Applying AlphaGo Zero to Non-Observable 3D State Spaces

Jordan Patterson
University of Victoria
jpatts@uvic.ca

Abstract

This project explores the implications of applying algorithms traditionally used for fully observable board states, such as Chess and Go, in non observable environments such as Doom, the 1991 first person shooter. The algorithm in question is AlphaGo Zero, by D. Silver et al., and is changed in minor ways to transition from Go to Doom [1]. This merge of Doom and AlphaGo Zero is presented as AlphaDoom. The majority contribution in this paper is a next state predictor, which is a generative network that takes a state s_0 and action a_i as input, and outputs the perceived next state s_1. This state predictor achieves very convincing results, and is the backbone of the Monte Carlo Tree Search (MCTS) used in AlphaDoom.

1. Introduction

Solving games has long been an avid area of research, and involves knowing the outcome of a game based on the assumption that players are playing perfectly. There are three different levels of proofs for solving a game. An ultra-weak proof determines if the first player will win, lose or draw from the initial state, assuming perfect play by all players. The idea behind ultra-weak proofs is if the advantage of going first is strong enough to consistently justify an outcome. This is typically done by reasoning about the inner workings of the game, and does not propose an algorithm to enact this policy. A weak proof is an enforceable algorithm that can win or draw any game from the initial state, assuming all players play perfectly. This is implemented as a table containing optimal moves for every state, or a computer program that generates, approximates or otherwise uses this table. The problem with weak proof algorithms is that they only know the optimal move based on an N_{th} lookahead, where N can't be infinity. So game trees that diverge from initially weak plays are not explored, despite the fact that "bad" plays can be a valid strategy in many games, in order to save resources. Thus, if opponents do not choose the algorithm's definition of a perfect move, the algorithm can fall apart. This is because the weak proof algorithm does not fully understand the inner workings of the game. Finally, a strong proof is the evolution of a weak proof, in that it can produce perfect moves that lead to a win or draw from any non losing state, regardless of an opponents actions. This means that strong proofs are invariant to imperfect plays, as they fully understand the game.

Before deep learning took off many games had already been solved, such as weakly solved Checkers and strongly solved Connect Four. However, games with much larger state spaces such as Chess and Go remain weakly unsolved, ignoring smaller variations of these games. This is due to exponential differences in complexity for games with larger state and action spaces. Despite the immense difficulty that goes into solving a game like Chess, these games only exist in the 2-dimensional plane. 3-Dimensional environments are home to the most complex games visualizable by humans. Thus, as Connect Four and Tic Tac Toe can be seen as the simplest 2D solveable games, Doom can be seen as one of the simplest 3D solveable games. So weakly solving Doom would be a step towards solving 3D environments, and the algorithm would be applicable to general intelligence as well. As a 3D game, the objectives of Doom are not as clearly defined as Chess or Go, as there can be several objectives, each inherent to a game mode. So for this paper, the primary game mode of concern is 1v1 deathmatch, where the objective is to simply kill the opposing player. The state space in Doom is all possible game frames in a level, for each level in the game. This is immeasurably larger than Go or Chess. The action space for this paper is the movement directions forward, back, left, right, plus the shoot action, for a total of 5 possible actions for any given state. This is relatively small compared to Go or Chess.

As a problem, this seems like a simple one; just create a strong lookup table of state action pairs, since there are only 5 actions, with an N state lookahead. However, its not that easy for three reasons, which are the focus of this paper.

1. This is not a turn based game like Go or Chess; there is a very small, finite amount of time to compute an action. This is actually the biggest problem, as many

deep learning methods are brute force solutions. This will become more relevant as the following two points are explained.

2. The states being image frames cause many problems. One is that successive states are very similar to each other, making the next state irrelevant until a certain number of frames have past. Compared to board states in chess, where two similar board states can lead to vastly different outcomes, this adds additional variability and complexity to the algorithm. Another issue is that Doom is a highly textured game, so repeating patterns are frequent along floors, walls and ceiling, obscuring relevent information like enemies, items and points of interest. This means some form of pattern recognition is needed to help highlight the important information within a state. Finally, images are magnitudes larger than the states in any board game, so working with them becomes expensive with both memory and computation.

