CIS 522

Homework 5: Reinforcement Learning

4/11/2020

Part 1: Value Iteration with Grid-World (15 pts)

Q1a (5 pts)

What do each of these variables mean? How big is your action space and what does each action value represent (i.e. what action does a 0 represent in the action space)? How big is your observation space? What does this mean? Answer all these questions in your writeup.
Answer:

1. Observation Space: This defines the structure of our state space. For example, in this case we have $4x12$ states. By definition of observation space in this environment, the state observed by an agent will be a tuple of two numbers of for each state where the first element will take 4 discrete values(0,1,2,3) and second element will take 12 discrete values(0 to 11 integer values).

2. Action Space: This defines the structure of our action space. In our environment, action space is $Discrete(4)$, which means each action is defined by a non-negative integer from 0 to 3. so each action will take a value among 0,1,2,3.

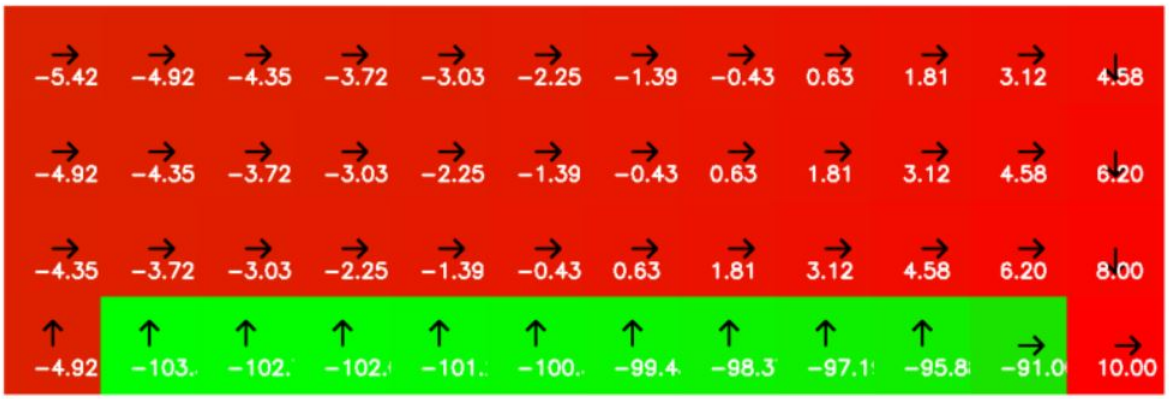
3. env: env is the instance of our environment class created to run to simulations.

In our action space, we have 4 discrete actions each of them taking a value 0,1,2,3 where **0 represents moving up**, **1 represents moving right**,**2 represents moving down**,**3 represents moving left**.
Our observation space totally has 48 states. Each of these states are represented by the current coordinates of the agent in the grid world. In our environment, the observation space is (Discrete(4).Discrete(12)). This means that the total grid world is of height 4 and width 12. So a state of(2,10) means that the agent is in the grid square whose location is (2,10).

Q1b (10 pts)

Put the image of your learned policy / value function in your writeup and explain the found policy. Now, using your optimal found policy, run your optimal policy function with the simulator – did it reach the goal state? Explain in your writeup.

Answer:



The goal for the agent is to reach the goal state in shortest path possible with the available actions. Let's say the agent starts at some arbitrary point which is not the goal state. Then it takes the minimal number of actions to reach the goal state. Thus from the predicted policy we see that the policy is to move right to the column containing goal state except for the that column where the agent's optimal policy changes to moving down that column and at state (3,0), it moves up to avoid the cliff and starts moving right as expected. Once the agent reaches the goal, it should take an action that doesn't make it enter into any other state of the environment, thus it should either move down or right. In this case, is moves right. When I ran the agent using the optimal policy it did reach the goal state. The agent started from (3,0) and followed optimal policy and reached end state as

```
[ ]  env.reset()
     done = False
     while done==False:
       action = policy_table[env.S[0]][env.S[1]]
       print('state: ',env.S,' action:',action)
       _,_,done,_ = env.step(action)
```

```
state:  (3, 0)  action: 0
state:  (2, 0)  action: 1
state:  (2, 1)  action: 1
state:  (2, 2)  action: 1
state:  (2, 3)  action: 1
state:  (2, 4)  action: 1
state:  (2, 5)  action: 1
state:  (2, 6)  action: 1
state:  (2, 7)  action: 1
state:  (2, 8)  action: 1
state:  (2, 9)  action: 1
state:  (2, 10)  action: 1
state:  (2, 11)  action: 2
```

shown in the snippet below:

Part 2: Setting up Deep Q-Learning with Cart Pole (35 pts)

Q2a (5 pts)

What do each of variables mean? What are the individual components of the observation space? In the writeup, explain what each of the four elements of the observation space are and what their bounds are. Explain the environment's action space in the writeup as well.

