Artificial Intelligence and Machine Learning

Cliffwalking

Authors:
Alessandro Rocchi
Onorio Iacobelli

Contents

1 Introduction

In this Machine Learning project we will focus on the subject of Reinforcement Learning applied to the "Cliff Walking" environment. The goal of the agent is to reach a specified

location, by moving in the four directions without falling down the cliff. This problem represents a fundamental challenge in Reinforcement Learning, combining elements of exploration/exploitation tradeoff, temporal credit assignment and policy optimization. This environment provides an excellent testbed for comparing different Reinforcement Learning approaches, in particular tabular Q-learning and Deep Q-Networks (DQN).

Our implementation explores both classical tabular methods, suitable for discrete state spaces and modern deep learning approaches that can handle more complex scenarios. Trough extensive experimentation and hyperparameter tuning, we demonstrate how different choices and configurations impact learning performance and convergence behavior.

2 Reinforcement Learning

Reinforcement Learning is a branch of machine learning focused on decision making trough interaction with the environment. Unlike supervised learning, where the correct outputs are given for training, in Reinforcement Learning the agent must discover the optimal behavior by trial and error, guided only by rewards. Formally, Reinforcement Learning is often modeled as a Markov Decision Process (MDP), defined by states, actions, transition probabilities and rewards. The agent's goal is to learn a policy, a mapping from states to actions, that maximizes the expected cumulative reward over time.

The core components of a Markov Decision Process are:

- **Agent**: the learner or decision-maker that interacts with the environment. In our case this is the entity navigating the cliff walking grid.
- **Environment**: the external system that the agent interacts with. For cliff walking, this includes the grid world, cliff locations, and reward structure.
- **State (S)**: a representation of the current situation in the environment. In cliff walking, this is the agent's current position in the grid.
- **Action (A)**: the choices available to the agent at any given state.
- **Reward (R)**: the feedback signal from the environment indicating how good or bad the last action was. Rewards guide the learning process.
- **Policy (π)**: the agent's strategy for choosing actions given states. This can be deterministic (always choose the same action in a given state) or stochastic (choose actions according to a probability distribution).
- $\delta(s'|s, a)$: the probability distribution over the transitions.

The Reinforcement Learning process follows a continuous loop:

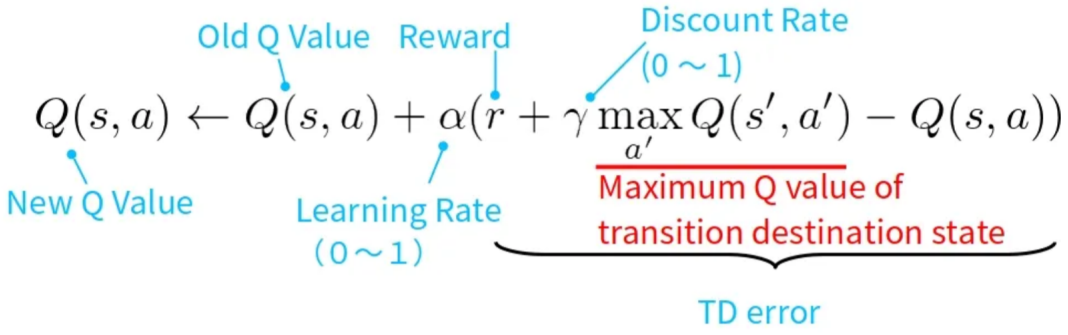
1. Observe the current state of the environment
2. Select an action based on the current policy
3. Execute the action in the environment
4. Receive a reward and observe the new state

5. Update the policy based on the experience
6. Repeat until learning converges or a stopping criterion is met

One of the fundamental challenges in Reinforcement Learning is balancing **exploration** (trying new actions to discover potentially better strategies) with **exploitation** (using knows good actions to maximize rewards). This balancing is crucial because too much exploration can lead to the agent wasting time on suboptimal actions, on the other hand with too little exploration the agent may get stuck in a local optimum and never discover the best strategy.

2.1 Q-Learning

Q-Learning is a model-free Reinforcement Learning algorithm that learns the quality of actions, telling the agent what actions to take under what circumstances. The "Q" stands for quality, representing the expected future reward for taking a specific action in a specific state.
The **Q-function** Q(s,a) represents the expected cumulative reward when taking action 'a' in state 's' and then following the optimal policy. The learning process involves iteratively updating these Q-values using the Bellman equation:



Where:

- α **(alpha)**: learning rate controlling how quickly new information overrides old information
- **r**: immediate reward received
- γ **(gamma)**: discount factor determining the importance of future rewards
- **s'**: next state after taking action 'a' in state 's'

Tabular Q-Learning maintains explicit Q-values for every (state, actions) pair in a lookup table. This approach works well for environments with small, discrete state and action spaces but becomes impractical as the state space grows.