3. Looking ahead immediately becomes an extremely difficult problem, as the next states are no longer observable. Rather than having the knowledge of what each future state would look like, the only option other than cheating is to approximate what the next state will be given a proposed action. Not only is this very computationally expensive, which is a problem for #1, but there is also a compounding error rate with this approach. Each lookahead will not be perfect, thus there will be an error term ε associated with each. These errors are multiplicative not additive. So if the error at predicted state s_{+1} is ε, the error at s_{+2} is ε^2, and error at s_n is ε^{+n}. Error is exponential, forcing a breadth over depth approach to game tree searching. This has the advantage of being resilient to imperfect plays, but ultimately has no chance against an algorithm that can look N_{+1} states ahead. Thus the problem of creating a perfect 3D agent for Doom boils down to creating the best possible next state predictor, in a performance vs error metric.

None of these problems have been solved, although there is an increasing amount of work being done in this area.

2. Related Work

As stated above, the brain of this algorithm is derived from the paper "Mastering the game of Go without human knowledge", by D. Silver et al. of Deepmind. Implementation details will be explained in depth later in the paper. To summarize the algorithm, a Monte Carlo Tree Search (MCTS) is combined with a convolutional neural network (CNN) that takes a state as input, and outputs probabilities of each action along with a value for the state [2]. The action probabilities are compared to the MCTS edge values using softmax loss, while the state value is compared to the reward in mean squared error (MSE) loss. This means that the MCTS is essentially the training target for the CNN, making it a policy improvement operator. Thus the network will train to only be as good as the lookahead depth and breadth of the MCTS. In the AlphaGo Zero implementation, 1000 simulations are ran every time before an action is taken, with each simulation either increasing depth or breadth of the MCTS by expanding a leaf node. The overall outcome of AlphaGo Zero is a revolution of how unsupervised and reinforcement learning are done. It is the current SOA of reinforcement learning. Flaws in the algorithm are the massive computational and time requirements to replicate their results.

Another group that tackled 3D reinforcement learning in Doom is from Carnegie Mellon, with papers "Playing FPS Games with Deep Reinforcement Learning" and "Arnold: An Autonomous Agent to play FPS Games" by G. Lample and D.S. Chaplot [3, 4]. The primary architecture is defined in the former paper, whereas the tuned implementation is defined in the latter. They approach the non-observable problem in several interesting ways. One is that they train with inherent game knowledge, and then test without it. This is to get around the texture problem described earlier, and allow the agent to learn the game features. This is done by passing a one-hot vector to the network, informing the agent whether relevant objects are in the state, such as enemies and health packs. Another key idea they had was to use an LSTM as the last layer in their CNN, to help deal with the similar state problem and allow for more educated guesses when predicting what action to perform based on prior results [5]. They achieved very good results across a wide variety of scenarios, and are one of the leaders in published Doom AI research at the moment. The only problems with their approach is that they do train on game features, which means this is not a pure pixel based approach. Furthermore, they do not do any form of lookahead, likely because they couldn't predict successive states. Thus, an algorithm that can do this should be able to outperform Arnold by a significant margin.

