Book Section:
Ratcliffe, Dino, Devlin, Sam orcid.org/0000-0002-7769-3090, Kruschwitz, Udo et al. (1 more author) (2017) Clyde: A deep reinforcement learning DOOM playing agent. In: What's Next For AI In Games.

Clyde: A deep reinforcement learning DOOM playing agent

Dino S. Ratcliffe
University of Essex
Colchester, England
dratcl@essex.ac.uk

Sam Devlin
University of York
York, England
sam.devlin@york.ac.uk

Udo Kruschwitz
University of Essex
Colchester, England
udo@essex.ac.uk

Luca Citi
University of Essex
Colchester, England
lciti@essex.ac.uk

Abstract

In this paper we present the use of deep reinforcement learning techniques in the context of playing partially observable multi-agent 3D games. These techniques have traditionally been applied to fully observable 2D environments, or navigation tasks in 3D environments. We show the performance of Clyde in comparison to other competitors within the context of the ViZDOOM competition that saw 9 bots compete against each other in DOOM death matches. Clyde managed to achieve 3rd place in the ViZDOOM competition held at the IEEE Conference on Computational Intelligence and Games 2016. Clyde performed very well considering its relative simplicity and the fact that we deliberately avoided a high level of customisation to keep the algorithm generic.

1 Introduction

Game playing in artificial intelligence (AI) has often been used as a method for benchmarking agents (Yannakakis and Togelius 2015; Mnih et al. 2015; Schaeffer et al. 2007; Silver et al. 2016). In many cases games provide noise free environments and can also encompass the whole world state in data structures easily. Much of the early work in this domain has focussed on digital implementations of board games, such as *backgammon* (Tesauro 1995), *chess* (Campbell, Hoane, and Hsu 2002) and more recently *go* (Silver et al. 2016). These games have then been used to benchmark many different approaches, including tree search approaches such as Monte Carlo Tree Search (MCTS) (Browne et al. 2012) along with other approaches such as deep reinforcement learning (Silver et al. 2016). They also provide a method for allowing comparisons to human performance by playing the agents against experts in their respective games.

More recently video games have started to be used in order to benchmark these systems, such as the Arcade Learning Environment (ALE) that allows agents to train on a variety of Atari 2600 games (Mnih et al. 2013). This brought rise to a form of reinforcement learning that learns how to play the games by using the image on the screen as the game state. These systems have shown above human performance in many of these games and competent play in most of the others (Mnih et al. 2015). This is an exciting area of AI as it puts the player and the AI agents on the same playing field, especially when it comes to partially observable games.

The majority of the games that have been used to test these AI agents have been in fully observable environments, meaning that the player has all the information needed to determine the entire state of the game, for example knowing where all the pieces are on a chessboard. In contrast, this work looks at the viability of some of these systems working in 3D partially observable games. DOOM is a 3D First Person Shooter (FPS) game from the early 1990s (Kempka et al. 2016). This game provides partial observability through the use of the first person perspective. DOOM provides a nice environment for this as it is not a fully 3D environment, the game takes place in a 3D environment, however the objects within the world are still 2D sprites, such as the enemies and the pick-ups in the world. This gives agents the challenge of navigating a 3D partially observable environment but makes it easier to identify objects within the scene due to the limited number of sprites that can be used per object.

VizDOOM[1] is a platform developed by the institute of computer science at Poznan University. It provides an interface for AI agents to learn from the raw visual data that is produced by DOOM (Kempka et al. 2016). They also run a competition that places these agents into death matches in order to compare their performance. A death match in the case of this competition is a time limited game mode where each agent must accumulate the highest score possible by killing other agents in the match. This is where our agent was submitted in order to assess its performance against other agents.

Our work focuses on the use of deep reinforcement learning agents in this multi-agent 3D environment. We are focussed on the performance of these neural network based reinforcement learning systems in this domain without tailoring the systems to this specific problem. This distinction is important as the main focus of our work is not to train algorithms to perform well in a specific game but instead to devise algorithms that are aimed at general game playing.

This paper will first ground our work by outlining related work in the field of reinforcement learning. We will then cover the setup of the competition as outlined by the organisers. We will give a detailed description of Clyde and how

[1] http://vizdoom.cs.put.edu.pl/competition-cig-2016

the agent was trained, this will be followed by the results and discussion of our agent's performance. We will conclude by outlining our contribution and the work we would like to perform in the future in relation to Clyde.

