Microsoft Innovation Lab
Summer Internship 2018

(4th June - 27th July)

Reinforcement Learning for Games

Niharika, Ishaan and Saksham
Mentor: Sree Hari

Contents

1 Introduction

Reinforcement learning is a method used to train an artificially intelligent model by a simple reward-punishment system where an agent in an environment tries to maximize its positive rewards and minimize it's negative rewards. Reinforcement learning[1] is a subtopic of machine learning, inspired from behaviorist psychology. The base of any reinforcement learning algorithm involves Markov's decision process(MDP), which provides a mathematical framework where outcomes are partly random and partly under the control of the agent's previous actions. In simple words, 'The future is independent of the past given the present'.

2 Concepts of Reinforcement Learning

2.1 Markov's Decision Process

The Markov's decision process is a 5-tuple problem of

$$(S, A, P_a, R_a, \gamma)$$

1. S is the number of finite states
2. A is a finite set of actions
3. P_a(s,s') = Pr(s_t+$_1$ = s'|s_t=s,a_t) is the probability that an action a in a state s at time t will lead to state s' at time t+1
4. R_a(s,s') is the immediate reward received from transitioning from state s to s'
5. γ is the discount factor varying from 0 to 1, differentiating the importance of the future rewards with the present reward.

The core problem of the MDP is to find the most optimal policy π that specifies the action π(s) that the decision maker will choose in the state s. Or, to minimize the expression,

$$\sum_{t=0}^{\infty} \gamma^t \; R_a[t](s_t, s_t{+}_1)$$

where the symbols mean the same as explained above.
The return Gt, which is the total discounted rewards from time step t, needs to be maximized in this scenario where,

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

where R_t denotes the reward at time t.

And this is the key aim of reinforcement learning, to train an agent to maximize the profit returns on every action that the agent in an environment makes randomly.

In reinforcement learning we use the MDP to find the optimum policy to minimize punishments and maximize rewards. But in reinforcement learning, this kind of random exploration is not clever enough to instantiate every kind of possible outcome. So various algorithms are implemented.

2.2 Different algorithms in reinforcement learning

- Criterion of optimality
  The agent must find a policy with the maximum expected return. But we can narrow the search for this policy by elimination of possibilities. For example, we can only consider the stationary states where the states whose outcome is dependent only on the previous action is considered and the other states are discarded. Another optimization that can be done is considering only deterministic stationary policies which selects policies based on a current state specified.

- Brute Force
  This involves only two steps: For each policy, sample returns following it and choose the policy with the highest possible return. But this again takes a massive load of time and the feasibility also subsequently decreases.

- Direct Policy Search
  This is an alternative method to search for actions in the immediate policy space. They can be gradient based methods or gradient free methods. In the gradient free model, the gradients of each policy are mapped and by gradient ascent, the optimum policy is found from the

feasible policy space. But this gives only a noisy estimate and is not very reliable. Gradient free methods use more complex algorithms such as simulated annealing, cross entropy searches and may achieve a global optimum in some cases. But these converge slowly, often to give noisy data, making Value functions a better approach.

- Value Function
  Value function approaches attempt to find a policy that maximizes the return by maintaining a set of estimates of expected returns for some policy. These rely on the theory of MDPs. We define the value policy by V where,
  V^{π}(s) = E[R|s,π]
  Expanding,
  $$V^{\pi}(s) = E_{\pi}[r_t + \gamma V^{\pi}(s_t + 1)|s_t = s]$$
  This is called the Bellman equation, where V is the value of the policy π and R stands for the random return associated with the following π from the initial state s. With this we can say that,
  V^{*} = max(π)V^{π}(s).
  Where the policy π is allowed to change and V^{*} denotes the maximum value of the value function V over any policy. A policy that achieves these optimal values is called the optimal policy.
  Although the state-value pairs are enough to define optimality, it is useful to define action-value pairs Q. Given a state s, an action a, and a policy π, action-value pairs of state action pair (s,a) under a policy π are given by
  Q^{π}(s,a) = E[R|s,a,π].
  where R now stands for the return associated with first taking the action a in state s and following π, thereafter.
  An important thing to note is that the knowledge of the optimal action-value function is sufficient to define optimality.// There are two methods to implement value functions: Monte Carlo methods and Temporal Difference Learning.
  Monte Carlo methods use random sampling to compute numerical results. A Monte Carlo simulation performs risk analysis by substituting a range of values-a probability distribution-for any factor that has inherent uncertainty. Depending upon the number of uncertainties and the ranges specified for them, a Monte Carlo simulation could involve thousands or tens of thousands of recalculations before it is complete.

Monte Carlo simulation produces distributions of possible outcome values.
But this method has a lot of drawbacks.

