Cardiff University

School of Computer Science and Informatics

CM3203 – Individual Project Final Report

Doom-playing AI via Deep Reinforcement Learning

Supervisor: Dr Frank C Langbein

Author: Hugo Huang

Table of Contents

1 Introduction

First-person shooter (FPS) is a highly-competitive genre of video game, players would control their own characters in first-person perspective and attempt to kill as much players or computer-controlled enemies as they can in a virtual world. Sometimes game companies would use Artificial Intelligence (AI) agents to simulate real players when the amount of active players dropped, or when they consider the average match-making[1] time too high. While there has been no previous study showing issues with these player-mimicking AI, some players have complained online about these bots[2] destroying their gaming experience by having unfair advantage in reflex speed and available in-game information. With enough practice, a human player can learn to perform some frame-perfect moves like unbuffered manual superswim [1,22] and use them in challenge runs[3] or speedruns[4], but any extra strategically-important information the AI players receive would still render the whole match unfair.

It would be desirable to replace these AI players with ones that can only access the information available to human players and perform at a human or near-human level. Human players play games using high-dimensional sensory inputs like vision and training AI agents to play games directly from these inputs was widely considered as one of the greatest challenges of reinforcement learning (RL) [26] until Deep Q-Network (DQN) [26] was proposed. Even with more performant variants of DQN [14], training agents to play FPS games like Doom[5] is still a non-trivial task. In this project, I have attempted to create an RL agent that plays Doom using a DQN and a variant of DQN with recurrency introduced, known as Deep Recurrent Q-Network (DRQN) [14].

Unlike the games traditionally played with DQN such as the Atari 2600 games [14,26,27,39,42], an FPS game does not provide the player with the full knowledge of the current state. A frame in Doom only represents a partial observation of the current state, therefore the RL agent has to learn a partially observable Markov decision process (POMDP) [45] in order to play the game with limited knowledge. Finding the optimal memoryless policy of a POMDP is an NP-hard problem [24], but as shown by [23], finding a near human-level policy for playing specific scenarios in Doom is possible with memory introduced through the use of long short-term memory (LSTM) [15]. Observations made in previous states help to describe the partially observable current state, therefore it is beneficial to introduce recurrent components such as LSTM to the DQN models. Arnold [23], a successful Doom-playing agent, made use of both a DQN model and a special DRQN model that utilizes extra game features in the training stage to improve the performance

[1] match-making: the pairing process of different players into the same match
[2] bot: the word gamers typically use when referring to these AI-controlled “players”
[3] challenge run: a playthrough of a specific game with additional restrictions usually not intended by the developers
[4] speedrun: a play-through of a specific game with the goal of completing all or part of the game as fast as possible
[5] Doom: a game released in 1993 by id Software that popularized the FPS genre

when inferencing. My agent uses a similar structure, but with a smaller memory for the agent, additional improvements such as prioritized experience replay [32], and less optimized hyper-parameters due to time constraints in combination with limitations in available resources.

I used ViZDoom [20] as the environment for training the agent. It is an open-source fork of one of the most popular source-ports[1] of Doom, ZDoom [6]. Because of its source-port nature, ViZDoom is able to provide APIs for accessing the game engine directly and several additional features (e.g. a label buffer that labels every object rendered on-screen) that were implemented to aid the training of AI agents using visual information as input.

In this report, I will discuss the structure of the DQN models in my Doom-playing RL agent, my training process for them, and evaluate my agent's performance in 3 different scenarios (custom game levels created specifically for aiding AI-related research).

2 Background

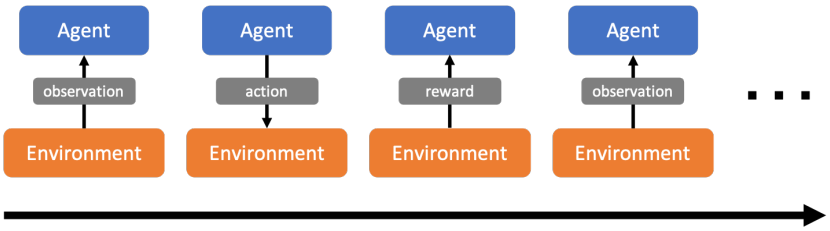

Figure 1: a diagram illustrating the training process in RL

2.1 Reinforcement Learning

Reinforcement learning (RL) is a subset of machine learning focusing on how an agent interacts with the environment in order to maximize the cumulative reward it receives. In the training stage, an RL agent would first observe the environment, choose an available action based on observation, execute the action and receive reward from the environment based on its action. Then the agent would update its decision-making policy to maximize the expected reward and repeat this cycle until a certain number of these cycles have taken place or a termination condition has been satisfied. Usually, such a cycle is referred to as a "step" and a training session is referred to as an "episode". The inferencing process is similar to training, but the agent no longer receive rewards or update its decision-making policy.

Aside from the normal rewards received from the environment that are measurements of whether the agent has completed the set objectives, another type of reward can be added to incentivise the agent for taking risky moves that are beneficial in the long run. This technique is called reward-shaping [29], the introduction of "shaping rewards" that are designed to give immediate rewards for performing an action that may benefit the agent and bring a high reward in the future.

Shaping rewards are only meant to incentivise agents into performing actions considered "better for the long run" by humans, therefore should be kept out of the training scores when evaluating

[1] source-port: software project for porting games to run on other platforms based on source code of game engines

performance to eliminate human bias. All “training score” mentioned in this report refer to the total of normal rewards an agent has received in each episode of training or testing.

2.2 Observable and Partially Observable Markov Decision Processes

Markov decision process (MDP) is a mathematical framework designed to model decision-making problems without guaranteed deterministic outcome but are not purely random. An MDP is defined as a 4-tuple (S, A, P, R) where state space S is the set of all possible states s, action space A_s is the set of all available actions in state s, state transition function $P_a(s, s')$ is the probability of the transition from state s at time t to state s' at time $t + 1$ given that action $a \in A_s$ is executed at time t and reward function $R_a(s, s')$ is the expected immediate reward of transitioning from s to s' via action a. The state transitions of an MDP satisfies the Markov property defined as "Given the present, the past and future are conditionally independent." [5] This essentially means that only knowledge of the current state is required to make an optimal decision.

MDPs are often used to model RL problems (playing Doom in this case), acting as the environment, since they satisfy the Markov property, an observation of the current state is all that is required to predict an optimal action that maximizes expected reward. The whole process is memoryless, meaning no memory of previous states are needed. FPS games however, are not fully observable, information like the position of opposing parties are hidden and the decision maker (the player) can only see the perspective projection of part of the whole game world. Therefore, instead of MDP, this specific problem is better modelled by a partially observable MDP (POMDP).

POMDP is very similar to MDP, except that the decision maker is unable to observe the current state directly, therefore only acquiring incomplete information of the current state from observations. Instead of using the current state to predict the optimal action that maximizes the expected reward, the decision maker would use the observation history to predict the optimal action in a POMDP.

2.3 Q-Learning

Q-Learning [43] is an RL algorithm that learns a Q-function $Q(s, a)$ to estimate the long-term reward of executing an action a at a current state s. The pair of state transition function $P_a(s, s')$ and reward function $R_a(s, s')$ of an MDP are often called the “model” of the MDP, Q-Learning is “model-free”, meaning that instead of using the model, it treats the “model” as the problem to solve. This allows Q-Learning to be used in situations where the “model” of an environment remains unknown, a common theme in real-world problem.

The Q-function is often presented as a Q-table, given the state space S and the action space A of an MDP, each row would represent a state $s \in S$ while each column would represent an action $a \in A$, such that each value at row s and column a represents $Q_\pi(s, a)$, the Q-value (quality value) of that state-action pair given policy π is the current decision-making policy. The higher a pair's Q-value is, the more beneficial it is considered by Q-Learning to execute that specific action given the state. Given current state s and policy π, the action chosen by would always be $\pi(s) = \max_a Q_\pi(s, a)$ given that $a \in A_s$, as Q-Learning is deterministic.

The training of a Q-Learning algorithm is performed by iteratively updating the Q-function with the formula $Q(S_t, A_t) = Q(S_t, A_t) + \alpha \cdot \{[R_t + \gamma \cdot \max_a Q(S_{t+1}, a)] - Q(S_t, A_t)\}$, with S_t, A_t and R_t denoting the current state, action chosen and reward received at time step t. Learning rate α controls how significant each update impacts the Q-function and discount rate γ is an arbitrary factor assigned to balance between optimizing towards maximizing immediate reward or future reward, both α and γ are known as hyperparameters as they are parameters not updated during training.

Q-Learning is not without its limitations, when the state space or the action space is massive (e.g. controlling a virtual character in a 3D environment), it is impractical to allocate memory for the Q-table or to train a Q-Learning algorithm due to the decision-making process relying purely on the Q-values of available state-action pairs given the current state. Several approaches were introduced to mitigate the issues presented with large state or action spaces, most notably Deep Q-Learning [26] (using an artificial neural network [17] to replace the Q-table, the method I've used in this project) and fuzzy rule interpolation [41] (replacing the discrete state-action spaces into continuous spaces and generate Q-values via interpolation).

2.4 Deep Q-Network

Deep Q-Network

In Deep Q-Learning, Deep Q-Network (DQN) is an artificial neural network (ANN) in any form that is used in place of the Q-function, this allows Q-Learning to be applied to complex RL problems with massive state spaces and action spaces. DQN would take the current state (or a time-series of state observations in the past if a POMDP is used to model the RL environment instead of an MDP) as input and produce predictions for the Q-values of all available state-action pairs (with the current state) as output. At each time step t, the loss function of a DQN would calculate the difference between the maximum of the predicted Q-value, corresponding the the chosen action, $\max \text{DQN}(S_t)$ and Q-target defined as $R_t + \gamma \cdot \max \text{DQN}(S_{t+1})$ where γ is the discount rate, and S_t and R_t are the current state and reward received at t just like in standard Q-Learning.

Usually, to encourage the exploration of different strategies, ϵ-greedy [35] is used instead of executing the action selected by the Q-function or DQN every step. At each step, there is a probability of ϵ that a random action is chosen instead, ϵ would start at 1 but decrease geometrically with every step by multiplying with a decay rate until it is decreased to a certain value called ϵ -min, generally set to 0.1, meaning at least 10% of the actions taken in the training stage are random. This would help to prevent the agent from performing the same action all the time and having the same strategy all the time. However, as indicated by my earlier models described in the Methodology section, ϵ-greedy does not guarantee that these would not happen.

Unlike a Q-table which is essentially just a mapping between state-action space and a 1D space of Q-values, a DQN is difficult to train even with ϵ-greedy. To assist the training of DQN, several techniques were proposed, most notably Double DQN [39] and experience replay [25].