2 Related Work

2.1 AI and Games

Games provide a good benchmark for AI as they require high level reasoning and planning (Yannakakis and Togelius 2015; McCoy and Mateas 2008). This means that they have often been used to signify advances in the state of the art such as with Deep Blue the *chess* playing AI that beat Gary Kasparov (Campbell, Hoane, and Hsu 2002) and AlphaGo the *go* playing agent that beat Lee Sedol (Silver et al. 2016). Games also provide a nice interface for agents to be able to either look into the internal state of the games, as needed for methods such as MCTS (Perez-Liebana et al. 2016), or can provide visual representations of state that can be used by agents (Bellemare et al. 2012). Games also allow agents to process experiences much faster than would be possible in the real world. This means data hungry methods are still able to learn in a relatively short period of time (Karakovskiy and Togelius 2012).

There has also been a lot of research into other areas of game playing, such as competitions aimed at agents passing a form of *Turing Test* in order to rank agents based on their ability to mimic human play (Hingston 2009). Most of these systems have revolved around using data from human play in order to achieve this goal (Polceanu 2013; Karpov, Schrum, and Miikkulainen 2013). There have also been competitions organised in order to assess agents ability to generalise across a variety of games, the General Game Playing Competition (Genesereth, Love, and Pell 2005) focusing on playing multiple board games, and the General Video Game Playing Competition (Perez et al. 2015) that assess agents performance across a broad range of video games. Other work has also started that uses modified versions of game engines in order to provide environments to teach AI, these include Project Malmo that uses the popular Minecraft Game in order to provide an experimentation platform for AI agents (Johnson et al. 2016), and OpenArena, a modified version of the ID Tech 3 engine used by Quake 3, in order to train and benchmark AI at a range of tasks including path finding in a Labyrinth and laser tag (Jaderberg et al. 2016).

There has also been some work in using neural evolution to evolve a network to control an agent in an FPS environment (Parker and Bryant 2012). Others have investigated the use of hierarchical approaches in the design of AI agents for FPS games (Van Hoorn, Togelius, and Schmidhuber 2009) that use networks in a hierarchical fashion to deconstruct the tasks into sub-skills.

2.2 Reinforcement Learning

Reinforcement learning is an area of machine learning that focuses on the use of rewards in order to train agents how to act within environments. Reinforcement learning provides a structure where the agent is in an environment in which it can take actions, then make observations and receive rewards. These rewards can then be used in order to improve the agent's policy for picking actions in the future, with the aim of maximising the rewards it receives.

Temporal Difference (TD) is a technique that has had a large impact in reinforcement learning (Kaelbling, Littman, and Moore 1996). It gives a method for predicting future reward through learning from experience. TD is written as

$$V_{s_t} = r_t + \gamma V_{s_{t+1}} \tag{1}$$

This means that the value V of state s at time t is the reward r you receive when leaving the state plus the future discounted reward you could expect from the state s_{t+1} discounted by γ. TD learning is the basis for many other reinforcement approaches (Watkins and Dayan 1992; Konda and Tsitsiklis 1999), one of the earlier methods was TD-lambda that was used to train a neural network to play backgammon (Tesauro 1995).

Q-Learning is a reinforcement learning method that builds on TD learning. It works by learning an action-value function $Q(s, a)$ that represents the expected sum of discounted rewards that will be received if action a is taken in state s, allowing for a greedy policy to be used in order to select the action that gives the best combination of short term and long term reward (Watkins and Dayan 1992).

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot \Big(r_{t+1} + \gamma \cdot max_a Q(s_{t+1}, a) - Q(s_t, a_t) \Big) \tag{2}$$

where α is the learning rate at which the agent will adjust its estimation of Q and γ is the discount factor that reduces the future rewards of states that are further away.

Actor-Critic methods take a different approach to Q-learning by splitting the agent into two separate entities. The actor learns a policy which is a vector of probabilities for selecting a specific action. The critic is then a more traditional state-value function that is used to guide the learning of the policy. The critic evaluates the TD error δ to assess whether the action chosen performed better or worse than expected (Konda and Tsitsiklis 1999). This delta value is calculated as follows.

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{3}$$

This is then used to evaluate the action that was just taken, if the action produced a better reward than expected the probability of choosing that action should be increased. However if the reward was less than expected the probability $p(s_t, a_t)$ of choosing action a_t should be reduced. The actor could be updated with:

$$p(s_t, a_t) \leftarrow p(s_t, a_t) + \beta \delta_t \tag{4}$$

where β is another scaling factor for how quickly to adjust the policy. This is one of the simpler forms of actor critic methods with other approaches introducing more factors into the calculation such as the entropy of the current policy (Williams and Peng 1991).

There are two main advantages to actor critic over value function methods, one is that it takes minimal amount of computation in order to select actions. There is no need to calculate the value of Q for every action in the action space before picking the action with the highest value. The other advantage is that they can learn explicitly stochastic policies, meaning it can learn the optimal probabilities for which action to pick. One example of this is that actor critic methods could learn the optimal policy for the game rock, paper, scissors. It would learn that the optimal policy is to pick each action one third of the time, this is the optimal if you are not modelling the decisions of your opponent (Niklasson, Engstrom, and Johansson 2001). Whereas a value function method with a greedy policy would constantly pick the same action depending on the experiences it had during training.

