# Module 6

**Introduction to Reinforcement Learning with Keras**

**Reinforcement Learning, Q-Learning, Q-Networks (DQNs)**

## 📌 Introduction to Reinforcement Learning (RL)

### 🔹 Reinforcement Learning Overview

Reinforcement learning is a machine learning paradigm where an agent interacts with an environment by selecting actions to maximize rewards over time.

The interaction forms a continuous feedback loop:

* **Agent**: The entity that makes decisions.

In games, it’s the player; in web applications, it might be a program that places ads.

* **Environment**: The space in which the agent operates.

In games, this could be the game board; for web applications, it’s the webpage.

* **Actions**: The choices available to the agent at any given state. These alter the environment and trigger feedback.
* **Reward**: The signal returned by the environment to indicate the quality of the action taken.

An action taken by the agent impacts the environment.

The environment then returns a reward to guide the agent's learning. These rewards are often delayed and uncertain, requiring the agent to estimate them over time.

The process repeats dynamically, as actions continuously alter the environment, making the state and outcome fluid and non-static.

### 🔹 Real-World Examples of RL

Recent advances in deep learning have expanded RL applications:

* **DeepMind (2013)**: Created a system that learned to play Atari games and outperformed humans.
* **AlphaGo (2017)**: Beat the world champion in the complex board game Go.

While **RL** has proven successful in games, it **has high data and computational demands due to the vast number of possible states and actions**.

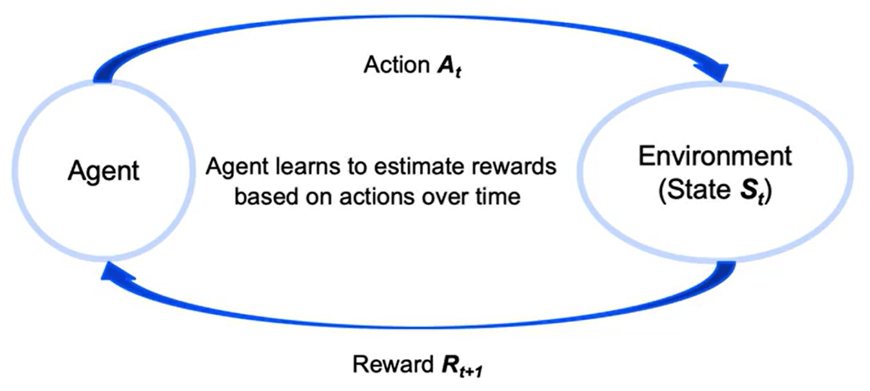
This challenge has slowed its broader adoption but hasn't stopped progress in practical business scenarios, including:

* **Recommendation Engines**: Reward defined by correct user predictions.
* **Marketing Optimization**: Rewards based on clicks or revenue.
* **Automated Bidding**: Rewards optimized around spending per item.

### 🔹 Policy and Rewards

In RL the agent takes some action, that action affects the **current environment**, and then **feedback** from that environment is passed back to the agent in terms of a **reward**.

* If the **feedback resulted in a positive result** in relation to our reward system, the agent’s actions are reinforced.
* Vice-versa for **negative results**, if it ended up in a bad environment state, then the agent is **reinforced not to take those same steps.**



The **policy** is the strategy an agent uses to decide which actions to take given a particular state.

The agent’s goal is to learn a policy that maximizes expected cumulative rewards over time.

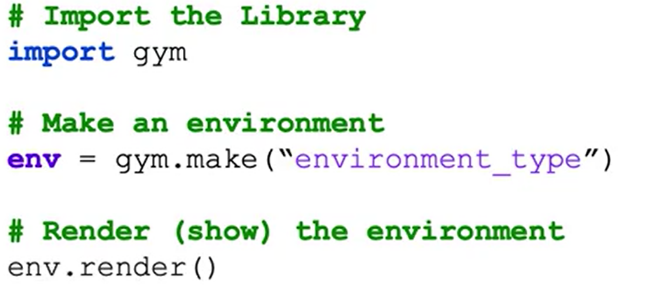
Unlike supervised learning, rewards are not labeled and are often unpredictable or delayed.

Agents must balance **short-term and long-term rewards**, which introduces a critical trade-off in decision-making.

As a result, reinforcement learning problems are highly dynamic. Each action changes the state, which changes the nature of the problem the agent is trying to solve.

### 🔹 Reinforcement Learning in Python

The implementation of RL in Python often uses the **OpenAI Gym** library, which provides predefined environments for training and evaluation:



* ***gym.make()*:** Instantiates a simulation environment.
* ***render()*:** Visualizes the environment state.

These tools allow interaction with simulated environments such as games or virtual physical systems, making it easier to test reinforcement learning agents.

### ✅ Takeaways

✅ Reinforcement learning enables agents to interact with environments by selecting actions that maximize rewards over time.

✅ Rewards are typically unknown in advance and must be estimated through repeated interaction with the environment.

✅ Reinforcement learning differs from traditional supervised learning because the feedback is not based on labeled outputs, but on delayed and sometimes uncertain rewards.

✅ Recent RL advances include DeepMind’s Atari systems and AlphaGo, with growing business applications in recommendation systems and bidding strategies.

✅ Python implementation often uses the OpenAI Gym library to simulate and visualize RL environments.

