Deep Reinforcement Learning for Tank-Based Warfare

Irish Senthilkumar (16342613)

School of Computer Science

National University of Ireland, Galway

Supervisors

Dr. Frank Glavin

In partial fulfillment of the requirements for the degree of

MSc in Computer Science (Artificial Intelligence)

August 23, 2021

DECLARATION I, Irish Senthilkumar, do hereby declare that this thesis entitled Deep Reinforcement Learning for Tank-Based Warfare is a bonafide record of research work done by me for the award of MSc in Computer Science (Artificial Intelligence) from National University of Ireland, Galway. It has not been previously submitted, in part or whole, to any university or institution for any degree, diploma, or other qualification.

Signature: 

Abstract

Ever since the creation of reinforcement learning, the development of intelligent agents which can play video games at a human level has been a growing area of research. Current video games are much more challenging than simple Atari games of the past, as they contain high dimensional continuous environment and action spaces, which pose insurmountable challenges to traditional reinforcement learning approaches. This project creates a custom implementation of World of Tanks using Unity and Blender, and attempts to create a game-playing AI which uses a distributed asynchronous back-end training architecture to train itself using self-play.

Contents

1 Introduction **1**

2 Background **2**

2.1 Introduction 2

2.2 Tank Warfare 3

2.2.1 Firepower 3

2.2.2 Armour 6

2.3 World of Tanks 8

3 Related Work **10**

3.1 Introduction 10

3.2 Deep Q-Learning 10

3.3 Prioritised Experience Replay 12

3.4 DQN in Continuous Action Spaces 14

3.5 DQN for FPS Video Games 16

3.6 Deep Q Learning Using Demonstrations 18

3.7 Asynchronous Deep Reinforcement Learning 20

4 Experimental Settings **22**

4.1 Introduction 22

4.2 Prerequisite Software 22

4.2.1 Game Development Environments 22
4.2.2 3D Models 24
4.3 Game Mechanics 25
4.3.1 Tanks 25
4.4 Battle Environments 30

5 Methodology **35**
5.1 Training Architecture 35
5.1.1 Introduction 35
5.1.2 DQN Training Architecture 36
5.1.3 A3C Training Architecture 39
5.2 Evaluation of Training Architectures 42
5.2.1 Preliminary Evaluation of Server Functionality 42
5.2.2 Preliminary Evaluation of Client-Server Architecture . . . 43
5.3 Proposed Methodology 45
5.3.1 Introduction 45
5.3.2 Nature of the Environment 46
5.3.2.1 Observations 46
5.3.2.2 Actions 46
5.3.2.3 Rewards 47
5.3.3 Proposed Training Process 48

6 Results **52**

7 Conclusion **56**

References **60**

List of Figures

2.1 Left: Primitive war elephants. Right: A modern PL-01 light tank. (Adapted from Dennis (2012) and OBRUM) 2
2.2 The turret and gun traverse limits affect the combat effectiveness of a tank. (Adapted from Database) 3
2.3 A suitable caption 5
2.4 Thick frontal armour is placed at the front, with weaker armour at the sides and rear. (Adapted from TanksGG) 6
2.5 The angled frontal armour of the IS-7 is outlined in red. (Adapted from Tank Archives) 6
2.6 Different armour plates have different relative thicknesses when attacking from different angles. (Adapted from TanksGG) 7
2.7 Hull down tactics make use of undulations in the terrain for protection. (Megapixie) 9
2.8 sidescraping correctly behind cover exposes very little weak spots to the enemy. (Adapted from TanksGG) 9

3.1 The DQN Algorithm. (Mnih et al., 2015) 11
3.2 The prioritised experience replay algorithm. (Schaul et al., 2016) . 13
3.3 Results of prioritised experience replay. (Schaul et al., 2016) . . . 14

3.4 The results of DDPG across a range of Atari games. (Lillicrap et al., 2019) 16
3.5 DRQN trains much better when game-specific features are provided, but training times are still very long. (Lample and Chaplot, 2018) 18
3.6 DQfD has good initial performance, and uses the demonstration data to learn even better policies. (Hester et al., 2018) 20
3.7 The asynchronous n-step Q-learning algorithm. (Mnih et al., 2018) 21

4.1 The game engines that were considered in this project. 23
4.2 The 3D modelling software considered in this project. 24
4.3 Left: Historical Image of the Maus. Top Right: Image of the custom Maus in Unity. Bottom Right: Image of the custom Maus in battle. 26
4.4 The armour profile of the Maus. (Adapted from TanksGG) 26
4.5 Left: Historical Image of the IS-7. Top Right: Image of the custom IS-7 in Unity. Bottom Right: Image of the custom IS-7 in battle. 27
4.6 The armour profile of the IS-7. (Adapted from TanksGG) 27
4.7 Left: Historical Image of the EBR. Top Right: Image of the custom EBR in Unity. Bottom Right: Image of the custom EBR in battle. 28
4.8 The armour profile of the IS-7. (Adapted from TanksGG) 28
4.9 Left: Historical Image of the Centurion. Top Right: Image of the custom Centurion in Unity. Bottom Right: Image of the custom Centurion in battle. 29
4.10 The armour profile of the Centurion. (Adapted from TanksGG) . 29
4.11 The custom IS-7 has over 20 distinct parts. 30
4.12 An overhead view of custom Prokhorovka in Unity. 31

4.13 Scenic views from custom Prokhorovka, highlighting the level of detail. 31
4.14 A heatmap of tank movements in Prokhorovka. (Adapted from WoT Inspector) 32
4.15 An overhead view of the Barcelona map. 33
4.16 Scenic views from Barcelona, highlighting the level of detail. . . . 34

5.1 The DQN training architecture 38
5.2 A suitable caption 41
5.3 The Cart Pole game. 42
5.4 The Mountain Cart game. 43
5.5 The Block Finder game. The yellow agent must learn to detect the red block. 44
5.6 The proposed training architecture should more time have been available for this project. 51

6.1 Results for Cart Pole with A3C (left) and DQN (right). 52
6.2 Results for A3C with various buffer sizes. 53
6.3 Results for Mountain Car with DQN. 54
6.4 Results for Block Finder with A3C. 54

List of Tables

5.1 The actions in the World of Tanks emulator. 47

Chapter 1

Introduction

With the rise of increasingly advanced computer technology, computers have become more powerful than every before. This rise in computing performance has allowed high fidelity video games to gain popularity. Artificial intelligence has also gained in feasibility as computing power increased.

Reinforcement learning is an exciting field in artificial intelligence, as it closely relates to how humans learn in the real world. Recent advances in reinforcement learning have achieved superhuman performance in Atari games. However, Atari games are simple games in 2D environments. This project takes inspiration from recent research advances in deep reinforcement learning and recurrent neural networks to attempt the creation an advanced agent that plays a significantly more complex game - World of Tanks.

An emulator of World of Tanks was completely custom built in Unity and Blender to create an authentic game-playing experience. An intuitive asynchronous distributed back-end training architecture was created for massively accelerated training times.

Chapter 2

Background

2.1 Introduction

Armoured fighting vehicles have existed for centuries as a key component of an army, and have taken many forms, from primitive war elephants to highly advanced main battle tanks. Even though armoured fighting vehicles have dramatically changed in form over the years, they have still served the same purpose - to support infantry using strong offensive and defensive capabilities, as shown in Figure 2.1.

Figure 2.1: Left: Primitive war elephants. Right: A modern PL-01 light tank. (Adapted from Dennis (2012) and OBRUM)

Modern main battle tanks can trace their immediate roots back to the British Mark 1 tank. This tank proved crucial in breaking the stalemate of trench warfare in WWI as its armour allowed it to withstand incoming fire while crossing trenches, and its strong firepower allowed it to assault enemy fortifications. This tank was so successful that other armies on both sides of the war quickly started to produce their own versions. Over the last century, the ideas put forward by the Mark 1 tank have evolved into the main battle tanks of today, which boast both incredible offensive capabilities such as laser guided missiles and defensive capabilities such as reactive armour. These main battle tanks have become a key component of modern armies around the world.

2.2 Tank Warfare

2.2.1 Firepower

The offensive capability of tanks is mainly governed by its firepower. The firepower of a tank comes from its main gun, which is usually housed on a turret that can rotate 360 degrees. This main gun also has fixed elevation and depression angles, as shown in Figure 2.2. Gun depression is an important characteristic, as more gun depression allows the tank to make use of undulations in the terrain for advanced tactics such as hull-down positioning.

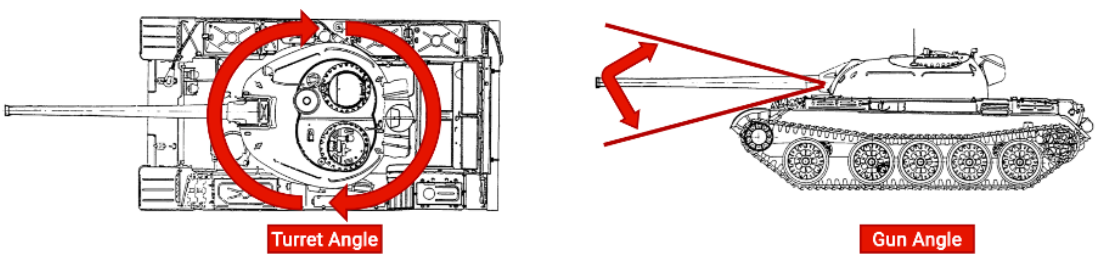

Figure 2.2: The turret and gun traverse limits affect the combat effectiveness of a tank. (Adapted from Database)

There are three main characteristics that define the behaviour of the main gun. Calibre, measured in millimetres, is the size of the shell and it dictates the amount of damage which can be done by the shell.

Muzzle velocity, measured in metres per second, is the speed of the shell. Faster muzzle velocities make it easier to hit moving targets, since there is less 'shell travel time' between the shell leaving the barrel and impacting the enemy tank.

Penetration, measured in millimetres, is the amount of armour that can be penetrated by the shell. Tanks usually have very thick armour at the front, and therefore shells with high penetration can be used to damage enemy tanks directly from the front.

A shell which maximises all three of these characteristics will be very useful in battle, but unfortunately this is not possible. High-calibre shells have greater mass in order to do greater damage, but more mass requires more propellants to fire the shell at the same muzzle velocity. Muzzle velocity can be improved by reducing the mass of the shell, but this will reduce the shell's damage. Shell penetration is improved by increasing the muzzle velocity and the shell calibre, since faster and heavier shells carry more kinetic energy, which allows enemy armour to be penetrated easier.

Armour-piercing (AP) are heavy shells forged from chromium steel that contain no explosive. They have medium velocity, damage, and penetration. These shells lose a medium amount of penetration over long-range combat.

Armour-piercing composite rigid (APCR) shells contain a very dense core of tungsten carbide, surrounded by an exterior of aluminium. These shells have high muzzle velocity and penetration, as the exterior allows the shell to be light, while the dense core prevents the shell from squishing and losing kinetic energy when impacting enemy armour. Since APCR shells contain no explosive, and

since the penetrating core is relatively small, these shells have low damage. Since these shells rely on their muzzle velocity to maximise kinetic energy, they perform poorly in long-range combat as the shell loses velocity over distance.

High-explosive anti-tank (HEAT) shells contain a shaped charge that is very effective against flat enemy armour. Due to this shaped charge, these shells lose no penetration over distance. These shells have medium damage, high penetration, and low muzzle velocity.

High-explosive (HE) shells explode on impact, dealing high damage to infantry and weakly-armoured targets. However, these shells have low penetration and muzzle velocity, but lose no penetration over distance. These shells can be used against the weakly-armoured sides and rear of enemy tanks at close range to deal devastating damage. Figure 2.3 shows the cross-sections of all four types of shells.

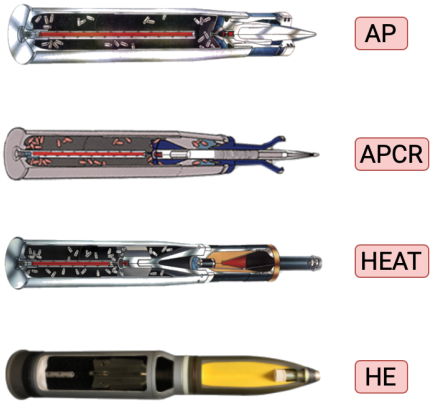

Figure 2.3: A suitable caption

2.2.2 Armour

Just as the main gun is essential for firepower, armour is essential for survivability. WW2 and Cold War era tanks utilised solid steel armour plates for protection. Thicker armour plates offer increased resilience at the cost of mobility, therefore thick plates are typically only placed at the front of the tank, with thinner armour plates on the sides and rear of the tank. Figure 2.4 highlights this strategic placement of armour at varying thicknesses.

