Homework 2

Nithya Thokala - nxt5283

Abstract:

This project deals with the **CartPole environment** and its solution using **Deep Q-Network (DQN)** using PyTorch. The CartPole task is one of the most commonly used tasks in RL as the agent learns to balance a pole with, by applying a force of +1 or -1 where the cart is moving. The objective is to learn the control law that can guarantee that the pole stays upright in order to gain the greatest overall reward.

In **DQN algorithm**, the Q-learning method is fused with deep learning to approximate Q-values, that are the summarized expected rewards from a particular state-space and action. Instead of the use of table that stores Q-values, a neural network estimates them of all actions possible given the current state. Some of the ideas are experience replay, where agent's interactions are stored in a buffer and then random batches are used for training while bridging the gap between consecutive experiences. This technique makes the agent more knowledgeable and also more stable in their training as well.

Moreover, the DQN employs a **target network** in order to overcome the problem of learning instability. The target network is a stale copy of the main Q-network updated occasionally to offer somewhat stable Q-value targets. The agent communicates with the environment with help of **epsilon-greedy policy** that helps to perform randomized actions from the best available ones and random ones. As time progresses the exploration rate (ϵϵ) is gradually lowered so as to force the agent to experience its acquired policy.

Each of the episodes involves state observation, action choice, reward gained, and transition to new state. Both of these transitions are stored in the replay buffer, while mini-batch is sampled to update the Q-network using **stochastic gradient descent**. The loss function compares the predicted Q-values and the target Q-values and helps train the network for better estimation. This goes on in many episodes where the agent's intent is to learn a policy that will always yield good results.

This project shows how using techniques such as DQN from **deep reinforcement learning** can effectively solve continuous control problems where action spaces are discrete. It explains how to implement common RL algorithm and introduces main ideas such as experience replay and target network to construct robust models. The implementation also provides valuable learning as to how neural networks & RL can be applied to solve a problem that involves decision making.

Core sections of the reinforcement learning implementation:

Initialization (agent.py)

```
 def __init__(self, n_states, n_actions, hidden_dim):
    """Initialize the Agent with the required networks and parameters.

    Args:
        n_states (int): input dimension (number of features in the state)
        n_actions (int): number of possible actions
        hidden_dim (int): number of units in the hidden layer of the Q-
network
    """
    # Local Q-network used to predict Q-values for the current state.
    self.q_local = QNetwork(n_states, n_actions, hidden_dim=16).to(device)

    # Target Q-network to generate target Q-values for stable training.
    self.q_target = QNetwork(n_states, n_actions,
hidden_dim=16).to(device)

    # Mean Squared Error loss function to compute the difference between
predicted and target Q-values.
    self.mse_loss = torch.nn.MSELoss()

    #  Adam optimizer for updating network weights.
    self.optim = optim.Adam(self.q_local.parameters(), lr=LEARNING_RATE)

    # Store state and action dimensions
    self.n_states = n_states
    self.n_actions = n_actions
    # Replay memory to store past experiences.
    self.replay_memory = ReplayMemory(10000)
```

Action Selection

```
def get_action(self, state, eps, check_eps=True):
    """Returns an action

    Args:
        state : 2-D tensor of shape (n, input_dim)
        eps (float): eps-greedy for exploration

    Returns: int: action index
    """
    global steps_done
    sample = random.random()

    if check_eps == False or sample > eps:
        with torch.no_grad():
            # Get action with highest expected reward from local Q-network
            return
self.q_local(Variable(state).type(FloatTensor)).data.max(1)[1].view(1, 1)
    else:
        # Choose a random action for exploration
        return torch.tensor([[random.randrange(self.n_actions)]],
device=device)
```

Learning

```
def learn(self, experiences, gamma):
    """Prepare minibatch and train them

    Args:
        experiences (List[Transition]): batch of `Transition`
        gamma (float): Discount rate of Q_target
    """
    # Check if replay memory has enough samples for a batch
    if len(self.replay_memory.memory) < BATCH_SIZE:
        return
    # Sample a batch of transitions from replay memory
    transitions = self.replay_memory.sample(BATCH_SIZE)

    # Unpack transitions into separate components
    batch = Transition(*zip(*transitions))

    states = torch.cat(batch.state)
    actions = torch.cat(batch.action)
    rewards = torch.cat(batch.reward)
    next_states = torch.cat(batch.next_state)
    dones = torch.cat(batch.done)

    # Expected Q values for current states using local network
    Q_expected = self.q_local(states).gather(1, actions)

    # Target Q values using target network and Bellman equation
    Q_targets_next = self.q_target(next_states).detach().max(1)[0]

    # Target values for current states
    Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))

    # Zero the gradients before backward pass
    self.optim.zero_grad()

    # Compute loss between expected and target Q values
    loss = self.mse_loss(Q_expected, Q_targets.unsqueeze(1))

    # Perform backpropagation to update network weights
    loss.backward()

    # Update weights using optimizer
    self.optim.step()
```

Agent Initialization (__init__):

Two Q-networks are initialized: A prediction network local to the agent and a target network to produce stable targets. The Adam optimizer is then used to update the local network's parameters when training is taken. We create replay buffer to make use of previously experienced things to make the training process stable with prior experience replay.

Action Selection (get_action):
Examine epsilon-greedy policy, where with epsilon probability a random action is taken, otherwise the best-known action is chosen. At the same time, as the agent becomes trained it decreases exploration and becomes more focused on exploitation. The agent employs the local Q-network to determine the best action at a time other than exploration.

Learning (Learn):
A few sequences of experience are selected randomly from the replay memory. The local network output calculates the Q-values for the current state and the target network generates the Q-values of the next state. MSE loss is then computed, and backpropagation is done to update the local network.

Github Link:

CartPole