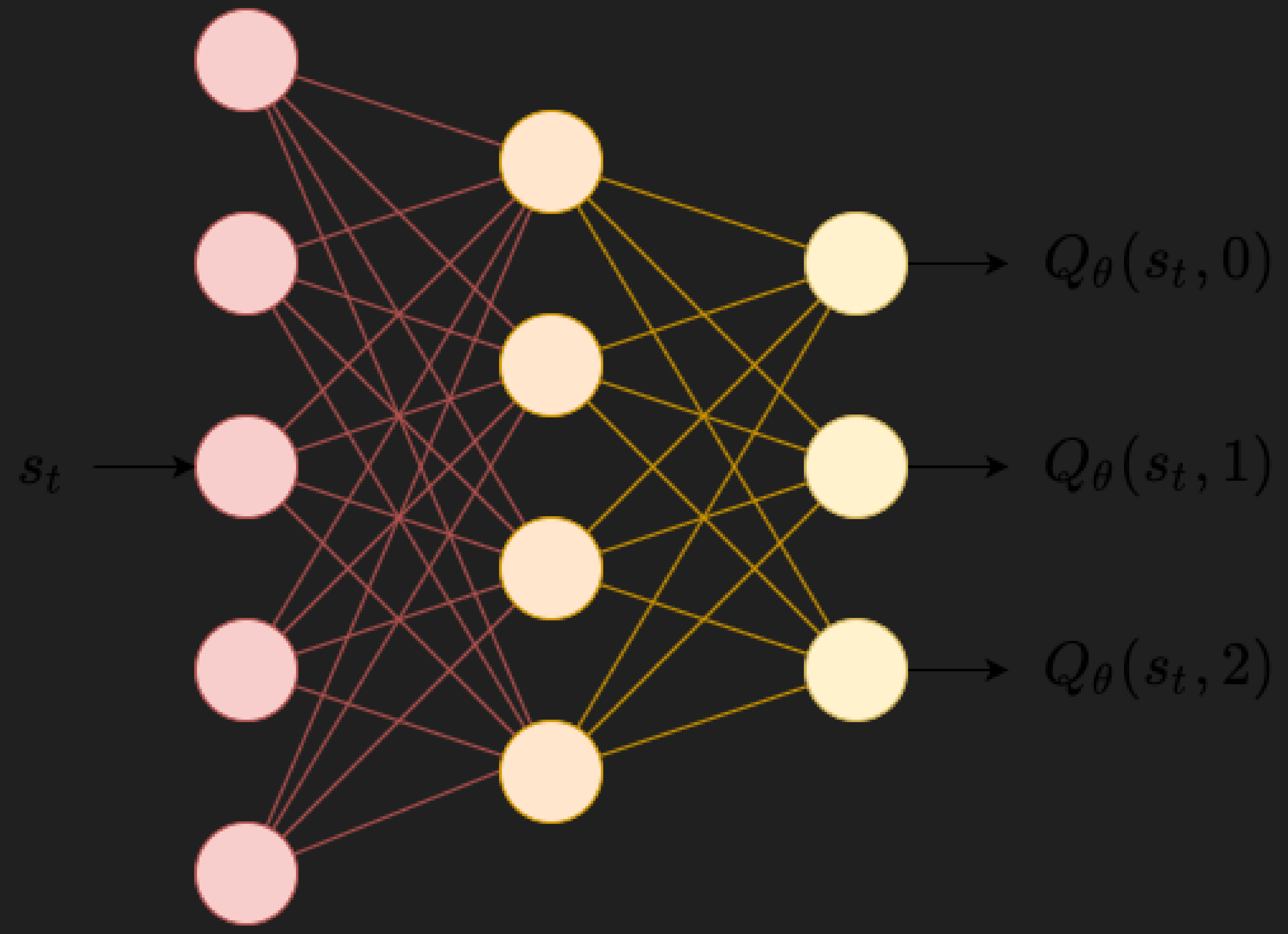
Midterm Report

Q-LEARNING & DQN

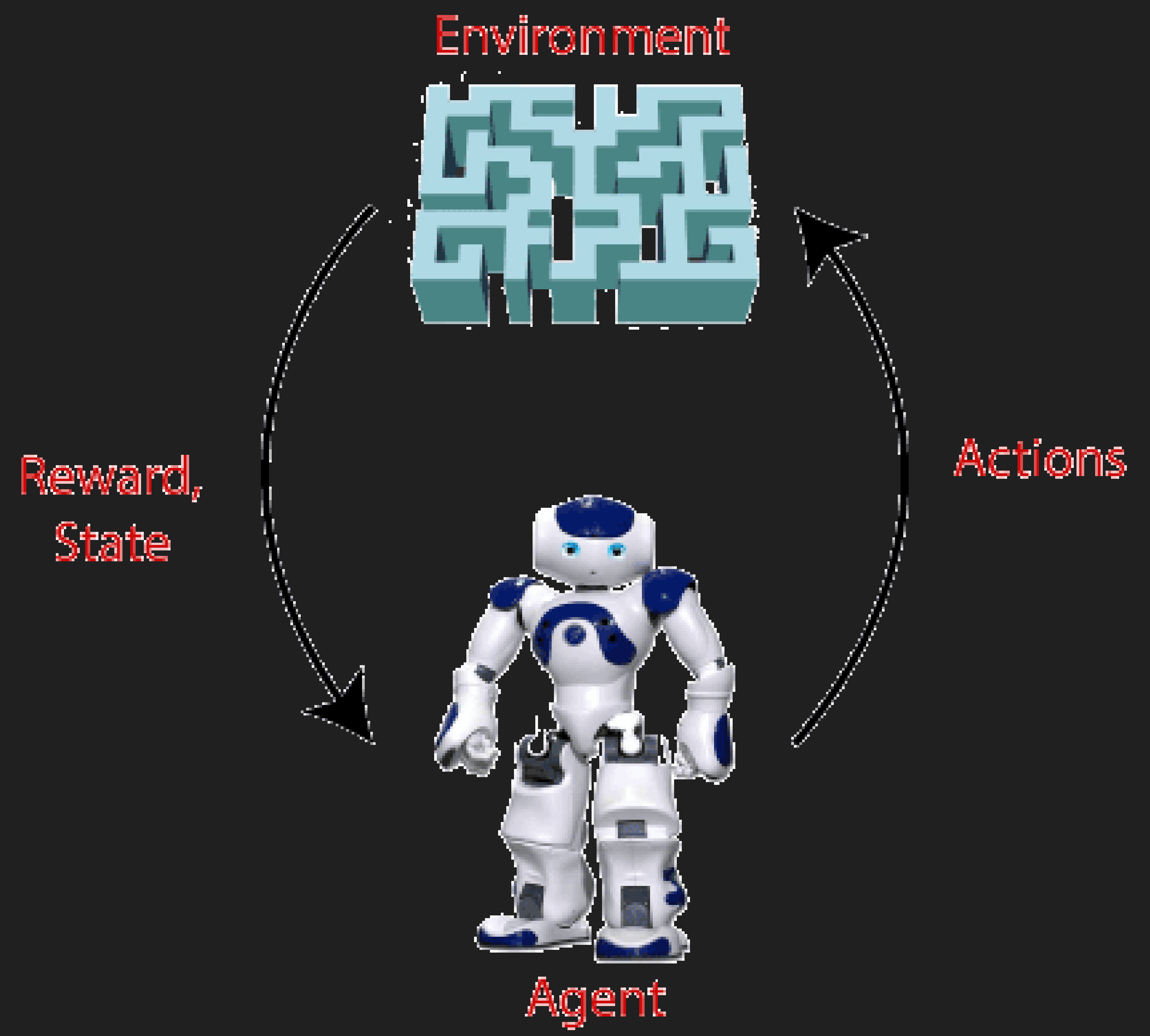


Soham Joshi
Chinmay Tripurawar
Shasant Jindal
Swarup Patil

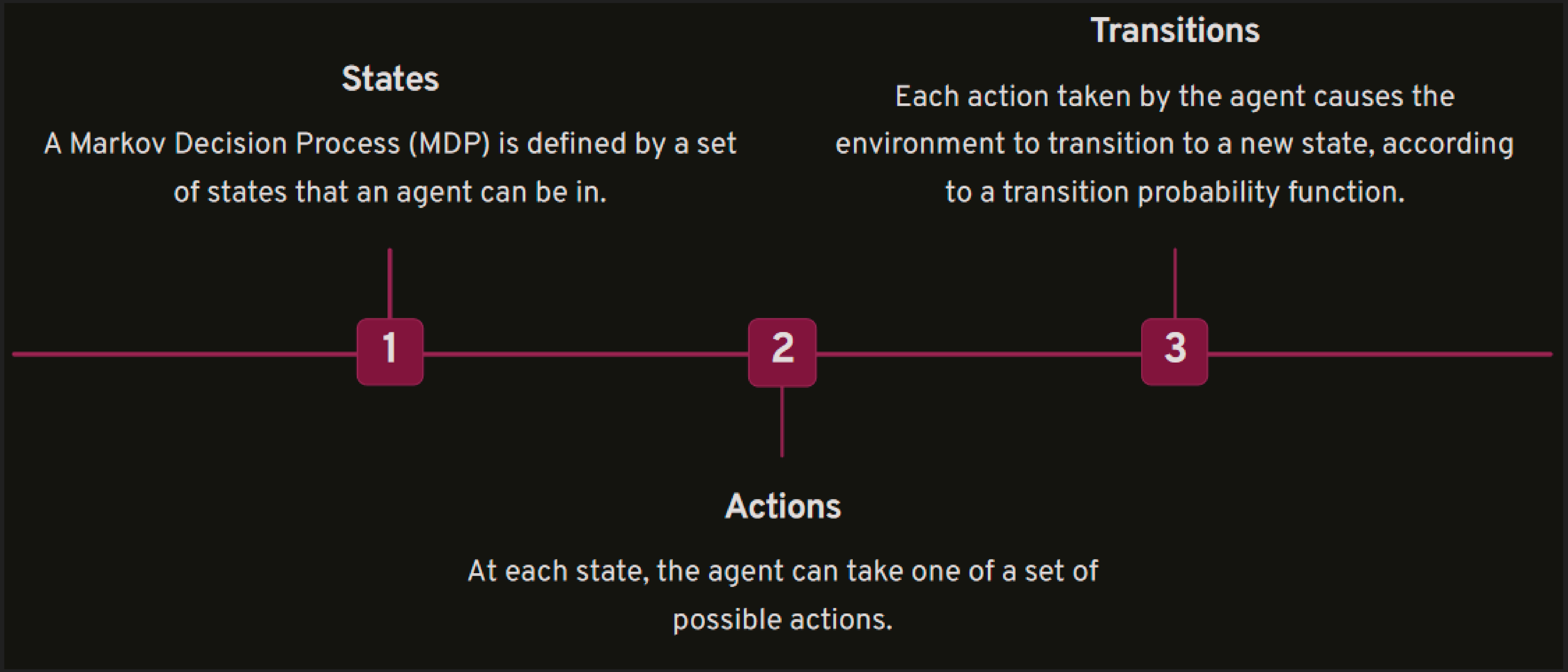
REINFORCEMENT LEARNING

"Reinforcement learning is where an agent learns by trial and error to maximize rewards"

- Core Concept: Reinforcement learning trains an agent to interact with an environment and maximize rewards through trial and error, using feedback from rewards or penalties.
- Key Elements: It involves an agent, environment, actions, and rewards. The agent's goal is to learn a policy that maps states to actions for maximizing total rewards over time.
- Applications and Algorithms: Algorithms like Q-learning and DQNs use deep neural networks to learn complex policies, successfully applied in gaming, robotics, and resource management.



MARKOV DECISION PROCESSES



Key Properties of MDPs

- Markov property: future states depend only on the current state and action, not the history
- Stochastic transitions: the next state is determined probabilistically based on the current state and action
- Delayed rewards: the agent may not receive an immediate reward for each action, but rather a sequence of rewards over time

THE Q-LEARNING ALGORITHM

State

Current situation of the agent in the environment.