- It uses samples inefficiently because evaluation of a long trajectory takes into account only a single state action pair.
- The procedure takes too much time finding the suboptimal policy.
- It works in episodic problems only.
- The variance is indirectly proportional to convergence.
- It works in finite, small MDPs only.

This is when we introduce temporal difference Learning, and this will be the base used throughout the project.
Temporal difference learning or TD Learning is an approach that depends purely upon the future actions of an agent in the environment.The prediction of each time step's outcome is updated to bring it closer to the outcome of the consequent time step. These are usually used in reinforcement learning to predict a measure of the total amount of reward expected over the future, but they can be used to predict other quantities as well. The agent has no idea what to expect from the unknown environment it's in, so it trains to finds patterns and learns from its raw experience in the environment.
The TD method aims to achieve an approximation V_θ^π as close to V^π as possible. The error of the approximation can be measured by

$$MSE(\theta) = \frac{1}{n}\sum_{i=1}^{\infty}(V_\theta^\pi(s_i) - V^\pi(s_i)^2)$$

By minimizing the MSE, the approximation of the value function can be optimized. As V^π(s) is unknown, it is estimated by applying the Bellman equation on the current approximation of V_θ^π.

$$V^\pi(s_t) \approx [r_t + \gamma(V_\theta^\pi(s_t{+}_1)]$$

The application of the Bellman equation is the foundation of TD Learning. This concept paves the way for one of the most favored algorithms in reinforcement learning, Q-Learning.

2.3 Q-Learning

Q-Learning focuses on training an agent by letting said agent traverse through an unfamiliar environment, where it tries to map what actions give it a reward and what gives it a punishment. It's second agenda becomes to increase the net reward gain and that is the foundation of Q-Learning. The way it remembers what actions give it a positive or negative outcome is described by a Q-table.
Let's take a simple example.

H---------P---------------C

This is a simple game where the player P, or our agent has to acquire the cheese denoted by C and avoid the hole H. The rewards work in a fashion say, when it encounters a hole, it gets a -10 punishment but when it encounters cheese, it gets a +10 reward. The agent performs random actions in this environment and gets rewards and/or punishments accordingly. The way the agent gets better at playing in the environment is with the use of a Q-table.
A Q-table is a matrix of states versus actions. This table is initialized with uniform values at the beginning. The agent the performs random actions in the unknown environment, and the rewards or punishments are updated onto the table.The table is updated using the Bellman equation where,

$$Q(s,a) = r + \gamma(max(Q(s',a'))$$

S represents the current state . a represents the action the agent takes from the current state. S' represents the state resulting from the action. r is the reward you get for taking the action and γ is the discount factor. So, the Q value for the state S taking the action a is the sum of the instant reward and the discounted future reward (value of the resulting state). The discount factor γ determines how much importance you want to give to future rewards. In the above example, the agent is constrained in such a way that if it is closer to the cheese, it expects a positive reward. Since the agent knows that every time it gets the cheese, it gets a positive reward, after training it will try to move towards the cheese and away from the hole. The Q table is formed in such a way that the action of going right is rewarded and the action

of going left is penalized. By using this concept, the agent is more constrained to move towards the right rather than the left.
The gist of Q Learning is that we can iterative approximate Q^* using the Bellman equation and the Q learning equation is thus given by,

$$Q_t +_1 (s_t, a_t) = Q_t(s_t, a_t) + \alpha(r_t +_1 + \gamma max Q_t(s_t +_1) - Q_t(s_t, a_t))$$

where α is the learning rate that controls how much the difference between the previous and new Q value is considered.
Q-learning can be inferred to be a very powerful algorithm in light of the above segment, but it's main drawback is it's lack of generality. A simple way to explain this is, that if an agent using a Q-learning algorithm has explored a state before, it has some idea about what it is supposed to do when the state is encountered again, but has no clue as to how to proceed in terms of states which it has never seen before. To overcome this problem, Deep Q Learning was introduced with the concept of no multi-dimensional arrays, but a neural network.

2.4 Deep Q Learning

A Deep Q Learning network or a DQN, utilizes a neural network to predict the output Q value of a state action pair by passing it through different layers of a neural network. For Atari games, the neural network takes a raw image as input, processes it through different layers and gives a Q value as an output. To train the network we use a modified version of the Q learning algorithm.

$$r_j + \gamma \max_{a'} Q\left(\phi_{j+1}, a'; \theta^-\right)$$

The ϕ is equivalent to the state S, while the θ stands for the parameters in the neural network. There are two other essential techniques required in a DQN.