Deep Q-Networks (DQN) use neural networks to approximate the Q-function, enabling the handling of large or continuous state spaces. Instead of storing individual Q-values, the network learns to map states to Q-values for all possible actions.

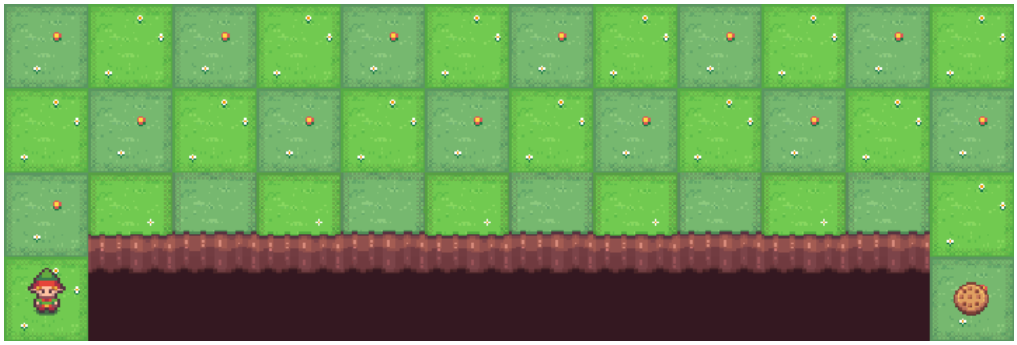
To balance exploration and exploitation, we implement an epsilon-greedy policy:

- With probability ε **(epsilon)**: choose a random action (exploration)
- With probability **(1-ε)**: choose the action with the highest Q-value (exploitation)

The epsilon value starts high (encouraging exploration) and gradually decreases (shifting toward exploitation) as learning progresses.

3 Environment

The Cliff Walking environment is part of the Toy Text environments which contains general information about the environment. It involves crossing a grid world from start to goal while avoiding falling off a cliff.



3.1 Description

The game starts with the player at location [3, 0] of the 4x12 grid world with the goal located at [3, 11]. If the player reaches the goal the episode ends.

A cliff runs along [3, 1..10]. If the player moves to a cliff location it returns to the start location.

The player makes moves until they reach the goal.

3.2 Action Space

The action shape is (1,) in the range {0, 3} indicating which direction to move the player.

- 0: Move up
- 1: Move right
- 2: Move down
- 3: Move left

Actions that would move the agent outside the grid boundaries result in no movement (the agent stays in the same position).

3.3 Observation Space

There are 3 x 12 + 1 possible states. The player cannot be at the cliff, nor at the goal as the latter results in the end of the episode. What remains are all the positions of the first 3 rows plus the bottom-left cell.

The observation is a value representing the player's current position as current_row * ncols + current_col (where both the row and col start at 0).
For example, the starting position can be calculated as follows: 3 * 12 + 0 = 36.

The observation is returned as an int().

3.4 Reward

The reward system is designed to encourage finding the shortest safe path while severely penalizing dangerous moves:

- **Standard step**: -1 reward (encourages shorter paths)
- **Falling off cliff**: -100 reward + episode termination
- **Reaching goal**: 0 reward + episode termination

This reward structure creates an interesting dilemma: the shortest path (along the cliff edge) is risky but potentially more rewarding if executed perfectly, while the safe path (around the cliff) is longer but more reliable.

The episode terminates when the Agent reaches the goal position, falls of the cliff or the episode can exceeds maximum step limit.

4 Implementation

Our approach to solving the Cliff Walking problem implements and compares two fundamental reinforcement learning methodologies: Tabular Q-Learning and Deep Q-Networks (DQN). This comparison allows us to understand the strengths and limitations of each approach while demonstrating how different choices affect learning performance.

Both implementations use the code Q-Learning update rule, which is an off-policy temporal difference learning method.

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\big]$$

This update rule is off-policy, meaning it learns about the optimal policy while following a potentially different policy (epsilon-greedy). This property makes Q-learning robust and widely applicable.

4.1 Framework

Our solution utilizes the following Python libraries:

- **Gymnasium**: standard Reinforcement Learning environment interface

- **NumPy**: efficient numerical computations for tabular methods
- **PyTorch**: deep learning framework for DQN implementation
- **Matplotlib**: visualization of learning curves and performance metrics

The modular design allows easy experimentation with different hyperparameters and direct comparison between approaches under identical conditions.