## 📌 Q-Learning with Keras

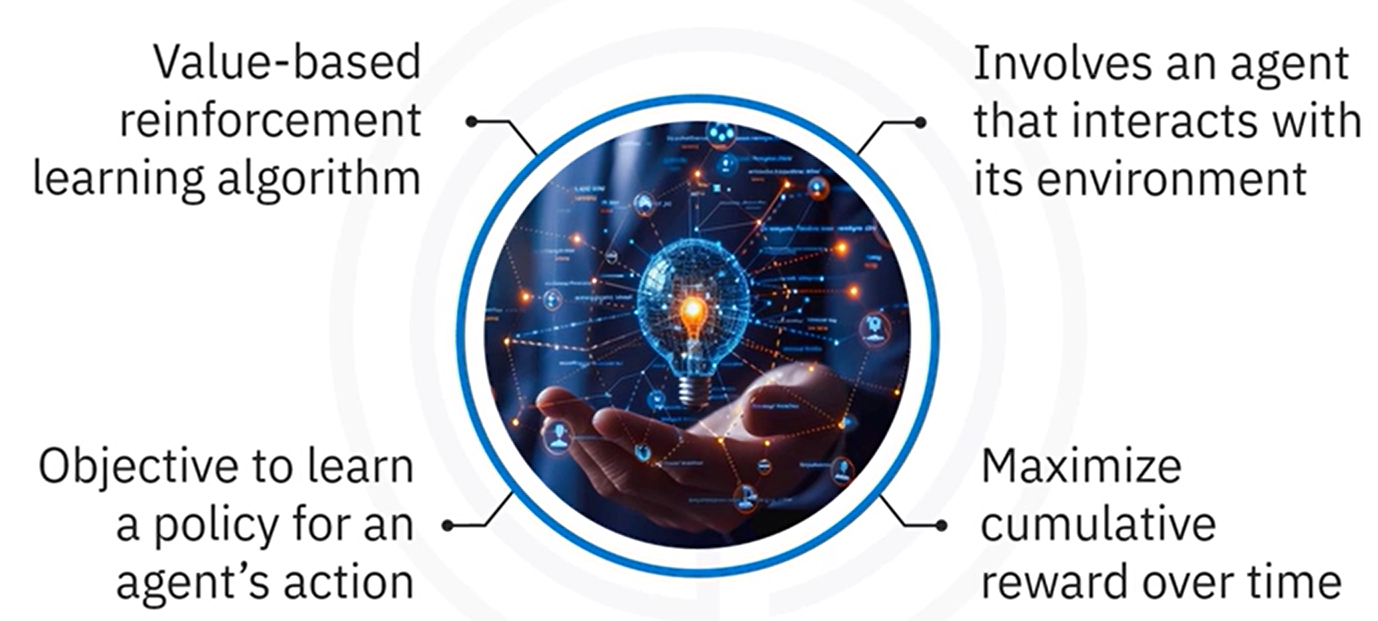
Q-learning is a foundational reinforcement learning algorithm that focuses on training agents to make sequential decisions by maximizing cumulative rewards.

### 🔹 Understanding Q-Learning

Reinforcement learning is defined as a powerful paradigm in machine learning focused on training agents to make **sequences of decisions** by maximizing a **cumulative reward**.

Q-learning is a **value-based** and **off-policy** reinforcement learning algorithm:

* **Off-policy:** It learns the optimal policy independently of the agent’s current behavior
* **Value-based:** The agent learns to estimate the **value** of states or state-action pairs, which represent the expected cumulative reward. These values guide the agent in selecting actions that maximize long-term rewards.



**Q-Learning objective:**

**Learn a policy** that tells the agent what action to take under which state to **maximize cumulative future rewards**.

### 🔹 Q-Value Function and Bellman Equation

Q-learning relies on the **Q-value function**, denoted as which measures the **expected utility** of taking **action** “**a**” in **state** “**s**”, and then following the **optimal policy** afterward.

The Q-values are updated **iteratively** using the **Bellman equation**, which incorporates:

* The **immediate reward**
* The **estimated future rewards**

The updated rule for Q-value is given by the **Bellman Equation**:

* **s** 🡪 Current state
* **a** 🡪 Current action
* **r** 🡪 Reward received after taking action a
* **s'** 🡪 Resulting state from taking action a
* **a'** 🡪Next action
* **α** (alpha) 🡪 **Learning rate**, controlling how much new information overrides old knowledge
* **γ** (gamma) 🡪 **Discount factor**, determining the importance of future rewards

### 🔹 Q-learning implementation steps

Q-learning implementation consists of **multiple steps** that are essential for allowing the agent to learn and perform effectively:

🔸 **Initialize the environment and parameters:**

Define the environment, using a platform like **OpenAI Gym**

Initialize the **Q-table** with state-action pairs.

Set hyperparameters:

* Learning rate α
* Discount factor γ
* Exploration rate ε

🔸 **Build the Q-network:**

Storing Q-table is infeasible when the state space is large or continuous.

To handle complex environments, we can use a **neural network** function approximator to learn the Q-value function – this approach is known as a **Deep Q-Network (DQN)**.

Instead of a Q-table, a DQN uses a neural network (called the **Q-network**) to **predict Q(s,a)** for any given **state** “**s**” and **action** “**a**”.

This allows Q-learning to scale to high-dimensional or continuous state spaces by leveraging function approximation

* Use Keras to construct a Q-Network that approximates the Q-value function.
* The Q-network replaces the Q-table for large or continuous state spaces.

🔸 **Train the Q-network:**

Training the Q-network involves letting the agent interact with the environment repeatedly and updating the network’s weights based on the Q-learning rule.

We typically train over many **episodes**. In each episode, the agent starts in an initial state from the environment and then proceeds through a sequence of time-steps until the episode ends (for instance, when a termination condition like a pole falling or time limit is reached).

**Training loop** implementation should be as follows:

* The agent interacts with the environment.
* Selects actions.
* Receives rewards.
* Transitions to new states.
* Updates Q-values using the Bellman equation.

🔸 **Evaluate the agent:**

After training the Q-network, we need to **evaluate the agent** to see how well it has learned to solve the task.