Action

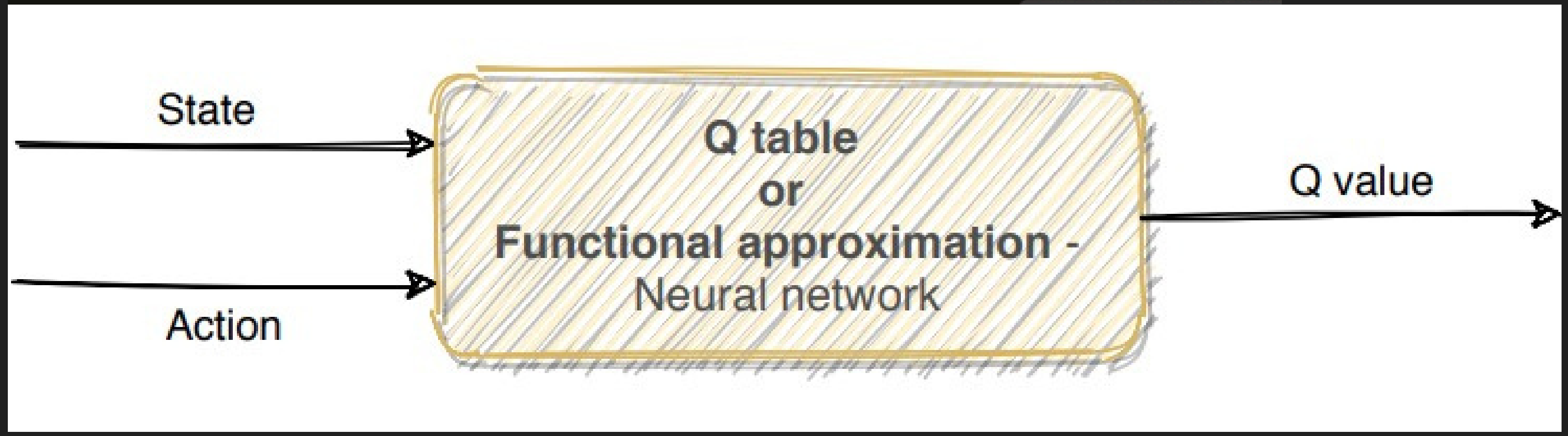
Choice made by the agent to influence the environment.

Reward

Feedback received by the agent for performing the action.

