UiA University of Agder

Master

Mastering DeepRTS with Transformers

ANDREAS HØIBERG EIKE AND PEDRO ALVES

SUPERVISOR
Morten Goodwin

University of Agder, 2023
Faculty of Engineering and Science
Department of Information and Communication Technology

Mandatory Group Declaration

Each student is responsible for familiarizing themselves with what constitutes legal aids, guidelines for their use, and rules regarding source usage. The declaration aims to raise students' awareness of their responsibility and the consequences of academic misconduct. Failure to submit the declaration does not exempt students from their responsibility.

1.	We hereby declare that our submission is our own work, and we have not used any other sources or received any assistance other than what is mentioned in the submission.	**Yes** / No
2.	**Furthermore, we declare that this submission:** • Has not been used for any other examination at another department/university/college, both domestic and international. • Does not reference the work of others without attribution. • Does not reference our own previous work without attribution. • Includes all references listed in the bibliography. • Is not a copy, duplicate, or transcription of someone else's work or submission.	**Yes** / No
3.	We are aware that a violation of the above is considered academic misconduct and can result in the annulment of the examination and exclusion from universities and colleges in Norway, pursuant to the Universities and Colleges Act §§4-7 and 4-8, and the Regulations on Examinations §§ 31.	**Yes** / No
4.	We are aware that all submitted assignments may be subject to plagiarism detection.	**Yes** / No
5.	We are aware that the University of Agder will handle all cases where there is suspicion of academic misconduct in accordance with the college's guidelines for handling cases of academic misconduct.	**Yes** / No
6.	We have familiarized ourselves with the rules and guidelines for the use of sources and references on the library's website.	**Yes** / No
7.	We, as a majority, have agreed that the effort within the group is noticeably different and therefore wish to be assessed individually. Ordinarily, all participants in the project are assessed collectively.	Yes / **No**

Publication Agreement

Authorization for electronic publication of the assignment The author(s) hold the copyright to the assignment. This includes, among other things, the exclusive right to make the work available to the public (Copyright Act §2).

Assignments that are exempt from public disclosure or subject to confidentiality will not be published.

We hereby grant the University of Agder the free right to make the assignment available for electronic publication:	**Yes** / No
Is the assignment bound by a confidentiality agreement?	Yes / **No**
Is the assignment exempt from public disclosure?	Yes / **No**

Acknowledgements

Mastering DeepRTS with Transformers is a master thesis written by Andreas Høiberg Eike and Pedro Alves, as students at the University of Agder (UiA) Grimstad, in the Department of Information and Communication Technology (ICT). As such, this research concludes the author's master's program in ICT.

The authors are grateful to our supervisor Professor Morten Goodwin, for lending his expertise and experience within artificial intelligence to help us complete our master's thesis research. We would also like to thank Associate Professor Per Arne Andersen for making the DeepRTS game used in this work, for letting us contribute to fixing issues in the game and for fixing some issues himself. Finally, we would like to thank Senior Advisor Sigurd Brinch for lending us access to UiA servers with a dedicated GPU, which accelerated the reinforcement learning training process. We could not have completed our research without their guidance and knowledge.

Grimstad
June 2023
Andreas Høiberg Eike and Pedro Alves

Abstract

The Transformer deep learning model has recently proven its superiority in tasks like natural language processing and computer vision, as tools like ChatGPT and DALL-E have become widespread and helps humans complete tasks faster with high accuracy. This comes from the ability of Transformer models to comprehend sequential data by weighing the importance of each token in sequences through an attention mechanism and being trained on massive amounts of data. As researchers seek to apply Transformer models to other disciplines, the sequential nature of reinforcement learning tasks becomes an interesting study area. Despite the demonstrated superiority of transformer-based models in various domains, their adoption within reinforcement learning paradigms, particularly within game-based learning environments, has yet to become widespread. In particular, reinforcement learning problems where an intelligent agent learns how to act in a video game are interesting, as they can help simulate real-life scenarios and therefore make autonomous systems less expensive and safer to train. Real-Time Strategy games are complex video games where players must develop a strategy in real-time to gain an advantage over other players, and reaching game objectives often involve performing a specific sequence of actions, making them an excellent area of study for reinforcement learning combined with Transformers.

