DQN Algorithm
Mid-Term Report

Team D47

July 13, 2024

Q-Learning Algorithm

Q-Learning is the most basic implementation of RL. It is a type of reinforcement learning algorithm used to train agents to make decisions in an environment. The goal is to learn a policy that tells the agent what action to take in a given state to maximize cumulative rewards over time. The basic work-flow is -

1. **Initializing the Q-table** Q-table stores the $Q(S, a)$ values (where $Q(S, a)$ represents the Q-value (expected total reward) if action a is taken in state S). Initially, each Q-value is set to 0.

2. **Choosing an action** If we use an exploration strategy (like ϵ-greedy), then choosing a random action (exploration) is done with probability ϵ and choosing the action with the highest Q-value (exploitation). So, the value of ϵ is mostly set to 1 initially to encourage exploration, but a decay-factor(less than 1, but close to 1) is multiplied with it at the end of each episode. And this prevent the agent from getting stuck in local optima by encouraging it to explore different actions, especially in the early stages of training and over time, ϵ gets reduced to favor more exploitation as the agent learns.

3. **Taking An Action** The chosen action in the environment is performed, and then the next state and the reward received are observed.

4. **Updating the Q-value** The following formula is used for updating $Q(S, a)$ values -

$$Q(S, a) = Q(S, a) + \alpha(r + \gamma(max_{a'} Q(S', a')) - Q(S, a))$$

   where, α is the learning rate
   r is the reward received
   γ is the discount factor (how much future rewards matter)
   S' is the resulting state after performing our action
   $max_{a'} Q(S', a')$ is the maximum Q-value in the resultant(next) state.

5. **Repeating the process** Whenever a dead-end is reached, i.e, a state is reached in which no possible further action can be taken, the present episode gets terminated and another episode begins, and this continues as per the number of episodes set before.

Workflow to be followed in each episode -

1. Resetting the environment to get the initial state S .

2. For each step in the episode - An action a is chosen according to the ϵ-greedy strategy. Then the reward r and the next state S' is observed and the Q-value is updated using the update formula and $S = S'$ is set for the next iteration.

A rough python code for implementing the above algorithm is shown below

-

```
# Defining the environment
num_states = 25  # Number of states in our environment
num_actions = 5  # Number of possible actions
goal_state = 24  # The goal state

# Initializing Q-table with zeros
Q_table = np.zeros((n_states, n_actions))
# Defining the parameters
alpha = 0.8          # Learning rate
gamma = 0.95       # Discount factor
epsilon = 1.0                # Exploration rate
epsilon_decay = 0.995        # Decay rate for epsilon
min_epsilon = 0.01           # Minimum exploration rate
num_episodes = 1000          # Number of episodes

# Training
for episode in range(num_episodes):
    # Start from a random state
    current_state = np.random.randint(0, n_states)
    while current_state != goal_state:
        # Choose action using epsilon-greedy strategy
        if np.random.rand() < epsilon:
            # Explore
            action = np.random.randint(0, num_actions)
        else:
            # Exploit
            action = np.max(Q_table[current_state])

        # Setting the next state
        next_state = (current_state + 1) % n_states

        # Reward of 1 is given if the goal state is reached, otherwise 0
        reward = 1 if next_state == goal_state else 0

        # Updating Q-value
        Q[current_state][action] += alpha * (reward +
        gamma * np.max(Q_table[next_state]) - Q[current_state][action])

        # Moving to the next state
        current_state = next_state

    # Updating epsilon
    epsilon = max(min_epsilon, epsilon * epsilon_decay)
```

DQN(Deep Q-Networks)

DQN is an advanced version of Q-learning that uses deep learning techniques to handle complex environments, especially those with high-dimensional state spaces. The normal Q-learning algorithm works well for small state spaces, but becomes impractical for complex environments.
Instead of using a Q-table,DQN employs a neural network to approximate the Q-values. This allows it to generalize and learn from states it has not encountered before.