Evaluation is typically done by deploying the agent in the environment **without** any exploratory randomness. In this phase, the agent uses the learned Q-network to select the action with the highest Q-value at each state, reliably exploiting its training.

Trained agent is tested in the environment, to assess:

* Overall performance.
* Ability to maximize rewards.

### 🔹 Q-learning implementation with Keras

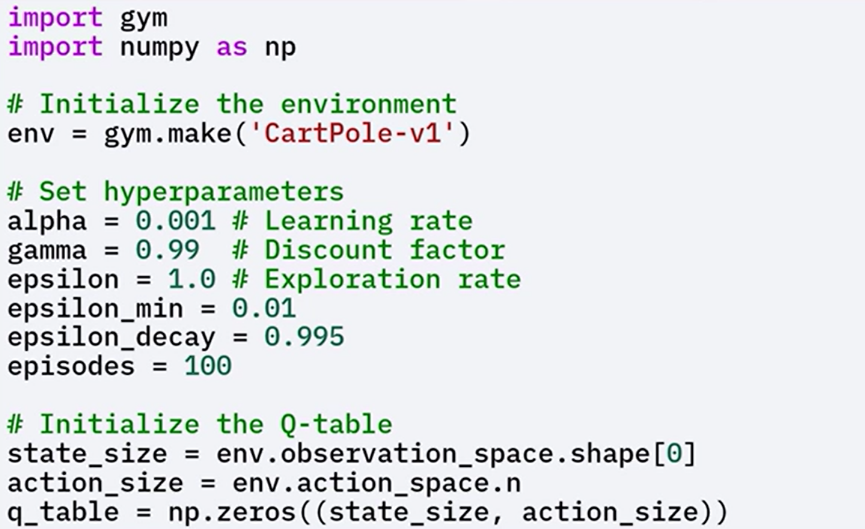
🔸 **Initialize the environment and parameters:**

The environment which the agent will interact, is from **OpenAI Gym**, **CartPole**.

CartPole is a classic control problem where the objective is to **balance a pole on a moving cart**.

The setup includes:

* Initilizing the Q-table with state-action pairs.
* Defining important parameters that directly influence learning quality and overall agent performance:
* Learning rate α
* Discount factor γ
* Exploration rate ε

The exploration rate Epsilon is initialized to 1.0 and decays over time to shift the agent's behavior from exploration to exploitation.

The state size and action size are determined based on the environment's observation and action spaces respectively.

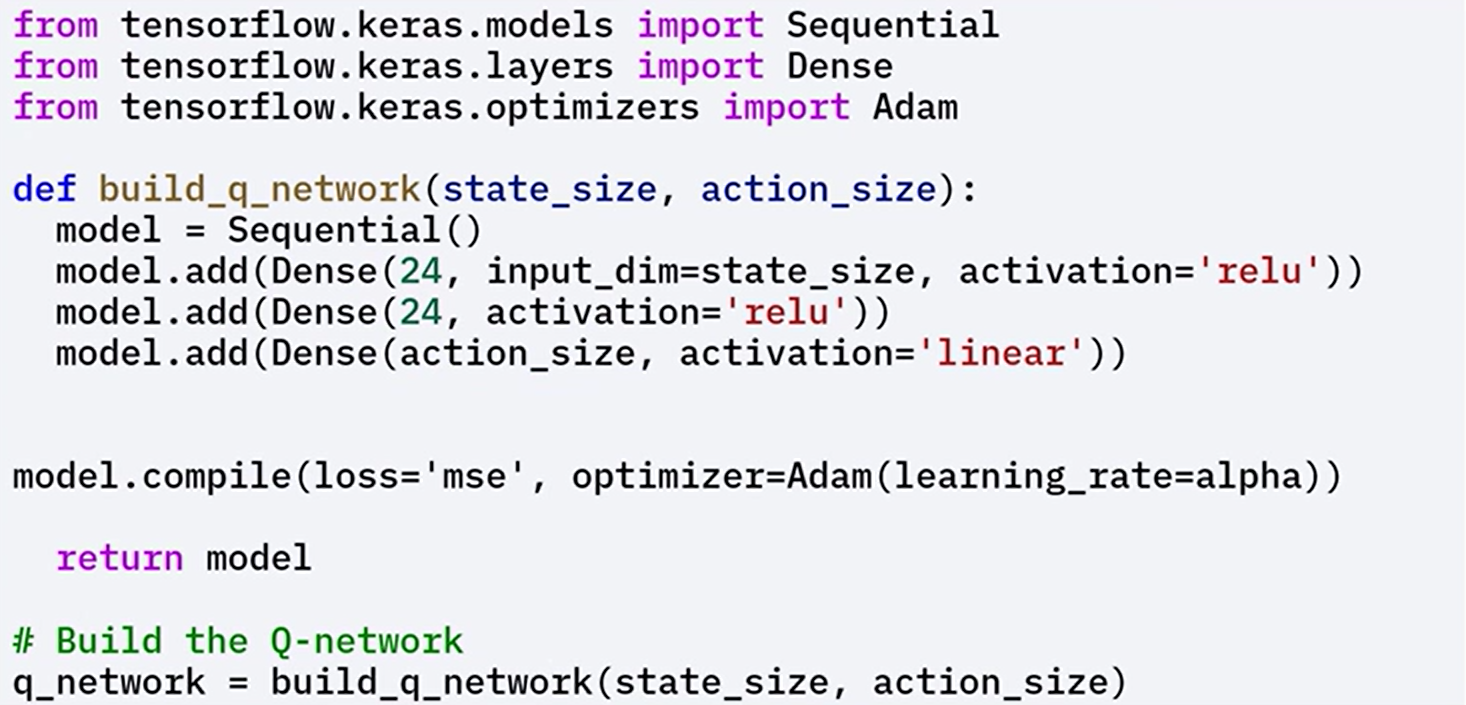
A Q-table is initialized with zeros, although it is not used directly in the neural network approach.

