Killer AI: Reinforcement Learning for DOOM Scenarios

Ganesh S, Jang W, Noah G, Yayan Z

gse@umich.edu, jangwu@umich.edu, noahgale@umih.edu, yayanzh@umich.edu

I. Introduction

Machine learning and AI research has long focused on solving games within isolated scenarios. Since researchers at the company DeepMind had posted their famous paper 2013 “Playing Atari with Deep Reinforcement learning” [Mnih et. al 2013], this research has shifted towards solving video games as well, with API’s for specific platforms and games developed, including Google’s Universe project. There are also several worldwide competitions for AI bots in video games, one of which is the ViZDoom AI competition, held in 2016 and 2017. While most competitions take place in 2D environments fully observable to the agent, The VizDOOM API was the first architecture to tackle 3D environments that involved partially observable states [Kempka et al. 2016]. Several universities and major Tech companies entered the VizDOOM competition in 2017, with the CMU team getting 2nd place using the DRQN algorithm we later attempt to replicate, while the Facebook team won with an A3C model.(Asynchronous Advantage Actor Critic Model)

Figure 1 Screenshot of DOOM games

Our group chose to use variants of Deep Q Learning methods in python and Tensorflow to optimally play the 1993 game DOOM, one of the world’s first ‘First-Person Shooter’ (FPS) games, on the VizDOOM API created by Kempa et. al (2016). Because DOOM is 3D, it is more complicated than the 2D Atari games trained by DeepMind. The advantage of training on DOOM is that it contains partially observable states in real time, but is computationally cheap to run due to its age. The main motivation for our project is that even though video game AI has progressed in the 25 years since DOOM’s release, game bots still must cheat against humans by accessing internal game data. So our research is focused on making an AI play at human-or-better levels in DOOM scenarios using only the visual pixel input from the screen. This has real-world robotics applications, due to the number of 3-Dimensional problems in that field.

II. Related Work

Though computers have used machine learning to compete and win against humans as early as DeepBlue’s famous defeat of Gary Kasparov, the use of Machine Learning to accomplish tasks and compete within video game environments is a relatively new and expanding field. With the increase in computing abilities and cloud computing, the OpenAI organization released the Universe environment in 2016 to allow anyone to train AI on a multitude of games from Mario to Grand Theft Auto V [OpenAI 2016]. In 2016, Activision Blizzard announced it would freely release its API for their game StarCraft II in collaboration with DeepMind [Blizzard DeepMind SC2 API 2017].

DeepMind, an AI research company later acquired by Google, released a paper in 2013 showing that a single Reinforcement Learning algorithm could perform at superhuman levels for a large variety of classic video games [Mnih et al. 2013].

Deep Q Networks have been shown to be efficient in playing Atari 2600 games [Mnih et. al 2013]. To further improve upon the basic DQN results, Hausenecket and Stone used LSTM’s combined with Deep networks to learn Q functions and play Atari games, referred to as DRQN’s [Hausknecht and Stone 2015].

In 2016, Double-DQN was introduced to the Atari AI problem by van Hesselt et. al. to correct for observed overestimations in DQN algorithms. Their implementation of Double-DQN derived from double tabular Q learning was tested successfully against 49 games with some results exceeding previous records [van Hesselt et. al. 2016].

In 2016, Wang et al. introduced Dueling-DQN algorithms and compared results over 57 Atari games, finding that the algorithm evaluated policies better than DQN’s while there were similar actions available [Wang et al. 2016].