2.3 Deep Reinforcement Learning

There have already been a few systems that tackle the problem of learning to play games from the screen buffer only, as will be discussed below all of these systems rely on the use of deep neural networks in order to process the images and learn which action produces the best possible outcome for the agent. Deep neural networks have risen in popularity, this is possibly due to the increase in computational power through the use of faster central processor units (CPU) and graphics processor units (GPU) (Coates et al. 2013). This increase in computation allows the networks to be fed a larger amount of data in shorter amounts of time making them more viable for a whole range of tasks.

Deep Q Networks (DQN) rely on many of the advancements in neural networks and Q-learning (Mnih et al. 2013). They have been used to obtain better than human performance in a variety of Atari games (Mnih et al. 2015). Deep Q networks have the advantage of only relying on using images of the game being played and the reward they receive from playing, meaning the network only has as much information about the game as a human player. The way these networks achieve this is by first using convolutional layers to process the input image, and then having fully connected layers after these that output the Q value for each action. The specific form of Q-Learning used is $\epsilon - greedy$ where epsilon is an explore vs exploit parameter that dictates when the policy is followed to exploit and when random actions are selected in order to explore the environment. Being based on traditional Q-Learning techniques these networks effectively treat each frame as a state and then give estimated Q values for the states the agent would end up in if it took a specific action. This is an improvement over older Q networks that would require a feedforward pass for each action (Mnih et al. 2013). Other techniques used in these networks are the stacking of images that are input to the network, this allows the network to distinguish between states that may look the same in a single frame but are different due to the movement of objects in the environment (Mnih et al. 2013). They also incorporate an experience replay mechanism allowing the agent to take a random batch of experiences, in the form of (state, action, reward, resulting-state, termination) from a buffer, this allows the agent to learn much faster by learning off experiences multiple times and also helps the network to converge by breaking up the experiences (Mnih et al. 2013). A limitation of these networks is the fact that they do not handle complex reward systems effectively. This means the performance of the network is poor for games such as Montezuma's Revenge, where rewards are given after a long sequence of intelligent moves (Schaul et al. 2016). On the other hand in reactionary games such as breakout the network manages to learn high level tactics such as digging a tunnel in order to get the ball behind the bricks (Mnih et al. 2015).

Recurrent Neural Networks (RNN) work by having outputs from neurons that feed back into themselves in the next forward pass through the network, this allows the network to pass information between timesteps, effectively giving the network some limited memory (Bengio, Simard, and Frasconi 1994). The simplest form of recurrent neural network simply treats this as another weighted input at each timestep, these can then be updated like traditional networks using backpropagation through time, where the recurrent neural network is effectively unrolled a set number of timesteps that allows the recurrent weights to be updated. These networks suffer from the *vanishing gradient problem*, this has the effect of the network not being able to remember for many timesteps making dependencies over long distances impossible to learn (Hochreiter 1998).

A more complex version of a recurrent neural network is a LSTM (Long Short Term Memory) network, these use cells that add a layer on top of regular RNNs. These cells not only have recurrent links but also have gates that allow it to decide what information to store in the cell and abilities to forget information. This negates the vanishing gradient problem because the network can make a decision to forget information and when to keep information, improving its ability to remember over a greater number of timesteps (Hochreiter and Schmidhuber 1997; Sutskever, Vinyals, and Le 2014).

Asynchronous Advantage Actor-Critic (A3C) is a method for using actor-critic reinforcement learning methods along with deep reinforcement learning. There are a few different intricacies to A3C networks, the first is the fact that the value and policy are calculated by two separate networks. Although in theory they are two different networks, in practice it is possible for the two networks to share lower level layers, that includes all the convolutional layers and any LSTM layers that the model may have. As it is an actor-critic method the policy is learnt online. Meaning the policy is followed during training at all times and there is no need for a explore vs exploit parameter. Instead one of the tricks used by A3C is to add the entropy of the policy to the error rate, this tries to force the network towards low entropy until it finds a policy that produces high rewards. A3C is also an asynchronous method, that allows the parallelisation of training across multiple CPUs and GPUs. This is accomplished by having a global network and then for each CPU core a separate thread that has a local network. Each thread also has an instance of the MDP that the agent can learn from. Once a set of experiences have been acquired on a particular thread then the gradients are calculated on the local network in order to then update the global network. Once the global network has been updated a new copy is taken by the local network. This means that A3C can train entirely on CPUs and does not require GPUs to attain acceptable training times (Mnih et al. 2016).