Since Q-Tables are impractical in environments with **large or continuous** state spaces, a **Q-network** is used to approximate the Q-value function.

🔸 **Build the Q-network:**

By using Keras, Q-Network can be build using a few dense layers:

* The **Input layer size** should match the **state size**.
* The **output layer size** should match the **action size** (number of possible actions), with **linear activation** function
* **Hidden layers** can have any architecture, but **typically two or three hidden layers**, with **ReLu activation** function.



🔸 **Train the Q-network:**

Training involves several steps:

1. **Initialize the state**

Reset the environment to get the initial state, agent interacts with that state for a given number of steps.

1. **Select an action**

Use an **epsilon-greedy policy** to balance exploration and exploitation:

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

1. **Take the action**

Execute the selected action in the environment to receive the next state and reward

1. **Update the Q-values**

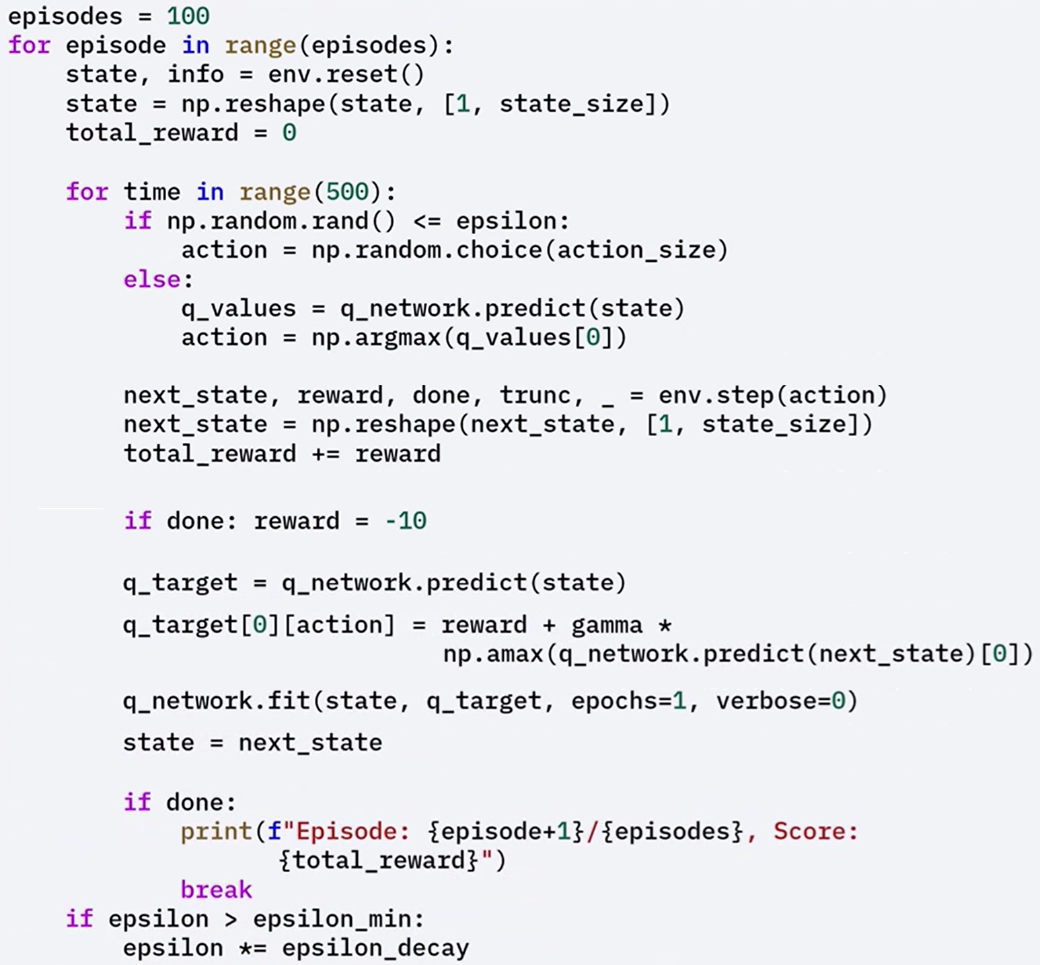
Use the **Bellman equation** to compute the target Q-value

Train the Q-network to **minimize the difference** between the predicted and target Q-values

1. **Repeat**

Continue the process until the agent reaches a **terminal state** or achieves the **goal**

Over multiple episodes, **gradually reduce** the exploration rate ε to shift from **exploration** to **exploitation**.



