```
# Cell 1: Deep Q-Network (DQN) Simplified Demonstration

# PYTHON CONCEPT: Dictionaries and Q-Learning Basics
# This is a simplified version of DQN to help understand t
class SimpleDQN:
    def __init__(self):
        # Q-table as a dictionary: (state, action) -> valu
        self.q_table = {}
        self.learning_rate = 0.1
        self.discount_factor = 0.9

    def get_q_value(self, state, action):
        # Get Q-value for state-action pair, default 0 if
        return self.q_table.get((state, action), 0.0)

    def update(self, state, action, reward, next_state):
        # Simple Q-learning update
        old_value = self.get_q_value(state, action)
        next_max = max([self.get_q_value(next_state, a) fo

        # Q-learning formula simplified
        new_value = old_value + self.learning_rate * (
            reward + self.discount_factor * next_max - olc
        )
        self.q_table[(state, action)] = new_value

# Example usage
dqn_agent = SimpleDQN()
state = "position_3_obstacle_ahead"
action = "stop"
reward = 10
next_state = "position_3_clear_path"

# Update Q-values
dqn_agent.update(state, action, reward, next_state)
print(f"Updated Q-value: {dqn_agent.q_table[(state, actior
```

```
Updated Q-value: 1.0
```

```
# Cell 2: Policy Gradient Simplified Example

# PYTHON CONCEPT: Probability-based Decision Making
import random

class SimplePolicyAgent:
    def __init__(self):
        # Policy as probability distribution
        self.action_probabilities = {
```

Explain code ✕

You

Please explain this code.

Gemini

Okay, let's break down this code which demonstrates a simplified version of Federated Learning.

Cell 4: Simplified Federated Learning Example

This code simulates a basic Federated Learning system using the previously defined `SimpleDQN` (Deep Q-Network) class.

Core Idea: Federated Learning allows multiple agents (think of them as separate devices or systems) to learn collaborativel without directly sharing their data. Each agent trains on its own data, then they share their *learned knowledge* (in this case, Q-values) which gets combined to improve everyone's learning.

Code Explanation:

1. **`SimpleFederatedSystem` Class:**
    - `__init__(self, num_agents=3)`: Cr s a federated system a specified numb agents (default is 3). Each agent is an instance of the `SimpleDQN` class.

```
            'move': 0.5,
            'stop': 0.5
        }
        self.learning_rate = 0.1

    def choose_action(self):
        # Choose action based on probabilities
        return random.choices(
            list(self.action_probabilities.keys()),
            list(self.action_probabilities.values())
        )[0]

    def update_policy(self, action, reward):
        # Simple policy update based on reward
        if reward > 0:
            # Increase probability of successful action
            self.action_probabilities[action] += self.le
            # Normalize probabilities
            total = sum(self.action_probabilities.values
            for action in self.action_probabilities:
                self.action_probabilities[action] /= tot

# Example usage
policy_agent = SimplePolicyAgent()
for _ in range(5):
    action = policy_agent.choose_action()
    reward = 1 if action == 'move' else -1  # Example re
    policy_agent.update_policy(action, reward)
    print(f"Action probabilities: {policy_agent.action_p
```

```
Action probabilities: {'move': 0.5454545454545454, '
Action probabilities: {'move': 0.5454545454545454, '
Action probabilities: {'move': 0.5454545454545454, '
Action probabilities: {'move': 0.5454545454545454, '
Action probabilities: {'move': 0.5454545454545454, '
```

```
# Cell 3: Multi-Agent System Demonstration

# PYTHON CONCEPT: Multiple Objects Interaction
# Cell 3: Multi-Agent System Demonstration

# PYTHON CONCEPT: Multiple Objects Interaction

# First, define these supporting components:
class SimpleAgent:
    def __init__(self, world_size):
        self.position = random.randint(0, world_size - 1
        self.world_size = world_size
```

```
class SimpleFederatedSystem:
       def __init__(self, num_
           self.agents = [Simp
```

