Improving the Efficiency of FPS Gaming Bot with Deep Reinforcement Learning

Love Kush Pranu B191184EE[#1] Raj Narayan B191076EE[#2]

R.M.Shaizal B190949EE[#3]

Artificial Neural Networks and Fuzzy Logic Systems Course Project(EE3026D), National Institute of Technology Calicut

Abstract

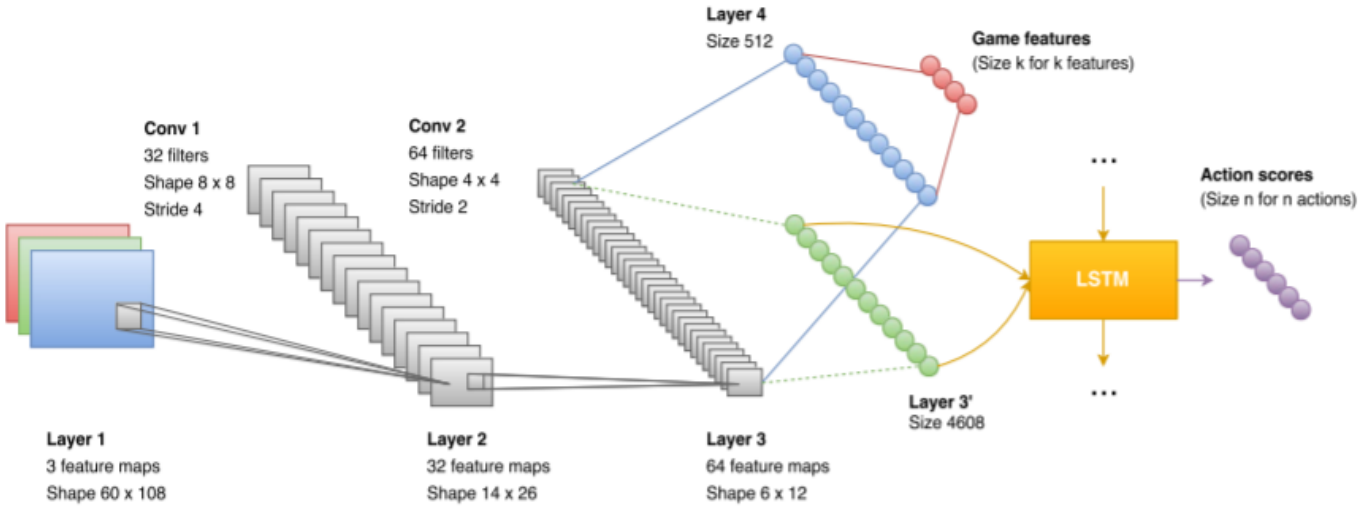
New advancements in deep reinforcement learning and artificial intelligence have made it such that bots can now outperform almost every single human player. However, most of these games take place in 2D environments that make it easier to differentiate between pixels. In this paper, we shall try to improve the efficiency in a 3D environment such as a first-person shooter game. Standard deep learning methods use visual inputs for training but since we have access to the source code files we can exploit them for game feature information such as enemies or items during the training phase. We train the model simultaneously for all these features using Q-learning which is known to have high success rates for visual learning models. Our model structure also allows flexibility for various maps. We test the performance of the various scenarios by testing it in a deathmatch scenario versus other AI agents and players.

I. Introduction

Deep reinforcement learning has proved to be very successful in mastering human-level control policies during a big variety of tasks like visual perception with visual attention, high-dimensional robot control, and solving physics-based control problems. In particular, Deep QNetworks (DQN) is shown to be effective in playing simple 2D pixel games and more recently, in defeating world-class Chess players. However, there are limitations in the above applications in their assumptions of having the full knowledge of the current environment, which is usually not true in real-world scenarios. In the case of partial states, the learning agent needs to remember previous states in order to select the most optimal actions. Deep Q-networks have recently been used to manage partial states in deep reinforcement learning by incorporating recurrency. One group of people used a deep recurrent neural network, particularly a Long-ShortTerm-Memory (LSTM) Network, to learn the Q-function to play simple 2D games. Because of their capacity to recall information for an indefinite length of time, recurrent neural networks are useful in circumstances with partially visible states.