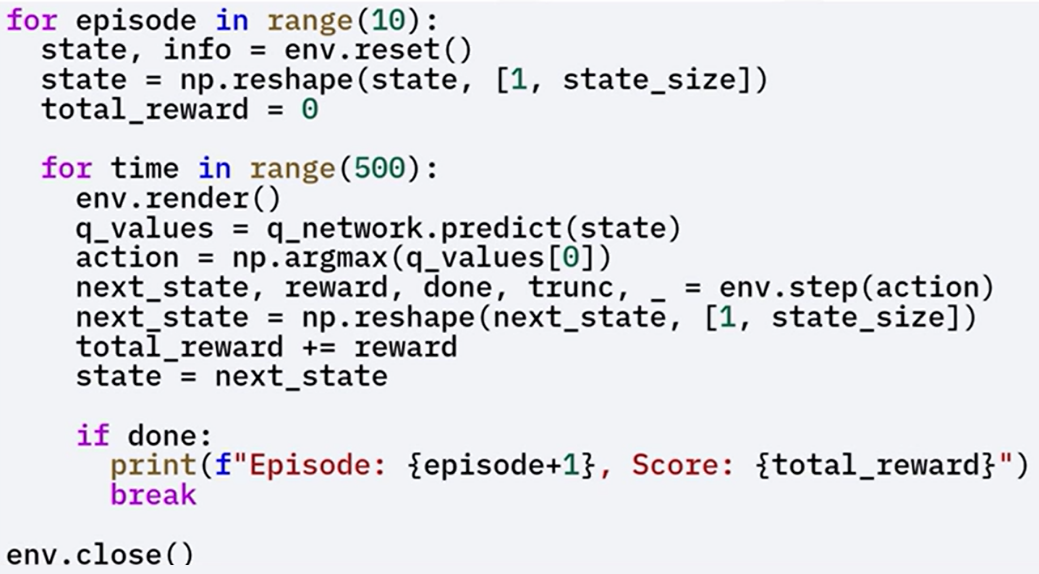
🔸 **Evaluate Trained Agent:**

After training is complete the agent is **evaluated** by letting it interact with the environment using the **learned policy**

During evaluation, the agent chooses actions based on the **maximum Q-values,** exploiting the learned Q-values to maximize rewards.

The agent's behavior is rendered, to measure performance based on the total rewards accumulated over several episodes.

The evaluation confirms whether the agent successfully learned an **effective policy** for solving the task.



### ✅ Takeaways

✅ Reinforcement learning trains agents to make **sequential decisions** to maximize **cumulative reward**.

✅ Q-learning is a **value-based**, **off-policy** algorithm that estimates the Q-value function Q(s, a).

✅ The **Bellman equation** is used to iteratively update Q-values.

✅ For large state spaces, **Q-networks** implemented with **Keras** replace Q-tables.

✅ The **epsilon-greedy policy** balances exploration and exploitation during training.

✅ After training, the agent is evaluated based on **reward accumulation** over multiple episodes.

✅ The Q-network is trained using **Keras**, **Adam optimizer**, and **mean squared error loss** to approximate Q-values.

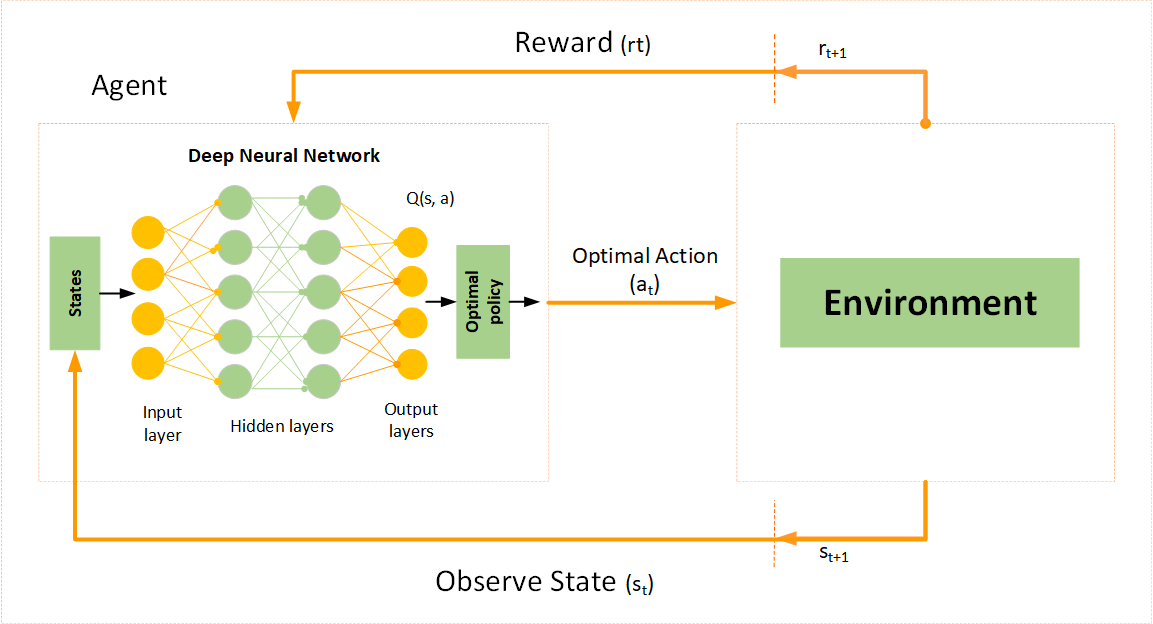
## 📌 Deep Q-Networks (DQNs) with Keras

DQNs leverage deep neural networks to approximate the Q-value function and enable reinforcement learning to scale in complex environments.

The key innovations (experience replay and target networks) help stabilize training and improve performance.

### 🔹 What are Deep Q-Networks

Deep Q-Networks (DQNs) extend the Q-learning algorithm by replacing the Q-table with a deep neural network that approximates the Q-value function.



Traditional Q-learning becomes impractical in environments with large or continuous state spaces due to the exponential growth of the Q-table. DQNs solve this problem by allowing the agent to estimate Q-values through a neural network, enabling scalability in more complex settings.

The success of DQNs lies in stabilizing training using two techniques:

* **Experience replay**
* **Target networks**.

### 🔹 Key Concepts of DQNs

🧠 **Q-Value Function Approximation**:

Instead of explicitly storing all state-action pairs, DQNs use a neural network to approximate the function ***Q(s,a)***, where “**s**” represents the state and “**a**” represents the action.

This generalizes Q-values across a broader range of states.

🧠 **Experience Replay:**

The agent stores its experiences in the form of (*state, action, reward, next\_state*) in a replay buffer.

During training, it samples random minibatches from this buffer. This technique breaks the correlation between consecutive experiences and stabilizes learning.

🧠 **Target Network:**

A secondary neural network, the target network, is introduced to generate more stable Q-value targets.

