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:

```
1  !apt-get update
2  !apt-get install -y xvfb x11-utils
3  %pip install pyvirtualdisplay==0.2.*
4  %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":

```
1  import gym
2  if gym.__version__ < '0.26':
3      env = gym.make('CartPole-v0', new_step_api=True,
         ↪  render_mode='single_rgb_array').unwrapped
4  else:
5      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:

```
1  class DQN(nn.Module):
2  # Architecture code (3 conv layers, 3 batch norms)
3  ...
4          def conv2d_size_out(size, kernel_size = 5, stride = 2):
5              return (size - (kernel_size - 1) - 1) // stride  + 1
6          convw = conv2d_size_out(conv2d_size_out(conv2d_size_out(w)))
7          convh = conv2d_size_out(conv2d_size_out(conv2d_size_out(h)))
8          linear_input_size = convw * convh * 32
9          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:

```
1  def get_cart_location(screen_width):
2  world_width = env.x_threshold * 2
3      scale = screen_width / world_width
4      return int(env.state[0] * scale + screen_width / 2.0)  # MIDDLE OF
         ↪  CART
5
6  def get_screen():
7  screen = env.render().transpose((2, 0, 1))
8      # Cart is in the lower half, so strip off the top and bottom of the
         ↪  screen
9      _, screen_height, screen_width = screen.shape
10  ...
11  cart_location = get_cart_location(screen_width)
12  ...
```

```
13   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.

```
1   n_actions = env.action_space.n
2   policy_net = DQN(screen_height, screen_width, n_actions).to(device)
3   target_net = DQN(screen_height, screen_width, n_actions).to(device)
4   target_net.load_state_dict(policy_net.state_dict())
5   target_net.eval()
6
7   optimizer = optim.RMSprop(policy_net.parameters())
8   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:

```
1   Transition = namedtuple('Transition',
2                           ('state', 'action', 'next_state', 'reward'))
3   class ReplayMemory(object):
4
5       def __init__(self, capacity):
6           self.memory = deque([],maxlen=capacity)
7
8       def push(self, *args):
9           """Save a transition"""
10          self.memory.append(Transition(*args))
11
12      def sample(self, batch_size):
13          return random.sample(self.memory, batch_size)
14
15      def __len__(self):
16          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:

```
1   def select_action(state):
2   ...
3       sample = random.random()
4       eps_threshold = EPS_END + (EPS_START - EPS_END) * \
```

```
5          math.exp(-1. * steps_done / EPS_DECAY)
6      steps_done += 1
7      if sample > eps_threshold:
8          with torch.no_grad():
9              return policy_net(state).max(1)[1].view(1, 1)
10     else:
11         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.

```
1  def optimize_model():
2      if len(memory) < BATCH_SIZE:
3          return
4      transitions = memory.sample(BATCH_SIZE)
5      batch = Transition(*zip(*transitions))
```

We first try to get a batch of Transitions into the model. If there isn't enough Transitions in the memory, we return. If we have enough information stored in the memory, we load a sample from there. Then, we create a batch variable from the transitions variable. The batch holds all the information regarding the state, action, reward and next state. Then, we divide each information from the batch into individual variables.

```
1  non_final_mask = torch.tensor(tuple(map(lambda s: s is not None,
2                                           batch.next_state)),
                                     ↪  device=device,
                                     ↪  dtype=torch.bool)
3      non_final_next_states = torch.cat([s for s in batch.next_state
4                                          if s is not None])
5      state_batch = torch.cat(batch.state)
6      action_batch = torch.cat(batch.action)
7      reward_batch = torch.cat(batch.reward)
```

On line 1, we are extracting every next state that is not final, which is not None and storing them as a boolean. True for every non-final state and False for final states. This will be helpful afterwards when we will compute the values of the next states. On line 3, we do same thing, except we store the values of the states. Line 5-7, we extract information about states, rewards and actions from the batch and store them into individual variables. Afterwards, we compute the state action values like this:

```
1  state_action_values = policy_net(state_batch).gather(1, action_batch)
```

We pass through the policy net with the state batch as the input and observe, which action the policy net would have taken. We get these from our action_batch variable. Next step is to calculate the next state values:

```
1  next_state_values = torch.zeros(BATCH_SIZE, device=device)
2  next_state_values[non_final_mask] =
   ↪  target_net(non_final_next_states).max(1)[0].detach()
```

On line 1, we allocate space for the results for our computation we are going to perform on line 2. On line 2, we pass through the target net with obtained next_state_values variable and extract the result/action that has the highest reward with the max function. We are using the target net for the improved stability that we mentioned before. The target net that is called on this line is the "old" target net that is constant. Another detail worth explaining is the significance of the non_final_mask variable. This is needed here since we will store a value in to next_state_values if the state was non-final and 0 if the state was final. This is determined by the values in the non_final_mask variable. The last important line of code in this function is the following:

```
1 expected_state_action_values = (next_state_values * GAMMA) +
  ↪  reward_batch
```

This is where we actually compute which action to take in order to get to the next state for each state. We multiply the next_state_values by gamma in order to guarantee convergence and add the rewards to it. This equation can be found under the DQN algorithm page in the tutorial, where the policy function is described.

For the training loop, the main structure looks like this:

```
1  num_episodes = any_number
2  for i_episode in range(num_episodes):
3      # Initialize the environment and state
4      env.reset()
5      last_screen = get_screen()
6      current_screen = get_screen()
7      state = current_screen - last_screen
8          ...
9          for t in count():
10         # Select and perform an action
11         action = select_action(state)
12         _, reward, done, _, _ = env.step(action.item())
13         reward = torch.tensor([reward], device=device)
14
15         # Observe new state
16         last_screen = current_screen
17         current_screen = get_screen()
18         if not done:
19             next_state = current_screen - last_screen
20         else:
21             next_state = None
22
23         # Store the transition in memory
24         memory.push(state, action, next_state, reward)
25
26         # Move to the next state
27         state = next_state
28
29         # Perform one step of the optimization (on the policy network)
30         optimize_model()
31         if done:
32             break
33
34         # Update the target network, copying all weights and biases in
             ↪  DQN
35         if t % TARGET_UPDATE == 0:
36             target_net.load_state_dict(policy_net.state_dict())
```

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