Fighting 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 survive as long as possible using reinforcement learning. Our agent is only equipped with a sword to attack the zombies while the zombies attack back and try to kill the agent. Our objective is to have the agent learn what policies to execute based solely on the pixels of the gameplay screen. Rather than giving the agent information about the environment or any guidance on killing zombies, it has to learn optimal actions based on its past experience.

If neural networks are an approximation of the human brain and Minecraft can be viewed as an approximation of the natural world, our broader goal is to implement our agent as if it is learning from and interacting with the world like a human. Using deep Q-learning, the agent showed marked 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 work. Our major inspiration was DeepMind's 2013 paper on deep reinforcement learning, in which an agent was trained with a single model architecture that used a convolutional neural network to perform a variant of Q-learning. The agent successfully played 7 different Atari 2600 games and outperformed human experts in a few of them [3]. Instead of solving the standard action-value function (Q function) with Bellman equations, a 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. Furthermore, no game specific knowledge was provided, and the only inputs were raw pixel values.

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 [1]. 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. Researchers can create environments and combine different states, actions, and rewards to simulate 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 and instead has to train a neural network and quickly decide its 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 with many of the Atari games tested by DeepMind and with Flappy Bird, where the game provides a bird's-eye view.

4 Method

4.1 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, $A = 1, \dots, K$. This action then changes the agent's state and the agent receives rewards based on its new state. The reward indirectly depends not only on the new state it just entered, but the entire sequence of actions and observations it made until it entered the new state. Thus, to make an appropriate action in the new 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 a finite

Markov decision process (MDP), where we use the sequence s_t as a distinct state at each time-step t.

4.2 Q-learning and Deep Q Network

4.2.1 Reinforcement Learning Background

In reinforcement learning, the agent is trained to maximize aggregate future rewards. With the discount factor of γ, the future discounted reward at time t is $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$, where T is the number of time-steps 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 reward achievable by executing any action, a, after seeing some sequence s, where π is a policy mapping sequences to 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 the action-value function for all 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 high due to the nature 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:

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

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 behavior 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 the distribution (see section 4.3.3 below).

4.2.2 Deep Q Network

We use a convolutional neural network to make predictions since our inputs are images. Our previous MDP formulation for Q-learning implied 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 for two reasons.

First, neural networks can only handle fixed input size and so we only use the three 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 evaluate 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 the outputs are estimates of the Q-value for each possible action. Thus our state is made up of only an image, rather than an image-action pair. For a detailed explanation of the exact network architecture, see section 5.

4.2.3 Experience Replay

There are several issues here which can lead our neural network to learn inefficiently:

- There is a delay between actions and resulting rewards
- There is sometimes little association between the most recent input and the reward the agent just received
- The inputs are not independent, since consecutive inputs are highly correlated
- 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.

To de-correlate the inputs and stabilize the fluctuation of the input sampling distribution, we use the 'experience replay' technique. During gameplay we keep a fixed number of the 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 smooths the change of input sampling distribution over time, and allows the images to be used for multiple weight updates, rather than just one.

4.2.4 ϵ-greedy policy

One of the fundamental trade-offs in reinforcement learning is exploration versus 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 and settle for the first "good" strategy it finds, limiting the range of subsequent experiences.

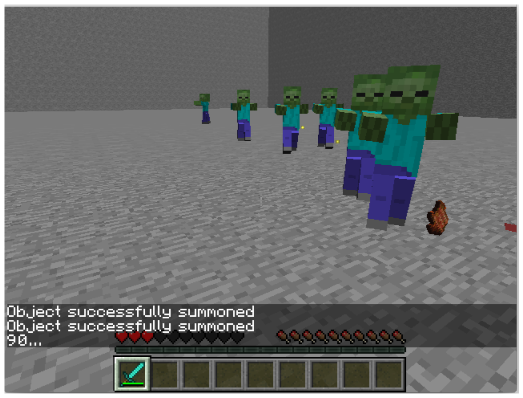
Randomly choosing an action which the algorithm does *not* believe would optimize the Q-function lets the algorithm gather more data about the environment. An ϵ-greedy strategy chooses the optimal action ("the greedy strategy") with probability $1-\epsilon$ and selects a random action with probability ϵ. We employ an ϵ-greedy strategy but anneal ϵ so that it decreases from 1.0 to 0.05 over a period of 80,000 frames.

5 Implementation

5.1 Scenario

The agent begins in the middle of a 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 while the agent is equipped with a sword to attack the zombies. As the agent destroys zombies, new ones continue to spawn (so that there is always a constant number of zombies), and the game continues to run until the agent loses all of its health. After each death, the scenario resets to the same initial state.

