Deep Q-Network Algorithm

Asfahan Shah (aks7824@psu.edu)

1 Abstract

A simple (Deep Q-Network) DQN is implemented. This algorithm is tested on CartPole environment. The DQN algorithm combines deep neural networks with Q-learning to learn optimal policies for an environment. The implementation has following key components:

- Experience Replay Memory: Stores state transitions. During training we a get a randomly sampled batch from experience replay memory. The benefit is that training in a batch is more effective rather than training one by one sample. Another benefit is by randomly sampling we are making data kind of I.I.D (Identically Independently Distributed) which is beneficial for us .
- Target Network: Uses a separate network for generating target Q-values. This network usually remains fixed and is updated via soft updates to provide stable training targets.
- Epsilon-Greedy Exploration: When choosing a action, if we always choose an action based on maximum Q-value it could be detrimental. This is because model is not know here so we have to explore. To do this we employ epsilon-greedy approach. For some probability we choose random action while for other we choose maximum Q-value action. This helps in balancing exploration and exploitation.

The training process follows these steps:

1. Initialize policy and target networks with identical weights
2. For each episode:
   - Reset environment and get initial state
   - For each time step:
     - Select action using epsilon-greedy approach
     - Execute action and observe reward and next state
     - Store transition in replay memory
     - Sample random batch from replay memory
     - Perform Q-learning update on policy network
     - Perform soft update on target network

2 Implementation Analysis

2.1 Policy Architecture

```
class Policy(torch.nn.Module):
    def __init__(self, state, action):
        super().__init__()
        # Three-layer neural network
        self.input = Linear(state, 128)      # First layer maps state space to
            hidden layer
        self.dense = Linear(128, 128)        # Hidden layer for feature extraction
```

```
        self.output = Linear(128, action)   # Output layer produces Q-values for
            each action

    def forward(self, tensor):
        x = F.relu(self.input(tensor))      # Apply ReLU to first layer output
        x = F.relu(self.dense(x))           # Apply ReLU to hidden layer output
        return self.output(x)               # Return raw Q-values
```

The neural network architecture implements a three-layer feed-forward network for Q-value approximation:

- Input Layer: Maps the state space to 128 hidden units.
- Hidden Layer: Maintains representation with 128 units for feature extraction.
- Output Layer: Produces Q-values for each possible action.

ReLU activation functions are used between layers.

2.2 Experience Replay Buffer

```
class memory():
    def __init__(self, capacity):
        # Initialize circular buffer with fixed capacity
        self.mem = deque([], maxlen=capacity)

    def add(self, state, action, reward, next_state):
        # Store transition tuple in memory
        self.mem.append((state, action, reward, next_state))

    def get_sample(self, size):
        # Randomly sample batch of a particular size of transitions for training
        return random.sample(self.mem, size)
    def __len__(self):
        return len(self.mem)
```

The experience replay buffer is crucial for stable learning. The following are key points:

- Uses a circular buffer (deque) with fixed capacity.
- Stores transitions as tuples: (state, action, reward, next_state).
- Random sampling breaks temporal correlations in the training data.
- Fixed capacity prevents memory issues.

2.3 Target Network Update Mechanism

```
def soft_update(self):
    # Get current weights of both networks
    w_1 = self.Policy_net.state_dict()
    w_2 = self.Target_net.state_dict()
    # Perform soft update.
    for key in w_1:
        # Update formula: theta_target = theta*policy + (1-theta)*target
        w_2[key] = self.theta * w_1[key] + (1-self.theta) * w_2[key]
    # Load updated weights into target network
    self.Target_net.load_state_dict(w_2)
```

2.3.1 Traditional Hard Updates vs Soft Updates

Problems with Hard Updates: In traditional DQN implementations, the target network is updated by completely copying the policy network's weights after every K steps. This approach has following problems:

- **Abrupt Changes**: When the target network is updated, there's a sudden large change in the target Q-values
- **Training Instability**: These abrupt changes can lead to significant oscillations in the learning process
- **Frequency Dilemma**:
  - If K is too small: Target values change too rapidly, leading to instability
  - If K is too large: Target network becomes outdated, slowing learning

Advantages of Soft Updates: Our implementation uses soft updates after each step with a small update rate ($\theta = 0.005$):

- **Gradual Updates**: Instead of copying all weights, we blend a small portion of the policy network's weights into the target network
- **Update Formula**: For each weight w: $w_{target} = \theta \cdot w_{policy} + (1 - \theta) \cdot w_{target}$ where $\theta = 0.005$
- **Continuous Learning**: Updates can occur after every step without destabilizing training
- **Smooth Transitions**: Prevents sudden jumps in target Q-values

2.4 Action Selection Strategy

```
def choose_action(self, state):
    self.steps += 1
    # Calculate exploration probability using exponential decay
    threshold = self.END + (self.START - self.END) * \
        math.exp(-1. * self.steps / self.DECAY)
    sample = random.random()

    if sample > threshold:
        # Exploit: choose action with highest Q-value
        with torch.no_grad():
            return self.Policy_net(state).max(1)[1].view(1, 1)
    else:
        # Explore: choose random action
        return torch.tensor([[self.env.action_space.sample()]],
                            device=self.DEVICE, dtype=torch.long)
```

The action selection implements epsilon-greedy exploration. Following are the key points:

- Exploration probability decays exponentially with steps.
- The exploration probability is decaying with steps because after lets say after some large steps, we would want policy to exploit as ideally we would have explored sufficiently.
- During exploration: Random action sampling from action space.
- During exploitation: Selects action with highest Q-value.