Use code with caution

- `aggregate_knowledge(self):` This is the heart of the federated learning process. It does the following:
    - **Collects all unique state-action pairs:** It gathers all the different states and actions that the agents have encountered.

```
all_pairs = set()
      for agent in self.agents
          all_pairs.update(age
```

Use code with caution

```
  - **Averages Q-values:** For e
```

```
averaged_q_table = {}
      for state_action in all_
          values = [agent.q_ta
          averaged_q_table
```

Use code with caution

```
  - **Updates agents:** It updat
```

```
for agent in self.agents:
    def sense(self, world):
        return world[self.position]

    def decide(self, observation):
        return 'move' if observation == ' ' else 'stop'

    def act(self, action):
        if action == 'move' and self.position < self.wor
            self.position += 1

def create_world(size):
    return [' ' for _ in range(size)]

# Then, define the main class:
class SimpleMultiAgentSystem:
    def __init__(self, num_agents=3, world_size=10):
        self.agents = [SimpleAgent(world_size) for _ in
        self.world_size = world_size
        self.world = create_world(world_size)


    def __init__(self, num_agents=3, world_size=10):
        self.agents = [SimpleAgent(world_size) for _ in
        self.world_size = world_size
        self.world = create_world(world_size)

    def communicate(self, agent_index):
        """Simulate communication between agents"""
        # Get positions of other agents
        other_positions = [
            a.position for i, a in enumerate(self.agents
            if i != agent_index
        ]
        return other_positions

    def coordinate_actions(self):
        """Coordinate actions between agents"""
        for i, agent in enumerate(self.agents):
            # Get information about other agents
            other_positions = self.communicate(i)

            # Simple coordination: don't move if another
            observation = agent.sense(self.world)
            if agent.position + 1 in other_positions:
                action = 'stop'
            else:
                action = agent.decide(observation)

            agent.act(action)

def visualize_multi_agent(world, agent_positions):
```

```
        agent.q_table = aver
```

Use code with caution

2. **Example Usage:**

    - `fed_system = SimpleFederatedSystem(` Creates an instance of the federated system.
    - **Simulate different experiences:** Gives each agent some initial, random Q-values to represent their individual learning.

```
for i, agent in enumerate(fed_
    state = f"position_{i}"
    action = "move"
    agent.q_table[(state, a
```

Use code with caution

- **Print Q-tables before and after aggregation:** Shows the Q-tables of the agents before and after the `aggregate_knowledge()` function is called to demonstrate how the knowledge is shared and combined.

```
print("Before aggregation:")
  # ... (printing logic)

  fed_system.aggregate_knowle

  print("\nAfter aggregation:
  # ... (printing logic)
```

```
    """Visualize multiple agents in the world"""
    visualization = "+--" * len(world) + "+\n|"

    for i in range(len(world)):
        if i in agent_positions:
            # Show agent number
            agent_num = agent_positions.index(i) + 1
            visualization += f"{agent_num} |"
        else:
            visualization += f"{world[i]} |"

    visualization += "\n+" + "--" * (len(world) * 2) + "
    return visualization

# Example usage
multi_agent_system = SimpleMultiAgentSystem(num_agents=3
for step in range(5):
    multi_agent_system.coordinate_actions()
    agent_positions = [agent.position for agent in multi
    print(f"\nStep {step + 1}")
    print(visualize_multi_agent(multi_agent_system.world
```

```
Step 1
+--+--+--+--+--+--+--+--+--+--+
|  |  |  |1 |  |  |  |  |2 |3 |
+----------------------------------------+


Step 2
+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |1 |  |  |  |2 |3 |
+----------------------------------------+


Step 3
+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |1 |  |  |2 |3 |
+----------------------------------------+


Step 4
+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |1 |  |2 |3 |
+----------------------------------------+


Step 5
+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |1 |2 |3 |
+----------------------------------------+
```

```
#Cell 4: Simplified Federated Learning Example
```

Use code with caution

In Simple Terms:

Imagine three friends are learning to play a video game. Each friend plays the game on their own and learns which actions are good or bad in different situations (this is like building their individual Q-tables). In Federated Learning, the friends would then come together and discuss what they've learned. They would combine their knowledge and agree on the best strategies for each situation. Then, they would each go back to playing the game with this improved understanding. That's essentially what this code is simulating. I hope this helps!

