Merve Tunçer 2040508

Neural Network and Deep Learning Homework 3

Deep Reinforcement Learning

The goal of this task is to use Reinforcement Learning to train and test neural network models to solve problems. A neural network will be trained as a Deep Q-learning (DQ) Agent in three different environments, provided by the OpenAI library gym.

- The first work involves adding some improvements to the code we saw in the lab lecture. To achieve the same accuracy with fewer training episodes, we'll have to tune the model.
- The same environment is used in the second task, but the screen pixels are used as inputs instead of the environment's state representation (cart position, cart velocity, pole angle, pole angular velocity).
- The third task includes the training of an RL agent in a new Gym environment. MountainCar has been chosen as the environment.

Reinforcement learning tasks mainly consist of training an *agent* A which interacts with an environment E. Every action performed causes a change in the state of the environment, which might either penalize or favor the agent assigning a *reward*. The agent naturally seeks to maximize its reward, therefore needs to learn the *optimal policy* which results in taking the best actions to obtain the largest reward given it finds itself in a certain state.

At every timestep t, the agent A performs an action at being in a given state st, receiving a reward rt for it. The next state is denoted as st+1. Formally, the goal of the agent is to maximize a total return Gt, which is weighted according to some discount factor $\gamma \in (0, 1)$, that quantifies how much A should care about future rewards:

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

CARTPOLE-V1 ENVIRONMENT

The cartpole-v1 environment consists of a pole attached by an un-actuated joint to a cart that moves along a frictionless track. A force of +1 or -1 is given to the cart to

control the system. The goal is to keep the pendulum from falling over when it starts upright. Every timestep that the pole remains upright earns you a +1 reward. When the pole is more than 15 degrees from vertical or the cart moves more than 2.4 units away from the center, the episode ends.

This problem is modeled by a state space vector of dimension 4 that represents the cart's current state and an action space vector of dimension 2 that represents the actions that the cart should execute given the state space vector's current state.

The state **space vector** is comprised of the follows:

- **0: Cart position** with the domain in [*-4.8, 4.8*]
- **1: Cart velocity** with the domain in [*-inf, +inf*]
- **2: Pole angle** with the domain in [*-24°, +24°*] or in radians [-0.418, +0.418]
- **3: Pole angular velocity** with the domain in [*-inf, +inf*]

The **actions space vector** is comprised of the follows:

- **0:** Push left
- **1:** Push right

The cart may achieve a maximum score of 500 points. Our goal is for our model to be able to learn in order for the cart to reach the highest possible score of 500.

The reward of +1 is given for every step that the pole remains upright. Our agent will move outside of the screen if we just employ this kind of reward. We don't want this kind of behavior. We'd want to see the cart on the screen. To achieve this, we must add a penalty to the position for the cart to keep close to the center of the screen.

The **reward** is the following:

reward = env_reward -1 * |cart_position|

where env reward is the environment's +1 reward and |cart position| is the absolute value of the cart's current position in the environment. The penalty is increased when the cart's position is higher in absolute terms.

The environment will continue to run until one of the following requirements is achieved:

- Pole angle is more than ±12°
- Cart Position is more than ±2.4 (center of the cart reaches the edge of the display)
- Episode length is greater than 200

