Reinforcement Learning

Advanced Reinforcement Learning : Deep Q-Learning and its Variants

Learning Objectives

By the end of this lesson, you will be able to:

- Discuss what are Q values and Q value iteration
- Explore the Q-learning algorithm and its components
- Describe exploration policies in Q-learning algorithms
- Discuss the deep Q-learning algorithm for RL problems
- Describe the various variants of DQL

Q-Value Algorithm

What Are Q Values?

The value iteration algorithm calculates the optimal value of a state.

- Knowing the optimal state values is useful to evaluate a policy but it doesn’t give us the optimal policy for an agent.
- It doesn’t provide an insight about the value of different possible action from a state.

Bellman found a very similar algorithm to estimate the optimal state-action values, generally called Q-Values (Quality Values).

What Are Q Values?

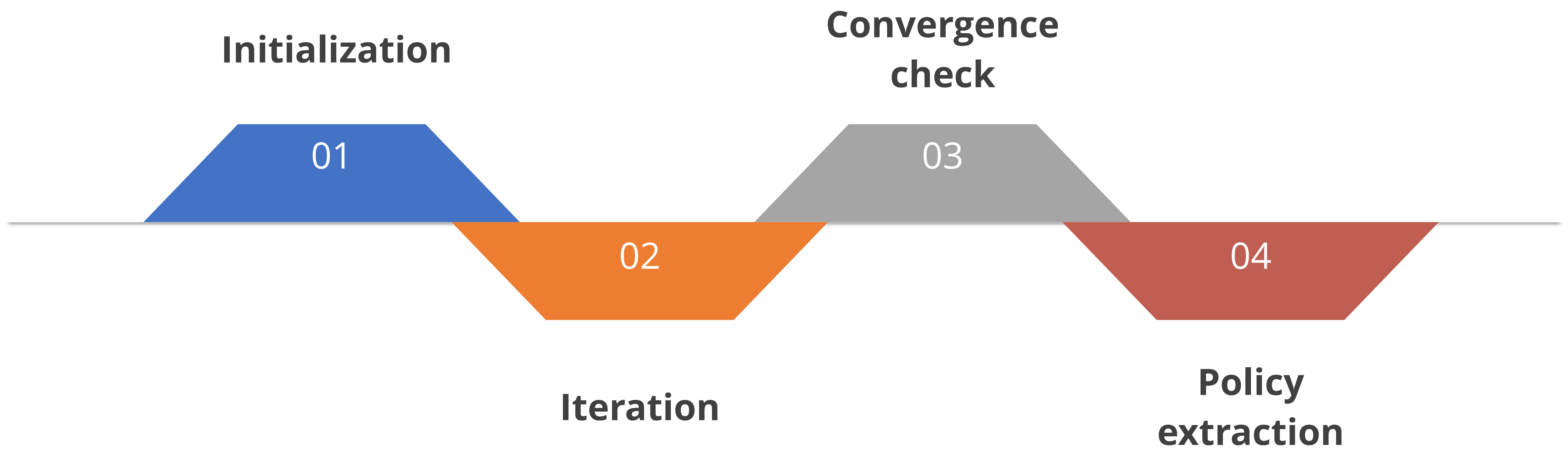
The optimal Q-Value for a given state-action pair (s, a), denoted as Q*(s, a).

- It represents the sum of discounted future rewards an agent can expect on average.
- It is calculated after the agent reaches state s and chooses action a, but before observing the action's outcome.
- The assumption is that the agent acts optimally after taking the specified action.

Q*(s, a) helps determine the optimal action to take in a given state, guiding the agent's decision-making process.

Q-Value Iteration Algorithm

The steps for the Q-value iteration algorithm are outlined here:



Q-Value Iteration Algorithm

The steps for the Q-value iteration algorithm are outlined here:

Initialization

Start by initializing all the Q-values estimates to ***zero*** for all state-action pairs.

Iteration

Update the Q values suing the Bellman equation until they converge to the optimal values.

Q-Value Iteration Algorithm: Update Function

The update function for Q-value iteration is described as follows:

$$for\ all\ s\ and\ a$$

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')\,[R(s,a,s') + \gamma . max_{a'} Q_k(s',a')]$$

Here,

T(s, a, s') is the transition probability from state **s** to state **s'**, given that the agent choses action **a**.

Q-Value Iteration Algorithm: Update Function

The update function for Q-value iteration is described as follows:

$$for\ all\ s\ and\ a$$

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')\,[R(s,a,s') + \gamma . max_{a'} Q_k(s',a')]$$

Here,

R(s, a, s’) is the reward that the agent receives when it goes from state **s** to state **s'**, given that the agent chose action **a.**

Q-Value Iteration Algorithm: Update Function

The update function for Q-value iteration is described as follows:

$for\ all\ s\ and\ a$

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')\,[R(s,a,s') +\ \gamma.max_{a'}Q_k(s',a')]$$

Here,

γ Is the discount factor.

Q-Value Iteration Algorithm: Convergence Check

The algorithm iteratively refines the Q-values until they converge to the optimal values.

Convergence

- Check for convergence by comparing new Q-values with previous Q-values.
- The convergence is typically determined by a small threshold or after a fixed number of iterations.

Q-Value Iteration Algorithm: Convergence Check

After defining the optimal Q-values for state and action pair, optimal policy as below:

Policy extraction

- Once the Q-values have converged, the optimal policy $\boldsymbol{\pi}$ can be extracted as:

$$\pi^*(s) = argmax_a \, Q^*(s, a)$$

- When the agent is in state ***s***, it should choose the action with the highest Q-value for the state.

Q Learning Algorithm

What Is Q Learning?

Q-learning is a model-free reinforcement learning algorithm to learn the value of an action in a particular state.

- Q-learning algorithm is an adaption of the Q-value iteration algorithm.
- It is modified for scenarios where initial transition probabilities and rewards are unknown to the agent.
- Addresses cases where the agent lacks prior knowledge about transition probabilities and immediate rewards.

It learns directly from interactions with the environment without requiring a complete model.

What Is Q Learning?

Q-learning works by watching an agent play and gradually improving its estimates of Q-value.

- Once it has close-enough estimates, then optimal policy is choosing the action with highest Q-value.
- Q-learning seeks to learn a policy, which tells an agent what action to take under what circumstances.
- It does this by learning a Q-function q(s, a), which represents the quality of a particular action a taken in a particular state ***s***.

Key Components of Q-Learning

The key components of Q learning algorithm include:

Q-Function

The core of Q-Learning is the Q-Function, Q(s, a), which provides the expected return of taking action ***a*** when in state ***s***.

Key Components of Q-Learning

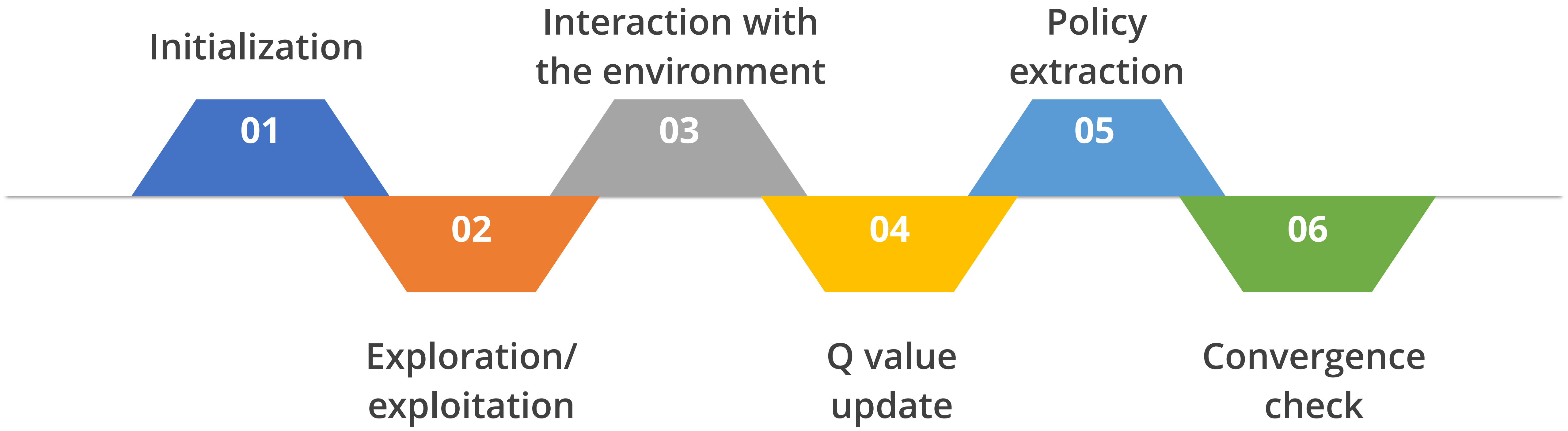
The key components of Q learning algorithm include:

Learning Process

- The agent explores the environment, and at each step, it updates the Q-values based on the observations.
- This process is repeated until the Q-values converge, indicating that the agent has learned the optimal policy.

Q-Learning Algorithm

The basic outline of the Q-learning algorithm is as below:



The steps 2-4 are repeated until convergence or for a predefined number of iterations.

Q-Learning Algorithm: Update Function

Update the Q-value for the current state-action pair using the bellman equation.

The update function is as below:

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

Here,

$Q(s',a')$: The current estimate of the Q-value for state ***s*** and action ***a***.

Q-Learning Algorithm: Update Function

Update the Q-value for the current state-action pair using the bellman equation.

The update function is as below:

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

Here,

α is the learning rate which determines to what extent newly acquired information overrides old information.

Q-Learning Algorithm: Update Function

Update the Q-value for the current state-action pair using the Bellman equation.

The update function is as below:

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

Here,

$\boldsymbol{R(s,a)}$: The reward received after taking action $\boldsymbol{a}$ in state $\boldsymbol{s}$.

Q-Learning Algorithm: Update Function

Update the Q-value for the current state-action pair using the Bellman equation.

The update function is as below:

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

Here,

γ is the discount factor

Q-Learning Algorithm: Update Function

Update the Q-value for the current state-action pair using the Bellman equation.

The update function is as below:

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

Here,

$max_{a'} Q(s', a')$: The estimated optimal future value after taking action ***a'*** in the new state ***s'***.

Q-Learning Algorithm: Update Function

It's observed that Q-Learning algorithm may take about 400 times more iterations than Q-value algorithm, to converge.

Can you guess why?

Take a pause and think.

Q-Learning Algorithm: Key Observation

Q-learning algorithm will converge to the optimal Q-values, but it will take many iterations, and possibly quite a lot of hyperparameter tuning.

- Q-value algorithm have model dynamics available that is transition probabilities and rewards.
- But the agent in Q-learning algorithm do not know transition probabilities and rewards information.

Obviously, not knowing the transition probabilities or the rewards makes finding the optimal policy significantly harder!

Q-Learning Algorithm: Key Observation

- In Q-Learning, the exploration policy is random but the policy being learned will always choose the actions with the highest Q-values.
- Essentially, exploration and training policies are different and hence Q-Learning is called **off-policy** algorithm.

Q-Learning Algorithm: Exploration Policies

Exploration in Q-learning poses several challenges due to the inherent trade-off between exploring new actions and exploiting known actions. These challenges include:

Insufficient Exploration

- With a random strategy for exploration, Q-Learning might converge to a suboptimal policy.
- If the agent only exploits current knowledge, it may repeatedly choose what appears to be the best action without exploring other potentially better options.

Q-Learning Algorithm: Exploration Policies

Exploration in Q-learning poses several challenges due to the inherent trade-off between exploring new actions and exploiting known actions. These challenges include:

Convergence to Local Optima

- In stochastic environments lack of exploration or sparse rewards increases the risk of the agent converging to a local optimum.
- This could occur because the agent might fail to identify alternative actions or strategies that result in greater cumulative rewards.

