CS 175: Minecraft Reinforcement Learning Agent

Learning to Navigate Caves

Allen Arani

Min Seop Lee

Seho Park

Seoyoung Ki

1. Project Summary

In the video game Minecraft, players increasingly grow in their strength through the quality of the materials they harvest, growing more powerful until ready to beat the final boss. The most valuable resources in the game lie in caves; the deeper a player goes, the more likely they are to find ultra-valuable ores, like diamonds, that can be crafted into some of the strongest gear possible. However, in this game, these valuables come at the risk of death, where the player loses everything they have. This can be a result of a number of factors, but one thing is certain: death in Minecraft should be avoided when possible. In this project, we aim to train a reinforcement learning agent to navigate common cave obstacles with the objective of being as quick as possible, as the slower you are in Minecraft, the more likely you are to be killed by enemy mobs.

To achieve this, we trained an agent to reach a goal point in minimum steps on a Minecraft Malmo generated cave using deep reinforcement learning. This cave served to replicate the typical obstacles one might find underground in Minecraft, such as slowing cobwebs and deadly lava. Unlike typical tabular methods, we utilized a neural network-based policy to dynamically navigate a continuous 3D state space. In addition, rather than simply relying on forward, goal-based rewards, we optimized the reward structure to directly incentivize minimum-step path exploration: accounting for factors such as time, hazardous contacts, exploration and reaching the goal.

In addition, we developed a visualization module that allows one to see the policy changes intuitively as the agent learns. By visually analyzing the Q-value distribution converging over time, its contrast from the early randomized behavioral distribution grows, and we can immediately understand and adjust the impact of reward design or hyperparameter settings.

This challenge is not trivial; it requires artificial intelligence techniques to apply reinforcement learning algorithms to real-world 3D simulation environments, and involves a variety of challenges, including reward design, state and behavior design,

and ensuring learning stability. Given the unpredictability of Minecraft, we find this problem necessitates AI as its primary approach.

2. Approaches

Our project started with an initial application of tabular Q-learning using *Formula 1* as a baseline.

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma a' maxQ(s', a') - Q(s, a)]$$

Formula 1: Tabular Q-Learning Update Function

In this formula,

- Q(s, a) is the value of taking action a in state s,
- α is the learning rate,
- γ is the discount factor,
- r is the immediate reward, and
- $a' maxQ(s', a')$ is the maximum value of Q among the possible actions in the next state s'.

We attempted training the agent by directly updating the Q-table, but we found that memory and generalization problems occurred as the number of states increased. In order to overcome this problem, we adopted the *Deep Q-Network* (DQN) method, a deep learning based Q-value approximation technique that uses *Formula 2* to train the neural network Q:

$$L(\theta) = E_{(s, a, r, s') \sim D}[(r + \gamma a' maxQ_{\theta-} - (s', a') - Q_{\theta}(s, a))^2]$$

Formula 2: Mean-Squared Error loss function used by DQN

In this formula,

- θ represents the network parameters being trained,
- $\theta-$ is the parameter of the target network copied at regular intervals,
- D is the set of transitions sampled from a *replay buffer*, a queue containing past runs, and

- The network $Q_{\theta}(s, a)$, trained with least squares error (LSE) to the target value $r + \gamma a' max Q_{\theta-} - (s', a')$.

We designed a neural network consisting of two hidden layers, containing 64 units each with ReLU, to output a Q-value for each action. With this, we pass a three-dimensional input vector of the agent's location information, normalized to the range [0, 1].

In order to ensure learning stability, we leveraged the replay buffer to randomly sample past transition experiences. As we pursued an exploration strategy, we adopted an ε-greedy approach with ε decay, with ε starting at 1.0 and decreasing by a decay factor of 0.995 for every episode, eventually reducing to a minimum of ε = 0.05. For the remaining hyperparameters, we used a learning rate (α) of 0.001, a discount factor (γ) of 0.9, a batch size of 64, and a buffer capacity of 10000. Under these configurations, we updated network parameters at each step by minimizing the loss function.

Also, as mentioned in the Project Summary section, we designed the reward function to prioritize speed, encouraging the agent to learn the minimum step path, which in turn incentivized the agent to reinforce policies that get it to the goal quickly and safely. In particular, we used reward scheme Formula 3 to guide the agent:

$$reward = 150 * goal - 3 * visited - 10 * hazard - steps$$

Formula 3: Reward Function

In this formula,

- *goal* is a binary variable representing whether the agent reached a goal state (Z coordinate > 127),
- *visited* represents the number of steps which the agent spent in an already explored state,
- *hazard* is the number of times the agent touched a hazard, namely, a cobweb or water, in a training run, and
- *steps* is the number of steps elapsed since beginning training, representing time.