Unlike the primary Q-network, the target network is updated less frequently, reducing oscillations and divergence during training.

### 🔹 DQNs implementation steps

The process of implementing DQNs follows the structure of Q-learning but introduces new components to improve stability and scalability.

Each step is critical to correctly build and train a deep reinforcement learning agent.

🔸 **Initialize the environment and parameters:**

Define the learning environment (por example, using OpenAI's Gym platform).

Set all necessary hyperparameters for training, including learning rate, discount factor, exploration rate, and replay buffer size.

🔸 **Build the Q-Network and Target Network:**

Two neural networks are created using Keras: the **primary Q-network** and the **target network**.

Both share the same architecture, but the target network’s weights are updated less frequently to serve as a stable reference for computing Q-value target.

🔸 **Implement Experience Replay:**

A replay buffer is initialized to store agent experiences.

During training, random minibatches are sampled from the buffer to update the Q-network. This reduces the impact of correlated experiences and improves learning efficiency.

🔸 **Train the Q-Network:**

The Bellman equation is applied to iteratively update Q-values, leveraging outputs from the target network as stable references.

The primary Q-network is trained using gradients computed from the loss between predicted Q-values and target Q-values.

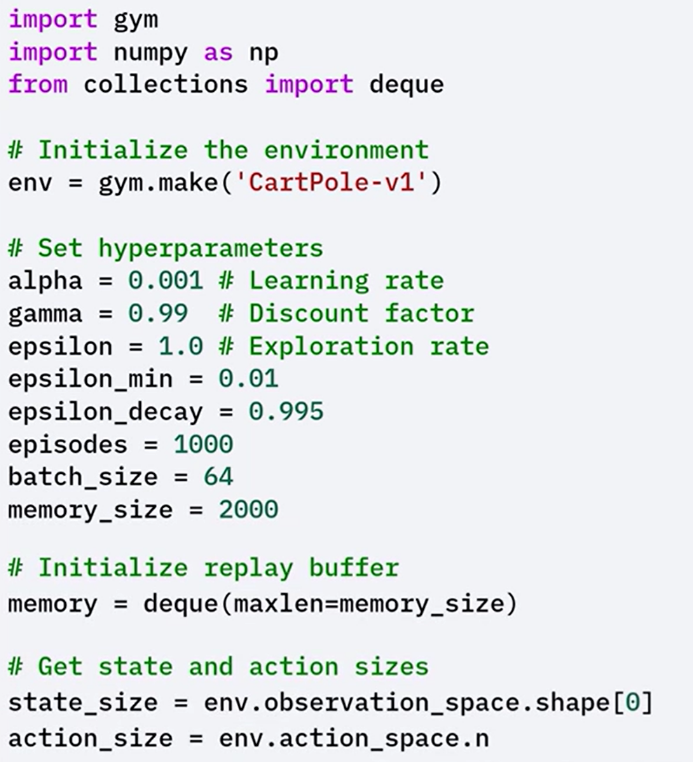
🔸 **Evaluate the Agent:**

After training, the agent interacts with the environment using the learned policy.

Its performance is measured based on cumulative rewards across multiple episode

### 🔹 DQNs Code Implementation Workflow with Keras

This section outlines the practical implementation of DQNs using the CartPole environment and Keras, including network architecture, training loops, and evaluation setup.

🔸 **Environment and Hyperparameters Initialization**:

The **CartPole** environment is initialized using gym.make.

Key hyperparameters such as learning rate, discount factor, exploration rate (ε), and batch size for experience replay are defined.

🔸 **Replay Buffer Initialization:**

A double-ended queue with a fixed size is created to store the agent’s experiences (state, action, reward, next\_state, done) over time.

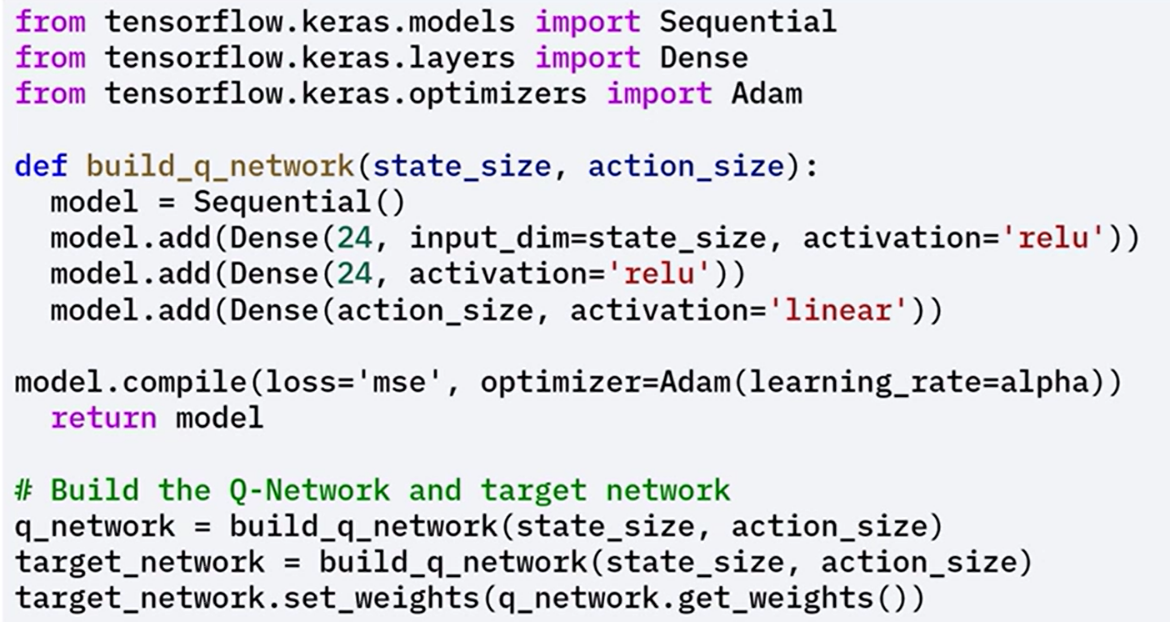
Deque, is a data structure that allows efficient insertion and deletion of elements from both its front, head, and back, tail.