Q-Learning Algorithm: Exploration Policies

Exploration in Q-learning poses several challenges due to the inherent trade-off between exploring new actions and exploiting known actions. These challenges include:

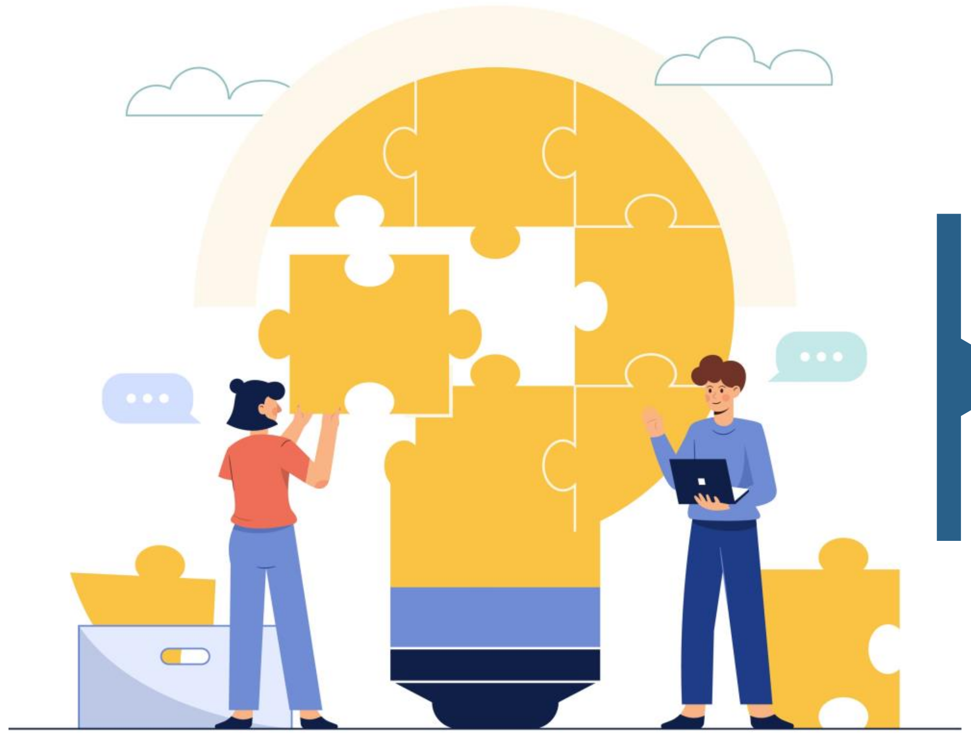
Dynamic environments

- In changing environments, an agent that doesn't continue to explore may fail to adapt to new conditions.
- It may also fail to learn about better actions as the environment evolves.

How to Address These Challenges?

The challenges presented in exploration in Q-learning can be mitigated by:

To address the exploration challenge in Q-Learning, various strategies have been devised.



The epsilon-greedy algorithm being one of the most popular due to its simplicity and effectiveness.

Epsilon-Greedy Exploration Policies

Epsilon-greedy exploration policies are a category of strategies used in reinforcement learning, including Q-learning, to balance exploration and exploitation.

In epsilon-greedy policies, the agent makes decisions by considering both:

Exploitation

The current best-known actions.

Exploration

Exploring new actions with a certain probability

Epsilon-Greedy Exploration Policies

In epsilon-greedy policies, the agent makes decisions by considering both exploitation and exploration policies.

Exploitation strategy

It often selects the optimal action with the highest estimated value.

Exploration strategy

It occasionally chooses a random action with a small probability.

Epsilon-Greedy Exploration Strategy

The epsilon greedy strategy is parameterized by ϵ, a value between 0 and 1. ϵ determines the probability of exploration versus exploitation during action selection.

With probability $1-\epsilon$, the agent **exploits** by choosing the action with the highest estimated value (greedy action).

With probability ϵ, the agent **explores** by randomly selecting an action from the set of available actions.

Epsilon-Greedy Exploration Strategy

ϵ is a small value, such as 0.1.

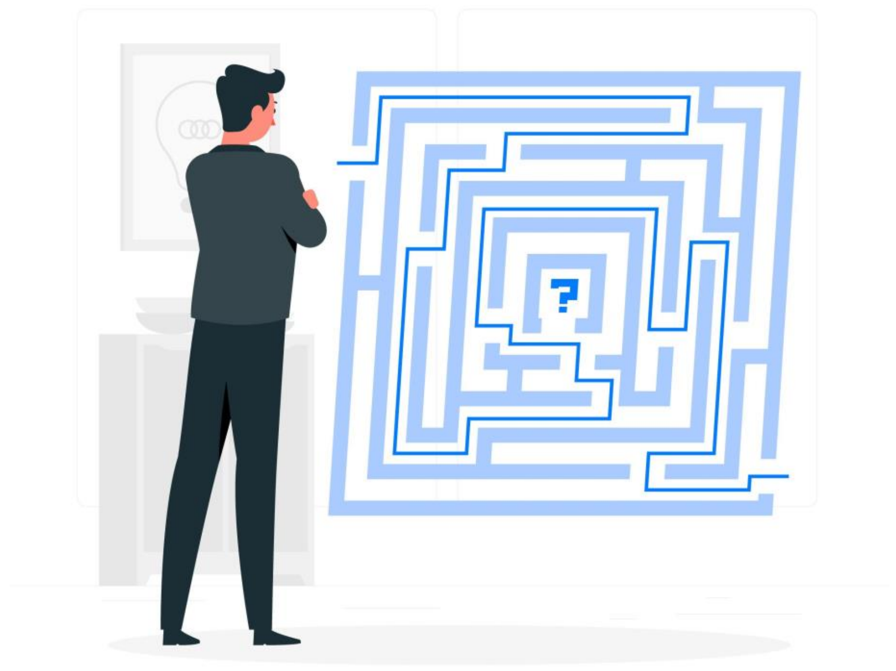
- This indicates that 10% of the time the agent explores random actions.
- Approximately 90% of the time, it leverages its existing knowledge for exploitation.
- The ϵ value of can be constant, decay over time, or adapted based on other strategies, to balance exploration and exploitation efficiently.

It is quite common to start with a high value for ϵ (e.g., 1.0) and then gradually reduce it (e.g., down to 0.05)

Epsilon Greedy Strategy: Example

In a maze navigation task, suppose the agent has found a path to the goal but hasn't explored the entire maze.

With an epsilon-greedy strategy

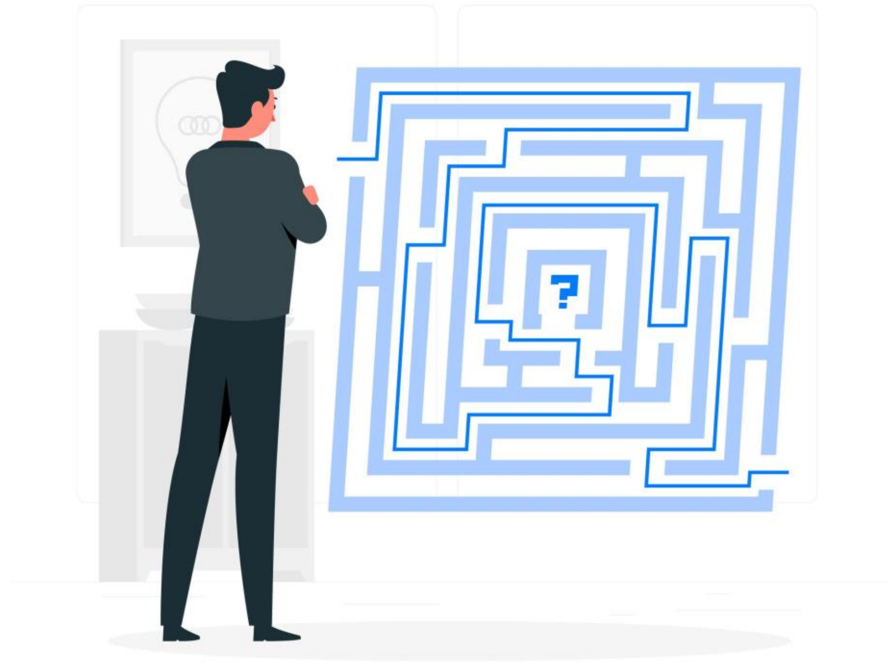


- Most of the time, it will follow the known path, which is the **exploitation** strategy.

Epsilon Greedy Strategy: Example

In a maze navigation task, suppose the agent has found a path to the goal but hasn’t explored the entire maze.

With an epsilon-greedy strategy



- But occasionally, it will try a random move, implementing the **exploration** strategy.
- This may lead to discovering a shorter path or other rewards.

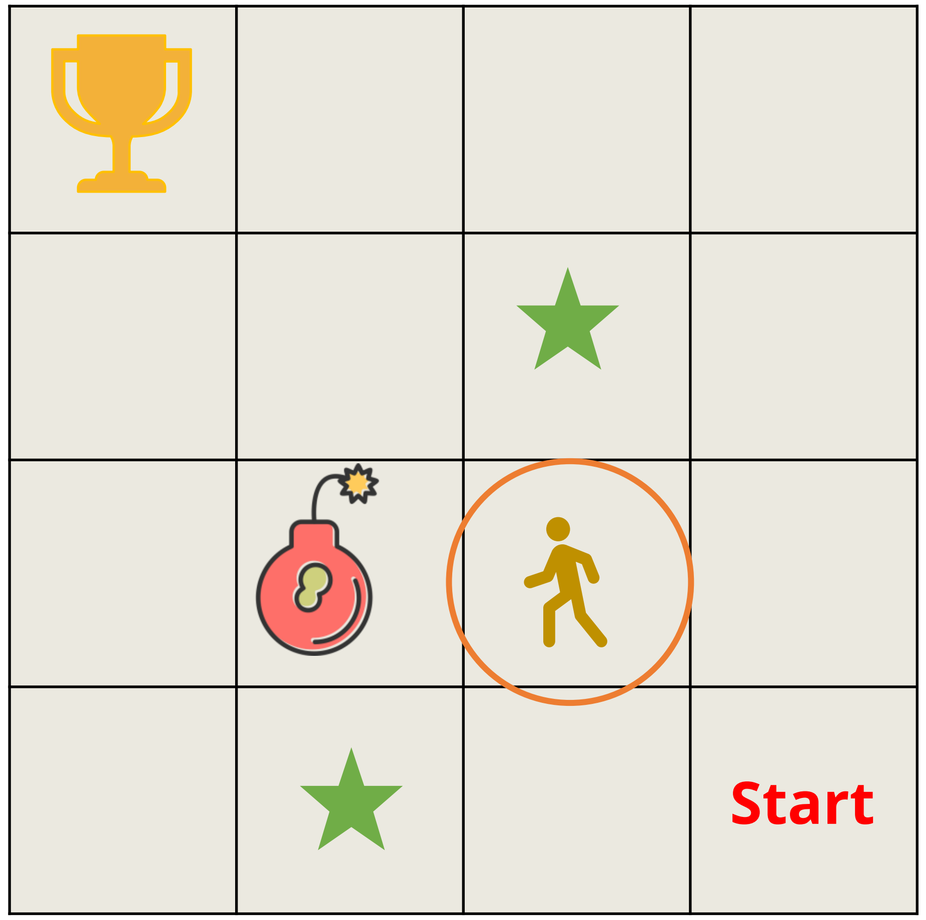
Q Learning Algorithm: Example Solution

Let's outline the solution for the example.

Setting up the grid world

State

Each cell in the grid represents a state.



