HOMEWORK 3

Yukselcan Sevil 1216127

Introduction

In this homework the aim is to train and test neural network models to solve Reinforcement Learning problems. The first task requires to implement some extensions to the code that we have seen during the lab lecture. We have to tune the model in order to achieve the same accuracy with fewer training episodes. The second task requires to implement a neural network model to solve the same problem as in the first task but using directly the screen pixels rather than the compact state representation used in the first task. The third task requires to train an RL agent on a different Gym environment. The environment chosen is Mountain Car.

CARTPOLE-V1 ENVIRONMENT

The cartpole-v1 environment consists of a pole attached by an un-actuated joint to a cart which moves along frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided from every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.

This problem is modeled using a **state space vector** of dimension 4 which represents the current state of the cart and the **action space vector** of dimension 2 which represents the actions that the cart should take given its current state described in **state space vector**.
The **state space vector** is composed as 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 composed as follows:

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

The maximum score that the cart can achieve is 500. We aim that our model can learn in order to make the cart achieve the maximum score of 500. The reward of +1 is given for every step that the pole remains upright. Using only this type of reward will cause our agent to move outside of the screen. This is not a behavior we want. We want to see the cart in the screen. To achieve this behavior, we need to add a penalty on the position in order to let the cart stay near the center of the screen.
The **reward** is the following:

$$reward = env\_reward - 1 * |cart\_position|$$

where ***env_reward*** is the +1 reward achieved from the environment and ***|cart_position|*** is the absolute value of the current position of cart in the environment. When the position of the cart is higher in absolute terms, higher is the penalty given.

The environment will run until one of these conditions are reached:

- 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 network architecture used to train in this environment is the following:

- 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)

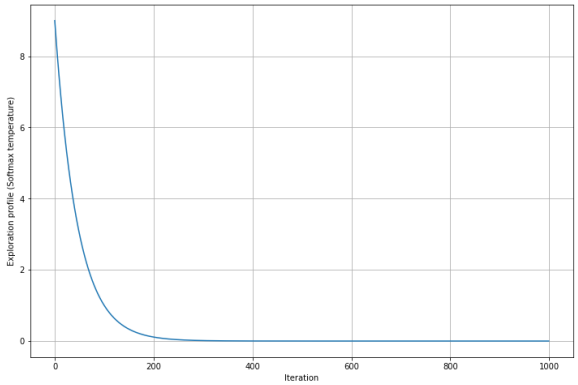
The activation function used is tanh.

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

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

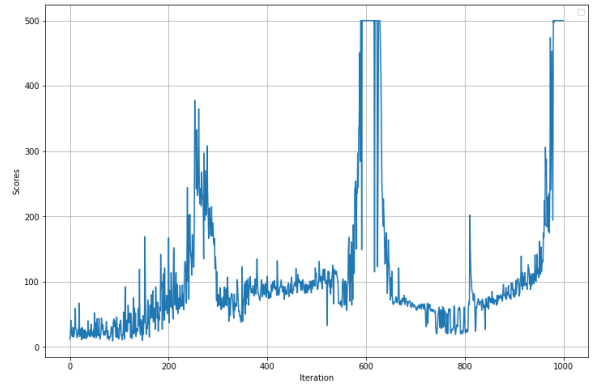
Where ***initial_val = 9, num_iterations = 1000*** and ***mul_iter = 10*** are the hyperparameters tuned for this function.
In the following you can see the plot of this function

The hyperparameters used are:



- **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 lab rather than gamma that is 0.98.
With these parameters, the convergence is faster. The convergence is reached after 600 episodes as you can see in the following figure. However, going forward with the training, the **catastrophic forgetting** issue happens. To avoid this issue, we have to set the number of episodes for training to 600.



CARTPOLE WITH PIXEL SCREEN

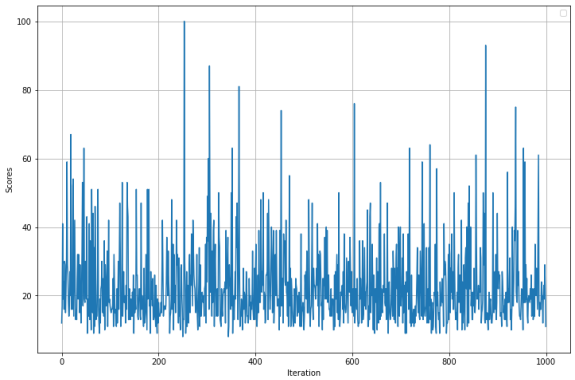
The input of the network in this case is an image frame of the environment. The network is going to predict the action based on the state of the cart in the frame. The image frame obtained from the environment which is 400x600, is rescaled using an average pooling layer with a kernel of dimension 8 which transforms the image of size 50x75. During the training, 4 frame images are grouped together into a single tensor in order to let the network know about the direction of the falling pole. The network so, is trained using a 4 channel tensor where each channel represents a single frame.

