Killing Zombies in Minecraft With Deep Reinforcement Learning

Hiroto Udagawa, Tarun Narasimhan, Shim-Young Lee

December 16, 2016

1 Introduction

In this project, we train an agent in the video game Minecraft to kill as many zombies and surivive as long as possible using reinfocement learning. Our agent is only equipped with a sword to attack the zombies, which can themselves attack back and kill the agent. The objective is for the agent to learn what policies to execute based on the visual information it receives about the surroundings from the raw pixels of the game play screen and the reward structure we have designed.

Ultimately, using a convolutional neural network and Q-function reinforcement learning, the agent showed definite improvement in its ability to kill zombies. This project displays the promise of combining deep learning with reinforcement learning to achieve non-trivial tasks.

2 Related Work

There have been several research projects in this area which we drew upon for our project. The major inspiration for our work was DeepMind's 2013 paper on deep reinforcement learning, in which an agent was trained with a single model architecture (Deep Q Learning) that uses a convolutional neural network to perform a variant of Q-learning. It successfully learned to play 7 different Atari 2600 games, in a few of which the agent outperformed human experts [1]. Instead of solving the standard action-value function (Q function) with Bellman equations, deep convolutional neural network was used as a nonlinear function approximator. DeepMind also employed a biology inspired technique known as experience replay, which attempts to fix the divergence due to correlation between sequential inputs. While a different network was trained for each game, the same architecture and hyperparameters were used across all of the games. Furthermore, no prior (or game specific) knowledge was used; only the raw pixel values were fed to the network as inputs. DeepMind's major achievement was using extremely high-dimensional states (the visual input from the agent) and no information about the game rules to train an algorithm which could outperform skilled human players, and we aimed to replicate this in our project.

For the actual implementation of our project, we built on the work of Chen (2015), which also relied on DeepMind's DQN and applied the same model architecture and algorithm to the game Flappy Bird [3]. We sought to expand on his work by applying DeepMind's architecture to a more complicated setting. While Flappy Bird has only two possible actions and no opponents, Minecraft has a continuous action space and our setting includes zombies actively trying to kill the agent.

3 Minecraft and Project Malmo

For the setting of our project, we chose Minecraft, a popular sandbox video game that was released in 2011. It allows players to control an agent, through which they can explore the map, build structures, craft tools, and destroy computer-generated zombies. Project Malmo is an open source platform for Minecraft created by Microsoft that offers an interface for researchers to experiment with different approaches to artificial intelligence by controlling an agent through Python scripts.

Malmo is a useful environment for artificial intelligence research because it allows users to test endless situations within a relatively simple framework by controlling agents' movements and environments via code. By combining different states, actions, and spaces, researchers can create different environments and define various tasks for which the agent trains with reinforcement learning.

Microsoft designed the agents who are deployed in Malmo to be a separate entity from the Minecraft world that they reside in. This means that the agent cannot manipulate the world, and the only information our algorithm can utilize are observations that the agent directly experiences from its direct environment. This makes Malmo a difficult but interesting platform to build upon. Minecraft is designed to be a rough representation of the real world, and the agents in Malmo are rough representations of human beings. Thus our agent cannot stop time, while it calculates it's next action, and instead it has to train a neural network and quickly decide it's next action in real time. Furthermore, the agent is trained on its first-person visual input, which means in order to know what's behind, it must turn around. This contrasts many of the Atari games tested by DeepMind or in Flappy Bird, where the game provides a bird's-eye view, which gives much more information than the agent's first-person view in Minecraft.

4 Method

4.1 Scenario

The agent begins in the middle of a completely flat room with a fixed number of randomly generated zombies. The room has uniform color and lighting to reduce the complexity of the environment. The zombies automatically approach the agent and damage it if they get too close. The agent will be equipped with a sword to fend off the zombies. As the agent destroys zombies, new zombies continue to spawn (so that there is always a constant number of zombies), and the game

continues to run until the character loses all of its health. The score is calculated based on the total number of zombies that the character kills as well as the amount of time the agent survived. After each death, the scenario resets to the same initial state.