Double DQN

Double DQN is the DQN version of a technique call Double Q-Learning [38]. Double Q-Learning was based on the tendency of Q function to overestimate Q-values for actions []. This is caused by the inevitable inaccuracy in estimating the long-term reward for each action: because a maximum operation is used to choose the action at every time step, the estimation inaccuracy (if positive) would be introduced to the Q-function in every update, accumulating over-estimation errors. By having two Q-functions or DQNs, each being used when updating the other Q-function or DQN, the over-estimation error can usually be reduced. The original approach set the two Q-functions to have equal probability of getting selected per training step and only the selected one is updated. However, in Double DQN, usually one model is updated at every training step using the Q-target calculated with a historical clone of itself a few updates behind. The cloned model, being a historical version of the main model, is considered less plagued by the cumulated over-estimation errors.

Experience Replay

Experience replay is a technique originally proposed for training ANNs that solve RL problems in robotics [25], but nowadays it is also widely used in areas like training AI to play games. If experience replay is used, instead of updating the model at each time step t, the agent would store the current step as a 3-tuple (S_t, A_t, R_t) consisting of the current state, chosen action and reward received. The tuple would then be added to a “replay memory” implemented with data structures like array or dequeue. Usually a replay memory without size limit is impractical as the number of entries would then be equal to the number of steps an agent is trained for. With fixed size limit, the oldest entries would always be dropped when adding to a full replay memory.

Once every step or every few steps in training, a set number of entires are randomly selected from the replay memory to update the model. Entry of the latest state would never be selected as it does not have a “next state”, required to calculate the Q-target which is needed for loss calculation. In standard implementations of experience replay, the probability distribution for choosing those entries is a uniform distribution. However, by defining a probability distribution based on the expected importance of each state-action-reward tuple (and the next state’s tuple) in training, the training time required by the model can be shortened significantly. Many definition of the expected importance in training can be defined, usually the magnitude of temporal difference (TD) error is used, so state-action-reward tuples with less TD error are “prioritized”, getting selected more frequently. This variant is called prioritized experience replay. A simplified version of prioritized experience replay is used in this project, using reward in each entry, normalized to sum up to one, as my probability distribution in selection of replay memory entries.

2.5 Special-Purpose Layers in Neural Networks

Other than the normal fully-connect (dense) layers, there exist several other layers that each serve a special purpose, here are the brief, high-level descriptions to some of them:

Convolutional Layer

Convolutional layer [11] is a kind of layer in ANN that perform a convolution of the input with one or more kernels of matching dimensions, each convolution produces a feature map that is packed with other feature maps as the output. Each kernel in the convolutional layer is “learnable”, meaning that its values would update like the weights in fully-connected layers during the backpropagation of errors. In real-world implementations, kernels don’t always produce full convolutions with the input, the kernels would sometimes move by more than one step after each convolution with a same-size part of the input, the number of steps taken after each convolution is called the “stride” of the convolutional layer.

Convolutional layers are good at extracting features from an image. By stacking them on top of each other, the first layer would extract low-level features from the input images and each subsequent layers after it would extract more abstract, higher-level features from last layer’s extracted features (see the Results and Evaluation part of this report for an example).

Pooling Layer

A pooling layer is a special type of convolutional layer with a “not learnable” kernel with specific purposes. There are several types of pooling layers, most notably max pooling [45] and average pooling [11]. Max pooling has the kernel only yielding the max value in every convolution with same-size parts of the input while average pooling has the kernel yielding the average of each

part. Pooling layers are often placed between convolutional layers to decrease the amount of data (reducing memory consumption) while aiding the extraction of higher-level features by combining multiple lower-level features into one.

Long Short-Term Memory

Long Short-Term Memory (LSTM) is a component designed specifically to deal with time-series data. Layers like convolutional or fully-connected layers don't contain any information of past inputs, resulting in poor performance when dealing with problems where past states of an input must be understood by the model. As mentioned before, POMDP problems require historical observations to describe the current state, therefore layers that introduce "recurrency" to the DQN model (such as LSTM) is required.

Explaining the implementation details of LSTM is outside the scope of this report, but in high-level terms, an LSTM layer memorizes its historical inputs by filtering inputs it has received in history with several gates that controls the "forget", "ignore" and "select" and combine them together with the current input to put into an ordinary fully-connected layer. The goal is that LSTM would learn to distinguish which informations are to be forgotten, which are to be memorized in a stream of inputs with temporal connections.

Due to its incapability to remember information for long, LSTM is generally considered to be superseded by Transformer [40]. However, I feel that understanding Transformer would have a not insignificant impact on my schedule for this project and I don't want to incorporate components I don't yet understand into my models, therefore I used the LSTM class in PyTorch [30] that was implemented with many adjustable options. I used a 2-layer LSTM in my DRQN model, which means that two LSTM layers are stacked together, with one layer's output being the other's input.

2.6 Variants of Deep Q-Network

Deep Recurrent Q-Network

A Deep Recurrent Q-Network (DRQN) is a variant of DQN that introduces recurrency to the network by the incorporation of a Recurrent Neural Network (RNN) [3], usually in the form of one or more Long Short-Term Memory (LSTM) units. No incorporation of more recent (attention-based) structures like Transformer is partially due to the fact that researches in building FPS-game-playing AI have met a bottleneck in recent years.

Dueling Architecture

A dueling architecture in DQN [42], in simple terms, means that the whole DQN model consists of two sub-models and a series of convolutional layers. Both sub-models would share the output of

the convolutional layers, one of them would produce a prediction of each action's Q-value just like in standard DQNs, but the other sub-model would produce a scalar output predicting the "value" of each state, acting as a estimator for the value function $V(s)$ of the MDP. The value function measures how desirable it is to stay in each state and the Q-function $Q(s, a)$ (and consequently the DQN used in place of it) estimates the output $V(s')$ of value function given that state s' is transitioned from s by executing a. The predicted state value and the predicted Q-values are multiplied together to form the final output action values. A dueling architecture is not used in this project due to time constraints, but can definitely be applied to improve performance.

2.7 Other Approaches

Several other algorithms can be used in the RL agent, most notably Proximal Policy Optimization (PPO) [33], Advantage Actor Critic (A2C) [27], Asynchronous Advantage Actor Critic (A3C) [27] and Soft Actor Critic (SAC) [13]. All of them have been applied to 3D games, generally yielding better results than DQN-based models.

In early stages of this project, I have planned to try out A2C and PPO (A3C is not viable for the limitation in computational resources and I have not researched SAC enough to consider using it), but importing them from libraries such as Stable Baselines 3 (SB3) seemed too restrictive (with only several fixed model structures) and would not teach me how to implement them, while trying to learn how they function and implementing them is impossible given my planned schedule for experiments with DQN-based agents. Instead, I explored the possibilities of implementing Asynchronous Q-Learning as described in the paper that proposed A3C, utilizing multiple Double DQN agents training in their own instance of the environment, sharing the same target network (the secondary model used for loss calculation in Double DQN), essentially increasing the search space of solutions (which are optimal DQN models to play the scenario). However, each agent needed to be implemented as a thread in CPU, given the low core-count and low power consumption of my laptop's mobile CPU, along with the limit in available memory and storage (I only had ~170GB available and saving a single agent with it's training memory takes at least 2GB per epoch), it's just not very viable to implement even without time constraints.

A previous Final Year Project (FYP) from Cardiff University [18] conducted a research similar to mine (training an RL agent that plays Doom) and had utilized A2C and PPO from SB3, but it was more focused on hyperparameter-tuning and included shaping rewards into the calculation of total rewards for each episode of training, therefore their results on Deadly Corridor (a scenario I have also tested in) are not comparable to mine.

2.8 Doom

Graphics

Doom is a video game released in 1993 that utilized a technique known as binary space partitioning (BSP) [28] to render a pseudo-3D world. Each scene the player can enter is defined as a convex set of points in a 2D plane with x and y coordinates and “lines” connecting two points in the set acting as walls, multiple lines can be combined to create “sectors” of polygonal shape with varying floor height and ceiling height. BSP functions by building binary trees via recursive subdivision of convex sets with each tree representing a scene. Trees are pre-calculated by developers similar to the pre-compilation of shader caches in modern games, the game engine uses these trees in real-time to determine the set of visible wall surfaces needed to be drawn onto the screen and the order to draw them in.

Each surface is rendered as textured vertical lines with length proportional to its distance from the player, therefore all surfaces are perpendicular to the flat ground and it’s impossible to look up or down in game. Lines can be single or double sided and can have a maximum of 3 texture maps, applied to its predefined upper, middle and lower part. Sectors have “texture maps” for its ceiling and floor, so structures like stairs can be represented with several sectors of varying floor height and identical floor texture. Due to hardware limitations, ceilings and floors do not have unique texture spaces, the game space is used instead, therefore there is no way to align the textures for floors and ceilings, the sectors themselves have to be in the right position instead to achieve alignment. To compensate for the simplistic environment, 2D sprites are used extensively to represent enemies and act as environmental details or decorations.

Due to the abovementioned limitations, Doom is graphically simple by modern standards and is suitable as the “Hello World” program in the task of training AI agents to play FPS games. It is however important to note that games rendered in real 3D would be significantly more complex to play due to the introduction of 3D polygons, surfaces not perpendicular to the ground, uneven ground height and more. Enemy detection specifically would be a much harder task when instead of eight 2D sprites representing 45° rotations, a near-infinite (limited by the range of floating-point numbers) amount of possible orientations are presented with 3D models.

Game Mechanics

Weapons in Doom all fall into two categories: hitscan and projectile. Hitscan weapons complete a hit scan by generating rays for hit detection when they are fired and any actor (both players and enemies) touching the rays at that frame would receive damage proportional to the number of rays they touch, the damage varies by a small amount randomly to given a sense of uncertainty when dealing or receiving damage. Projectile weapons create physical projectile objects when

fired, the projectiles are designed to not deal damage when hitting the type of enemy that created them or specific types of enemies if explicitly scripted in the game engine. Enemy hitscan weapons have higher damage and firing rate at harder difficulties, acting as the main source of difficulty increase due to their unavoidable nature; but projectile weapons also receive the same enhancements, often also doubling in flying speed of projectiles, just that they are easier to avoid. All attacks in Doom are done by weapons, even melee attacks from enemies and player (by equipping fist) are just hitscan weapons with a short range.

There are 5 difficulty settings in Doom, also known as the 5 skill levels, the agent is only trained on and tested on level 4 (Ultra Violence) and 5 (Nightmare) since most scenarios in ViZDoom would be too easy using the first 3 settings. The original release of Doom only contained 4 difficulties and Nightmare is later added to the game with the warning “Are you sure? This difficulty level isn't even remotely fair” after a developer read several posts on online forums complaining Ultra Violence is not difficult enough [7]. Compared to Ultra Violence, Nightmare monsters move and attack much faster (up to 100%), no longer take a few steps before attacking when they heard or saw the player, and respawns after 30 seconds (Doom runs at 35 ticks/frames per second, so 1050 ticks). These added difficulties provide a unique challenge for RL agents playing Doom.