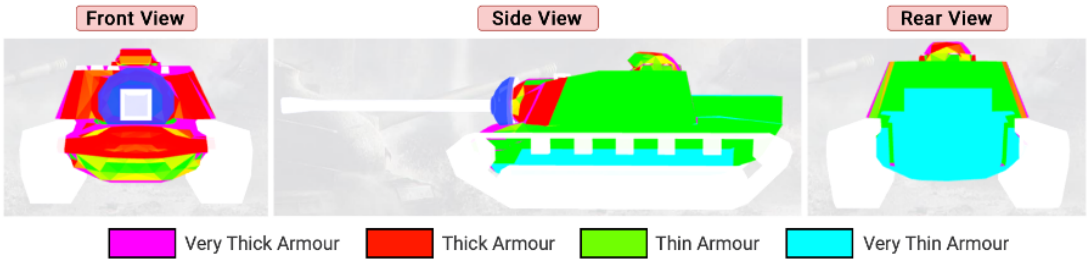

Figure 2.4: Thick frontal armour is placed at the front, with weaker armour at the sides and rear. (Adapted from TanksGG)

The relative thickness of armour depends on both its raw thickness and the impact angle of the shell which hits it. Angled armour is significantly more effective than flat armour, but it reduces the usable space inside the tank. Therefore, angled armour is typically placed on the front of the tank and on the turret, as shown in Figure 2.5.

Figure 2.5: The angled frontal armour of the IS-7 is outlined in red. (Adapted from Tank Archives)

Frontal angled armour is only effective when enemy shells impact the tank directly from the front, and this limitation can be exploited by enemy tanks as attacking the frontal armour at a perpendicular angle can substantially reduce relative armour thickness. Figure 2.6 illustrates the increase and decrease in relative armour thickness from two different shell impact angles. When the shell impacts the tank frontally, the upper glacis plate (highlighted in blue) is angled away from the shell and therefore has thick armour, while the lower glacis plate (highlighted in black) is angled towards the shell and therefore has thin armour. When the enemy exploits the weakness of angled armour by repositioning to fire from a diagonal angle, the upper glacis plate loses its relative angle and becomes thin, while the relative angle of the lower glacis plate increases significantly.

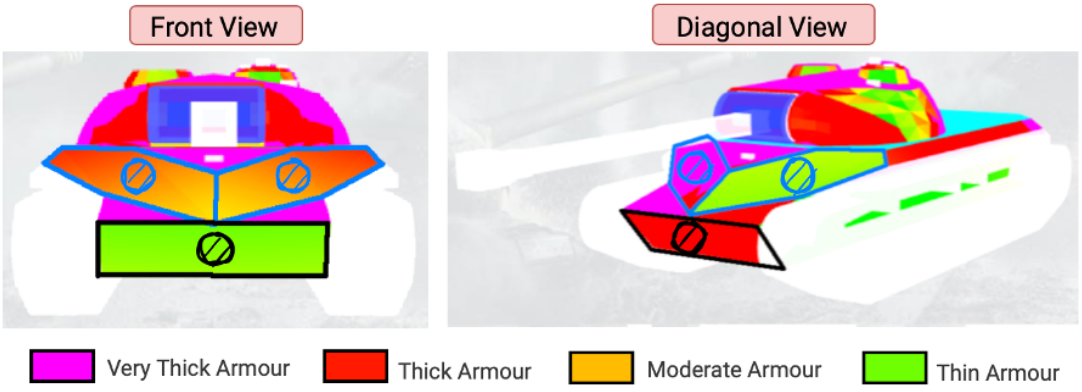

Figure 2.6: Different armour plates have different relative thicknesses when attacking from different angles. (Adapted from TanksGG)

2.3 World of Tanks

World of Tanks is popular massively online multiplayer game where players partake in armoured battles using WWI, WWII, and Cold War era tanks across a variety of different battlegrounds. There are hundreds of historical tanks to choose from, and each have their own set of unique characteristics, advantages, and disadvantages for different situations (Wargaming).

The tank battles take place in a 15 vs 15 format. Score is divided into three segments - damage caused, assisted damage caused, and damage blocked by armour. For the sake of simplicity, only damage caused and damage blocked will be considered in this project.

Although World of Tanks was immensely popular upon launch, the player population has steadily decreased from 2016 onwards, particularly in specific regions such as Asia and Australia. This decrease in player population has a negative impact on the morale of existing players, since queue times (the time taken to find a suitable battle) has increased due to the lack of suitable matches. To combat this issue, World of Tanks introduced bots onto servers with low population. Unfortunately, these bots have noticeable poor performance due to their tendency to make critical errors in gameplay very often. These bots also cannot make use of advanced gameplay tactics such as side-scraping and hull-down manoeuvres. Hull down tactics, as shown in Figure 2.7, reduce the tank's exposed profile while firing. Sidescraping, as shown in Figure 2.8, is the sideways angling of a tank behind solid cover, allowing the tank to return fire while preventing enemy tanks from shooting at its weak spots.

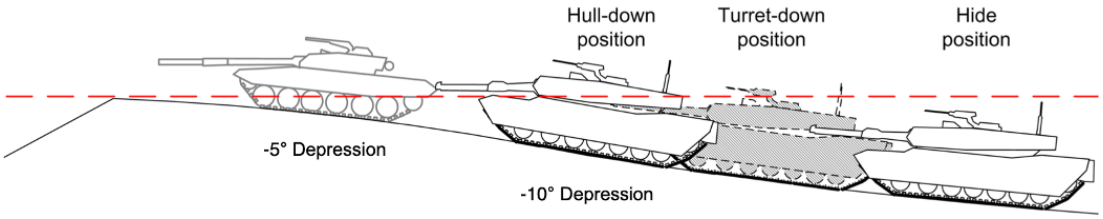

Figure 2.7: Hull down tactics make use of undulations in the terrain for protection. (Megapixie)

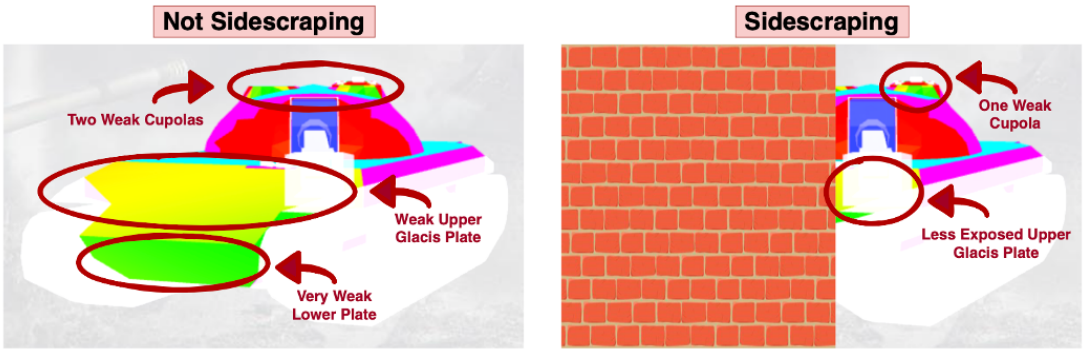

Figure 2.8: sidescraping correctly behind cover exposes very little weak spots to the enemy. (Adapted from TanksGG)

Chapter 3

Related Work

3.1 Introduction

Before the introduction of improved techniques in neural networks, reinforcement learning was not easily applicable to complex domains such as autonomous game-playing. Neural networks have high representational power and can therefore be used by the agent to learn complex policies for complex scenarios. This section explores relevant advances in deep reinforcement learning that have inspired the work done in this project.

3.2 Deep Q-Learning

Mnih et al. (2015) developed an effective approach that utilised a neural network to approximate the value function for reinforcement learning when playing Atari games. One of the fundamental advances put forward in this paper was the use of a replay buffer, which stored state transitions. These state transitions could then be sampled randomly to create an uncorrelated batch for mini-batch gradient descent. Uncorrelated batches bring about training data with low variance, which

can improve the training accuracy and model stability. Using replay buffers also improves data efficiency since a state transition can potentially be used in multiple training iterations.

This paper also highlighted the importance of preprocessing inputs for deep reinforcement learning. Reducing input dimensionality can improve model stability and convergence while also reducing the computational overhead when training. The authors converted the RGB frames from the Atari emulator to gray-scale and down-sampled the frames to only one fifth of their original dimensions. Since the inputs to the neural network were preprocessed images, convolutional layers were used extensively in the training network. The neural network outputs a Q-value for each action in the action space. The action with the highest Q-value is chosen as the next action in the environment. The DQN algorithm is shown in Figure 3.1.

Algorithm 1 Deep Q-learning with Experience Replay

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

Figure 3.1: The DQN Algorithm. (Mnih et al., 2015)

The network architecture used by the authors was proven to be robust, as the same network architecture with identical hyperparameters showed impressive

results across a variety of games without incorporating game-specific information. The cumulative reward per episode was used as the model evaluation metric. DQN performed very well across a range of Atari games, sometimes surpassing the proficiency of expert human players.

3.3 Prioritised Experience Replay

DQN uses uniform experience replay which samples and discards transitions regardless of their significance. In reality, some transitions may be more important than others, and therefore prioritising certain transitions when sampling for training can improve model accuracy and convergence. Following this ideology, Schaul et al. (2016) developed a modified replay buffer sampling approach which yields improved performance over uniform sampling.

Temporal difference error is the difference between bootstrap estimates of the transitions, and can be used to measure the unexpectedness of a transition. The authors record the temporal difference errors of all transitions in the replay buffer, and the transition with the highest temporal difference is always replayed. This simple greedy prioritisation approach yields better results than uniform sampling, but it is sensitive to noise, computationally expensive, and prone to over-fitting. The authors introduced a stochastic sampling approach which interpolates between greedy prioritisation and uniform sampling, using both proportional prioritisation and rank-based prioritisation. Bias correction is also performed by importance sampling weights that compensate for non-uniform probabilities. The full process is shown in Figure 3.2.

Algorithm 1 Double DQN with proportional prioritization

Input: minibatch k, step-size η, replay period K and size N, exponents α and β, budget T.
Initialize replay memory $\mathcal{H} = \emptyset$, $\Delta = 0$, $p_1 = 1$
Observe S_0 and choose $A_0 \sim \pi_\theta(S_0)$
for $t = 1$ **to** T **do**
  Observe S_t, R_t, γ_t
  Store transition $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$ in $\mathcal{H}$ with maximal priority $p_t = \max_{i<t} p_i$
  if $t \equiv 0 \mod K$ **then**
    for $j = 1$ **to** k **do**
      Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
      Compute importance-sampling weight $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$
      Compute TD-error $\delta_j = R_j + \gamma_j Q_{\text{target}}(S_j, \arg\max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$
      Update transition priority $p_j \leftarrow |\delta_j|$
      Accumulate weight-change $\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1})$
    end for
    Update weights $\theta \leftarrow \theta + \eta \cdot \Delta$, reset $\Delta = 0$
    From time to time copy weights into target network $\theta_{\text{target}} \leftarrow \theta$
  end if
  Choose action $A_t \sim \pi_\theta(S_t)$
end for

Figure 3.2: The prioritised experience replay algorithm. (Schaul et al., 2016)

Proportional bias-corrected prioritised experience replay and rank-based bias-corrected prioritised experience replay both yielded a significant increase in episodic reward over uniform sampling for a variety of Atari games. Both prioritisation variants overall displayed similar results to each other, but sometimes outperformed the other depending on the specific Atari game being learned. As shown by Figure 3.3, prioritised experience replay is a quick modification to DQN that is proven to significantly increase performance.

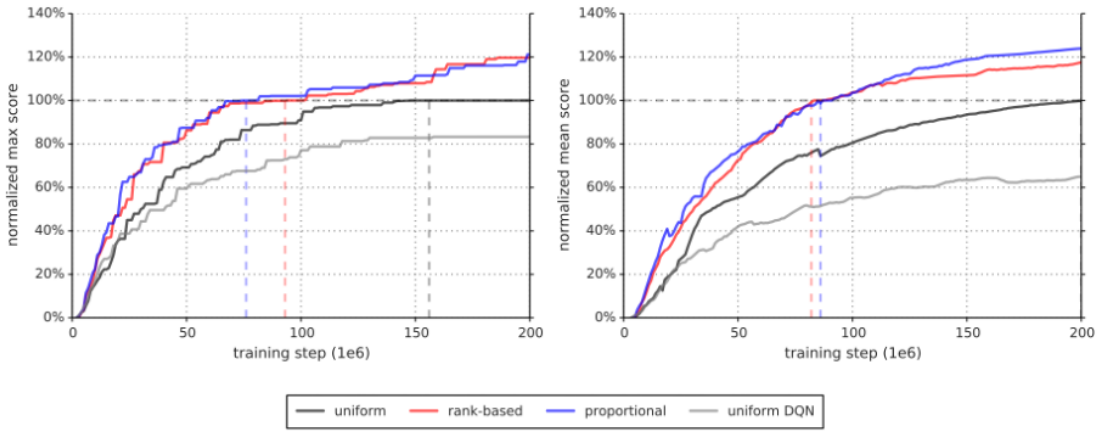

Figure 3.3: Results of prioritised experience replay. (Schaul et al., 2016)

3.4 DQN in Continuous Action Spaces

The traditional deep Q-network proposed by Mnih et al. (2015) can effectively solve reinforcement learning problems in discrete, low-dimensional action spaces such as Atari. However, many real-world applications of reinforcement learning take place in continuous, high-dimensional action spaces such as computer mouse movement. A naive approach to tackle this problem would be discretising the action space, but this approach quickly falls apart due to the curse of dimensionality. Lillicrap et al. (2019) recently developed an off-policy, model-free, actor-critic algorithm, titled Deep Deterministic Policy Gradient (DDPG), which can learn policies in high-dimensional action spaces.

