# Deep Reinforcement Learning for Developers

Abdul Hagi

# Chapter 1: Introduction to Reinforcement Learning

## Part 1: Understanding the Fundamentals

### Overview

Reinforcement learning (RL) is a paradigm in machine learning that provides a framework for agents to learn how to behave in an environment by performing actions and seeing the results. This chapter introduces RL, explores its key concepts, and applies them in a simple Gridworld environment.

### Objectives

- Understand the core principles of RL.
- Identify RL applications in different industries.
- Establish a foundational project in RL.

### What is Reinforcement Learning?

RL involves an agent that learns to make decisions by taking actions in an environment to maximize some notion of cumulative reward. It's characterized by trial and error, feedback loops, and adaptability to changing situations.

### Key Components of RL

- **Agent**: The learner or decision-maker.
- **Environment**: Where the agent takes actions.
- **Action**: A set of operations that the agent can perform.
- **State**: The current situation of the environment.
- **Reward**: Feedback from the environment to the agent.

### Why Reinforcement Learning?

Reinforcement learning is unique in its approach to problem-solving, allowing agents to learn from their own experiences rather than being told the correct actions. This is particularly useful in complex, unpredictable environments or when the desired behavior is difficult to express with explicit rules.

### Applications of RL

- **Game playing**: Achieving superhuman performance in complex games.
- **Robotics**: Teaching robots to perform tasks autonomously.
- **Autonomous vehicles**: Driving safely and efficiently in dynamic environments.

### Project Overview: The Gridworld

The Gridworld is an introductory RL project where an agent must navigate through a grid to reach a goal while avoiding obstacles.

**Project Goal** Create an agent that finds the shortest path to a goal within a grid, considering obstacles.

**Key Learning** This project will teach you about the interaction between an agent and its environment, the role of rewards, and how to implement these concepts in code.

### Coding the Gridworld Environment

We construct a `Gridworld` class in Python to simulate the environment for our agent.

- File: `gridworld.py`

```python
# gridworld.py
import numpy as np

class Gridworld:
    def __init__(self, width, height, start, goal, obstacles):
        """
        Initializes a Gridworld object.

        Parameters:
        - width (int): The width of the grid.
        - height (int): The height of the grid.
        - start (tuple): The starting position of the agent as a tuple (x, y).
        - goal (tuple): The goal position of the agent as a tuple (x, y).
        - obstacles (list): A list of obstacle positions as tuples [(x1, y1), (x2, y2), ....
        """
        self.width = width
        self.height = height
        self.start = start
        self.goal = goal
        self.obstacles = obstacles
        self.reset()

    def step(self, action):
        """
        Takes an action in the environment and updates the agent's position.

        Parameters:
        - action (tuple): The action to be taken.

        Returns:
        - new_state: The new state after taking the action.
        - reward: The reward received after taking the action.
        - done: A boolean indicating if the goal has been reached.
        """
        # Calculate the new position after taking the action
        new_x = self.agent_position[0] + action[0]
        new_y = self.agent_position[1] + action[1]

        # Check if the new position is within the grid bounds and not an obstacle
        if (0 <= new_x < self.width) and (0 <= new_y < self.height) and not (new_x, new_y) i
            # Update the agent's position
            self.agent_position = (new_x, new_y)

        # Check if the new position is the goal
        done = self.agent_position == self.goal

        # Define the reward for reaching the goal or taking a step
        reward = 0 if done else -1

        return self.get_state(), reward, done

    def reset(self):
        """
        Resets the grid and agent position to the initial state.
        """
        self.agent_position = self.start
        self.grid = np.zeros((self.height, self.width))
        for obstacle in self.obstacles:
            self.grid[obstacle] = -1
        self.grid[self.goal] = 1
        return self.get_state()  # Return the initial state

    def render(self):
        """
```

```python
        Renders the current state of the gridworld.
        """
        for y in range(self.height):
            for x in range(self.width):
                if (x, y) == self.agent_position:
                    print('A', end=' ')  # Agent's position
                elif (x, y) == self.goal:
                    print('G', end=' ')  # Goal position
                elif (x, y) in self.obstacles:
                    print('#', end=' ')  # Obstacle
                else:
                    print('.', end=' ')  # Empty cell
            print()  # Newline at the end of each row

        print()  # Extra newline for better separation between steps

# Example usage
gridworld = Gridworld(width=5, height=5, start=(0, 0), goal=(4, 4), obstacles=[(1, 1), (2, 2
gridworld.render()
```

### Summary

This section established the basics of RL and introduced a simple Gridworld environment for future exploration.

