**Ivancich Stefano 1227846**

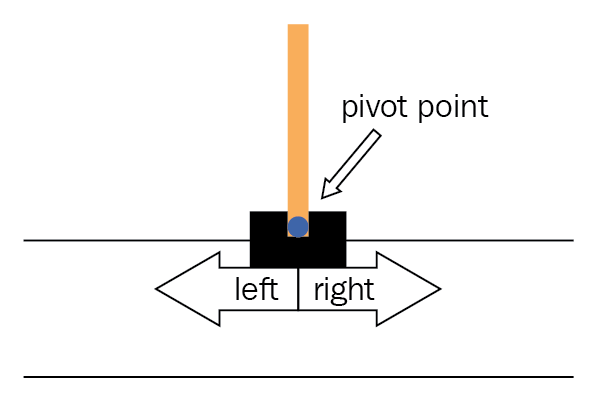
**Neural Network and deep learning course 2020/21**

**Homework 3**

1. **Introduction**

In this homework we have to implement and test…..

The ***CartPole-v1* environment** consist of a pole that is attached by an un-actuated joint to a cart, which moves along a 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 by increasing and reducing the cart's velocity. A reward of +1 is provided for 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.



**State space vector:**

* 0: Cart Position
* 1: Cart Velocity
* 2: Pole Angle
* 3: Pole Velocity At Tip

**Actions space vector:**

* 0: Push cart to the left
* 1: Push cart to the right

Note: The amount the velocity is reduced or increased is not fixed as it depends on the angle the pole is pointing. This is because the center of gravity of the pole increases the amount of energy needed to move the cart underneath it

**Reward:** Reward is 1 for every step taken, including the termination step. The threshold is 475.

….. we added

# We apply a (linear) penalty when the cart is far from center

pos\_weight = 1

reward = reward - pos\_weight \* np.abs(state[0])

**Starting State:** All observations are assigned a uniform random value between ±0.05.

**Episode Termination:**

* Pole Angle is more than
* Cart Position is more than (center of the cart reaches the edge of the display)
* Episode length is greater than 500

1. **Speeding up learning Convergence**

…..

In this case we will use the Huber loss as loss function (https://pytorch.org/docs/stable/generated/torch.nn.SmoothL1Loss.html). The Huber loss uses a squared term if the absolute element-wise error falls below beta and an L1 term otherwise. It is less sensitive to outliers than the MSELoss and in some cases prevents exploding gradients.

To overcome the exploration-exploitation dilemma, we will be using the epsilon-greedy approach to slowly decrease the randomization factor overtime. This will ensure that our agent will have a wide variety of state-action training samples and in the later part of the training, it will allow the agent to follow it’s own “trained strategy” as opposed to random actions.

Technically, in the code, we will be using a temperature term to smooth the probability of actions, and epsilon to decide between whether to take a random action or the predicted action output from the policy.

gamma = 0.97 # gamma parameter for the long term reward

replay\_memory\_capacity = 10000 # Replay memory capacity

lr = 1e-2 # Optimizer learning rate

target\_net\_update\_steps = 10 # Number of episodes to wait before updating the target network

batch\_size = **256** # Number of samples to take from the replay memory for each update

bad\_state\_penalty = 0 # Penalty to the reward when we are in a bad state (in this case when the pole falls down)

min\_samples\_for\_training = 1000 # Minimum samples in the replay memory to enable the training

REACH 500 at: 790 episodes

####################################################à

gamma = 0.97 # gamma parameter for the long term reward

replay\_memory\_capacity = 10000 # Replay memory capacity

lr = 1e-2 # Optimizer learning rate

target\_net\_update\_steps = 10 # Number of episodes to wait before updating the target network

batch\_size = **256** # Number of samples to take from the replay memory for each update

bad\_state\_penalty **= 0.1** # Penalty to the reward when we are in a bad state (in this case when the pole falls down)

min\_samples\_for\_training = 1000 # Minimum samples in the replay memory to enable the training

