Course: CSE584 – Machine Learning
PSU ID: 904302574
Name: Eric Chou
Date: Oct. 22nd, 2024

Homework #2

Access my Code link here: Eric_Chou_DQN(RL)_on_CartPole_v1_Environment_with_Pytorch.ipynb

1. Abstract

This piece of code is one of my previous class projects that in particular focuses on building a **Deep Q-Network (DQN)**, which enables the agent to solve the CartPole-v1 environment in reinforcement learning. **The objective of this project is to train a reinforcement learning neural network that can predict the optimal actions to keep the pole upright during transportation on a moving cart.** It is based on DQN algorithm, which incorporates using an agent that interacts with a designed environment that the agent at all times seeks to maximize its cumulative reward by learning optimal policies.

All processes for the project begin by setting up the **CartPole-v1** environment using the **Gymnasium** library. This environment has the dynamics of a pole that attaches to a cart which is able to slide back and forth along a straight track (One-dimensional). Besides, DQN was incorporated through Pytorch, employing GPU acceleration where possible. The neural network implemented in this project consists of fully connected layers whose primary purpose was to estimate Q-values for each action.

As part of training the network, there is a replay buffer to which the agent's experiences are saved so that the network does not have to learn from experiences only in the present but also in some past experiences. An important aspect of this implementation is having a target network which is updated periodically to ensure stable learning of the network. The training loop is built between the agent and the environments' interactions where the agent chooses which action to perform using ε-greedy policy and the result are stored in the replay buffer. The DQN learns from the buffer using experience replay where batches are sampled and loss between the predicted Q values and the target Q values are minimized.

The final result shows that the agent did indeed learn the task of balancing the pole as episodes increase by time. In the beginning, from 0 to 300 episode, there were some fluctuations in the performance of the agent since it was still trying to explore the optimal action. However, during the remaining episodes, the agent's pole balance ability improved, as we can see from **Figure 1** that around the 450th episode, **the duration** markedly **increased** and continued to steadily

achieve to 500 around the 600th. This implementation also shows some important conclusions about DQN algorithm in reinforcement learning techniques.

2. Core Sections of the DQN Implementation

2.1 Replay Memory (Experience Replay):

In Code Section 1 below, the ReplayMemory is a crucial component that stores past experiences up to a fixed capacity. The aim of the replay memory is to store the experiences of the agent which encodes the **state**, **action**, and **reward**, as well as the **next_state**. New experiences are added using the push method while the sample method returns a fixed random mini batch for training. The len method returns the number of experiences which have already been stored in memory. In general, this helps in the stability of the training process since the agent learns from randomly selected past experiences.

2.2 Define Q-Network:

In Code Section 2 below, The Q-network is the neural network that contains the Q-values which are approximated for each action with regard to a given state. This network helps the agent in making decisions for its best possible action according to all the Q-values acquired by the agent. The architecture of the Q-network has **two hidden layers with activation functions** and **an output layer that outputs Q-values for each action**. The input and output layers consist of number of neurons that are equal to the number of observations and number of possible actions that are in the environment, respectively.

2.3 Optimization:

In Code Section 3 below, the optimization function is responsible for training the Q-network. It samples batches from the replay memory, computes Q-values, estimates the target Q-value using reward and next state based on target network, **minimizes the loss between the predicted Q-values and the target Q-values**, and finally uses a backward pass to update the network weights.

2.4 Training:

In Code Section 4 below, the training loop interacts with the environment, stores experiences in the replay memory, and calls the optimization function to update the

model. In detail, it starts with resetting the environment each episode, then selects an action based on the current policy. Next, it takes the selected action in the environment and receives the resulting next_state, reward, stores the experience in replay memory, calls the optimize model function to update Q-Network, and finally updates the target network to stabilize training.