## Part 2: Building the Agent

### Implementing the Agent

We now add an agent to interact with the Gridworld, beginning with a basic agent that makes random moves.

### Agent and Policy

- **Agent**: The entity that acts in the environment.
- **Policy**: The decision-making strategy of the agent.

**The RandomAgent** We implement a `RandomAgent` class, which serves as our initial, naive agent.

- File: `gridworld_agent.py`

```python
# gridworld_agent.py
import random

class RandomAgent:
    def __init__(self, actions):
        """
        Initializes a RandomAgent object.

        Parameters:
        - actions (list): A list of possible actions the agent can take.
        """
        self.actions = actions

    def choose_action(self, state):
        """
        Chooses the next action at random from the list of possible actions.

        Parameters:
        - state: The current state of the agent (not used in this random policy).

        Returns:
        A randomly selected action from the agent's list of possible actions.
        """
        return random.choice(self.actions)
```

**Agent Actions in Gridworld** The agent can move in four directions: up, down, left, and right.

**Interactions and Outcomes** We simulate the agent's interactions with the Gridworld, using a reward system to guide its learning.

**Running the Agent** We execute the agent within the Gridworld, observing its behavior over time.

### Summary and Next Steps

We introduced a simple agent to our Gridworld project. This sets the foundation for more advanced learning algorithms to come.

---

# Full Chapter 1: The Complete Code

We combine the elements from Part 1 and Part 2 to run our Gridworld simulation with the RandomAgent. The code for the agent's interaction with the Gridworld is in the `gridworld_random_agent.py` file.

- File: `gridworld_random_agent.py`

```
# gridworld_random_agent.py
from gridworld import Gridworld
from gridworld_agent import RandomAgent
```

```
# Define the Gridworld environment
gridworld = Gridworld(width=5, height=5, start=(0, 0), goal=(4, 4), obstacles=[(1, 1), (2, 2

# Define the actions and create the RandomAgent
actions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Actions corresponding to [Up, Right, Down, L
agent = RandomAgent(actions)

total_reward = 0

# Run the agent in the environment
for _ in range(100):  # Run for a certain number of steps or until the goal is reached
    current_state = gridworld.agent_position
    action = agent.choose_action(current_state)
    new_state, reward, done = gridworld.step(action)
    total_reward += reward
    print(f"Action taken: {action}")
    print(f"Reward received: {reward}")
    print(f"Total reward: {total_reward}")
    gridworld.render()
    if done:
        print("Goal reached!")
        break
```

### Why These Actions?

We selected standard grid movement actions for simplicity and to establish a baseline for agent behavior.

### Following Equations and Logic

At this stage, the agent's decisions are random. We'll later introduce more sophisticated strategies based on RL algorithms and mathematical foundations.

### Summary

Chapter 1 sets the stage for an exploration into reinforcement learning, providing the fundamental concepts and a practical project to apply these ideas. As we progress, the agents will evolve from making random moves to employing advanced strategies informed by their interactions with the environment.

# Chapter 2: Introduction to Q-Learning

## Part 1: Theoretical Foundations

### Overview

This chapter explores the concept of Q-Learning, a model-free reinforcement learning algorithm that enables agents to learn optimal action-value functions and derive an optimal policy.

### Objectives

- Introduce the Q-Learning algorithm and its role in reinforcement learning.
- Understand the components and operation of the Q-Learning algorithm.
- Prepare for the practical application of Q-Learning in the Gridworld project.

### What is Q-Learning?

Q-Learning is a method that allows agents to learn the value of an action in a particular state, guiding them to make optimal decisions through trial and error.

### Key Concepts of Q-Learning

- **Q-value (Q-function)**: An estimation of the total expected rewards an agent can get, given a state and action.
- **Q-table**: A table where Q-values for each state-action pair are stored.

### Why Q-Learning?

Q-Learning is advantageous as it can handle environments with stochastic transitions and rewards without requiring a model of the environment.