This thesis explores, evaluates and improves Transformer models applied in Real-Time Strategy Game environments with a particular focus on limited data and computational power resources. To this end, DeepRTS [7] is chosen as the reinforcement learning environment for its high performance and simplified game mechanics, but also to enrich its relatively small research domain. This work implements several sub-environments in DeepRTS with various objectives and levels of complexity to give agents a diverse range of tasks and to compare deep learning algorithms. The authors of this thesis also contributed to the DeepRTS project by fixing source code issues to improve performance. As there is no publicly available dataset for DeepRTS to train a Transformer model on, this thesis proposes a novel model, namely the Genetic Algorithm Decision Transformer, a new implementation for data generation in reinforcement learning environments by leveraging the autoregressive Decision Transformer [19] model for action prediction. The novelty lies in using the genetic algorithm to select the best data samples from a pool to train a Decision Transformer agent. Results are compared against a Double Deep Q-learning agent and a standard Decision Transformer agent, the latter being trained using different datasets, and results show its dependency on high-quality data. Genetic Algorithm Decision Transformer improves the aforementioned algorithms by generating its own dataset with high-quality data samples while using the same underlying Decision Transformer model. Results show that Genetic Algorithm Decision Transformer outperforms its counterpart Decision Transformer algorithm by a magnitude of up to 3.3 times the reward (see table 5.6). However, improvements to data collection could improve the model further.

Contents

Acknowledgements ii

Abstract iii

List of Figures vii

List of Tables viii

1 Introduction 1
- 1.1 Motivation 2
- 1.2 Thesis Outline 2
  - 1.2.1 Goals 3
  - 1.2.2 Hypotheses 3
  - 1.2.3 Document Outline 3

2 Background 4
- 2.1 Artificial Neural Networks 4
  - 2.1.1 Forward pass 4
  - 2.1.2 Backpropagation 5
- 2.2 The Transformer 5
  - 2.2.1 Attention 6
  - 2.2.2 Architecture 6
- 2.3 Generative Pre-trained Transformer (GPT) 8
  - 2.3.1 GPT-2 8
- 2.4 Reinforcement Learning 8
  - 2.4.1 Markov Decision Process 8
  - 2.4.2 Offline & Online learning 9
  - 2.4.3 Q-learning 10
  - 2.4.4 Deep Q networks (DQN) 10
  - 2.4.5 Double Deep Q-Network (DDQN) 10
- 2.5 Real-Time Strategy Games 11
  - 2.5.1 DeepRTS 12

3 State of the Art 13
- 3.1 Real-time Strategy Game Environments 13
  - 3.1.1 MicroRTS 13
  - 3.1.2 StarCraft II Learning Environment 14
  - 3.1.3 TorchCraft 14
  - 3.1.4 DeepRTS 14
- 3.2 Machine Learning in Video Games 15
  - 3.2.1 Chess & DeepBlue 15
  - 3.2.2 Go & AlphaGo 16
  - 3.2.3 Starcraft II & AlphaStar 16

3.3 Transformers in Reinforcement Learning 17
3.3.1 Representation learning 17
3.3.2 Sequential decision-making 19
3.3.3 World Model learning 20
3.3.4 Generalist agents 21
3.3.5 Comparison of Deep Reinforcement Learning Algorithms 21

4 Methods **23**
4.0.1 Overview 23
4.1 DeepRTS 24
4.1.1 Observation space 24
4.1.2 Action Space 25
4.1.3 Game Logic 25
4.1.4 Improvements and Fixes 27
4.1.5 Environments 27
4.2 Reward functions 29
4.2.1 PVP Rewards 29
4.2.2 Harvest Reward 30
4.3 Implementation 31
4.3.1 Gym implementation 31
4.3.2 Agent implementation 34
4.4 Datasets 34
4.5 Baseline 35
4.5.1 Double Deep Q-Network 35
4.6 Transformers 35
4.6.1 Decision Transformer 36
4.6.2 Genetic Algorithm Decision Transformer 36