Alternative Games and Justification for Choosing Doom

There are other FPS games to train my agent with, several previous studies like [31] have targeted other famous FPS games in the 1990s such as Unreal Tournament[1] (UT), however most of them rely purely on in-game information to make decisions and control the in-game characters with high-level actions such as “ChaseEnemy” (run to the closest visible enemy) or “ShootPrimaryAtEnemy” (fire weapon towards the closest enemy). My goal is to create an RL agent that relies purely on visual information a human player can get and controls the in-game character using strictly button inputs that a human player would have access to, therefore POGAMUT [12], the middleware platform those studies relied on to communicate with UT could not be used as it doesn’t provide access to frame buffer, audio buffer or button inputs.

From the easy-to-develop standpoint of building a training platform from scratch, a more feasible target would be FPS games developed with the Unity game engine [37]. Unity games are relatively easy to modify when using a tools known as Bepis Injector Extensible (BepInEX) [4]. Games using both scripting backends of Unity, Mono (just-in-time compiler) and IL2CPP (ahead-of-time compiler) can be modified by injecting a GameObject instance into Unity and executing custom codes from the instance [10]. The platform-specific problem with this approach would be the in-

[1] Unreal Tournament: a game released in 1999 by Epic Games and Digital Extremes

ability to train agents at reasonable speed or even inferencing them at real-time after training as Unity game engine is relatively modern.

Even if the mentioned difficulties are somehow mitigable, there are currently no success in published studies about using RL-based AI to play FPS games newer than Doom, at least not by using high-dimensional sensory inputs such as visual information instead of in-game features. Due to the drastic increase of complexity as mentioned in the Graphics section, I doubt that I would be able to get any meaningful result out of a project conducted with any newer FPS games.

2.9 ViZDoom, Level Editing and ACS Scripting

ViZDoom is an open-source project designed specifically for training AI agents using visual information, allowing programs written in C++ or Python to interact with the game engine directly and accessing internal variables. ViZDoom is highly customizable in its rendering options, the resolution, color depth and various frame buffers for different on-screen elements can all be configured to avoid wasting resources on rendering unwanted pixels.

Additional buffers are also provided to aid the training: depth buffer provides a grayscale heatmap indicating the distance between the player and each pixel, audio buffer provides the audio information human players have access to while playing and labels buffer provides labels for all visible objects currently rendered.

To train an agent, custom levels are designed specifically to cover different aspects of the gaming experience, such as engaging in combat with enemies in a corridor or fighting hoards of demon in an arena, these levels are called scenarios in ViZDoom. A scenario consists of a .cfg configuration and a .wad WAD file. The configuration file contains default rendering options for the scenario, timeout for each episode (in unit of tics), list of available button inputs, list of available in-game variables and the values for living reward or death penalty. Maps inside the level (usually just 1) and behavior scripts that control scripted events are stored in the WAD file.

An open-source software called SLADE [19] can be used to open or modify a WAD file, it provides a graphical user interface for editing the map and an integrated development environment (IDE) for the editing and compiling of behavior scripts. The behavior scripts are usually written in a scripting language called Action Code Script (ACS) [8], originally developed by Raven Software for their 1995 game Hexen utilizing the same game engine as Doom, nowadays it is mostly supported by the modding[1] community. An Action Code Compiler (ACC) is required to compile ACS scripts within SLADE. The official release of Doom doesn't have a support for ACS scripting, but support was later added to source-ports like ZDoom, and consequently ViZDoom.

[1] modding: the act of modifying contents of an existing game for entertainment or special use cases

Most scenarios rely heavily on scripted events to function and calculate reward, therefore it is possible to modify scenarios by editing and recompiling their ACS scripts. The original map format of Doom does not support ACS, therefore, if a map from the original game is considered useful as a scenario, it must first be converted into the Universal Doom Map Format (UDMF) [9].

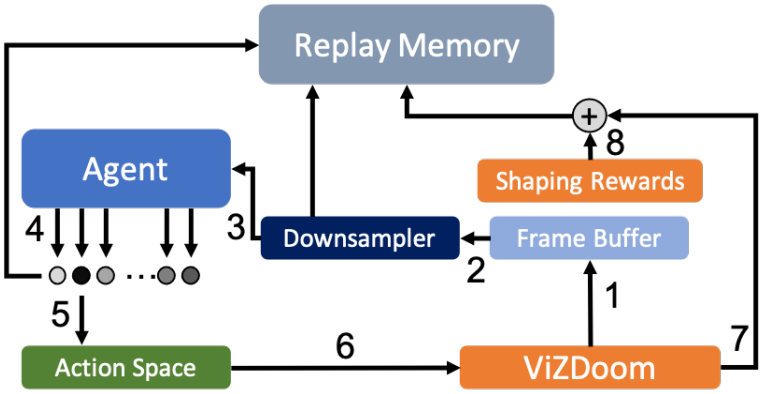

Figure 2: dataflow diagram illustrating the training process of RL agent, the state-action-reward tuple is saved into replay memory after step 8

3 Methodology

Detailed descriptions for scenarios I have used when training and testing the final version of my RL agents are included in the Results and Evaluation section. This does not include the **hanger** scenario as it was removed in a later stage of the project in favor of a new deathmatch scenario.

3.0 Overview

As stated before, the goal of this project is to design and train an RL agent such that it is able to use only visual informations to play certain levels in Doom via button inputs available to human players. As displayed in Figure 2, in each step, the environment (ViZDoom) would first render the current frame inside the frame buffer, then the content of the frame buffer would be downsampled by a downsampler (an algorithm like bilinear interpolation) and sent to the agent as the input. The agent would predict the Q-values for all available actions in the action space and the action with the highest Q-value would be executed in ViZDoom. After running the in-game logic, ViZDoom would calculate the normal reward the agent receives in the frame after executing the action and the shaping rewards would be calculated based on ViZDoom's internal variables. The two types of rewards are then added to produce the (combined) reward for this state-action-reward tuple (or frame-action-reward tuple since a frame is just an observation of the current state), the tuple is then added into the replay memory. After these, several states would be sampled from the replay memory and be used to train the agent, either using standard or prioritized experience replay.

3.1 Initial Attempt with DQN only

The agent contained only a standard DQN model in my initial attempt. The model was similar in structure and training procedure (shaping reward wasn't in used yet) to the one in the example PyTorch program provided in ViZDoom's GitHub repository. It had two color channels in its input: a 320x240 grayscale image of the current frame and a 320x240 depth map for the current frame fetched from the depth buffer provided by ViZDoom. A max-pooling layer was placed at the start to downsample the 2x240x320 input into 2x60x80, two subsequent convolutional layers were

used to extract the features and a fully-connected hidden layer connected to the output layer to predict the Q-values for each available action.

The max-pooling layer while providing better performance than the downsampling function used in the example program, had a negative impact on memory usage as full-sized grayscale images and depth maps of every frame were stored in the replay memory, limiting replay memory's maximum size to about 40,000. This lead to a situation where, if the agent was still unable to learn a "good enough" strategy after 40,000 steps, it would "forget" the randomized steps when ϵ value was high and stop learning entirely as most states stored in the replay memory are associated with high, negative rewards.

Part of the first level "hanger" in Doom's retail release was converted to a UDMF-format map and a **hanger** scenario of the first room was created via ACS scripting in SLADE. The goal for this scenario is to avoid taking damage from enemies at the left side of the room while also advancing into a curved corridor with an exit door at its end. The victory condition is to obtain a green armor pickup at the exit door before the time limit was reached. Several explosive barrels scattered throughout the room, providing challenges by blocking the path and having the potential to instantly kill the agent if shot by an enemy. Only move forward, turn left and turn right were available as button inputs in this scenario to simplify the navigation task. The skill level was set to 4.

This agent was able to learn and beat the **deadly corridor** scenario semi-consistently at skill level 1 to 3 within a reasonable amount of time, but similar to the results obtained in the other FYP mentioned in the Other Approaches section, this agent struggled to learn level 4 and 5. Results for the **hanger** scenario were initially promising, but a gradient explosion would always occur at some point in the training process, leading to a badly-performing agent constantly turning in both directions and occasionally moving forward.

Gradient explosion is the situation where the weights and other learnable parameters in an ANN are constantly growing in magnitude due to the large gradient values associated with them in the backpropagation of losses, resulting in a highly unstable model sometimes also having NaN values in learnable parameters if left training in this state for long. This is usually caused by the model receiving too much loss or loss without a clear meaning, both are true in this case. In retrospect, multiple incorrect decisions from my inexperience contributed greatly to this result, the learning rate was set way too high for playing Doom levels (contributing to the large gradient values in backpropagation) and my decay rate for ϵ was set too low, resulting in an agent that rapidly loses help from the randomness of ϵ-greedy while the decisions it made were still worse than executing random actions.

3.2 Various Modifications Applied to the DQN Model

The 1st revision of my agent saw the introduction of Double Q-Learning, it had no effect on the training scores and the gradient explosion problem despite acquiring twice the video memory in the GPU.

The 2nd revision made use of a deeper DQN model with added convolutional layers and fully-connected layers, it had slightly worse training scores than before and still suffered from all the abovementioned problems despite occupying more video memory.

The 3rd revision used an additional feature map in the input. It was an edge map produced by the convolution of the input grayscale image and two 2D sobel kernels. The idea was that this would help the DQN model if its convolutional layers could not learn to produce feature maps for the edges, this did yield a marginal increase in training scores but the increase was negligible.

3.3 Modifications Applied to the Training Process

The 4th revision focused on improving the training procedure instead of the DQN model predicting Q-values. Shaping rewards were added to both the **deadly corridor** scenario for killing enemies and the **hanger** scenario for moving in the direction of the exit door, an idle penalty was also added to the **hanger** scenario in the form of a negative shaping reward. The **hanger** scenario saw almost no improvement, but a newly-learnt behavior in the **deadly corridor** scenario became popular for skill level 5. The agent learnt to consistently take a low but positive shaping reward by killing one enemy semi-consistently and turning to face a wall after that. It associated seeing enemies with getting killed and without another shaping reward incentivizing it to preserve health it failed to associate facing the wall while doing nothing with dying.

3.4 In Search of Other Approaches

The possibilities of switching to SB3 and directly importing the PPO and A2C models were discussed with my supervisor but I decided to continue with DQN-based models. Two options still remained: Asynchronous Q-Learning and DRQN.

Asynchronous Q-Learning would take a fairly long development and debugging time while underperforming with the limited amount of cores and limited power consumption of my laptop's CPU. Thus, the idea was abandoned.

A rewrite of my codebase was scheduled to implement a special architecture described in [23]. Performance-wise, the rewrite was a success, significantly improving CPU utilization, CPU temperature during training increased from 70-75°C to 93-100°C as a result. Replay memory also saw a 9x increase in maximum size by the use of lower-precision data types such as 8-bit un-

signed integer or 16-bit floating-point, these are sufficient to store the data at the original precision, but needed to be casted to 32 or 64 bit floating-point values when used in training.