Figure 1: A screenshot of Wolfenstein.

Previous approaches were often used in 2D settings that had little resemblance to the actual world. In this paper, we tackle the task of playing an FPS(First Person Shooter) game like Wolfenstein that runs in a 3D environment. This task is much more challenging than playing most 2D arcade games as it involves a wide variety of skills, such as navigating through a map, collecting items, recognizing and fighting enemies, etc. Furthermore, the agent navigates a 3D environment in a first-person viewpoint, making the job more suited for real-world robotics applications.



In this paper, we present an AI agent for playing deathmatches in Wolfenstein using only the pixels

on the screen. The problem is divided into two phases by our agent:

- Navigation - collecting stuff and locating foes on the map
- Action - fighting enemies when they are observed, and using separate networks for each phase of the game.

In addition, the agent deduces high-level game information, such as the presence of adversaries on the screen, in order to determine its current phase and optimize its performance. We show that co-training highly improves the training speed and efficiency of the model. We show that the proposed architecture substantially outperforms built-in bots of the game as well as humans in deathmatch scenarios and we demonstrate the importance of all components of our model.

II. Selecting the Network

Considering this is a model that requires visual input we use Deep Q-Networks. At each time t, the network observes the screen at state s_t and chooses to do an action according to a probability of (phi) and gets the corresponding reward (rt). The goal of the network is to maximize the reward earned.

$$Rt = \sum \gamma^{t'-t} r_{t'}$$

The Q function for an action a ins state s is:

$$Q^{\pi}(s,a) = \mathbf{E}[Rt \mid s_t = s, a_t = a]$$

Now we need to maximize the value of Q-learning estimate E. Q* is the highest value we can expect.

$$Q^*(s,a) = \max_{\pi} \mathbf{E}[R_t \mid s_t = s\ ,\ a_t = a] = \max_{\pi} Q^{\pi}(s,a)$$

$$Q^*(s,a) = \mathbf{E}[r + \gamma \max_{a'} Q^*(s'\ ,\ a') \mid s,a]$$

In the above model, we observe the full screen s_t so another way to get more training data from the current screen buffer is by converting our system to a recurrent network by creating partially observable environments and the agent can only observe o_t of the environment at a time such that it is not enough to infer the full state of a system. An FPS game like Wolfenstein, where the agent field of view is limited to 90 centered around its position, obviously falls into this category. To deal with such environments, we use Deep Recurrent Q-Networks (DRQN), which does not estimate $Q(s_t, a_t)$, but $Q(o_t, h_{t-1}, a_t)$, where h_t is an extra input returned by the network at the former step, that represents the retired state of the agent. An intermittent neural network like an LSTM can be implemented on top

of the normal DQN model to do that. Here, $h_t = LSTM(h_{t-1}, o_t)$, and we use that to estimate $Q(h_t, a_t)$.

III. Model

The first thing we did was apply a baseline DRQN model to solve the problem. Although this model achieved good performance in relatively simple scenarios (where the only available actions were to turn or attack), it did not perform well on deathmatch tasks(with health and ammo pots). The resultant operatives were firing at will, trying to catch a foe in their crosshairs. Giving a penalty for using ammo did not help: with a small penalty, agents would keep firing, and with a big one they would just never fire. Hence we try Game feature augmentation We know the agents were not able to accurately detect enemies. The VIZDoom environment gives access to the internal variables generated by 3D game systems. Since Wolfenstein also runs a similar environment we can use VIZDoom. We changed the game engine such that it now delivers information on the visible entities with every frame. As a result, the network receives a frame at each step, as well as a Boolean value for each entity indicating whether or not that entity is in the frame (an entity can be an enemy, a health pack, a weapon, ammo, etc). This internal information is not available during the test, but it can be used during training. To include this information and make it responsive to game aspects, we redesigned the DRQN architecture. In the initial model, the output of the convolutional neural network (CNN) is given to an LSTM that predicts the score for each action based on the current frame and its hidden state. We linked the output of the CNN to two fully-connected layers of size 512 and k, where k is the number of game characteristics we wish to detect. The cost of the network during training is a mix of the standard DRQN cost and the cross-entropy loss. It's worth noting that the LSTM only receives the CNN output as input and is never given the game features directly. We only employed one indicator of the existence of adversaries on the current frame, despite the fact that there was a lot of game information accessible. Adding this game feature dramatically improved the performance of the model in every scenario we tried.