DDPG is a simple extension to DQN which requires a straightforward actor-critic architecture. This actor-critic approach, based on the original DPG algorithm by Silver et al. (2014), utilises a separate actor and critic networks to train the model. The actor network dictates the current policy by mapping states to actions, and the critic network generates the Q-values using the actor network. Similarly to DQN, both networks are updated by sampling mini-batches from a

replay buffer.

One of the big changes implemented by the authors is the introduction of soft target updates. The actor and critic neural networks use the target value to iteratively improve themselves, but the target value depends on the critic's output. This makes the Q-updates vulnerable to divergence. Instead of directly copying the weights of the target network to the Q-network, the authors used 'soft' target updates, where the actor and critic networks (with parameters θ) are duplicated, and these duplicates (with parameters θ') are instead used to calculate the target value. The weights of the target network slowly track the weights of the learned networks: $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$, where $\tau \leq 1$. By weakening the link between the target weights and learned weights, the learning problem moves closer to the case of supervised learning. To further improve learning stability and convergence, batch normalisation (Ioffe and Szegedy, 2015) was used to manually scale features to a similar range across different environments and units. Exploration is a significant challenge in continuous, high dimensional action spaces, and this hurdle was overcome using the Ornstein-Uhlenbeck noise process (Uhlenbeck and Ornstein, 1930) to generate temporally correlated exploration.

In tests across a variety of continuous control problems including Cart Pole, Gripper, and Puck Shooting, DDPG performed optimally when either game-specific data or raw pixels were provided as input. Even though DDPG was prone to over-estimating the Q-values, good policies were still learned. Identical networks and hyperparameters were used across all scenarios, highlighting the robustness of DDPG. The results of DDPG in various Atari games is shown in Figure 3.4.

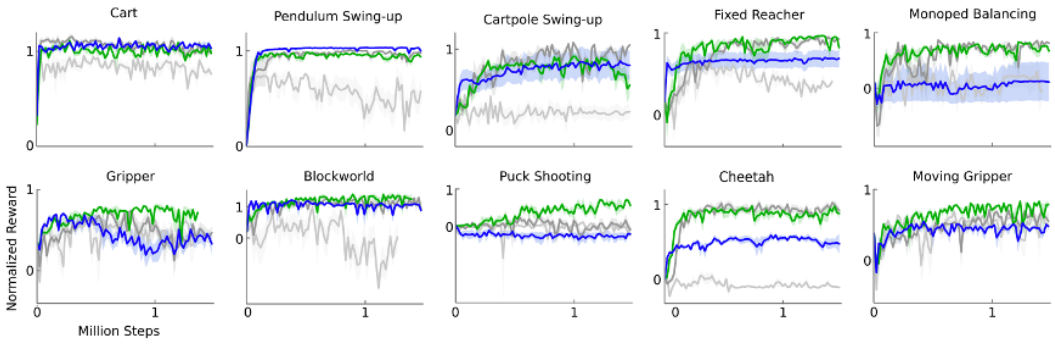

Figure 3.4: The results of DDPG across a range of Atari games. (Lillicrap et al., 2019)

3.5 DQN for FPS Video Games

Autonomous game-playing agents is a new field in artificial intelligence, and recent developments have primarily focused on using DQN in Atari environments. Even though Atari environments pose a number of design challenges for the development of game-playing agents, these primitive video games take place on 2D fully observable environments. Modern video games typically take place in 3D partially observable environments, where the player's field of view is limited. Lample and Chaplot (2018) recently developed the first architecture which tackles 3D partially observable environments in Doom, a first-person shooter video game, where only raw pixels and a small amount of game-specific information were provided as inputs.

The largest challenge with partially observable environments is the need to remember previous states in order to select the optimal actions. Hausknecht and Stone (2015) tackled this issue using Deep Recurrent Q-Networks (DRQN). Here, the model estimates the Q-values as $Q(o_t, h_{t-1}, a_t)$, using the observation of the environment, the hidden state of the agent represented by an output of the network at the previous step, and the action respectively. In order to increase

the model training accuracy, information about the visible entities in the frame was also provided to the model. The training architecture developed by the authors consists of three segments. The first segment contains the convolutional layers for image processing. The output of the first segment is the input of the second and third segment, which analyse the game-specific information and run the recurrent neural network respectively. The authors discovered that sharing the convolutional filters between predicting game features and the Q-learning objective is decisive in the performance of the model.

Unlike Atari environments, first-person shooter environments consist of many challenging tasks, such as navigation through the game environment. In Doom, the authors split the training task into navigation and action phases. Each phase uses its own model, allowing the overall architecture to be modular (different models can be trained and tested at each phase), efficient (models can be trained in parallel), and simple (the relevant actions can be assigned to the appropriate phase, therefore allowing irrelevant actions to be excluded in a phase. For example, the agent does not need to shoot its gun when an enemy is not present in the frame). In the Doom environment, rewards are delayed and are not the result of one specific action, therefore giving rise to a sparse replay buffer. The authors overcame this hurdle by shaping the reward function (Ng, 2003) to include small intermediate rewards which increase learning speed. Similarly to the original Atari DQN approach by Mnih et al. (2015), frame skipping was used to help decorrelate updates and increase training speed. To perform the DRQN updates, a sequence of observations is randomly sampled from the replay buffer, but only the errors from the latter portion of observations are backpropagated through the network, since early observations in the sequence will be estimated from almost non-existent history. Kill-death ratio was used to measure model performance. The model performed very well, and even beat human players across single player

and multiplayer scenarios. The model reached convergence after seventy hours of training, as shown in Figure 3.5.

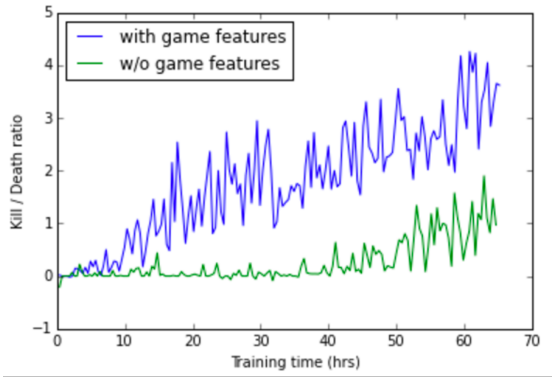

Figure 3.5: DRQN trains much better when game-specific features are provided, but training times are still very long. (Lample and Chaplot, 2018)

3.6 Deep Q Learning Using Demonstrations

Deep Q-Learning requires a vast amount of training data to deliver an acceptable policy. During training, the performance can initially be poor, which hinders the applicability of many reinforcement learning approaches such as DQN in physical, real-world scenarios where good performance is necessary during training. In reality, humans learn using imitation learning where the learner observes an expert execute a task. Hester et al. (2018) devised an approach which combines imitation learning with DQN, titled Deep Q-Learning from Demonstrations (DQfD), where expert demonstration data was capable of significantly accelerating the learning process for a variety of games in the Atari environment.

A number of sessions with expert gameplay in the Atari environment is carried out and the state transitions are recorded. Before learning in the environment, the agent is pre-trained to imitate the expert with a value function. To perform pre-training, the agent samples random mini batches from the demonstration

data and updates the weights of the model using a 1-step Q-learning loss, an n-step Q-learning loss, a supervised large margin classification loss, and finally an L2 regularisation loss. The Q-learning losses make the model satisfy the Bellman equation, and the supervised loss is used to classify the expert's actions. The demonstrations only cover a narrow area of the state and action space, therefore many state-action pairs are unrepresented in demonstration data and therefore do not have data to ground themselves to realistic values. The supervised loss helps to ground the values to unseen actions to reasonable values, helping the greedy policy of the pre-trained value function imitate the demonstrator. The n-step Q-learning loss helps to propagate the rewards of actions to earlier states, which leads to more stable training. The L2 regularisation helps helps to prevent the network from over-fitting, since the demonstration dataset is small.

After pre-training, the model is trained normally as in DQN, but with some changes to ensure that the agent makes use of the demonstration data. Demonstration data is permanently stored in the replay buffer, and prioritised sampling is used to ensure that the sampled mini batch for training contains an appropriate mixture of both learned transitions and expert transitions. DQfD performed very well across a range of Atari games. In the Hero, Road Runner, and Pitfall games, DQfD achieved higher scores than any previously published approach. This is especially impressive in the case of Pitfall, which is considered to be the most difficult Atari game, due to its sparse positive rewards and dense negative rewards. Not only did DQfD display significantly better initial performance when learning, it also used the demonstration data to learn superb policies, as shown in Figure 3.6. DQfD excels in hard exploration games, proving that demonstration data can be used in place of smart exploration strategies.

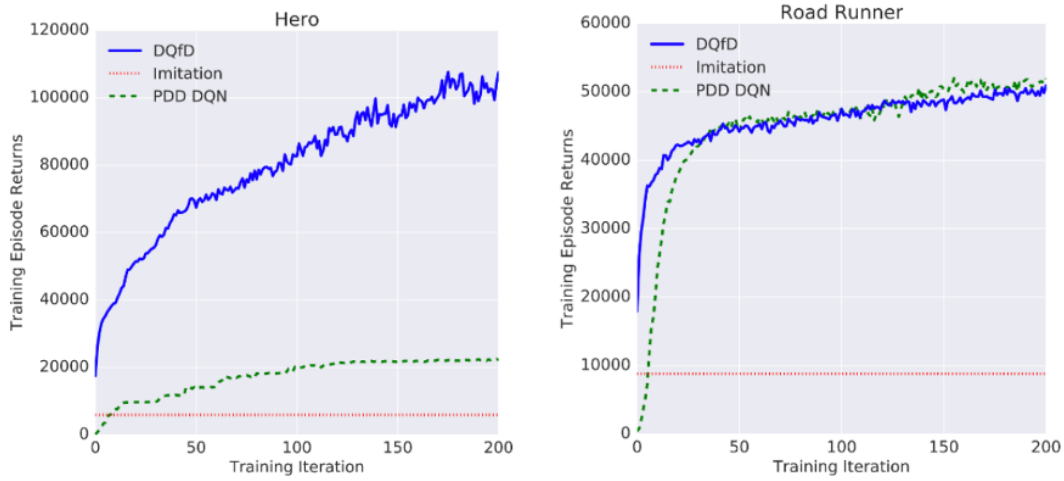

Figure 3.6: DQfD has good initial performance, and uses the demonstration data to learn even better policies. (Hester et al., 2018)

3.7 Asynchronous Deep Reinforcement Learning

Experience replay is a fundamental aspect of deep reinforcement learning, but complex environments require a very large replay buffer, and this can be computationally expensive to maintain. Nair et al. (2015) modified the original deep-Q network architecture to introduce parallel asynchronous training on distributed devices, but unfortunately a distributed system of computers was not available for this project. Mnih et al. (2018) introduced an approach for asynchronous learning that trains significantly faster than normal DQN using only a home computer.

Experience replay is necessary to decorrelate the observations of the environment, but if multiple instances of agent-environment pairs are running in parallel, the incoming observations will be decorrelated anyways. Using this ideology, the authors created two algorithms for asynchronous learning - asynchronous n-step Q-learning, and asynchronous advantage actor-critic (A3C). In both algorithms,

the agent-environment pairs are executed on the CPU, thereby removing the communication overhead imposed by a distributed parallel execution. In order to decorrelate incoming observations even further, different exploration policies are executed on each thread. The full process for asynchronous n-step Q-learning is shown below in Figure 3.7.

Algorithm S2 Asynchronous n-step Q-learning - pseudocode for each actor-learner thread.