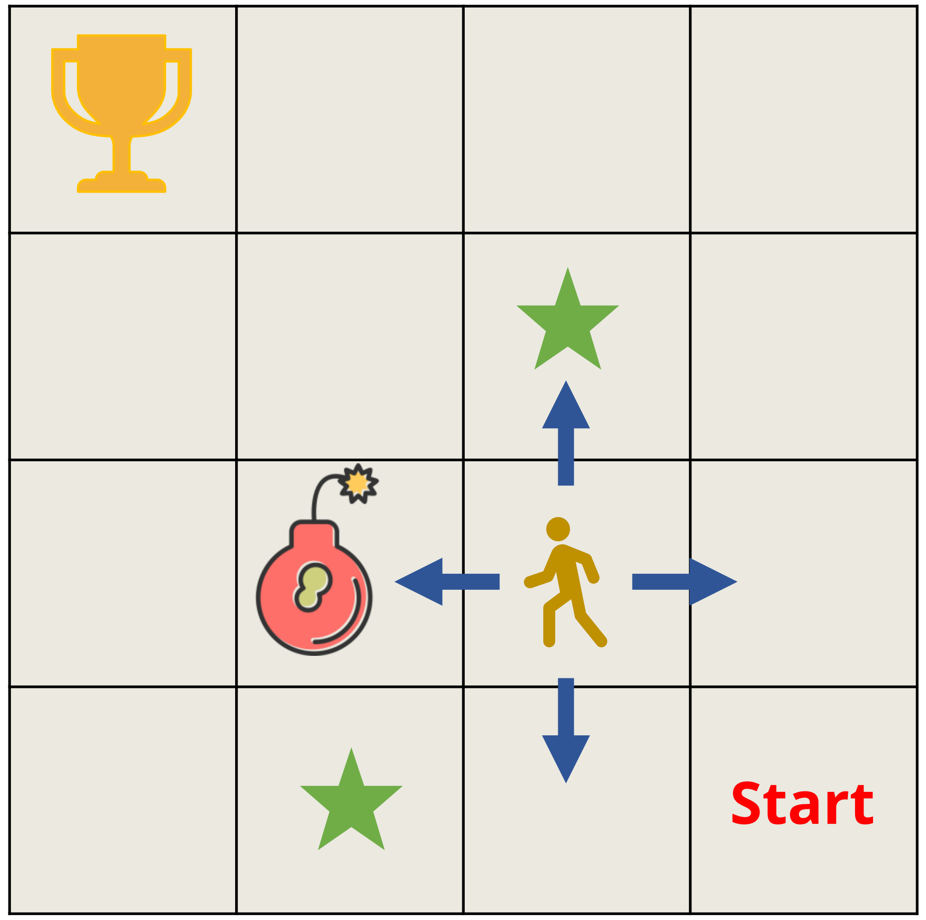
Q Learning Algorithm: Example Solution

Let’s outline the solution for the example.

Setting up the grid world

Actions

At each state (cell), the agent has several actions available, typically including moving up, down, left, or right.



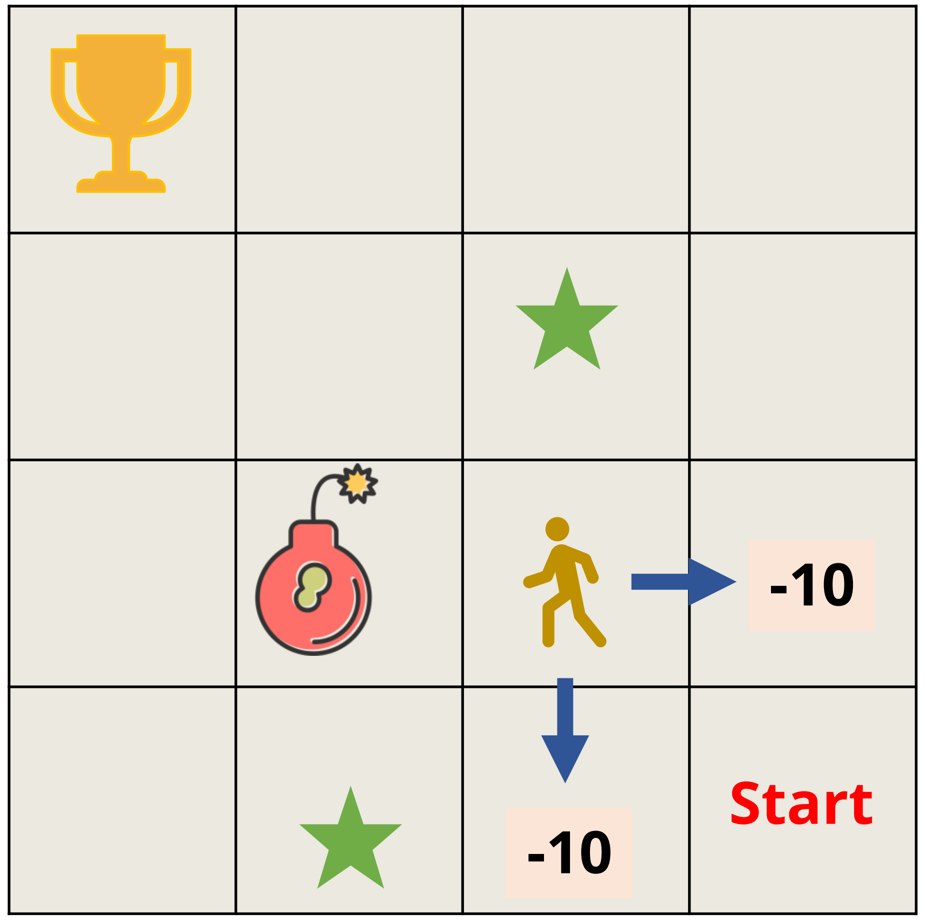
Q Learning Algorithm: Example Solution

Let’s outline the solution for the example.

Setting up the grid world

Rewards

Moving to a regular cell might incur a small negative reward; encouraging the shortest path.



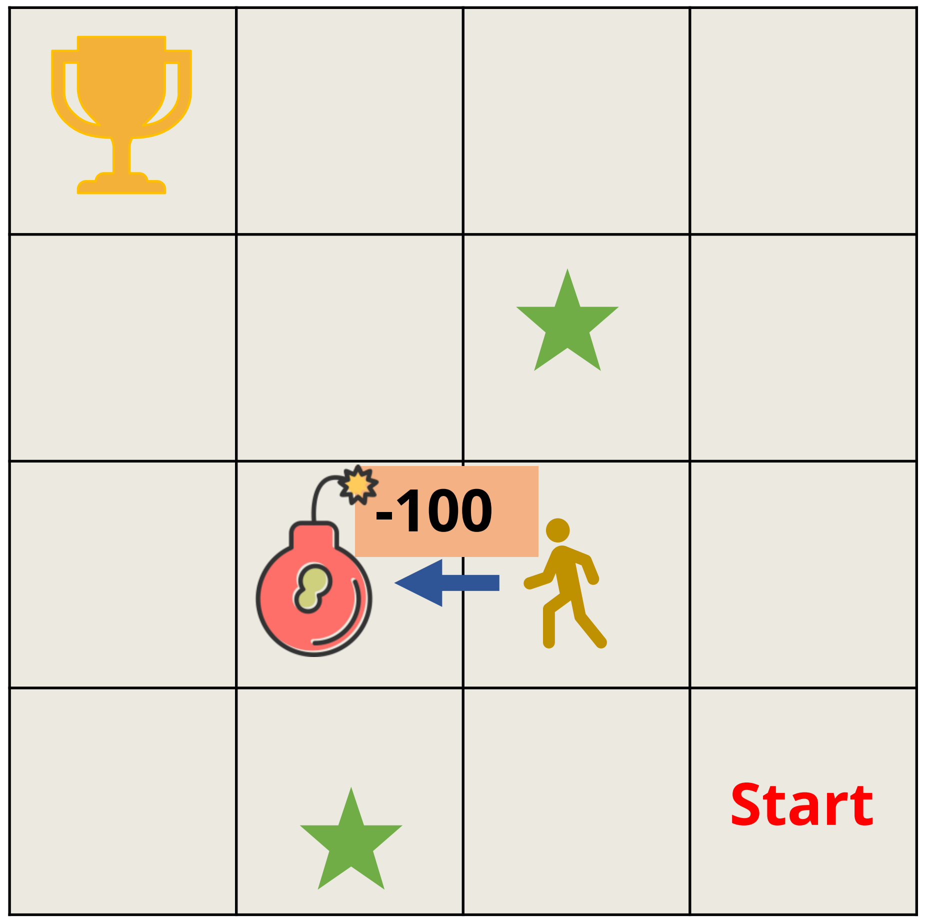
Q Learning Algorithm: Example Solution

Let's outline the solution for the example.

Setting up the grid world

Rewards

Moving to a cell with a penalty gives a larger negative reward.



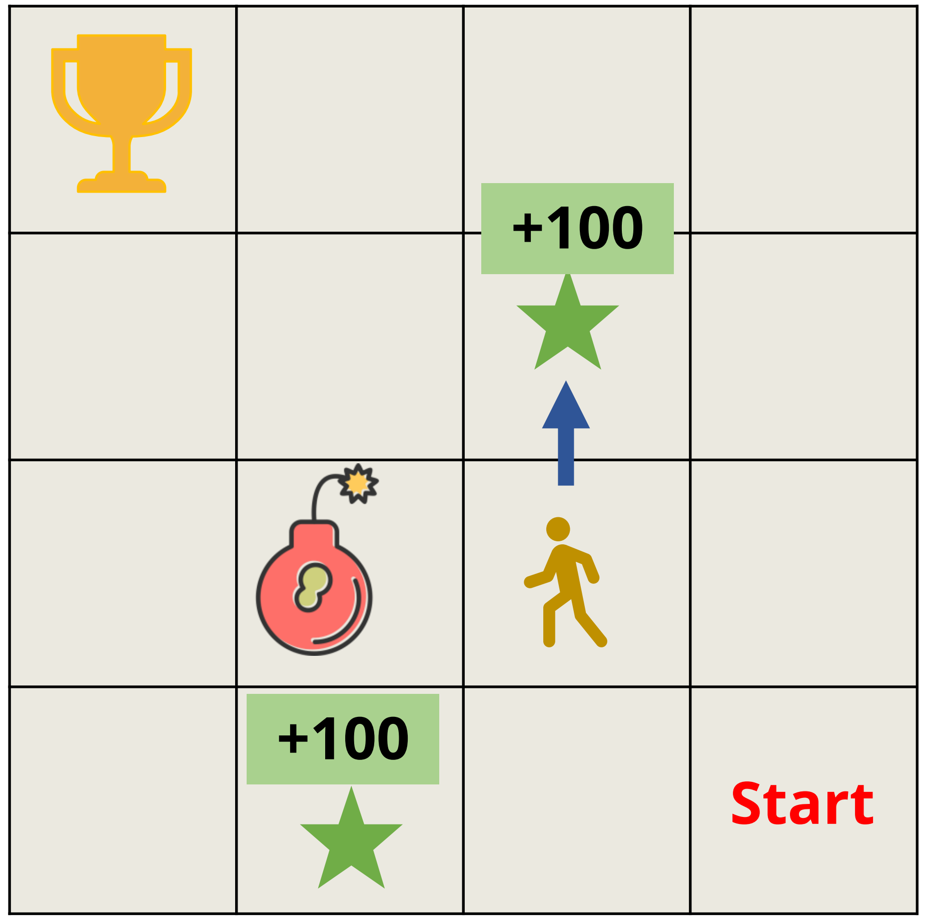
Q Learning Algorithm: Example Solution

Let’s outline the solution for the example.

Setting up the grid world

Rewards

Moving to a cell with a positive reward gives a positive reward.