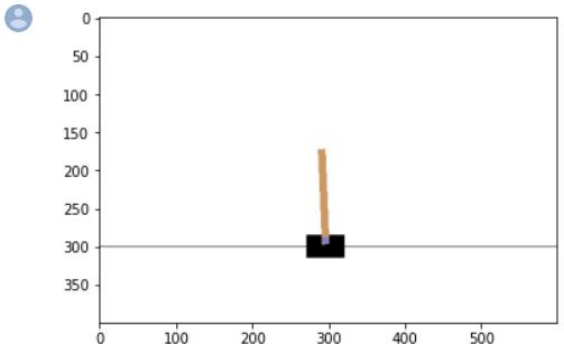
The env which contains information about the CartPole environment has 2 spaces - **the observation space and the action space.** The **observation space** is of **'Box'** type which represents an n-dimensional box with valid observations in an array of n numbers. The 'Box' type has an upper and a lower bound. The **action space** is a **Discrete** space which allows a fixed range for non-negative numbers, so in this case valid actions are either 0 or 1.

In the case of Cartpole problem, we have 4 **observation spaces** (Cart position, Cart velocity, Pole angle, Pole velocity at tip). The upper and lower bound values for them are as follows.

```
print(env.observation_space)
print(env.observation_space.high)
print(env.observation_space.low)
print(env.action_space)
```

```
Box(4,)
[4.8000002e+00 3.4028235e+38 4.1887903e-01 3.4028235e+38]
[-4.8000002e+00 -3.4028235e+38 -4.1887903e-01 -3.4028235e+38]
Discrete(2)
```

```
[ ] import matplotlib.pyplot as plt
    env = gym.make('CartPole-v0')
    state = env.reset()
    state
    img = plt.imshow(env.render('rgb_array'))
```



Observation space: Cart position= -4.8 to 4.8 **Cart velocity** = -infinity to infinity **Pole angle** = -41.8 degrees to 41.8 degrees. **Pole velocity at tip** = -infinity to infinity

The 4 variables define the state space of the CartPole and the goal is to balance the cartpole stay upright by applying 2 actions- applying force to the left or to the right.

Action space: The two actions 0 and 1 make the cart push to the left and the right.

Q2b (5 pts)

In your writeup describe the reward function for this problem. How much is the actor rewarded on each timestep of an episode, and under what conditions does it occur?

A reward of +1 is provided for every timestep that the pole remains upright including the termination step. The threshold is 475 for CartPole-v1. The episode ends when one of the following points take place.

1. Pole Angle is more than $\pm 12^{\circ}$

2. Cart Position is more than ± 2.4 (center of the cart reaches the edge of the display)

3. Episode length is greater than 200 (500 for v1).

CartPole-v0 defines "solving" as getting average reward of 195.0 over 100 consecutive trials. The pendulum starts upright, and the goal is to prevent it from falling over by increasing and reducing the cart's velocity.

Q2c (10 pts)

Using only the values below, code up the loss function for a Q-Learning algorithm in PyTorch specifically with respect to the Cart-Pole problem. Submit a screenshot of your implementation in your writeup as well as an explanation.

```
  current_qvalues = current_qvalues.gather(1,action.unsqueeze(1))
  next_qvalues = model(next_state).detach()
  next_qvalues, _ = torch.max(next_qvalues,1)
  next_qvalues[reward==0] = 0
  loss = calculate_loss(GAMMA, reward, current_qvalues, next_qvalues)
```

```
def calculate_loss(gamma, reward, Q_t, Q_next):
  """ Method header to calculate the loss.
  NOTE: THIS CODE SNIPPET WILL NOT WORK RIGHT NOW, BUT SHOULD ONCE YOU HAVE
  EVERYTHING ELSE IMPLEMENTED

  Args:
      gamma  (float) : Discount for rewards
      reward (int)   : Reward for action taken
      Q_t    (tensor): Tensor of Q-Values for each action at time t
      Q_next (tensor): Tensor of Q-Values for each action at time t+1
  """

  # TODO: Insert your code here
  loss= F.smooth_l1_loss(reward + gamma*Q_next, Q_t.squeeze())
  return loss
```