Memory: Save the data to reuse later

```
[ ]  Transition = namedtuple('Transition',
                             ('state', 'action', 'next_state', 'reward'))

    #Store past experiences for training
    class ReplayMemory(object):

        def __init__(self, capacity):
            #Initialize a deque to store experience tuples with a fixed capacity
            self.memory = deque([], maxlen=capacity)

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

        def sample(self, batch_size):
            #Randomly sample a batch of experiences from memory
            return random.sample(self.memory, batch_size)

        def __len__(self):
            #Return the current size of the memory
            return len(self.memory)
```

▲ Code Section 1: Replay Memory

Define the Neural Network Architectures and Implement the DQN Algorithm

```
[ ]  #Define a Deep Q-Network
     class DQN(nn.Module):

         #Constructor to initialize the network layers
         def __init__(self, n_observations, n_actions):
             super(DQN, self).__init__()

             #Defining the first fully connected layer with 128 units
             self.layer1 = nn.Linear(n_observations, 128)
             #Defining the second fully connected layer with 128 units
             self.layer2 = nn.Linear(128, 128)
             #Defining the output layer with 'n_actions' units for each action
             self.layer3 = nn.Linear(128, n_actions)

         #Forward pass through the network
         def forward(self, x):
             #Apply ReLU activation to the output of the first layer
             x = F.relu(self.layer1(x))
             #Apply ReLU activation to the output of the second layer
             x = F.relu(self.layer2(x))
             #No activation on the output layer to get action values
             return self.layer3(x)
```

▲ Code Section 2: Q-Network

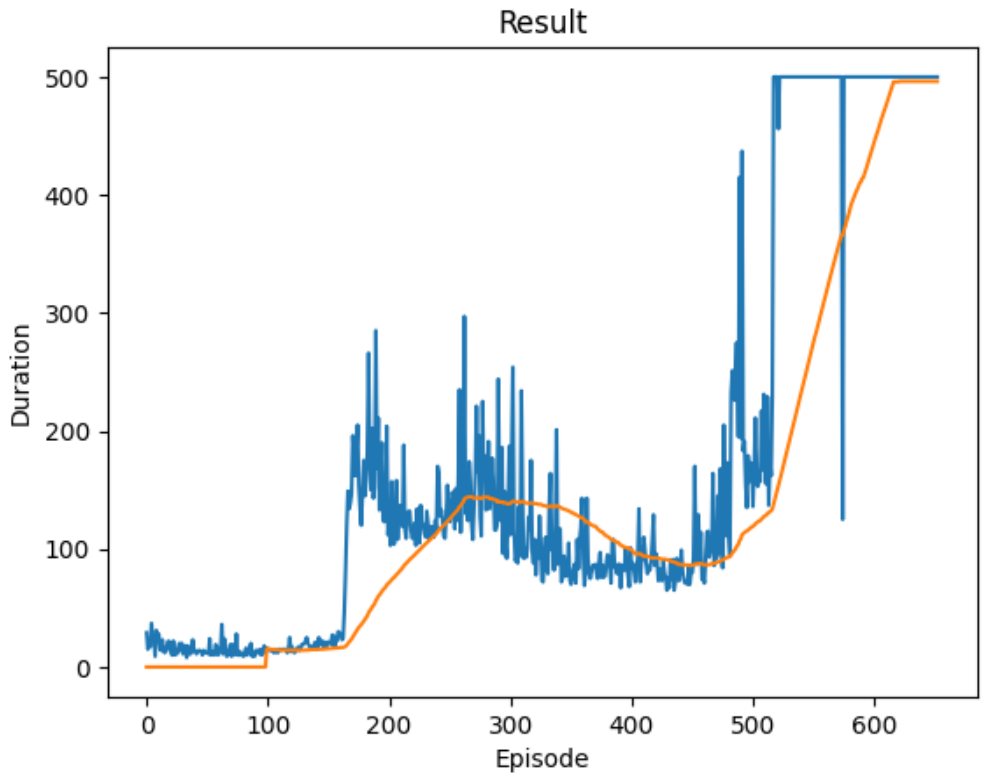
```
[ ]  def optimize_model():
         if len(memory) < BATCH_SIZE:
             return
         #Sample a batch of transitions from the replay memory
         transitions = memory.sample(BATCH_SIZE)

         #Convert the batch of sampled transitions into a Transition object
         batch = Transition(*zip(*transitions))

         #Create a mask to filter out final states (where next_state is None)
         non_final_mask = torch.tensor(tuple(map(lambda s: s is not None,
                                                 batch.next_state)), device=device, dtype=torch.bool)
         #Concatenate the non-final next states into a single tensor
         non_final_next_states = torch.cat([s for s in batch.next_state
                                                    if s is not None])
         #Concatenate state, action, and reward tensors into batches
         state_batch = torch.cat(batch.state)
         action_batch = torch.cat(batch.action)
         reward_batch = torch.cat(batch.reward)

         #Compute Q(s, a) using the policy network for the given state-action pairs
         state_action_values = policy_net(state_batch).gather(1, action_batch)

         #Compute the maximum Q-value for the next states using the target network
         next_state_values = torch.zeros(BATCH_SIZE, device=device)
         with torch.no_grad():
             next_state_values[non_final_mask] = target_net(non_final_next_states).max(1).values
         #Compute the expected Q-values using the Bellman equation
         expected_state_action_values = (next_state_values * GAMMA) + reward_batch

         #Compute Huber loss
         criterion = nn.SmoothL1Loss()
         loss = criterion(state_action_values, expected_state_action_values.unsqueeze(1))

         #Optimize the model
         optimizer.zero_grad()
         loss.backward()

         #Clip gradients to prevent large updates
         torch.nn.utils.clip_grad_value_(policy_net.parameters(), 100)

         #Perform a single optimization step (parameter update)
         optimizer.step()
```