As stated earlier A3C maintains both a policy $\pi(a_t|s_t;\theta)$, which outputs the probability at which action a_t should be selected when in state s_t, given a certain network θ, and a value network $V(s_t;\theta_v)$. That gives the value of state s_t given a certain network θ_v. Though the networks θ and θ_v are treated as separate networks in practise they can share the lower layers of the network. These networks are then updated every t_{max} number of actions or at the stage a terminal state has been reached before t_{max}. The error gradients for the network are calculated as shown below:

$$\nabla_{\theta'} \log \pi(a_t|s_t;\theta')(r_t - V(s_t;\theta_v)) + \beta \nabla_{\theta'} H(\pi(s_t;\theta')) \quad (5)$$

where $\pi(a_t|s_t;\theta')$ is the output of the policy network, r_t represents the actual reward received by the agent, $V(s_t;\theta_v)$ is the estimated reward from the value network, H is the entropy and β is the strength of the entropy regularization term. The action taken is factored by the amount of error in the prediction of the value of the state. This has the effect of increasing the probability of actions that it underestimates the value of the states they produce, and reduces the probability of actions that it overestimates the value of the states. The term H is the entropy of the current policy, this has been shown to find better policies by increasing exploration and preventing the system from converging on a sub optimal policy too early (Williams and Peng 1991).

This method negates many of the tricks required by the DQN technique that are there to improve the convergence of those systems. The fact that the local threads take a copy of the global network prevents the need for network freezing, where old copies of the network are used for the error calculation. Having multiple threads all updating the global network with different experiences also means there is no need for experience replay. A3C has been shown to perform better than DQN approaches in a variety of Atari 2600 games (Mnih et al. 2016).

2.4 DOOM/VizDOOM

DOOM is a first person shooter game that was released in 1993. It was a game that pioneered 3D graphics and online multiplayer. VizDOOM is a platform that wraps around the DOOM game, giving agents access to the screen buffer and the ability to perform actions within the game. It does this by providing an interface to the game and allowing you to specify certain attributes through config files. This includes things like screen buffer resolution and the set of actions an agent can make in the game (Kempka et al. 2016).

3 Experimental Setup

Our system was submitted to Track One of the competition held at the IEEE Conference on Computational Intelligence and Games 2016. Track One means that the agent would play on a known map with only access to the rocket launcher weapon. The competition would be ranked based on the number of "frags" each agent achieved. Frags are incremented when an agent kills another agent and then decremented when an agent commits suicide. Deaths from other agents do not affect an agent's frag count although it would give competitors an advantage as their frag count would be incremented.

Competitors were able to access the screen buffer and through a config file specify what was contained in this image, such as rendering a crosshair, specifying a resolution, limiting the rendering of the Heads Up Display (HUD) and other objects such as particles. This file could also be used in order to specify what actions were available to the agent, and what game variables were available including health and ammunition values.

The agents would be run on PCs with access to dedicated GPUs for computation, the specifics of the computers used are Intel® Core™ i7-4790 CPU @ 3.60GHz and an Nvidia® GTX 960 4GB

The competition was run over 12 ten minute rounds, due to the fact VizDOOM can only support 8 players per round and there were 9 entrants to Track One, each bot would not participate in one of first 9 rounds. For the final 3 rounds all bots that had $frags > 0$ would be used.

4 Method

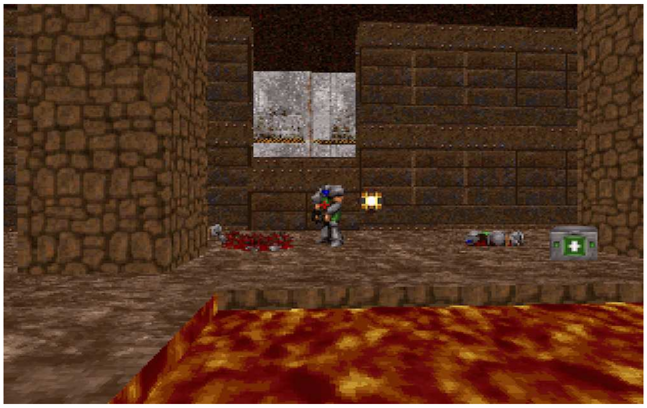

Figure 1: Screen buffer presented to network (in colour)

We decided to use A3C as the base for Clyde, this was due to the fact that A3C was showing the current state of the art performance on the ALE environment. It has also shown good results in 3D based environments such as TORCS and Labyrinth a 3D benchmarking environment developed by Google DeepMind (Mnih et al. 2016). A3C also allows very easy integration of recurrent cells such as LSTM cells. The LSTM cells were used for a variety of reasons. It allows the agent to remember locations of objects that have just left the