3. Architecture

Design of the model is composed of a main network and a sub network. The main network is derived from AlphaGo Zero, with little changed (see figure 1). The primary difference is depth of residual layers, which are capped at 19 for Doom to keep performance at acceptable levels, compared to 39 for the original model [6]. Loss is unchanged from AlphaGo Zero, taking the mean squared error (MSE) between the MCTS z values and the predicted v values generated from the value head:

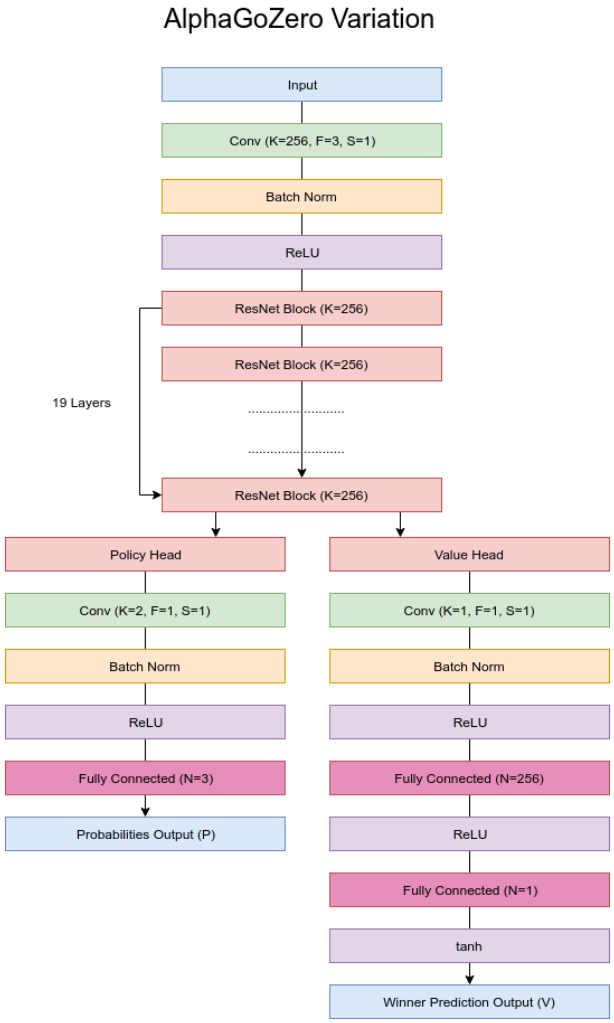

Figure 1. AlphaDoom variation of AlphaGo Zero

$$L_{MSE} = \sum_{i=1}^{N} (z_i - v_i)^2 \tag{1}$$

As well as the softmax cross entropy (SCE) loss of the MCTS one-hot chosen action pi against the probabilities p generated from the policy head:

$$L_{SCE} = \sum_{i=1}^{N} -pi \log p \tag{2}$$

L2 regularization is added with strength parameter c:

$$L_2 = c||\theta||^2 \tag{3}$$

The end result for total loss calculation is:

$$L_{total} = L_{MSE} + L_{SCE} + L_2 \tag{4}$$

I did not bother including the temperature hyperparameter T with pi^T in the SCE loss, as it is unnecessary in a state space as small as this one to have many exploratory variables.

The sub network is the main contribution of this paper, and is a next state simulator. Given image input of (N, 32, 32, 1), and action input of (N, 1, 1, 3), it will output the predicted transformation that will occur between the current state and the future state after following action a_i. It does this with a V-net auto-encoder, which starts and ends at its input size. As input progresses through the network, it is downsized by convolutions with strides of 2 until its size is (N, 1, 1, F), where F is maximum number of filters, before being up-sized back to original input shape with transposed convolutions. At the centre of the network, when shape is (N, 1, 1, F), the simulator concatenates the action vector corresponding to each action a_i with input N_i. This is how the network knows what to transform the frame with to get the predicted next frame. See figure 2 for more details. Loss for the simulator is just the MSE between the predicted transformation and the true transformation:

$$L_{simulator} = \sum_{i=1}^{N} (truth_i - pred_i)^2 \tag{5}$$

A key difference from AlphaGo Zero and ResNet in my implementation is the ordering of batch normalizations and ReLUs in the residual blocks. I found a significant performance increase when doing ReLU before batch normalization, and then not taking the ReLU of the element-wise sum for the simulator. The training statistics that inspired this are shown in figure 3, where there is a clear advantage to doing ReLU beforehand than otherwise for the simulator. I noticed significant differences in the AlphaGo Zero implementation when changing this as well, however decided to use the original residual block for it instead, to make conclusions on the unmodified algorithm.