In 2016 researchers from Carnegie Mellon documented results of running Deep Q-Network and DRQN Networks on a 3D game, [Lampel and Chaplot 2016] arguing that the original DOOM was a good way to train AI for potentially useful problems in 3D world environments based only on visual input. Other methods for solving VizDoom challenges and Deathmatch scenarios were implemented by teams from FaceBook [Wu, Y. and Tian, Y. 2017], IntelAct [Dosovitskiy, A. and Koltun, V. 2017], etc. The IntelAct team used high-dimensional sensory streams and Reinforcement Learning algorithms similar to Monte Carlo methods which learns to act based on raw sensory input from a complex three-dimensional environment. The presented formulation enables learning without a fixed goal at training time, and pursuing dynamically changing goals at test time. They won the Full Deathmatch track of the Visual Doom AI Competition. The Facebook team deployed A3C with CNN to train an agent from recent four raw frames and game variables, to predict next action and value function, following the curriculum learning [Bengio et al., 2009] approach of starting with simple tasks and gradually transition to harder ones. It is nontrivial to apply A3C to such 3D games directly, partly due to sparse and long term reward. They won the champion of Track1 (known map,deathmatch) in ViZDoom AI Competition.

We base our research off of replicating and expanding upon the results of Lampel and Chaplot's paper while applying methods of Double and Dueling DQNs from Wang et. al. and van Hesselt et. al. on the DOOM architecture.

III. METHODOLOGY

A. Deep Q-Networks (DQN)

A deep Q-network (DQN) is a multi-layered neural network that for a given state s of the environment, it outputs a vector of action values $Q_\theta(s, a)$, where θ are the parameters of the network, and decides an action a_t according to a policy π, and observes a reward r_t. For an n-dimensional state space and an action space containing m actions, the neural network is a function from $\mathbb{R}^n$ to $\mathbb{R}^m$. The goal of the agent is to find a policy that maximizes the expected sum of discounted rewards R_t.

$$R_t = \sum_{i=t}^{T} \gamma^{t'-t} r_t$$

where T is the termination time, and $\gamma \epsilon (0,1)$ is a discount factor that determines the importance of future rewards.The Q function of a given policy π is defined as the expected return from executing an action a in a state s:

$$Q^\pi(s.a) = E[R_t | s_t = s, a_t = a]$$

Since it's hard to use a function to approximate the action-value function Q, DQN uses a neural network to estimate the Q-function of the current policy which is close to the optimal Q-function Q^*.

$$Q^*(s,a) = max\, E[R_t | s_t = s, a_t = a] = \max Q^*(s', a')$$

Thus the training process of DQN is trying to find the optimal θ such that $Q(s,a) \sim Q * (s,a)$.

$$Q^*(s,a) = E[r + \gamma max Q^*(s', a') | s, a]$$

According to Bellman equation, the loss function is

$$L_t(\theta_t) = E_{s,a,r,s'}[(y_t - Q_{\theta_t}(s,a))^2]$$

where t is the current time step, and $y_t = r + \gamma max Q(s', a')$.

$$\nabla_{\theta_t} L_t(\theta_t) = E_{s,a.r.s'}[(y_t - Q_\theta(s,a)) \nabla_{\theta_t} Q_{\theta_t}(s,a)]$$

Two important ingredients of the DQN algorithm (shown in Algorithm 1) as proposed by Mnih et al. (2013) are the use of a target network, and the use of experience replay. The target network, with parameters $\bar{\theta}$, is the same as the online network except that its parameters are copied every τ steps from the online network, so that then $\overline{\theta_t} = \theta_t$, and kept fixed on all other steps.

Algorithm 1 Deep Q-learning with Experience Replay

```
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode $= 1, M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
    end for
end for
```

B. Double – DQN

The DQN is known for overestimation due to erroneously giving some Q values high rewards initially, and then choosing future maximized outcomes from that baseline. The Double-DQN algorithm [van Hesselt et. al. 2016] instead takes the max over Q-values when computing the Target-Q value for the training step. Double-DQN uses a primary network to choose an action, and a separate target network to generate the target Q-value for that action. By decoupling the action choice from the target Q-value generation, it's able to substantially reduce the overestimation, and train faster and more reliably than DQN.

$$Y_t^{DoubleDQN} = R_{t+1} + \gamma Q(S_{t+1}, argmax Q(S_{t+1}, a; \theta_t), \theta_t^-)$$

C. Dueling _ DQN

As shown above, the Q function $Q(s,a)$ indicates how good it is to take a certain action given a certain state in time. This action-given state can be decomposed into two more fundamental notions of value:

$$Q(s,a) = V(s) + A(a).$$

The first is the value function $V(s)$, which says simple how good it is to be in any given state. The second is the advantage function $A(a)$, which tells how much better taking a certain action would be compared to the others. We can achieve more robust estimates of state value by decoupling it from the necessity of being attached to specific actions.