Next State

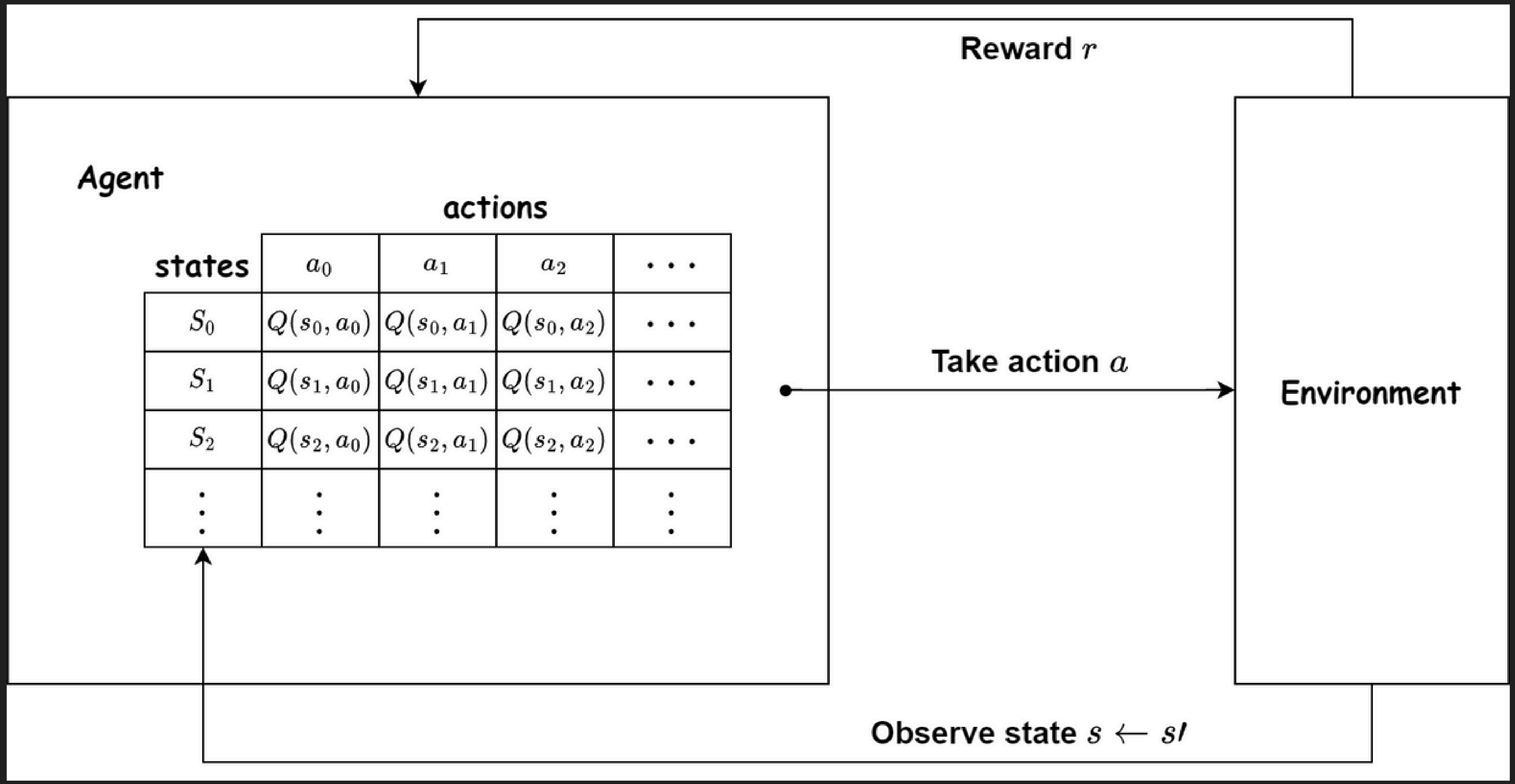
The state the agent transitions to after taking the action.



- Q learning is a type of value based method that is used to find optimal policy that maximizes the total reward
- It inputs state and action and maps them to a real value
- It use Q table to store all the Q-values of each possible state and action
- It iteratively updates the Q-values in Q-table using temporal difference and Bellman equations
- The policy derived from Q learning is that for any state the action with maximum Q-value the is taken

Q TABLE

- State-Action Values: A Q-table stores the values (Q-values) for each state-action pair, representing the expected reward of taking a particular action in a given state.
- Lookup Table: It serves as a lookup table where rows represent states, columns represent actions, and each cell contains the Q-value, guiding the agent in selecting the best action.
- Learning and Updating: During the learning process, the Q-table is iteratively updated based on the agent's experiences, improving the policy by increasing the accuracy of the Q-values through exploration and exploitation.