### The Q-Learning Algorithm

The Q-Learning algorithm is a reinforcement learning technique used to solve Markov Decision Processes (MDPs). It learns an optimal policy by iteratively updating the Q-values of state-action pairs based on the rewards received.

Algorithm Steps: Initialize the Q-table with zeros for all state-action pairs.

Set the learning rate (alpha), discount factor (gamma), and exploration rate (epsilon).

Repeat the following steps until convergence or a maximum number of episodes:

a. Observe the current state (s).

b. Choose an action (a) based on the current state using an exploration-exploitation strategy (e.g., epsilon-greedy).

c. Perform the chosen action and observe the next state (s') and the reward (r).

d. Update the Q-value of the current state-action pair using the Q-Learning update equation:

Q(s, a) = (1 - alpha) * Q(s, a) + alpha * (r + gamma * max(Q(s', a'))) e. Update the current state (s) to the next state (s').

Return the learned Q-table, which represents the optimal action-value function.

Parameters: alpha (learning rate): Determines the weight given to the new information when updating the Q-values. A higher value gives more weight to the new information.

gamma (discount factor): Controls the importance of future rewards. A value closer to 1 considers long-term rewards, while a value closer to 0 focuses on immediate rewards.

epsilon (exploration rate): Determines the balance between exploration and exploitation. A higher value encourages more exploration, while a lower value favors exploitation of the learned Q-values.

The Q-Learning algorithm is a powerful technique for solving reinforcement learning problems. It allows an agent to learn an optimal policy by iteratively updating the Q-values based on the rewards received. By balancing exploration and exploitation, the agent can find the best actions to take in different states to maximize its cumulative reward.

### Coding Q-Learning Components

To implement Q-Learning in our Gridworld project, we will create the Q-table and implement the update rule as part of the agent's learning algorithm.

### Summary

In this section, we've introduced the theoretical underpinnings of Q-Learning, setting the stage for a practical implementation in the Gridworld environment.

## Part 2: Practical Implementation

### Implementing Q-Learning in Gridworld

To extend our Gridworld project with the Q-Learning algorithm, we will add functionality to our agent to learn from the environment using a Q-table.

**Q-table Initialization** We initialize a Q-table that maps each state-action pair to its Q-value, which will be updated as the agent learns.

**Q-Learning Update** After taking an action, we update the Q-table entries using the Q-Learning formula, which incorporates the reward received and the estimated value of the next state.

**Exploration vs. Exploitation** We'll implement an $\epsilon$-greedy strategy, enabling the agent to explore the environment while exploiting its current knowledge.

### Enhancing the Agent with Q-Learning

We upgrade our `RandomAgent` to a `QLearningAgent` that can choose actions based on learned Q-values and improve its decisions over time.

**File: `q_learning_agent.py`**

```python
# q_learning_agent.py
import numpy as np
import random

class QLearningAgent:
    def __init__(self, state_space, action_space, alpha, gamma, epsilon, action_mapping):
        self.q_table = np.zeros((state_space, action_space))
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon = epsilon
        self.action_space = action_space
        self.action_mapping = action_mapping  # Add this line to store the action mapping

    def choose_action(self, state):
        if random.uniform(0, 1) < self.epsilon:
            action_index = random.randint(0, self.action_space - 1)  # Explore action space
        else:
            action_index = np.argmax(self.q_table[state])  # Exploit learned values

        # Return the action index instead of the action itself
        return action_index

    def update_q_table(self, current_state, action_index, reward, next_state):
        # Find the best action at the next state
        best_next_action = np.argmax(self.q_table[next_state])

        # Calculate the target for the Bellman equation
        td_target = reward + self.gamma * self.q_table[next_state, best_next_action]

        # Calculate the difference between the target and the current Q-value
        td_delta = td_target - self.q_table[current_state, action_index]
        # Update the Q-value for the current state-action pair
        self.q_table[current_state, action_index] += self.alpha * td_delta
```

**Running the Q-Learning Agent** We will simulate the `QLearningAgent` within the Gridworld, observing how it learns from experiences to improve its navigation strategy.