5 Results & Discussion **39**
5.1 Results 39
5.1.1 Experiment 0: Baselines 39
5.1.2 Experiment 1: Training DT on various data 40
5.1.3 Experiment 2: Improving results by increasing sequence length 41
5.1.4 Experiment 3: GADT 41
5.1.5 Win rate in PVP environments 43
5.1.6 Harvest rate 44
5.1.7 Summary 44
5.2 Discussion 45
5.2.1 Genetic Algorithm Decision Transformer results 45
5.2.2 Data Dependency 45
5.2.3 Decision Transformer scalability 45
5.2.4 Using DeepRTS for reinforcement learning research 45
5.2.5 Hypothesis Review 46
5.3 Future work 46
5.3.1 Data quality 47
5.3.2 Data gathering 47
5.3.3 Increasing Action Space 47
5.3.4 Multi-Agent 47
5.3.5 Combining Agents 48

6 Conclusions **49**

A Appendices **50**
A.1 Implementation . . . 50
A.2 DeepRTS Configurations . . . 50
A.2.1 *Full1v1* . . . 51
A.2.2 *Harvest* . . . 51
A.3 Training Configuration . . . 51
A.3.1 DDQN . . . 51
A.3.2 Vanilla DT . . . 52
A.3.3 GADT . . . 52

Bibliography **54**

List of Figures

2.1 ANN vizualized. 4
2.2 Attention visualized. 6
2.3 The Transformer architecure. Encoder on the left, Decoder on the right. [80] 7
2.4 **Left:** GPT1 model. **Right:** GPT2 model. [1] 8
2.5 The feedback loop in an MDP, where an agent acts in an environment and collects data to learn the best actions for a given state [64]. 9
2.6 Bellman equation. The equation for updating $Q[s, \alpha]$ in the table. [70] . . . 10
2.7 The user interface for the RTS game Starcraft [27]. 11
2.8 Different maps available in DeepRTS [5]. 12

3.1 MicroRTS UI [36]. 13
3.2 Example observations of the PySC2 environment. Left side is a simplified graphical rendering of StarCraft II, while the right side shows the feature layers used as input for AI algorithms [83]. 14
3.3 Chess [20] 15
3.4 A game of Go [30] 16
3.5 AlphaStar vs "TLO" [76] 17
3.6 Architectures using transformers. [48] 17
3.7 AlphaStar model. [82] 18
3.8 The AlphaStar League. [4] 19

4.1 Overview of the DeepRTS DT implementation, showing how an agent predicts its actions. 24
4.2 3-D matrix with DeepRTS game data, expanded on [7] in accordance with the DeepRTS source code [6]. mXn is the size of the map. 25
4.3 An example map screenshot showing different units in DeepRTS. 26
4.4 The game state array in a Python debugger. Note the min and max values, showing that the array contains undefined values. 27
4.5 The first frame of a DeepRTS game. 29
4.6 Overview of implementation. 32
4.7 The architecture for the DDQN model used in this work 35
4.8 DT architecture [19] 36

5.1 Losses for training DT in Simple1v1 using Random dataset with various sequence lengths. 41
5.2 Reward for training DT in Simple1v1 using Random dataset with various sequence lengths. 41
5.3 Simple1v1 plots 42
5.4 Random1v1 plots 42
5.5 Full1v1 plots 43
5.6 Harvest plots 43

List of Tables

3.1 Maximum frames per second for MicroRTS, PySC2, TorchCraft and DeepRTS [7]. . . . 15
3.2 Results as gathered reward for IRIS [50] and DreamerV3 [33] in a range of Atari games. Best results for an environment are marked in **bold**. . . . 21
3.3 Average rewards for different Transformer based algorithms, comparing the highest reported result for each environment. Best averages for a given environment are presented in **bold**. . . . 22

5.1 Average reward and deviation for the random agent, evaluated for 1000 episodes across three seeds. . . . 39
5.2 Average reward and deviation for the DDQN agent, evaluated for 1000 episodes across three seeds. . . . 40
5.3 Highest average reward and deviation for DT in DeepRTS, using three different datasets. Models were evaluated for 1000 epochs across three seeds. The highest reward for a given environment is marked with **bold**. . . . 40
5.4 Win rate for our own models versus [65], across different environments. Results are not directly comparable, as different environments with different objectives were used. . . . 44
5.5 Harvest rate for DDQN, DT and GADT. Results were gathered from 1000 episodes across three seeds. Highest value is marked in **bold**. . . . 44
5.6 Highest reward in each of the four environments, regardless of dataset. Best results are marked in **bold**. . . . 45