Within the AlphaGo Zero architecture are two mini architectures, an MCTS for choosing states and policy improvement, as well as an experience replay for storing memories [7]. The MCTS starts with a root node, and expands from that node for the number of simulations being performed. If there are 1000 simulations, there will be 1000 nodes. The expansion is done by choosing successive actions (edges) based on a criteria that determines quality of

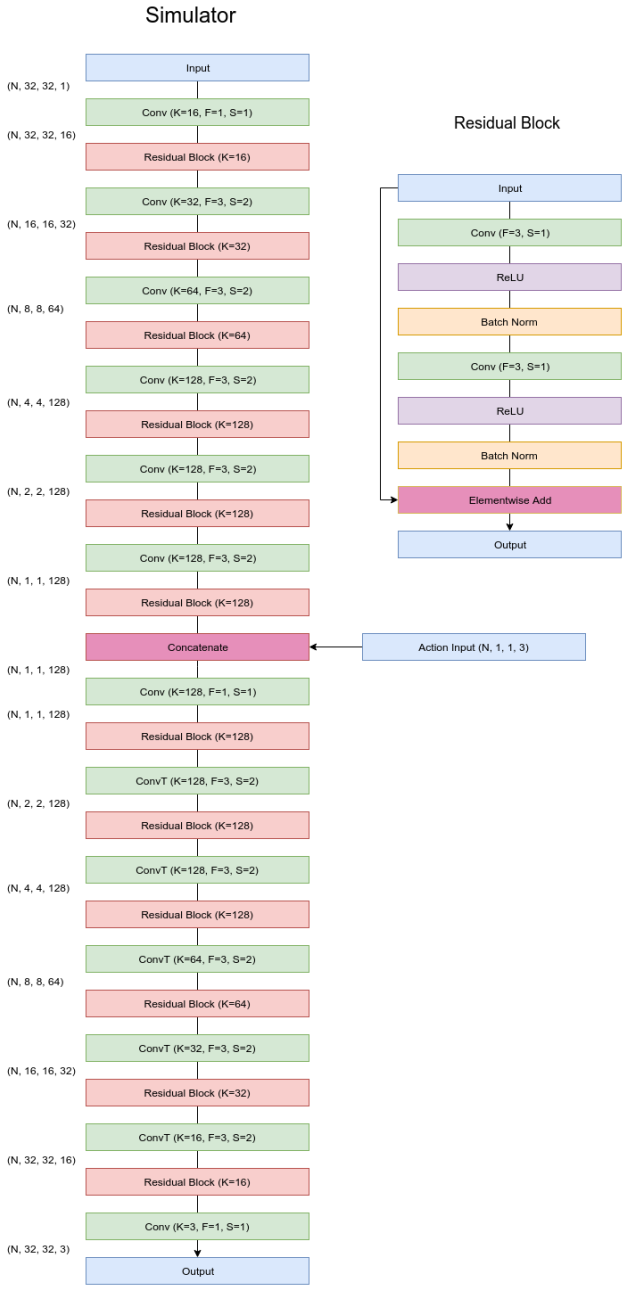

Figure 2. Next state simulator architecture

these edges. After an edge without a node is chosen, it is expanded into a leaf node, and the tree is backprogogated to update a visit count parameter which is used to both detect quality of state and perform exploration. At the end of each set of simulations, the edge chosen is from the root node, and is based on visit counts of each edge with an exploration hyperparameter. The tree is then thrown away, as the key idea is that the MCTS improves the predictions of the policy head, while the policy head's improved predictions

