Craig O'Loughlin 12/11/2022 CS370

**Project Two: Pathfinding DQN Design Defense**

In this project we have implemented a machine learning algorithm that solves a pathfinding problem, specifically, an implementation of an AI agent that can find its way through a grid maze consisting of a start position, goal position, open tiles, and blocked tiles. The agent we created is initially a blank slate, having never before seen the maze (or any maze). Further, it knows nothing except its own current position in the maze and the available directions it may travel (up, down, left, right).

Before considering how we designed our computer agent to learn to find its way out of this predicament, we could briefly consider how a human may do so. This can give us some insight into the application. Given a new maze, a human would likely begin by analyzing in its entirety, from the ‘top-level’. For a simple maze, we can begin mentally tracing out some paths and seeing how close they get us to the goal. For a complex maze, however, we may instead need to try some paths blindly and see where they go. Better, longer paths will then get investigated more deeply, and continually refined until a solution is reached.

A computer agent can take the same approach; in a basic sense, trying paths until good ones are found, and refining towards the best solution. The computer agent we created in this project does just that. The exact implementation we used is a DQN (Deep Q-Network), a model which has had great success in other applications (Lamba, 2018). A DQN is set to randomly try some paths and store the results of said paths at each step in a Q table of quality-per-action values. It can then continue to iterate on the most successful paths, learning better actions in order to optimize the paths towards a solution. A CNN (Convolutional Neural Network) is used in order to analyze the steps after each maze solve attempt and learn to recognize certain states and like states. Unlike a human, this ML (machine learning) approach will attempt to always find the best solution, that is, the shortest. Also, at no time does this algorithm step back and look at the problem as a whole, as a human may, theorizing what the best line may be (no ‘top-level’ analysis). I would argue that the DQN solution is fairly simple compared to other ML approaches yet is still effective and straightforward at finding the best solution in this case.

One of the major considerations for such a ML network is how it goes about pathfinding. How much time should the network spend considering new paths, and how much time should instead be spent refining the ones it knows? In terminology, the behavior of finding new paths is known as ‘exploration’ and the behavior of refining known paths is know as ‘exploitation’. If the network only ever exploits, it can surely learn a solution quickly. However, the solution may not be optimal, as it has not explored all but one path. On the other hand, if the network were to only ever explore, it is not guaranteed that any solution will ever be reached.

Our algorithm uses, as is standard for this type of application, a modifier value known as ‘epsilon’ to regulate time spent in either mode. A random number generator is then queried for each step in the maze; if the number falls below epsilon, exploration is used, and above, exploitation. Using this epsilon value, we can easily control exploration and exploitation behavior. We experimented with both high and low values of epsilon. In general, low values (more exploitation) will work better for learning, as the network must leverage what it knows often in order to continue learning. However, we found the best results by applying an Epsilon Greedy approach (PyLessons, 2019). In this approach, we start with a high epsilon and mostly exploration behavior, as the network should learn as much as possible about the maze environment first (assume it has no knowledge to exploit). With each run, we slowly decrease epsilon, introducing more exploitation as the knowledge the network has about the maze has increased in quality. Eventually, the network is almost completely using exploitation.

Overall, our algorithm first sets the initial epsilon to 0.9. Then, for each epoch, a game is played. An action is selected either randomly or predicted by a neural network using the epsilon value. After each action is performed, the state of the maze and the reward for the action are put into a memory bank. The win condition is checked, and if the maze is not solved, a random set of action/reward memory conditions are chosen to train the neural network and update the Q-table accordingly. When the maze is solved or the move limit is reached, epsilon is decayed slightly and the process continues, until a win rate of 100% is reached.

After each action in the maze, as discussed, a neural network is used to analyze the step taken and how good or bad it was (known as ‘reward’, a positive/negative modifier that helps the agent learn what is working and what is not). So, the algorithm spends this time learning from the moves it makes and updating its Q-table with more information to exploit, a process known as experience replay (Torres, 2020). We found doing a round of training using 10 maze states from memory, by default, worked well; and there was no noticeable improvement from using 20 or 50 states at a time.

It was noticed that the algorithm can spend a somewhat inordinate time repeating bad moves until a better move in a certain state is learned. This is a result of the algorithm optimizing itself in small steps, coupled with the random action memory selection outlined above. Small steps, however, are actually desired, since they help ensure the algorithm does not overtune itself (in other words, jumping straight to conclusions that may be only partially correct). When we tried increasing the rate that algorithm learns by increasing the reward values, the algorithm actually performed worse, never reaching a 100% win rate.

The algorithm’s performance was measured by running it from a blank slate on a sample maze. It was set to repeatedly play games on this maze until it could reach a 100% win rate. The remarkable thing about this, and machine learning in general, is that this code was able to find its own set of solutions automatically, without any programmer having to hard code the solution for it.

This algorithm reached a 100% win rate (defined as winning 32 games in a row) after 957 games, winning a total of 590 along the way. The time taken for this was 16.48 minutes as run on a lab-provided virtual machine with unknown specifications. After this, performance was checked by playing a sample game with a top-left-corner start position, and bottom-right-corner end position. Here is the resulting path the algorithm supplied:

A black and white checkered square

Description automatically generated with low confidence

Given that a machine can learn its own solution to a pathfinding issue like this without any intervention from the programmer, other than setting the starting rules and reward states, is a marvel of modern computing in a field that continues to grow and solve complex modern computational problems. The algorithm that we created performs our task well and has accomplished the goal of finding the optimal solution to a grid maze.

**References**

Lamba, A. (2018, Sep 3). *An introduction to q-learning: reinforcement learning.* Medium. <https://medium.com/free-code-camp/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc>

PyLessons. (2019, Nov 3). *Epsilon greedy in deep q learning.* <https://pylessons.com/Epsilon-Greedy-DQN>

Torres, J. (2020, Aug 15). *Deep q-network: Experience replay and target networks.* Towards Data Science. <https://towardsdatascience.com/deep-q-network-dqn-ii-b6bf911b6b2c>