3.5 Second Attempt with DQN and DRQN Combined

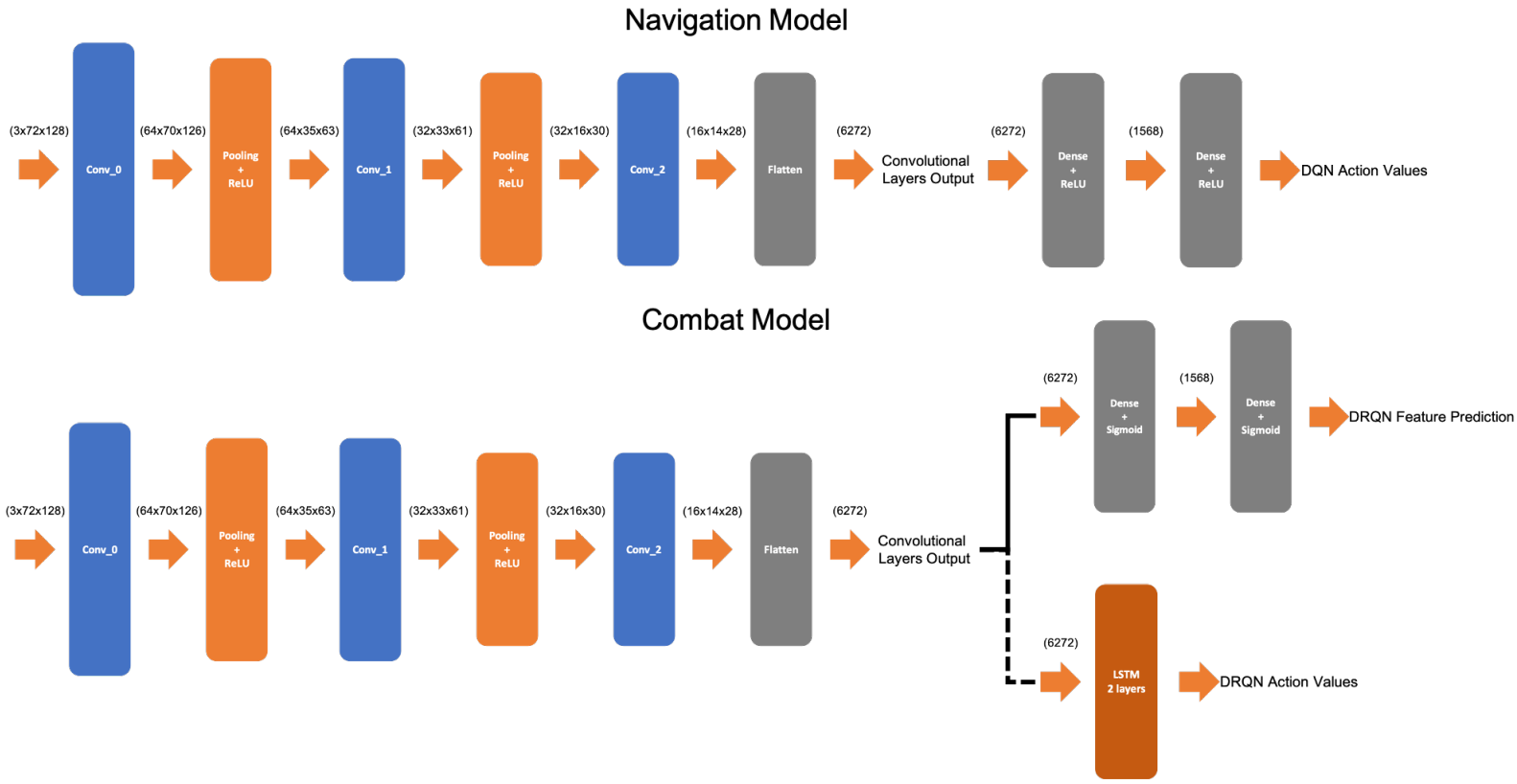

Figure 3: an illustration of the two models used in my latest agent. At each step, either the DRQN model (bottom) or the DQN model (top) is used as the decision maker based on whether enemy is present in the current frame. In training, enemy presence is provided by the labels buffer of ViZDoom but the DRQN model would predict the presence in inference.

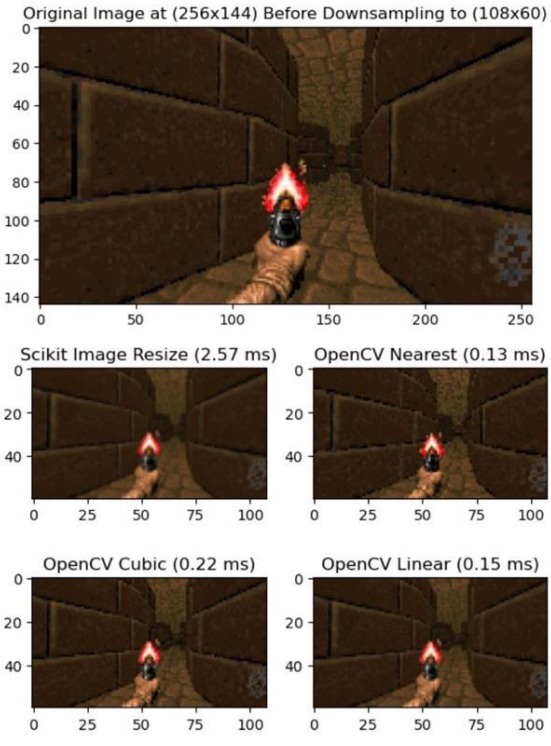

Figure 4: comparison between the algorithm used in ViZDoom's example program and three interpolation options (nearest, bicubic, bilinear) in OpenCV when downsampling

The final revision of my agent made use of two models: a DRQN model specifically trained for combat and a DQN model trained for navigation. Instead of 320x240, a rendering resolution of 256x144 is set, this is the minimum resolution with a 16:9 aspect ratio. Aspect ratio determines the viewing angle for each frame, 16:9 has the largest viewing angle [23] and is preferable as more information is provided to the agent at every frame. The image is then downsampled to 128x72 via bilinear interpolation as it provides a good balance between detail preservation and performance (see Figure 4 for comparisons).

Both the combat and the navigation model take an RGB input with size 3x72x128, the depth map and sobel edge map used in previous revisions were removed. The combination of three convolutional layers and two average

pooling layers between them allowed for the combination of lower-level features into fewer, higher-level features, a flatten layer is added at the end to flatten the output feature maps from to 1D. This 6-layer "feature extractor" structure is present in both models for feature extraction.

Unlike the DQN model for navigation, the DRQN model for combat splits into two streams after the feature extractor (as illustrated in Figure 3). A copy of the extractor's output is created and fed to a two-layer LSTM for predicting Q-values for available actions while the original output data is fed to two fully-connected layers to predict game features. In my case, only one game feature is used: the presence of enemy on-screen. In the training stage, if one or more enemies appear in the current frame, the game feature is set to True and combat model is used instead of navigation model, but in inference stage the agent is not supplied with the game feature and the combat model would predict whether there are enemies on screen. A simplified version of the training procedure used by this agent is included as Appendix P1.

Prioritized experience replay is implemented in the replay memory after the code rewrite, taking the reward values (the sum of shaping and normal reward at each state) as the expected importance for each state. Indices of the chosen states are returned when sampling the states to train my agent with, instead of the frame-action-reward pairs. This is due to the changes that have been made to the training procedure in order to better model the POMDP environment (ViZDoom) for the combat model. The navigation model still treats the environment as an MDP.

As mentioned in the Background section, in POMDP multiple historical observations (frames in this case) are needed to more accurately represent the current state. Therefore, I have introduced three parameters: state length s_{len}, history length h_{len} and padding length p_{len}. s_{len} is defined by user to be an even number > 3 while $h_{len} = s_{len} - 2$ and $p_{len} = \frac{s_{len}}{2}$ are calculated automatically. As shown in Figure 5, a group of s_{len} states is fetched for each index i returned from the replay memory when sampling states for training. For each group of states, the agent is trained for p_{len} times. In the case of Figure 5, s0 and s1 would first be used as historical observations, s2 as an observation of the "current state" and s3 as an observation of the "future state", so for the calculation of Q-target used to update the combat DRQN model, the current state would be represented by s0 to s2 while the future state would be represented by s1 to s3. If the observation of the "current state" is labelled to be a navigation one, s2 and s3 would be used as the current and future state when calculating the Q-target for updating the navigation DQN model. This process would repeat with s3, s4, s5 acting as the obser-

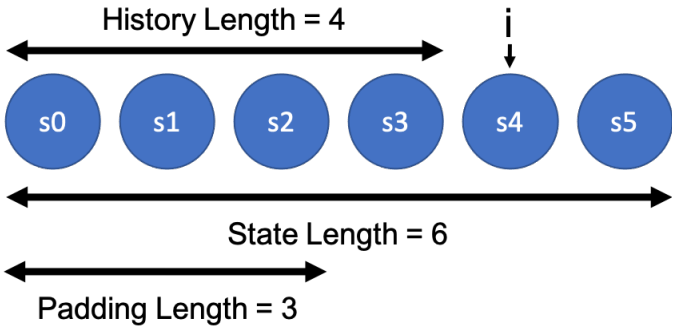

Figure 5: an example diagram for when state length = 6

vation for the future states.

The main difficulty encountered in designing such a training procedure is that navigation frames usually are sparse and non-contiguous as enemies frequently show up in the agent's view, as a result, a DQN model is used for navigation instead of the same DRQN model used for combat.

It was discovered in [23] that a special layer called dropout [34] is crucial in training the agent to detect enemies in sight. Therefore I have added dropout layers in the LSTM and after every convolutional layer with a dropout probability of 50% (same value as the original paper [34], as I don't have the time to experiment with different values). A dropout layer is simply a "filter" that has a probability to replace each value that passes through it with 0. With a dropout probability of 50%, half of the values are expected to be turned to zero.

4 Results and Evaluation

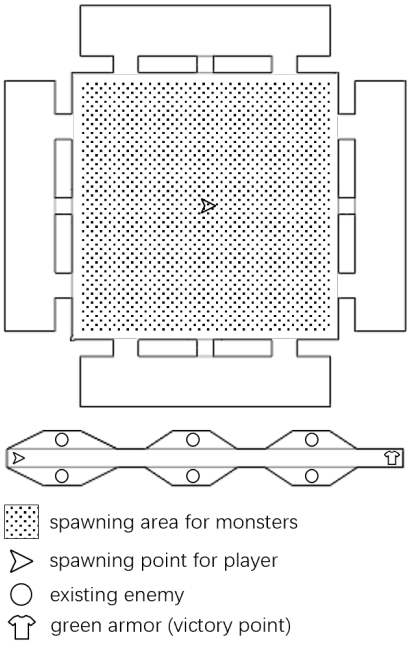

Figure 6: maps for deathmatch (top) and deadly corridor (bottom)

4.0 Scenarios

I trained my agent in 3 scenarios:

- **Deathmatch** (modified)

- **Deadly Corridor** (unmodified)

- **Deadly Corridor** (modified)

Deathmatch and deadly corridor are scenarios provided by ViZDoom and I modified them to create 2 of my training scenarios.

