Agent Deep Q-Network

Systèmes multi agents et intelligence artificielle distribuée

Master 1

Systèmes Distribués et Intelligence Artificielle

Préparé par : **Yahya Ghallali**

Institution : **ENSET**

March 27, 2025

1 Deep Q-Network (DQN)

Deep Q-Network (DQN) is a reinforcement learning algorithm that combines Q-Learning with deep neural networks. It is used to approximate the Q-value function, which helps an agent learn optimal policies in environments with large state spaces.

1.1 Overview of DQN

The DQN algorithm uses a neural network to estimate the Q-values for each state-action pair. The agent interacts with the environment, collects experiences, and stores them in a replay memory. These experiences are then sampled to train the neural network, which helps stabilize the learning process.

1.2 Key Components of DQN

- **Replay Memory:** A buffer that stores past experiences $(s, a, r, s', done)$ to break the correlation between consecutive experiences.
- **Neural Network:** A model that approximates the Q-value function $Q(s, a)$.
- **Loss Function:** The mean squared error (MSE) between the predicted Q-values and the target Q-values.
- **Exploration-Exploitation Tradeoff:** Controlled by the ϵ-greedy strategy, where the agent explores random actions with probability ϵ and exploits the learned policy otherwise.

1.3 Implementation of DQNAgent

The following code snippet shows the implementation of the `DQNAgent` class, which encapsulates the DQN algorithm:

```
class DQNAgent:
    def __init__(self):
        self.state_size = STATE_SIZE
        self.action_size = ACTION_SIZE
        self.memory = deque(maxlen=MEMORY_SIZE)
        self.epsilon = EPSILON
        self.model = self.build_model()

    def build_model(self):
        model = Sequential([
            Input(shape=(self.state_size,)),
            Dense(24, activation="relu"),
            Dense(24, activation="relu"),
            Dense(self.action_size, activation="linear"),
        ])
        model.compile(loss="mse", optimizer=Adam(learning_rate=
           LEARNING_RATE))
        return model
```

```
    def act(self, state):
        if np.random.rand() < self.epsilon:
            return random.randrange(self.action_size)
        q_values = self.model.predict(np.array([state]), verbose=0)[0]
        return np.argmax(q_values)

    def replay(self):
        if len(self.memory) < BATCH_SIZE:
            return
        batch = random.sample(self.memory, BATCH_SIZE)
        for state, action, reward, next_state, done in batch:
            target = self.model.predict(np.array([state]), verbose=0)[0]
            if done:
                target[action] = reward
            else:
                target[action] = reward + GAMMA * np.max(
                    self.model.predict(np.array([next_state]), verbose=0)[0]
                )
            self.model.fit(np.array([state]), np.array([target]), epochs=1, verbose=0)
        if self.epsilon > EPSILON_MIN:
            self.epsilon *= EPSILON_DECAY
```

1.4 GridWorld Environment

The agent interacts with a simple 4x4 grid environment, as implemented in the `GridWorld` class. The environment provides the state, reward, and transition dynamics.

```
class GridWorld:
    def __init__(self):
        self.grid_size = GRID_SIZE
        self.reset()

    def reset(self):
        self.agent_position = (0, 0)
        self.goal_position = (3, 3)
        self.obstacle_position = (1, 1)
        return self.get_state()

    def step(self, action):
        x, y = self.agent_position
        dx, dy = MOVES[action]
        new_x, new_y = x + dx, y + dy
        if 0 <= new_x < GRID_SIZE and 0 <= new_y < GRID_SIZE:
            self.agent_position = (new_x, new_y)
        if self.agent_position == self.goal_position:
            return self.get_state(), 10, True
        elif self.agent_position == self.obstacle_position:
            return self.get_state(), -5, False
        else:
            return self.get_state(), -1, False
```

1.5 Training the Agent

The following code snippet demonstrates how the agent is trained using the DQN algorithm:

```
grid_world = GridWorld()
agent = DQNAgent()

for ep in range(EPISODES):
    state = grid_world.reset()
    total_reward = 0
    for step in range(50):
        action = agent.act(state)
        next_state, reward, done = grid_world.step(action)
        agent.remember(action=action, state=state, reward=reward,
            next_state=next_state, done=done)
        state = next_state
        total_reward += reward
        if done:
            break
    agent.replay()
    print(f"Episode {ep+1}/{EPISODES} :")
    print(f"\tScore: {total_reward}")
    print(f"\tEpsilon: {agent.epsilon:.2f}")
```

1.6 Conclusion

The Deep Q-Network algorithm is a powerful method for solving reinforcement learning problems. By combining Q-Learning with deep neural networks, it enables agents to learn optimal policies in complex environments.