Through this set of techniques, the DQN converged reliably on large state spaces that tabular methods struggled to handle, and showed progressive improvements in reward and success rate as training progressed.

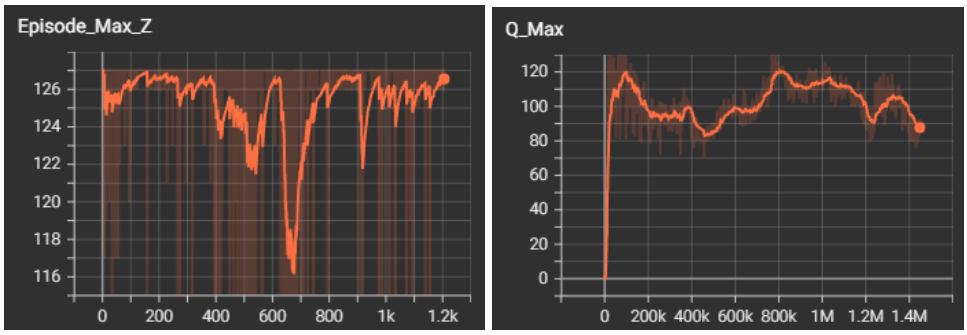
As a result, we were able to conclude that Tabular Q-learning is simple to implement and has convergence guarantees for small Markov Decision Processes, but its memory and generalization limitations become apparent as the state space grows. DQN, on the other hand, can generalize continuous and large-scale states by approximating them with neural networks, but suffers from hyperparameter sensitivity, decreased sample efficiency, and unstable initial learning. Based on these comparisons, along with our results, our project demonstrated that DQNs can reliably solve this large-scale 3D navigation problem.

Before we settled on our final version of the project, we tried and tested a few different maps. One primary benchmark map was a glass maze. The agent was trained on the similar DQN neural network (with minor variation in hyperparameters) as described above, but the outcome was noticebly different. On this map, the first 100 episodes showed promise with steadily increasing reward, but then it got stuck at a low negative value which reflects no successful episodes. According to Tensorboard, over 13000 episodes, the agent only reached the goal once, around 6000 episodes in. Also, the rewards per step was non-increasing, and the agent's z coordinate value (representing forward progress) failed to surpass our goal states, indicating that behaviour was not improving. Since we were still far from the result that we wanted, we changed the map to a wider corridor so the agent would have more room to side step obstacles. Alongside this, we reshaped the reward scheme to incentivize progress and accelerate learning. We also made a few more technical changes such as trimming the agent's action choice from (south, east, west, north) to (south, east, west), preventing backtracking, and declaring our ε-greedy scheme outside the episode loop to ensure proper decay throughout training.

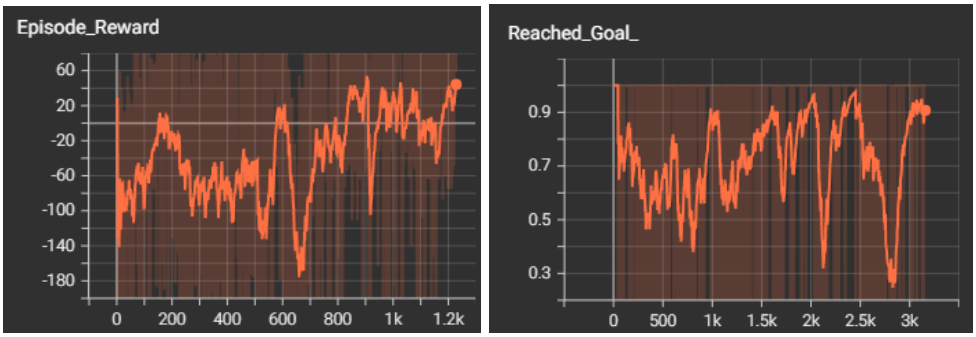
3. Evaluations

Many factors played a role in determining the success of our agent. Qualitatively, the agent must reach the end of the cave, but quantitatively, it must do so quickly. As

such, we found ourselves relying on several varying metrics to determine how well the DQN model is fitting to our environment.