MODEL HYPERPARAMETER TUNING

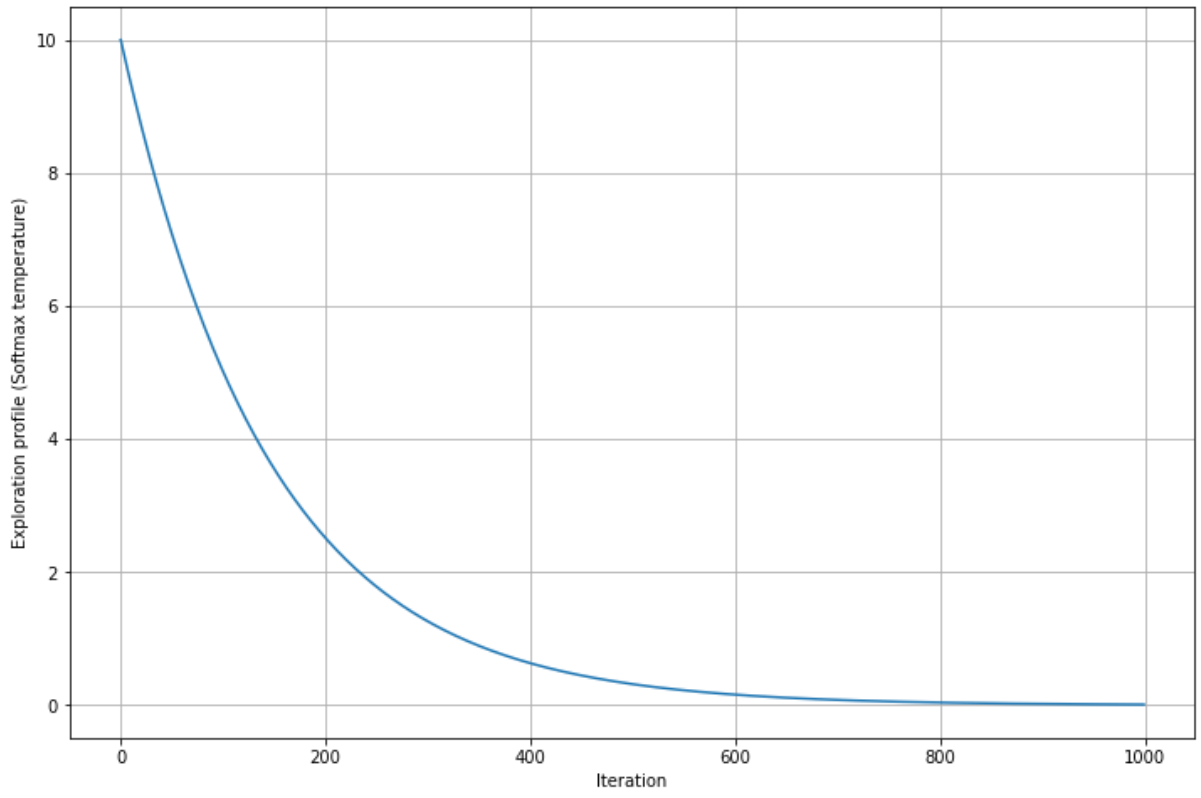
The following is the network architecture utilized to train in this environment:

- Input of 4 (the state space vector)
- First linear layer with number of neurons of 128
- Second linear layer with number of neurons of 128
- Output layer of dimension 2 (the action space vector) The activation function used is tanh.

The policy used is the softmax policy. The temperature used for the softmax policy is the following:

$$softmax\_temperature = init\_val * e^{\left(-\frac{log(initial\_val)*mul\_iter}{num\_iterations}\right)^{i}}$$

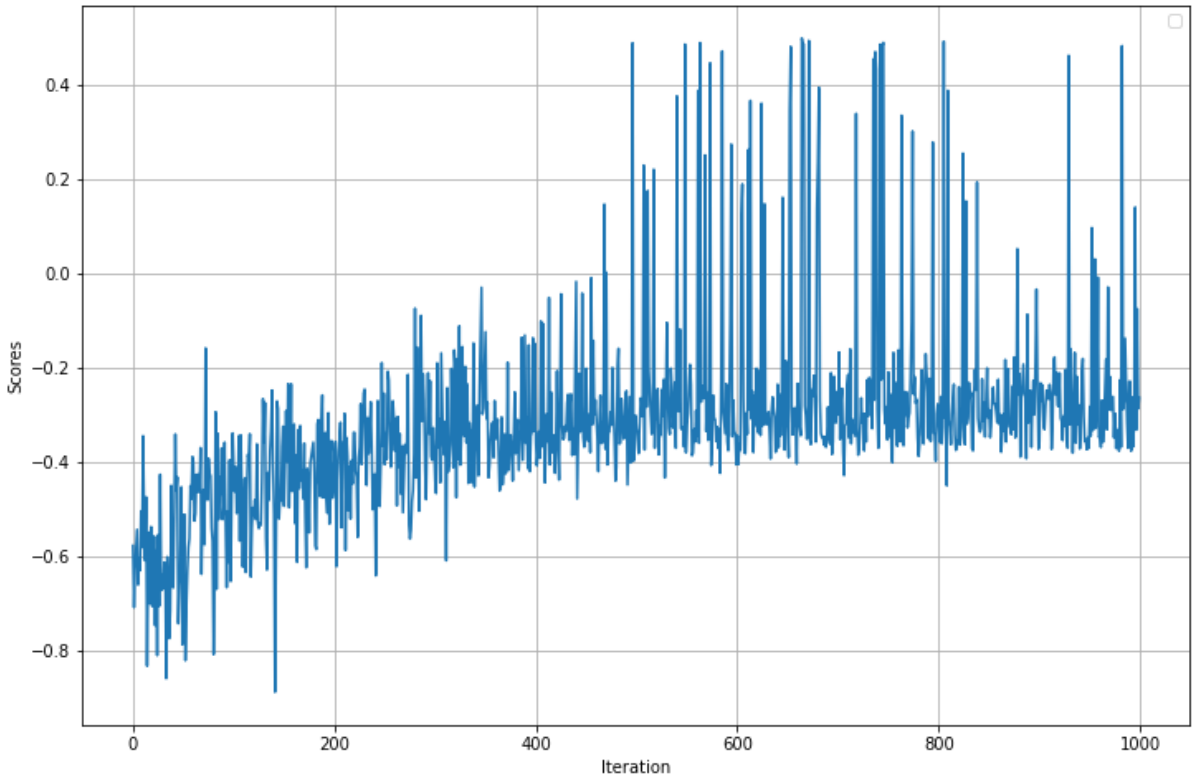
The hyperparameters tuned for this function are initial_val = 9, num_iterations = 1000, and mul_iter = 10. In the following you can see the plot of this function.



