



Agent Deep Q-Network

Systemes multi agents et intelligence artificielle distribuée

Master 1

Systemes 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:

```
1 class DQNAgent:
2     def __init__(self):
3         self.state_size = STATE_SIZE
4         self.action_size = ACTION_SIZE
5         self.memory = deque(maxlen=MEMORY_SIZE)
6         self.epsilon = EPSILON
7         self.model = self.build_model()
8
9     def build_model(self):
10        model = Sequential([
11            Input(shape=(self.state_size,)),
12            Dense(24, activation="relu"),
13            Dense(24, activation="relu"),
14            Dense(self.action_size, activation="linear"),
15        ])
16        model.compile(loss="mse", optimizer=Adam(learning_rate=
17            LEARNING_RATE))
18        return model
```

```

18
19 def act(self, state):
20     if np.random.rand() < self.epsilon:
21         return random.randrange(self.action_size)
22     q_values = self.model.predict(np.array([state]), verbose=0)[0]
23     return np.argmax(q_values)
24
25 def replay(self):
26     if len(self.memory) < BATCH_SIZE:
27         return
28     batch = random.sample(self.memory, BATCH_SIZE)
29     for state, action, reward, next_state, done in batch:
30         target = self.model.predict(np.array([state]), verbose=0)[0]
31         if done:
32             target[action] = reward
33         else:
34             target[action] = reward + GAMMA * np.max(
35                 self.model.predict(np.array([next_state]), verbose=0)
36                 [0]
37             )
38             self.model.fit(np.array([state]), np.array([target]), epochs
39                             =1, verbose=0)
40     if self.epsilon > EPSILON_MIN:
41         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.

```

1 class GridWorld:
2     def __init__(self):
3         self.grid_size = GRID_SIZE
4         self.reset()
5
6     def reset(self):
7         self.agent_position = (0, 0)
8         self.goal_position = (3, 3)
9         self.obstacle_position = (1, 1)
10        return self.get_state()
11
12    def step(self, action):
13        x, y = self.agent_position
14        dx, dy = MOVES[action]
15        new_x, new_y = x + dx, y + dy
16        if 0 <= new_x < GRID_SIZE and 0 <= new_y < GRID_SIZE:
17            self.agent_position = (new_x, new_y)
18            if self.agent_position == self.goal_position:
19                return self.get_state(), 10, True
20            elif self.agent_position == self.obstacle_position:
21                return self.get_state(), -5, False
22            else:
23                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:

```
1 grid_world = GridWorld()
2 agent = DQNAgent()
3
4 for ep in range(EPISODES):
5     state = grid_world.reset()
6     total_reward = 0
7     for step in range(50):
8         action = agent.act(state)
9         next_state, reward, done = grid_world.step(action)
10        agent.remember(action=action, state=state, reward=reward,
11                        next_state=next_state, done=done)
12        state = next_state
13        total_reward += reward
14        if done:
15            break
16    agent.replay()
17    print(f"Episode {ep+1}/{EPISODES} :")
18    print(f"\tScore: {total_reward}")
19    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 (DDQN)

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 DDQN Agent

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

```
1 class DDQN Agent:
2     def __init__(self):
3         self.state_size = STATE_SIZE
4         self.action_size = ACTION_SIZE
5         self.memory = deque(maxlen=MEMORY_SIZE)
6         self.epsilon = EPSILON
7         self.model = self.build_model()
8         self.target_model = self.build_model()
9         self.update_target_model()
10
11     def build_model(self):
12         model = Sequential([
13             Input(shape=(self.state_size,)),
14             Dense(24, activation="relu"),
15             Dense(24, activation="relu"),
16             Dense(self.action_size, activation="linear"),
17         ])
18         model.compile(loss="mse", optimizer=Adam(learning_rate=
19             LEARNING_RATE))
20         return model
21
22     def update_target_model(self):
23         self.target_model.set_weights(self.model.get_weights())
24
25     def replay(self):
26         if len(self.memory) < BATCH_SIZE:
27             return
28         batch = random.sample(self.memory, BATCH_SIZE)
29         for state, action, reward, next_state, done in batch:
30             target = self.model.predict(np.array([state]), verbose=0)[0]
31             if done:
32                 target[action] = reward
33             else:
34                 target[action] = reward + GAMMA * np.max(
35                     self.target_model.predict(np.array([next_state]),
36                         verbose=0)[0])
```

```

36         self.model.fit(np.array([state]), np.array([target]), epochs
37                         =1, verbose=0)
38     if self.epsilon > EPSILON_MIN:
39         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:

```

1  grid_world = GridWorld()
2  agent = DDQNAgent()
3  period = 10
4
5  for ep in range(EPISODES):
6      state = grid_world.reset()
7      total_reward = 0
8      for step in range(50):
9          action = agent.act(state)
10         next_state, reward, done = grid_world.step(action)
11         agent.remember(state, action, reward, next_state, done)
12         state = next_state
13         total_reward += reward
14         if done:
15             break
16     agent.replay()
17     if ep % period == 0:
18         agent.update_target_model()
19     print(f"Episode {ep+1}/{EPISODES} :")
20     print(f"\tScore: {total_reward}")
21     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.