🔸 **Q-Network and Target Network Creation:**

Two identical networks are created, **primary Q-Network** and **target network.** These networks are used to predict Q-values and compute targets for training.

Both networks have the same architecture, with two hidden layers and ReLU activation.

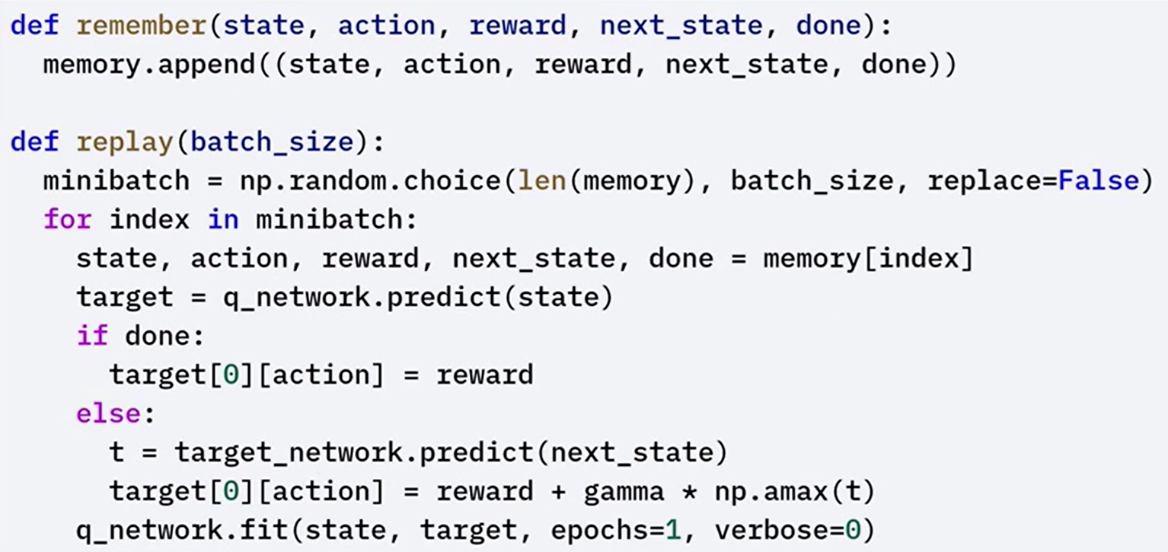
The target network is periodically updated to match the weights of the primary Q-network. This delay helps prevent instability during learning.



🔸 **Experience Storage and Sampling:**

A **remember()** function stores (state, action, reward, next\_state, done) **tuples** in the replay buffer.

The **replay()** function samples minibatches randomly to break experience correlation between consecutives experiences and improve convergence.



🔸 **Training Loop and Policy:**

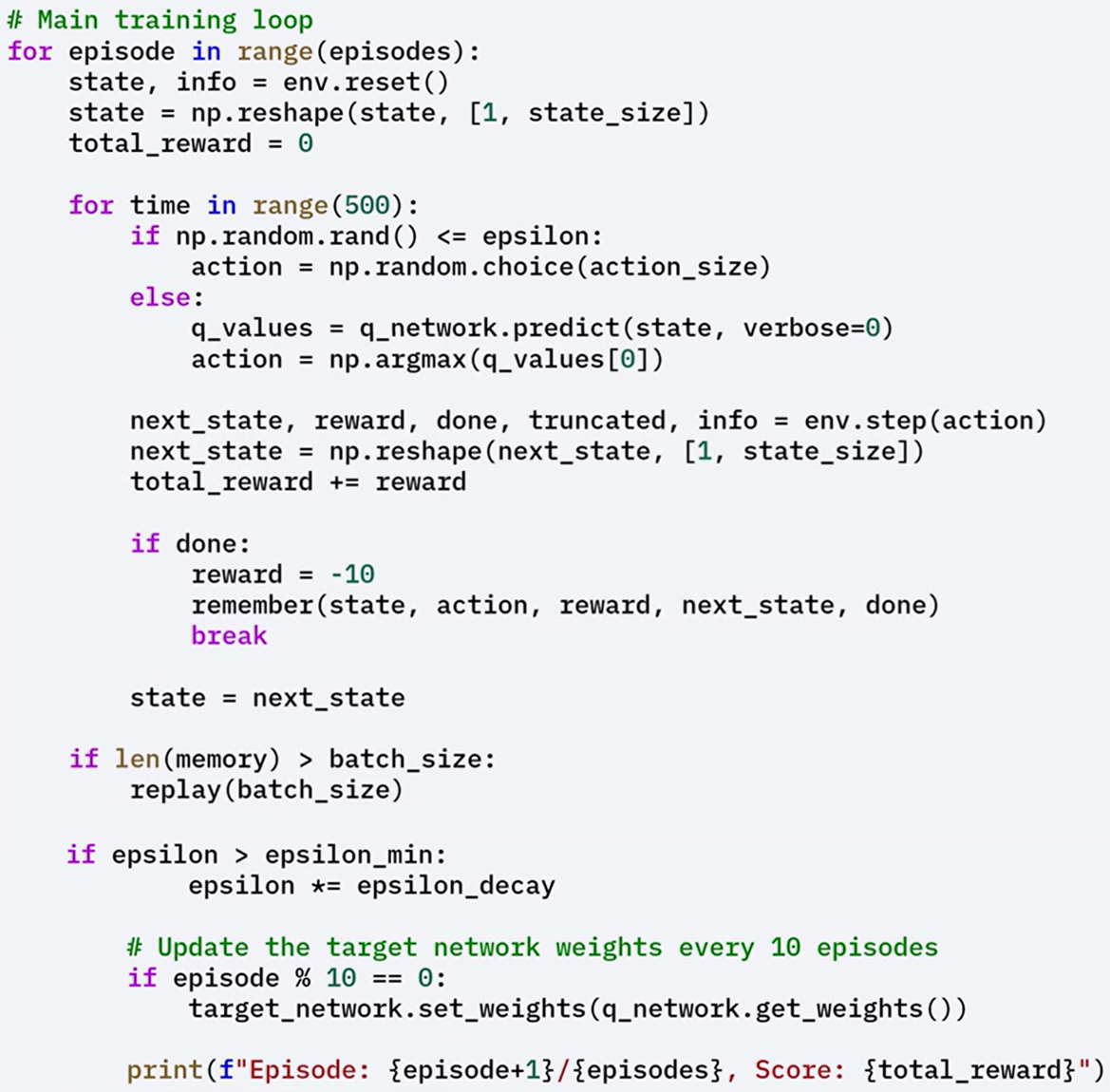
The training loop runs for a defined number of episodes.