A work-flow of DQN can be roughly represented as -

Initializing the components(Environment, Q-network, Replay Buffer)
↓
For Each Episode
↓
Resetting Environment
↓
While Episode not done
↓
Selecting Action(using ϵ-greedy policy)
↓
Taking Action
↓
Storing Experience(state, action, reward, next state)
↓
Sample from Replay Buffer
↓
Computing targets and updating Q-Network
↓
Updating Target Network (Periodically)
↓
Updating Epsilon
↓
End of Episode
↓
Testing Learned Policy

Algorithm and the code

Imports and Environment Setup

```
import numpy as np # for numerical operations
import gym # for creating and interacting with environments
import random # for random action selection
from collections import deque # for replay buffer
import tensorflow as tf # for building and training the neural network
env = gym.make("FrozenLake-v1", is_slippery=False)
```

Hyperparameters

```
alpha = 0.001            # Learning rate
gamma = 0.99             # Discount factor
epsilon = 1.0            # Initial exploration rate
epsilon_decay = 0.995    # Decay rate for epsilon
min_epsilon = 0.01       # Minimum exploration rate
num_episodes = 1000       # Number of episodes
batch_size = 32           # Batch size for training
replay_buffer_size = 2000  # Size of replay buffer
```

Replay Buffer Class

```
# Replay Buffer
class ReplayBuffer:
    def __init__(self, max_size):
        self.buffer = deque(maxlen=max_size)

    def add(self, experience): # adds new experiences to the buffer
        self.buffer.append(experience)

    def sample(self, batch_size):
    # returns a random sample of experiences for training
        return random.sample(self.buffer, batch_size)

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

Neural Network Creation

```
def create_q_network(input_shape, action_space):
model = tf.keras.Sequential([
    tf.keras.layers.Dense(24, input_shape=input_shape, activation='relu'),
    tf.keras.layers.Dense(24, activation='relu'),
    tf.keras.layers.Dense(action_space, activation='linear')
])
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=alpha),
loss='mse')
return model
```

*** The above neural network has two hidden layers with 24 neurons and ReLU activation. Output layer has a size equal to the action space and uses linear activation. And it uses the Adam optimizer and mean squared error loss for training. ***

Initializing the Q-networks

```
# Main Network: Used for selecting actions
main_q_network = create_q_network(input_shape, action_space)
# Target Network: Used for calculating target Q-values
target_q_network = create_q_network(input_shape, action_space)
```

Training Loop

```
for episode in range(num_episodes):
    state = env.reset() # resets the environment and gets the initial state
    state = np.reshape(state, [1, 1])
    done = False
    while not done:
        if np.random.rand() < epsilon:
            action = env.action_space.sample()
        else:
            action = np.max(main_q_network.predict(state))
        next_state, reward, done, _ = env.step(action)
        next_state = np.reshape(next_state, [1, 1])

        # Storing experience
        replay_buffer.add((state, action, reward, next_state, done))

        # If enough experiences are stored, a batch is sampled for training
        if len(replay_buffer) >= batch_size:
            batch = replay_buffer.sample(batch_size)
            for state_b, action_b, reward_b, next_state_b, done_b in batch:
                target = reward_b
                if not done_b:
                    target = target +
                    gamma * np.max(target_q_network.predict(next_state_b))
                target_f = main_q_network.predict(state_b)
                target_f[0][action_b] = target
                # Train the model
                main_q_network.fit(state_b, target_f, epochs=1, verbose=0)

        state = next_state
    epsilon = max(epsilon, epsilon*epsilon_decay)
    # Periodically update the target network
    if episode % 20 == 0:
        target_q_network.set_weights(main_q_network.get_weights())
```

Advantages

- **DQN uses deep neural networks** to approximate the Q-value function, allowing it to handle environments with higer dimensional state spaces.The deep architecture automatically learns relevant features from the input data, reducing the need for manual feature engineering.
- **DQN uses replay buffer** to store agent experiences(state, action, reward, next state).By randomly sampling from this buffer during training, DQN mitigates the problem of correlated samples, leading to more stable and efficient learning. Also, this mechanism allows the algorithm to reuse past experiences multiple times, enhancing sample efficiency and speeding up the learning process.
- **DQN maintains a separate target network** that is updated less frequently (e.g., every few thousand steps). This helps in stabilizing the training process by reducing oscillations in the Q-value estimates, as the targets for the Q-learning updates remain fixed for a period.
- **DQN serves as foundation for more advanced algorithms** like Double DQN, Dueling DQN, and Prioritized Experience Replay, which address specific limitations of the basic DQN.

Limitations

- **Sensitivity to Hyperparameters** DQN's performance is highly dependent on careful tuning of hyperparameters, which can be complex and time-consuming
- **Lack of much exploration** The epsilon-greedy strategy can lead to suboptimal exploration, potentially missing better policies in early training stages.
- **Over-estimation of action-values** The Q-learning update used in DQN can lead to overestimation of action-values.
- **Sample inefficiency** It often requires a large number of samples to converge, making it less efficient in data-scarce environments.