- Experience Replay
  Training samples in an RL model are highly correlated with very less distinguishing factors. To reduce this, the sample transitions

are stored and are selected from a random transition pool to update the model. Essentially, This means instead of running Q-learning on state/action pairs as they occur during simulation or actual experience, the system stores the data discovered for $(state, action, reward, nextstate)$ - typically in a large table. The learning phase is then logically separate from gaining experience, and based on taking random samples from this table. The two processes of acting and learning are recommended to interleave, because improving the policy will lead to different behavior, exploring or that should explore actions more closer to the optimal ones, and learn from them.This is the most efficient use of the previous training episodes and since these do not converge quickly, multiple passes to the same data is beneficial, especially when the variance of the data is low.

– Separate Target Network
  We initialize a new target network, which is very close to the optimal one, and this target network gets updated in every T steps of the iteration, to provide more stabilized results. It's similar to when you show a finished model to your agent and then tell the agent that this is what you need to achieve, through learning, the agent will try to mimic the exact same behavior as that of the behavior you have shown it.

There is also another approach within the DQN framework called the Epsilon greedy policy. This is best described by the multi-armed bandit problem. In this problem, there are 3 slot machines with different probabilities of a reward and different outcomes every single iteration. There are two main goals, one to experiment with a few coins and get the maximum payout and second to get the maximum payout possible. The terms 'explore' and exploit' are used to define that you have to use some coins to find the machine that gives the maximum average payout and the second to use the machine with the maximum payout to get the maximum rewards. So in the epsilon greedy method, each machine's average payout is kept in mind. The machine with the highest average payout is selected, so the probability distribution of obtaining a reward is not too varied. And then the machines are arranged according to the highest average maximum payout. The machine with the maximum payout gets played more and the others less, thus increasing your total

probability to get a reward. In short, the epsilon greedy method chooses the best current action most of the time, but picks a random option with a small epsilon value.
DQN seemed to be a powerful tool as well, but came with it's own drawbacks of overestimation of Q-values. This was overcome by using a Double DQN model.

2.5 Double Deep Q Networks

A major problem with the concept of DQNs is that when a non-optimal action is performed, the agent might still get a high Q-value because the net gain was high. If this occurs, the agent tries to overestimate that any action in the vicinity of the action just performed is going to yield it a high Q value. This may lead to drastically non-optimal results. The solution is given by a double DQN where two networks are used to decouple the action selection from the target Q-value generation. We use the first network to find the Q-value of the action just performed and the target network to check the feasibility of performing the action to get it closer to the optimal state.

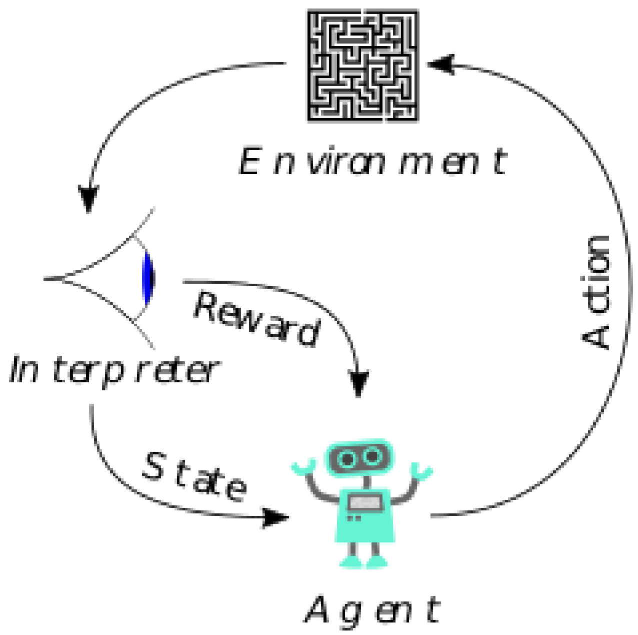
$$Q(s,a) = r(s,a) + \gamma Q(s', argmax_a Q(s',a))$$

We use this concept in the final experiment of training the agent to play a first person shooter game where one network is used for finding the optimal action the agent can do, and the other for how the action fits in the final target network.