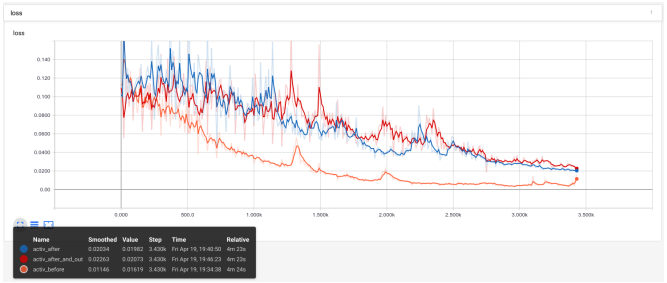

Figure 3. Performance differences of residual block designs from the simulator. Orange is ReLU before batch norm, and is not applied after element-wise sum. Blue is ReLU after batch norm, and is not applied after element-wise sum. Red is the standard residual layer implementation

improve the MCTS edge decisions. This ends up being a dog chasing its tail relationship, where the dog is the policy head and the tail is the MCTS. The experience replay is simply a database of memories, where each memory is a tuple of (s, pi, p). If the replay is full, it removes the oldest memories, which should encourage learning. Memories are chosen randomly however, as prioritized replay adds additional time complexity.

4. Methodology

The Doom engine is a python library called ViZDoom that can be pip installed [8]. It offers many features for user control of scenarios, and in this paper the most basic was used. The scenario involves an agent spawning in the middle of a room, with an enemy spawning randomly along the x axis, while being at a constant z distance away. The goal is simply to shoot the enemy once, and if success is 50% or greater, then learning will have occurred, as random policies succeed $< 50\%$ of the time. Possible actions are translation along the x axis, so movement either left or right, and shooting. Only one action can be performed at a time.

During an episode, if the enemy is not killed within 60 actions (300 frames with a 5 frame skip), then the session will be considered a failure, and the reward for all (state, action, new_state) tuples associated with this episode will be -1. If, however, the enemy is killed within 60 actions, a reward of +1 is provided. Some experimentation was also done where increased rewards were given to sets of states that killed the enemy in less actions, with good results. A reward of +2 was given for less than 30 actions, and +3 for less than 15. However, this was removed from the final version, as I wanted to evaluate AlphaGo Zero without modification.

The simulator did not operate on the raw images produced from the Doom engine. Rather, a complicated preprocessing method is applied to the images beforehand to

reduce number of unique features from [0, 255] to 4 possible values, selected by the method. The preprocessing is shown in figure 4, and is described in detail here.

First, a large (39, 39) Gaussian blur is applied to the image, which is of size (640, 480, 3). This is an effort to reduce the variance in texture that is rampant in each image, since texture is useless and detrimental for detection of important objects. Next, a central crop is made of the blurred frame, as information along the edges of the screen have a strong chance of being unimportant, particularly the ceiling, which takes up a large section of the screen. At this point the downsample occurs to a size of (32, 32) using the nearest neighbour method. It is able to keep a large amount of detail due to a combination of the large input size, large Gaussian filter and large central crop. Changing any of these have a large effect on the image at this point, as too low of a resolution means finer detail is completely lost, particularly around the enemy. Too low of a blur preserves too much texture after the downsample, and not taking the central crop makes the enemy too small to be considered its own class.

These are all very important when the next operation is a k-means clustering, as it is a necessity for there to be only 4 clusters. This is because one is needed for the floor, wall and ceiling textures, with the 4th needed for all other objects, in this case the enemy. Due to the previous steps the k-means performs exceptionally, as seen in figure 4, regardless of where the frame is on the screen. None of the textures apparent in the original image remain. The final step is to reduce complexity of the image by making it greyscale, as the colour channels are useless when only 4 different colours are present. Thus the final output is an extremely simple representation of the original image, without a loss of any important information, and being of very low complexity. This preprocessing algorithm was definitely the hardest part of this project, as it took a long time to make it well tuned and fast for the task at hand.