Figure 1: Minecraft screenshot

4.2 MDP Formulation

We formalize the reinforcement learning task as a Markov Decision Process. The agent interacts with an environment by making observations and actions, and receiving rewards. At each iteration, the agent selects an action from the action space, $A = 1, \ldots, K$. This action then changes the state the agent is in and the agent receives rewards based on the new state it enters. The reward often depends not only on the new state it just entered, but the entire sequence of actions and observations it made until it entered the current state. Thus, to make an appropriate action in the current state, the agent needs information from previous actions and observations as well as the current observation. We therefore consider sequences of actions and observations $s_t = x_1, a_1, x_2, ..., a_{t-1}, x_t$, where x_t is the pixel values of the visual input from the agent at time t. All such sequences are finite; thus we now have finite Markov decision process (MDP), where we use the sequence s_t as a distinct state at each time-step t.

4.2.1 Actions and States

The actions for this problem are the discrete movements of the agent (the agent can move forward, back, turn left, turn right, or remain stationary) as well as the ability to attack with a sword. To simplify the action space, the attack command was turned on the entire time, and the agent only chooses how to move at each time-step. Minecraft allows the agent to move and turn with different speed, but we only use five possible actions with fixed moving and turning speed. The agent will register a hit with its sword if there is a zombie close enough and in the center portion of the screen.

The observation at time-step t, x_t, is an image of the player's vision represented by the pixels values. Project Malmo provides the functionality to retrieve the image from the screen. As stated earlier, the state at time-step t is a sequence of observations and actions made until and at time t.

4.2.2 Reward

We assign the algorithm the following reward structure:

- +5 for a successful hit on a zombie
- +40 for a successful kill
- -5 for getting hit by a zombie
- +0.03 for every frame in which the agent stays alive

The last reward is designed to train the agent's behavior to incentivize staying alive; holding the kill count equal, we wanted the agent to have a higher reward for surviving longer.

4.3 Q-learning and Deep Q Network

4.3.1 Reinforcement Learning Background

In reinforcement learning, the agent is trained to maximize aggregate future rewards. With the discount factor of γ, the future discounted return at time t is $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$, where T is the number of time-steps (or iterations) in that episode of the game. We define the optimal action-value function $Q^*(s, a) = \max_\pi \mathbb{E}[R_t | s_t = s, a_t = a, \pi]$, which is the maximum expected return achievable by following any strategy, after seeing some sequence s and then taking some action a, and where π is a policy mapping sequences to actions (or distributions over actions).

In simpler settings, the optimal policy is obtained by solving the Bellman equation iteratively, using the equation $Q^*_{i+1}(s, a) = \mathbb{E}[r + \max_{a'} Q^*_i(s', a') | s, a]$, where i is the current iteration. However, maintaining estimates of action-value function for all sequences (states) is impossible and yields no generalization for newly encountered states. This is especially problematic in our setting, where the number of states is extremely large due to the high dimensionality of our input (i.e. pixels from the agent's visual feed).

Instead, we use a convolutional neural network as a nonlinear function approximator to estimate the action-value function. Let θ be the neural network weights We train the neural network approximator $Q(s, a, \theta)$ by minimizing the loss function $L_i(\theta)$:

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho()}[(y_i - Q(s, a; \theta_i))^2]$$

Where $y_i = \mathbb{E}_{s'\sim\epsilon}[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1} | s, a]$ is the target for iteration i and $\rho(s, a)$, the behaviour distribution, is a distribution over states s and actions a. Note that the target value used in iteration i is calculated with the previous iteration's network weights. For faster computation at each iteration, stochastic gradient descent is used; instead of calculating the expectation, we update the weights with a single sample from distribution (see section 4.3.3 below).