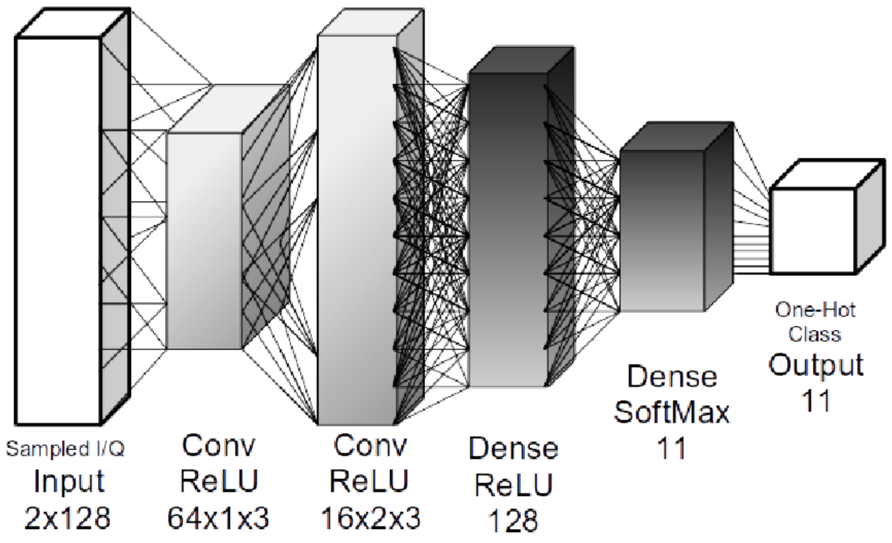
3 Problem Statement

To create an artificially intelligent bot that can play games better than a human, improve continuously over several iterations in a virtual environment and finally integrate it to any general game environment through reinforcement learning, of a particular genre.

4 Block Diagram/ Flow Chart



This is a basic representation of the key algorithm of reinforcement learning. The agent interacts with the environment through a learning platform such as ViZDoom and then analyses the environment and chooses an action to perform. The agent follows a DQN model for training the initial Atari and flashgames and uses Double DQNs to train the final first person shooter. The network architecture in a DQN is designed with convolutional and fully connected layers, where the convolutional layers break down the raw image input and the fully connected layers piece together the different objects in the environment and calculate the Q value of the possible action-state pair.



This is how the convolutional network architecture looks like. The dense and convolutional layers are usually hidden and the ourput is the Q-value of the input action state pair. The ReLu and SoftMax are activation functions that bring the tensors between a 0 and 1 value to provide a good interface for calculations and processing.

5 Progress

5.1 Basic Overview

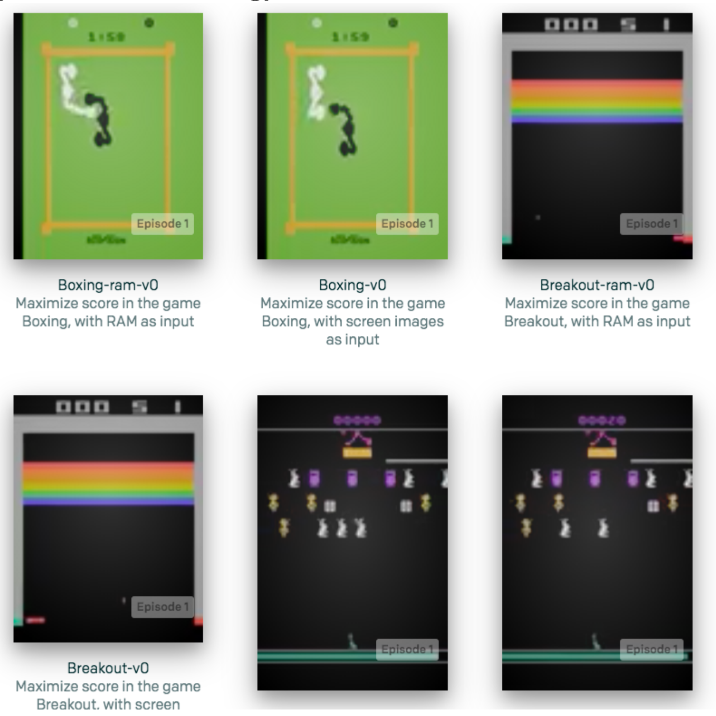
Finding the most optimum policy to use was the first step to begin this project. Our literature survey dissected various machine learning algorithms performed by different sources, with special focus to reinforcement learning. After exploring various environments for reinforcement learning such as Unity's machine learning agents, OpenAI's Gym, Retro learning Environment and Arcade Learning brought to light the ViZDoom environment, for the first person shooter game, Doom 3D.
This environment was used to develop our agent, using keras libraries and reinforcement learning with tensorflow, where our agent was thrown into an environment to kill monsters approaching it.