All code was written in TensorFlow with a modular design [9]. Both neural networks were optimized with the Adam optimizer, and initialized with the Xavier initializer [10, 11]. Learning rate was set at 10^{-3} for the simulator and 10^{-4} for AlphaDoom; learning rate scheduling was not used for either. Batch size was 100 for the simulator, and 48 for AlphaDoom. The simulator was trained for 500 epochs, whereas AlphaDoom was only trained for 60, as it does not yield better results with longer trains. As stated above, a skiprate of 5 was used between selecting an action and recieving a new frame, meaning actions were repeated for 5 consecutive frames. AlphaDoom trained on 4 stacked frames at a time, in order to depict history and motion. This gives the model a sense of direction. All networks had 1

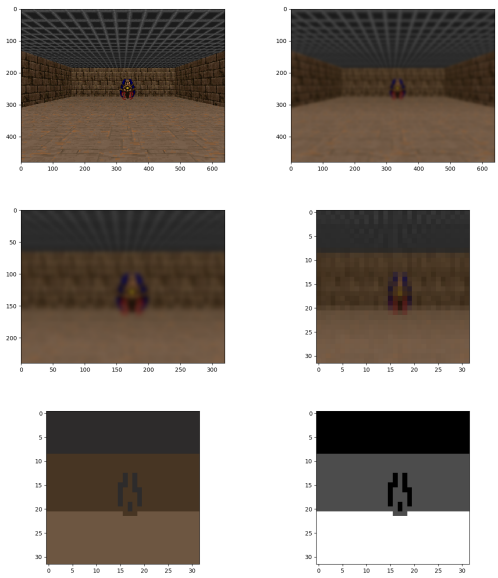

Figure 4. Preprocessing steps, from top left to bottom right: input, 39x39 gaussian blur, central crop, resize to (32, 32), kmeans clustering with 4 clusters, greyscale

colour channel for inputs, meaning all input was greyscale. The replay memory stored a maximum of 1000 states at a time, and once memory size is reached, earlier and hopefully worse states were culled. As for the MCTS, only 8 simulations were done before selecting each action, for two reasons. The first is that the compounding error rate of simulations would cause exponentially worse simulations as simulated states are simulated on. The second is each simulation is relatively expensive, and doing more than 8 simulations removes the real time aspect of this. The last parameter in the MCTS is for applying Dirichlet noise to probabilities assigned to edge nodes, set to a strength of 0.03 and epsilon of 0.25.

5. Results

Performance was highly variable based on the weight initialization and the quality of initial moves made. It fluctuates from significantly worse than random to slightly better than random; alas this is the curse of reinforcement learning. Figure 5 shows one of the better models, where it learns some patterns but can't seem to locate the enemy. After nearly 8 hours of testing, I figured out why this was the case. It was because the images returned by the simulator were random noise, despite during training being nearly perfect, as shown in figure 6. So the entire time, the MCTS was deciding what to do based on noise. And as stated earlier, since it is the only policy improvement in the network, the policy head had nothing to chase.

After many tests, I confirmed the reason behind the ran-

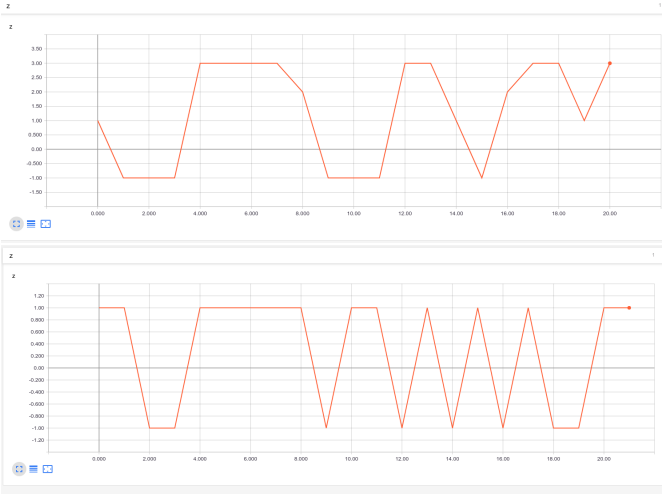

Figure 5. Comparison between AlphaDoom and random for rewards over 20 episodes. Top: AlphaDoom, Bottom: random