Figure 1: Minecraft screenshot


Object successfully summoned
Object successfully summoned
90...

5.2 Actions and States

We have given the agent five actions: the agent can move forwards, backwards, turn left, turn right, or remain stationary. To simplify the action space, the attack command is turned on the entire time. Malmo allows the agent to move and turn at different speeds, but we chose a single speed at which to move for our configuration.

The observation at time-step t, x_t, is the image of the player's vision represented by pixel values. Project Malmo provides the functionality to retrieve this 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.

5.3 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 incentivize the agent to survive longer.

5.4 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 a more manageable size, the images were preprocessed: we converted the image from the RGB to greyscale 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.

5.5 Model Architecture and Algorithm

The overall neural network consists of three convolutional layers and two fully connected layers. A rectifier nonlinearity is applied at each stage except for the output layer, which is a fully-connected linear layer. The following table summarizes the input and filter size at each layer of the network.

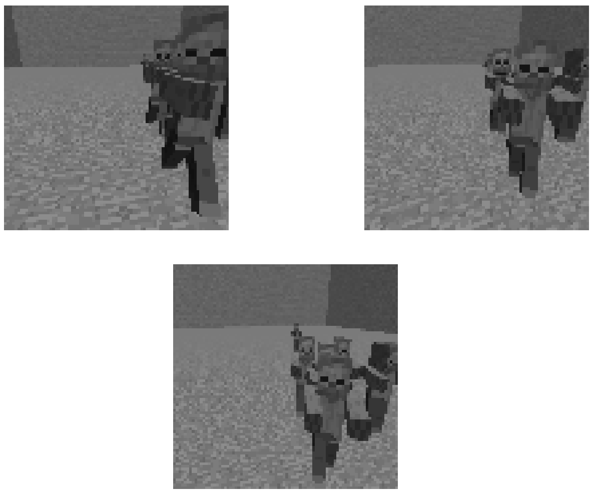
Table 1: Input and Filter Sizes of Neural Network

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				5

Algorithm 1: Adapted from Mnih et al, 2015

```
initialize $Q$-function with random weights ;
initialize replay memory $\mathcal{D}$ ;
repeat
    observe initial state $s_0$ and preprocess to $\phi_0$ ;
    while agent is alive do
        with probability $\epsilon$:
            generate random action $a$ ;
        otherwise:
            select $a = \arg\max_{a'} Q^*(\phi, a'; \theta)$ ;
        carry out action $a$ ;
        observe reward $r$ and new image $x'$ ;
        preprocess $x'$ into $\phi'$ ;
        update $\mathcal{D}$ with new data: $< \phi, a, r, \phi' >$ ;
        sample random transitions from $D$ ;
        update the $Q$ Network weights to $\theta'$:
            Set $y = \mathbb{E}[r + \gamma \max_{a'} Q(\phi', a'; \theta)]$;
            Perform gradient descent on
              $L(\theta) = \mathbb{E}[(y - Q(\phi, a; \theta))^2]$ ;
    end
until terminated;
```

Figure 2: Images of processed features

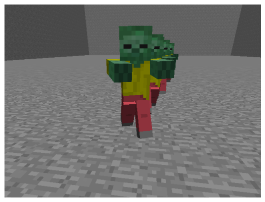


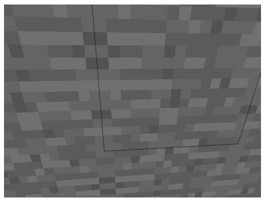
6 Results and Analysis

We chose four metrics to measure agent performance over training time: cumulative reward per episode, kills per episode, time alive, and average Q-value. As a baseline configuration, we chose the five-action space described earlier and an ‘easy’ difficulty of five zombies.

The baseline results shown in Figure 4 display clear improvement in the agent’s performance across the first three metrics. Kills per episode increased from 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 these metrics tend to be noisy because small changes to the weights of a policy can lead to large changes in the states that the policy visits.

Figure 3: Images Corresponding to High and Low Q-values


(a) High Q-value

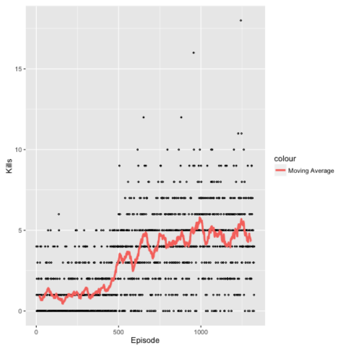

(b) Low Q-value

Behaviorally, the agent started out by moving and spinning arbitrarily. By the middle of training, it recognized zombies as objects that provide potential rewards and began to keep them in the center of the frame to achieve this reward. By the end of training, 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 achieve kills.