The state,reward,action,next-state values are sampled from the Replay Buffer. The state values are passed in to the DQN Network and Q-values are obtained as outputs. These are the Q-values corresponding to the current state. The q-values corresponding to the action from the Replay Buffer which was selected through epsilon greedy policy are gathered. The image below shows how the Q-values are being gathered.

```
before tensor([[6.7474, 6.6521],
        [6.6147, 6.5755],
        [6.3776, 6.4240],
        [5.7820, 6.0347],
        [6.5647, 6.5126],
        [5.9405, 6.4757],
        [6.3041, 6.6221],
        [6.5388, 6.5711],
        [6.0500, 6.5822],
        [6.2510, 6.2667],
        [6.5328, 6.5876],
        [6.2988, 6.3996],
        [6.4718, 6.7185],
        [6.1671, 6.2787],
        [5.9199, 6.1039],
        [6.6195, 6.3669],
        [5.8312, 6.0852],
        [6.9988, 6.7208],
        [6.5434, 6.5069],
        [6.0593, 6.4711]], device='cuda:0', grad_fn=<AddmmBackward>)
torch.Size([20, 2])
after tensor([[6.6521],
        [6.6147],
        [6.3776],
        [5.7820],
        [6.5126],
        [6.4757],
        [6.3041],
        [6.5711],
        [6.0500],
        [6.2667],
        [6.5876],
        [6.3996],
        [6.7185],
        [6.1671],
        [6.1039],
        [6.3669],
        [6.0852],
        [6.7208],
        [6.5069],
        [6.4711]], device='cuda:0', grad_fn=<GatherBackward>)
```

The q-values for the next state are got from the model and detached. The maximum q-values in the tensor are selected for the next state and the states where there are no rewards are zeroed out. Then the loss is calculated using Huber loss(smooth L1 loss)

$$\text{loss}(x, y) = \begin{cases} 0.5(x - y)^2, & \text{if } |x - y| < 1 \\ |x - y| - 0.5, & \text{otherwise} \end{cases}$$

$$((r_t + \gamma \max_a Q(s_{t+1}, a)) - Q(s_t, a_t))^2$$

where the first part (reward+gamma*qnext) is 'x' and second part (qcurrent) is 'y'. Huber L1 loss was chosen as they generally perform well for most of the problems,especially RL where feature have large values. The improvement in rewards were seen when loss function was switched from MSE to smooth.L1 loss.

Q2d (5 pts)

Implement the neural network to approximate the Q function below. In your writeup explain what the inputs and outputs are for your neural network.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
class DQNetwork(nn.Module):
    """ Template class for your DQN. Please feel free to modify headers as needed.
    """
    def __init__(self):
      torch.manual_seed(5)
      super(DQNetwork, self).__init__()
      self.fc1 = nn.Linear(4,64)
      self.fc2 = nn.Linear(64,2)

    def forward(self,x):
      x = F.elu(self.fc1(x))
      x = self.fc2(x)
      return x
```

The neural network which we used to solve the CartPole problem was a simple 2 layer Fully Connected Network with only linear layers alone. The **inputs** to the network are 4 state parameters (cart position, velocity, pole angle, pole velocity at tip). The 2 **output** parameters are the 2 q values corresponding to the actions (force applied to left or to right).
The network for this model was made very small as Cartpole is a relatively simple problem and using a deep network for this task can take longer than necessary to converge. Also, there was no need to use Convolutional layers as there were no images involved and to avoid network complexity.
Initially, ReLu was used as activation function and when replaced by ELU, the convergence was very fast and the target of 175 rewards averaged over 10 episodes were reached before 200 episodes. torch.manual-seed() is added to fix the random generator to a particular value to avoid random initialisation.

Q2e (5 pts)

Implement a ReplayBuffer below as a FIFO queue of maximum capacity that you initialize the buffer with. You should be able to push a new memory to the replay buffer, and also randomly sample a batch from the ReplayBuffer. In the writeup, describe your tuple "replay" in the push function. What are the elements of your tuple? Why?