▲ Code Section 3: Optimization

```
[ ]  if torch.cuda.is_available() or torch.backends.mps.is_available():
         num_episodes = 600
     else:
         num_episodes = 50

     for i_episode in range(num_episodes):
         #Initialize the environment and get its state
         state, info = env.reset()
         state = torch.tensor(state, dtype=torch.float32, device=device).unsqueeze(0)
         for t in count():
             #Select an action based on the current state
             action = select_action(state)
             #Take the selected action in the environment
             observation, reward, terminated, truncated, _ = env.step(action.item())
             #Convert reward to a tensor and set up termination flags
             reward = torch.tensor([reward], device=device)
             done = terminated or truncated

             #Define the next state or set it to None if the episode is done
             if terminated:
                 next_state = None
             else:
                 next_state = torch.tensor(observation, dtype=torch.float32, device=device).unsqueeze(0)

             #Store the transition in memory
             memory.push(state, action, next_state, reward)

             #Move to the next state
             state = next_state

             #Perform one step of the optimization (on the policy network)
             optimize_model()

             #Update the target network weights using a soft update strateg
             target_net_state_dict = target_net.state_dict()
             policy_net_state_dict = policy_net.state_dict()
             for key in policy_net_state_dict:
                 target_net_state_dict[key] = policy_net_state_dict[key]*TAU + target_net_state_dict[key]*(1-TAU)
             target_net.load_state_dict(target_net_state_dict)

             if done:
                 episode_durations.append(t + 1)
                 plot_durations()
                 break
```

▲ Code Section 4: Training

3. Final Result



▲ Figure 1: Result of DQN in CartPole-v1

3.1 Visualize and Performance:

In Figure 1, DQN performs excellently in the CartPole-v1 environment. Around episode 300, the performance is unstable, but by episodes 500 to 600, the reward curve steepens. This improvement is likely due to DQN using past experience (experience replay), making it better at predicting optimal actions. Besides, since CartPole-v1 is a discrete action space task and DQN is a discrete algorithm, this makes DQN highly suitable for CartPole-v1.

3.2 Data Efficiency:

DQN is a discrete algorithm, which means it can use experiences gathered from past episodes and store them in a replay buffer to update its Q-values. As you can see in Figure 1, the performance starts low but rapidly improves as the model gets better at predicting the optimal actions for given states. DQN is generally more sample-efficient because it uses experience replay, which allows it to recycle and learn from old transitions multiple times. This reuse of data makes DQN capable of extracting more learning from fewer episodes.

DQN's data efficiency is one of its strong suits, but it comes with the downside of being more challenging to stabilize, especially in environments with large state spaces or continuous actions.

3.3 Strength and Weakness of the algorithm:

- Strength: High data efficiency, especially in discrete action environments.
- Weakness: Can be unstable

3.4 Summary and Reflection:

- **DQN** is effective in **discrete action spaces** and offers faster learning when **data efficiency** is critical. However, it can **struggle with stability**, especially in complex or continuous environments.
- In the future, I plan to do experiments with **modifying** other parameters, such as the **learning rate** and **update rate**, to determine whether these adjustments can **enhance the algorithms' performance**. Additionally, if needed once some parameters changed, I plan to **increase the number of episodes** to see if the performance improves with extended training.