// Assume global shared parameter vector θ.
// Assume global shared target parameter vector θ^-.
// Assume global shared counter $T = 0$.
Initialize thread step counter $t \leftarrow 1$
Initialize target network parameters $\theta^- \leftarrow \theta$
Initialize thread-specific parameters $\theta' = \theta$
Initialize network gradients $d\theta \leftarrow 0$
repeat
  Clear gradients $d\theta \leftarrow 0$
  Synchronize thread-specific parameters $\theta' = \theta$
  $t_{start} = t$
  Get state s_t
  repeat
    Take action a_t according to the ϵ-greedy policy based on $Q(s_t, a; \theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ \max_a Q(s_t, a; \theta^-) & \text{for non-terminal } s_t \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \frac{\partial \left(R - Q(s_i, a_i; \theta')\right)^2}{\partial \theta'}$
  end for
  Perform asynchronous update of θ using $d\theta$.
  if $T \mod I_{target} == 0$ **then**
    $\theta^- \leftarrow \theta$
  end if
until $T > T_{max}$

Figure 3.7: The asynchronous n-step Q-learning algorithm. (Mnih et al., 2018)

A3C was very successful, and it beat the original DQN approach by Mnih et al. (2015) in half the training time, using only a fraction of the computational power used by DQN. Even though replay buffers were not used by the authors, they have suggested that combining replay buffers with asynchronous architectures will yield even better results.

Chapter 4

Experimental Settings

4.1 Introduction

Data from World of Tanks cannot be observed by an external program during runtime due to anti-cheat policies, and it also does not support an environment for training and evaluating reinforcement learning agents. Therefore, it was absolutely necessary to create a customised implementation of World of Tanks, using a game engine and 3D modelling software.

4.2 Prerequisite Software

4.2.1 Game Development Environments

One of the daunting challenges that have been overcome in this project is the creation of a game environment that accurately emulates the gameplay mechanics in World of Tanks. Many game engines were considered for use in this project, as shown in Figure 4.1.

Figure 4.1: The game engines that were considered in this project.

Unreal Engine 4 is a highly regarded game engine developed and maintained by Epic Games (UnrealEngine). Unreal Engine is used by many popular triple-A video games including Fortnite, Tekken 7, and Valorant. It contains a complete suite of development tools designed for real time technology. Unreal Engine is written on C++, a cross-platformed language that can be used to create complex, high performance applications. It also has a highly modifiable rendering pipeline and is fully open-source. However, Unreal Engine is more suited to large, complex, long-term projects such as triple-A video games, rather than a relatively simple World of Tanks emulator.

CryEngine is a game engine designed by Crytek (CryEngine). CryEngine has been used for popular video games including the Far Cry Series and the Crysis series. CryEngine uses an advanced rendering pipeline that is well optimised and produces high-fidelity visuals with low performance overhead. However, CryEngine has a small user base and therefore the support and resources are quite limited.

Unity is a popular cross-platform game engine which can be used to create real-time 2D and 3D projects for a variety of applications, including video games, animation, and engineering (Unity). It is used by indie and major game developers alike to create popular video games such as Escape from Tarkov, Fall Guys, and Call of Duty Mobile. Unity is written in C-Sharp, a highly popular language that is simple, modern, and general purpose. It has a simple yet powerful ren-

dering pipeline that can be used to quickly create small-scale and medium-scale projects. Unity has a very large user base, therefore support and resources are readily available. However, Unity is single-threaded and closed-source, making it difficult to create high-performance game environments using this game engine.

After careful consideration, Unity was chosen as the game engine for this project, due to its large user base and ease of use. Had more time been available for this project, Unreal Engine would have been used due to its high performance.

4.2.2 3D Models

3D Modelling software (3DMS) is computer software that aids the user in creating a mathematical representation of a 3D object. Since the game environment was completely custom built, 3D modelling software was used to create game meshes which had a clean and consistent topology which accurately portrayed the tanks from World of Tanks. Advanced 3D modelling is not natively supported on most game engines, including Unity, therefore a separate 3DMS is required. Since many 3DMS packages such as SolidWorks and Houdini are far too expensive to use in a single project, only two 3DMS were considered, as shown in Figure 4.2.

Figure 4.2: The 3D modelling software considered in this project.

The first 3DMS considered was Autodesk Maya (Autodesk). Maya is the industry standard for 3D modelling and animation, and it includes a large variety of features including particle simulation and cloth modelling. However, Maya is not beginner-friendly and it is very difficult to master.

Blender (Blender) is an open-source 3DMS that is very popular with industry specialists and hobbyists alike. Blender is relatively easy to learn and it has a large user base. It also has many features such as sculpting and manipulation of meshes at the polygon-level.

After a thorough analysis, Blender was chosen as the 3DMS for this project, due to its plentiful features, readily-available support, and direct compatibility with Unity.

4.3 Game Mechanics

4.3.1 Tanks

Tanks are the agents in this reinforcement learning project, and a variety of different tanks were developed to simulate the flow of a battle in World of Tanks. The tanks are divided into three distinct classes - heavy tanks, medium tanks, and light tanks.

The Panzer VIII Maus was a famous super-heavy tank built in 1941 by the German army. Although it never saw combat, this heavily-armoured mobile fortress holds the record for the heaviest tank ever produced, weighing in at 188 tonnes (Hemmings and Guest). Although the Maus made little use of angled armour, it was armoured with 20 cm of hardened steel, making it very resilient. It was armed with a 128 mm main gun with 8 degrees of gun depression, allowing the Maus to make good use of hull-down tactics. Due to its heavy weight, the Maus only has a top speed to 20 kph, making relocation difficult. The armour profile of the Maus is shown in Figure 4.4.

Figure 4.3: Left: Historical Image of the Maus. Top Right: Image of the custom Maus in Unity. Bottom Right: Image of the custom Maus in battle.

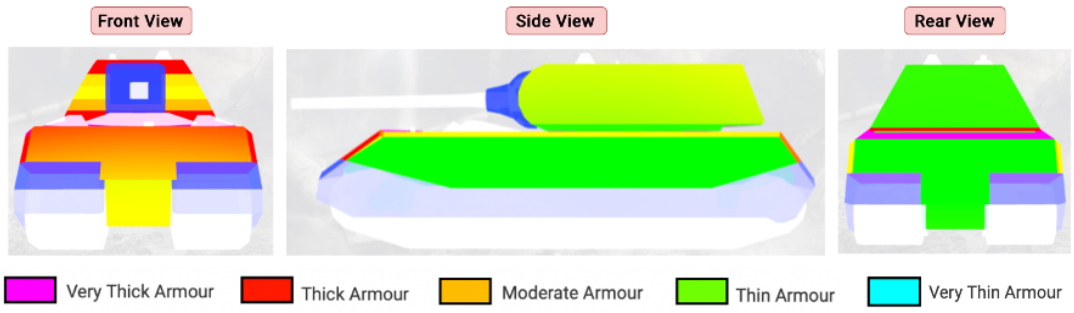

Figure 4.4: The armour profile of the Maus. (Adapted from TanksGG)

The IS-7 was a heavy tank built in 1948 by the Soviet army (Nash). Similarly to other post-WWII Soviet tanks, the IS-7 featured heavy usage of angled armour. Even though the raw armour thickness is only 15 cm, the angling on the armour enables it to reach relative thicknesses up to 30 cm, making the IS-7 very hard to penetrate when the armour is angled correctly. The IS-7 turret is very heavily armoured and rounded in shape, therefore making it impervious to enemy firepower. The IS-7 features a 130 mm main gun. This gun is limited by 5 degrees of gun depression, which inhibits the IS-7 from fully using terrain undulations to its advantage. The IS-7 has a top speed of 35 kph, allowing it to

relocate somewhat efficiently on the battlefield. The armour profile of the IS-7 is shown in Figure 4.6.

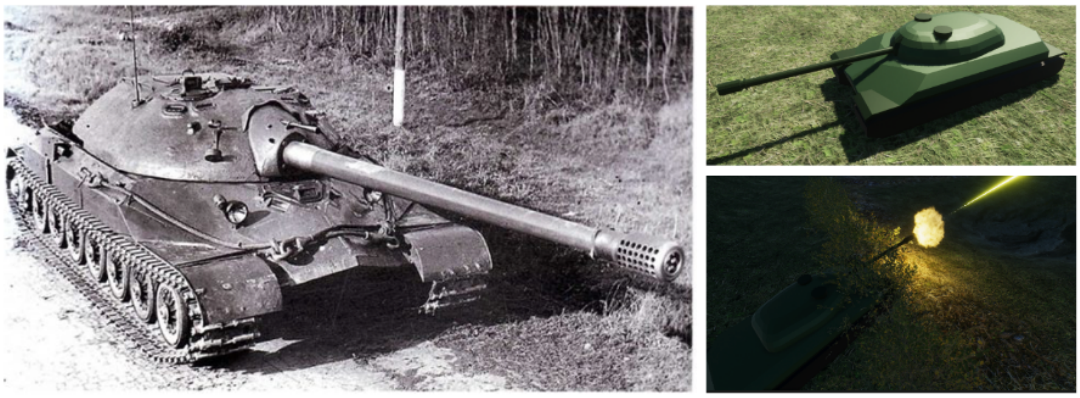

Figure 4.5: Left: Historical Image of the IS-7. Top Right: Image of the custom IS-7 in Unity. Bottom Right: Image of the custom IS-7 in battle.

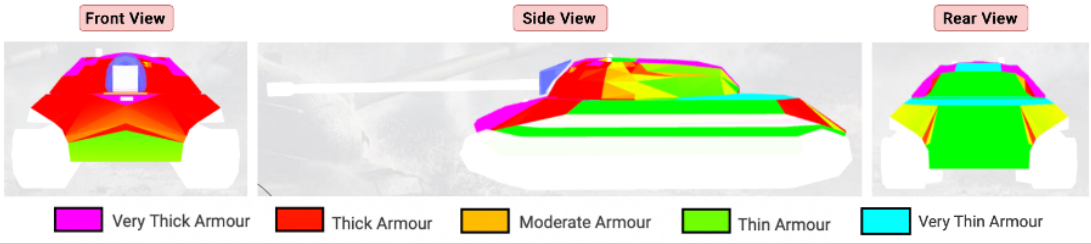

Figure 4.6: The armour profile of the IS-7. (Adapted from TanksGG)

The Panhard EBR was a light tank built in 1950 by the French army (Tanks-Encyclopedia). Unlike the IS-7 and Maus, the EBR saw active combat with armies around the world. The combination of a low weight and a top speed of 72 kph allows the EBR to relocate at a moment's notice. However, the EBR compromises its armour in order to gain speed. The EBR is armed with a 90 mm gun with a quick reload, allow it to deliver consistent firepower onto enemy tanks. This gun boasts 10 degrees of gun depression, allowing the EBR to make full use of the surrounding terrain for 'peek-a-boo' tactics. The armour profile of the EBR is shown in Figure 4.8.

Figure 4.7: Left: Historical Image of the EBR. Top Right: Image of the custom EBR in Unity. Bottom Right: Image of the custom EBR in battle.

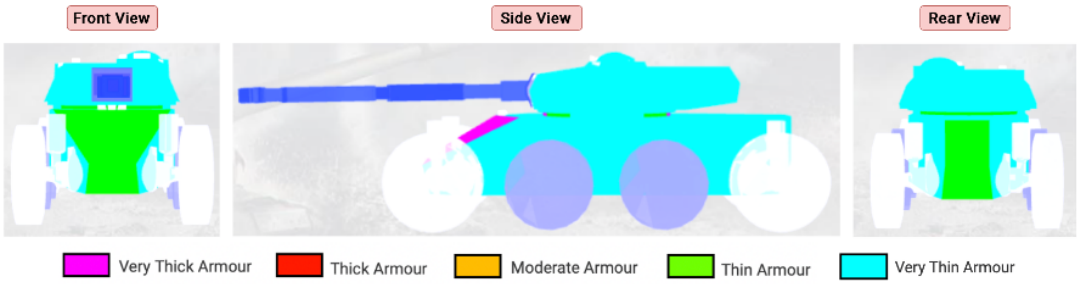

Figure 4.8: The armour profile of the IS-7. (Adapted from TanksGG)

The final tank developed for this project is the Centurion, a medium tank developed in 1945 by the British army (The Tank Museum). The Centurion was used extensively around the world, and even served in 2003 with the British army. The Centurion has thin hull armour, but its frontal turret armour is very strong only when using its gun depression. The Centurion is armed with a 105 mm main gun and 10 degrees of gun depression, allowing it to deal solid damage from all parts of the battlefield. The Centurion also features good mobility with a top speed of 50 kph, allowing it to relocate quickly on the battlefield. The armour profile of the Centurion is shown in Figure 4.10.

Figure 4.9: Left: Historical Image of the Centurion. Top Right: Image of the custom Centurion in Unity. Bottom Right: Image of the custom Centurion in battle.

