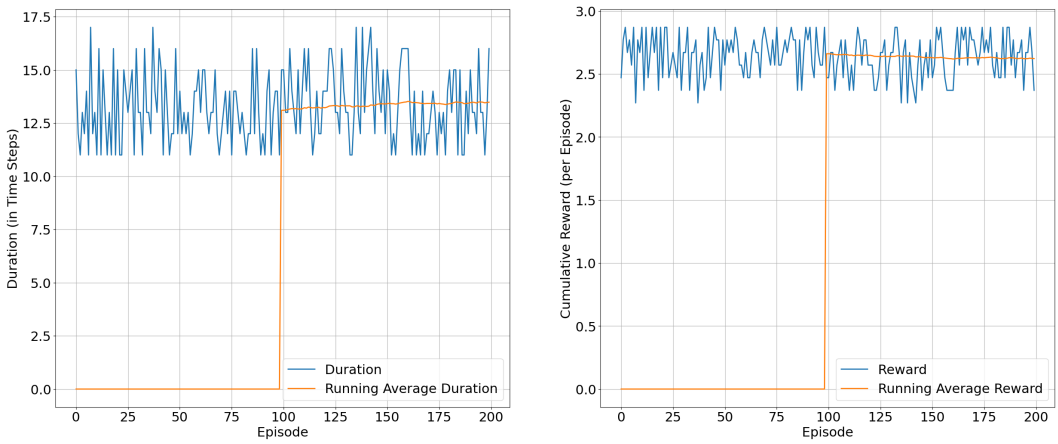

Figure 11: Grid Environment Evaluation

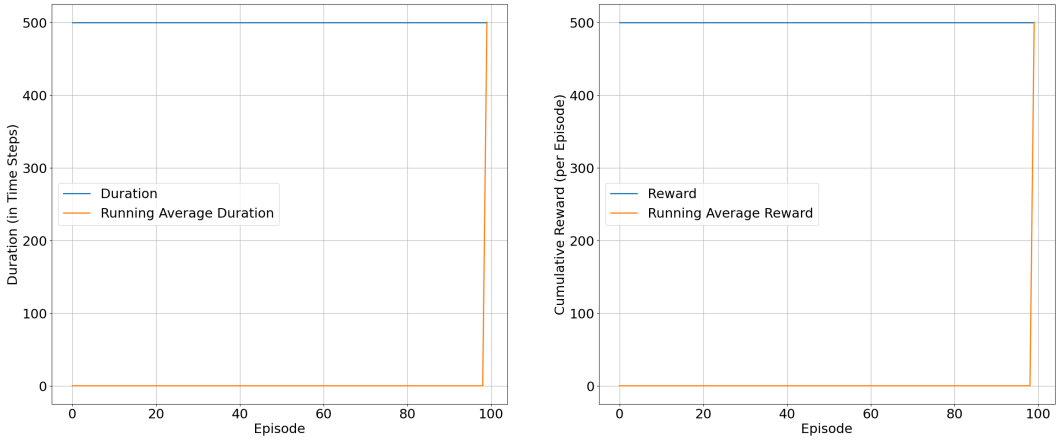

Figure 12: Cart Environment Evaluation

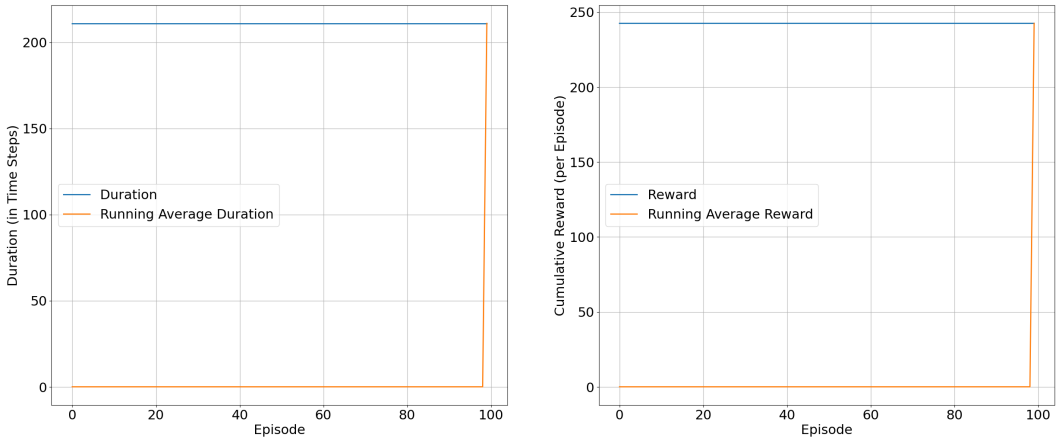

Figure 13: Lunar Environment Evaluation

The simple `FeedForward` neural network approach has successfully learned the optimal policy for all three environments. The evaluation result for the GridEnvironment is somewhat inconsistent because the agent is placed at a random starting square, which affects the number of steps required to reach the goal. To encourage the agent to reach the goal as quickly as possible, we applied a negative reward of -0.1 for each step taken.

References

[1] Andrew G Barto, Richard S Sutton, and Charles W Anderson. "Neuronlike adaptive elements that can solve difficult learning control problems". In: *IEEE transactions on systems, man, and cybernetics* 5 (1983), pp. 834–846.

[2] Greg Brockman et al. *OpenAI Gym*. 2016. eprint: `arXiv:1606.01540`.