For all 3 scenarios, I removed the agent's ability to turn at any speed in order to mimic the experience of playing the original release of Doom without mouse support. The control for navigation mode also mimics how the original control configuration "tank controls" works, a player would rest three fingers on the arrow keys and control move forward, turn left, turn right respectively.

Deathmatch (Modified) (Skill level 5 - Nightmare Difficulty)

Deathmatch is an empty arena with a probability of spawning enemies every 15 frames (0.43 seconds at Doom's fixed frame-rate of 35 fps), enemies are scripted to remain dead after killed despite the nightmare difficulty setting. The agent starts with 100% health, 0% armor, a shotgun and max ammo (50). Armor is a special property in Doom, 1/3 of the damage a player receives is absorbed by armor until it reaches 0%. The agent deals double damage, receives half damage in this scenario and is granted a bonus reward of 20% health, 20% armor and 50 ammo upon killing

any enemy. This reduction in difficulty is given to compensate for the removal of the abundant health, armor, ammo and more powerful weapon pickups available in the original scenario. I removed all pickups and the decorative crater at the centre of the map to simplify the task for my agent to learn due to limitations in computational resources and available memory. The texture for this map has been modified to show less aliasing artefact at low resolution (e.g. wobbly lines that can be observed at distant brick walls in Figure 7) and having four white strips that act as a reference object in the 3D world, a controlled experiment has been conducted to prove the new textures' positive effect on my agent's performance.

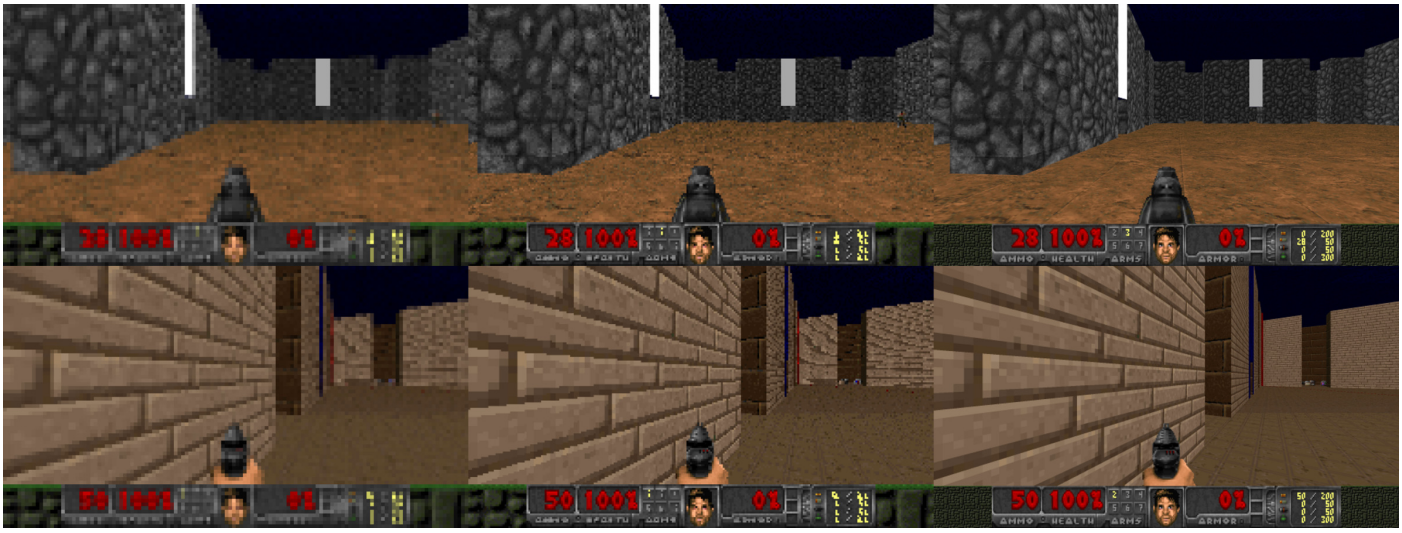

Figure 7: a comparison of the modified (top) and the original (bottom) texture in deathmatch scenario with three different resolution: rendered frame at 256x144 (middle), bilinear downsampled image at 128x72 (left), 1080p ground truth (right)

Spawning probability for each enemy type:

Zombieman	Shotgun Guy	Marine (chainsaw)	Chaingun Guy	Demon	Hellknight
4%	4%	2%	2%	1%	0.1%

Access to button inputs:

	fire	move forward	turn left	turn right	move backward	move left	move right
Combat Mode	☑	☑	☑	☑	☑	☑	☑
Navigation Mode	☐	☑	☑	☑	☐	☐	☐

Weapon availability:

Available Weapon	Attack Type (Hitscan or Projectile)	Method for Obtaining Ammo
Shotgun (Start With)	Hitscan	Killing Enemy Refills

Reward definition:

Event	Reward	Reward Type	Bonus (normal) / Explanation (shaping)
killing a zombieman	+10	Normal	+20 health, +20 armor, ammo refill

killing a shotgun guy	+30	Normal	+20 health, +20 armor, ammo refill
killing a marine (chainsaw)	+30	Normal	+20 health, +20 armor, ammo refill
killing a chaingun guy	+40	Normal	+20 health, +20 armor, ammo refill
killing a demon	+30	Normal	+20 health, +20 armor, ammo refill
killing a hellknight	+100	Normal	+20 health, +20 armor, ammo refill
losing x% health	-x	Shaping	encourage agent to preserve health
being in combat mode	+5	Shaping	encourage training combat model
being in navigation mode	-5	Shaping	discourage training navigation model

Deadly Corridor (Unmodified) (Skill level 4/5 - Ultra Violence/Nightmare Difficulty)

Deadly corridor is a simple but also difficult scenario: map structure and victory goal are simple, yet even experienced Doom players like myself would struggle to beat it. The agent, scripted to take double damage, starts at one end of a narrow corridor with 100% health, no armor and only a pistol. 6 enemies are situated at the two sides of the corridor, 3 at the left and 3 at the right, most of them have hitscan weapons that are able to kill the agent in one hit if standing close enough. Because enemies at nightmare difficulty has no delay between seeing or hearing the agent and shooting it, doubled firing rate (compared to ultra violence) and the ability to respawn in 30 seconds after killed, it is almost impossible to beat with my agent. As a result, I have trained my agent on all difficulty settings in this scenario to investigate the difficulty and see if I can understand it.

Access to button inputs:

	fire	move forward	turn left	turn right	move backward	move left	move right
Combat Mode	☑	☑	☑	☑	☑	☑	☑
Navigation Mode	☐	☑	☑	☑	☐	☐	☐

Weapon availability:

Available Weapon	Attack Type (Hitscan or Projectile)	Method for Obtaining Ammo
Pistol (Start With)	Hitscan	None

Reward definition:

Event	Reward	Reward Type	Explanation
moving +dx units in x-axis	+dx	Normal	encourage moving toward exit point
moving -dx units in x-axis	-dx	Normal	discourage moving toward start point

picking up green armor	+1000	Normal	reaching exit point
dying	-100	Normal	discourage agent to die
killing a zombieman	+50	Shaping	encourage agent to kill enemy
killing a shotgun guy	+50	Shaping	encourage agent to kill enemy
killing a chaingun guy	+50	Shaping	encourage agent to kill enemy
losing x% health	-x	Shaping	encourage agent to preserve health
being in combat mode	+5	Shaping	encourage training combat model
being in navigation mode	-5	Shaping	discourage training navigation model
having green armor in view when in navigation mode	+15	Shaping	encourage agent to learn that moving towards green armor is key to success

Deadly Corridor (Modified) (Skill level 5 - Nightmare Difficulty)

The original deadly corridor is difficult even for many human players and I don't consider it representative for Doom's combat and level design. In this modified version of deadly corridor, the agent starts with a shotgun instead of the pistol and does not take double damage from enemies.

Access to button inputs:

	fire	move forward	turn left	turn right	move backward	move left	move right
Combat Mode	☑	☑	☑	☑	☑	☑	☑
Navigation Mode	☐	☑	☑	☑	☐	☐	☐

Weapon availability:

Available Weapon	Attack Type (Hitscan or Projectile)	Method for Obtaining Ammo
Shotgun (Start With)	Hitscan	None

Reward definition:

Event	Reward	Reward Type	Explanation
moving +dx units in x-axis	+dx	Normal	encourage moving toward exit point
moving -dx units in x-axis	-dx	Normal	discourage moving toward start point
picking up green armor	+1000	Normal	reaching exit point
dying	-100	Normal	discourage agent to die
killing a zombieman	+50	Shaping	encourage agent to kill enemy
killing a shotgun guy	+50	Shaping	encourage agent to kill enemy
killing a chaingun guy	+50	Shaping	encourage agent to kill enemy
losing x% health	-x	Shaping	encourage agent to preserve health

being in combat mode	+5	Shaping	encourage training combat model
being in navigation mode	-5	Shaping	discourage training navigation model
having green armor in view when in navigation mode	+15	Shaping	encourage agent to learn that moving towards green armor is key to success

Empty Corridor (Skill level 4 - Ultra Violence Difficulty)

A special version of the Deadly Corridor scenario was created to provide special training episodes for the navigation model. This is mostly due to the fact for Deadly Corridor, there is almost always at least one enemy in the agent's view, making the navigation model had to train. In Empty Corridor, enemies spawn with 0% health and die instantly, level 4 was used instead of 5 due to the undesirable 30-second enemy respawn mechanics. The access to button inputs and reward definition are identical to the Deadly Corridor (modified or unmodified) scenario.

4.1 Standards for my Experiments

My experiments were all carried out using the last revision of my RL agent, training scores for earlier attempts were not collected and stored, therefore could not be analyzed.

All of my experiments were carried out using these settings:

- dropout probability = 0.5
- discount rate = 0.99
- ϵ-decay = 0.99995
- ϵ-min = 0.1
- frame repeat = 4
- state length = 10
- rendering resolution = 256x144
- downsample target resolution = 128x72
- downsample algorithm = Bilinear Interpolation
- optimizer = Adam
- loss function = Huber Loss
- combat action space = all possible actions except for idle
- navigation action space = { (turn left), (turn right), (forward, turn left), (forward, turn right) }

Justifications for using these settings:

For the discount rate and ϵ-min, I picked the most widely-used values in the works I've cited as reference. The ϵ-decay value was obtained through trials and errors, it ensures that the randomness of ϵ-greedy would help the agent until it starts to learn a good policy without dragging the training time too long.