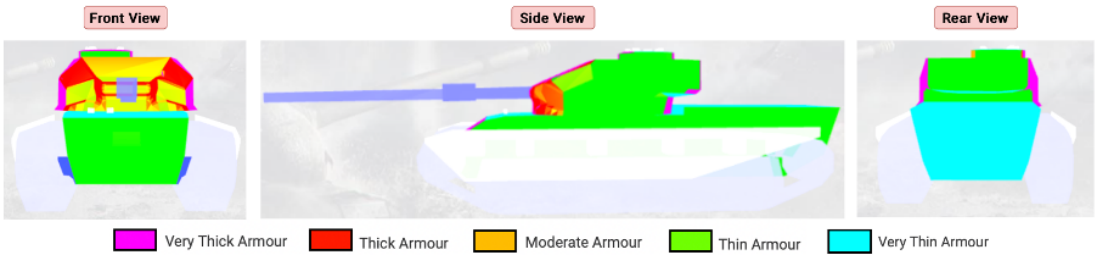

Figure 4.10: The armour profile of the Centurion. (Adapted from TanksGG)

All tanks were created with intricate detail in Blender, as the topology of the tank's mesh directly impacted the characteristics of its armour. Advanced features such as Boolean cutting and low-level mesh manipulation were used to accurately shape the armour model before importing it into Unity. Figure 4.11 highlights the level of detail in the IS-7 armour model in Blender.

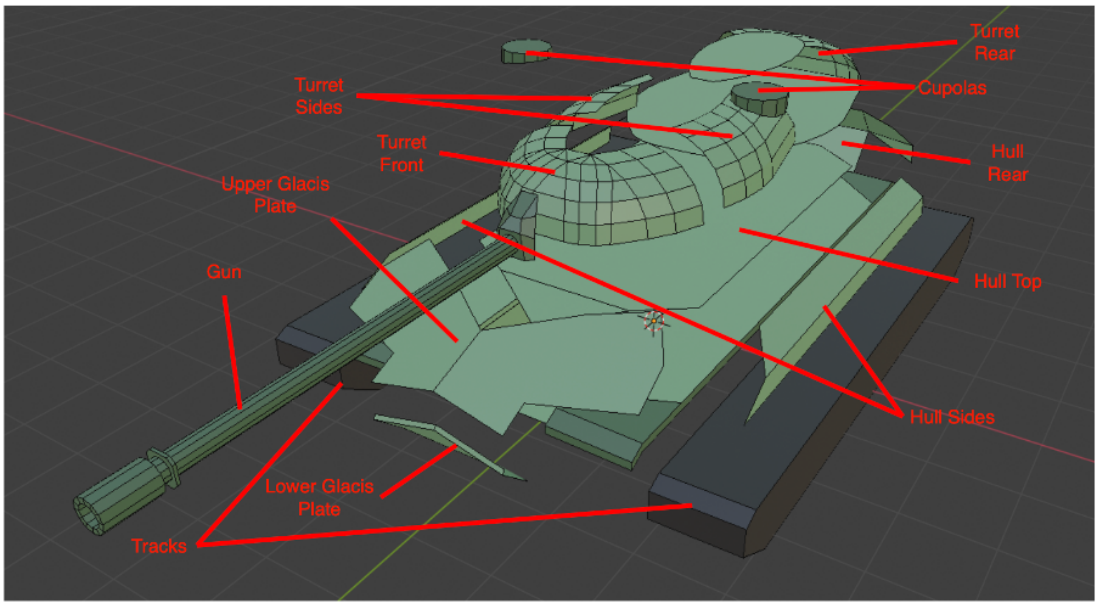

Figure 4.11: The custom IS-7 has over 20 distinct parts.

4.4 Battle Environments

World of Tanks contains over thirty maps in many different settings, including cities, mountains, and valleys. Each map poses its own set of challenges for different tanks.

Slow, lumbering heavy tanks will struggle in open plains since cover will be limited, leaving them exposed to enemy fire with nowhere to hide. On the other hand, light tanks can easily move around the open plains to flank vulnerable heavy tanks. Medium tanks can use their good mobility and armour to quickly make their way into advantageous positions on the map. Prokhorovka is a rural map that is situated in open plains, and a heat map of tank movements is shown in Figure 4.14. The custom Unity implementation of Prokhorovka is shown in Figure 4.12 and Figure 4.13. Water was excluded from the map to reduce algorithm complexity.

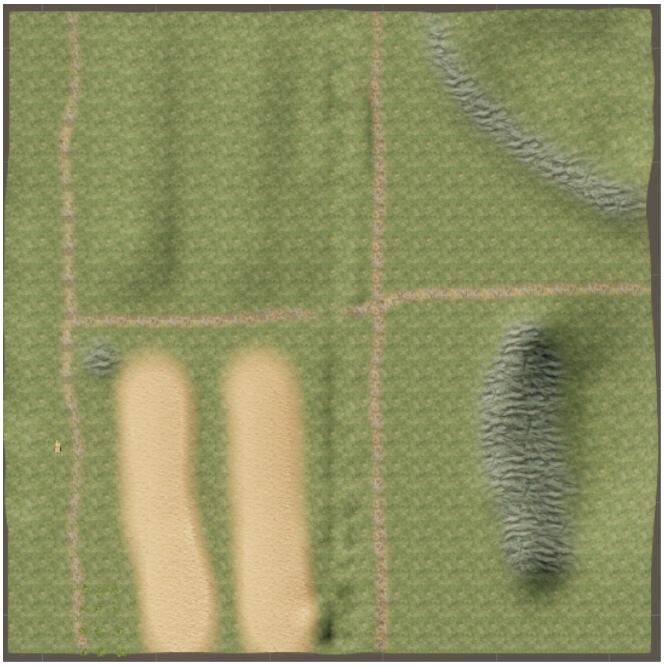

Figure 4.12: An overhead view of custom Prokhorovka in Unity.

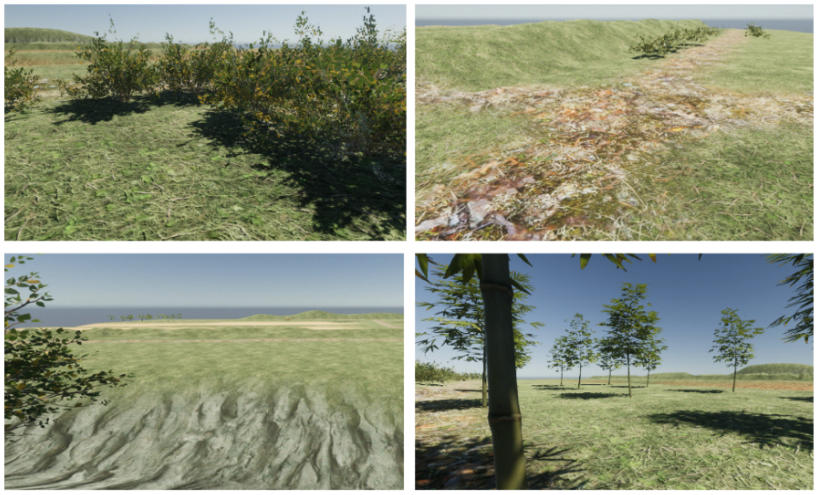

Figure 4.13: Scenic views from custom Prokhorovka, highlighting the level of detail.

The teams spawn on the north and south sides of the map. Heavy tanks typically move towards the centre of the map, where they are sheltered from

enemy fire by the train tracks and boulders. Medium tanks typically move to the east side of the map, where they can use their gun depression to go hull-down behind hilly terrain. Light tanks typically move through the middle east, west and centre of the map, where they may stay undetected behind bushes while firing at enemy heavy and medium tanks. As the battle unfolds, the tanks will need to move to other positions on the map, since the advantageous positions remain fluid throughout the battle.

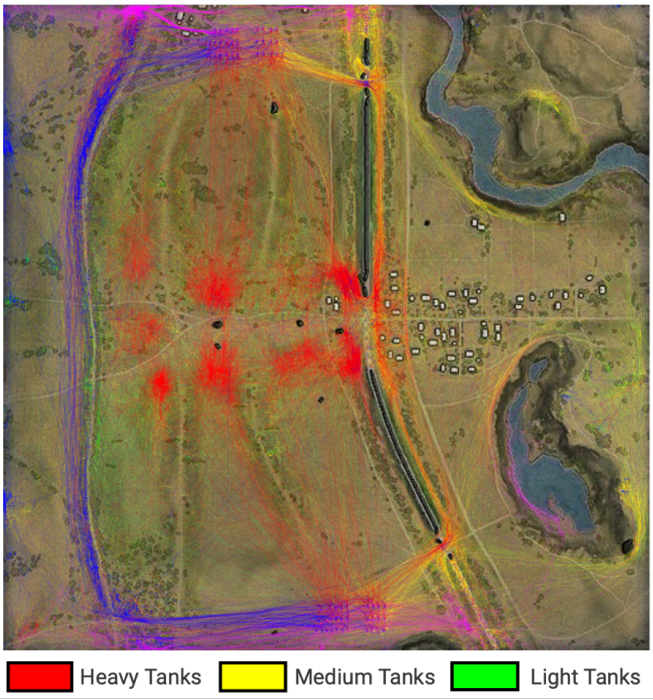

Figure 4.14: A heatmap of tank movements in Prokhorovka. (Adapted from WoT Inspector)

Heavy tanks excel in urban maps, where their lack of mobility is not an issue. Urban maps also offer plenty of cover, such as buildings and rubble, allowing heavy tanks to side-scrape and go hull-down. Light tanks will struggle in city maps, since the narrow streets will limit their evasive tactics, and their lack of armour will leave them very vulnerable. Even if medium tanks can arrive at

advantageous positions quickly in urban maps, they will soon be pushed out by the advancing heavy tanks. An close-quarters urban map was created for this project. This claustrophobic map takes inspiration from the block-style architecture in central Barcelona, giving rise to exciting close-range battles where heavy tanks can showcase their armour. An overhead view of this map is shown in Figure 4.15, and scenic views are shown in Figure 4.16, highlighting the intricate details in the custom-made buildings and props. All props and 3D models were completely custom made in Blender for this project.

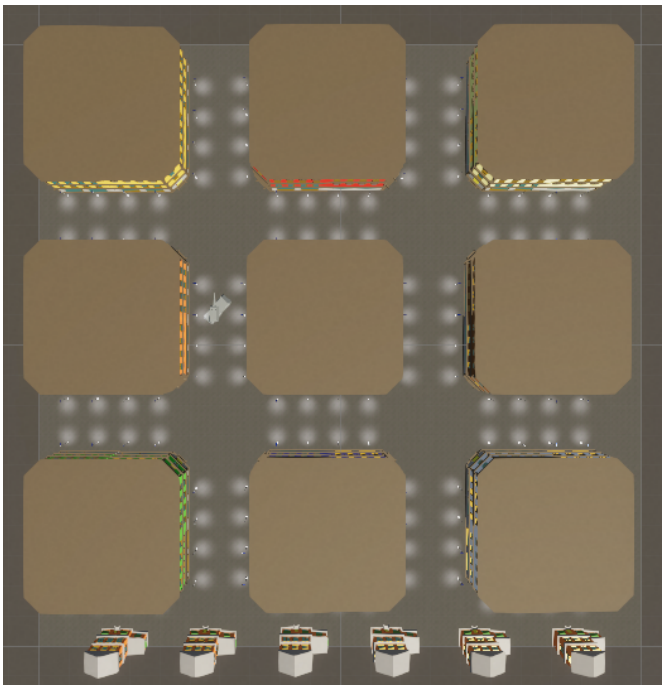

Figure 4.15: An overhead view of the Barcelona map.

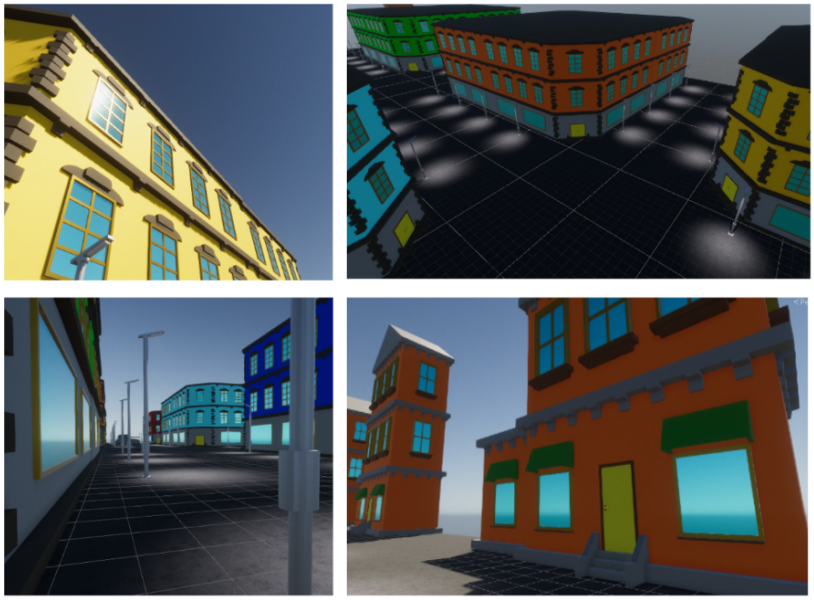

Figure 4.16: Scenic views from Barcelona, highlighting the level of detail.

Chapter 5

Methodology

Here you can describe your proposed new models.