```
class ReplayBuffer:
  """ ReplayBuffer is a Buffer that allows us to implement Memory Replay.
      Feel free to modify the method signatures as you like, this is simply
      how we implemented it.
  """
  def __init__(self, capacity=100):
    """ Initializes the ReplayBuffer
    Args:
      capacity (int): The maximum number of memories stored in the ReplayBuffer
    """
    self.memory = []
    self.capacity = capacity
    self.position = 0


  def push(self, replay):
    """ Pushes a new memory to ReplayBuffer. If at capacity, removes first
    memory stored (FIFO).

    Args:
      replay (Tuple): The tuple of all pertinent information for the replay
    """
    if self.position == self.capacity:
      self.memory.pop(0)
      self.position -= 1

    self.memory.append(replay)
    self.position += 1

  def sample(self, batch_size):
    """ Randomly samples a batch of size 'batch_size' from the ReplayBuffer

    Args:
      batch_size (int): Integer representing the desired batch size to be sampled

    Returns:
      Tuple representing the replay

    Hint: You can use random.sample to do this really easily
    """
    return random.sample(self.memory, batch_size)

  def __len__(self):
    """
    Returns:
      (int) Returns the current length of the Replay Buffer
    """
    return self.position
```

The tuple 'replay' consists of 4 elements (state, action, next state,reward). This tuple is appended every time to the 'memory' list when the push() function is called and when the position of the list is filled ie equal to capacity, then the first stored memory is popped out. The sample

function randomly samples a batch from the memory for training.

The reason why a Replay Memory with a tuple of trajectories added every time is because converging Q learning using one sample at a time is not very effective. Adding a Replay memory stabilizes and improves the DQN model and makes the model smart and robust. It also makes sure previous experiences are not thrown and are considered for the Q-learning algorithm. By sampling from it randomly, the transitions from a batch are decorrelated enhancing learning from more past experience.

Q2f (5 pts)

If you have simulated $N = 10000$ steps, and have a ε of .99, what is the expected number of random actions that you will take? What about if ε is .9 instead? How many more random actions will you take with the latter value of ε than the former. Explain how varying this parameter changes how the agent learns.

Answer:

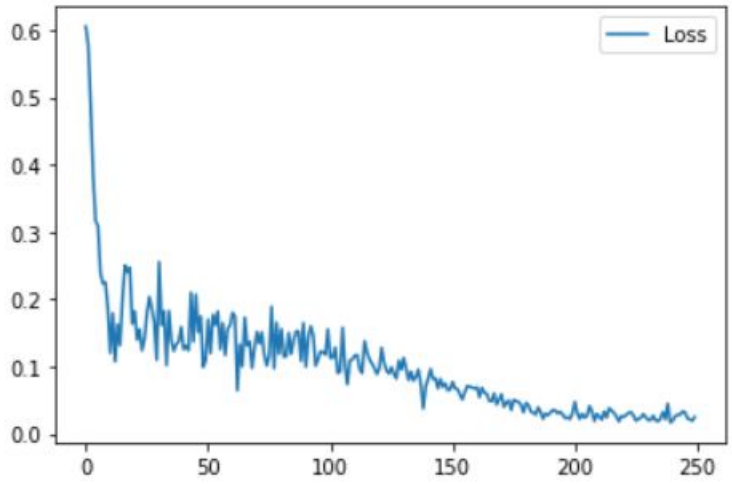
If ε is 0.99, then the expected number of random actions is 100. If ε is 0.9, then the expected number of random actions is 1000. The number of random actions taken in second case will be 900 actions more than the number of random actions taken in first case. If ε is small we the agent takes a lot of random action. Thus it explores a lot by taking actions randomly. So, convergence is slow. But if epsilon is high, then the number of random actions taken by the agent will be less. The agent learns with optimal policy faster with exploring other actions which may be more optimal. Thus, constricting the learning done by the agent. An optimal epsilon would be the one which gives the agent the freedom to explore in the beginning and increases with time and make the agent exploit its found optimal policy and making it choose lesser random actions with the passage of time.

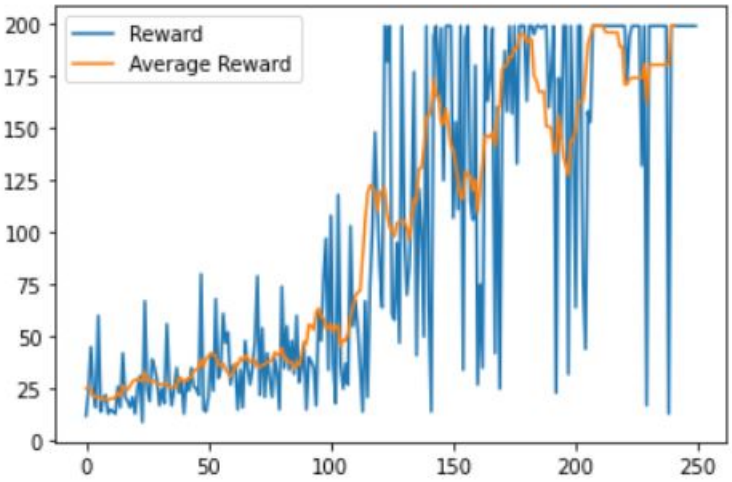
Part 3: Running Deep Q-Learning with Cart Pole (50 pts)

Q3a (30 pts)

In your writeup report your loss graph (you can either choose to plot the loss per step of each episode, or the average loss over an entire episode – the former will be more noisy). Also in your writeup report a plot that shows the total reward obtained per episode. Also report your average reward over your final 10 episodes of training.

Last 10 episodes: Avg reward- 180.4





The average loss for an episode is plotted and the rewards and the average rewards for 10 episodes are plotted in the second graph. It can be seen that the model is able to reach 175 threshold in episodes after 180 and the model was able to maintain over 175 constantly after episode 220.