TEMPORAL DIFFERENCE AND BELLMAN EQUATION

Temporal Difference

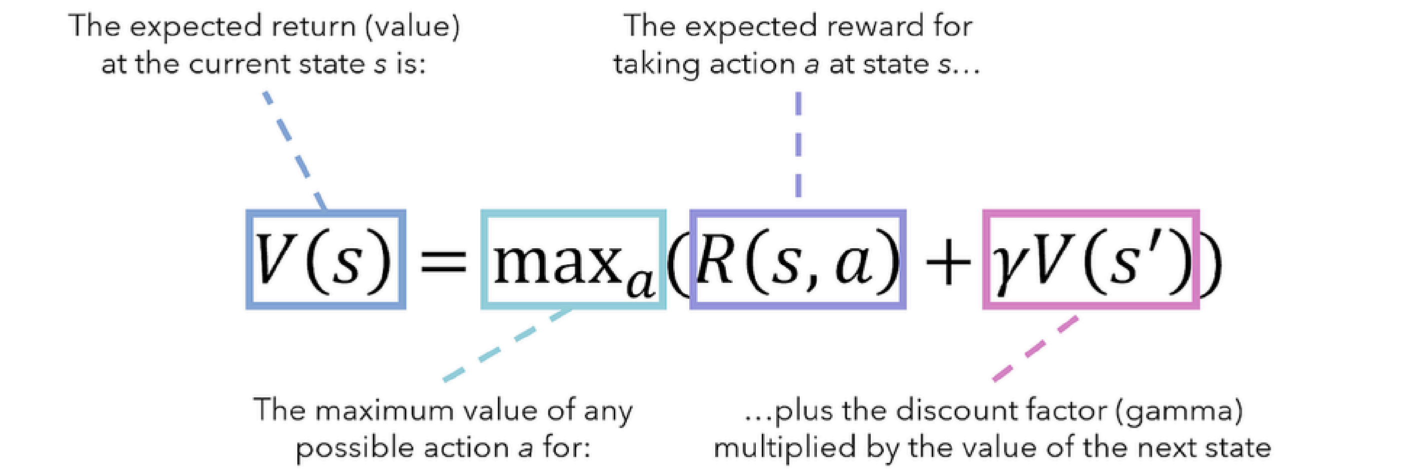
- Combination of Monte Carlo and Dynamic Programming: TD learning is a reinforcement learning method that combines ideas from Monte Carlo methods (which learn from complete episodes) and dynamic programming (which updates estimates based on other learned estimates).
- Bootstrapping: Unlike Monte Carlo methods, TD learning updates its estimates based partly on other learned estimates without waiting for the final outcome. This is known as bootstrapping.
- TD Update Rule: The core of TD learning is the update rule, which adjusts the value of a state based on the difference (temporal difference) between the predicted value and the actual reward received plus the estimated value of the next state.

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Initialize $V(s)$ arbitrarily (e.g., $V(s) = 0, \forall s \in \mathcal{S}^+$)
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
    $S \leftarrow S'$
  until S is terminal

Bellman equation

- Recursive Definition of Value: The Bellman equation provides a recursive definition for the value of a policy. It expresses the value of a state as the expected return of the immediate reward plus the discounted value of the next state, assuming the agent follows a particular policy.
- Optimality Principle: The Bellman optimality equation refines this by defining the optimal policy, stating that the value of a state under the optimal policy is the maximum expected return achievable by any policy.
- Foundation for Algorithms: Both dynamic programming methods like Value Iteration and Q-learning algorithms are based on the Bellman equation, as it helps iteratively improve the value estimates until convergence.



The DQN Architecture

DQN employs a deep neural network to approximate the Q-function. The network takes a state as input and outputs the estimated Q-values for all possible actions. The network is trained using a loss function that encourages it to predict accurate Q-values.

Input Layer

Represents the state of the environment, typically encoded as a vector or image.

Hidden Layers

Extract features and relationships from the input, allowing the network to learn complex patterns.

Output Layer

Produces the estimated Q-values for each possible action, forming the network's output.

Combining Q-Learning with Deep Learning:

- Deep Q-Network (DQN): DQN is an extension of Q-learning that leverages deep neural networks to approximate the Q-values, allowing it to handle high-dimensional state spaces such as those in video games.
- Function Approximation: Instead of using a Q-table, DQN uses a neural network to estimate the Q-values for each state-action pair

Key Components:

- Replay Buffer: Experiences (state, action, reward, next state) are stored in a replay buffer. The network is trained on mini-batches of experiences sampled from this buffer to break the correlation between consecutive samples and stabilize training.
- Target Network: DQN maintains a separate target network with the same architecture as the Q-network. The target network's weights are periodically updated to match the Q-network, reducing the oscillations and divergence during training.