The frame repeat parameter (also known as frame-skip) [20] essentially controls the "frame-rate" of the game from the agent's perspective. Having it set to 4 means that after the agent observes a frame and decides on an action to execute, the action is executed for four frames before the agent can make the next observation. With Doom's frame-rate being 35 fps, the frame-rate in the agent's perspective would be 8.75 fps. A higher frame-skip lowers the precision for aiming significantly (the agent is more likely to rotate over the target) while a lower frame-skip slows the training time down geometrically. I chose 4 because it was mentioned to have the best trade-off between training time and agent's performance in [23].

Due to the time constraints for this project, I decided to only use one optimizer and one loss function throughout my experiments. Adam optimizer [21] was chosen for its reputation to converge fast and Huber loss [16] was chosen because it was less sensitive to outliers in data compared to other loss functions like L1 loss (Manhattan distance between two vectors) and L2 loss (Euclidean distance between two vectors). Since the quality of actions my agent would choose varies a lot, Huber loss's property of being less sensitive to outliers would be desirable.

4.2 Learning Rate

Learning rate is a hyperparameter generally having significant impacts on the performance of Machine Learning (ML) models, therefore, I started my investigation with several controlled experiments using learning rate as my variable.

Prior Investigations

My initial candidates for suitable learning rates were 0.1, 0.01 and 0.001. The reason was simple: the convolutional neural networks I have worked with in the past worked well with learning rates between 0.1 and 0.01 and Adam, the optimizer I am currently using, has a default learning rate of 0.001 in both TensorFlow [2] and PyTorch. Unfortunately neither of them worked in all 3 scenarios and all experiments failed as the training scores quickly approached the lowest possible values.

I decided to have a smaller scope and investigate on the Deathmatch scenario next. After several attempts at using random numbers I came up with as learning rates, I found that 0.00005 worked well in deathmatch and proceeded to set up another experiment, this time with 0.00001, 0.00005, 0.00025 as the learning rates. The model with 0.00005 seemed to work the best while 0.000001 struggled to improve and 0.00025 saw a gradual decrease in training scores after a small peak. It

was at this time that I found that I have implemented my training procedure and loss calculation slightly incorrect.

While fixing the issues, I took the time to implement a system that saves all training scores and kill count per episode for analysis and visualization, labelled by date and time. A mean to send the statistics and diagrams of training scores and kill counts (or error message if something went wrong) to my phone was added to the system so that I could stay away from my 93-100°C laptop and still see updates for the current training task (Figure 8).

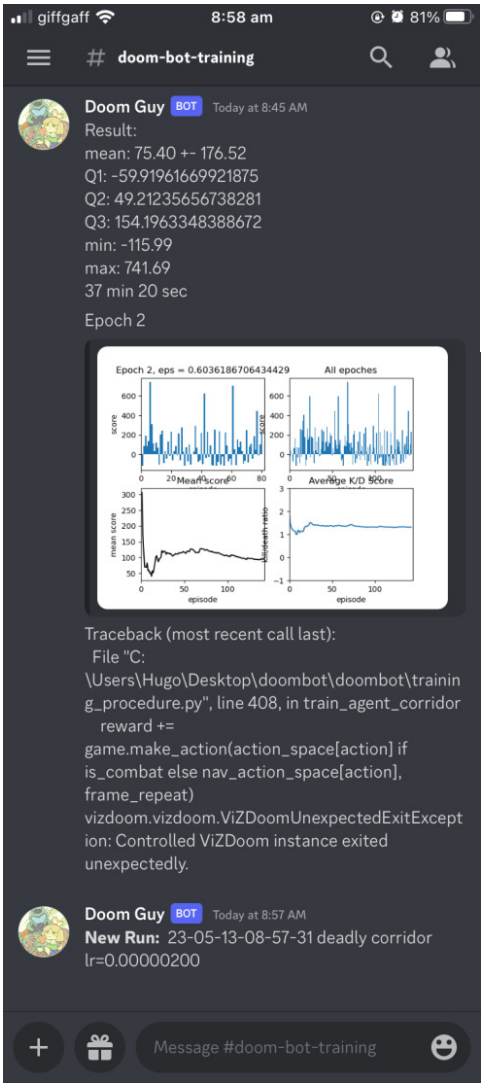

Figure 8: the system in action

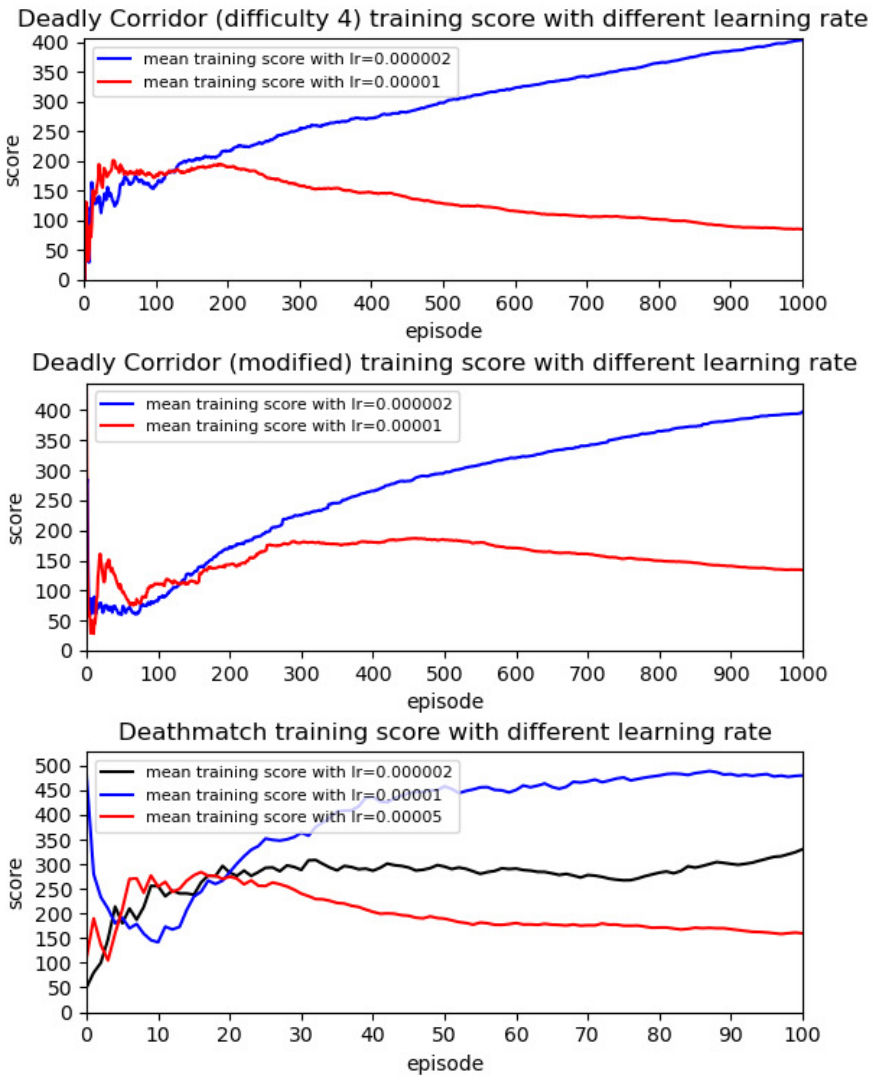

Figure 9: average training scores with different learning rates

Patterns in Training Scores with Respect to Learning Rate

After the issues with training procedure and loss calculation were resolved, the previous experiment was repeated with the same learning rate values. The average training score graphs of 0.00001 and 0.00005 exhibited similar patterns with the same graphs of 0.00005 and 0.00025 in the previous attempt. The training with 0.00025 was replaced with 0.000002 to investigate further into the phenomenon and it also showed similar patterns with the average training score graph of 0.00001 in the previous attempt.

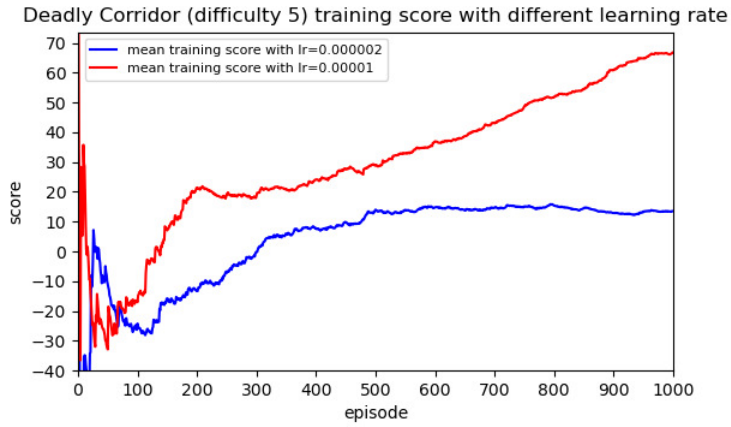

Figure 10: a slightly different pattern probably due to difficulty

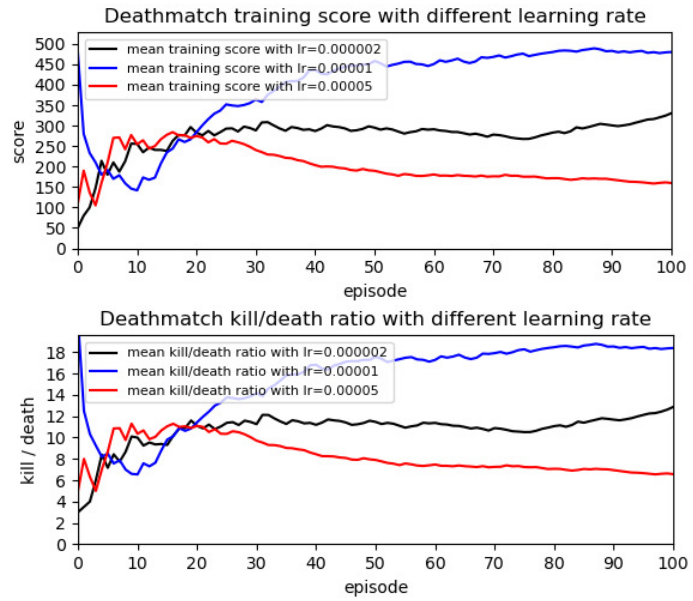

Figure 11: the two graphs are almost identical except for scale

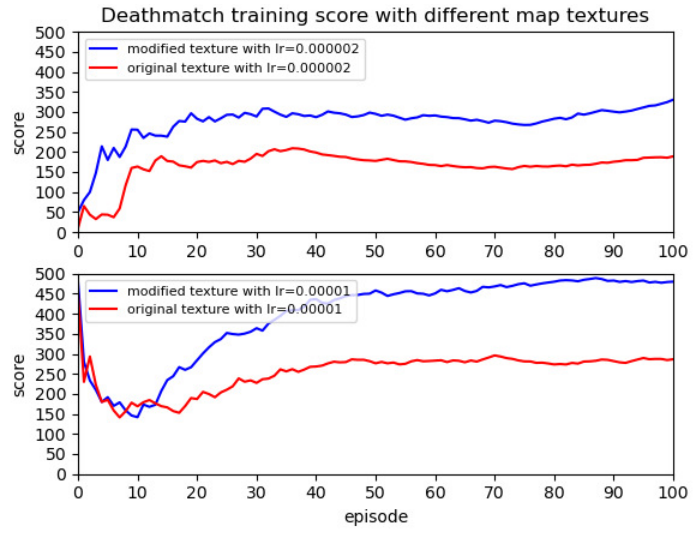

Figure 12: difference in performance with different textures

It is likely that these patterns are indications for learning rates that are too low (0.000002), suitable (0.00001) and too high (0.00005). I decided to train my agent in other scenarios to confirm this hypothesis. Both the modified version of Deadly Corridor at skill level 5 and the original version at level 4 have shown mostly similar patterns, confirming my hypothesis (Figure 9).