4.3.2 Deep Q Network

We use a convolutional neural network to make predictions since our inputs are images. Our previous MDP formulation for Q-learning implies training the weights using the entire sequence of actions and observations, and making predictions using a state-action pair. In practice, we slightly modify this previous formulation for two reasons.

First, neural networks can only handle fixed input size and so we only use the 3 most recent images instead of the entire sequence. Second, the previous formulation would require doing a 'forward-pass' of the network (i.e. making a prediction using the neural network) every time you wanted to evalute the Q-value of a state-action pair. For a given state and a set of k actions, this would require k forward-passes. Instead, we use a network architecture in which only the image is fed to the input layer and outputs are estimates of Q-value for each possible action, and so our state is made up of only an image, rather than an image-action pair. The outputs correspond to the predicted Q-values of the individual actions for the input state. For the detailed explanation of the exact network architecture, see section 4.4.

4.3.3 Experience Replay

Often, there is a delay between actions and resulting rewards, and little association between the most recent input and the reward it just received. Also, the inputs are not independent, since consecutive inputs are highly correlated. Furthermore, the distribution of the input states is influenced heavily by recently taken actions and will vary widely if we use only the recently obtained frames as inputs.

These issues can lead to neural network taking too long to converge, or being very unstable and unable to learn efficiently. To de-correlate the inputs and stabilize the fluctuation of input sampling distribution, we use 'experience replay' technique. During gameplay we keep a fixed number of most recent experiences $\{s, a, r, s'\}$ in a 'replay memory', and train the network with random batches of experiences (processed inputs) sampled from the memory, rather than the most recently obtained states. Experience replay smoothes the change of input sampling distribution over time, and allows the images to be used for multiple weight updates, rather than just one.

4.3.4 ϵ-greedy policy

One of the fundamental trade-offs in reinforcment learning is exploration vs. exploitation. If the algorithm starts in a certain state and only chooses actions which optimize the Q function based on what it has seen, it could get stuck in a local minimum of the loss function since it settles for the first "better" strategy it finds and the spectrum of subsequent experiences would be relatively limited.

Randomly choosing an action which the algorithm does *not* believe would optimize the Q function would let the algorithm gather more data about the environment. An ϵ-greedy strategy chooses the optimal action ('the greedy strategy') with probability $1 - \epsilon$ and selects a random action with probability ϵ. We employ an ϵ-greedy strategy but anneal ϵ so that it decreases from *** to *** over a period of *** .

4.4 Training

4.4.1 Preprocessing

The raw pixel frames we received through the Malmo platform are 480 x 640 x 3 in dimension with 256 possible values for each color channel. To reduce the input to more manageable size, the images were preprocessed: we converted the image from the RGB to grayscale and rescaled it to 84 x 84. Since the actual input fed into the neural network is a stack of the 3 most recent frames, the final input dimension is 84 x 84 x 3.

4.4.2 Model Architecture

The overall neural network consists of three convolutional layers and two fully connected layers. The input layer takes an 84 x 84 x 3 preprocessed image. The first hidden layer convolves the input with 32 filters of 8 x 8 with stride 4 and applies a rectifier nonlinearity. The second hidden layer convolves with 64 filters of 4 x 4 with stride 2, followed by a rectifier nonlinearity. The third layer convolves 64 filters of 3 x 3 with stride 1, again followed by a rectifier. The final hidden layer is a fully-connected layer of 512 rectifier units. The output layer is a fully-connected linear layer (no rectifier) with an output node for each action in the action space. Our network has 5 output nodes. The following table summarizes the input and filter size at each layer of the network.