In our experiments, it only takes a few hours for the model to reach an optimal enemy detection accuracy of 87%. After that, the LSTM is given features that often contain information about the presence of enemies and their positions, resulting in accelerated training. Augmenting a DRQN model with game features is straightforward. However, the strategy described above is difficult to apply to a DQN model. Indeed, sharing convolution filters between predicting game characteristics and the Q-learning objective is a key element of the model. The DRQN is well-suited to this situation since it only receives a single frame as input and must forecast what will be seen in that frame. In a DQN model, however, the network gets k frames at each time step and must anticipate if particular characteristics occur only in the final frame of the content of the $k - 1$ previous frame.

We divided the deathmatch challenge into two parts: the first includes exploring the map for things and foes, and the second entails killing them. We call these phases the navigation and action phases. Having two networks work together, each trained to act in a specific phase of the game should naturally lead to better overall performance. Current DQN models do not allow for the combination of different networks optimized for different tasks. The current game phase, on the other hand, maybe identified by anticipating whether an adversary is visible in the current frame during the action phase or not during the navigation phase, which can be deduced directly from the game elements available in the suggested model architecture.

Splitting the job into two parts and training a distinct network for each step provides a number of advantages. For starters, this makes the architecture modular and enables for independent training and testing of multiple models for each step. Both networks can be trained in parallel, which makes the training much faster as compared to training a single network for the whole task. Furthermore, the navigation phase only requires three actions (move forward, turn left and turn right), which dramatically reduces the number of state-action

pairs required to learn the Q-function, and makes the training much faster. More importantly, using two networks also mitigates camping, that is the tendency to stay in one area of the map and wait for enemies, which is exhibited by a couple of players and seemed like it might be a possibility for the agent. For the action network, we utilized a DRQN enhanced with game elements, and for the navigation network, we used a plain DQN. At each phase of the evaluation, the action network is invoked. If no enemies are identified in the current frame, or if the agent is out of ammunition, the navigation network is summoned to determine the next course of action. Otherwise, the action network is tasked with making the choice.

IV. Training

We reward the agent based on the number of kills minus the number of suicides. If the reward is only dependent on the score, the replay table for state-action combinations with non-zero rewards is exceedingly sparse, making it extremely difficult for the agent to learn favorable behaviors. Furthermore, awards are incredibly slow to arrive and are seldom the consequence of a particular action: obtaining a good reward necessitates the agent exploring the area in search of an adversary and precise aiming and shooting it with a handgun. Because of the delay in reward, the agent has a hard time determining which set of behaviors is accountable for which reward. We present reward shaping, which is the modification of the reward function to incorporate short intermediate incentives to speed up the learning process, to address the problem of sparse replay tables and delayed rewards. We propose the following intermediate rewards for customizing the reward function of the action network, in addition to positive rewards for kills and negative rewards for suicides:

- the positive incentive for picking up an object
- the penalty for losing one's health
- the penalty for shooting or running out of ammunition.
- We don't have to worry about firing at teammates as this is deathmatch mode.