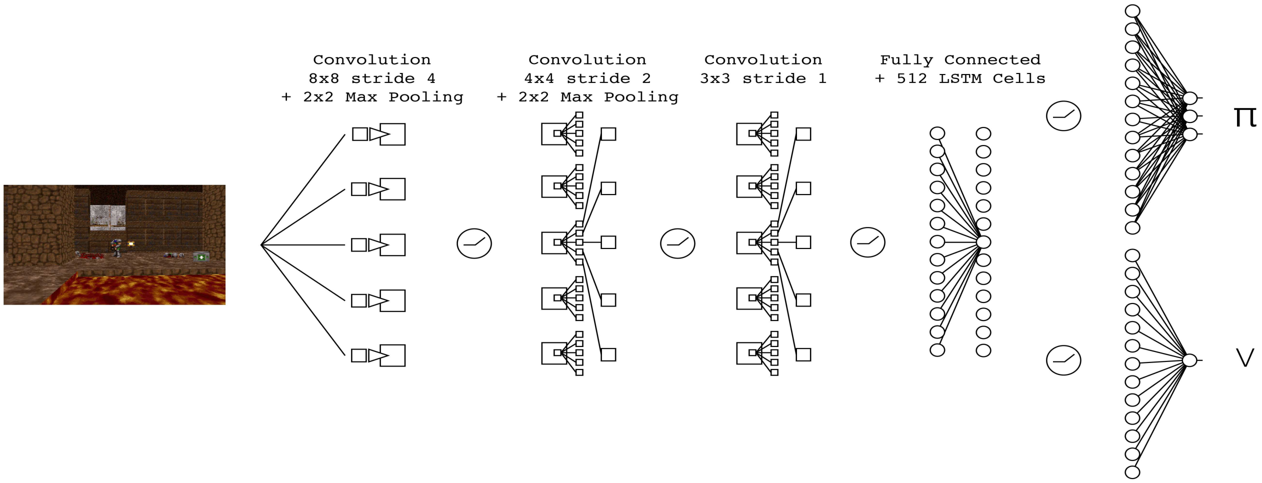

Figure 2: Clyde Architecture

screen buffer, such as when collecting a pick-up that scrolls off the bottom of the screen before that agent walks over it.

Clyde was very closely based on the networks used for these Atari playing A3C agents. There were however some changes to the layers of the network in order to accommodate for the larger images sizes we were using within VizDOOM.

The screen buffer resolution that was used was 320 by 256 pixel images. This was deemed to be the best compromise between small images that could be processed quickly that still allowed for enemies to be identified from a distance. Colour images were used in order to allow the network to more accurately distinguish between items. All of the UI was turned off in order for the agent to see as much of the game world as possible, with only the cross-hair being turned on in order to aid with aiming as shown in Figure 1. This configuration meant the inputs to the network were single RGB images resulting in tensors of shape 320x256x3.

The available actions that we gave to the agent were as follows:

- Move Forward
- Shoot
- Turn 1 degree right
- Turn 1 degree left
- Turn 20 degrees right whilst moving forward
- Turn 20 degrees left whilst moving forward

VizDOOM allows us to combine actions to perform more than one action per timestep, along with turning more than 1 degree in order to simulate mouse movement. This is how we managed to provide actions to the network that allowed it to turn 20 degrees in one action whilst also taking a step forward. We wanted to keep the action space small and also provide a method for the agent to be able to keep moving forward whilst navigating corners, this is why the 20 degree movements also included moving forward. Each action is then performed at a time step and then a new frame is received from the VizDOOM framework.

Clyde consists of an initial convolutional layer with kernel sizes of 8x8 and a stride length of 4, they then output 16 channels. All of the neurons within the network use the ReLU (Rectified Linear Unit) activation function. This first convolutional layer is then followed by 2x2 max pooling that halves the height and width of the resulting images. The 16 channels are then fed into another convolutional layer with a kernel size of 4x4 with a stride of 2, this layer outputs 32 channels. Again this is followed by a 2x2 max pooling operation. The final convolutional layer has a kernel size of 3x3 and a stride of 1 which outputs 64 channels. These channels are then flattened and fed into a fully connected layer, this fully connected layer is then fed into a layer of 512 LSTM cells. All of the previous layers are shared by the value and policy network, but at this point they split into two separate layers as shown in Figure 2. One layer for the value network that outputs the estimated value for the state, and another separate layer for the policy network that outputs a probability for each action. The experiences were a batch of 32 continuous frames allowing the update to the network to unroll 32 time steps and perform backpropagation through time on these inputs. That allows the LSTM layers to remember details about previous frames and use this information to assess the current state the agent is in.