Table 1: Input and Filter Sizes of Neural Network

Layer	Input	Filter size	Stride	Num Filters	Output
conv1	84x84x3	8x8	4	32	20x20x32
conv2	20x20x32	4x4	2	64	9x9x64
conv3	9x9x64	3x3	1	64	7x7x64
fc4	7x7x64			512	512
fc5	512			18	18

Algorithm 1: Adapted from Mnih et al, 2015

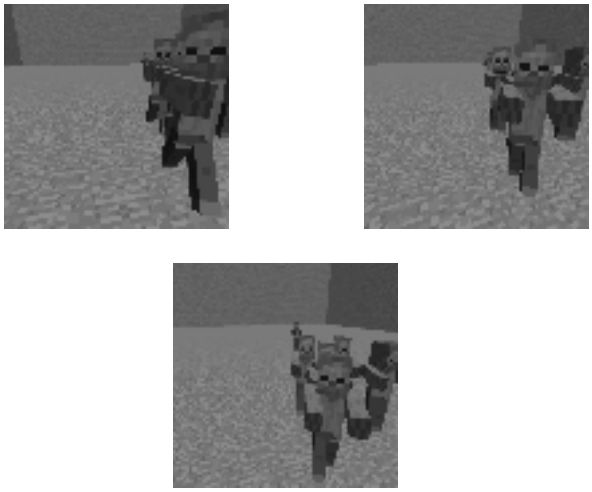
initialize Q function with random weights ;
initialize replay memory $\mathcal{D}$;
repeat
  observe initial state S ;
  while *agent is alive* **do**
    with probability ϵ:
      generate random action a ;
    otherwise:
      select $a = \arg\max_{a'} Q^*(s, a', \theta)$;
    carry out action a ;
    observe reward r and new image x' ;
    preprocess x' into ϕ' ;
    update $\mathcal{D}$ with new data: $< \phi, a, r, \phi' >$;
    sample random transitions from D ;
    update the Q Network weights to θ': ;
      Set $y = \mathbb{E}[r + \gamma \max_{a'} Q(s', a', \theta)]$;
      Perform gradient descent on
      $L(\theta) = \mathbb{E}[(y - Q(s, a, \theta))^2]$;
  end
until *terminated*;

4.4.3 Algorithm

The algorithm presented summarizes the pipeline for Deep Q-learning and other important aspects such as experience replay and ϵ-greedy strategy. We utilize the work done by

Google DeepMind to use a convolutional neural network to approximate the Q-function iteration after iteration.

Figure 2: Images of processed features



5 Results and Analysis

We chose four metrics to measure agent performance over training time: cummulative reward per life, kills per life, time alive, and average q-value.

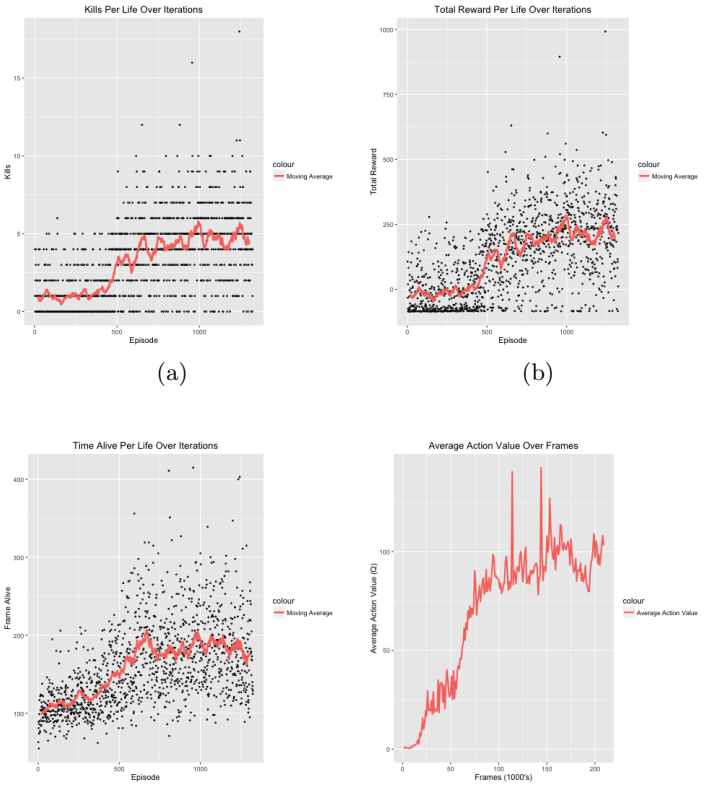
On all four metrics, the results shown in Figure 3 display clear improvement in the agent's performance across all four metrics. Kills per life increased form an initial low of between zero to two to around seven while the agent's lifespan doubled from 100 to around 200 frames. The algorithm was clearly very successful at optimizing the given reward structure: the lifetime reward increased from below zero to around 250, while at some points it achieved well over 500. Note that the average total reward metric tends to be noisy because small changes to the weights of a policy can lead to large chagnes in the states that the policy visits.