As described in paper [Wang et al. 2016], the Q calculated from the above method is unidentifiable. In the sense, given $Q(s,a)$, we cannot get $V(s)$ and $A(a)$ uniquely. In order to tackle this problem, the authors of Dueling Q-Networks have proposed the following method to calculate $Q(s,a)$.

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + (A(s,a;\theta,a) \\ - maxA(s,a';\theta,\alpha))$$

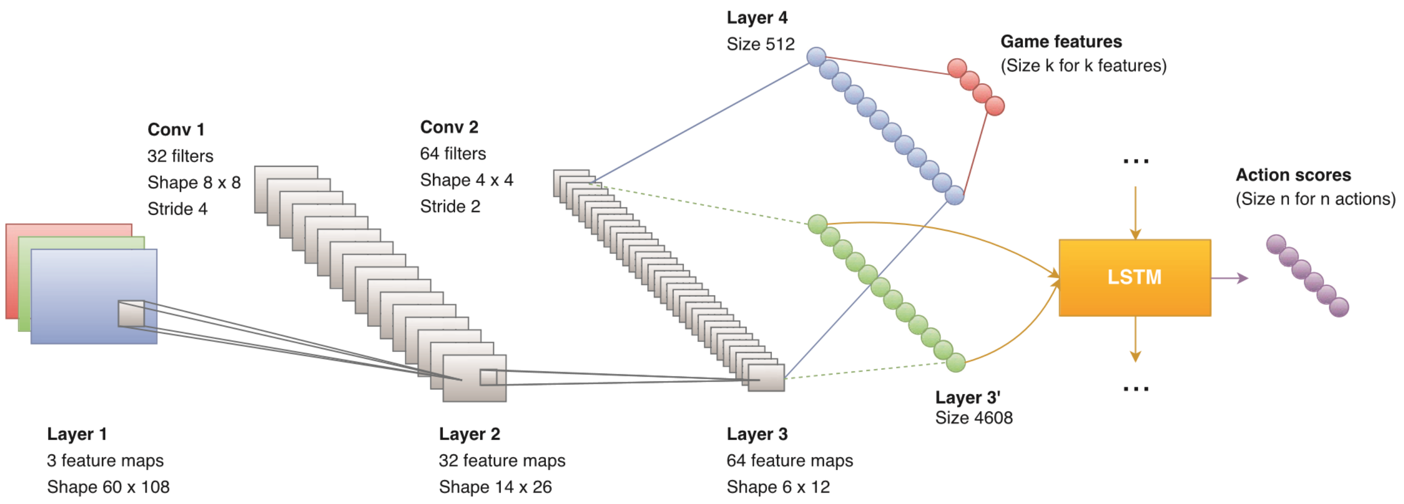

Figure 2 Illustration of architecture (excluding game features). Image Source [Lampel and Champlot, 2016].

D. *Deep Recurrent Q-Network (DRQN)*

For us humans, having access to a limited and changing world is a universal aspect of our shared experience. Despite our partial access to the world, we are able to solve all sorts of challenging problems in the course of going about our daily lives. However, for AI agents in video games like Atari, Deep Q-networks are limited in that they learn a mapping from a single previous state which consist of a small number of game screens. In practice, the DQN is trained using an input consisting of the last four game screens. Thus, DQN algorithms performs poorly at games that require the agent to remember information more than four screens ago.

DQRN is a deep recurrent neural network which is a combination of a recurrent neural network (Long-Short Term Memory) and deep Q network (DQN).