The error calculation for the system was again exactly the same as previous A3C agents that were trained on Atari games. As shown in Equation 5. In the case of Clyde $\beta = 0.1$ through preliminary experimentation this was found to produce the best results. The value network is simply updated with the least square error between the estimated value of the state and the actual value. These experiences where not batched during training, instead the network was updated after every experience resulting in a batch size of 1.

The network weights are updated using Root Mean Square Propagation (RMSProp), we use the centred version of RMSProp that shares the moving average of element-wise gradients g as used in the original paper introducing the A3C algorithm (Mnih et al. 2016). The learning rate of the network was set to 10^{-5} with it being reduced linearly to 0 over the course of 80 million experiences. 80 million experiences was also something that was chosen through preliminary experimentation.

The reward structure for Clyde was heavily influenced by the amount of frags that the agent received during training. The various rewards are shown in Table 1.

Table 1: Reward Structure

Task	Reward
Kill Enemy	+10
Suicide	-10
Collect Health	+1
Collect Armour	+1
Collect Ammo	+1

The rewards for killing the enemy and committing suicide fit perfectly with how frags are awarded throughout the course of a match. Small rewards were added in order to signify to the agent that collecting items such as armour and ammunition would be beneficial, it also has the effect of incentivising the agent to move around the map without the immediate presence of an enemy.

Table 2: PC Specifications

CPU	Intel® Core^{TM} i7-5930 CPU @ 3.50GHz
RAM	64GB DDR4
GPU	2x Maxwell Nvidia® Titan X 12GB Maxwell

The agent was trained on 2 minute matches against the built-in DOOM bots, it was trained for 30 million time steps with 12 threads. Everything was calculated on the CPU cores apart from the gradient calculations that were performed on the GPU. The training for this agent took around 27 hours on a powerful desktop PC as specified in Table 2.

5 Results

Table 3: Competition Results

Place	Bot	Total Frags
1	F1	559
2	Arnold	413
3	CLYDE	393
4	TUHO	312
5	5Vision	142
6	ColbyMules	131
7	Abyssll	118
8	WallDestroyerXxx	-130
9	Ivomi	-578

As stated earlier the competition was run over 12 matches 10 minutes in length, with bots that had $frags < 0$ at the end of the first 9 rounds not being in the final three rounds. In the case of this competition it meant that bots *WallDestroyerXxx* and *Ivomi* would not participate in the last 3 rounds. The results of the competition are shown in Table 3.

6 Discussion

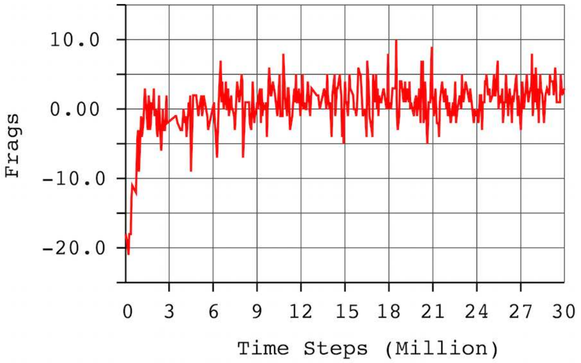

Figure 3: Number of frags achieved during training, run on 2 minute rounds

As shown in Figure 3 the network quickly converges onto a strategy that dramatically reduces the number of suicides, as the frag count dramatically rises from a large negative value. This is not necessarily a good thing as it indicates the network is committing to a strategy very early and then simply refining it over a long period of time. This is most probably a problem with selection of the hyper parameters, but due to time constraints of the competition these were the best results obtained before the submission deadline. Another issue that could have contributed to this result is sparse rewards within the domain. This could be solved by introducing extra reward shaping but would have meant incorporating domain knowledge into the training of the agent, this is a step we wanted to avoid given that our future focus is to increase the performance of these networks in general game playing. We expect an increase in performance if hyper parameters can be found that prevent this early convergence on a local optimum. There is a difference between the number of frags achieved during training (around 3 in two minute matches) and the results in the competition. This is most likely due to the differences in performance of the built in DOOM AI used to train the agent and the performance of the bots entered into the competition.

In Figure 4, which shows a sample of values given to states over the course of training, we can see that the value network also quickly increases the value of states and then they hold pretty steady throughout training. There does seem to be some divergence in the values towards the end of training that could indicate the network beginning to differentiate between good and bad states. However this is not necessarily the cause as the states are sampled at random.