4.2 Tabular Q-Learning Implementation Details

The algorithm follows the standard Q-Learning approach but is tailored for the Cliff Walking environment.

4.2.1 Initialize Q-table

The Q-table is initialized with zeros. Its shape corresponds to the number of state and the number of possible actions in the environment:

```
n_states = env.observation_space.n
n_actions = env.action_space.n
Q = np.zeros((n_states, n_actions))
```

4.2.2 Training Loop

For each episode, the environment is reset and the agent interacts with it until termination.

4.2.3 Action Selection (ε-greedy)

The agent chooses an action using the ε-greedy strategy: with probability ε it selects a random action (exploration), and otherwise it selects the action with the highest Q-value (exploitation).

```
if np.random.rand() < epsilon:
    action = env.action_space.sample()
else:
    action = np.argmax(Q[state])
```

4.2.4 Environment Step and Q-value Update

After takin an action, the agent observes the next state and reward. The Q-value is updated using the Temporal Difference (TD) learning rule:

```
old_value = Q[state, action]
best_next = np.max(Q[next_state])
td_target = reward + gamma * best_next * (not done)
td_error = td_target - old_value
Q[state, action] += alpha * td_error
```

4.2.5 Track Metrics

During the training, the following metrics are stored for later visualization:

- Episode reward: cumulative sum of the rewards per episode
- Epsilon: exploration rate over time
- TD errors: average temporal-difference errors per episode

4.2.6 Epsilon Decay

At the end of each episode, epsilon is decayed to gradually shift from exploration to exploitation:

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

4.3 Deep Q-Network (DQN) Implementation

Our DQN implementation uses a neural network to approximate Q-values, enabling more complex function approximation capabilities. The network architecture is configurable trough hyperparameters.

4.3.1 Network Architecture

```
class DQN(nn.Module):
def __init__(self, state_size, action_size):
    super().__init__()
    self.embedding = nn.Embedding(state_size, 16)

    layers = []
    input_size = 16
    for _ in range(DQN_HIDDEN_LAYERS):
        layers.append(nn.Linear(input_size, DQN_NODES_PER_LAYER))
        layers.append(nn.ReLU())
        input_size = DQN_NODES_PER_LAYER

    layers.append(nn.Linear(input_size, action_size))
    self.net = nn.Sequential(*layers)

def forward(self, x):
    x = self.embedding(x)
    return self.net(x)
```

- **State Embedding**: 16-dimensional embedding layer converts discrete states to dense representations
- **Configurable Depth**: number of hidden layers determined by hyperparameters
- **ReLU Activation**: non-linear activation functions enabling complex function approximation
- **Output Layer**: linear layer producing Q-values dor all actions

DQN incorporates Experience Replay Buffer to break temporal correlations and improve sample efficiency:

```
replay_buffer = deque(maxlen=10_000)

# Store experiences
replay_buffer.append((state, action, reward, next_state, done))

# Sample random batch for training
if len(replay_buffer) >= batch_size:
    batch = random.sample(replay_buffer, batch_size)
    # Process batch for network update
```

Benefits of Experience Replay:

- **Decorrelated Updates**: reduces harmful correlation between consecutive experiences
- **Sample Efficiency**: reuses experiences multiple times for training
- **Stable Learning**: smoother gradient updates trough diverse batch sampling

```
policy_net = DQN(n_states, n_actions)  # Updated every step
target_net = DQN(n_states, n_actions)  # Updated periodically

# Target network update
if episode % target_update_freq == 0:
    target_net.load_state_dict(policy_net.state_dict())
```

Target Network Purpose:

- **Stable Targets**: reduces moving target problem in Q-learning
- **Reduced Correlation**: separates action selection from target value computation
- **Improved Convergence**: more stable learning dynamics

Our implementation includes a comprehensive configuration system enabling systematic experimentation

5 Hyperparameters Description

```
RANDOM_START = True
SLIPPERY = True
EVALUATION_EPISODES = 1000

# Tabular Q-learning parameters
TAB_ALPHA = 0.1
TAB_GAMMA = 0.99
TAB_EPSILON = 1
TAB_MIN_EPSILON = 0.01
TAB_EPSILON_DECAY = 0.995
TAB_EPISODES = 500

# DQN parameters
DQN_BATCH_SIZE = 64
DQN_GAMMA = 0.99
```

```
DQN_EPSILON = 1
DQN_MIN_EPSILON = 0.01
DQN_EPSILON_DECAY = 0.995
DQN_BUFFER_SIZE = 1000
DQN_EPISODES = 500
DQN_TARGET_UPDATE_FREQ = 10
DQN_GRAD_UPDATE_FREQ = 4
DQN_HIDDEN_LAYERS = 2
DQN_NODES_PER_LAYER = 32
```