An LSTM is a simple recurrent neural network which can be used as a building component for a recurrent neural network (RNN). An LSTM is composed of 4 parts: a cell, an input gate, an output gate, and a forget gate. The cell is responsible for "remembering" values over arbitrary time intervals; hence the word "memory" in LSTM. Each of the three *gates* can be thought of as a "conventional" artificial neuron, as in a multi-layer (or feedforward) neural network: that is, they compute an activation of a weighted sum. Intuitively, they can be thought as *regulators* of the flow of values that goes through the connections of the LSTM [Long-short term memory, 2017].

The hidden state LSTM carries conveys information about patterns that evolve along time. In our case, it is expected to carry information on portions of the game that have been observed but which are no more visible.

It's able to retain information from states further back in time and incorporate them into predicting better Q values, the algorithm performs better on games requiring long time planning.

E. *Model*

Our neural network, as presented in Figure 2 is similar to one presented in [Lample and Chaplot, 2016] excluding the game features. The first two layers of the DQN and DRQN networks are the same. The first layer is a convolutional layer with 32 filters having filter size 8x8 and stride 4. The second layer is a convolutional layer with 64 filters having filter size 4x4 and stride 2.

In the DQN network, the final convolutional layer output is flattened and sent through a fully connected layer with output size 512. The output of the fully connected layer is sent through another fully connected layer with output size equal to the number of available actions and this results in the Q values.

In the Dueling DQN network, the final convolutional layer output is split into two parts and sent through separate fully connected layers with output size 1 for the value and output size equal to the number of available actions for the advantage. The advantage and value are combined to give the Q values.

In the DRQN network, the final convolutional layer output is flattened and sent through a LSTM cells with sequence length eight and hidden layer size 300. The output from the LSTM is sent through a fully connected layer with output size equal to the number of available actions and this results in the Q values.

The Q function to be minimized depends on the algorithm used, namely DQN, Double-DQN or Dueling-DQN.

IV. Experiment and Results

A. *Test Environment*

We ran our implemented code on a Google cloud compute instance with 2 vCPU with standard memory and 1 NVIDIA Tesla K80 GPU. The framework used to build and train the network is Tensorflow GPU version running on Python 3.5.

B. *Training*

All networks were trained using the RMSProp algorithm and mini batches of size 32 and learning rate is set to 0.00025. The replay memory contained the 10000 most recent frames. The discount factor was set to $\gamma = 0.99$. We used an e-greedy policy during the training, where ϵ was linearly decreased from 1 to 0.1 over the 10000, and then fixed to 0.1. The screen resolution of images used were 60x108 and Grayscale image is used.

Our models are trained and tested in various scenarios. In the following subsections we have given the details and results of one scenario each from a Fully observable environment and a Partially observable environment. For the fully observable environment, we run train and test the code on the DQN, Dueling-DQN and DQRN network. For the partially observable scenario, we train and test our code on Dueling-DQN and DQRN. We chose to leave out the Double-DQN due to large running time of our implementation. All the networks were trained for 20 epochs for the fully observable scenario and 40 epochs for the partially observable scenario. Each epoch consisted of 2000 learning steps.

C. Fully Observable Scenario

The fully observable scenario we use is the *Basic* game mode in the ViZDoom environment. In this scenario the player is spawned against the longer wall, in the center and monster is spawned randomly somewhere along the opposite wall. The available actions for the player are as follows, {MOVE_LEFT, MOVE RIGHT, SHOOT}. The objective of the game is to kill the monster in the shortest amount of time. So the rewards are set accordingly, +101 for killing the monster and -1 for every living time step. This encourages our agent to kill the monster in the shortest possible time. The episode ends when the monster is killed or on timeout and the timeout here is set to 300 tics.