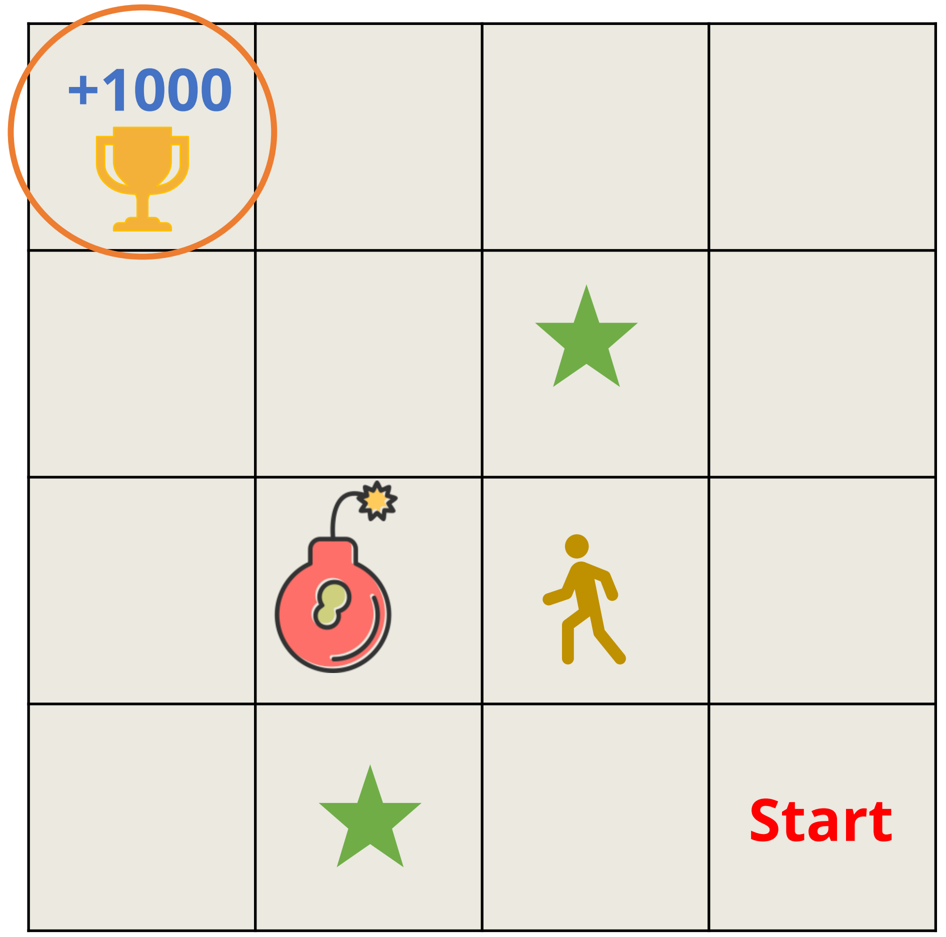
Q Learning Algorithm: Example Solution

Let’s outline the solution for the example.

Setting up the grid world

Rewards

Reaching the goal gives a large positive reward.



Q Learning Algorithm: Example Solution

Implementing Q-learning in the grid-world

	Action			
State	↓	→	↑	←
Start	**0**	**0**	**0**	**0**
Regular cell	**0**	**0**	**0**	**0**
Penalty cell	**0**	**0**	**0**	**0**
Reward cell	**0**	**0**	**0**	**0**
Goal cell	**0**	**0**	**0**	**0**

Initialization

- Initialize the Q-values, Q(s, a), for all possible state-action pairs.
- These might start at zero or some small initial value.

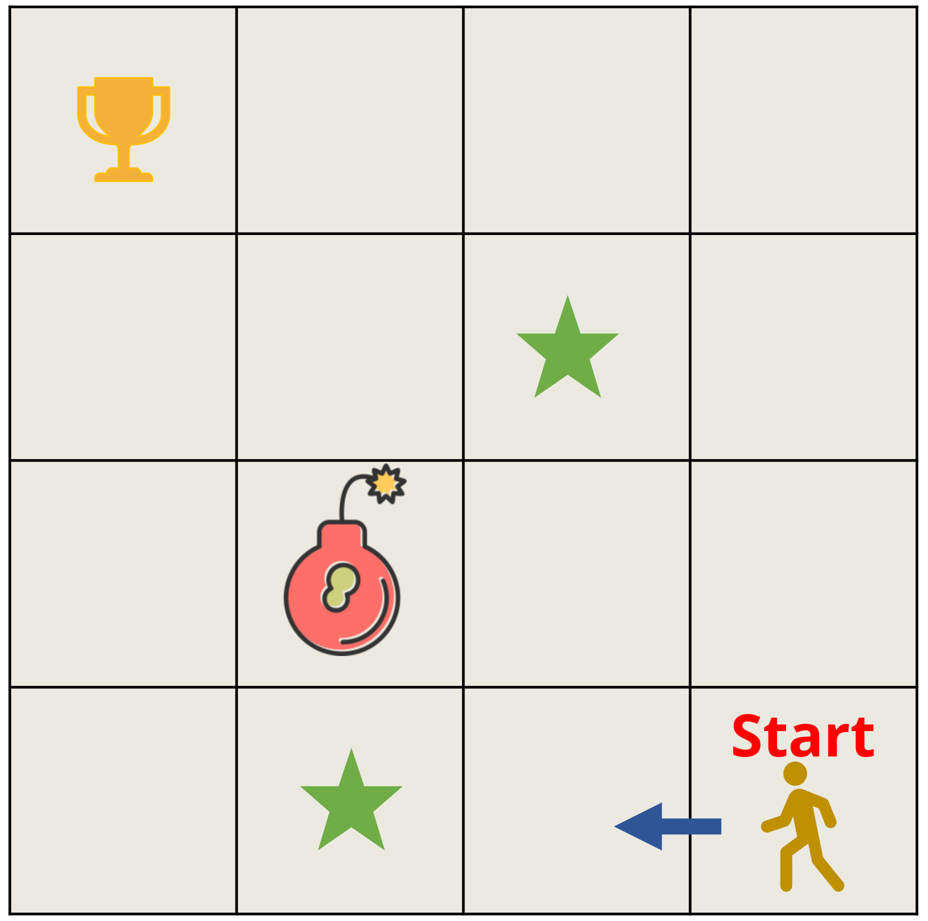
Q Learning Algorithm: Example Solution

Implementing Q-learning in the grid-world

Q-Value Update

After each move:

- Observe the immediate reward.
- This could be a penalty, reward, or neutral based on the specific cell the agent moves into.



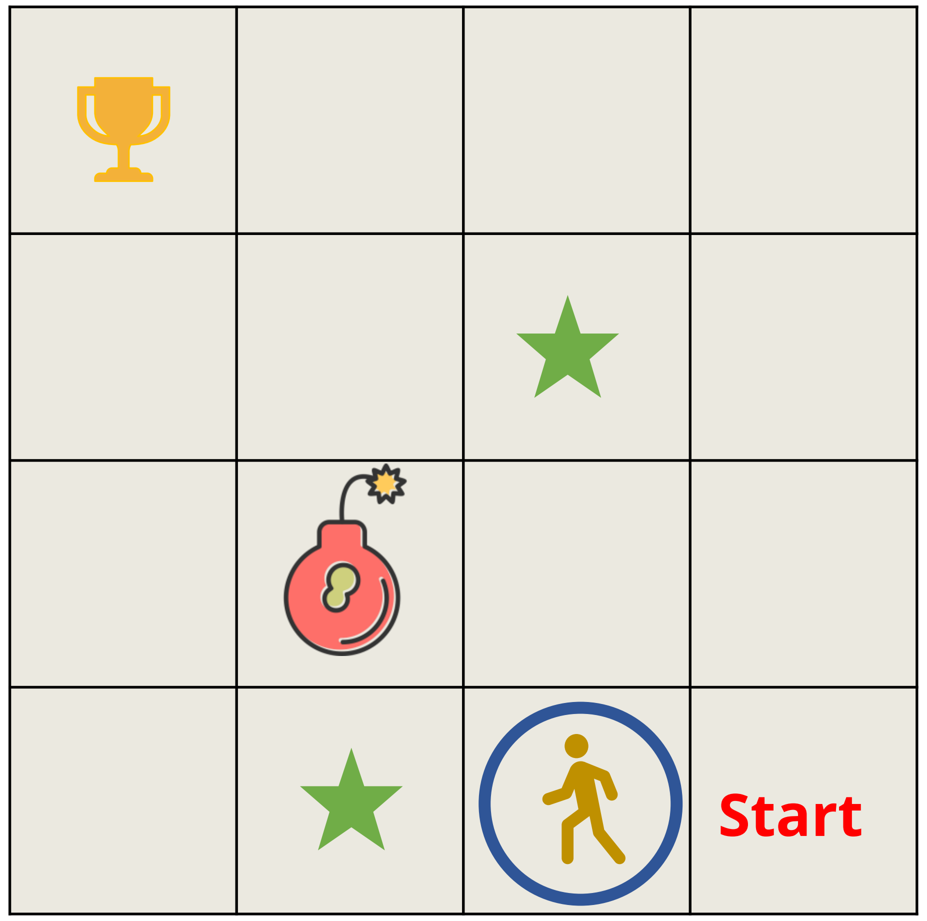
Q Learning Algorithm: Example Solution

Implementing Q-learning in the grid-world

Q-Value Update

After each move:

- Observe the new state (the new cell).



Q Learning Algorithm: Example Solution

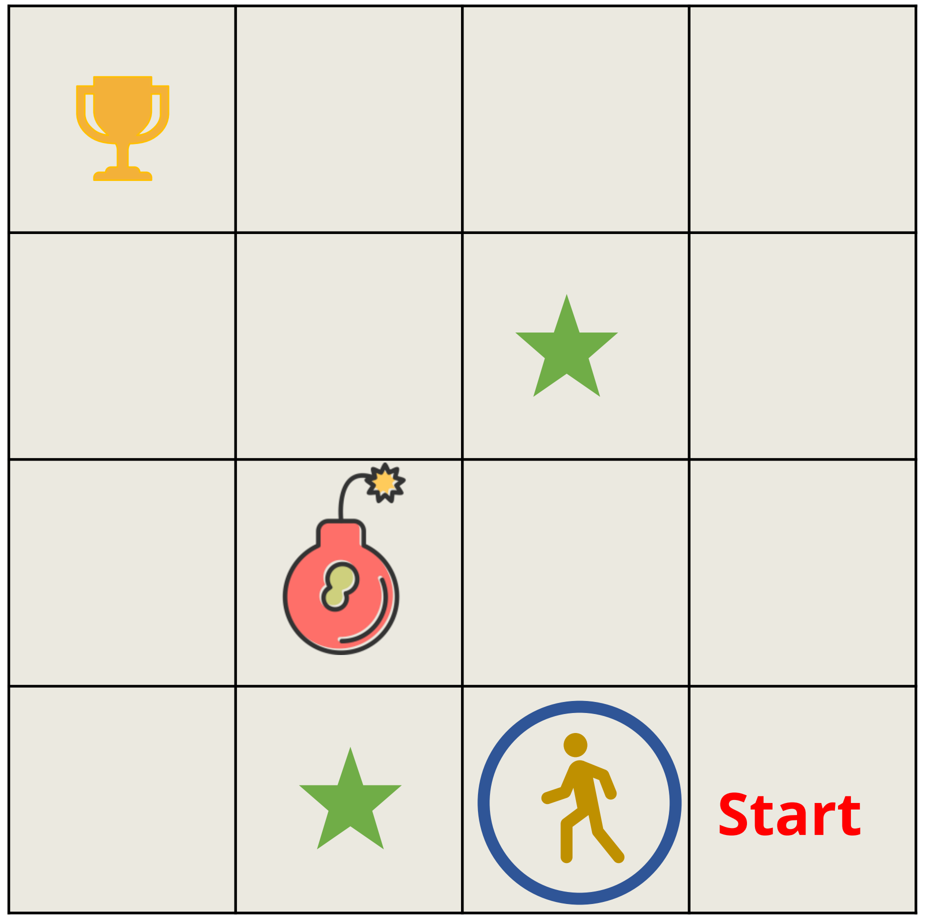
Implementing Q-learning in the grid-world

Q-Value Update

After each move:

- Update the Q-Value for the state-action pair based on the observed reward and the maximum future value from the new state.

