RUTGERS UNIVERSITY
School of Engineering
Department of Electrical & Computer Engineering

ECE 472 – Robotics & Computer Vision– Fall 2022

Project 2 - Reinforcement Learning

Name (last, first) : Mehmet Ali Soner

netID : mas996

RUID: 196000499

Date: November 26, 2022

1 Problem 1

Cart Pole code assignment, code + explanation

I have structured the code into two main parts: a part before and after training. The part before training is mainly to test out components such as the video capturing of an episode, the overall environment and the functions built within. First we call the following terminal commands:

```
!apt-get update
!apt-get install -y xvfb x11-utils
%pip install pyvirtualdisplay==0.2.*
%pip install gym[classic_control]
```

I am calling apt-get update since I ran into issues with xvfb couple of times and this seem to resolve it. We install xvfb and pyvirtualdisplay for the video capturing. Then we import the gym package and create the environment for "CartPole":

```
import gym
if gym.__version__ < '0.26':
    env = gym.make('CartPole-v0', new_step_api=True,
    ↪  render_mode='single_rgb_array').unwrapped
else:
    env = gym.make('CartPole-v0', render_mode='rgb_array').unwrapped
```

After this, I will skip few things such as the video capturing and importing other various packages. I want to focus on the DQN's linear layer. In this model, we have to have two possible outputs in our final/linear layer since we will either push the cart to the right or to the left. In order to add this layer, we have to know the output size of each convolutional layer so that we have the right input size for the linear layer. That's why we call conv2d_size_out three times since we have three convolutional layers. We then calculate the input size and add it with:

```
class DQN(nn.Module):
# Architecture code (3 conv layers, 3 batch norms)
...
        def conv2d_size_out(size, kernel_size = 5, stride = 2):
            return (size - (kernel_size - 1) - 1) // stride  + 1
        convw = conv2d_size_out(conv2d_size_out(conv2d_size_out(w)))
        convh = conv2d_size_out(conv2d_size_out(conv2d_size_out(h)))
        linear_input_size = convw * convh * 32
        self.head = nn.Linear(linear_input_size, outputs)
```

We then implement two functions that will get the cart's location on the screen and will get the screend itself:

```
def get_cart_location(screen_width):
world_width = env.x_threshold * 2
    scale = screen_width / world_width
    return int(env.state[0] * scale + screen_width / 2.0)  # MIDDLE OF
    ↪  CART

def get_screen():
screen = env.render().transpose((2, 0, 1))
    # Cart is in the lower half, so strip off the top and bottom of the
    ↪  screen
    _, screen_height, screen_width = screen.shape
...
cart_location = get_cart_location(screen_width)
...
```

```
screen = screen[:, :, slice_range]
```

I have mainly included the important lines of the two functions in the snippet. These two functions are crucial since we can describe the current state of the agent through the location of the cart on the screen. We achieve this with some algebra and the information we have about the environment's/simulation's dimensions.

Now we get to the part where we train the DQN. First we initialize the network, the optimization function and the memory.

```
n_actions = env.action_space.n
policy_net = DQN(screen_height, screen_width, n_actions).to(device)
target_net = DQN(screen_height, screen_width, n_actions).to(device)
target_net.load_state_dict(policy_net.state_dict())
target_net.eval()

optimizer = optim.RMSprop(policy_net.parameters())
memory = ReplayMemory(10000)
```

n_actions is the amount of actions we can take in the environment. In this case, it will be two (move cart to right or left). Then, we initialize the policy and the target network. This is mainly for stability since it servers as a "back-up" of the policy network. The target network is basically a copy of the policy network with frozen parameters that get updated once in a while (we copy the parameters into target_net on line 5 and put the network in eval mode on line 6). So, if overfitting or other errors were to happen in the policy network, the target network is there for stability. On line 6, we initialize the memory by creating a ReplayMemory object. The ReplayMemory class looks like this:

```
Transition = namedtuple('Transition',
                        ('state', 'action', 'next_state', 'reward'))
class ReplayMemory(object):

    def __init__(self, capacity):
        self.memory = deque([],maxlen=capacity)

    def push(self, *args):
        """Save a transition"""
        self.memory.append(Transition(*args))

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)
```

The ReplayMemory class can hold Transition and can return random Transitions of a certain batch size. This is crucial since the policy network will learn from recent Transition tuples in the ReplayMemory object. It will use that information to optimize the model, to achieve a better policy. So, the memory object serves as the memory for the training.

Next, I will explain some details in the select_action() and optimize_model() function. The way we choose an action is the following:

```
def select_action(state):
...
    sample = random.random()
    eps_threshold = EPS_END + (EPS_START - EPS_END) * \
```

```
        math.exp(-1. * steps_done / EPS_DECAY)
    steps_done += 1
    if sample > eps_threshold:
        with torch.no_grad():
            return policy_net(state).max(1)[1].view(1, 1)
    else:
        return torch.tensor([[random.randrange(n_actions)]],
         ↪  device=device, dtype=torch.long)
```

Given an input state, we select the action. First, we get a random value and compare it to our epsilon threshold. If this random value is over the threshold, we go through the policy net with the input state and take the action with the highest reward. If the random value doesn't cross the threshold, we then select a random action out of the possible actions in this environment (right, left). With this function, we can now define the model optimization function.

The optimize_model() function is where the RL algorithm is implemented. First, I will explain how we "load" the necessary data.

```
def optimize_model():
    if len(memory) < BATCH_SIZE:
        return
    transitions = memory.sample(BATCH_SIZE)
    # Transpose the batch (see https://stackoverflow.com/a/19343/3343043
     ↪  for
    # detailed explanation). This converts batch-array of Transitions
    # to Transition of batch-arrays.
    batch = Transition(*zip(*transitions))
```

2 Problem 2

Explain DQN algorithm in paragraphs, include definitions of state, action,environment, reward.

Notes: Agents and environment: agent(s) interacts with environment, which can be a simulation, such as the cart pole example. Each step, the agent observes the environment (state of environment) and takes action and receives a reward based on that. Agents learn from repeated trials; these are called episodes. RL framework trains a policy for the agent to follow. Policy shows which actions to take one after the other in order to maximize reward.

In RL, want to train the agent to make better decision or act better with each episode. Policy == neural network.

3 Problem 3

Performance metrics and plots

4 Problem 4

Three other problems for RL; state, action, environment and reward for each