The following hyperparameters were used:

Gamma	0.98
Replay memory capacity	10000
Learning rate	0.02
Target net update steps	10
Batch size	128
Bad state penalty	0
Min samples for training	1000

The parameters are the same as in the lab rather than gamma that is 0.98. Convergence is faster with these parameters. The convergence is reached after 600 episodes as you can see in the following figure. However, as the training progresses, the catastrophic forgetting problem arises. To overcome this problem, we must limit the number of training episodes to 600.



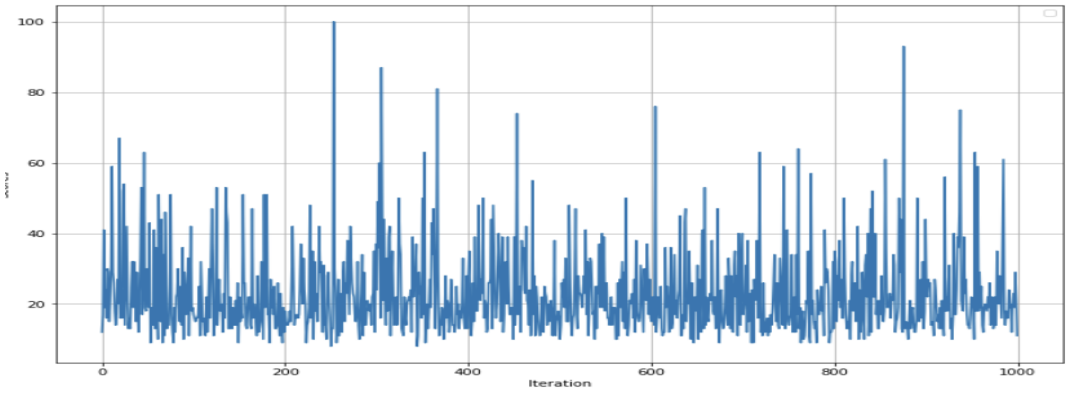
CARTPOLE WITH PIXEL SCREEN

In this section, the same gym environment will be studied but changing the observation space from the one provided by the library to the screen pixels. In this