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



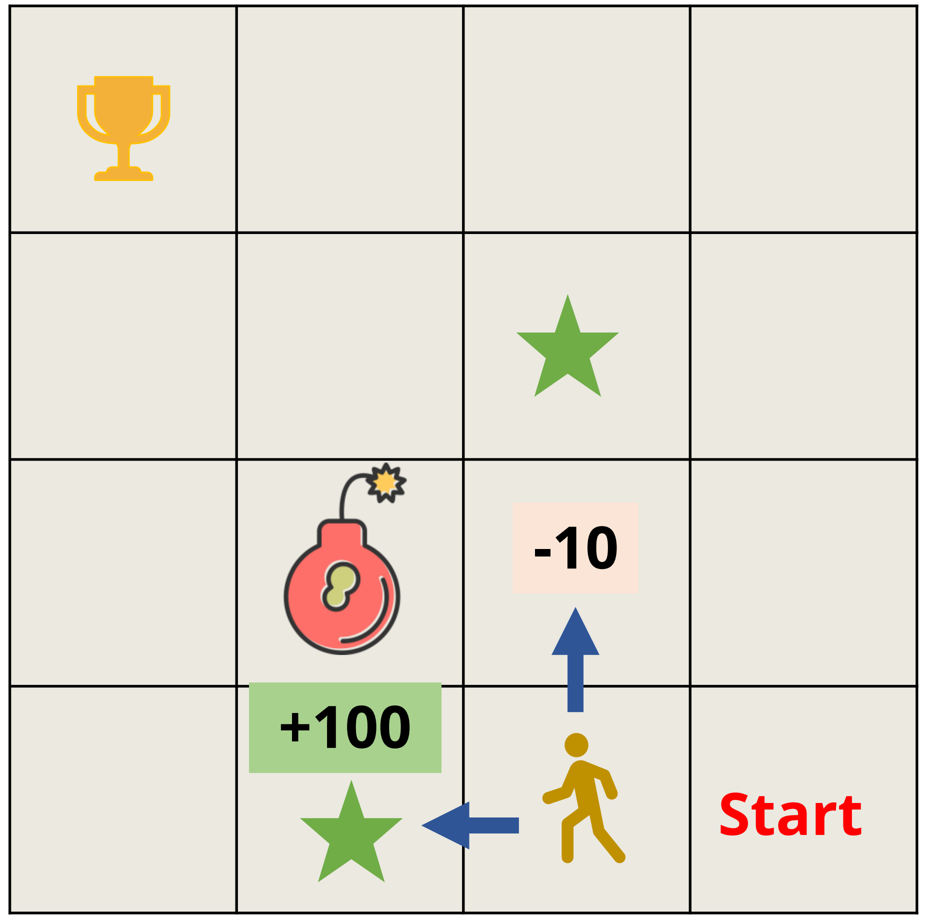
Q Learning Algorithm: Example Solution

Implementing Q-learning in the grid-world

Q-Value Update

After each move:

- The update occurs following each move made by the agent.
- The agent's strategy is modified considering immediate rewards and penalties.
- Adjustments also account for expected future rewards in the update process.



Q Learning Algorithm: Example Solution

Implementing Q-learning in the grid-world

The agent receives different rewards depending on the cell it moves to.

Continued Exploration and Learning

- The agent persists in grid exploration, obtaining immediate rewards or penalties.
- Q-values are continually updated throughout this exploration process.

Q Learning Algorithm: Example Solution

Implementing Q-learning in the grid-world

The agent receives different rewards depending on the cell it moves to.

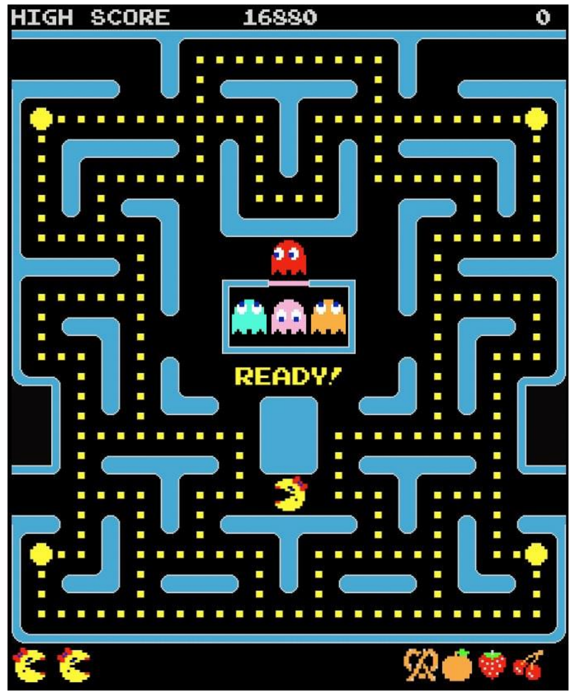
Continued Exploration and Learning

- The exploration continues until the agent either reaches the goal or the episode concludes.
- Over numerous episodes, Q-values converge, indicating a strategy optimization for reaching the goal, maximizing rewards, and minimizing penalties.

Scalability Challenges

Traditional Q-Learning encounters scalability issues in large Markov decision processes (MDPs).

The challenge arises from managing Q-values for every conceivable state-action pair.



This becomes impractical, especially in expansive state spaces.

The game of Ms. Pac-Man serves as an example, with **over 10^{45}** possible states.

Approximate Q-Learning for Scalability

Approximate Q-Learning aims to estimate Q-values using a function Q (s, a).

1

This function is defined by a manageable number of parameters (θ).

2

The approach is particularly suitable for addressing complex or large-scale problems.

3

The use of a parameterized function enhances feasibility in such scenarios.

Approximate Q-Learning for Scalability

For transition from handcrafted features to DNNs:

Initial method — Utilizing linear combinations of handcrafted features.

Example features — Distance to the closest points and their directions (e.g., Pac-Man scenario).

Objective — Approximating Q-values through this approach.

Approximate Q-Learning for Scalability

For transition from handcrafted features to deep neural networks (DNNs):

DeepMind's breakthrough in 2013 demonstrated that DNNs could significantly outperform these methods.

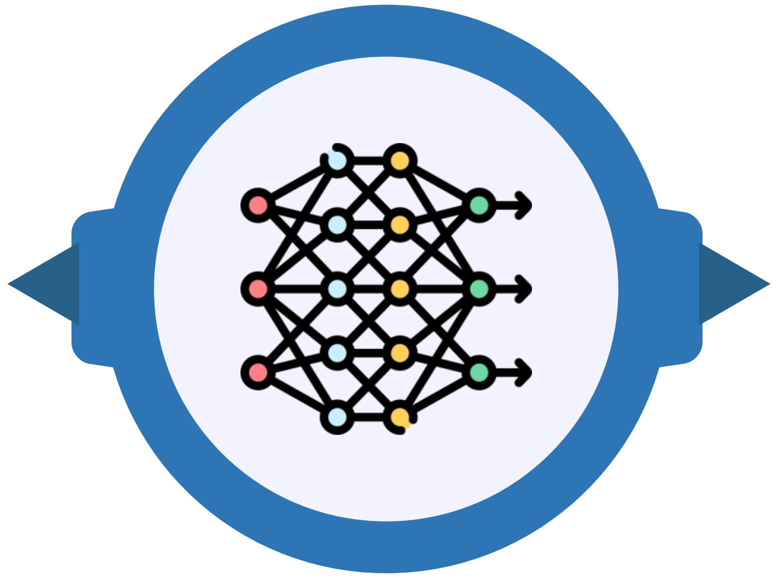
Google DeepMind

Especially for complex tasks, eliminating the need for manual feature engineering.

Scalability Challenge

The Deep Q Network (DQN) also aids in addressing scalability challenges.

A DNN used to approximate Q-values is referred to as a deep Q-network (DQN).



Deep Q-learning utilizes DQNs to effectively approximate Q-values across vast state spaces.

Deep Q-Learning

Deep Q Learning

Deep Q-Learning (DQN) is a reinforcement learning technique that combines Q-Learning with deep neural networks.

It utilizes a neural network, referred to as a deep Q-network, to approximate Q-values.

It is designed to tackle challenges posed by problems featuring high-dimensional state spaces.

This approach has shown significant success in training agents to make decisions and learn optimal strategies in complex environments.

These include playing video games or robotic control tasks.

Training Deep Q Learning

Training the DQN aims to closely approximate the true Q-Values.

Process

- The training process includes observing the immediate reward **r** and the subsequent state **s** after taking action **a** in state **s**.

Training Deep Q Learning

Training the DQN aims to closely approximate the true Q-Values.

Target Q values

- The target Q value $y(s, a)$, is calculated as the observed reward r plus the discounted value of the best possible future action from state **s**.
- The target Q-Value for state-action pair (s, a) is expressed as:

$$y(s,a) = r + \gamma . max_{a'} . Q_\theta(s', a')$$

Training Deep Q Learning

The DQN loss function involves minimizing the difference between the predicted Q-value (Q_θ) for a given state-action pair and the target Q-value (y) associated with that pair.