- **RANDOM_START**: enables random starting positions for each episode instead of always starting from the same state
- **SLIPPERY**: introduces stochasticity in action execution (actions have probability of not executing as intended)
- **EVALUATION_EPISODES**: number of episodes used to evaluate agent performance after training
- **TAB_ALPHA** (Learning Rate): controls how much new information overrides old Q-values in updates.
- **TAB_GAMMA** (Discount Factor): determines importance of future rewards relative to immediate rewards
- **TAB_EPSILON** (Initial Exploration Rate): starting probability of taking random actions for exploration
- **TAB_MIN_EPSILON** (Minimum Exploration Rate): lowest bound for exploration probability
- **TAB_EPSILON_DECAY** (Exploration Decay Rate): rate at which exploration decreases per episode
- **TAB_EPISODES** (Training Episodes): total number of training episodes for Tabular Q-Learning
- **DQN_BATCH_SIZE** (Mini-batch Size): number of experiences sampled from replay buffer for each training step
- **DQN_GAMMA** (Discount Factor): same as tabular version but affects neural network target computation
- **DQN_EPSILON/DQN_MIN_EPSILON/DQN_EPSILON_DECAY**: same exploration-exploitation mechanism as tabular version
- **DQN_BUFFER_SIZE** (Replay Buffer Size): maximum number of experiences stored in replay buffer
- **DQN_EPISODES** (Training Episodes): total training episodes for DQN
- **DQN_TARGET_UPDATE_FREQ** (Target Network Update Frequency): how often (in episodes) the target network copies weights from policy network

- **DQN_GRAD_UPDATE_FREQ** (Gradient Update Frequency): how often (in environment steps) to perform network parameter updates
- **DQN_HIDDEN_LAYERS** (Network Depth): number of hidden layers in the neural network
- **DQN_NODES_PER_LAYER** (Network Width): number of neurons in each hidden layer

6 Evaluation Framework

Both algorithms share a common evaluation framework ensuring fair comparison:

```
MAX_EVAL_STEPS = 500
MAX_CONSECUTIVE_REPEAT = 10  # episode ends if agent repeats a state this many times in a row

def evaluate_agent(device, env, policy_net=None, Q=None, tabular=True, random_start=False):
    episodes = config.EVALUATION_EPISODES

    successes, falls, total_rewards, steps_list = 0, 0, [], []

    agent = "DQN" if not tabular else "Tabular Q-learning"
    if not tabular:
        policy_net.eval()

    for ep in trange(episodes, desc="Evaluating " + agent):
        state, _ = env.reset()
        done, ep_reward, steps = False, 0, 0

        last_state = None
        repeat_count = 0

        while not done and steps < MAX_EVAL_STEPS:
            if config.RENDER:
                print(env.render())

            # Select action
            if tabular:
                action = np.argmax(Q[state])
            else:
                state_tensor = torch.tensor([state], device=device)
                with torch.no_grad():
                    action = policy_net(state_tensor).argmax(dim=1).item()

            next_state, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated
            ep_reward += reward
            steps += 1

            if last_state == next_state:
                repeat_count += 1
            else:
                repeat_count = 0
```

```
            last_state = next_state

            # Trigger early stop if stuck in a repeated state
            if repeat_count >= MAX_CONSECUTIVE_REPEAT:
                if config.DEBUG:
                    print(f"Stuck in state {next_state} for {MAX_CONSECUTIVE_REPEAT} steps. Ending episode early.")
                break

            state = next_state

            if reward == -100:
                falls += 1
            if done and next_state == (env.observation_space.n - 1):
                successes += 1

        total_rewards.append(ep_reward)
        steps_list.append(steps)

        if steps >= MAX_EVAL_STEPS and config.DEBUG:
            print("Max steps reached, ending episode.")

    if not tabular:
        policy_net.train()

    print(f"Evaluation Results (over {episodes} episodes):")
    print(f"  Success Rate: {successes}/{episodes} ({successes/episodes*100:.1f}%)")
    print(f"  Cliff Falls: {falls}")
    print(f"  Avg Total Reward: {np.mean(total_rewards):.2f}")
    print(f"  Avg Steps per Episode: {np.mean(steps_list):.2f}")
```

Evaluation Metrics:

- **Success Rate**: percentage of terminated episodes
- **Cliff Falls**: number of cliff accidents
- **Average Reward**: mean cumulative reward per episode
- **Average Steps**: mean episodes length, indicating path efficiency

7 Testing

To validate our implementation we performed various tasting on different hyperparameters to simulate different situations.