Training Process:

- Experience Collection: The agent interacts with the environment, collects experiences, and stores them in the replay buffer.
- Mini-Batch Updates: Random mini-batches of experiences are sampled from the replay buffer. The Q-network is trained to minimize the loss, which is the mean squared error between the predicted Q-values and the target Q-values.
- Target Q-Value: The target Q-value is calculated using the Bellman equation: y=r+γmaxa′Q′(s′,a′)y = r + \gamma \max_{a'} Q'(s', a')y=r+γmaxa′Q′(s′,a′), where Q′Q'Q′ represents the target network. This target is used to update the Q-network.

Experience Replay and Target Networks

To address the instability of learning from highly correlated data, DQN uses experience replay. This technique stores past experiences in a memory buffer and randomly samples from it during training, breaking the correlation and enhancing learning stability.

1. **Experience Replay**

   Stores past experiences (state, action, reward, next state) in a memory buffer, allowing for efficient data reuse and reduced correlation.

2. **Target Network**

   A separate neural network used to compute the target Q-values during training. The target network's weights are updated less frequently, helping to stabilize learning.

3. **Deep Neural Network**

   Approximates the Q-function, taking a state as input and outputting estimated Q-values for all possible actions.

Limitations and Future Developments of DQN

Despite its impressive achievements, DQN faces challenges, particularly in handling high-dimensional state spaces and dealing with continuous action spaces. Ongoing research focuses on addressing these limitations and expanding the capabilities of DQN.

1. **High-Dimensional State Spaces**

   Handling complex environments with numerous states poses a challenge, requiring efficient feature extraction and representation learning.

2. **Continuous Action Spaces**

   Dealing with actions that can take on a continuous range of values requires specialized techniques and can impact learning efficiency.

3. **Sample Efficiency**

   DQN often requires a significant amount of training data to achieve optimal performance, limiting its applicability in scenarios with limited data.

Code for Inverted Pendulum using DQN Algorithm

class QNetwork:

- The QNetwork class defines a neural network architecture designed to estimate Q-values, which are measures of the value of taking a particular action from a given state.
- The network takes a state as input and outputs Q-values for all possible actions. This is crucial in reinforcement learning for selecting optimal actions.

get_action_dqn( ):

- The get_action_dqn function is responsible for selecting actions based on an epsilon-greedy policy.
- In this policy, with probability epsilon, a random action is selected (to encourage exploration), and with probability (1 - epsilon), the action with the highest estimated Q-value is chosen (to exploit the knowledge gained so far). Epsilon decays over time to shift the balance from exploration to exploitation.

prepare_batch( ):

- The prepare_batch function samples a batch of experiences from the replay memory. This replay memory stores past experiences in the form of state, action, reward, next state, and done flag.
- The sampled batch is then converted into tensors and transferred to the GPU for efficient training of the neural network.

learn_dqn ( ):

- The learn_dqn function handles the training of the Q-network. It calculates the DQN (Deep Q-Network) loss function, which typically involves the mean squared error between the predicted Q-values and target Q-values.
- The Q-network is updated using this loss, and periodically, the target network (a stable copy of the Q-network) is updated to match the current Q-network, ensuring stable learning.

dqn_main( ):

- Initialization:

1. Sets hyperparameters such as learning rate, discount factor, epsilon values, and batch size.
2. Initializes the environment where the agent will interact.
3. Creates instances of the Q-network and target network.
4. Sets up the optimizer for training the neural network.
5. Initializes the replay buffer to store experiences.

- Training Loop:

1. Runs episodes where the agent interacts with the environment.
2. Selects actions using the epsilon-greedy policy defined in get_action_dqn.
3. Stores the resulting experiences (state, action, reward, next state, done) in the replay memory.
4. Periodically samples batches from the replay memory and trains the Q-network using the learn_dqn function.
5. Periodically updates the target network to match the current Q-network.
6. Records cumulative rewards for each episode to track the agent's performance over time.

- Visualization:

After training, plots the cumulative rewards over episodes to visualize the agent's training progress and performance. This helps in understanding how well the agent is learning to maximize its rewards.

- Execution:

The dqn_main function orchestrates the entire process, from initialization to training and visualization. Running this function will execute the entire Deep Q-Network training pipeline and generate plots to visualize the performance of the agent.

THANK YOU