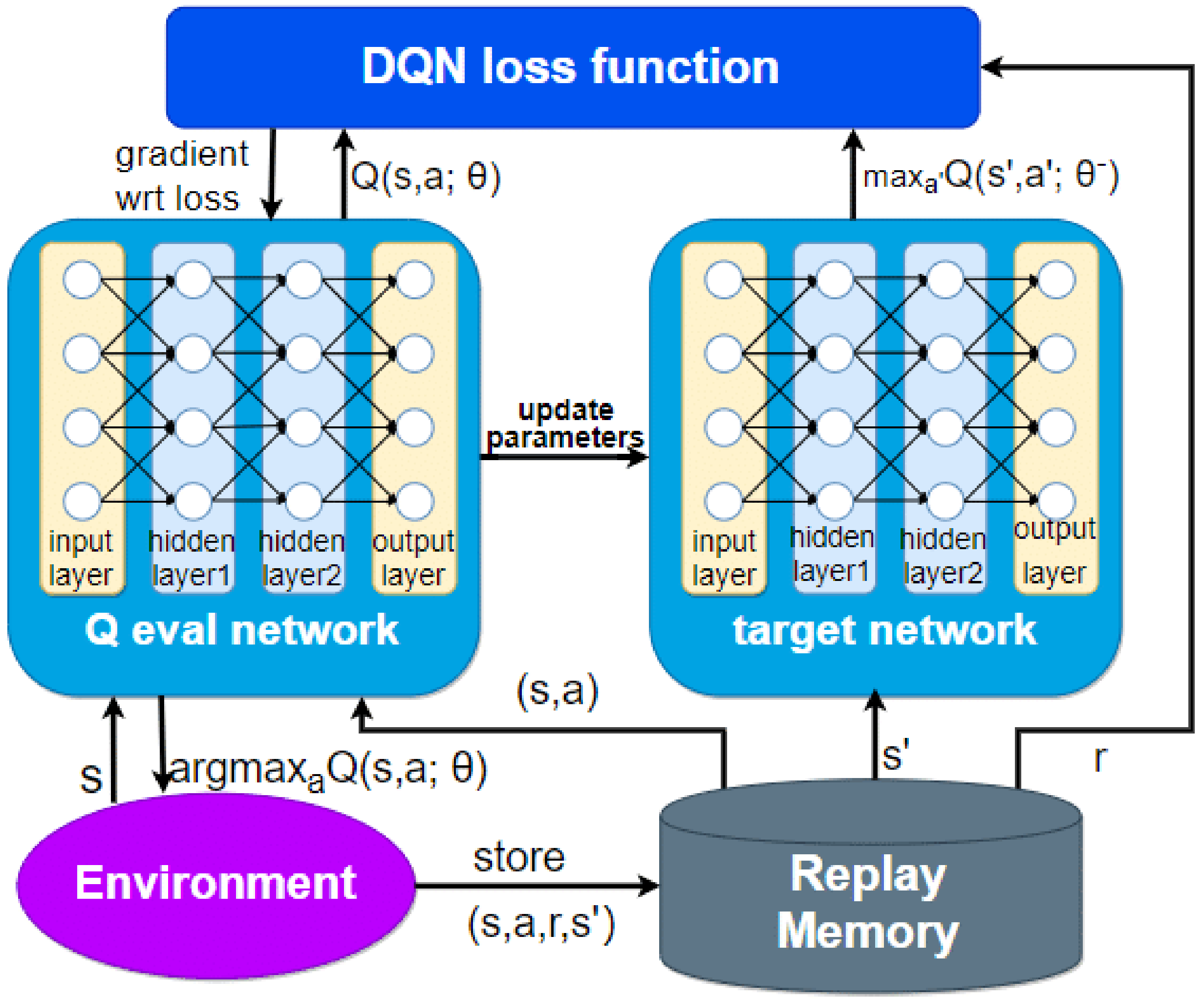
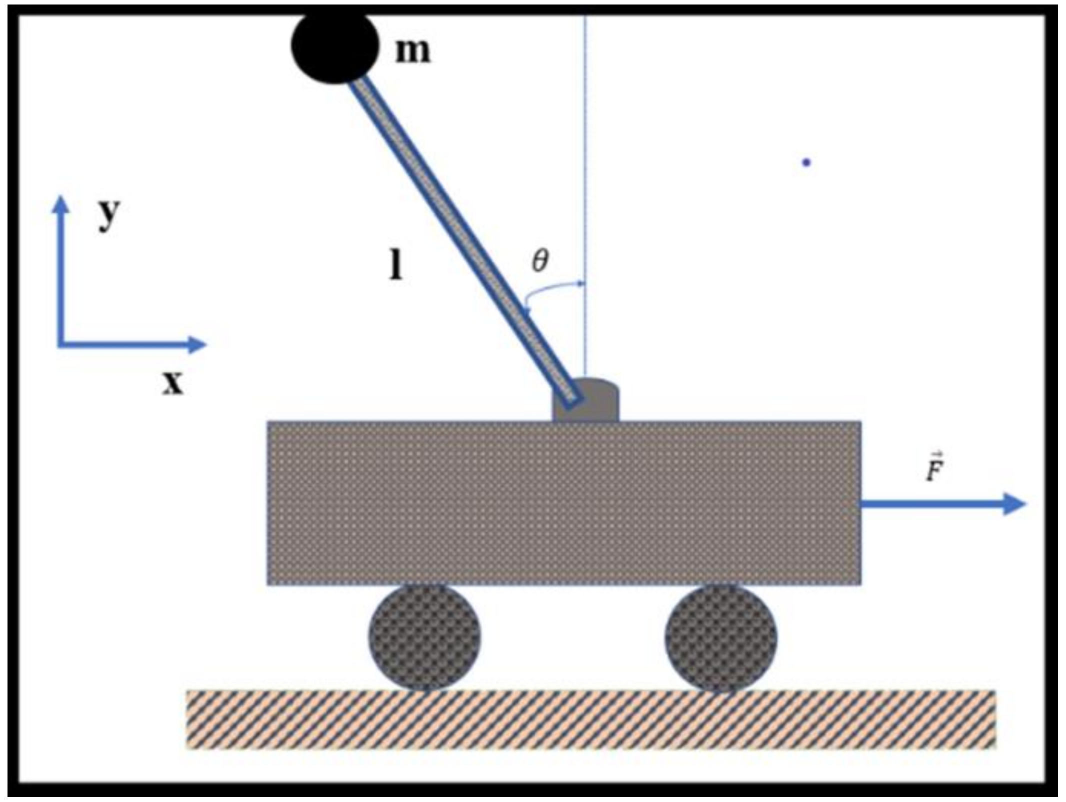


Image source: Learn to Navigate: Cooperative Path Planning for Unmanned Surface Vehicles Using Deep Reinforcement Learning - Scientific Figure on ResearchGate.

This minimization is typically achieved using mean squared error (MSE) or another appropriate loss function.

Deep Q-Learning: Cart-Pole Problem

Let’s consider solving the cart-pole balancing problem using deep Q-learning algorithm.



The objective is to train a deep Q-network (DQN) to control a cart-pole system, optimizing actions to keep the pole balanced.

Image source: A Huber reward function-driven deep reinforcement learning solution for cart-pole balancing problem

Deep Q-Learning: Cart-Pole Problem

Implementing a DQN for environments like cartpole involves a structured approach. It includes:

- Defining the network
- Using an ε-greedy policy for action selection
- Managing experiences through a replay buffer
- Iteratively training the network using sampled batches.

Refer to jupyter notebook for practical implementation

Deep Q-Learning: Cart-Pole Problem

The algorithm includes below steps:

Network architecture for the CartPole

Environments like cart-pole require a simple network with a few hidden layers.

Complexity is not necessarily due to the problem's straight forward nature.

Utilize the openAI Gym library to initialize the environment and define input shape and the number of outputs.

Deep Q-Learning: Cart-Pole Problem

The algorithm includes below steps:

Network architecture for the CartPole

Define the neural network (NN) architecture using the keras API.

Emphasize that the same NN is used for both learning and setting Q-value targets.

Deep Q-Learning: Cart-Pole Problem

The algorithm includes below steps:

Action selection via ε-greedy policy

1

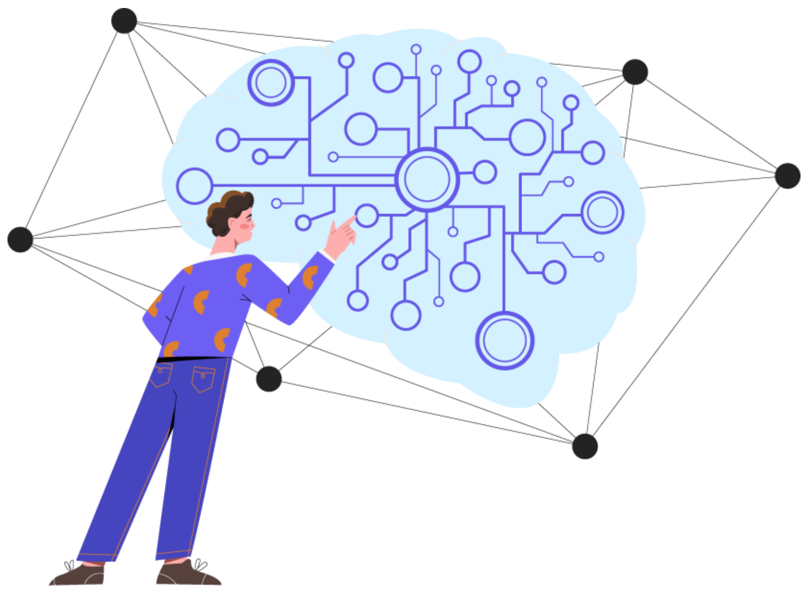
Implement an ε-greedy policy for action selection.

2

3

4

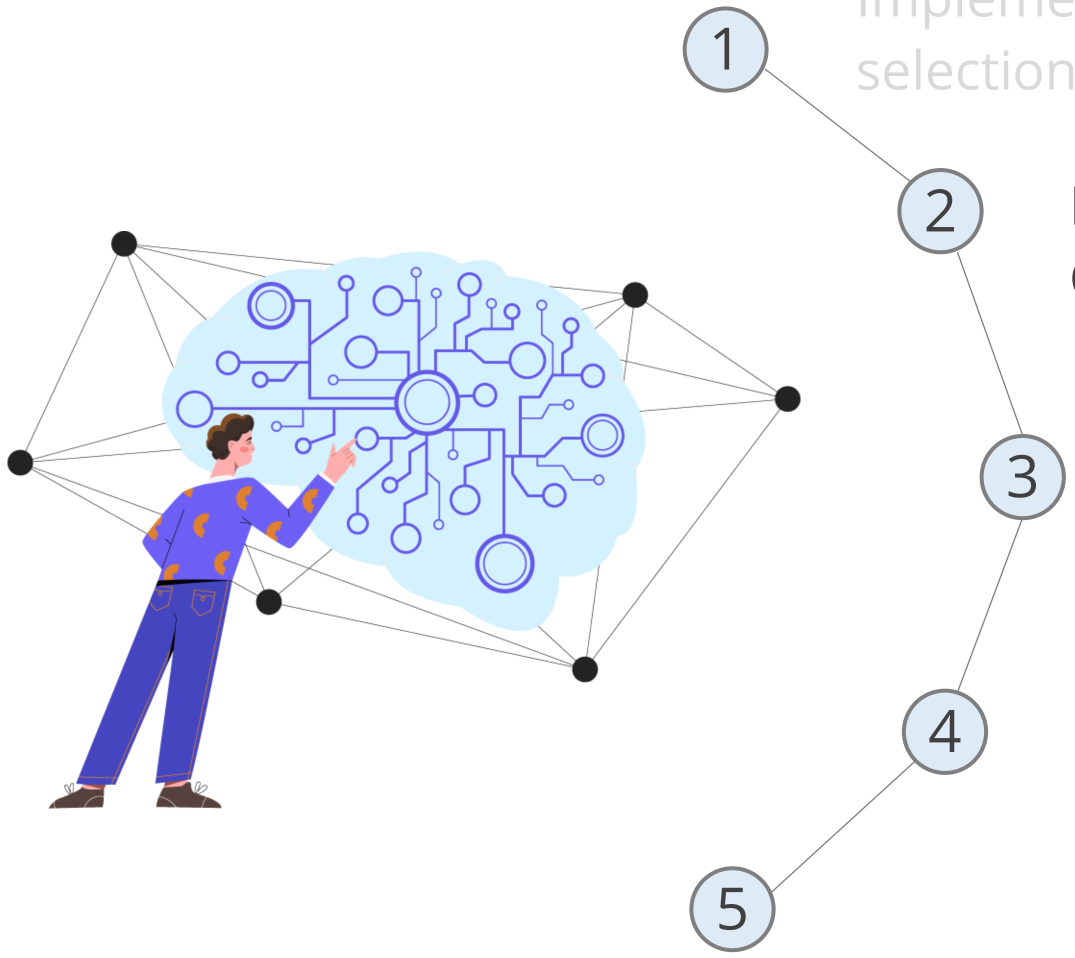
5



Deep Q-Learning: Cart-Pole Problem

The algorithm includes below steps:

Action selection via ε-greedy policy



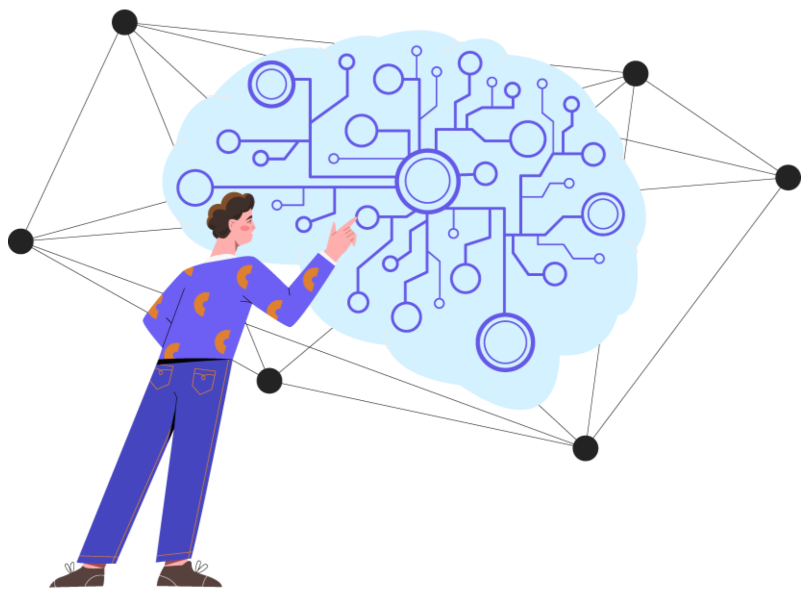
Implement an ε-greedy policy for action selection.