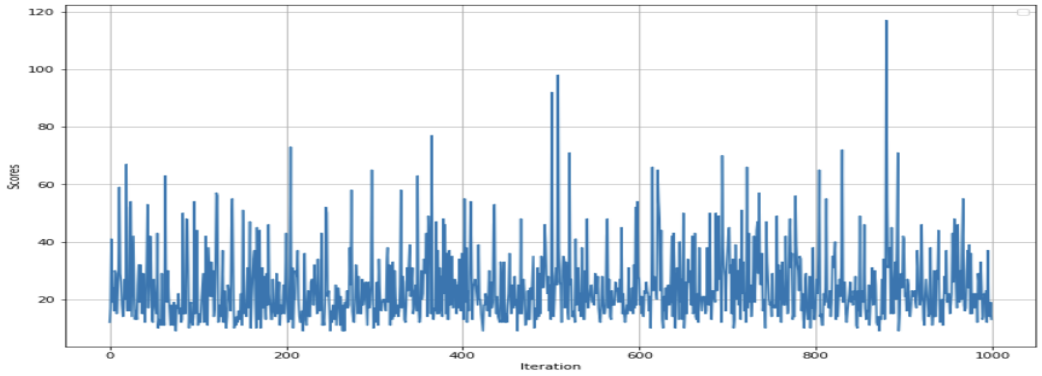
case, the network's input is an image frame of the environment. Based on the condition of the cart in the frame, the network will predict the action. An average Pooling layer with a kernel of dimension 8 transforms the image of size 50 x 75 from the image frame obtained from the environment, which is 400 x 600. During the training, 4 frame images are combined into a single tensor to inform the network about the falling pole's direction. As a result, the network is trained with a 4 channel tensor, each channel representing a single frame.
The network used is a convolutional neural network with the following architecture:

- a convolutional layer with filter size = 16, kernel size = 3, stride = 2 and padding = 1, ReLU activation
- a convolutional layer with filter size = 32, kernel size = 3, stride = 2 and padding = 1, ReLU activation
- a convolutional layer with filter size = 64, kernel size=3, stride=2 and padding = 1, ReLU activation
- a convolutional layer with filter size = 64, kernel size = 3, stride = 2 and padding = 1,ReLU activation
- a flatten layer of neurons = 1280
- a linear layer neurons= 640,tanh activation
- a linear layer neurons= 320,tanh activation
- a linear layer neurons= 2 (action state dim)

This network's learning rate is extremely slow, and it was unable to learn how to achieve the goal. It is left with a very low score. I attempted stacking frames because the traditional method of training with a single frame was ineffective. However, the stacked frames method used here does not produce improved outcomes. Unfortunately, due to a lack of time, it was difficult for me to experiment with other techniques and/or apply networks that had been researched in academic literature. The results of both ways used throughout the homework implementation are shown in the following.



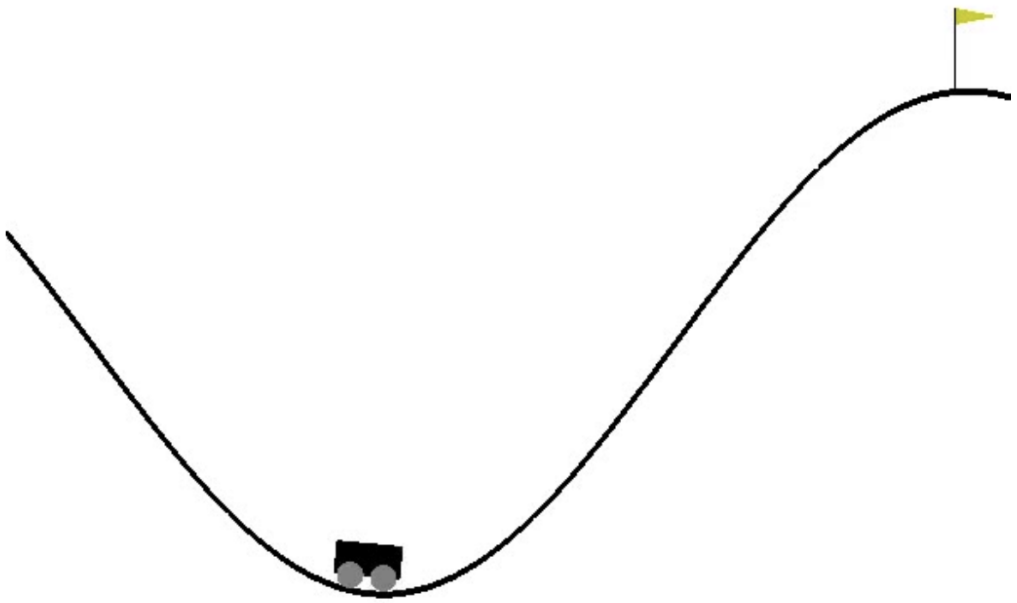
1 frame training scores



4 frames training scores

MOUNTAINCAR-V0

In this case, the same DQN model presented for the Cart-Pole game is used to solve another gym environment: MountainCar. In this setting, a car is on a one dimensional track, positioned between two "mountains". The goal is to drive up the mountain on the right; however, the car's engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum. The environment is represented in the following image:



This problem is modeled using a two-dimensional state space vector that represents the car's current state and a three-dimensional action space vector that describes the actions that the car should do given its current state as stated in the state space vector. The **state space vector** is comprised of the follows:

- **0: Car position** along the x axis with domain in [*-1.2, 0.6*]

- **1: Car velocity** with domain in [*-0.07, 0.07*]

The **actions space vector** is comprised of the follows:

- **0:** Push left
- **1:** No push
- **2:** Push right

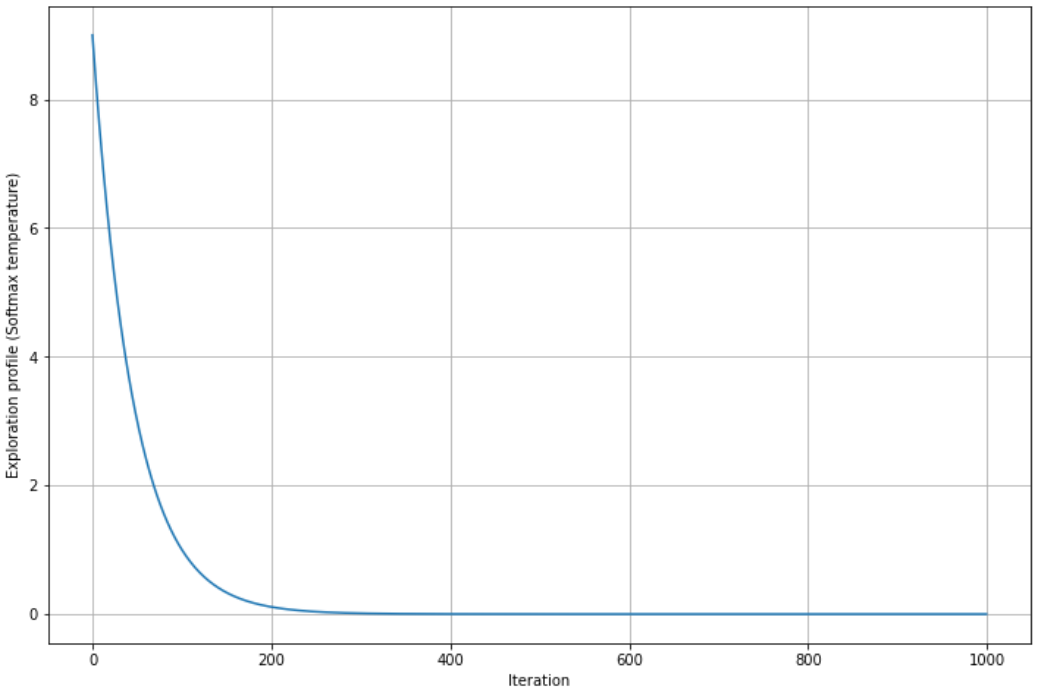
Since the aim of this environment is reached when the car reaches the 0.5 position, the score for this environment equals the car's position. The initial position of the car is the position -0.5, at the bottom of the hill.

The **reward** function is calculated as follows:

Reward = position + 0.5

In order to have a 0 reward in the initial position, the constant 0.5 is added. A +1 reward is added to the existing reward if the car reaches a good position on the hill, for example, when the car reaches a position greater or equal to 0.2. When the car does not reach the maximum position it previously held, a -2 penalty is applied. This penalty stimulates the car to enhance its performance by attempting to obtain a better position. The **episode ends** when the position 0.5 is reached, or if 200 iterations are reached.

The epsilon greedy policy is utilized, with the epsilon value determined by the following function divided by the beginning value of this function, resulting in an epsilon value between 0 and 1. This function is similar to the first task's temperature function, but with different parameters.



Initial=val = 10, num_iterations = 1000, and mul_iter = 3 are the parameters utilized in this case. In the first 200-300 episodes, this function ensures a higher probability of choosing random actions, whereas in later training, it ensures the best actions based on the car's status.

The hyperparameters used to train this environment are:

Gamma	0.97
Replay memory capacity	10000
Learning rate	0.02
Target net update steps	10
Batch size	128
Bad state penalty	0
Min samples for training	1000

The convergence is reached after the 900th episode, as shown in the plot below. After the 500th episode, the model began to achieve the goal, but it had not yet learned how to achieve it much more frequently. The network's convergence speed could be improved with the best parameter optimization.