5.2 Literature Review

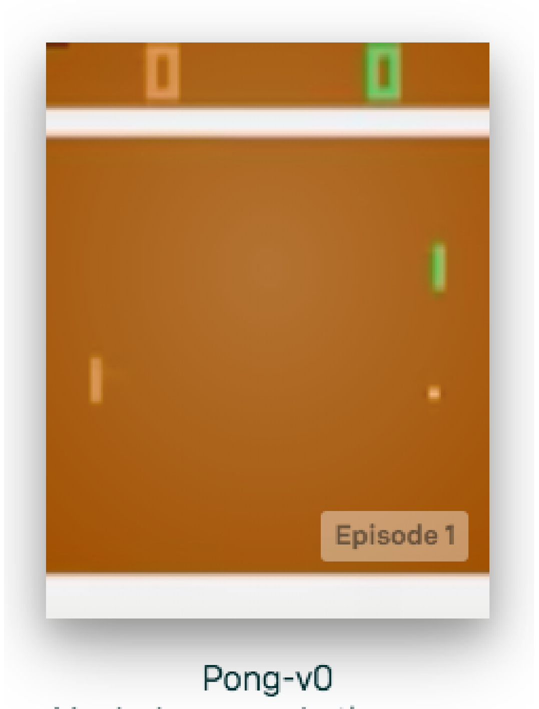
The literature review involved exploration into the various methods of reinforcement learning, supervised and unsupervised learning. First the above theory was studied in detail and papers published on topics related to these and previously done projects were understood as well. Different interfaces and modules were also introduced such as OpenAI Gym, Arcade Learning Environment, Retro Learning Environment, ViZDoom, tensorflow, keras, and many more.

5.3 June 11 to 15

Week two started with further survey into OpenAI's Gym and using this[2] repository to try to understand how machine learning was implemented in this environment. OpenAI is an enterprise created by Elon Musk, with a powerful toolkit to render environments of basic Atari games and Box2D games to enable machine learning architectures onto their base platforms. It was made specifically for reinforcement learning, due to two major problems faced by reinforcement learning, the overall generality and the need for benchmarks for any RL algorithm and the lack of standardisation. Gym fixes both the problems as it provides a acceptable benchmark in terms of it's many applications and environment that an agent can train on any also provides a standard set of functions which are general and standardized for any environment on gym.


Episode 1
Boxing-ram-v0
Maximize score in the game Boxing, with RAM as input
Boxing-v0
Maximize score in the game Boxing, with screen images as input
Breakout-ram-v0
Maximize score in the game Breakout, with RAM as input
Breakout-v0
Maximize score in the game Breakout. with screen

A lot of environments were tested and the first model of Pong-v0 was tested with an agent playing against the CPU. The experiment resulted in the agent winning in almost every iteration, using reinforcement learning to train it in two days. The pong model was trained on the environment provided by gym, where the paddle on the right was the agent and the left paddle was the CPU. The rewards were given based on the fact that every time the left paddle missed the ball, it would get a punishment and every time the it obtained a point it would get a positive reward.

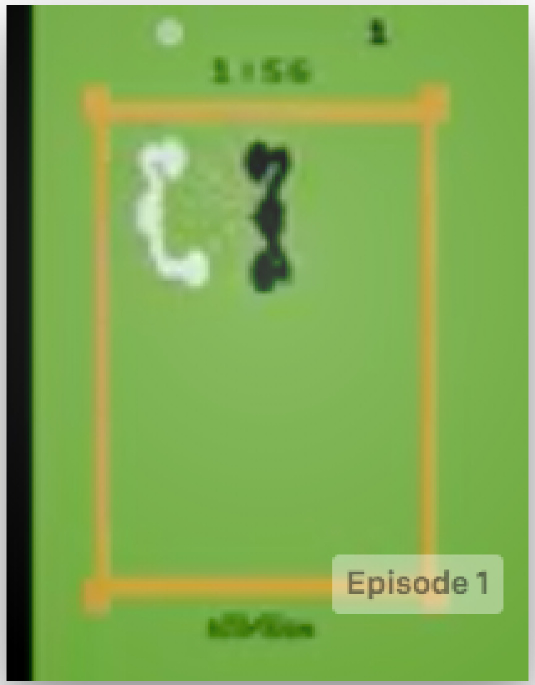


Pong-v0
Maximize score in the game Pong, with screen images as input

This used a simple Q learning algorithm using experience replay, where the agent learns on the basis of it's previous actions. First, the raw input image was down sampled to a 640 x 380 screen resolution and then converted from BGR to GRAY settings to make the training more efficient. The environment renders iteratively, and the agent is subjected to repeated trials for 'n' episodes and with each episode, it updates it's Q-table with the actions performed. The final model is then able to take decisions and win effectively with a small margin.

5.4 June 18 to 22

Week three involved exploration in depth into Gym and the Arcade Learning environment, where simple Atari games were used to train agents by using reinforcement learning. A model of a boxing game was chosen. The agent was thrown into an environment where it had to land punches on an enemy boxer and each punch would gain him a point.