Prefer the action with the highest predicted Q-value (exploitation) most of the time.

Deep Q-Learning: Cart-Pole Problem

The algorithm includes below steps:

Action selection via ε-greedy policy

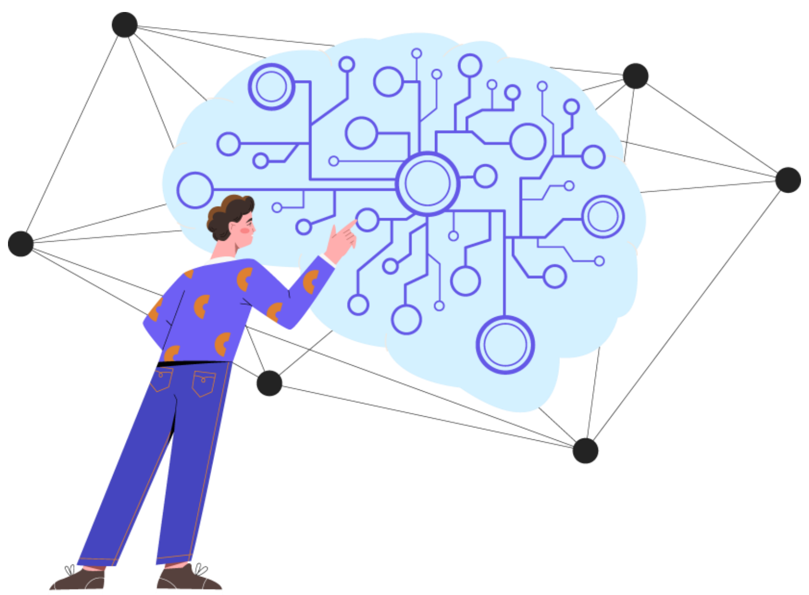


1. Implement an ε-greedy policy for action selection.
2. Prefer the action with the highest predicted Q-value (exploitation) most of the time.
3. Occasionally choose a random action (exploration) with a probability ε.
4.
5.

Deep Q-Learning: Cart-Pole Problem

The algorithm includes below steps:

Action selection via ε-greedy policy



1 Implement an ε-greedy policy for action selection.

2 Prefer the action with the highest predicted Q-value (exploitation) most of the time.

3 Occasionally choose a random action (exploration) with a probability ε.

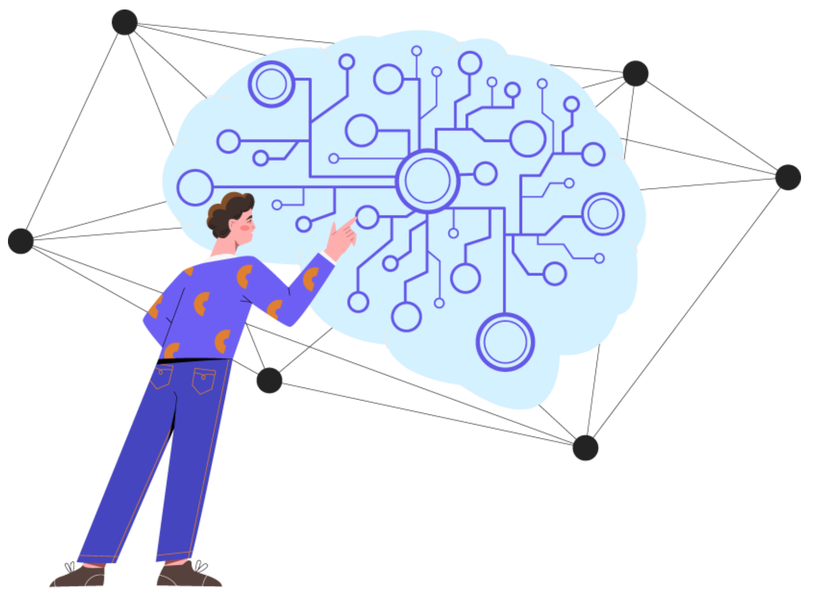
4 Ensure the agent avoids getting stuck in a local optimum and continues learning about the environment.

5

Deep Q-Learning: Cart-Pole Problem

The algorithm includes below steps:

Action selection via ε-greedy policy

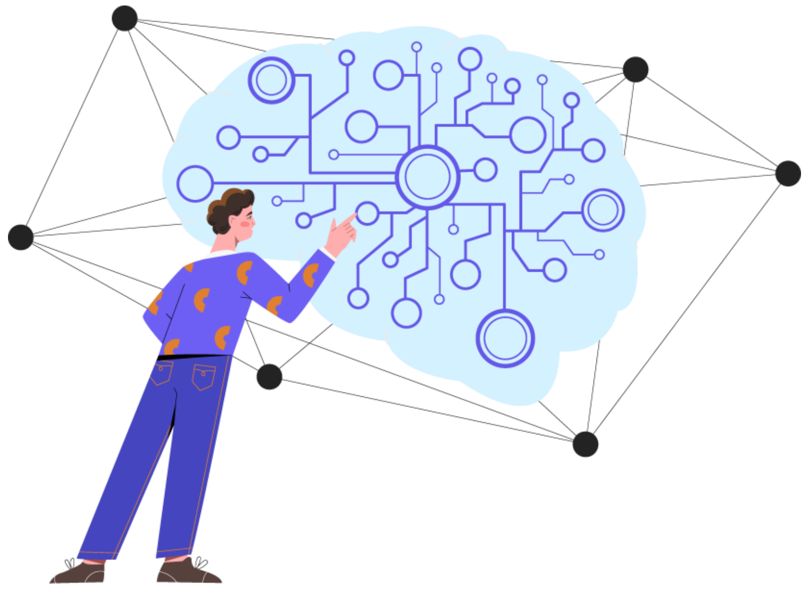


1. Implement an ε-greedy policy for action selection.
2. Prefer the action with the highest predicted Q-value (exploitation) most of the time.
3. Occasionally choose a random action (exploration) with a probability ε.
4. Ensure the agent avoids getting stuck in a local optimum and continues learning about the environment.
5. Achieve this by defining a function that takes 'state' and the epsilon value as input.

Deep Q-Learning: Cart-Pole Problem

The algorithm includes below steps:

Action selection via ε-greedy policy



1. Implement an ε-greedy policy for action selection.
2. Prefer the action with the highest predicted Q-value (exploitation) most of the time.
3. Occasionally choose a random action (exploration) with a probability ε.
4. Ensure the agent avoids getting stuck in a local optimum and continues learning about the environment.
5. Achieve this by defining a function that takes 'state' and the epsilon value as input.

The function returns the max Q-value or a random action based on the epsilon value.

Deep Q-Learning: Cart-Pole Problem

The algorithm includes below steps:

Experience replay buffer

- The experiences that are state, action, reward, next state, done are stored in a replay buffer.
- This reduce correlations between consecutive learning episodes.
- It is crucial for stable and efficient learning.
- It utilizes a deque list for efficient append and pop operations.

Deep Q-Learning: Cart-Pole Problem

The algorithm includes below steps:

Sampling and storing experiences

Sampling

Randomly sample a batch of experiences from the replay buffer for each training iteration.

Randomness is essential for breaking correlations in the observation sequence.

Create a function that takes the batch size and samples experiences from the replay buffer.

Deep Q-Learning: Cart-Pole Problem

The algorithm includes below steps:

Sampling and storing experiences

Storing

After each action using the ε-greedy policy, store the state, action, reward, next state, and done flag in the replay buffer.

Create a function that takes env, state, and epsilon as input.

The function plays a single step using the ε-greedy policy and stores the resulting experience in the replay buffer.

Deep Q-Learning: Cart-Pole Problem

The algorithm includes below steps:

Training process

Batch sampling

- At each training step, begin by sampling a batch of experiences.

Deep Q-Learning: Cart-Pole Problem

The algorithm includes below steps:

Training process

Predicting Q-Values

- Use the DQN to predict the Q-values for the next states.
- Keep only the maximum Q-value for each next state as the best possible future reward.

Deep Q-Learning: Cart-Pole Problem

The algorithm includes below steps:

Training process

Target Q-Value

- Combine the rewards with the maximum predicted next-state Q-values (adjusted by the discount factor) to form the target Q-values for each state-action pair.

Deep Q-Learning: Cart-Pole Problem

The algorithm includes below steps:

Masking irrelevant actions

- The objective is to focus the training on the Q-values of the actions that were actually taken.
- Use ***tf.one_hot()*** to create a mask for the actions taken, apply it to the DQN's output to filter out all Q-values except for those corresponding to the taken actions.

Deep Q-Learning: Cart-Pole Problem

The algorithm includes below steps:

Loss calculation and optimization

- The mean squared error between the predicted Q-values and the target Q-values is used to calculate the loss.
- This loss is minimized by performing a gradient descent step, thus updating the model's weights to better approximate the true Q-values.

Observations and Improvements Required in DQL

Performance Observations

Evaluating the performance of Deep Q Learning involves careful scrutiny of various aspects.

These include:

1

Volatile performance

- In the initial training stages, extended periods of no observable progress may occur.
- Subsequently, a sudden spike in performance might be observed.
- This pattern is often attributed to high exploration rates (ε) employed at the outset of training.

Performance Observations

Evaluating the performance of Deep Q Learning involves careful scrutiny of various aspects. These include:

2

Catastrophic forgetting

- While the algorithm learns about one part of the environment, it may forget information about other parts.
- This phenomenon can result in a significant drop in performance.
- The tendency to forget previously learned information is a common challenge observed across various RL algorithms.

Core Challenges

Deep Q-Learning algorithms also face certain challenges, which include:

Correlated Experiences

- Experiences encountered by the agent often exhibit correlations.
- Correlations in experiences may result in biased learning.
- The risk of overfitting to specific transitions arises due to these correlations.

Core Challenges

Deep Q-Learning algorithms also face certain challenges, which include:

Non-Stationary Environment

- The learning environment undergoes constant changes as the policy updates.
- Gradient descent-based methods face challenges in adapting to this dynamic learning environment.

Strategies for Improvement

Various strategies can be implemented in order to improve the algorithm performance. These include:

Increase Replay Buffer

- A larger replay buffer has the capacity to store a broader range of experiences.
- This capability reduces the impact of catastrophic forgetting and minimizes the influence of correlations in the learning process.

Strategies for Improvement

Various strategies can be implemented in order to improve the algorithm performance. These include:

Adjust Learning Rate

- Decreasing the learning rate can result in more stable, albeit slower, learning.

Strategies for Improvement

Various strategies can be implemented in order to improve the algorithm performance. These include:

Hyperparameter Tuning

- Reinforcement learning is highly sensitive to the selection of hyperparameters, including the neural network architecture.
- Experimentation with various settings is frequently essential to optimize performance.

DeepMind's Approach to Stability

To enhance stability and robustness, DeepMind employs a various approaches to overcome the challenges.

These include:

Enhancements

- The basic DQN demonstrates instability in complex tasks like playing Atari games.
- DeepMind introduces key modifications to stabilize and improve the learning process.

DeepMind's Approach to Stability

To enhance stability and robustness, DeepMind employs a various approaches to overcome the challenges. These include:

Need for Advanced Techniques

The observation prompted the development of advanced techniques, including:

- Fixed Q-Value Targets
- Double DQNs
- Prioritized Experience Replay
- Dueling Networks

Deep Q-Learning Variants

Fixed Q-Value Targets

In traditional deep Q-learning, the same neural network is used to predict both the current Q-values and the target Q-values for the next state.