REACH 500 at: 800

#####################################

gamma = **0.9** # gamma parameter for the long term reward

replay\_memory\_capacity = 10000 # Replay memory capacity

lr = 1e-2 # Optimizer learning rate

target\_net\_update\_steps = 10 # Number of episodes to wait before updating the target network

batch\_size = 128 # Number of samples to take from the replay memory for each update

bad\_state\_penalty = 0 # Penalty to the reward when we are in a bad state (in this case when the pole falls down)

min\_samples\_for\_training = 1000 # Minimum samples in the replay memory to enable the training

REACH 500 at: never

#####################################

gamma = **0.98** # gamma parameter for the long term reward

replay\_memory\_capacity = 10000 # Replay memory capacity

lr = 1e-2 # Optimizer learning rate

target\_net\_update\_steps = 10 # Number of episodes to wait before updating the target network

batch\_size = 128 # Number of samples to take from the replay memory for each update

bad\_state\_penalty = 0 # Penalty to the reward when we are in a bad state (in this case when the pole falls down)

min\_samples\_for\_training = 1000 # Minimum samples in the replay memory to enable the training

REACH 500 at: 720

#####################################

1. **Control CartPole using the screen pixels**

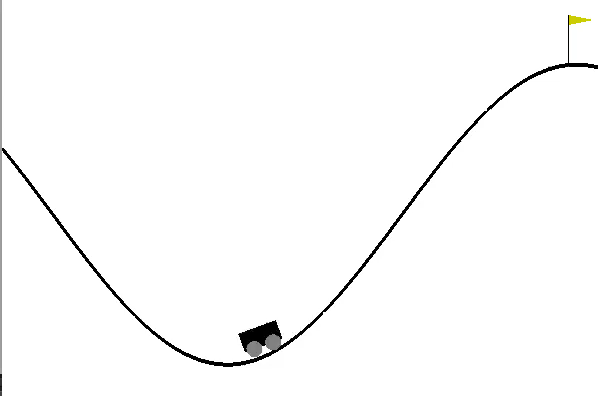
…..

To get the state in the previous approach we used: next\_state, reward, done, info = env.step(action) but now we will ignore the next\_state variable and use as state the 800x600x3 tensor returned by env.render(mode='rgb\_array')

…

1. **MountainCar-v0 Gym Environment**

The ***MountainCar-v0* environment** consist of a car running on a hill, that can only choose to push left, push right or do nothing. The objective is to get the car to the top of the hill (top = 0.5 position).



**State space vector:**

* 0: position
* 1: velocity

**Action space vector:**

* 0: push left
* 1: no push
* 2: push right

**Reward:** -1 for each time step, until the goal position of 0.5 is reached. There is no penalty for climbing the left hill, which upon reached acts as a wall.

BUT we decided to create our own reward function based on the position and direction (velocity) of the car.

pos\_weight = 6

if (action ==0 and state[1]<0) or (action==2 and state[1]>0):

reward= reward + pos\_weight \* np.abs(state[0]+0.5)

else:reward= reward-2

**Starting State:** Random position from -0.6 to -0.4 with no velocity.

**Episode Termination:** when you reach 0.5 position, or if 200 iterations are reached.

The training procedure is exactly the same used in the *CartPole-v1* environment except for the reward function, and the score that here we define as the final position of the car.

* 2 pt: extend the notebook used in Lab 07, in order to study how the exploration profile (either using eps-greedy or softmax) impacts the learning curve. Try to tune the model hyperparameters or tweak the reward function in order to speed-up learning convergence (i.e., reach the same accuracy with fewer training episodes).
* 3 pt: extend the notebook used in Lab 07, in order to learn to control the CartPole environment using directly the screen pixels, rather than the compact state representation used during the Lab (cart position, cart velocity, pole angle, pole angular velocity). This will require to change the “observation\_space”.
* 3 pt: train a deep RL agent on a different Gym environment. You are free to choose whatever Gym environment you like from the available list, or even explore other simulation platforms: https://gym.openai.com/envs