Boxing-v0
Maximize score in the game Boxing, with screen images as input

The Arcade learning environment is a simple object oriented framework to develop agents for Atari 2600 games. The agent uses a DQN framework to train. First the hyperparameters such as epsilon, gamma, explore count, etc were described. Then a class containing the Brain was initialized, which contained the main neural network and the target neural network. The weights and biases were updated every training episode as the agent trained in the environment. This model made the use of the Epsilon greedy policy of DQN, and the epsilon values decreased rapidly as the training furthered. Upon training for 6 hours, we saw the agent faring well against his opponent, losing to it by only small margins. Upon further training, it was able to defeat it as well.

5.5 June 25 to 29

The fourth week involved a more complex game - Dusk Drive - which is a flash game, with significantly higher graphics and consequently more parameters. The model after training saw the agent able to navigate a track with traffic and was able to learn from when it went off the road or crashed into obstacles.

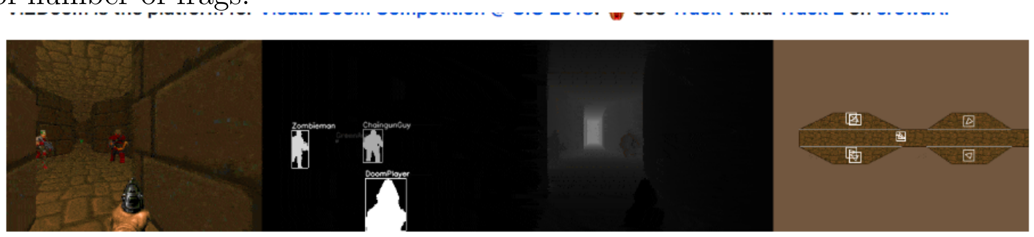
Flash games exist in the gym repository, under a module named Universe. Universe provides an interface to interact with more complex games such as flash games and handles the various parameters that the agent perceives in the environment. The agent is given the set of actions that it can perform and thrown into the environment. The agents performs random actions and gets its rewards accordingly.

5.6 July 2 to 6

The fifth week was dedicated to learning about Unity's ML-Agents repository, a beta model introduced by Unity technologies. Unity's ML agents provided a python interface for users to code the agents onto the Game objects that were made in an environment. It involved making an environment in unity and coding a "Brain" and an "Academy", signifying the actions of the agent and the training library of the agent respectively. The brain could be used to train the agent through various algorithms such as reinforcement learning, imitation learning, neuroevolution and other machine learning techniques. It also provides support to tensorboard and provides functions for proximal policy optimization and strong support to the python interface for learning. Since the model was in a beta stage, it was hard to understand how to inculcate reinforcement learning in this model, so the idea was dropped.

5.7 July 9 to 13

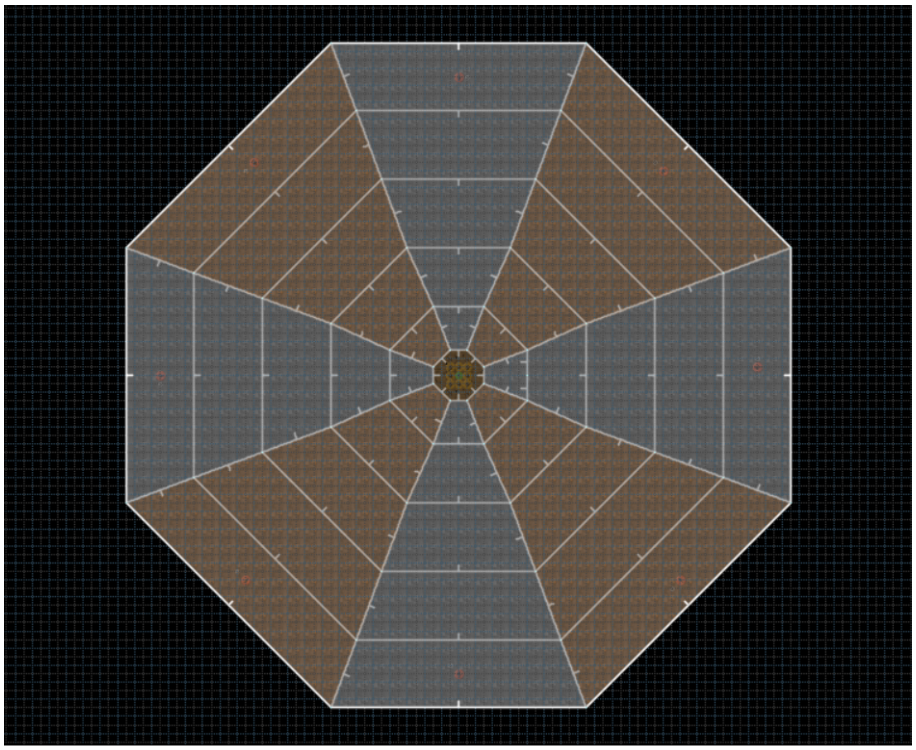
The sixth week involved exploration into the ViZDoom environment, with Arnold, a bot created by Devendra Singh Chaplot and Guillame Lample, which played the ViZDoom deathmatch tournament and came second next to F1, a bot by Intel. It used Deep Q learning to train the bot and was inserted in an environment with other bots, and was rewarded on the basis of number of frags.