2 Double Deep Q Network

Double Deep Q-Network (DDQN) is an improvement over the standard DQN algorithm. It addresses the overestimation bias in Q-value updates by using two separate networks: the main network and the target network.

2.1 Key Modifications in DDQN

- **Target Network:** A separate neural network is used to calculate the target Q-values, which helps stabilize training.
- **Periodic Updates:** The weights of the target network are updated periodically by copying the weights from the main network.
- **Target Calculation:** The target Q-value for a given state-action pair is calculated using the target network instead of the main network.

2.2 Comparison with DQN

The main difference between DQN and DDQN lies in how the target Q-values are calculated:

- In DQN, the same network is used to select and evaluate actions, which can lead to overestimation bias.
- In DDQN, the main network selects the action, but the target network evaluates the Q-value of that action, reducing overestimation bias.

2.3 Implementation of DDQNAgent

The following code snippet shows the implementation of the `DDQNAgent` class, which encapsulates the DDQN algorithm:

```
class DDQNAgent:
    def __init__(self):
        self.state_size = STATE_SIZE
        self.action_size = ACTION_SIZE
        self.memory = deque(maxlen=MEMORY_SIZE)
        self.epsilon = EPSILON
        self.model = self.build_model()
        self.target_model = self.build_model()
        self.update_target_model()

    def build_model(self):
        model = Sequential([
            Input(shape=(self.state_size,)),
            Dense(24, activation="relu"),
            Dense(24, activation="relu"),
            Dense(self.action_size, activation="linear"),
        ])
        model.compile(loss="mse", optimizer=Adam(learning_rate=
            LEARNING_RATE))
        return model

    def update_target_model(self):
        self.target_model.set_weights(self.model.get_weights())

    def replay(self):
        if len(self.memory) < BATCH_SIZE:
            return
        batch = random.sample(self.memory, BATCH_SIZE)
        for state, action, reward, next_state, done in batch:
            target = self.model.predict(np.array([state]), verbose=0)[0]
            if done:
                target[action] = reward
            else:
                target[action] = reward + GAMMA * np.max(
                    self.target_model.predict(np.array([next_state]),
                        verbose=0)[0]
                )
```

```
            self.model.fit(np.array([state]), np.array([target]), epochs=1, verbose=0)
        if self.epsilon > EPSILON_MIN:
            self.epsilon *= EPSILON_DECAY
```

2.4 Advantages of DDQN

- Reduces overestimation bias in Q-value updates.
- Improves stability and convergence of the learning process.
- Enables better performance in complex environments compared to DQN.

2.5 Training the DDQNAgent

The following code snippet demonstrates how the `DDQNAgent` is trained using the GridWorld environment:

```
grid_world = GridWorld()
agent = DDQNAgent()
period = 10

for ep in range(EPISODES):
    state = grid_world.reset()
    total_reward = 0
    for step in range(50):
        action = agent.act(state)
        next_state, reward, done = grid_world.step(action)
        agent.remember(state, action, reward, next_state, done)
        state = next_state
        total_reward += reward
        if done:
            break
    agent.replay()
    if ep % period == 0:
        agent.update_target_model()
    print(f"Episode {ep+1}/{EPISODES} :")
    print(f"\tScore: {total_reward}")
    print(f"\tEpsilon: {agent.epsilon:.2f}")
```

2.6 Conclusion

The Double Deep Q-Network algorithm builds upon the strengths of DQN while addressing its limitations. By incorporating a target network and separating action selection from evaluation, DDQN achieves more stable and reliable learning in reinforcement learning tasks.