When observing the agent it becomes obvious that the strategy taken by the agent is to keep shooting as often as possible without any regard for ammunition conservation. The learnt strategy also has no mechanism for conservation of life apart from making sure that it navigates the map in a

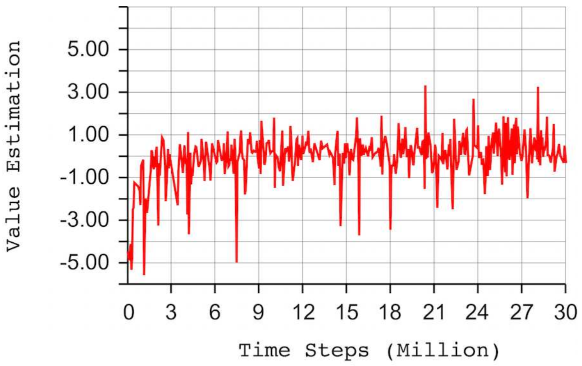

Figure 4: Value assigned to states during training, states sampled at random

manner that prevents suicides. Due to the fact that the current hyper parameters of the network allow the convergence on a simple strategy so early in training adding a negative reward for shooting simply means the network learns to never fire the weapon. This is due to the fact it doesn't explore other strategies enough to acquire an understanding of shooting enemies.

Despite our agent learning a relatively simple strategy it still performed well against more complex systems, such as the agent Arnold who relies on two separate networks for navigation and target acquisition, and also requires preprocessing of the image to indicate when items of interest are present on screen (Lample and Chaplot 2016). This is something we chose to avoid in order to not customise the system to one specific game and keep the system general.

The winner of the competition *F1* also uses A3C as the architecture for the agent (Wu and Tian 2016), however more domain specific knowledge is used in the training phase and during running of the agent. Extra data from the game is given to the network as input this includes the agents current health and available ammo. The more interesting difference though comes from how the agent was trained. A form of curriculum learning (Bengio et al. 2009) was used where the complexity of the task was slowly ramped up during trainings, this was done in a number of ways including the complexity of the map, the number of enemy bots and the strength of the enemy bots.

7 Conclusion

In this paper we have continued to use games as a benchmark for the performance of AI as has been so successful in the past. Utilising the frameworks and competitions developed and run by other researches. Allowing us to compare our agent to other AIs in a controlled competitive environment.

This work has shown the effectiveness of using A3C within the context of complex 3D multi-agent environments, with Clyde scoring competitively against other agents in the VizDOOM competition. This includes avoiding methods that could have improved performance but would limit the general game playing ability of the system.

We also believe that these systems could be applied to commercial games in order to provide competent opponents, without the need for hand coding behaviours.

8 Future Work

For future work we would like to begin with trying to tune the hyper parameters in order to obtain a system that does not commit to a strategy so early on in training. We believe that this could be achieved through a combination of the β factor of the entropy, and tuning the convolutional layers to better distinguish between frames. We would then like to combine this with self play in order to not overfit to the DOOM bots allowing the agent to learn from itself and become a more competent player.

Another area that we are looking to explore is the use of image preprocessing in a manner that is game independent. This includes running the game image through Laguerre filters (Mäkilä 1990), that can encode more data about previous states into the current images theoretically helping the network to take previous states into consideration when evaluating a current state.

Since the work in this paper new enhancements to A3C in the form of the UNsupervised REinforcment and Auxiliary Learning (UNREAL) agent have been made (Jaderberg et al. 2016). This agent learns auxiliary tasks in order to better guide the feature extraction of the lower layers of the network. We would therefore like to see the benefit of learning auxiliary tasks, especially combined with the use of Laguerre filters.

References

Bellemare, M. G.; Naddaf, Y.; Veness, J.; and Bowling, M. 2012. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*.

Bengio, Y.; Louradour, J.; Collobert, R.; and Weston, J. 2009. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, 41–48. ACM.

Bengio, Y.; Simard, P.; and Frasconi, P. 1994. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks* 5(2):157–166.

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4(1):1–43.

Campbell, M.; Hoane, A. J.; and Hsu, F.-h. 2002. Deep blue. *Artificial intelligence* 134(1):57–83.

Coates, A.; Huval, B.; Wang, T.; Wu, D.; Catanzaro, B.; and Andrew, N. 2013. Deep learning with COTS HPC systems. In Dasgupta, S., and Mcallester, D., eds., *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, volume 28, 1337–1345. JMLR Workshop and Conference Proceedings.

Genesereth, M.; Love, N.; and Pell, B. 2005. General game playing: Overview of the AAAI competition. *AI magazine* 26(2):62.

Hingston, P. 2009. A turing test for computer game bots. *IEEE Transactions on Computational Intelligence and AI in Games* 1(3):169–186.

Hochreiter, S., and Schmidhuber, J. 1997. Long short-term memory. *Neural computation* 9(8):1735–1780.

Hochreiter, S. 1998. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6(02):107–116.

Jaderberg, M.; Mnih, V.; Czarnecki, W. M.; Schaul, T.; Leibo, J. Z.; Silver, D.; and Kavukcuoglu, K. 2016. Reinforcement learning with unsupervised auxiliary tasks. *arXiv preprint arXiv:1611.05397*.

