Using Machine Learning to Train an Agent to Terminate Mobs in Minecraft

Farhib S.
Georgia Institute of Technology
Atlanta, GA 30332

Rishav B.
Georgia Institute of Technology
Atlanta, GA 30332

Dominik K.
Georgia Institute of Technology
Atlanta, GA 30332

Jean M.
Georgia Institute of Technology
Atlanta, GA 30332

Saagar M.
Georgia Institute of Technology
Atlanta, GA 30332

Abstract

As robots become more prevalent in society and the workplace, the need for robust algorithms that can learn to control autonomous agents in a wide range of situations becomes paramount. Prior work has shown that deep reinforcement learning models perform reliably in 3D environments, even when rewards are sparse and only visual input is provided to the agent. Here, we use Project Malmö, an AI research platform based on the popular sandbox video game Minecraft, to train an RL agent to combat in-game entities known as "mobs." We implement a RL algorithm called a Deep Q Network (DQN), as well as a pretrained residual neural network as a baseline model, and compare the differences in performance. We expect that with minimal hyperparameter tuning, the RL model will learn significantly more than the baseline, and that the agent will succeed in defending itself to some extent.

Introduction and Background

Minecraft is a popular sandbox video game that contains a number of hostile non-player entities known as "mobs"; these entities are meant to attack and kill the player character. Our agent will have to learn strategies to deal with each type of hostile mob with the goal of defeating as many mobs and surviving as long as possible. Additionally, the environment in a Minecraft "world" can be randomly generated using an algorithm or built by the player. To create a closed environment for our agent to learn and fight against these mobs, we will be using Microsoft's Project Malmo. Using machine learning in minecraft is the focus of a large competition called MineRL, which provides rigorous guidelines towards achieving an agent that can operate autonomously in minecraft. It is our hope that methods like the ones we are using to train an agent in a simulated environment can be extrapolated to real life applications like robotics in the physical world. Since minecraft as an environment is completely customizable, it makes it ideal for entry level testing of potential real world use cases.

Problem Definition

The agent will have to last as long as possible while defeating as many distinct hostile entities as possible and navigating the environment. The agent will receive positive rewards for defeating

entities/surviving and negative rewards for being defeated and losing health itself. We are utilizing a fairly dense reward structure, with the hope that this will enable the agent to learn good behaviors more reliably. Since we are rewarding the agent for successful hits on mobs and survival, and are negatively rewarding it for taking damage and dying, we can see our reward structure is dense. Additionally, to increase the chance of the agent learning the reward for attacking mobs, we let the agent continually attack, so it has to learn to face the mobs, rather than face them and then attack. Below are listed the present actions and rewards we used to train our preliminary RL model:

- Action Space: Move Forward, Move Backward, Turn Left, Turn Right, Do Nothing
- Rewards: Rewards: Death (-100), Damage Taken (-4), Damaging Zombie (15), Per Action Taken (0.05), Zombie Killed (150)

Data Collection

Since we are using Deep Q Learning, we did not have to collect any data. The agent's observations in the environment was our "data," on which the neural network trained on. The agents observations in the environment were 640x480, which we rescaled to an 84x84 image.

Definitions

Step: A step is every iteration in an episode. Each step, the agent makes an observation, takes an action, and learns from previous memories.
Episode: Each run of the game in which the agent plays (until it dies) is called an episode. Reward: The agent receives a positive reward for being in a good state and taking an optimal action like hitting a zombie. It receives a negative reward for things like getting hit.
Q-value: The Q value is essentially a numeric value assigned to a state action pair determining how "good" that action is given the current state.
DQN: A neural network that we are using to approximate the Q-value of a state action pair. Target Network: It is a copy of the DQN, but is only updated periodically. This is used to increase the stability of the algorithm.
ResNet50: A large image recognition CNN.