A.1 Default DeepRTS config. Each environment uses these by default, unless specified otherwise. . . . 50
A.2 Config for *Full1v1*. Any options that are not specified is set to default, as shown in table A.1. . . . 51
A.3 Config for *Harvest*. Any options that are not specified are set to default, as shown in table A.1. . . . 51
A.4 Default hyperparameters for DDQN. . . . 51
A.5 Default hyperparameters for Vanilla DT trained on a dataset. . . . 52
A.6 Default hyperparameters for GADT. . . . 52

List of Listings

1 Generic Gym Inteface, and the *Simple1v1* class. Some code was omitted for simplicity. 33
2 Generic Agent Training loop using Gym. Some code has been omitted for simplicity. 34
3 DT Training loop. Some code has been omitted for simplicity. 36

Chapter 1

Introduction

The Transformer model has accelerated Artificial Intelligence (AI) research in a big way. Since the model was introduced in 2017 by Google Brain [81], as an attention-based model handling sequential data and outperforming the previous state-of-the-art, Transformers have become a de facto model for natural language processing (NLP) and computer vision. Models like Generative Pre-trained Transformer (GPT) iterations have become state-of-the-art when it comes to tasks like text generation and question/answering [60] [62] [16] [54]. Transformers consider sequential data of any type, and while GPT considers sequential text data, researchers have also applied it to sequential image data in various Reinforcement Learning (RL) environments [19] [39] [68] [50] [86].

RL considers an environment with observations, actions, and rewards and an agent that is iteratively trained to act in an environment according to a certain policy. RL algorithms like Q-learning handle data as independent of each other, predicting the best action for each given observation based on previous experience using a value estimation function. However, standard RL algorithms do not encode sequential information for observations. To enable this, the experience can also be collected and stored sequentially as labeled data, which can turn RL into a sequence modeling problem similar to NLP, predicting the next token given a sequence of data. This means RL can benefit from using large-scale datasets and state-of-the-art Transformer models by using data generated by human players or policy-driven agents to learn the best actions given an input sequence.

Large datasets are easily accessible for popular RL environments with popular research environments such as the D4RL dataset [28] for OpenAI Gym environments or Atari datasets [10] for a wide range of Atari games. However, this data dependency makes it difficult to apply Transformers to RL environments without high-quality datasets. One solution to this problem is to generate data from random agents, or recording play by humans or pretrained agents. However, random data has poor quality due to noise, and is generally unable to generate data sequences with high total reward similar to a policy-driven player. In addition, recording human data and data from pretrained agents is inefficient as it requires human resources and computational power.

This thesis explores how Transformers can be used in a limited RL environment, even without large high quality datasets. To this end, the Decision Transformer (DT) model proposed by Chen et al. [19] was chosen for its ability to learn even from suboptimal data. DT predicts the following action from a sequence of observations, rewards-to-go and actions using an autoregressive attention mechanism. The RL environment chosen for this work is DeepRTS, a real-time strategy (RTS) RL environment for deep learning [7]. DeepRTS is a new environment for RL and has not been extensively used in research, but [65] and [9] have trained RL agents with different algorithms to play DeepRTS. However, there is no publically available dataset for DeepRTS.

This thesis proposes Genetic Algorithm Decision Transformer (GADT), a novel implementation of DT by Chen et al. [19] that does not rely on a static dataset for training. GADT generates its own data by using the epsilon-greedy strategy to explore the environment and selecting the best sequences based on reward through the genetic algorithm to generate high-quality training data. This data is then fed into a standard DT model to learn from gathered experience. As the model improves, the dataset increases in size. To evaluate the performance of GADT, this work also use the standard DT algorithm and implement a Q-learning algorithm for various tasks in DeepRTS. DT algorithm was trained on random and data from a Q-learning agent to see how it performs with varying data quality. Experiments shows that GADT on average outperforms DT and Q-learning algorithms on tasks in DeepRTS, and the DT architecture is scalable as it can only improve with higher-quality data.