**File: `gridworld_q_learning.py`**

```
# gridworld_q_learning.py
from gridworld import Gridworld
from q_learning_agent import QLearningAgent

# Define the Gridworld environment dimensions and attributes
grid_width = 5
grid_height = 5
num_states = grid_width * grid_height
num_actions = 4  # Up, down, left, and right

# Define action mapping that maps action indices to actual actions (movements)
action_mapping = [(0, -1), (1, 0), (0, 1), (-1, 0)]  # Up, Right, Down, Left

# Initialize the Gridworld environment
gridworld = Gridworld(width=grid_width, height=grid_height, start=(0, 0), goal=(4, 4), obsta

# Initialize the Q-Learning agent with hyperparameters alpha, gamma, and epsilon
alpha = 0.1  # Learning rate
gamma = 0.9  # Discount factor
epsilon = 0.1  # Exploration rate
agent = QLearningAgent(state_space=num_states, action_space=num_actions, alpha=alpha, gamma=

# Run the Q-Learning agent in the Gridworld
num_episodes = 20  # Total number of episodes to run
for episode in range(num_episodes):
    done = False
    total_reward = 0
    gridworld.reset()
    current_state = gridworld.get_state()

    print(f"Starting episode {episode + 1}")



    while not done:

        # Agent chooses an action index
        action_index = agent.choose_action(current_state)

        # Perform the chosen action and get the result
        _, reward, done = gridworld.step(agent.action_mapping[action_index])

        # Convert the next position to the next state index
        next_state = gridworld.get_state()  # Assuming Gridworld has this method

        # Update the Q-table with the action index
        agent.update_q_table(current_state, action_index, reward, next_state)

        # Update total reward and current state
        total_reward += reward
        current_state = next_state

    print(f"Episode {episode + 1}: Total Reward: {total_reward}")
```

### Summary

We've now set the theoretical foundation and implemented the Q-Learning algorithm in our Gridworld agent. The agent's ability to learn from interactions with the environment will be demonstrated in the following simulation.

## Part 3: Updating the Gridworld Environment

### Enhancing State Representation

To facilitate our Q-Learning agent's interaction with the Gridworld, we must convert the environment's grid coordinates into a state index.

### Objectives

- Enable the Gridworld to provide a state index representation for the Q-Learning agent.
- Update the Gridworld class with a `get_state()` method.

### The `get_state()` Method

We add a method to the `Gridworld` class that will map the agent's 2D grid coordinates to a unique state index.

### Implementation

We will implement the `get_state()` method to provide a state representation by flattening the grid.

**File: `gridworld.py`**

```python
class Gridworld:
    # Existing methods...

    def get_state(self):
        """
        Returns a unique state index for the agent's current position.
        """
        x, y = self.agent_position
        return y * self.width + x
```

### Summary and Integration

The `Gridworld` class now provides a state index that our Q-Learning agent can use to update the Q-table and make decisions. This integration is crucial for the upcoming simulations where the agent learns to navigate the environment more effectively.

### Next Steps

With state representation in place, we can proceed to initialize the Q-table and run the Q-Learning agent through training episodes to observe its learning and performance improvements. The next sections of this chapter will focus on these aspects, leading to a fully functional Q-Learning agent in the Gridworld.

# Chapter 3: Fundamentals of Deep Learning for Reinforcement Learning

## Overview

This chapter delves into the synergy between deep learning and reinforcement learning, leading to the burgeoning field of deep reinforcement learning (DRL). We aim to elucidate the essential principles of deep learning and show how neural networks, when paired with RL, can effectively address intricate decision-making challenges.

## Objectives

- Understand the fundamental concepts of deep learning, with a focus on neural networks.
- Learn how deep learning can be integrated with RL algorithms to improve their performance.
- Prepare to implement a deep Q-network (DQN) in the Gridworld project.

## Introduction to Neural Networks

Neural networks are designed to mimic the workings of the human brain and are made up of interconnected units called neurons. These networks have the capacity to learn from data, enabling them to perform tasks such as predictions or decision-making without the need for explicit programming.

### Key Concepts in Neural Networks

- **Neurons**: The building blocks of neural networks that take input, process it using a weighted sum followed by an activation function, and produce an output.
- **Layers**: A neural network comprises different layers of neurons, which include the input layer, one or more hidden layers, and the output layer.
- **Weights and Biases**: Parameters of the model that adjust during training to make the neural network's predictions as accurate as possible.

## How Neural Networks Learn

The process of learning in neural networks involves adjusting their weights and biases. The goal is to reduce the discrepancy between the model's predictions and the actual target values.