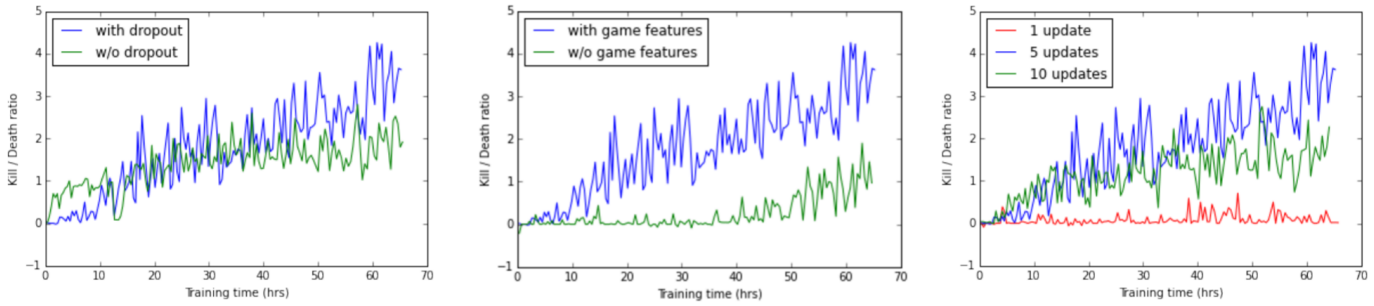
We used different rewards for the navigation network. We just give it a positive reward when it picks up an item because it matures on a map without adversaries and its aim is to accumulate objects, and a negative reward when it runs into an explosive. We also found it very helpful to give the network a small positive reward proportional to the distance it traveled since the last step. That way, the agent is faster to explore the map and avoids turning in circles. To decrease the time taken for training we use the frame-skip technique. The agent only gets a screen input every k + 1 frames in this manner, where k is the number of frames skipped between each step. The network's decision is then repeated for all of the skipped frames. A faster frame-skip rate speeds up training but degrades performance. Aiming at an enemy usually necessitates rotating a few degrees, which is difficult when the frameskip rate is too high, even for human players, because the agent will repeat the rotate operation numerous times, rotating more than it planned to. A frameskip of k = 4 turned out to be the best trade-off.

Evaluation Metric	Human	Agent
Number of objects	5.2	9.2
Number of kills	12.6	27.6
Number of deaths	8.3	5.0
Number of suicides	3.6	2.0
K/D Ratio	1.52	5.12

Table 1: Comparison of human players with agents.

V. Scenario

We use the VIZDoom platform to conduct all our experiments and evaluate our methods on the deathmatch scenario as it is perfect for grading 3D FPS games such as Wolfenstein. In this scenario, the agent plays against built-in bots, and the final score is the number of bots killed by the agent minus the number of suicides committed. The agent is trained and evaluated on the same map, and the only available weapon is a basic pistol as it is



simple and accurate. agents can gather health packs and ammo. The model is trained and tested on different maps. The limited deathmatch task is ideal for demonstrating the model design effectiveness and for choosing hyperparameters, as the training time is significantly lower than on the full deathmatch task. In order to showcase the generalizability of our model, we can use the deathmatch mode to show that our model also works effectively on unknown maps as well.

VI. Metrics

All networks were trained using the RMSProp algorithm and mini-batches of size 32. Network weights were updated every 5 steps, so experiences are sampled on average 8 times during the training. The one million most recent frames were stored in the replay memory. $\gamma = 0.99$ was used as the discount factor. We used a greedy policy during the training, which was linearly decreased from 1 to 0.1 over the first 1,00,000 steps, and then fixed to 0.1. Different screen resolutions of the game can lead to different fields of view. In particular, a 16/9 resolution in Wolfenstein has a 108-degree f.o.v (as presented in Figure 1). We selected a resolution of 440x225 to increase the agent game awareness. The kill to death (K/D) ratio is used as a scoring tool in deathmatch situations. We also report the number of kills to see if the model can roam the map to find foes, as the K/D ratio is sensitive to "camper" behavior to reduce fatalities. In addition to these, we also report the total number of objects gathered, the total number of deaths, and a total number of suicides. Suicides are caused when the agent stays in the same location for too long. Since suicides are counted in deaths, they provide a good way for penalizing the K/D score when the agent is shooting arbitrarily.

	Known Map	
Evaluation Metric	Without navigation	With navigation
Number of objects	14	46
Number of kills	167	138
Number of deaths	36	25
Number of suicides	15	10
Kill to Death Ratio	4.64	5.52

Table 2: Performance of the agent against in-built game bots with and without navigation in only the base map.