```
Episode no:  0 Avg reward over 10 episodes 1.2
Episode no:  10 Avg reward over 10 episodes 25.4
Episode no:  20 Avg reward over 10 episodes 20.1
Episode no:  30 Avg reward over 10 episodes 29.6
Episode no:  40 Avg reward over 10 episodes 26.9
Episode no:  50 Avg reward over 10 episodes 29.1
Episode no:  60 Avg reward over 10 episodes 42.1
Episode no:  70 Avg reward over 10 episodes 37.2
Episode no:  80 Avg reward over 10 episodes 35.1
Episode no:  90 Avg reward over 10 episodes 38.8
Episode no:  100 Avg reward over 10 episodes 55.7
Episode no:  110 Avg reward over 10 episodes 52.2
Episode no:  120 Avg reward over 10 episodes 71.2
Episode no:  130 Avg reward over 10 episodes 120.4
Episode no:  140 Avg reward over 10 episodes 102.0
Episode no:  150 Avg reward over 10 episodes 159.7
Episode no:  160 Avg reward over 10 episodes 131.7
Episode no:  170 Avg reward over 10 episodes 122.7
Episode no:  180 Avg reward over 10 episodes 179.3
Episode no:  190 Avg reward over 10 episodes 192.5
Episode no:  200 Avg reward over 10 episodes 138.4
Episode no:  210 Avg reward over 10 episodes 162.6
Episode no:  220 Avg reward over 10 episodes 199.0
Episode no:  230 Avg reward over 10 episodes 171.0
Episode no:  240 Avg reward over 10 episodes 180.4
```

The average of the last 10 episodes were 180.4

Q3b (10 pts)

In the writeup explain how your process for tuning hyperparameters as well as what your final hyperparameters are.

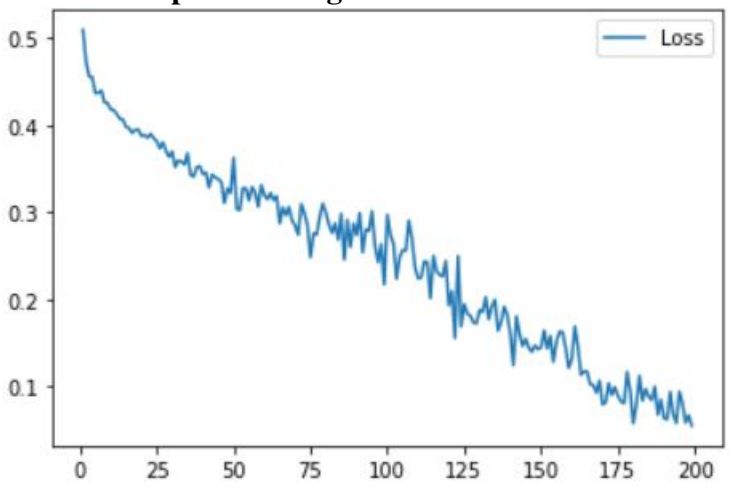
- The DQN network was made simple with just 2 linear layers as the Cartpole problem was a relatively simple problem and the model wasn't required to be too deep.
- ELU was used as the activation function. Initially, ReLu was used but the rewards generated weren't up to the mark and was replaced by ELU.
- Adam optimizer was used and after fine-tuning, the learning rate was fixed to 0.001.
- The replay buffer was initialized with a capacity of 10,000.
- Huber loss(smooth L1 loss) was used for the loss function. Initially, MSE Loss was used and using smooth L1 loss generated better convergence.
- Gamma value or the Discount factor was fixed to 0.8.

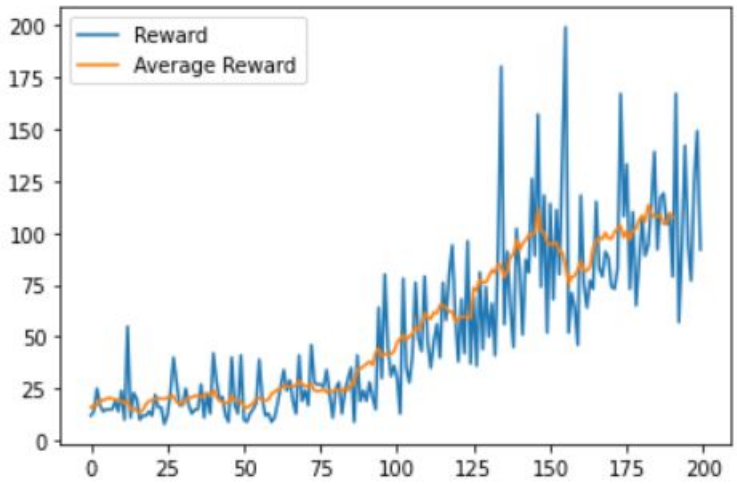