The original Deadly Corridor at level 5 (Figure 10) had a slightly different pattern from others, but it is possible that it was too difficult for the agent to learn and a longer training session is required to see the pattern for it.

Another interesting finding is that the average training score graph and the average kill-death ratio (number of kills / number of deaths) graph for Deathmatch are almost identical (Figure 11), indicating that the agent's ability to kill each type of enemy was consistent throughout the whole training process as different types of enemies have different killing rewards.

4.3 Impact of Aliasing Artefacts

As shown in Figure 7, the textures of the original Deathmatch scenario has been modified to show less aliasing artefact at low rendering resolutions like 256x144 and 128x72. The texture maps for floors and walls have been replaced with ones that don't have high-density lines while 4 white stripes are added to the walls, acting as reference objects in space. This additional spatial relation information can potentially be learnt by the agent to navigate the scenario better. A controlled experiment was conducted in the Deathmatch scenario with the two sets of textures, showing an increase in training scores by 40% to 60% (Figure 12). This approach is more viable than applying anti-aliasing al-

gorithms since super-sampling beats the purpose of using lower resolutions in the first place and algorithms based on temporal information would produce an even blurrier frame than the low resolution input the agent is already receiving.

4.4 Importance of Luck in Early Episodes

It has been observed that if the agent is unable to get a few “high enough” scores in the early stage of training when the exploration (ϵ value) was high, the agent may never learn a good strategy even with extremely long training times. Despite the experiment results from Figure 8-11 being quite reproducible, further validation experiments have shown a few cases of failures. It is, however, possible to calculate the gradient for average training score every few epochs and re-start the experiment automatically.

4.5 Interesting Testing Footages

Some of the footages I have captured during the testing of my agents are quite interesting and I have attached the mp4 files for some of them in the appendix. The videos have been captured with a 256x144 resolution at 35 frames per second despite the agent using a 128x72 resolution at 8.75 frames per second as its input.

As shown in **corridor_5_instant_death**, for skill level 5 of the original Deadly Corridor, a lot of the times the agent would just die instantly when the episode begins. I have removed the “idle state” with no button input from the action space, so the agent was trying to react, just that it was stun-locked by the enemies and died within 2 seconds. At level 5, Deadly Corridor without modification is a heavily luck-based scenario.

Sometimes the agent would learn strategies that exploit certain game mechanics, in **deathmatch_farm_in_corner_70** (Deathmatch with the original textures), the agent stayed in a relatively safe corner and waited for enemies to kill each other by friendly fire while it was sheltered by a wall. Had it not been that one Chaingun Guy spawned at that exact position, the agent could have farmed more than 70 rewards without doing anything.

Both **deathmatch_typical_690** and **deathmatch_1250** (Deathmatch with modified textures) saw the agent employing an interesting strategy, circle-strafing around the arena while firing constantly, just like a human player (although a human player would probably conserve the ammo more). This has also illustrated a limitation in my training procedure for the agent as it seemed to be always firing its shotgun, which is an indication that combat mode is active at all time. This would be further discussed in the Future Works section. Aside from this behavior, the agent performs quite well in the Deathmatch scenario with modified textures, 690 was both the mode and median for the testing scores.

The reason why the original Deadly Corridor at level 1 to 4 is not a big challenge can be observed in **corridor_4_run_victory** and **corridor_4_run_failed**, the agents tend to learn the strategy of running directly towards the exit point. According to my observations, the agent can only run to the exit point alive when the last Chaingun Guy die of friendly fire (its death animation can be briefly noticed in the **corridor_4_run_victory** video), but even if it doesn't make it to the exit point, the scores would still be high as it had moved very close of the exit.

4.6 Brief Investigation into the Features Extracted

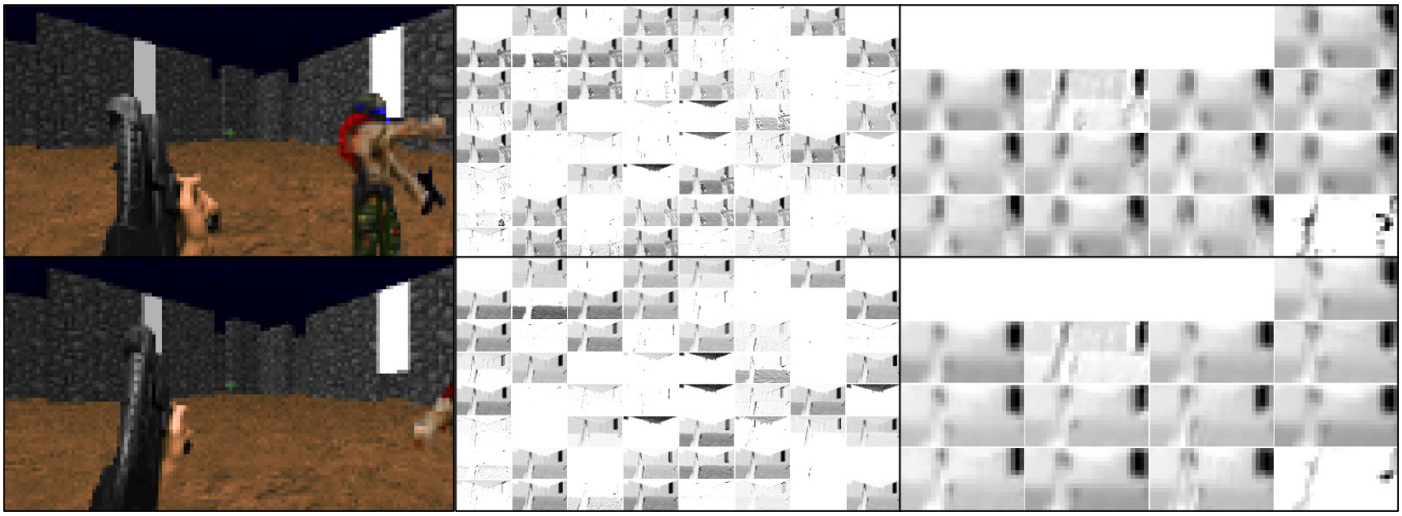

Figure 13: 2 frames (left) with feature maps from the first convolutional layer (middle) and third convolutional layer (right)

This specific agent (also recorded in **deathmatch_typical_690** and **deathmatch_1250**) has been trained in Deathmatch for 385 minutes through 20 epochs, equivalent to playing the scenario for 3 to 4 hours in real-time at 8.75 frames per second. Feature maps have been extracted from 2 successive combat frames the agent observed during a test run (Figure 13). Several interesting phenomenons can be observed here:

1. Similar "ceiling features" can be noticed in multiple feature maps from the output of the first convolutional layer, yet the third layer's feature maps don't contain anything similar. Likely due to the ceiling being unimportant to the combat model.
2. The white strips that were explicitly added as reference objects when modifying the scenario have been extracted as features, helping the agent to execute its circle-strafing strategy.
3. The feature map at the bottom-right corner of the third layer's output seem to be related to enemy detection, even in the second frame where only an arm is shown, the activation values for it has been quite high. Although the feature map is also picking up the weapon wielded by the agent itself.
4. Many feature maps are similar and there is only one that detects enemy presence, indicating that the convolutional layers are not well-trained to identify and locate enemies.

4.7 Limitation in Current Training Procedure

Several evidences indicate that the combat model is not well-trained to detect and locate enemies, this is due to my training procedure only updating the navigation model when no enemy is present, therefore all of the training data the combat model receives for enemy detection would have at least one enemy within the frame, leading to the model underperforming in the enemy detection task. The enemy detection task was given the combat model so that the convolutional layers would be more likely to pick up enemy features, but my current agent does not receive this benefit. The combat model still detects enemy features as shown in the activation maps, but the accuracy is very limited.

5 Conclusion

In this project, an AI agent based on Deep Q-Learning was created, utilizing a DQN model for navigation and a DRQN model for combat. The agent was trained with custom training procedure and tested in three scenarios of the 1993 video game Doom, with scenario-specific shaping rewards defined, yielding results of varying degree of success.

A modification to the textures of the Deathmatch scenario provided by ViZDoom was proposed and proved to be beneficial for training AI agents that use low resolution input like 128x72 RGB images. Several modifications to the Deadly Corridor scenario were made to closer resemble the realistic experience of playing Doom.

Investigation in learning rate's effect on the agent's learning ability was conducted to reveal the patterns in the average training score graph for learning rates that are too low, too high or relatively suitable. Limitations of the training procedure created for this project were revealed in test runs and feature maps extracted by convolutional layers of the combat model.

The tools developed for the project and the methodology used can be applied to other projects that utilize the ViZDoom platform or a further in-depth extension of the same project.

6 Future Works

A wider range of optimizers, loss functions downsampling algorithms can be investigated to show their effects on the agent's ability to learn. Different values for dropout probability, ϵ-decay, ϵ-min, frame-skip, learning rate and state length can be tested to provide a better understanding of each of these's effect and underlying meaning to the agent when solving specific problems. Searches for better definitions of the "priority" in prioritized experience replay can be made and potentially a better formula for calculating Q-target can be derived through trials and errors to have better emphasis on long-term returns.

The model architectures can also be improved, a Transformer can be added to the DRQN model to improve its long-term memory, vital to playing FPS games like Doom. Convolutional layers that take larger input sizes can be used with more computational resources available, as shown in the attached video files it's difficult to understand the environment when the rendering resolution is low (the video frames contain 3x more pixels than the input received by the agent). Double Q-Learning and dueling architecture can be used to fix over-estimation problems and improve performance of the agent.