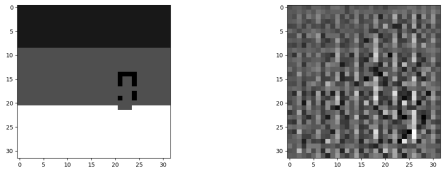

Figure 6. Left: Input to simulator, right: Output of simulator

dom noise; it was because the model was not restoring properly. TensorFlow never said anything about this however, it let the model run just fine despite explicit calls to restore it. The console was devoid of any errors or warnings. I later found out that the status of the restoration needs to be manually called for console output [12]. Furthermore, while the model was not being restored, the epochs and global step were. So it was inconsistently restoring some checkpointed objects, but not all of them. With further extensive testing, I was unable to restore the model no matter what I tried.

This was made even worse by the delayed restore TensorFlow does, which causes the model to restore only after receiving input. This makes it harder to determine when a restore is failing, and at a point I had to manually print out the weights before and after restoration to prove that they were different. The big issue here is that the model will just take on the initialized weights after recieving input, so checking that they are empty does not work. Even without any changes between creation and restoration, TensorFlow would complain about some error while restoring. If it was restored with an optimizer, it outputs the error in figure 7. If restored by itself, another error appears. There is no reasonable explanation I could come to behind these errors, and thus I was forced to stop here.

```
Traceback (most recent calllast):
 File "simulator.py", line 131, in<module>
  main()
 File "simulator.py", line 128, inmain
   model.predict(s0,action)
 File "simulator.py", line 117, inpredict
  self.status.assert_consumed()
 File
"/home/jpatts/.local/lib/python3.6/site-packages/tensorflow/python/t-
aining/checkpointable/util.py", line 1013, inassert_consumed
   raise AssertionError("Unresolved object in
checkpoint: %s" %(node,))
AssertionError: Unresolved object in checkpoint: attributes{
 name:"VARIABLE_VALUE"
 full_name:"beta1_power"
 checkpoint_key:
"optim/beta1_power/.ATTRIBUTES/VARIABLE_VALUE"
}
```

Figure 7. Error message while attempting to restore

I tried training the simulator before running AlphaDoom in the same file, but this adds 10 minutes to training time before I can see any results; needless to say this makes for a poor workflow. From the few training iterations I was able to run while training the simulator in AlphaDoom, it strongly overfit to just moving left or right after a while. Figure 8 shows the result of this, achieving a positive reward 50% of the time, due to only moving in half the possible directions. This would be easier solved if I could run sessions quicker, as often a bad initialization or gradient update kills all chances of decent performance, so many sessions are required to evaluate true performance. I strongly believe that this algorithm would work with some tweaks, based on it learning local optimum solutions (moving only left or right).

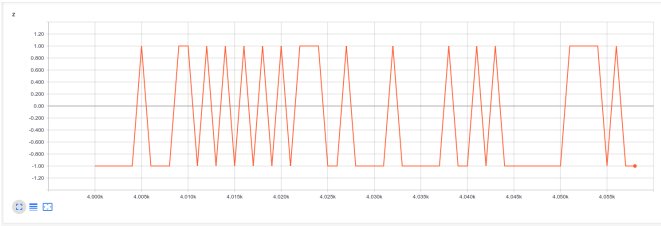

Figure 8. Best trained model achieving positive rewards 50% of the time. Starts off well, but learns to only move and shoot in one direction, as this is a local optimum. Does not learn the enemy behaviour.