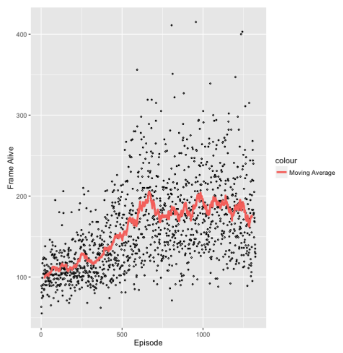
Figure 7 shows the increase in the average chosen Q-value (the red line displays the baseline configuration). It shows that over time the agent gets better at picking an action that results in a higher Q-value. The Q-value provides an estimate of how much discounted reward the agent can obtain by following its policy. Figure 3 presents two screenshots associated with high Q-value and low Q-value respectively. This shows that the agent has learned to predict higher Q-values when it has turned towards a zombie, since it then has a chance of attacking the zombie and getting higher rewards.

We experimented with changing our configuration in two ways. First, we increased the action space from five to nine actions by allowing the agent to turn left and right at three different speeds. As shown in Figures 5 and 7, this change resulted in *worse* performance for the agent. The cumulative reward per episode increased to an average of just below 100, while it broke 200 in the baseline five-action space configuration, while the average Q-value was noticeably lower than under the baseline configuration. This could be because the neural network has to make predictions across more actions for any given state, and with the same amount of data, the quality of those predictions would suffer. If this is the case, the algorithm simply needs more time to train. Alternatively, it could be that the nature of the nine-action space adds more complexity without adding performance and its performance might never equal the performance under the five-action space. It is difficult to speculate on which explanation is more likely without running longer trials.

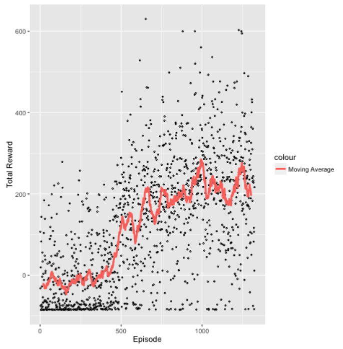
Figure 4: Results for Baseline



(a) Kills per Life over Iterations



(b) Time Alive per Life over Iterations

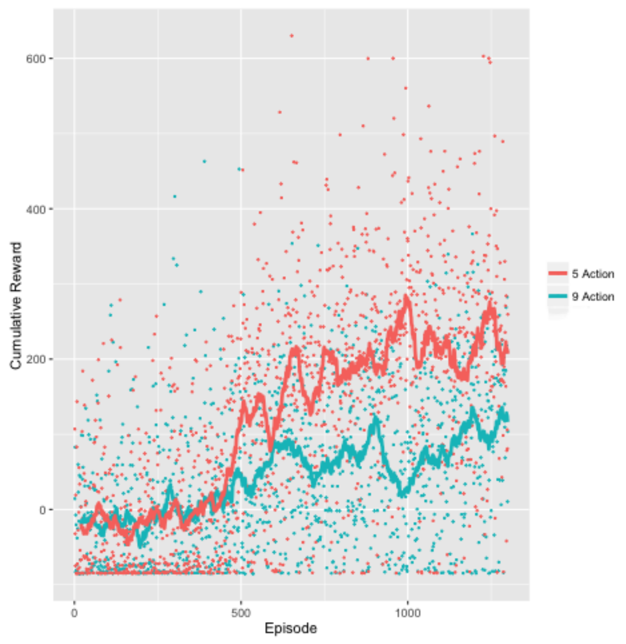


(c) Average Action Value over Frames

The other experiment we ran was increasing the number of zombies from five to seven, doubling the zombies’ health, and spawning them scattered around the room rather than clumped in one space. We call this ‘hard’ difficulty as opposed to the ‘easy’ difficulty under the baseline configuration. As expected, the agent’s performance is noticeably worse under this difficulty, with both reward (Figure 6) and average Q-value (Figure 7) being lower than under the easy difficulty. Interestingly, the agent’s initial change in reward and average Q-value is close to zero and sometimes even negative under the hard difficulty before increasing. This suggests that that the greedy policy initially led to worse outcomes than the random action.

One challenge we encountered was the agent’s inability to respond to attacks outside its field of view. Since the agent’s visual input does not provide any information about the source of the attack, the agent did not learn to respond effectively. This behavior typifies the limitations of simulations with Malmo: without a third-person view or more environmental information, it would be difficult to train the agent to respond to these attacks.