Methods

We used a Convolutional Deep Q Network to take in the image input and output what action(s) to take. One of the ways that Project Malmo allowed our agent to "see" in the Minecraft world was through images, so using a convolutional neural network made logical sense. Similar to most CNNs, we started with the CNN workflow (Convolution, Max Pooling, Activation) and then used some fully connected layers. We also used a replay buffer to allow the agent to have "memory," giving the agent a way to utilize past trials. Another implementation detail is that we used a target network that we copied the weights to periodically every (300 steps) so that our DQN converged to a more stable solution. As prior research had shown us, using a recurrent neural network would not give us significant improvements so this is not a path we decided to follow [2].
With those implementation details, we followed the regular Q-learning algorithm, which is as follows:

1. Get state of the environment
2. Take action using an epsilon greedy policy
3. Create SARS tuple (state, action, reward, best state) and store it into replay buffer
4. Sample from the replay buffer and use the sample to update the weights of the DQN.
5. Update target network (only do this once every 300 episodes).

As a baseline model we took the feature representation from a large pre-trained CNN such as ResNet50, by using the model and excluding the final dense layer, and using this in place of our convolution layers. We had predicted that this would likely get us some performance, but would inherently be worse, since we had fixed some of our trainable parameters.
Using a DQN on Minecraft is not a very novel method as Minecraft is a fully-observable, deterministic environment, which is well suited for reinforcement learning.

Metrics

While we do have a loss function we can plot to track learning, for our reinforcement learning problem, tracking the metric of total reward per episode is a better measure of our progress. This metric is essentially how well the agent played in its environment during that specific episode. We do not really have data to split for K-fold validation, but we have trained entirely new models for different configurations of our hyperparameters and evaluate them based on the total reward per episode metric. Each of these models are trained on new "data" since each run of each episode will be different. The loss function we used for the DQN is MSE (mean-squared error) since our Q-value function is a continuous function and approximating the Q-value function for a continuous input state is a regression problem. Applying gradient descent or another optimization method to minimize this loss function allowed our network to learn the Q-value function.

Initial Results

The results of our training did show learning within our reward scheme, but that reward scheme was not optimal for what we wanted our agent to learn. With the large negative reward of -1000 for dying and the maximum number of steps set to 50, the agents reward was usually either 50 (since it survived the entire episode) or something less than -950. Because of this, any other rewards that could have been explored would be overlooked since the negative reward for dying had such a large magnitude. We can see this in the graph below, in which the rewards are sporadic and shifting between around 0 and -1000. Although as time passes, we see that the density of rewards that were around the 0 mark increases while the opposite occurred for the -1000 rewards. The Savitzky-Golay smoothing filter visualizes this nicely. While these results were less than ideal, we were still able to get something running and learning in the minecraft environment, which was our main goal for this touchpoint.

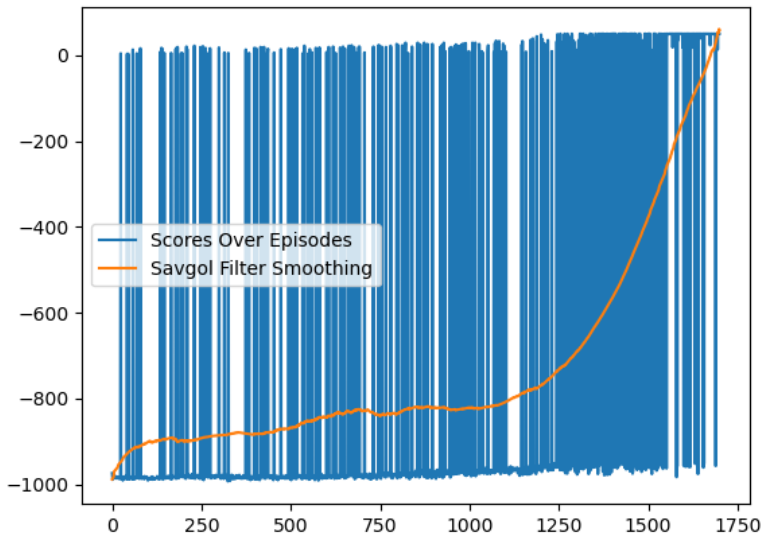

Figure 1: Initial agent rewards while training our DQN

While neither of the networks performed optimally, the CNN did perform better than ResNet50 as expected.