	Train maps		Test maps	
Evaluation Metric	Without navigation	With navigation	Without navigation	With navigation
Number of objects	52.9	92.2	62.3	94.7
Number of kills	43.0	66.8	32.0	43.0
Number of deaths	15.2	14.6	10.0	6.0
Number of suicides	1.7	3.1	0.3	1.3
Kill to Death Ratio	2.83	4.58	3.12	6.94

Table 3: Performance of the agent against in-built game bots with and without navigation in Train and Test maps.

VII. Results

Here you may see navigation and deathmatch demonstrations on both known and unknown maps. Enhancement of the navigation network. Tables 2 and 3 display the results of both the navigation and non-navigation activities. The agent was tested for 60 minutes on all of the maps, with the results averaged across all of the deathmatch maps. In a complete deathmatch, the agent starts with a handgun, which makes killing adversaries tough

unless the agent goes for the head. Therefore, headshots and collecting ammo is much more important in the full deathmatch, which explains the larger improvement in the K/D ratio in this scenario. Because the map was tiny and there were numerous bots, navigation was not necessary to discover other agents in the restricted deathmatch scenario. With navigation, however, the agent was able to pick up over three times as many goods, such as health packs and ammunition. The agent's ability to repair itself on a regular basis reduced the frequency of deaths and enhanced the K/D ratio.

The performance on the test maps is better than on the training maps, which is not surprising considering how diverse the maps all look. In particular, the test maps contain fewer stairs and differences in level, which are usually difficult for the network to handle since we did not train it to look up and down. Comparison to human players. In both single-player and multiplayer settings, Table 1 demonstrates that our agent outperforms human gamers. In the single-player scenario, human players and the agent compete for three minutes against ten bots on a restricted deathmatch terrain. Over the course of 20 players, human scores are averaged. It's worth noting that humans have a relatively high suicide rate, implying that aiming effectively in a limited response time is difficult for humans.

Detecting opponents is crucial to our agent's success, but it's not an easy process because attackers might arrive from varied distances, perspectives, and situations. The model's performance was significantly improved when game characteristics were included during training. The best K/D score of the network without game elements is less than 2.0 after 65 hours of training, but the network with game features may obtain a maximum value of almost 4.0. Another benefit of employing game features is that it provides rapid feedback on the quality of the convolutional network's features. The LSTM will not get important information about the presence of adversaries in the frame if the enemy detection accuracy is low, and the Q-learning network will struggle to learn a decent policy. It takes a few hours for the adversary detection accuracy to converge, whereas training the entire model might take up to a week. Our design allows us to easily modify our hyperparameters without having to retrain the whole model since adversary detection accuracy corresponds with final model performance. For example, the accuracy of foe detection with and without dropout soon converged around 0.80, allowing us to conclude that dropout is critical for the model's effective performance. The above graphs back up our conclusion that employing a dropout layer increases the action network's performance on the limited deathmatch.

VIII. Conclusion

In this paper, we have presented a complete architecture for playing deathmatch scenarios in FPS games. We developed a way for supplementing a DRQN model with high-level game data and modularized our architecture to include autonomous networks responsible for distinct game stages. When applied to complex tasks like a deathmatch, these approaches produce significant gains over the basic DRQN model. We proved that the suggested model can outperform both built-in bots and human gamers, as well as the model's generalizability to unknown landscapes. Furthermore, our approaches are compatible with current DQN advancements and might be simply integrated with dueling structures and prioritized replay.

Code for the assignment

https://github.com/lutherleo/Wolfenstein-bot

References

[1] Ba, Mnih, and Kavukcuoglu 2014 Ba, J.; Mnih, V.; and Kavukcuoglu, K. 2014. Multiple object recognition with visual attention..

[2] Bellemare et al. 2012 Bellemare, M. G.; Naddaf, Y.; Veness, J.; and Bowling, M. 2012. The arcade learning environment: An evaluation platform for general agents.

[3] Journal of Artificial Intelligence Research.

[4] Chaplot and Lample 2017 Chaplot, D. S., and Lample, G.

[5] 2017. Arnold: An autonomous agent to play fps games. In

[6] Thirty-First AAAI Conference on Artificial Intelligence.