Behaviorally, the agent started out by moving and spinning arbitrarily. By the middle of training, it recognized zombies as object that provide potential reward and began to deliberately keep them in the center of the frame in order to reap this reward. By the end of trainig, the agent even began to learn more complex strategies like moving backward while facing zombies to avoid getting hit, and occasionally lunging into them to capitalize on kills.

Finally, **??** shows the increase in the average chosen Q-value over time. It shows that over time our agent gets better at picking an action that results in a higher Q-value, which provides an estimate of how much discounted reward the agent can obtain by following its policy.

*** pick what we need from here: Both averaged reward plots are indeed quite noisy, giving one the impression that the learning algorithm is not making steady progress. Another, more stable, metric is the policys estimated action-value function Q, which provides an estimate of how much discounted reward the agent can obtain by following its policy from any given state. We collect a fixed set of states by running a random policy before training starts and track the average of the maximum2 predicted Q for these states. The two rightmost plots in figure 2 show that average predicted Q increases much more smoothly than the average total reward obtained by the agent and plotting the same metrics on the other five games produces similarly smooth curves. ***

Figure 3: Results on Easy Difficulty



6 Conclusions and Future work

We succesfully trained a Deep Q Network to move around a room and kill zombies by taking visual input from the agent. Our results suggest that *foo* and *bar* methods worked best because [...]

Future work on this task might focus on modifying or expanding the agent's action space so that the agent can perform better or wider range of movements and therefore can better simulate natural movement. Additionally, researchers could focus on tuning convolutional neural network to further enhance the performance or learning curve of the algorithm. This might involve using different gradient descent algorithms or initial weights, as well as considering different network architectures. Furthermore, updating scheme of the replay memory could be altered so that instead of sampling uniformly at random, important transitions that helped the learning the most have higher probability of being sampled. Also, when the replay memory runs out of space, the current algorithm always overwrites the oldest experiences. However, a better strategy would be to discard the least helpful transitions first.

The most interesting direction for future work, however, would be creating, training on and testing on many other relevant and interesting tasks with the platform Malmo. Mincraft

not only allows for tasks as simple as navigation or killing, but also is capable of providing more complicated tasks such as building or tasks that involve multiple agents.

Overall, this project has shown the power of applying deep reinforcment learning in settings with high degrees of difficulty and with limited knowledge of the environment or game rules.

References

[1] Mnih, Volodymyr; Kavukcuoglu, Koray; Silver, David; Graves, Alex; Antonoglou, Ioannis; Wierstra, Daan; Riedmiller, Martin. *Playing Atari with Deep Reinforcement Learning*, December 2013.

[2] Matiisen, Tambet. *Demystifying Deep Reinforcement Learning.* December 2015. `https://www.nervanasys.com/demystifying-deep-reinforcement-learning/`

[3] Chen, Kevin. *Deep Reinforcment Learning for Flappy Bird* 2015.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, Demis Hassabis. *Human-level control through deep reinforcement learning.* February 2015.