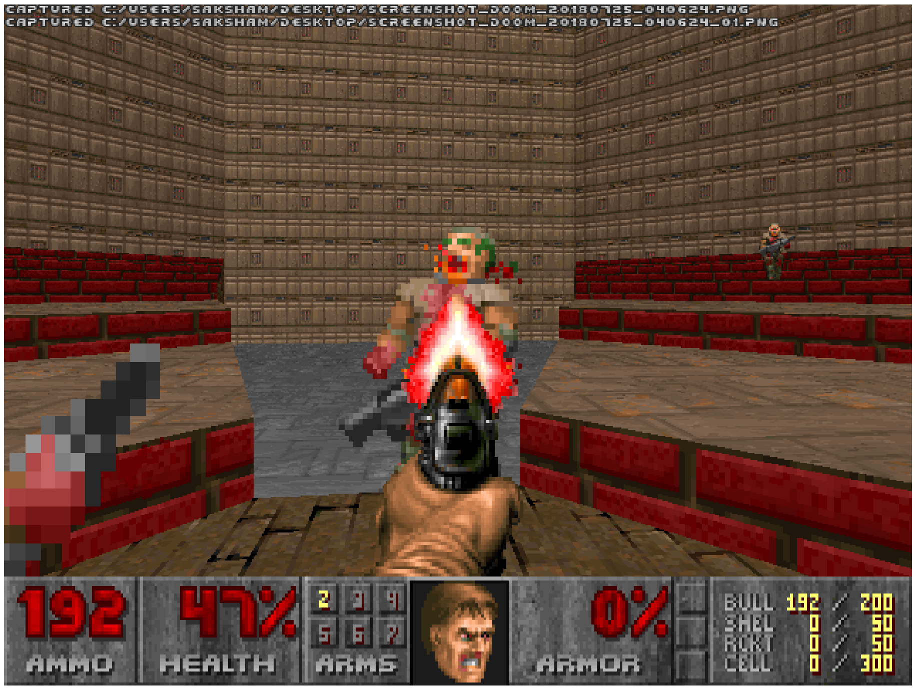
The interface provided by ViZDoom for training the agent use a python API and provided functions to access killcount, ammo, health and other parameters required to describe the environment completely. The Arnold framework was tested on our own system and we were able to recreate the bot in our own systems and were able to run it on a model that used double deep Q networks. One of the networks consisted the actions of the agent in present time by taking the raw image input of the screen and then passing it through filters to decide the Q-value of the action it was going to take, while the second calculates how far away it is from the most optimal target network. The losses were found to be slowly decreasing as the epsilon values started to decrease and hence, making the actions less and less random.

5.8 July 16 to 20

The seventh week focused on creating our own Deep Q AI Bot and making an environment on Doom Builder 2, so as to inject said agent into a self created environment.



The environment created resembled the defend the center map of Doom, where the agent was at the center of the map and monsters approached it from all directions. Using the keras and tensorflow module, the agent was trained to shoot any monster approaching him. The agent was put to training on a GPU and upon training, was observed to survive till his ammo was elapsed, while not missing many of its shots. It's actions were observed to be less and less random and the model seemed to mimic a human playing the game rather than an artificially intelligent bot.


CAPTURED C:/USERS/SAKSHAM/DESKTOP/SCREENSHOT_DOOM_20180725_040624.PNG
CAPTURED C:/USERS/SAKSHAM/DESKTOP/SCREENSHOT_DOOM_20180725_040624_01.PNG
192
AMMO
47%
HEALTH
2 3 4
5 6 7
ARMS
0%
ARMOR
BULL 192 / 200
SHEL 0 / 50
RCKT 0 / 50
CELL 0 / 300

5.9 July 23 to 27

The last week included training the model and further optimising the hyper parameters to make the model perform better in the environment. The model trained on an NVIDIA GTX 1080 for 2 days. This agent was subjected to an unknown environment, so the best framework we could use was the double DQN and the model was able to train perfectly. The double DQN architecture seemed to be the best fit for this project in terms of net loss and training speed, hence it was chosen.

6 References

1. Reinforcement learning
https://medium.freecodecamp.org/an-introduction-to-reinforcement-learning-4339519de419
Model-Free Episodic Control, C. Blundell et al., arXiv, 2016. Safe and Efficient Off-Policy Reinforcement Learning, R. Munos et al., arXiv, 2016.