The network used is a convolutional neural network which the architecture is the following:

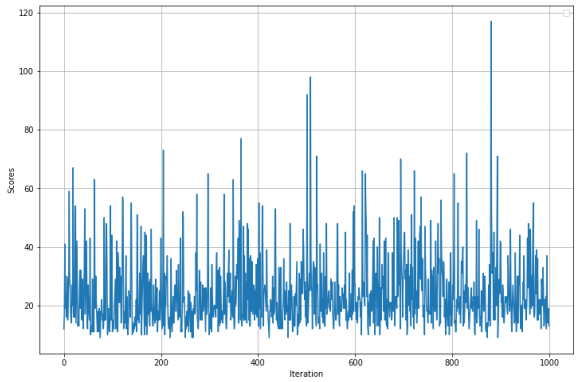
- Conv layer with filter size of 16, kernel size 3, stride 2 and padding 1
- ReLU activation
- Conv layer with filter size of 32, kernel size 3, stride 2 and padding 1
- ReLU activation
- Conv layer with filter size of 64, kernel size 3, stride 2 and padding 1
- ReLU activation
- Conv layer with filter size of 64, kernel size 3, stride 2 and padding 1
- Flatten layer of 1280 neurons
- Linear layer with 640 neurons
- Tanh activation
- Linear layer with 320 neurons
- Tanh activation
- Linear layer with 2 (action state dim) neurons

The learning speed to train this network is very slow, and unfortunately it didn't learn how to reach the goal. It gets stuck with a really low score.

I've tried to stack frames because the normal approach of training using single frame was unsatisfactory. However, also the stacked frames approach implemented here doesn't lead to better results. Unfortunately, the lack of time available made me difficult to try other techniques and/or implement networks studied in academic literature.
In the following you can see the results of both approaches implemented during the homework implementation.



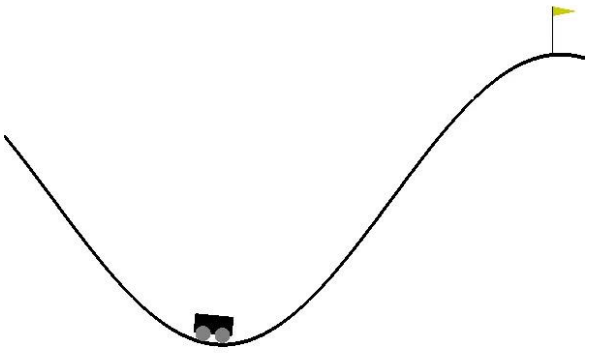
1 frame training score



4 frames training scores

MOUNTAINCAR-V0

This environment consists of a car that 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.
In the following you can see the environment



This problem is modeled using a **state space vector** of dimension 2 which represents the current state of the car and the **action space vector** of dimension 3 which represents the actions that the car should take given its current state described in **state space vector**.
The **state space vector** is composed as 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 composed as follows:

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

The **score** of this environment is the car's position since the **goal** of this environment is achieved when the car goes to the 0.5 position.

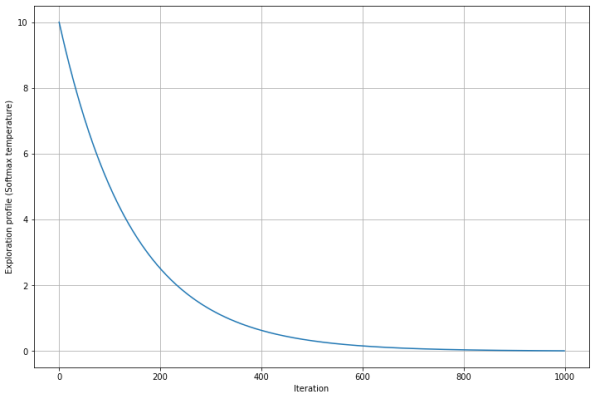
The **initial** position of the car is the position -0.5, at the bottom of the hill.
The **reward** function is calculated as follows:

Reward = position + 0.5

Where the constant 0.5 is added in order to have a 0 reward in the initial position.
The car is rewarded also if it reaches a good position in the hill, for instance, when the car reaches a position greater or equal than 0.2, a +1 reward is added to the current reward.
A penalty of -2 is added when the car doesn't achieve the maximum position that it reached previously. This penalty is useful to stimulate the car to improve itself by trying to reach a better position.

The **episode ends** when the position 0.5 is reached, or if 200 iterations are reached.

The policy used is the epsilon greedy policy with the epsilon value following the following function divided by the initial value of this function in order to have the **epsilon** value between 0 and 1. This function is similar to the temperature function of the first task but with different parameters.



In this case the parameters used are ***initial_val = 10, num_iterations = 1000*** and ***mul_iter = 3.*** This function ensures higher probabilities to choose random actions in the first 200-300 episodes and the best actions based on car's state in the later training.

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

As you can see in the following plot, the convergence is reached after the 900th episode. The model started to achieve the **goal** after the 500th episode but it didn't yet learn how to achieve the goal much more frequently. Best parameter optimization could improve the convergence speed of the network.