5.1 Training Architecture

5.1.1 Introduction

Keras is a popular Python library which provides an intuitive API for Tensorflow. Keras runs on the Python runtime environment, but the game itself runs on the Unity environment. Bridging the communications between Keras and Unity proved to be a significant challenge in this project, and two different training architectures were tested.

The first training architecture utilised SciSharp, an open-source .NET library for Tensorflow and Keras on C-Sharp. However, this soon proved to be difficult to manage, since Unity does not natively support NuGet packages (NuGet packages are a collection of code assets bundled into a library, which can then be imported into existing projects). Even though SciSharp was eventually imported into the project, several issues such as poor performance and high complexity soon emerged. As a result of these issues, SciSharp was not used to import Keras

functionality into C-Sharp.

The second training architecture involved a client-server framework consisting of a local Python server and Unity client connected via TCP. The Keras model was managed on the server, and the client can utilise and train the model by sending commands and data to the server. This approach proved to be significantly more effective and easier to both use and scale upwards than SciSharp, and therefore was used for this project.

5.1.2 DQN Training Architecture

DQN was the first algorithm implemented for this project. DQN only needs one instance of the agent and environment for training. A replay buffer that records recent transitions in order to break up correlation in the input data was also required, and this was stored on the server.

Before training can take place, the replay buffer needs to be sufficiently populated in order for random sampling to be carried out. For each buffer population iteration, the Unity client observes the environment, takes a random action, and receives a reward for this action. The environment is observed again to detect any changes caused by the action. The initial state, the action taken, the reward, the resulting state, the episode terminator (a boolean which denotes if the action taken lead to a terminal state), and a 'Populate Buffer' command are encapsulated into a string and sent to the input queue of the client's TCP thread. The client's TCP thread sends the data and the command to the python server's TCP thread over the designated port. The server then appends the state transition data to its replay buffer. This process is repeated a number of times until the replay buffer is sufficiently populated.

In every training iteration, the Unity client observes the environment and sends this observation with a 'Prediction' command to its TCP thread, which in

turn forwards it to the server's TCP thread. The python server then converts this state string to a numpy format and passes it into the prediction neural network. The prediction network outputs the Q-values of each action given the input state, and these Q-values are sent back to the client's TCP thread. The client's TCP thread then appends the network output to its output queue. The client reads the output queue and takes an appropriate action with epsilon-greedy exploration. Epsilon is annealed slowly as training progresses, so the agent takes random actions at the beginning, and slowly starts to follow its learned policy. The initial observation, the action taken, the reward received, and the resulting observation are sent to the server via the client's TCP thread to be added to the replay buffer.

After a fixed interval of a number of steps taken by the agent, the prediction neural network is trained. The client sends a 'Train' command to the server, and waits for the server to finish training. Meanwhile, the server samples a random batch of transitions from the replay buffer. For every transition in the mini batch, the resulting state of the transition is passed into the prediction network. The highest Q-value for any action is combined with the reward taken for that action in the transition, and this sum is multiplied by the discount factor γ, giving the target Q-value. The initial state of the transition is also passed into the prediction network, giving the predicted output. The Q-value of the action which corresponds to the action of the target Q-value is then replaced by the target Q-value, which gives the optimal prediction network output for the initial state of the transition. The initial states of every transition are encapsulated into a list of neural network inputs, and the optimal prediction network outputs are encapsulated into a list of optimal neural network outputs. The prediction network is then trained using these inputs and outputs. The entire DQN training architecture is illustrated in Figure 5.1.

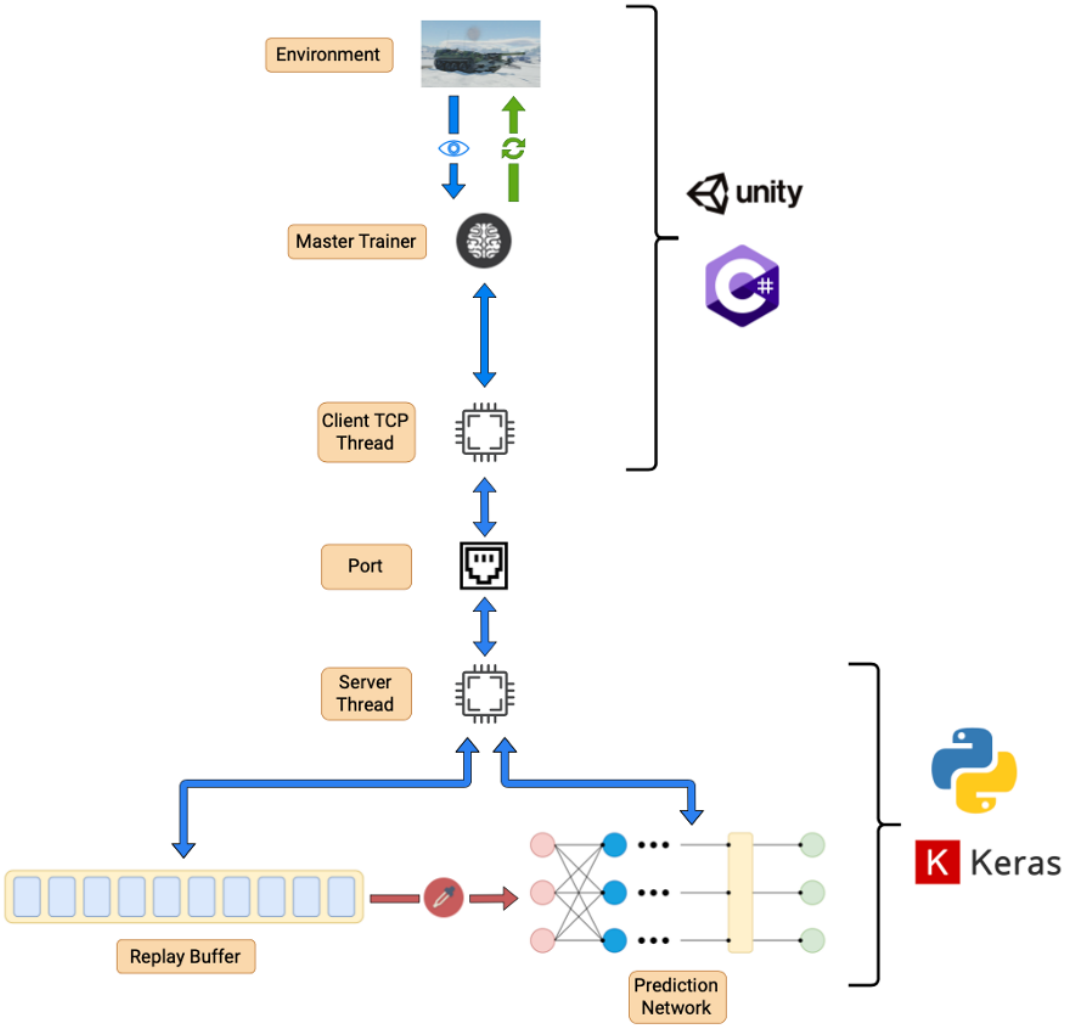

Figure 5.1: The DQN training architecture

5.1.3 A3C Training Architecture

Unity is inherently single-threaded as all game code is executed on the main thread. This characteristic was introduced to reduce development difficulty, but it also prevents multi-threaded computation as the variables at runtime cannot be accessed outside of the main thread. Even though this effectively prevents running multiple environments simultaneously in a true multi-threaded fashion, the training architecture can still make use of multi-threaded computation. The architecture of the back-end training framework is presented in Figure 5.2.

In every training iteration, the master trainer observes the states of all environments and sends the string representation of these observations and a 'Prediction' command to the input queue of each environment's corresponding TCP thread. This TCP thread forwards these observations to its corresponding server thread over a designated port. Each server thread acts as an A3C worker, as each thread contains its own local neural network and A3C buffer. After reading the command and the observation, the server thread converts the string representation of the state into a suitable input format and passes it through the local neural network. The output of the local neural network is a set of probabilities for each action in the action space and the value of the current state of the environment. The output is converted into a string and sent back to the environment's TCP thread via the designated port.

After receiving the response from the server, the communication thread appends this response to its output queue. After waiting for the responses for every environment to be received from the server, the master trainer takes an appropriate action on every environment. The environment updates are executed in a single-threaded fashion on the Unity main thread. The previous state, action taken, the reward received, the observation of the next state, and the episode terminator are appended to the input queue of the environment's TCP thread

with an 'Populate Buffer' command. These values are sent to the appropriate server thread, unpacked, and promptly added to the server thread's A3C buffer.

Once an A3C worker's buffer has reached maximum capacity, or if the current episode has reached a terminal state, the A3C worker's local model is trained using the transitions from its buffer. The n-step return and entropy are calculated, the loss is computed, and the weights of the server's global model are updated accordingly in a thread-safe fashion.

This multi-threaded approach was three times faster than using single-threaded training across multiple environments. The majority of the time required to complete one training update across multiple environments came from the neural network computation time itself on the server. Due to this issue and Unity's single-threaded limitations, training was very slow. When training with twelve environments for A3C, the game environment was only running at ten frames per second, therefore taking many hours for even simple reinforcement learning scenarios to converge.

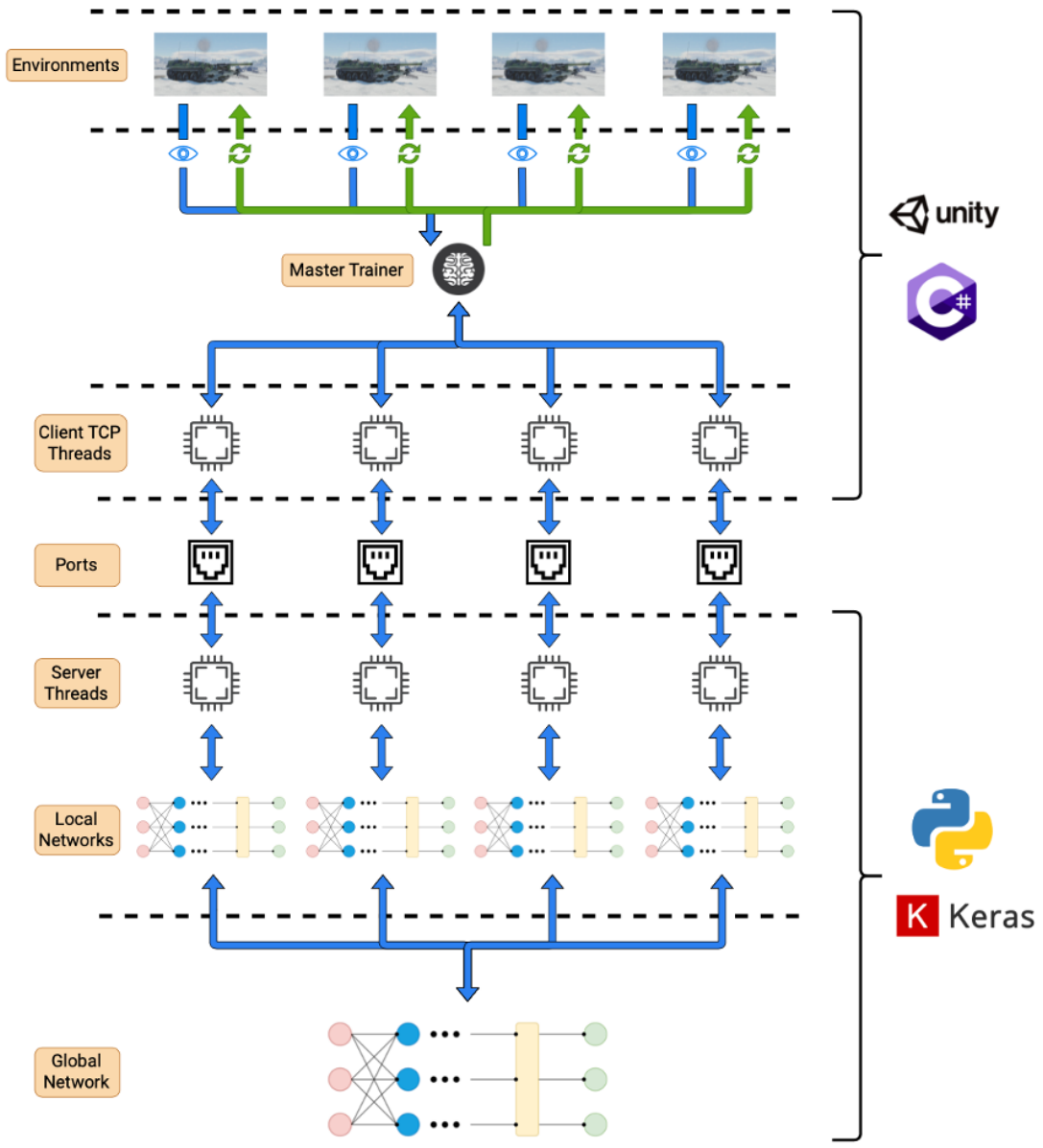

Figure 5.2: A suitable caption

5.2 Evaluation of Training Architectures

5.2.1 Preliminary Evaluation of Server Functionality

The unmodified implementations of DQN and A3C were tested in a number of diverse mini-games from the OpenAI Gym library. This testing helped to ensure that the server-side Keras models and AI functionality were performing as expected.

