**Design Defense: Deep Q-Learning Pirate Agent**

**How a person would solve this maze**

A person looks at the grid, spots the treasure in the bottom right, and scans for open corridors from the start. They try a route, notice blockers, back up, and try a different turn. They remember which cells were dead ends and avoid them next time. If there is a clear path, they follow it. If not, they explore until they find one. They use simple rules like stay away from walls, move toward the goal, and do not repeat mistakes.

**How my agent solves the same problem**

My agent does not plan ahead with a full map in mind. It learns by interacting with the maze. At each step it sees the current state, chooses an action, collects a reward, and moves on. It stores these experiences in memory and trains a neural network to estimate the value of taking each action from each state. Early on it explores a lot. As it learns which actions lead to better future rewards, it exploits that knowledge more often. Over time it converges on a short route to the treasure.

Concretely, the agent observes the flattened maze state, picks a legal move using an epsilon greedy rule, applies the move with qmaze.act, stores the transition in replay memory, samples batches from memory, and updates the network so the Q value for the taken action matches the bootstrapped target. I also mask out illegal moves so the agent never tries to walk through walls. I start at the top left when that cell is free to match the grader.

**Similarities and differences**

Both the human and the agent combine exploration and exploitation. Both avoid repeating obvious mistakes once they see them. The difference is the mechanism. The human uses working memory and simple heuristics. The agent uses learned value estimates and a stochastic policy that shifts from exploration to exploitation as training improves. The human can reason abstractly about the layout. The agent only learns from rewards and transitions.

**Why the intelligent agent matters in pathfinding**

The goal is not just to get to the treasure once. The goal is to learn a policy that generalizes from many starts and still finds the treasure quickly. The intelligent agent learns from experience, improves over time, and does not need hand written rules for each maze. That is the point of reinforcement learning in a pathfinding game. You get behavior that adapts as the environment changes and as the agent gains more experience.

**Exploration and exploitation**

Exploration tries unfamiliar actions to gather information. Exploitation chooses the best known action to get rewards now. The ideal mix changes through training. Early on you want heavy exploration because the agent knows almost nothing. Later you want more exploitation so it uses what it learned. I used epsilon greedy. I begin with epsilon near 0.1 as the course starter suggests and lower it to 0.05 once the recent win rate crosses about ninety percent. That keeps a little randomness so the policy does not get stuck but still locks in the good route.

**How reinforcement learning finds the path**

Rewards drive behavior. The environment grants a small penalty for each step and a positive reward on the treasure. That shapes the agent to finish fast and avoid wandering. The Q-network learns to assign higher action values to moves that lead into states that can reach the treasure. Bootstrapping with the Bellman update pushes value back along the path. Experience replay stabilizes learning by mixing old and new transitions, which reduces correlation between samples and makes gradient steps more consistent (Mnih et al., 2015). After enough episodes the best actions line up into a direct route.

**How I implemented deep Q-learning here**

I built a small feedforward network with PReLU activations. The input is just the flattened maze state, and the output layer has four units that match the possible moves: left, up, right, and down. Each output is a Q value that tells the agent how good that move is from the current state.

For training, I followed the notebook’s pseudocode. At the start of each run, I reset the maze, observed the state, checked which actions were legal, and picked a move using epsilon greedy. The agent then stepped into the new state, got its reward, stored the transition in replay memory, and trained the model for one mini-batch. The targets were computed with the standard one-step rule. If the next state was terminal, the target was just the reward. Otherwise, I added the discounted maximum next Q value. This cycle repeated until the episode ended.

To keep the agent from wasting moves, I masked out any illegal actions before applying argmax. I also added the instructor’s hint to start in the top-left cell when it’s open. Progress was printed every epoch with loss, episodes, wins, and recent win rate. If the recent history showed perfect wins and the model passed the completion check, training stopped early.

For hyperparameters, I used a replay buffer of around one thousand transitions and a batch size of fifty, which matches the starter setup. The discount factor stayed at the template default. Epsilon began at the default and dropped to five percent once the win rate crossed ninety percent. When faster convergence was needed, I raised the epoch count to five hundred or one thousand, which usually pushed the win rate to one hundred percent in this small maze.

**What worked and what I would improve**

Valid action masking removed a lot of wasted steps. The fixed start made grading consistent. Replay memory made training stable. The main lever for results is the number of epochs. More epochs means more wins and a shorter route. If I had more time, I would try a target network to reduce moving target noise, anneal epsilon smoothly over time, and reward shaping that gives a tiny bonus for moves that reduce Manhattan distance to the treasure. Those changes often speed up convergence without changing the final policy.

**Takeaway**

Humans solve mazes with vision, memory, and simple rules. The agent solves the same problem with learned action values and a balance of exploration and exploitation. Deep Q-learning fits this game because the action space is small and the reward is clear. With replay and a small neural network, the pirate learns a reliable path to the treasure and reaches the goal fast.

**References**

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., et al. 2015. Human level control through deep reinforcement learning. Nature, 518, 529–533. https://doi.org/10.1038/nature14236

Sutton, R. S., and Barto, A. G. 2018. Reinforcement Learning. An Introduction. MIT Press.

Suleiman, A. 2018. An introduction to Q-learning. Reinforcement learning. freeCodeCamp. <https://medium.com/free-code-camp/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc>

TreasureMaze.py and GameExperience.py provided in course starter code. CS 370.