2.5 Training Process

```python
def train(self):
    # Skip if not enough samples in buffer
    if len(self.buffer) < self.BATCH_SIZE:
        return None

    # Sample random batch from replay buffer
    batch = self.buffer.get_sample(self.BATCH_SIZE)
    # Create mask for non-final states
    non_final_index = torch.tensor(
        tuple([True if i[3] != None else False for i in batch]),
        device=self.DEVICE, dtype=torch.bool)

    non_final_states = torch.cat([i[3] for i in batch if i[3] != None])

    # Prepare batch data
    batch = list(zip(*batch))
    state_batch = torch.cat(batch[0])    # Current states
    action_batch = torch.cat(batch[1])   # Actions taken
    reward_batch = torch.cat(batch[2])   # Rewards received

    # Get predicted Q-values using policy network
    predicted_value = self.Policy_net(state_batch).gather(1, action_batch)

    temp = torch.zeros(self.BATCH_SIZE).to(self.DEVICE)

    # Calculate target Q-values using target network
    with torch.no_grad():
        temp[non_final_index] = self.Target_net(non_final_states).max(1)[0]

    # Compute target values using Bellman equation
    target_value = reward_batch + self.GAMMA * temp

    # Calculate loss
    criteria = torch.nn.SmoothL1Loss()
    loss = criteria(predicted_value, target_value.unsqueeze(1))

    # Update the policy network
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()
```

The training process implements the core DQN algorithm. The following are key points:

- Samples random batches from experience replay.
- Use policy network to get predicted value.
- Uses target network to get initial target values
- Apply Bellman equation on initial target values to compute target values
- Calculate loss based on target values and predicted values
- Update the policy model based on this loss.

2.5.1 Training Loop Implementation

```python
def loop(self):
    max_ep_len = 500  # maximum length of episode
    max_training_timesteps = int(1e5) # maximum number of training timesteps
    save_freq = 400*2 # save frequency
    log_running_reward = 0
    log_running_episodes = 0

    while self.steps <= max_training_timesteps:
        state, _ = self.env.reset()
        state = torch.tensor(state, dtype=torch.float32,
                             device=self.DEVICE).unsqueeze(0)
        total_reward = 0

        for t in range(1, max_ep_len+1):
            action = self.choose_action(state) # get action
            observation, reward, terminated, truncated, _ = \
                self.env.step(action.item())
            done = terminated or truncated # execute action to get next state and reward
            total_reward += reward

            reward = torch.tensor([reward], device=self.DEVICE)
            if terminated:
                next_state = None
            else:
                next_state = torch.tensor(observation, dtype=torch.float32,
                                          device=self.DEVICE).unsqueeze(0)

            self.buffer.add(state, action, reward, next_state) # add the (state, action,reward,next_state)
            state = next_state
            self.train() # start training model
            self.soft_update() # update the model

            if self.steps % save_freq == 0: #Save the average rewards (For plotting purposes)
                log_avg_reward = log_running_reward / log_running_episodes
                log_avg_reward = round(log_avg_reward, 4)
                self.avg_reward.append((self.steps, log_avg_reward))
                log_running_reward = 0
                log_running_episodes = 0

            if done:  # if episode is terminated we dont need to go to max_ep_len
                break

        log_running_reward += total_reward
        log_running_episodes += 1
```

The training loop implements the core interaction between the agent and environment, managing both episode execution and logging. Key components include:

- **Training Parameters**:
  - Maximum episode length: 500 steps
  - Total training timesteps: 100,000
  - Logging frequency: Every 800 steps
- **Episode Structure**:

- Resets environment at start of each episode
- Converts state observations to tensors
- Tracks cumulative reward per episode

- **Step-by-Step Execution**:
  - Action selection using epsilon-greedy policy
  - Environment interaction and reward collection
  - State processing and storage in replay buffer
  - Training update and target network soft update

- **Performance Logging**:
  - Maintains running average of episode rewards
  - Logs performance metrics at regular intervals
  - Handles both natural termination and truncation

2.6 Visualization and Analysis

```
def plot(self, name):
    import matplotlib.pyplot as plt
    plt.xlabel("Time steps")
    plt.ylabel("Avg episodic reward")
    plt.title(f"DQN plot for {name}")
    plt.plot([i[0] for i in self.avg_reward],
             [i[1] for i in self.avg_reward])
    plt.show()
```

The visualization component has following important points:

- Plots average episodic reward against training timesteps
- Helps identify learning stability and convergence
- Facilitates comparison between different hyperparameter settings

2.7 Usage Example

```
env = gym.make("CartPole-v1")
cart = DQN(env, Policy)
cart.loop()
cart.plot("CartPole-V1")
```

The implementation demonstrates practical usage with the CartPole-v1 environment:

- Environment initialization using OpenAI Gym
- DQN agent creation with custom policy network
- Training execution through the main loop
- Performance visualization using matplotlib