Deep Successor Reinforcement Learning, T. D. Kulkarni et al., arXiv, 2016.
Unifying Count-Based Exploration and Intrinsic Motivation, M. G. Bellemare et al., arXiv, 2016.
Control of Memory, Active Perception, and Action in Minecraft, J. Oh et al., ICML, 2016.
Dynamic Frame skip Deep Q Network, A. S. Lakshminarayanan et al., IJCAI Deep RL Workshop, 2016.
Hierarchical Reinforcement Learning using Spatio-Temporal Abstractions and Deep Neural Networks, R. Krishnamurthy et al., arXiv, 2016.
Hierarchical Deep Reinforcement Learning: Integrating Temporal Abstraction and Intrinsic Motivation, T. D. Kulkarni et al., arXiv, 2016.
Deep Exploration via Bootstrapped DQN, I. Osband et al., arXiv, 2016.
Learning to Communicate to Solve Riddles with Deep Distributed Recurrent Q-Networks, J. N. Foerster et al., arXiv, 2016.
Asynchronous Methods for Deep Reinforcement Learning, V. Mnih et al., arXiv, 2016.
Mastering the game of Go with deep neural networks and tree search, D. Silver et al., Nature, 2016.
Increasing the Action Gap: New Operators for Reinforcement Learning, M. G. Bellemare et al., AAAI, 2016.
How to Discount Deep Reinforcement Learning: Towards New Dynamic Strategies, V. Franois-Lavet et al., NIPS Workshop, 2015.
Multiagent Cooperation and Competition with Deep Reinforcement Learning, A. Tampuu et al., arXiv, 2015.
MazeBase: A Sandbox for Learning from Games, S. Sukhbaatar et al., arXiv, 2016. Dueling Network Architectures for Deep Reinforcement Learning, Z. Wang et al., arXiv, 2015.
Better Computer Go Player with Neural Network and Long-term Prediction, Y. Tian et al., ICLR, 2016.
Actor-Mimic: Deep Multitask and Transfer Reinforcement Learning, E. Parisotto, et al., ICLR, 2016.
Policy Distillation, A. A. Rusu et at., ICLR, 2016.
Prioritized Experience Replay, T. Schaul et al., ICLR, 2016.
Deep Reinforcement Learning with an Action Space Defined by Natural Language, J. He et al., arXiv, 2015.
Deep Reinforcement Learning in Parameterized Action Space, M. Hausknecht et al., ICLR, 2016.
Variational Information Maximisation for Intrinsically Motivated Reinforce-

ment Learning, S. Mohamed and D. J. Rezende, arXiv, 2015.
Deep Reinforcement Learning with Double Q-learning, H. van Hasselt et al., arXiv, 2015.
Continuous control with deep reinforcement learning, T. P. Lillicrap et al., ICLR, 2016.
Language Understanding for Text-based Games Using Deep Reinforcement Learning, K. Narasimhan et al., EMNLP, 2015.
Giraffe: Using Deep Reinforcement Learning to Play Chess, M. Lai, arXiv, 2015. Action-Conditional Video Prediction using Deep Networks in Atari Games, J. Oh et al., NIPS, 2015.
Deep Recurrent Q-Learning for Partially Observable MDPs, M. Hausknecht and P. Stone, arXiv, 2015.
Incentivizing Exploration In Reinforcement Learning With Deep Predictive Models, B. C. Stadie et al., arXiv, 2015.
Universal Value Function Approximators, T. Schaul et al., ICML, 2015.
Massively Parallel Methods for Deep Reinforcement Learning, A. Nair et al., ICML Workshop, 2015.
Trust Region Policy Optimization, J. Schulman et al., ICML, 2015.
Human-level control through deep reinforcement learning, V. Mnih et al., Nature, 2015.
Deep Learning for Real-Time Atari Game Play Using Offline Monte-Carlo Tree Search Planning, X. Guo et al., NIPS, 2014.
Playing Atari with Deep Reinforcement Learning, V. Mnih et al., NIPS Workshop, 2013.
2. OpenAI Gym
https://github.com/openai/gym/tree/522c2c532293399920743265d9bc761ed18eadb3
3. Arcade learning environment and Retro Learning environment
https://github.com/mgbellemare/Arcade-Learning-Environment
https://github.com/nadavbh12/Retro-Learning-Environment
4. ViZDoom
https://github.com/mwydmuch/ViZDoom
5. Arnold
https://github.com/glample/Arnold
6. Unity ML Agents https://github.com/Unity-Technologies/ml-agents
7. Universe https://github.com/openai/universe