- **Forward Propagation**: This is the process where input data is passed through the network to generate output predictions.
- **Loss Functions**: These are measures used to assess the deviation between the network's predictions and the true outcomes.
- **Backpropagation**: A key algorithm in neural network training, backpropagation calculates the gradient of the loss function with respect to each weight and bias, using these gradients to update the parameters via gradient descent.

## Deep Learning in Reinforcement Learning

Deep learning equips RL with powerful function approximation capabilities, essential for dealing with high-dimensional state or action spaces.

### Deep Reinforcement Learning

- **Function Approximation**: The application of deep neural networks to estimate value functions or policy functions.
- **Advantages of Deep RL**: These include the ability to process complex inputs, generalize across different states, and discern underlying patterns in the data.

## Enhancing Gridworld with Deep Learning

To integrate deep learning into our Gridworld example, we will employ a deep Q-network (DQN). A DQN utilizes a neural network to estimate the Q-value function, a central concept in RL that predicts the quality of actions taken in various states.

- **Project Update**: We will implement a DQN to substitute the Q-table with a neural network that predicts Q-values for Gridworld.
- **Implementation Steps**: We outline the necessary modifications to the existing Gridworld project to incorporate a DQN.

## Summary

We recap the deep learning concepts introduced in this chapter and discuss their relevance to RL. This provides the foundation for a hands-on implementation of a DQN in the following sections.

---

## Implementing a Deep Q-Network (DQN) with PyTorch

We will utilize PyTorch, a widely-used deep learning framework, to create our DQN model. PyTorch offers a comprehensive and intuitive environment for constructing neural networks.