At each step, the agent selects actions using an ε-greedy policy (random action with probability ε, or greedy action otherwise).

After each episode, the ε value decays gradually to promote exploitation over exploration.

For each sampled experience, the Bellman equation is applied to compute the target Q-value. The Q-network is trained to minimize the difference between predicted and target values.

 Target networks weights are periodically updated to match the primary networks weights.

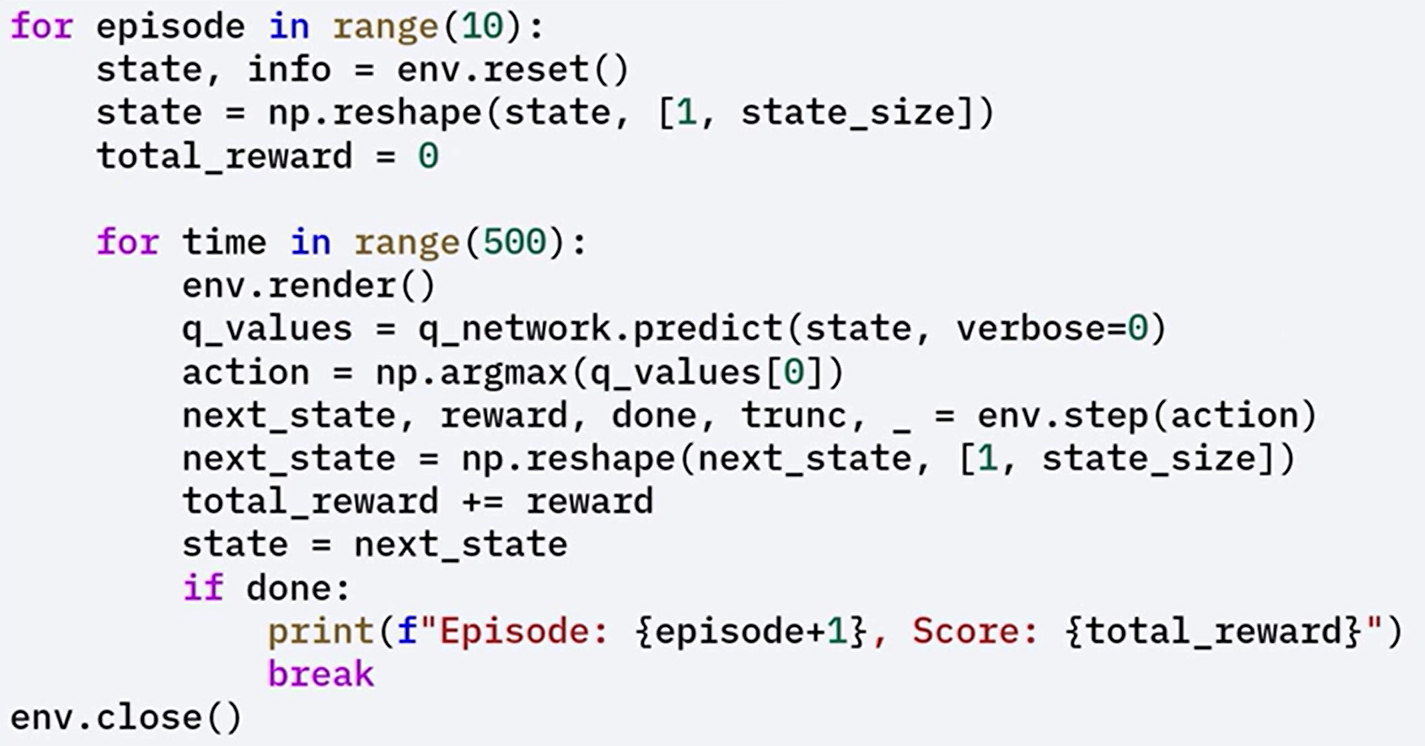


🔸 **Agent Evaluation:**

After training, the agent is evaluated by running episodes with the learned policy.

The environment is rendered for visual inspection, and total rewards are printed for each episode.

During this phase, the agent focuses on exploiting the learned Q-values.



### ✅ Takeaways

✅ Deep Q-Networks (DQNs) extend Q-learning by using neural networks to approximate Q-values, enabling scalability to large or continuous state spaces.

✅ DQNs use experience replay and target networks to stabilize training and improve learning performance.

✅ The DQN implementation includes initializing the environment, defining networks, storing and sampling experience, training with the Bellman equation, synchronizing target networks, and evaluating the agent with the learned policy.