Final Results

As we had expected, the baseline performed worse for a couple of reasons. Because the feature representation of ResNet50 is so large, this model ended up having more parameters to train even

though the dense layers that we added were of the same size. This made it a lot more challenging to train: the main reason the baseline did not perform as well as the CNN was because the overhead required for a backward pass through the larger dense layers caused the agent to take actions at a much slower rate. Because of this, instead of taking actions a few times each second it took a few seconds per action. This knocked performance down by a lot since the chances of the agent actually hitting the zombies decreased by a large factor. This makes killing zombies a lot more challenging since the agent must line its sword as the agent takes the attack action (which now happens less frequently). Although there are many hindering factors, the baseline agent still was able to show some initial learning, but the overall rewards were significantly lower than the regular agent and the algorithm did not converge. Below is the graph for baseline training.

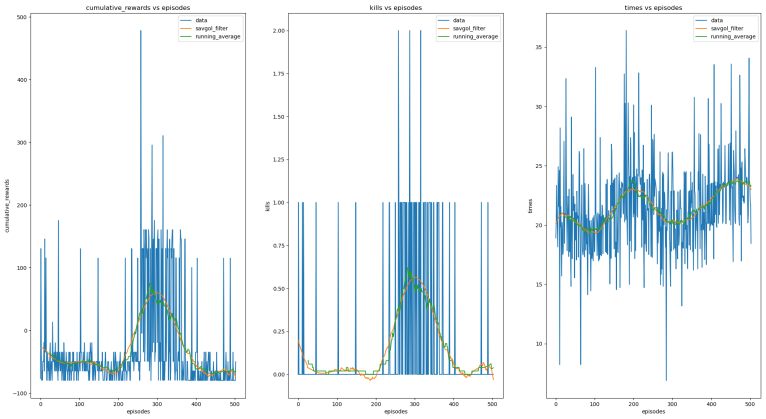

Figure 2: Reward, Kills, and Time Survived per Episode for ResNet50

As for the CNN model, we were able to show learning by the agent as shown in the graphs below.

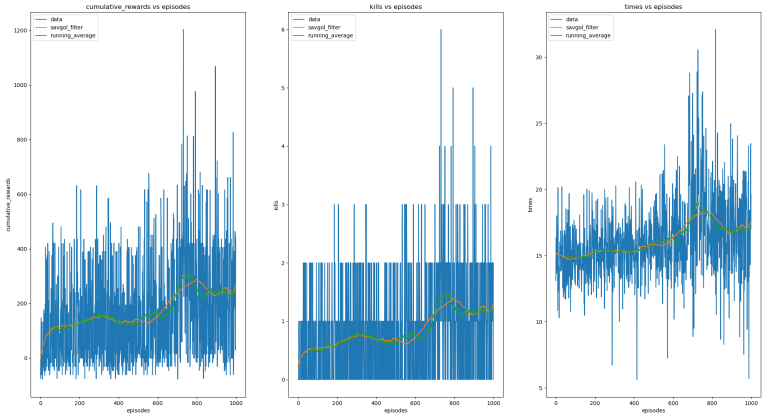

Figure 3: Reward, Kills, and Time Survived per Episode for DQN

However, the strategies of the agent were not optimal. The main strategy the agent followed was to put itself into a corner and then spin or just face the zombies. While this does allow the agent to hit the zombies, it also allows the zombies to hit the agent, which is suboptimal. One of the better strategies the agent learned was that if the zombies' locations were unknown but the agent was getting

hit, it would face a wall and then back up, which usually put the zombies back in view as they would follow. This also allowed the agent to get a few hits on the zombie. The baseline agent ended up learning similar strategies but implemented them worse because of the hindering factors previously mentioned.