Another aspect of this project that can be greatly improved is the training procedure, as mentioned in 4.7, the DRQN model for combat should also receive training data without enemies in order to detect enemies at better accuracy and as mentioned at the end of the Methodology section, a better training procedure that allows for a DRQN to be used as the navigation model would be desirable as navigating through a 3D video game world should still be modelled by a POMDP if possible.

7 Reflections

I have learnt quite a lot during the development of this project and have been able to put some of my theoretical knowledge into practical use. Reinforcement learning has always been the field in AI that I admired the most but never had the courage to step into, I craved to build game-playing AI agents ever since I read the news of someone using a Convolutional Neural Network to play Super Mario Bros. However, despite dreaming about it for years and working on a few Computer Vision-related projects, I never gathered the time and courage to attempt such a project due to my lack in RL knowledge. This project allowed me to drag myself out of my comfort zone and actively learn new knowledge while experimenting with models designed by myself.

As a fan of retro-games and games with fluid controls in general, Doom has always been my favourite. Therefore, when I picked the project, I thought I would be at an advantage due to my in-depth understanding of the in-game mechanics. I was overly optimistic as playing FPS games with RL agents using only visual information is not an easy task. As the project progressed (or being lack of progress to be exact), my anxiety stacked as I watch all of my ideas not changing anything about the AI's performance. My initial implementation for all of the components were awful and it slowed me significantly when testing new ideas, but I was too afraid to redo everything with more than half of the semester passed. At one point I even planned to just import models from SB3 and call it a day, running away from the nightmares which are Tuesday supervisor meetings and the non-working models I created.

However, knowing I still love AI and plan to stay in the research field, I decided that I would continue to write my own programs and learn the inner-working of each component and technique used in my models. I spent most of Easter holiday learning RL-related topics while rewriting everything I've done in the semester, it ended up being a very valuable experience and taught me that I can't let sunk cost fallacy stop my plans. I was able to implement the architecture mentioned in [23] and came up with my own network structures, hyperparameters and training procedure. This eventually led to me finishing the project at an acceptable state and conduct interesting experiments while evaluating my results.

Overall, the project has broadened my horizons and introduced me to the world of RL, I also learnt a lot about level-editing in SLADE and ACS scripting language. I'm sure that all of these would be valuable skills for my future. I now very much look forward to actually proposing inspiring architectures and concepts instead of just learning.

Reference

1. 246011111 2020. [Online] Available at: https://www.reddit.com/r/speedrun/comments/f02cmj/linkus7_gets_manual_superswim_unbuffered_the_wind/ [Accessed: 14 May 2023].
2. Abadi, M. et al. 2016. TensorFlow: A system for large-scale machine learning [Online] Available at: https://arxiv.org/abs/1605.08695.
3. Amari, S.-I. 1972. Learning patterns and pattern sequences by self-organizing nets of threshold elements. *IEEE Transactions on Computers* C–21(11), pp. 1197–1206.
4. BepInEx 2018. BepInEx/bepinex: Unity / XNA game patcher and plugin framework [Online] Available at: https://github.com/BepInEx/BepInEx [Accessed: 19 May 2023].
5. Blitzstein, J. Markov chains handout for STAT 110 - Harvard University [Online] Available at: https://projects.iq.harvard.edu/files/stat110/files/markov_chains_handout.pdf [Accessed: 14 May 2023].
6. Daniel Gimmer, R.A. 2022. Beyond doom [Online] Available at: https://zdoom.org/ [Accessed: 19 May 2023].
7. decino 2023. Doom's nightmare difficulty: Everything you need to know [Online] Available at: https://www.youtube.com/watch?v=38ZyMEPUAb4 [Accessed: 19 May 2023].
8. DoomWiki.org 2022a. ACS [Online] Available at: https://doomwiki.org/wiki/ACS [Accessed: 19 May 2023].
9. DoomWiki.org 2022b. UDMF [Online] Available at: https://doomwiki.org/wiki/UDMF [Accessed: 19 May 2023].
10. EphiTV 2022. I hacked Iron Lung to reveal everything the game doesn't show you [Online] Available at: https://www.youtube.com/watch?v=iUr3ZB3kLbI [Accessed: 19 May 2023].
11. Fukushima, K. 1980. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics* 36(4), pp. 193–202.
12. Gemrot, J. et al. 2009. Pogamut 3 can assist developers in building AI (not only) for their videogame agents. *Agents for Games and Simulations* 5920, pp. 1–15.
13. Haarnoja, T. et al. 2018. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor [Online] Available at: https://arxiv.org/abs/1801.01290.
14. Hausknecht, M. and Stone, P. 2017. Deep recurrent Q-learning for partially observable mdps [Online] Available at: https://arxiv.org/abs/1507.06527.

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

16. Huber, P.J. 1964. Robust estimation of a location parameter. *The Annals of Mathematical Statistics* 35(1), pp. 73–101.

17. Ivakhnenko, A.G. and Lapa, V.G. 1965. *Cybernetic predicting devices*. CCM Information Corporation.

18. James, A. 2022. VizDoom AI via Deep Reinforcement Learning [Online] Available at: https://pats.cs.cf.ac.uk/archive?Y=2022.

19. Judd, S. 2004. Slade: It's a doom editor [Online] Available at: https://github.com/sirjuddington/SLADE [Accessed: 19 May 2023].

20. Kempka, M. et al. 2016. Vizdoom: A doom-based AI research platform for Visual Reinforcement Learning. *2016 IEEE Conference on Computational Intelligence and Games (CIG)*.

21. Kingma, D.P. and Ba, J. 2017. Adam: A method for stochastic optimization [Online] Available at: https://arxiv.org/abs/1412.6980.

22. KlydeStorm Superswim [Online] Available at: https://www.zeldaspeedruns.com/tww/Techniques/Superswim [Accessed: 14 May 2023].

23. Lample, G. and Chaplot, D.S. 2017. Playing FPS games with deep reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence* 31(1).

24. Liu, Q. et al. 2022. When is partially observable reinforcement learning not scary? [Online] Available at: https://arxiv.org/abs/2204.08967.

25. Metrics, O.M.A. 1992. Reinforcement learning for robots using Neural Networks [Online] Available at: https://dl.acm.org/doi/abs/10.5555/168871.

26. Mnih, V. et al. 2013. Playing Atari with deep reinforcement learning [Online] Available at: https://arxiv.org/abs/1312.5602.

27. Mnih, V. et al. 2016. Asynchronous methods for deep reinforcement learning [Online] Available at: https://arxiv.org/abs/1602.01783v2.

28. Naylor, B. et al. 1990. Merging BSP trees yields polyhedral set operations. Proceedings of the 17th annual conference on Computer graphics and interactive techniques.

29. Ng, A. et al. 1999. Policy invariance under reward transformations: Theory and application to reward shaping. *ICML '99: Proceedings of the Sixteenth International Conference on Machine Learning*, pp. 278–287.

30. Paszke, A. et al. 2019. Pytorch: An imperative style, high-performance deep learning library [Online] Available at: https://arxiv.org/abs/1912.01703.

31. Polceanu, M. 2013. MirrorBot: Using human-inspired mirroring behavior to pass a turing test. *2013 IEEE Conference on Computational Inteligence in Games (CIG)*.

32. Schaul, T. et al. 2016. Prioritized experience replay [Online] Available at: https://arxiv.org/abs/1511.05952.

33. Schulman, J. et al. 2017. Proximal policy optimization algorithms [Online] Available at: https://arxiv.org/abs/1707.06347.

34. Srivastava, N. et al. 2014. Dropout: A simple way to prevent neural networks from overfitting: The Journal of Machine Learning Research: Vol 15, no 1 [Online] Available at: https://dl.acm.org/doi/abs/10.5555/2627435.2670313.

35. Sutton, R.S. and Barto, A.G. 1998. Reinforcement learning: An introduction. Cambridge, MA: MIT Press.

36. Thrun, S. and Schwartz, A. 1999. [Online] Available at: https://www.ri.cmu.edu/pub_files/pub1/thrun_sebastian_1993_1/thrun_sebastian_1993_1.pdf [Accessed: 19 May 2023].

37. Unity Technologies 2023. [Online] Available at: https://unity.com/ [Accessed: 19 May 2023].

38. Van Hasselt, H. 2010. Double Q-learning. Advances in Neural Information Processing Systems 23 (NIPS 2010) 23.

39. Van Hasselt, H. et al. 2015. Deep reinforcement learning with double Q-learning [Online] Available at: https://arxiv.org/abs/1509.06461.

40. Vaswani, A. et al. 2017. Attention is all you need [Online] Available at: https://arxiv.org/abs/1706.03762.

41. Vincze, D. and Kovacs, S. 2009. Fuzzy rule interpolation-based Q-Learning. 2009 5th International Symposium on Applied Computational Intelligence and Informatics.

42. Wang, Z., Schaul, T., Hessel, M. and Hado Hasselt, Marc Lanctot, Nando Freitas 2016. Dueling Network Architectures for Deep Reinforcement Learning. *Proceedings of Machine Learning Research* 48, pp. 1995–2003.

43. Watkins, C. 1989. Learning from delayed rewards [Online] Available at: https://www.cs.rhul.ac.uk/~chrisw/thesis.html [Accessed: 19 May 2023].

44. Yamaguchi, K. et al. 1990. A neural network for speaker-independent isolated word recognition. *First International Conference on Spoken Language Processing (ICSLP 1990)*.

45. Åström, K.J. 1965. Optimal control of Markov processes with incomplete state information. *Journal of Mathematical Analysis and Applications* 10(1), pp. 174–205.

Appendix

P1: simplified pseudocode for training procedure

```
for step <- 1 to steps_per_epoch
    // fetching state information
    state           <- game.get_state()
    frame           <- downsample(state.frame_buffer)

    // calculating shaping reward
    health_lost     <- health - state.health
    new_kills       <- state.kill_count - kill_count
    shaping_reward  <- new_kills * kill_factor
    shaping_reward  <- shaping_reward - health_lost * health_factor

    // updating recorded health and kill counter
    health          <- state.health
    kill_count      <- state.kill_count

    // checking if an enemy is present in current frame
    is_combat       <- False

    for i <- 1 to state.labels.length
        if state.labels[i] is an enemy then
            is_combat <- True
            break
        endif
    endfor

    // having the suitable model as the decison-maker
    if is_combat == True then
        action_index    <- argmax(combat_model(frame))
        action          <- combat_action_space[action_index]
    else
        action_index    <- argmax(navigation_model(frame))
        action          <- navigation_action_space[action_index]
    endif

    game.make_action(action)
    is_terminated = game.is_episode_terminated()

    // combining two types of rewards and save to replay memory
    normal_reward   <- game.get_reward()
    reward          <- normal_reward + shaping_reward

    replay_memory.add(frame, action_index, reward, is_combat, is_terminated)

    // training the two models
    train(combat_model, navigation_model, replay_memory)

    if is_terminated == True then
        game.create_new_episode()
    endif
endfor
```

Videos are in the attached appendix-mp4.zip file.