Johnson, M.; Hofmann, K.; Hutton, T.; and Bignell, D. 2016. The malmo platform for artificial intelligence experimentation. In *International joint conference on artificial intelligence (IJCAI)*.

Kaelbling, L. P.; Littman, M. L.; and Moore, A. W. 1996. Reinforcement learning: A survey. *Journal of artificial intelligence research* 4:237–285.

Karakovskiy, S., and Togelius, J. 2012. The mario ai benchmark and competitions. *IEEE Transactions on Computational Intelligence and AI in Games* 4(1):55–67.

Karpov, I. V.; Schrum, J.; and Miikkulainen, R. 2013. Believable bot navigation via playback of human traces. In *Believable Bots*. Springer. 151–170.

Kempka, M.; Wydmuch, M.; Runc, G.; Toczek, J.; and Jaśkowski, W. 2016. ViZDoom: A Doom-based AI Research Platform for Visual Reinforcement Learning. *IEEE Conference on Computational Intelligence in Games (CIG)*.

Konda, V. R., and Tsitsiklis, J. N. 1999. Actor-Critic Algorithms. In *Neural Information Processing Systems (NIPS)*, volume 13, 1008–1014.

Lample, G., and Chaplot, D. S. 2016. Playing FPS Games with Deep Reinforcement Learning. *arXiv preprint arXiv:1609.05521*.

Mäkilä, P. M. 1990. Approximation of stable systems by Laguerre filters. *Automatica* 26(2):333–345.

McCoy, J., and Mateas, M. 2008. An Integrated Agent for Playing Real-Time Strategy Games. In *Association for the Advancement of Artificial Intelligence (AAAI)*, volume 8, 1313–1318.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540):529–533.

Mnih, V.; Badia, A. P.; Mirza, M.; Graves, A.; Lillicrap, T. P.; Harley, T.; Silver, D.; and Kavukcuoglu, K. 2016. Asynchronous Methods for Deep Reinforcement Learning. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, 1928–1937.

Niklasson, L.; Engstrom, H.; and Johansson, U. 2001. An adaptive rock, scissors and paper player based on a tapped delay neural network. *Application and Development of Computer Games Conference in 21st Century (ADCOG21)* 130–136.

Parker, M., and Bryant, B. D. 2012. Neurovisual control in the Quake II environment. *IEEE Transactions on Computational Intelligence and AI in Games* 4(1):44–54.

Perez, D.; Samothrakis, S.; Togelius, J.; Schaul, T.; Lucas, S.; Couëtoux, A.; Lee, J.; Lim, C.-U.; and Thompson, T. 2015. The 2014 general video game playing competition.

Perez-Liebana, D.; Samothrakis, S.; Togelius, J.; Lucas, S. M.; and Schaul, T. 2016. General Video Game AI: Competition, Challenges and Opportunities. In *Thirtieth Association for the Advancement of Artificial Intelligence (AAAI) Conference on Artificial Intelligence*.

Polceanu, M. 2013. Mirrorbot: Using human-inspired mirroring behavior to pass a turing test. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, 1–8. IEEE.

Schaeffer, J.; Burch, N.; Björnsson, Y.; Kishimoto, A.; Müller, M.; Lake, R.; Lu, P.; and Sutphen, S. 2007. Checkers is solved. *Science* 317(5844):1518–1522.

Schaul, T.; Quan, J.; Antonoglou, I.; and Silver, D. 2016. Prioritized experience replay. *International Conference on Learning Representations*.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529(7587):484–489.

Sutskever, I.; Vinyals, O.; and Le, Q. V. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, 3104–3112.

Tesauro, G. 1995. Temporal difference learning and TD-Gammon. *Communications of the ACM* 38(3):58–68.

Van Hoorn, N.; Togelius, J.; and Schmidhuber, J. 2009. Hierarchical controller learning in a first-person shooter. In *2009 IEEE Symposium on Computational Intelligence and Games*, 294–301. IEEE.

Watkins, C. J., and Dayan, P. 1992. Q-learning. *Machine learning* 8(3-4):279–292.

Williams, R. J., and Peng, J. 1991. Function optimization using connectionist reinforcement learning algorithms. *Connection Science* 3(3):241–268.

Wu, Y., and Tian, Y. 2016. Training Agent for First-Person Shoother Game with Actor-Critic Curriculum Learning. https://openreview.net/pdf?id=Hk3mPK5gg. Accessed: 2016-12-04.

Yannakakis, G. N., and Togelius, J. 2015. A panorama of artificial and computational intelligence in games. *IEEE Transactions on Computational Intelligence and AI in Games* 7(4):317–335.