The first game tested was Cart Pole. In this game, the goal of the agent is to keep a pole balanced over a moving platform. The environment observations consist of four velocities. There are two possible actions which move the agent left or right. The episode terminates when the joint velocities are too high, indicating that the pole is about to tip over. The agent receives a positive reward for each step of the episode, since the pole will be balanced as long as the episode is running. The agent receives a negative reward when the episode has terminated.

Figure 5.3: The Cart Pole game.

The second game tested was Mountain Cart. In this game, the agent is situated between two steep hills, with the goal location being at the top of the second hill. The agent cannot move to the goal location directly at the start of the episode, since it lacks the engine power to do so. Instead, the agent must gain momentum by moving between the two hills. The environment observations con-

sist of the height and X position of the cart. There are two possible actions which move the agent left or right. The episode terminates when the agent reaches the goal location, and a positive reward is given when it does so. A negative reward is given for all steps until episode termination.

Figure 5.4: The Mountain Cart game.

The preliminary testing of DQN and A3C gave positive results, and proved that the server-side models and AI functionality were implemented correctly. These results are shown and explained in Section 6.

5.2.2 Preliminary Evaluation of Client-Server Architecture

The cohesion of Unity with Keras using both architectures were evaluated using a number of custom-built Unity mini-games.

The first Unity mini-game is Block Finder, where an agent attempts to move towards a target position on a 2D surface. The environment observation consists of the coordinates of the agent and the target block. The agent has four actions which allow it to move up, down, left or right. The episode terminates when the agent touches the target block. The agent receives a small positive reward if it moves closer to the target, a small negative reward if it moves away from the

target, and a large positive reward upon termination.

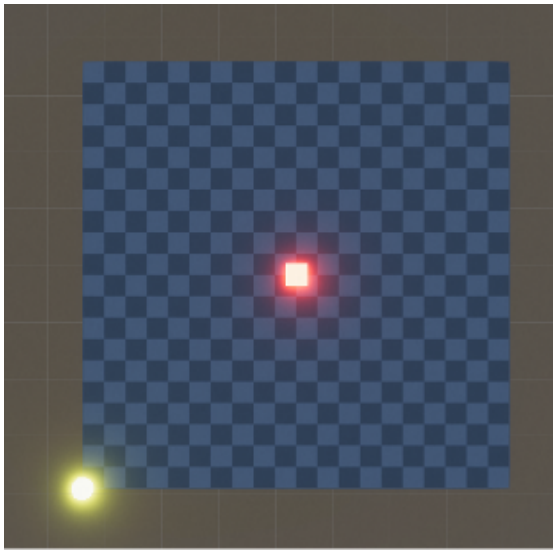

Figure 5.5: The Block Finder game. The yellow agent must learn to detect the red block.

The second Unity mini-game is Block Aimer, where an agent attempts to aim at a block as quickly as possible. This mini-game looks identical to Block Finder The environment consists of the coordinates of the agent and the target block. The agent has two actions, which allow it to turn to the left or to the right by one degree. The reward is the squared difference between the target angle (the angle of the agent which makes it point directly at the target block) and the current angle (the current angle of the agent). At the start of each episode, the agent is instantiated at a random position. The episode terminates when the agent has either reached a maximum number of steps, or if the difference between the target angle and the current angle is less than five degrees.

Unfortunately, Block Aimer did not converge. Many different configurations were tested with both A3C and DQN. Different neural network architectures, buffer sizes, batch sampling sizes, and other hyperparameters were adjusted, but no solution worked. Block Aimer could not be skipped, since aiming functionality

was critical in World of Tanks. If the model could not aim at a block in 2D space, it would absolutely not be able to aim at a weak spot in 3D space.

After four weeks of constant testing and debugging, Block Aimer was fixed. The issue was not with the training architecture, but a bug in the Random package for generating random numbers. Unfortunately, it became too late to continue with the project after this time, and further work was ceased.

5.3 Proposed Methodology

5.3.1 Introduction

Even though that the results of training the models were not plentiful, a significant amount of theoretical groundwork was laid out for this project. This groundwork, inspired by recent advances in the fields of artificial intelligence and autonomous game-playing agents, would form the foundation of the approaches taken to overcome further challenges in this project.

All software components on both the client and the server are built following object-oriented principles, and particular care was taken when designing and developing the World of Tanks Unity emulator and Python server to facilitate the efficient creation of new features and changes to training architectures.

The learning task can be divided into two categories - armour knowledge and navigation knowledge. Armour knowledge is the agent's ability to use its armour effectively and exploit enemy armour weak spots knowledgeably. After observing the locations of enemy tanks, the agent must orient itself in an optimal manner which maximises the relative thickness of the armour exposed to the enemy. Navigation knowledge is the agent's ability to move to advantageous positions on the map as the battle progresses. This knowledge would be very difficult for the agents to learn, since even professional human players may find it difficult to

decode the information on the battle map and act accordingly, as this is the most important skill in World of Tanks.

5.3.2 Nature of the Environment

5.3.2.1 Observations

Before any learning can commence, the structure of the environment observations and a suitable reward function will be put forward. The environment details will be captured by recording the image frames of the game. Even though this approach gives a very descriptive representation of each state, the game will now be partially observable since objects outside the camera's field of view will not be detected by the agent. This can make reinforcement learning significantly more complex since an observation of a state will need to be represented by a sequence of game frames in order for the Markov decision process property to hold true.

The game frames will be pre-processed by converting them to gray scale and significantly down scaling them, as proposed by Mnih et al. (2015), to minimise input dimensionality.

5.3.2.2 Actions

There are 12 actions in the World of Tanks, as shown below in Table 6.1.

Action	Description	Type
Move Forward	Moves the tank forward	Discrete
Move Backward	Moves the tank backward	Discrete
Turn Left	Turns the tank to the left	Discrete
Turn Right	Turns the tank to the right	Discrete
Mouse X-axis	The change in the mouse X position since the last frame	Continuous
Mouse Y-axis	The change in the mouse Y position since the last frame	Continuous
Load AP	Reloads an AP shell into the gun	Discrete
Load APCR	Reloads an APCR shell into the gun	Discrete
Load HEAT	Reloads a HEAT shell into the gun	Discrete
Load HE	Reloads a HE shell into the gun	Discrete
Fire shell	Fires the tank's gun	Discrete
Zoom in/out	The position of the mouse scroll wheel	Discrete

Table 5.1: The actions in the World of Tanks emulator.

5.3.2.3 Rewards

Separate reward functions will be used for the learning tasks. In the offensive armour knowledge learning task, if the agent fires at the enemy and penetrates the armour, the reward will be a function of the enemy's health and the damage done by the agent:

$$r_{pen} = \frac{a_A}{h_E} \text{ if } a_A > h_E, \text{ else r} = 1$$

Where a_A is the damage caused by the agent's shell, and h_E is the remaining health on the enemy. This reward is limited to a maximum value of 1 for learning

stability. This reward function manipulates the agent into prioritising enemy tanks with low health, since less enemy tanks will always mean an easier chance to win the game. The penalty for missing or not penetrating the enemy armour will be -0.1, since the agent will need to protect itself for the duration of its reload time until it can fire again, leaving it vulnerable to incoming fire. From this reward function, the agent will learn about weak spots on enemy armour.

In the defensive armour knowledge learning task, if the agent blocks an incoming shell, it receives a positive reward of 0.5. If the incoming shell penetrates its armour, it receives a negative reward of -0.5:

$$r_{arm} = 0.5 \text{ if blocked, else } r_{arm} = -0.5$$

In the navigation task, the agent will receive a small positive reward of proportional to the distance it travelled from the last step, which encourages exploration of the maps.

5.3.3 Proposed Training Process

World of Tanks is a difficult game to master, and it can take years for human players to grasp complex gameplay tactics. This steep learning curve gives rise to very long training times until the agent can learn policies similar to professional human gameplay. The training time is further increased by the distributed nature of the training architecture and performance limitations of the Unity game engine. Therefore, it is absolutely essential to utilise deep learning from demonstrations (Hester et al., 2018), since demonstrations can help an agent to develop a policy inspired by an expert policy.

A gameplay recording system can be constructed in Unity to break down key tactics of an expert player into a set of state transitions. These tactics include side-scraping, hull-down, flanking manoeuvres, and general navigation. The tactics will be carried out across both game maps. Hester et al. (2018)

recorded up to 75,000 transitions for the demonstration data, but since World of Tanks is significantly more complex than Atari, more than a million transitions will be required to form the demonstration dataset for each map.

Each state transition stores the initial state of the environment, the action taken, the reward for that action, and the next state of the environment. The state transitions will be forwarded to the Python server via the client-server architecture put in place previously, where they can be stored in the replay buffer.

Neural network architecture is an important consideration. Lample and Chaplot (2018) showed that breaking down the entire learning task into different phases promotes simplicity and efficiency, since the neural networks can be treated as modular entities which can be interchanged and trained in parallel. Following this ideology, the armour and navigation tasks will be executed on separate neural networks.

The agent will be pre-trained to imitate the expert player before training can commence. Randomly sampled mini batches will be used to update the weights of the model during pre-training. Following Hester et al. (2018), four different losses will be applied at this stage: a 1-step Q-learning loss, an n-step Q-learning loss, a supervised large margin classification loss, and an L2 regularisation loss. The Q-learning losses guide the network to satisfy the Bellman equation for Q-learning prior to training, and the supervised loss will classify the expert's actions. The supervised loss is as follows:

$$J_S(Q) = max_{a \in A}[Q(s,a) + l(a_E, a)] - Q(s, a_E)$$

Where a_E is the action taken by the expert in state s, and $l(a_E, a)$ denotes the difference between the expert action and the agent action. As with A3C, the n-step returns will help to propagate the rewards of actions back to earlier states:

$$r_t + \gamma r_{t+1} + ... + \gamma^{n-1} r_{t+n-1} + max_a \gamma^n Q(s_{t+n}, a)$$

Where r_t is the reward at time t, γ is the discount factor, and n is the total number

of steps in n-step return. Once pre-training has completed, the agent can start learning the policy. Since the armour and navigation tasks are to be handled in separate networks, they will be trained in parallel. For training, prioritised experience replay (Schaul et al., 2016) will be used to select a mini batch of transitions which contain both demonstration transitions and agent transitions.

The A3C training architecture developed for this project can be quickly re-purposed to facilitate asynchronous n-step Q-learning (Mnih et al., 2018).

Mouse movement actions for aiming the tank are continuous, therefore the DDPG approach presented by Lillicrap et al. (2019) will be used to handle the continuous actions. However, the methodology for combining DDPG with LSTM networks and performing soft target network updates with DRQN is not certain, and this would be a large upcoming hurdle in this project. Instead, mouse movements could be discretised to just up, down, left and right, but this gives rise to very slow mouse movements, nor is this the correct means of controlling the mouse in World of Tanks.

Game-specific features such as the number of enemy tanks visible on the screen, and the number of friendly and enemy tanks remaining in the battle, will be provided to the model to aid the learning process. Lample and Chaplot (2018) proved that combining game-specific information with deep reinforcement learning can boost training performance.

Even though the asynchronous multi-threaded architecture will be significantly faster than a synchronous single-environment approach, the limitations of the Unity main thread will still make training painstakingly slow, especially since objects of high graphical fidelity such as tanks and trees will be on the screen. The proposed training architecture is presented in Figure 5.6.