- Epsilon was initially made constant but using epsilon decay provided better convergence. This was because initially, the model requires exploration as it doesn't have much information about the environment. After gathering enough information, the model should start exploiting and finding the optimal Q-value rather than exploration. So, the epsilon was made to decay over time-steps with a decay rate of 200 starting from 0.05 to 0.9. In our case, if the random probability is more than the epsilon value, a random action is selected or else the action with the highest Q-value(greedy) is selected.
- The number of episodes trained was 250.
- The sample size was made to 20.

Q3c (10 pts)

Remember how in step 4 of the learn function you detached the output from the previous step? Try not detaching it instead. Report the new loss / reward curves in the writeup. Also in your writeup, intuitively explain why this occurs / why it is therefore necessary to detach the output.

Last 10 episodes: Avg reward- 105.5





The detach() function detaches the output from the computational graph and hence no gradient will be backpropagated along this variable and thus disables automatic differentiation for it and stops keep tracking of the gradients. Once, the detach() is removed from the next state, the model learns slowly as seen from the graph and was able to reach only 105 in 200 episodes whereas in the previous model, it reached over 175 by episode 180 itself.

This is because the gradient of the model is computed only through the online parameters ie the current state and resulting in action selection from the q-value through epsilon greedy. The next state parameters are calculated from the offline parameters as next state is obtained from the step() function. So, the generation of the q-value for the next state here play the role of a target network which are calculated offline and to avoid updating the target network, the next state values are detached thus freezing the update. The target network should be made to learn offline and this learning process should not be disrupted by being updated online through back-propagation as this enables more accurate learning only from the offline inputs and thereby faster convergence.

Even without detaching, the model may converge but it'll take a lot of episodes and is not an optimal learning algorithm.