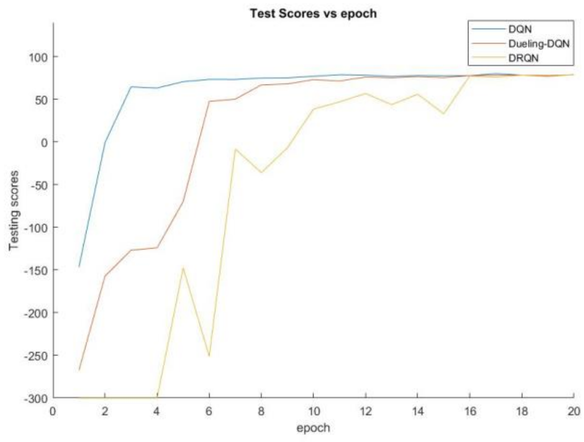

Figure 3 Test scores vs epoch for DQN, Dueling-DQN and DRQN.

Figure 3 shows the results for the test scores vs number of Epochs run. It can be seen that in the fully observable scenario, the three networks converge to the around the same test score. Table 1 gives more details about the average running time per epoch including testing, the standard deviations, the minimum and maximum for the test scores. The minimum score suggests that the agent always finishes the episode by killing the monster. This suggests that the Q function learned by the agent is excellent.

One important thing to note here is that the max possible score achievable is 95 since it takes one tic for making the action shoot and hence -1 and +101 upon killing the monster and -5 for episode end and this occurs when the monster spawns right in the center in front of the player. The testing scores are the average scores over 100 episodes. And since the spawning location of the monsters are random, our average test scores are higher when the average spawning of the monster is closer to the center than near the walls and hence the test scores gives a good idea about convergence and a general idea about the performance but doesn't depict the performance entirely.

Table 1: Mean, standard deviation, minimum and maximum of test scores and time after 20 epochs.

Method	mean	Standard deviation	min	max	Time for 20 epoch (min)
DQN	78.8	10.09	59	95	15.5
Dueling-DQN	78.62	11.68	52	95	11.16
DRQN	78.92	11.16	42	95	14.45

The Q loss vs learning step is plotted in figure 4. It is seen that the loss continuously decreases and converges. We hypothesize that the slowness for the convergence of the DRQN can be explained by the break in the convergence property of DQN due to the recurrent part of the network.

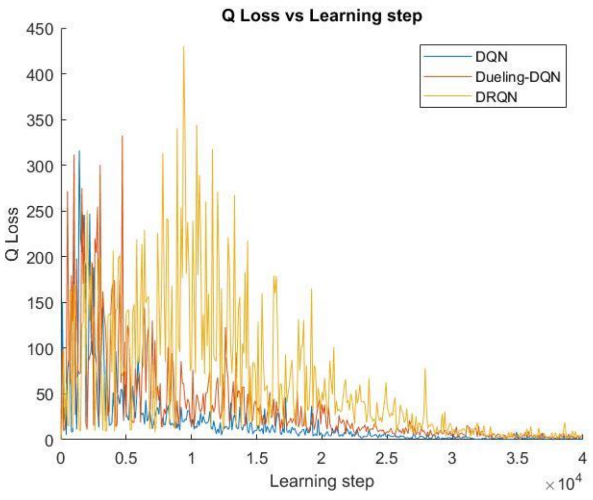

Figure 4 Q-loss vs epoch for DQN, Dueling-DQN and DRQN

It is interesting to note that the running time for the Dueling-DQN is faster than the DQN. This could be attributed to lesser computation after the final convolution layers in the Dueling DQN network.

D. Partially Observable Scenario

The Partially observable scenario we use is the *Defend the center* scenario in the ViZDoom environment. In this scenario the player is spawned in the exact center with limited ammunition. Five melee-only monsters are spawned along the wall at a time and respawn after dying. The available actions for the agent are as follows, {TURN_LEFT, TURN_RIGHT, SHOOT}. The objective of this scenario is to teach the agent that it is good to kill the monster and it is bad to die. It is also important that the agent learns that wasting ammunition is not good either. In this scenario, the agent is rewarded with +1 reward when it kills a monster. So the agent has to figure the rest on its own. The episode ends when the player dies.