***Figures 1 and 2*: Maximum Z-coordinate attained (left) and Maximum Q-value on each step (right)**



***Figures 3 and 4*: Episode Reward (left) and whether goal was reached (right, repolled within episodes)**

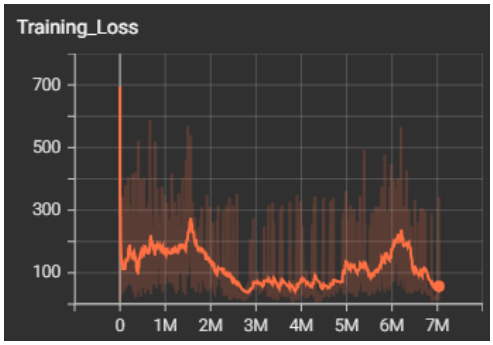


***Figure 5*: Training Loss per step**

Using the popular visualization software TensorBoard, in combination with our custom Q-value visualization module, we were able to retrieve quick, live reports of our agent's progress over long training periods. In each of the above figures, a smoothing exponential moving average was used to determine the underlying trends of our data,

as the original charts are too noisy to interpret (note the blurs of orange behind figures 1, 3, and 4; this is the unsmoothed data).

In the presented figures, we see that in over 1200 episodes, the agent was able to very consistently reach the goal states (Figure 4). Data post-processing shows an 88.6% completion rate of the obstacle course, indicating promise in the agent's ability to achieve the first objective: successfully navigating the cave. With this, we can observe the agent's reward in each episode (Figure 3); analysis shows that it struggled early on but eventually began receiving higher rewards. Given the reward function's (Formula 3) emphasis on reaching the goal as timely as possible by avoiding setbacks, and noting that the agent can very easily still receive harsh penalties despite reaching the goal (due to the visited + hazard parameters, both of which indicating drastic slow-downs), we see the agent learned to complete the cave-simulating course while avoiding obstacles, thus reaching the goal in faster time. With this, we can conclude the reward function is indicative of the timeliness of the agent, and therefore, the agent learned to navigate the environment faster as time progressed. Thus, both goals were achieved using a DQN.

In the given map the agent is trained on, the maximum Q value that can be obtained (that is, global optimum) is 120, calculated by 150*1 - 3*0 - 10*0 - 30, given that the agent never revisited past states, never stepped on hazardous blocks, and took a minimal 30 steps. The Q-Max graph (Figure 2) shows that the agent reached, and even surpassed, the maximum Q value of 120 several times. Such a global optimum is hard, since it is the best the agent can perform in the given world (even human players cannot do better than this policy). Also, the agent reached local optima (between the Q value of 110 and 120) several times between the 800k-th step and the 1.2M-th step. This shows that the agent does not settle for the global optimum once reached, and instead explores for different policies, indicating prevention of the model falling into local optima and avoiding overfitting.

The loss starts around 700 and drops to around 100 after a few episodes. Then, the agent keeps exploring different policies to seek the better and ultimately the best policy. The loss value is high in the beginning because the epsilon value at this stage is large, causing the agent to randomly explore new policies, even if it had previously

reached a state of lower loss. Then, it starts to decrease around the 1.5M-th step region and gets lower than the initial loss (which is 100) as it finds more and more optimal policies with decreasing epsilon value. Then, we see the loss converge between the 3M-th and 5M-th step range, due to a lower epsilon value than the beginning. The loss then increases again around 5M-th step, as there's still a non-zero epsilon value at this point requiring the agent to try other policies. It eventually decreases back to around 6M-th episode, and reaches the lowest loss value of about 50 at 7M-th step. This shows that the model does not simply overfit as it reaches the lowest loss rate, but instead explores to see if improvements could be discovered.

4. References and Resources Used

For implementing Q-learning neural network class (that is, DQN), we used *PyTorch*, which allowed us to handily experiment and tune the various hyperparameters to build the best model for this project.

For visualization of the agent's learning status and progress, we used tensorboardX. This allowed us to inspect the episode reward, goal reaching rates, maximum Q value, maximum z-coordinate reached, and training loss, which tremendously helped us to determine whether the learning is appropriately being done throughout the huge amount of episodes.

We used the Project Malmo environment for running the agent. This provided us with flexibility in using Python to manipulate the agent's behavior, programmatically supply in-game commands, and in conjunction with XML, develop the worlds and environments to train the agent on. Thanks to Malmo's functionality, we were able to easily generate many variations of maps, even altering them at runtime using Python string formatting to create beautiful dynamic environments and provide adequate challenge.

For the library version control, we used Malmo version 0.36.0.0, tensorboardX version 2.6, and Torch version 1.10.2+cpu, with Python downgraded to 3.6.15 and Numpy downgraded to 1.18.5 in order to fit the version compatibility with the Malmo environment.

The version of Minecraft which the experimentation was performed on was 1.11.2, released in late 2016.

Lastly, for documentation, the XML syntax for our map creation in the Malmo environment was based on the Project Malmo XML Schema Documentation, open-sourced from Microsoft.