**File: `gridworld_deep_q_learning.py`**

```
import numpy as np
import random
import torch
import torch.nn as nn
import torch.nn.functional as F
from gridworld import Gridworld  # This imports the Gridworld environment

# Define the neural network architecture for Deep Q-Learning
class SimpleMLP(nn.Module):
    # Constructor for the neural network
    def __init__(self, input_size, output_size, hidden_size=64):
        super(SimpleMLP, self).__init__()
        # First fully connected layer from input_size to hidden_size
        self.fc1 = nn.Linear(input_size, hidden_size)
        # Second fully connected layer, hidden to hidden
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        # Third fully connected layer from hidden_size to output_size
        self.fc3 = nn.Linear(hidden_size, output_size)

    # Forward pass definition for the neural network
    def forward(self, x):
        # Apply ReLU activation function after first layer
        x = F.relu(self.fc1(x))
        # Apply ReLU activation function after second layer
        x = F.relu(self.fc2(x))
        # Output layer, no activation function
        return self.fc3(x)

# Deep Q-Network agent class definition
class DQNAgent:
    # Constructor for the DQN agent
    def __init__(self, input_size, output_size, hidden_size=64):
        # The neural network model
        self.model = SimpleMLP(input_size, output_size, hidden_size)
        # Epsilon for the epsilon-greedy policy (initially set to 1 for full exploration)
        self.epsilon = 1.0
        # Minimum value that epsilon can decay to over time
        self.epsilon_min = 0.01
        # Rate at which epsilon is decayed over time
        self.epsilon_decay = 0.995
        # List to hold past experiences for replay
        self.memory = []

    # Method to decide an action based on the current state
    def act(self, state):
        # Check if we should take a random action (exploration)
        if np.random.rand() <= self.epsilon:
            # Return a random action within the action space size
            return random.randrange(output_size)

        # If not exploring, process the state through the DQN to get the action values
        # First, ensure state is a PyTorch tensor
        if not isinstance(state, torch.Tensor):
            state = torch.from_numpy(state).float().unsqueeze(0)

        # Forward pass through the network to get action values
        action_values = self.model(state)
        # Return the action with the highest value
        return np.argmax(action_values.detach().numpy())

    # Method to store experiences in replay memory
    def remember(self, state, action, reward, next_state, done):
        # Append the experience as a tuple to the memory list
        self.memory.append((state, action, reward, next_state, done))

    # Method to decay epsilon over time for less exploration
    def update_epsilon(self):
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay

# Function to train the neural network model
def train_model(agent, optimizer, batch_size, gamma):
    # Check that there are enough experiences in memory to sample a batch
    if len(agent.memory) < batch_size:
        return
    # Sample a minibatch of experiences from memory
    minibatch = random.sample(agent.memory, batch_size)
    # Unpack the experiences
    states, actions, rewards, next_states, dones = zip(*minibatch)

    # Convert experience components to PyTorch tensors
    states = torch.from_numpy(np.vstack(states)).float()
    actions = torch.from_numpy(np.vstack(actions)).long()
    rewards = torch.from_numpy(np.vstack(rewards)).float()
    next_states = torch.from_numpy(np.vstack(next_states)).float()
    dones = torch.from_numpy(np.vstack(dones).astype(np.uint8)).float()

    # Calculate the expected Q values from the neural network
    Q_expected = agent.model(states).gather(1, actions)
    # Calculate the Q value for the next states and get the max Q value for each next state
    Q_targets_next = agent.model(next_states).detach().max(1)[0].unsqueeze(1)
    # Calculate the target Q values for the current states using the Bellman equation
    Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))

    # Calculate the loss between the expected Q values and the target Q values
    loss = F.mse_loss(Q_expected, Q_targets)
    # Backpropagate the loss, update the network weights
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

# Function to log the performance of the agent
def log_performance(episode, total_reward, steps, epsilon):
    # Print out the episode number, total reward, number of steps and the epsilon value
    print(f"Episode: {episode}, Total Reward: {total_reward}, Steps: {steps}, Epsilon: {eps

# Example usage of the defined classes and functions
if __name__ == "__main__":
    # Initialize the Gridworld environment
    gridworld = Gridworld(width=5, height=5, start=(0, 0), goal=(4, 4), obstacles=[(1, 1), (
    # Define action mapping that maps action indices to movements
    action_mapping = [(0, -1), (1, 0), (0, 1), (-1, 0)]  # Up, Right, Down, Left
```

```
# Calculate the input size based on the environment's state space
input_size = gridworld.width * gridworld.height
# The output size is the number of possible actions
output_size = 4
# Initialize the DQN agent
agent = DQNAgent(input_size, output_size)
# Define an optimizer for the neural network (Adam optimizer with a learning rate of 0.
optimizer = torch.optim.Adam(agent.model.parameters(), lr=0.001)
# Batch size for experience replay
batch_size = 32
# Discount factor for future rewards
gamma = 0.99

# Function to convert a state to a tensor for neural network input
def state_to_tensor(state, grid_size=5):
    # Create a one-hot encoded vector for the state
    state_vector = torch.zeros(grid_size * grid_size, dtype=torch.float32)
    state_vector[state] = 1
    return state_vector

# Main training loop
for episode in range(100):
    # Reset the environment at the start of each episode
    state = gridworld.reset()
    # Initialize total reward and steps counter
    total_reward = 0
    done = False
    steps = 0

    # Loop for each step in the episode
    while not done:
        # Convert state to tensor format
        state_vector = state_to_tensor(state)
        # Select an action using the DQN agent's policy
        action = agent.act(state_vector)
        # Take the action in the environment and observe the next state and reward
        next_state, reward, done = gridworld.step(action_mapping[action])
        # Convert the next state to tensor format
        next_state_vector = state_to_tensor(next_state)
        # Remember the experience
        agent.remember(state_vector, action, reward, next_state_vector, done)

        # Move to the next state
        state = next_state
        # Update the total reward
        total_reward += reward
```

```
            # Increment the step counter
            steps += 1

            # Train the model with experiences from memory
            train_model(agent, optimizer, batch_size, gamma)

        # After the episode, decay epsilon for less exploration in future episodes
        agent.update_epsilon()
        # Log the performance metrics for the episode
        log_performance(episode, total_reward, steps, agent.epsilon)
```

In this code snippet, we have outlined the creation of a simple feedforward neural network model using PyTorch for approximating Q-values in the Gridworld environment. The code includes a DQN agent that selects actions using an epsilon-greedy policy and stores experiences in a replay buffer. The `train_model` function updates the neural network's weights using sampled experiences to reduce the loss between predicted and target Q-values.

Please note that this example is a simplified version of DQN. For practical and more complex scenarios, additional mechanisms like experience replay buffers with more sophisticated sampling techniques and separate target networks to stabilize the Q-value predictions are recommended.