Figure 4: A Screen Capture of the Agent Learning to Back Away

Overall, the performance of the agent was okay, but it had room for improvement. Unfortunately, we did not get enough time to tune the hyperparameters and reward scheme as much as we had liked, due to limitations in time and compute. Due to the fact that we were training this on a laptop, each train of the agent took around 6-10 hours. So, we were only able to train a few different versions of the model, our best of which ended being version 3. Below is one of the earlier versions of the model, which did not learn very well since the agent was too scared to try anything since the penalty for getting hit was too high.

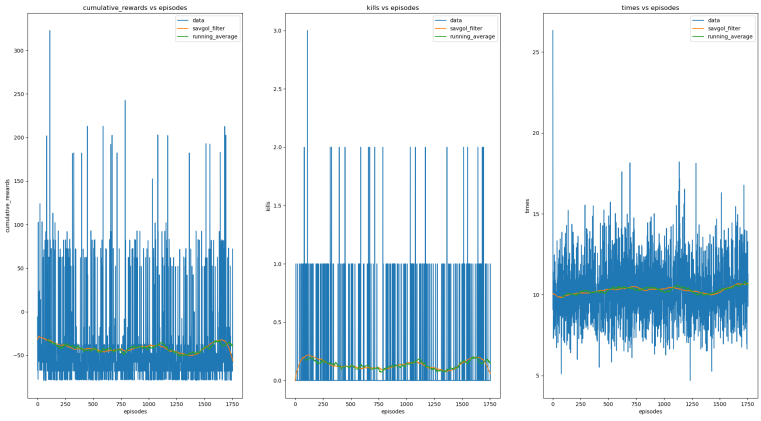

Figure 5: Reward, Kills, and Time Survived per Episode for DQN with Poor Hyperparameters

Conclusion

In this project we learned more about Deep Q networks and practical convolutional neural networks work and utilized them. To improve upon our research we would add things like current health, zombie location, and ambient noises to the observation space of the agent in order to alleviate the problem of the agent taking nearly random actions when zombies are not on the screen. These are values that we can usually deduce while playing but the agent had no way of directly seeing these values. Given more time we could further tune hyperparameters, and rewards, hopefully letting the agent learn optimal behavior more reliably and quickly. Also we would have used an auto encoder instead of ResNet50, as we could have avoided many of the problems with processing time taking too long with that approach.

Ethics Statement

The societal impacts of this project are difficult to determine due to the fact that the agent works with a videogame. The most direct impact of this project would be to improve video game artificial intelligence, thus improving the experience of playing video games. However, if we were to consider our project as an agent performing actions in response to visual stimuli, we can expand the potential societal impacts to a much wider set of applications, especially in robotics.

Because of the wide range of applications robotics brings, the societal benefits and harms are equally numerous. Ensuring that the benefits sufficiently outweigh the harms comes down to two main things: the accuracy of the model itself and the quality of the data. The model presented here had some accuracy issues that could result in significant problems if applied to crucial services such as surgery or medical care in general. The solution to this issue would be to tune hyperparameters until the model performed acceptably.

Quality of data is arguably even more important than the model itself; it determines the biases and features the model will focus on. Special thought needs to be taken to ensure quality of input data the model will train on to ensure undue. If our agent was applied to trying to identify human faces, for example, this would take the form of diversity in the dataset to allow for better identification of different races. Ignoring these consideration could potentially result in significant societal harm.

References

[1] Christian S., Yanick S. & Manfred V. (2020). Sample Efficient Reinforcement Learning through Learning from Demonstrations in Minecraft. arXiv. Retrieved March 1, 2021, from https://arxiv.org/abs/2003.06066

[2] Clément R. & Vincent B. (2019). Deep Recurrent Q-Learning vs Deep Q-Learning on a simple Partially Observable Markov Decision Process with Minecraft. arXiv. Retrieved March 1, 2021, from https://arxiv.org/abs/1903.04311

[3] Volodymyr M., Koray, K., David, S., Alex, G., Ioannis A., Daan W. & Martin R. (2013). Playing Atari with Deep Reinforcement Learning. arXiv. Retrieved March 1, 2021, from https://arxiv.org/abs/1312.5602

Github Pages Link: https://minerl.bhagat.io/