This approach, however, can lead to instability due to a feedback loop.

To address this, the concept of **fixed Q-value targets** was introduced.

Fixed Q-Value Targets

The traditional DQN is modified in order to stabilize learning by employing two networks.

An online network

for learning/ exploration.

A target network

for setting stable Q-value targets.

Fixed Q-Value Targets

The algorithm can be implemented as below:

Target network

- A clone of the online network, updated less frequently to provide stable targets.

Fixed Q-Value Targets

The algorithm can be implemented as below:

Updating target

- Update the target network's weights to match the online network's weights.
- Perform this update periodically, for example, every 50 steps.
- The purpose is to reduce the correlation between target and predicted Q-values.

Fixed Q-Value Targets

A crucial adjustment implemented by DeepMind to enhance the overall effectiveness of deep Q-learning algorithms.

For parameters for Atari games, DeepMind's researchers

- Utilized a small learning rate (0.00025).
- Updated the target model every 10,000 steps.
- Employed a large replay buffer (1 million experiences).
- Gradually reduced epsilon over 1 million steps.
- Gradually reduced epsilon over 1 million steps.

Fixed Q-Value Targets

A crucial adjustment implemented by DeepMind to enhance the overall effectiveness of deep Q-learning algorithms.

Infrequent updates to the target network played a crucial role.

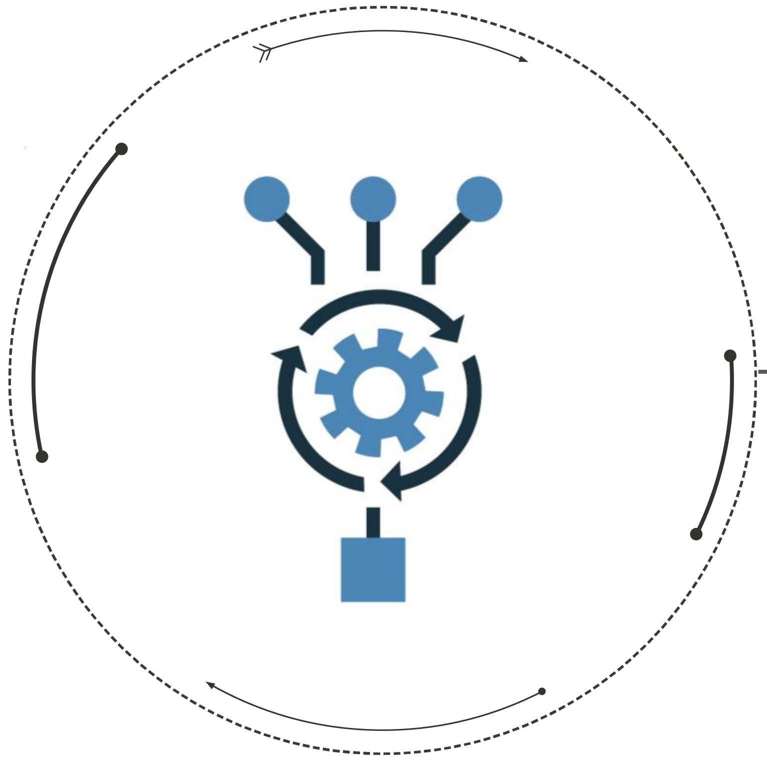
These settings significantly contributed to stabilizing the network.

Enabled agents to learn complex tasks, including playing Atari games from raw pixels.

Fixed Q-Value Targets

To summarize, fixed Q value target algorithm provides certain benefits over the traditional algorithm.

Fixed Q-value targets significantly improve the stability of the learning process in deep Q-learning, leading to better performance in complex environments.



This approach is one of many in the evolution of DQN algorithms, leading to more sophisticated and capable RL agents.

Double DQN

Double DQN was introduced by DeepMind in 2015. It is a variant that increases performance and stabilizes training of the Deep Q-Network.

The traditional DQN's target network tends to overestimate Q-values, as it always selects the maximum Q-value for the next state, leading to biased high-value predictions.

Double DQN addresses this by decoupling the action selection from the target Q-value generation.

Double DQN

Decoupling Selection and Evaluation involves:

Action selection

Use the online network to select what it believes is the best action for the next state.

Q-value estimation

Use the target network to provide the Q-value estimate for these selected actions

Double DQN

This approach reduces the tendency to overestimate Q-values and leads to more reliable learning.

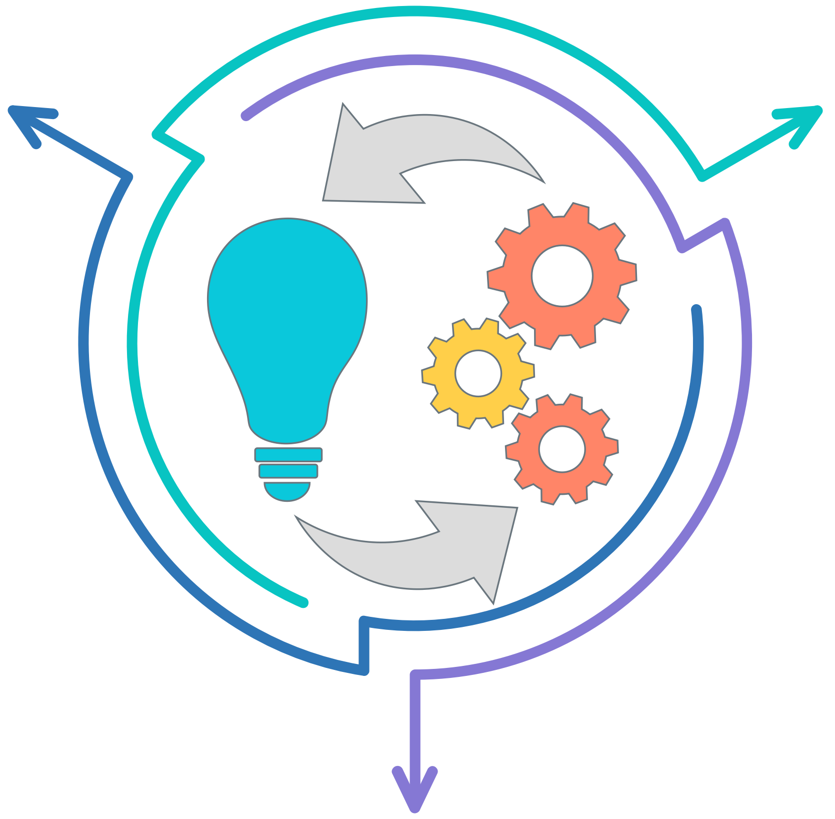
By mitigating the overestimation bias, Double DQN enhances the stability and reliability of the training process.

Due to its effectiveness, Double DQN has become a standard component in many DQN architectures and reinforcement learning studies.

Prioritized Experience Replay

Prioritized Experience Replay (PER) is an enhancement to traditional experience replay.

PER prioritizes sampling of more important experiences.



Importance is determined by their potential to facilitate faster learning progress.

It improves the efficiency and effectiveness of the learning process in reinforcement learning.

Prioritized Experience Replay

Experiences are deemed important based on the magnitude of their Temporal Difference (TD) error.

TD error is represented as:

$$\delta = r + \gamma . V(s') - V(s)$$

A large TD error signifies a surprising or unexpected transition, suggesting high learning potential from that experience.

Prioritized Experience Replay

Implementation of PER can be done as below:

Priority assignment

Initially, experiences are assigned high priority for at least one sampling.

Upon sampling, an experience's priority is updated to its absolute TD error.

This dynamic updating of sampling probabilities ensures effective learning progress.

Prioritized Experience Replay

Implementation of PER can be done as below:

Sampling probability

The probability of sampling an experience is directly proportional to its priority.

The priority is raised to the power of 3, determined by a hyperparameter

The hyperparameter controls the extent of prioritization in the sampling process.

Prioritized Experience Replay

PER introduces a bias towards more important experiences in sampling.

It is crucial to adjust the training weights of each sampled experience.

Prioritized experience replay

Prevents the model from overfitting to frequent, high-priority experiences.

Prioritized Experience Replay

Each experience's weight is expressed as:

$$w = (n * P)^{-\beta}$$

Here,

- n is the number of experiences in the buffer.
- ß is a hyperparameter that controls the compensation for sampling bias.

Prioritized Experience Replay

The hyperparameter adjustment for PER is done for following parameters:

ζ

- Controls the greediness of sampling.
- 0 corresponds to a uniform distribution, and 1 signifies full prioritization.

β

- Governs the bias compensation.
- Starts with less compensation and progresses to full compensation towards the end of training.

In practice, ζ is often set as 0.6 and ß is increased from 0.4 to 1 across the training period, as suggested in the original paper.

Prioritized Experience Replay

Using Prioritized Experience Replay (PER) in reinforcement learning changes the way we prioritize important experiences, transforming how we learn.

By focusing on transitions that have the most to teach, PER accelerates learning and can lead to more robust policies.

It is essential to fine-tune $\boldsymbol{\zeta}$ and $\boldsymbol{\beta}$ for the specific task, and to find balance the benefits of focused learning with the need for diverse experiences.

Key Takeaways

- Q values (or action-values) signify the expected cumulative rewards associated with taking a specific action in a given state.
- Q-Learning works by watching an agent play and gradually improving its estimates of Q-Value.
- In epsilon-greedy policies, the agent makes decisions by considering both exploitation and exploration policies.
- Deep Q-Learning (DQN) is a reinforcement learning technique that combines Q-Learning with deep neural networks.

Knowledge Check

Knowledge Check

1

Which algorithm is often used to stabilize training in Deep Q Learning by addressing the problem of overestimation of Q-values?

A. Double Q-Learning

B. Dueling DQN

C. Policy Gradient

D. Actor-Critic

Knowledge Check

1

Which algorithm is often used to stabilize training in Deep Q Learning by addressing the problem of overestimation of Q-values?

A. Double Q-Learning

B. Dueling DQN

C. Policy Gradient

D. Actor-Critic

The correct answer is **A**

Double Q-Learning is commonly used to address the overestimation of Q-values by using two separate networks to estimate the action values, reducing the overestimation bias.

Knowledge Check

2

What is the role of the epsilon-greedy strategy in deep Q learning?

A. Encourages exploration by selecting the action with the highest Q-value.

B. Discourages exploration by always selecting the action with the highest Q-value.

C. Balances exploration and exploitation by randomly selecting actions with a certain probability.

D. Always selects actions randomly to explore the environment thoroughly.

Knowledge Check

2

What is the role of the epsilon-greedy strategy in deep Q learning?

A. Encourages exploration by selecting the action with the highest Q-value.

B. Discourages exploration by always selecting the action with the highest Q-value.

C. Balances exploration and exploitation by randomly selecting actions with a certain probability.

D. Always selects actions randomly to explore the environment thoroughly.

The correct answer is **C**

The epsilon-greedy strategy balances exploration and exploitation by randomly selecting actions with a certain probability (epsilon) and choosing the action with the highest Q-value with probability (1 - epsilon).

Knowledge Check

3

What is the temporal difference error in the context of deep Q learning?

A. The time it takes for the agent to make a decision.

B. The difference between the predicted Q-value and the target Q-value.

C. The time delay in the reward signal.

D. The difference between the state-action value and the state value.

Knowledge Check 3

What is the temporal difference error in the context of deep Q learning?

A. The time it takes for the agent to make a decision.

B. The difference between the predicted Q-value and the target Q-value.

C. The time delay in the reward signal.

D. The difference between the state-action value and the state value.

The correct answer is **D**

The temporal difference error is the difference between the predicted Q-value and the target Q-value. It is used in the Q-learning update equation to adjust the Q-values during the learning process.

Thank You