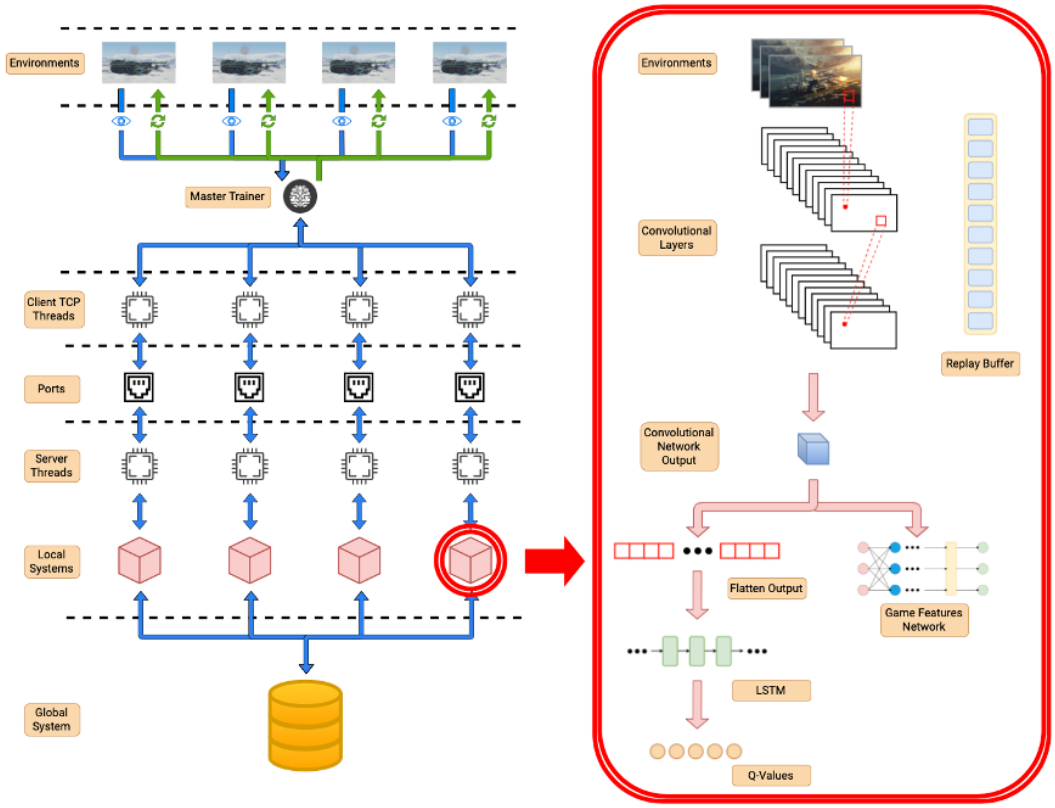

Figure 5.6: The proposed training architecture should more time have been available for this project.

Chapter 6

Results

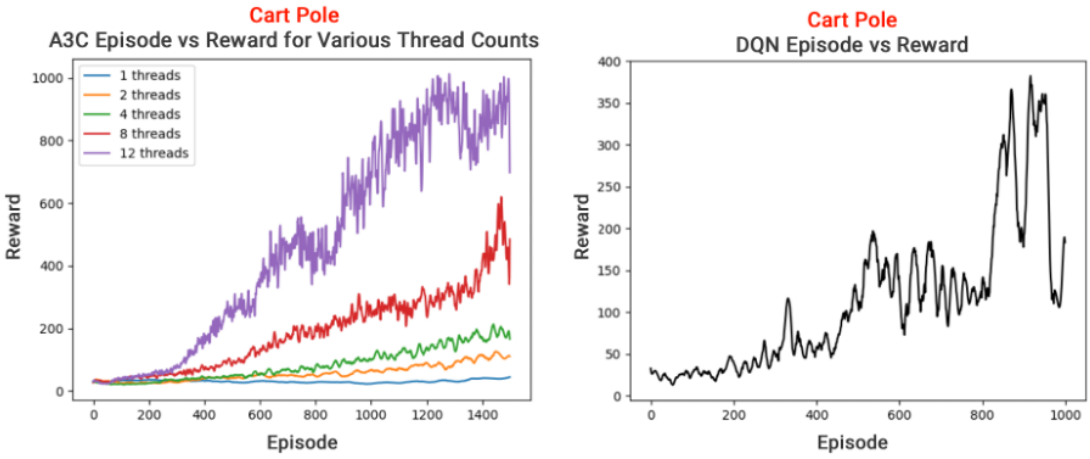

Figure 6.1: Results for Cart Pole with A3C (left) and DQN (right).

Figure 6.1 shows the results for Cart Pole when training with A3C and DQN with 1000 episodes. In both implementations, a neural network configuration of two dense layers with 64 neurons each was used. For A3C, the buffer size was 30, the learning rate was 0.0001, and the optimiser was Adam optimiser. For DQN, the replay buffer size was 2000, the learning rate was 0.0001, and Adam optimiser was also used.

The results show that A3C reaches convergence faster when more agent-

environment pairs are used during training, as this helps to correlate incoming data. However, A3C still learns with a single thread, albeit very slowly. With DQN, the best score is 400, which shows that A3C learns faster than DQN. Cart Pole does not terminate with an excellent policy, so the episode length was capped at 1000 for both A3C and DQN.

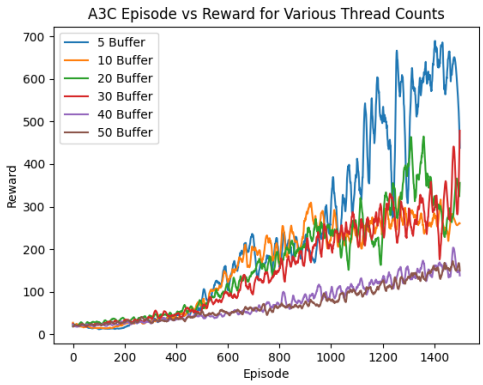

Figure 6.2: Results for A3C with various buffer sizes.

The effects of changing the buffer sizes in A3C were also investigated, as shown in Figure 6.2. The same hyperparameters were used as before, and four threads were used for training. For Cart Pole, training speed was inversely proportional to the buffer size. However, this is to be expected, as Cart Pole is a simple game, and a large buffer is not necessary. For more complex games, a small buffer size will inhibit the agent from learning.

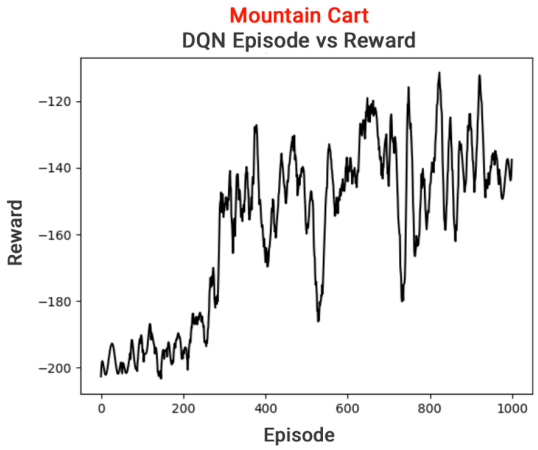

Figure 6.3: Results for Mountain Car with DQN.

Figure 6.3 shows the results of Mountain Car with DQN. The same neural network configuration of two dense layers with 64 neurons each was used here. The replay buffer size was 2000, the learning rate was 0.0001, and Adam optimiser was used. Mountain Car is densely populated with negative rewards and a positive reward is only given when the agent reaches the goal. This prevents on-policy algorithms such as A3C from being used in this game. DQN performs very well, and the optimal policy was found after 800 episodes.

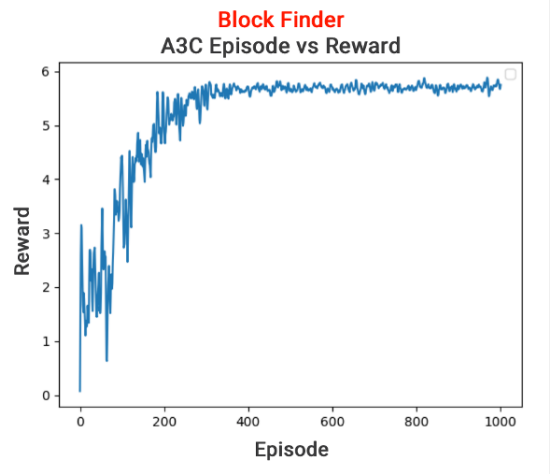

Figure 6.4: Results for Block Finder with A3C.

Figure 6.4 shows the results of Block Finder with A3C. Once again, the same neural network configuration of two dense layers with 64 neurons each was used here. The buffer size was 30, the learning rate was 0.0001, and Adam optimiser was used. 12 agent-environment pairs were used for training, which allowed the agent to learn the optimal policy after just 350 episodes.

Chapter 7

Conclusion

Here you must zoom back out to evaluate the thesis. Mention limitations and weaknesses as well as contributions.

This project created an emulator of World of Tanks. Four different tanks, the Maus, IS-7, EBR, and Centurion were developed in Blender to maintain a diverse selection of tanks in the emulator. The emulator was created using the Unity game engine's HDRP pipeline for high graphical fidelity. Two different maps were also created simulating different environments.

A distributed asynchronous training architecture was created, which trains the agent using multiple agent-environment pairs. Each agent-environment pair runs on its own thread. This architecture performed very well in Cart Pole, Mountain Car, and Block Finder. However, time ran out for this project after encountering a bug with Block Aimer.

Even though practical development of the project ceased after this, a well thought out plan was created, detailing how to overcome future issues in this project should Block Aimer have been solved sooner. This solution involved using deep recurrent Q-learning and LSTMs, using pre-processed game image frames as the input to the model.

References

Autodesk. Autodesk maya homepage. https://www.autodesk.eu/products/maya/overview. Accessed: 2021-17-08. 24

Blender. Blender homepage. https://www.blender.org. Accessed: 2021-17-08. 25

CryEngine. Cryengine homepage. https://www.cryengine.com. Accessed: 2021-17-08. 23

Drawing Database. T-54 blueprints. https://drawingdatabase.com/t-54/. Accessed: 2021-15-08. v, 3

Peter Dennis. New vanguard 150, war elephants. https://www.historynet.com/carthaginian-war-elephant.htm, 2012. Accessed: 2021-15-08. v, 2

Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. pages 29–37, 2015. 16

Jay Hemmings and Guest. The maus tank – an crazy invention, but would it have been effective enough to change the outcome of wwii? https://www.warhistoryonline.com/war-articles/the-maus-tank.html. Accessed: 2021-17-08. 25

Hester, Vecerik, Pietquin, Lanctot, Schaul, Piot, Horgan, Quan, Sendonaris, Osband, Dulac-Arnold, Agapiou, Leibo, and Gruslys. Deep q-learning from demonstrations. In *Proceedings of The Thirty-Second AAAI Conference on Artificial Intelligence*, 2018. vi, 18, 20, 48, 49

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, 07–09 Jul 2015. 15

Guillaume Lample and Devendra Singh Chaplot. Playing fps games with deep reinforcement learning, 2018. vi, 16, 18, 49, 50

Lillicrap, Hunt, Pritzel, Heess, Erez, Tasa, Silver, and Wiersta. Continuous control with deep reinforcement learning. In *Proceedings of ICLR 2016*, 2019. vi, 14, 16, 50

Megapixie. Hull-down tactics demonstration. `https://commons.wikimedia.org/wiki/File:Hull_down_tank_diagram.png`. Accessed: 2021-20-06. v, 9

Mnih, Kavukcuoglu, Silver, Rusu, Veness, Bellemare, Graves, Riedmiller, Fidjeland, Ostrovski, Petersen, Beattie, Sadik, Antonoglou, King, Kumaran, Wierstra, Legg, and Hassabis. Human-level control through deep reinforcement learning. volume 518, pages 529–533, 2015. v, 10, 11, 14, 17, 21, 46

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1928–1937, 2018. vi, 20, 21, 50

Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, Shane Legg, Volodymyr Mnih, Koray Kavukcuoglu, and David Silver. Massively parallel methods for deep reinforcement learning. In *Proceedings of The International Conference on Machine Learning*, 2015. 20

Mark Nash. Is-7 (object 260). `https://tanks-encyclopedia.com/coldwar/USSR/is-7-object-260/`. Accessed: 2021-17-08. 26

Andrew Ng. *Shaping and Policy Search in Reinforcement Learning*. PhD thesis, University of California Berkeley, 2003. 17

OBRUM. Polish light tank pl-01. `http://www.strategic-bureau.com/en/pl-01-light-tank-poland/`. Accessed: 2021-15-08. v, 2

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. 2016. v, 12, 13, 14, 50

Silver, Lever, Hess, Degris, Wierstra, and Riedmiller. Deterministic policy gradient algorithms. In Eric P. Xing and Tony Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 387–395, Bejing, China, 22–24 Jun 2014. PMLR. 14

Tank Archives. Is-7 historical photograph. `http://www.tankarchives.ca/2016/05/is-7-titan-late-for-war.html`. Accessed: 2021-15-08. v, 6

Tanks-Encyclopedia. Panhard ebr. `https://www.tanks-encyclopedia.com/coldwar/France/Panhard_EBR.php`. Accessed: 2021-17-08. 27

TanksGG. Wot armour models. `https://tanks.gg`. Accessed: 2021-15-08. v, vi, 6, 7, 9, 26, 27, 28, 29

The Tank Museum. Centurion. https://tankmuseum.org/tank-nuts/tank-collection/centurion. Accessed: 2021-17-08. 28

Uhlenbeck and Ornstein. On the theory of brownian motion. In *Proceedings of the Physical Review Conference*, volume 36, pages 823–841, Sep 1930. 15

Unity. Unity homepage. https://unity.com. Accessed: 2021-17-08. 23

UnrealEngine. Unreal engine homepage. https://www.unrealengine.com/en-US/. Accessed: 2021-17-08. 23

Wargaming. Wourld of tanks homepage. https://worldoftanks.com. Accessed: 2021-17-08. 8

WoT Inspector. Prokhorovka heatmap. https://wotinspector.com/en/heatmaps/?ts=1616230933&id=4&platform=pc. Accessed: 2021-17-08. vii, 32