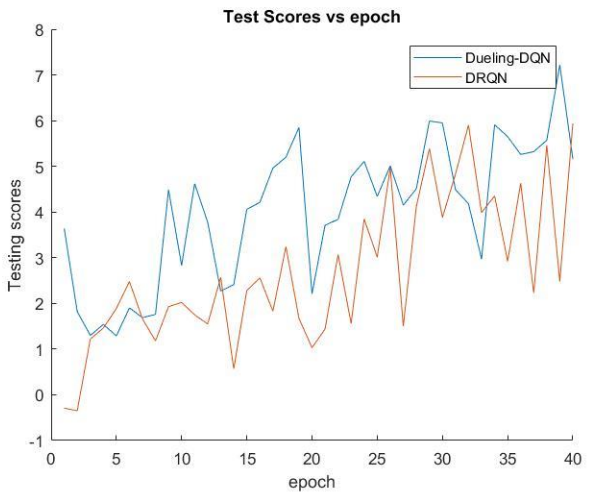

Figure 5 Test scores vs epoch for Dueling-DQN and DRQN in partially observable scenario.

Figure 5 shows the results for the test scores vs number epochs run. It can be seen that the core continuously increases but hasn't converged yet. It is also noisier than the training scores as shown in Figure 6. We hypothesize from the two figures that the network needs more examples or training steps before it can converge. Due to a lack of time, we have limited the training time to 40 epochs.

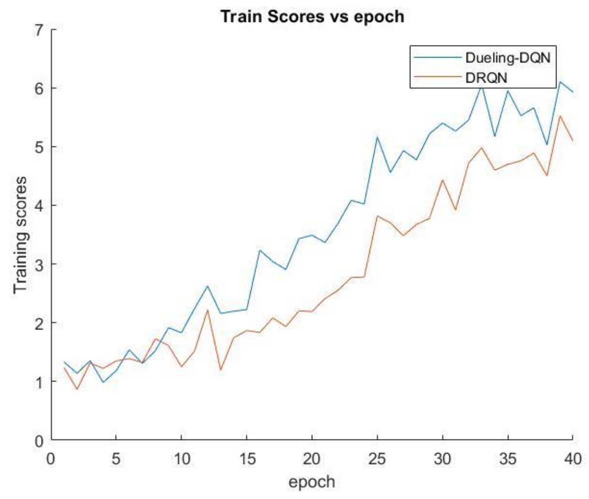

Figure 6 Training scores vs epoch for Dueling-DQN and DQRN in partially observable scenario.

The Q loss vs learning step is plotted in Figure 7. It is interesting to note that the loss for DRQN in the partially observable scenario is very similar to the fully observable scenario, it increases absurdly initially. We again think that the adding the recurrent part breaks the convergence property. This explains the lower training scores and test scores vs epoch for DRQN as compared to Dueling-DQN as seen in Figure 5 and Figure 6. We believe that running them through more epochs will increase the test scores of both the Dueling-DQN and DQRN.

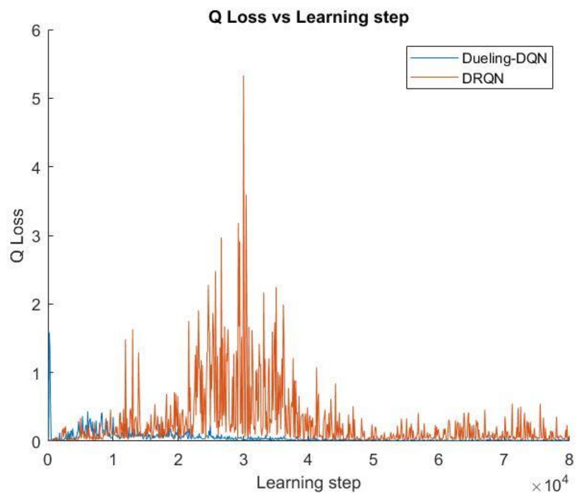

Figure 7 Q-Loss vs learning steps for Dueling-DQN and DRQN in partially observable scenario.

Table 2 gives more details about the average running time per epoch including training and testing, the standard deviations, the minimum and maximum for the test scores. It is again seen that the Dueling DQN is much faster than DRQN.

