**Design Defense: Deep Q-Learning Pirate Agent for Treasure Hunt Maze Game**

In the treasure hunt maze game, the player races against an AI-controlled pirate agent to reach a hidden treasure. The pirate acts as an intelligent agent that must learn to navigate a maze filled with pathways and obstacles, reaching the treasure before the human player. This task is a classic pathfinding problem and was solved using a deep Q-learning algorithm—a reinforcement learning approach that enables the agent to learn optimal strategies through trial and error. The goal of this document is to explain how the pirate agent works, compare its strategy to human problem-solving, and justify the design choices based on fundamental AI principles.

Humans typically solve mazes by visually identifying paths, using logic and memory to avoid previously failed routes, and applying heuristics like the right-hand rule(Keeping the right hand on the wall at all times). They can recognize patterns and adjust strategies quickly. A human solving this maze would start by scanning the maze layout, planning a route from start to goal, and using trial-and-error combined with memory of past attempts to improve.

In contrast, a machine learning agent begins without knowledge of the environment and learns through iterative interaction. In each episode, it tries actions, receives feedback in the form of rewards or penalties, and adjusts its internal model (a neural network) to improve future decisions.

The pirate agent, powered by deep Q-learning, learns by exploring different routes in the maze and updating a neural network to predict the quality of state-action pairs. Unlike a human, the pirate does not "see" the entire maze or reason ahead; instead, it forms a policy based on rewards received from past actions. While a human may succeed in fewer attempts due to intuition, the agent compensates by running thousands of episodes to refine its decisions.

The purpose of the pirate agent is to find an optimal or near-optimal policy that allows it to reach the treasure before the player. It does so using the reinforcement learning cycle:

1. Observing the current environment state using qmaze.observe().
2. Selecting an action using an ε-greedy strategy (if np.random.rand() < epsilon for exploration).
3. Executing the action with qmaze.act(action) to receive the next state and reward.
4. Storing the experience [state, action, reward, next\_state, done] using experience.remember().
5. Updating the model with a training batch from experience.get\_data().

These steps repeat for every episode until the pirate either wins or loses. Each episode's performance is tracked using win\_history, and training can end early if a 100% win rate is reached over a sliding window.

Exploration involves choosing random actions to discover new strategies. Exploitation means using the best-known strategy learned so far. The agent balances these using an ε-greedy approach: with probability ε, it explores; otherwise, it exploits. This is crucial because too much exploitation early on can trap the agent in a suboptimal strategy. Too much exploration late in training prevents it from using what it has learned (Sutton & Barto, 2018).

In this implementation, ε starts high (e.g., 1.0) and decreases over time (e.g., epsilon = 0.05 when win rate > 90%). This shift ensures the agent explores enough in early training, then exploits its learned policy to maximize success.

Reinforcement learning (RL) helps determine the optimal path by enabling the agent to learn from interactions. When the pirate moves toward the treasure and receives a positive reward, the neural network updates to make similar decisions more likely in the future. This process, repeated over thousands of episodes, allows the pirate to discover which sequences of actions (paths) lead to the highest reward.

The function qtrain() includes all steps of the learning cycle. For example:

* Exploration/exploitation: if np.random.rand() < epsilon:
* Action: action = random.choice(...) or np.argmax(model.predict(state))
* Experience storage: experience.remember(episode)
* Training: inputs, targets = experience.get\_data(...) and model.train\_on\_batch(inputs, targets)

Deep Q-learning replaces a traditional Q-table with a neural network to approximate Q-values for state-action pairs. This is necessary due to the size of the state space in an 8x8 maze. The network architecture is built with build\_model(maze), which uses multiple Dense() layers and activation functions like PReLU. The network is trained with the Adam optimizer and MSE loss.

A replay buffer (GameExperience) stores past experiences. This buffer helps decorrelate training data and smooth learning by allowing the model to learn from random batches of previous episodes (Mnih et al., 2015). This approach is more stable than learning from the most recent experience only.

The algorithm proved effective. Over time, the pirate agent improved its pathfinding ability and win rate. When the agent consistently reached the treasure, the model was saved and training stopped early. The performance was evaluated by tracking the win rate using a sliding window (hsize) and checking for 100% win rate with completion\_check(model, qmaze).

Unlike algorithms like A\*, which are deterministic and pre-defined, this learning-based approach is adaptive. If the maze layout changes, the pirate can continue learning and adapt to the new environment, making it suitable for dynamic game worlds.

**References** Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540), 529-533. <https://doi.org/10.1038/nature14236>

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (2nd ed.). MIT Press. <https://www.andrew.cmu.edu/course/10-703/textbook/BartoSutton.pdf>