6. Conclusion

Overall, this project did raise some interesting questions, some of which were answered. The next state simulation is an interesting solution to the problem of seeing future states based on prospective actions, however I do think there are more areas that can be explored here. One idea I have is

based on board states in Chess and Go. It is possible to receive the state input from a bird's eye view; with this, one can model the state space as a game board, using a number for the agent, a number for enemies, and a number for items. I think this would be very successful, with the primary issue being not actually learning from raw pixels anymore. Another option would be to preprocess input with background removal such that only items of interest are showing (no walls, floors, or ceiling); this could improve prediction of next states.

With AlphaGo Zero, I think many modifications need to be made for good performance in Doom or 3D environments in general. This is not Go; complicated decision making is not required. Thus I would significantly reduce the number of ResNet layers; this will help training and running time, likely without reducing performance. Furthermore, I would modify the reward of the value head to reflect time taken to solve scenarios; faster decision making is far more important than deeper thinking for Doom, so rewards need to decrease the longer the scenario takes. Finally, an MCTS is overkill for this problem, and a heuristic that requires many simulations to yield good results is not very useful. When doing less than 10 simulations, the visit count N can not be trusted. I believe a better heuristic would be Boltzmann exploration, as it is very effective in similar tasks and the state spaces are not very complex anyways. A better data structure than an MCTS would be a simple 1 layer lookahead for all potential actions; the one with the best Boltzmann parameter would be selected. This allows for exploration early on, powerful on policy decision making later, and sweet real time performance.

Even without making changes as drastic as those suggested above, results are not bad as is. The algorithm was trained for only 60 epochs, and figured out a way to reliably get rewards half the time. These were the initial results of the Deep Q-Network (DQN) I made prior to this as well [13]. Obviously with some fine tuning and longer training it will be able to solve this environment. Overall, I think that this project was most useful in establishing a potential method for applying fully observable state algorithms to non observable state spaces. Furthermore, with the code designed to be modular, it will be very easy to test new algorithms or variations by removing and adding components. Finally, as a term project I am happy with the end result, despite the small disappointment of not solving the problem optimally. I think the problem solving and research methods I learned will help me to do everything quicker and without less help next time, as well as have the experience to know not to put all my faith in TensorFlow.

References

[1] D. Silver et al, *Mastering the game of Go without human knowledge,* Google Deepmind, doi:10.1038/nature24270, (2017)

[2] Y. LeCun, Y. Bengio, G. Hinton, *Deep learning,* Nature, 521(7553), pp.436-444. (2015)

[3] G. Lample, D.S. Chaplot, *Playing FPS Games with Deep Reinforcement Learning,* Carnegie Mellon, AAAI. (2016)

[4] D.S. Chaplot, G. Lample, *Arnold: An Autonomous Agent to play FPS Games,* Carnegie Mellon, AAAI. (2017)

[5] Restoring from checkpoints are broken in TF 1.13.1 S. Hochreiter, J. Schmidhuber, *Long Short-Term Memory,* Neural Computation archive Volume 9 Issue 8, Pages 1735-1780 (1997)

[6] K. He, X. Zhang, S. Ren, J. Sun *Deep Residual Learning for Image Recognition,* Microsoft Research, arXiv preprint arXiv:1512.03385 (2015)

[7] V. Mnih et al, *Human-level control through deep reinforcement learning,* Deepmind, doi:10.1038/nature14236 (2015)

[8] M. Kempka et al, *ViZDoom: A Doom-based AI Research Platform for Visual Reinforcement Learning,* Poznan University of Technology, arXiv:1605.02097v2 (2016)

[9] M. Abadi er al, *TensorFlow: A system for large-scale machine learning,* Google, OSDI (2016)

[10] D.P. Kingma, J.L. Ba, *ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION,* arXiv:1412.6980v9 (2017)

[11] X. Glorot, Y. Bengio *Understanding the difficulty of training deep feedforward neural networks,* University of Montreal, AISTATS (2010)

[12] Allen Lavoie, *Restoring from checkpoints are broken in TF 1.13.1,* Google, (2019)

[13] V. Mnih et al, *Playing Atari with Deep Reinforcement Learning,* Deepmind, Nature (2015)