Table 2: Mean, standard deviation, minimum and maximum test scores and time after 40 epochs.

Method	mean	Standard deviation	min	max	Time for 40 epoch (min)
Dueling-DQN	5.16	2.50	2	15	65.19
DRQN	5.94	1.65	3	10	100.22

V. Conclusion

In this project, we have implemented a stable DQN, Dueling-DQN and DQRN capable of playing the DOOM game. We have trained and tested our networks in both fully observable scenarios and partially observable scenarios and were able to achieve positive results. We think that for the partially observable scenario, running the algorithm for more time would lead to much better results. We tried to run the deathmatch scenario but were unfortunately limited by GPU memory for running it. Further we have used Grayscale images at low resolution for our network. Increasing the resolution and using RGB images would further increase the performance of the implemented networks at the cost of computation time.

The game play of our agent after the training can be found in link https://www.youtube.com/watch?v=h5pZeMhMBuI&list=PLyZbqMF4T4x3ahJrtmEPa_bNFbuzqNK4w.

A link to our gitlab repository is https://gitlab.eecs.umich.edu/gse/EECS545_doom.

ACKNOWLEDGMENT

We would like to thank Prof. Mert Pilanci, Prof. Salimeh Yasaei Sekeh, Haozhu Wang and Ankit Bansal for their valuable help and feedback during the course of our project. We would also like to thank the ZDoom community for their help in utilizing the Doom game engine.

TEAM ROLES

The individual roles of each of our team members are given below. We read and studied through literature and concepts individually but final implementation, data analysis for result and report were done together as a team.

Ganesh: Setting up environment to extract and update actions on VizDOOM. DQN and DRQN algorithm.

Yayan: Double-DQN and Dueling-DQN algorithm.

Noah: Double-DQN and Dueling-DQN algorithm.

Jang: DQN, Dueling-DQN and DRQN algorithm.

REFERENCES

[Mnih et al. 2013] 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.

[Lampel and Champlot, 2016] Lampel G.; Chaplot D. 2016. Playing FPS games with deep reinforcement learning. arXiv print arXiv: 1609.05521

[Hausknecht and Stone 2015] Hausknecht, M., and Stone, P. 2015. Deep recurrent q-learning for partially observable mdps. arXiv print arXiv:1507.06527.

[Kempka et al. 2016] Kempka, M.; Wydmuch, M.; Runc, G.; Toczek, J.; and Jaskowski, W. 2016. Vizdoom: A doom-based ai research platform for visual reinforcement learning. arXiv print arXiv:1605.02097.

[Wang et al. 2016] Wang, Z.; Schaul, T.; Hessel M.; van Hasselt, H.; Lanctot, M.; de Freitas N.; Dueling network architectures for deep reinforcement learning. arXiv print arXiv: 1511.06581.

[Blizzard DeepMind SC2 API, 2017] DeepMind and Blizzard Open SC2 Research Environment. Aug 9, 2017. url: https://deepmind.com/blog/deepmind-and-blizzard-open-starcraft-ii-ai-research-environment/

[OpenAI, 2016] OpenAI Universe blog introduction page. Dec 6, 2016. url: https://blog.openai.com/universe/

[Wu, Y. and Tian, Y. 2017] Wu, Y. and Tian, Y.Training agent for first-person shooter game with actor-critic curriculum learning. In the International Conference on Learning Representations (ICLR).

[Dosovitskiy, A. and Koltun, V. 2017] Dosovitskiy, A. and Koltun, V.Learning to act by predicting the future. In the International Conference on Learning Representations (ICLR).

[Van Hasselt et al. 2015] Van Hasselt, Hado, Guez, Arthur, and Silver, David. Deep reinforcement learning with double q-learning. arXiv preprint arXiv:1509.06461, 2015.

[Long-short term memory, 2017] Long-short term memory (n.d.). In *Wikipedia*. Retrieved December 17, 2017, from https://www.wikiwand.com/en/Long_short-term_memory