▶ Sources

Rate this answer

New

```
#Cell 4: Simplified Federated Learning Example

# PYTHON CONCEPT: Distributed Learning and Averaging
class SimpleFederatedSystem:
    def __init__(self, num_agents=3):
        self.agents = [SimpleDQN() for _ in range(num_age

    def aggregate_knowledge(self):
        """Simulate federated learning by averaging Q-valu
        # Collect all unique state-action pairs
        all_pairs = set()
        for agent in self.agents:
            all_pairs.update(agent.q_table.keys())

        # Average Q-values across agents
        averaged_q_table = {}
        for state_action in all_pairs:
            values = [
                agent.q_table.get(state_action, 0.0)
                for agent in self.agents
            ]
            averaged_q_table[state_action] = sum(values) /

        # Update all agents with averaged values
        for agent in self.agents:
            agent.q_table = averaged_q_table.copy()

# Example usage
fed_system = SimpleFederatedSystem(num_agents=3)

# Simulate different experiences
for i, agent in enumerate(fed_system.agents):
    state = f"position_{i}"
    action = "move"
    agent.q_table[(state, action)] = random.random()

print("Before aggregation:")
for i, agent in enumerate(fed_system.agents):
    print(f"Agent {i} Q-table:", agent.q_table)

fed_system.aggregate_knowledge()

print("\nAfter aggregation:")
for i, agent in enumerate(fed_system.agents):
    print(f"Agent {i} Q-table:", agent.q_table)
```

```
Before aggregation:
Agent 0 Q-table: {('position_0', 'move'): 0.2539239
Agent 1 Q-table: {('position_1', 'move'): 0.0427973
Agent 2 Q-table: {('position_2', 'move'): 0.7378632

After aggregation:
```

```
Agent 0 Q-table: {('position_0', 'move'): 0.08464132
Agent 1 Q-table: {('position_0', 'move'): 0.08464132
Agent 2 Q-table: {('position_0', 'move'): 0.08464132
```

```
#Cell 5: Training Loop with Different Learning Approache

def compare_learning_approaches(episodes=100):
    """Compare different learning approaches"""
    # Initialize agents
    dqn_agent = SimpleDQN()
    policy_agent = SimplePolicyAgent()

    # Training results
    dqn_rewards = []
    policy_rewards = []

    for episode in range(episodes):
        # DQN approach
        state = "start"
        dqn_total_reward = 0
        for _ in range(5):
            action = max(
                ['move', 'stop'],
                key=lambda a: dqn_agent.get_q_value(stat
            )
            reward = random.choice([-1, 1])  # Simplifie
            next_state = f"state_{random.randint(1,5)}"
            dqn_agent.update(state, action, reward, next
            dqn_total_reward += reward
            state = next_state

        # Policy Gradient approach
        policy_total_reward = 0
        for _ in range(5):
            action = policy_agent.choose_action()
            reward = random.choice([-1, 1])  # Simplifie
            policy_agent.update_policy(action, reward)
            policy_total_reward += reward

        dqn_rewards.append(dqn_total_reward)
        policy_rewards.append(policy_total_reward)

        if episode % 10 == 0:
            print(f"\nEpisode {episode}")
            print(f"DQN Average Reward: {sum(dqn_rewards
            print(f"Policy Average Reward: {sum(policy_r
```

Enter a prompt here

0 / 2000

Responses may display inaccurate or offensive information that doesn't represent Google's views. [Learn more](#)

New