Figure 5: Comparing Rewards between 5-Action space vs 9-Action space



7 Conclusions and Future work

We successfully trained a deep Q-Network to move around a room and kill zombies by taking visual input from the agent. This project shows the promise of combining deep Q-learning with reinforcement learning to play games by accomplishing difficult tasks. Specifically, the convolutional neural network allowed us to make meaningful predictions for optimal actions even with extremely high numbers of states.

Future work on this task might focus on modifying or expanding the agent's action space so that it can better simulate natural movement. We could also incorporate other kinds of sensory information (e.g. audio) from the game so that when the agent has zombies behind it, it can respond by facing to turn and attack. This would better approximate human sensory input.

The updating scheme of the replay memory could be altered so that instead of sampling uniformly, we sample important transitions that helped the learning with higher probability. Also, when the replay memory runs out of space, the current algorithm always overwrites the oldest experiences; a better strategy would be to discard the least helpful transitions.

The most interesting direction for future work, however, would be creating, training and testing many other relevant and interesting tasks with the platform Malmo. Minecraft 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 involving multiple agents. It could also be interesting to train agents in 'outdoor' environments, as opposed to a controlled indoor setting with uniform color and lighting.

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

Figure 6: Comparing Rewards between Easy vs Hard Difficulty

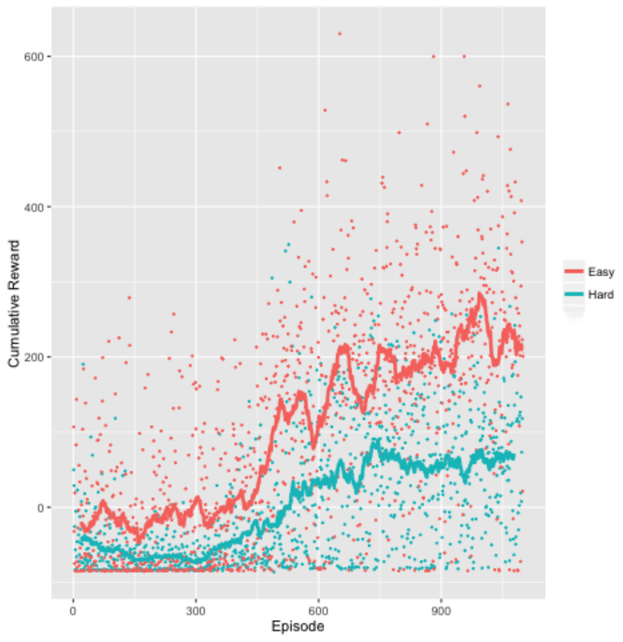
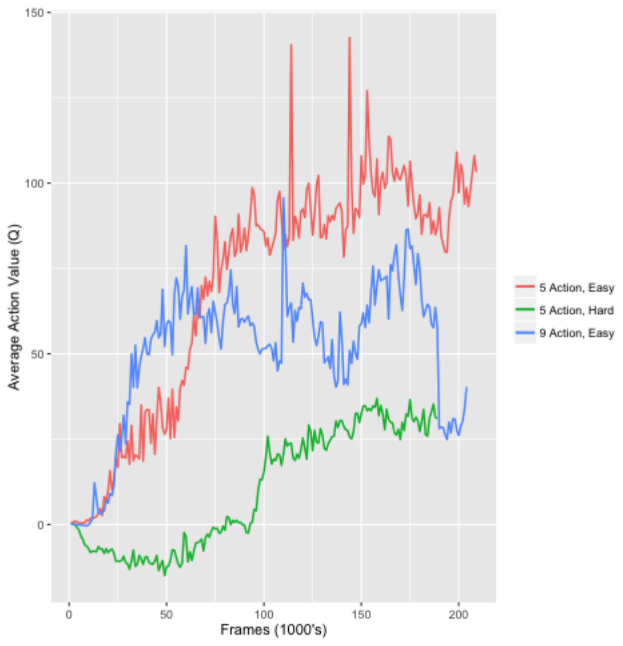


Figure 7: Comparing Q-values across Difficulties and Action Spaces



References

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

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

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

[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.

[5] Martn Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Man, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Vigas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng.

TensorFlow: Large-scale machine learning on heterogeneous systems. 2015. Software available from tensorflow.org.

[6] Johnson M., Hofmann K., Hutton T., Bignell D. (2016) The Malmo Platform for Artificial Intelligence Experimentation. Proc. 25th International Joint Conference on Artificial Intelligence, Ed. Kambhampati S., p. 4246. AAAI Press, Palo Alto, California USA. https://github.com/Microsoft/malmo