1.1 Motivation

RL can create a virtual environment that an agent can learn to interact in according to a policy. Hence, we can compare the process to human learning, similar to how a baby would learn to take its first steps; The baby observes their environment through its cognitive senses and chooses actions to attempt walking. Starting from zero experience, it learns from trial-and-error to find the best actions to maintain balance and speed when walking. This learning process is interesting to analyze and simulate through virtual environments, because of the gradual improvement as experience in an environment increases, how the walking objective is achieved and what issues are faced when exploring new environments. Furthermore, by simulating these scenarios in RL environments we can explore how to design environments and provide signals to agents to optimize behaviour and speed of learning.

Applications of RL can be simulations of real-world environments. Having a virtual model of a real-world environment means we can train an agent to act in it without building physical prototypes and extensive testing in the real world. Therefore, the nature of RL agents allows them to keep learning in real-world environments, as it is exposed to new states while recording their rewards and actions. This makes RL an interesting and important area of AI research. Furthermore, the performance improvements that transformers bring in tasks like NLP and time-series prediction make it interesting to apply it to RL, an area where there is significantly less research in regards to transformers. Because of the limited research available in this regard, this thesis will be a contribution to the research of a new deep learning method in a new environment. However, state-of-the-art Transformer models like GPT, LLaMA are trained over multiple weeks using large clusters of enterprise-level hardware and terabytes of data [45] [78]. For RL, Models like IRIS [50], AlphaStar [3] and DreamerV3 are also trained on expensive hardware and over long periods of time, where DreamerV3 citing up to 16 GPU days for Atari benchmarks [33, p. 18]. This thesis will provide research on proving decent performance for Transformers using cheaper hardware and limited resources. This thesis also considers the RTS game DeepRTS. RTS games are complex environments for RL because it involves strategic planning, multiple controllable units, and players. The terminal state of the game is not pre-defined, and players must react to opposing players' actions. Still, the DeepRTS has a small discrete actions space and a configurable environment, making it simple to define experiments and custom learning environments and tasks.

1.2 Thesis Outline

The main goal of this thesis is to evaluate and improve the performance of Transformer models in a RTS game environment, namely DeepRTS. Transformer algorithms will be compared to Q-learning algorithms to measure improvements.

1.2.1 Goals

- **Goal 1:** Research and present the state-of-the-art in RL methods combined with transformers
- **Goal 2:** Research and present the state-of-the-art for Real-time strategy games in RL, and compare them to DeepRTS
- **Goal 3:** Train a Deep Q-learning agent to play DeepRTS
- **Goal 4:** Train a Decision Transformer agent to play DeepRTS
- **Goal 5:** Evaluate the performance of the Decision Transformer and Genetic Algorithm Decision Transformer agent and compare with other RL algorithms
- **Goal 6:** Present how Genetic Algorithm Decision Transformer improves data collection and improve on Decision Transformer performance
- **Goal 7:** Evaluate DeepRTS as an RL environment and propose changes to improve use in RL research

1.2.2 Hypotheses

- A Decision Transformer agent will outperform a Deep Q-learning agent in terms of total reward in DeepRTS
- A Decision Transformer agent will learn the best sequences of actions from a dataset with mixed policies for action generation, and outperform a Deep Q-learning agent in terms of cumulative reward in DeepRTS
- Training a Decision Transformer using only randomly generated actions will outperform a random agent in terms of reward
- Gathering game data using the Genetic Algorithm and using the epsilon-greedy strategy will improve on Decision Transformer and allow an agent to explore the environment and avoid dependencies on labeled data
- Decision Transformer will be dependent on sequence length for performance during training

1.2.3 Document Outline

The following thesis will start by introducing required background knowledge in Chapter 2, presenting neural networks, the Transformer model and RL. The RL environment DeepRTS is also presented in this chapter before Chapter 3, where the state-of-the-art RL environments are showcased, offering a comparative analysis and an introduction to the history of Transformers in the RL domain. This chapter also presents state-of-the-art Transformer models in general and models applied to RL problems. Chapter 4 presents the implementation and methods of this thesis, presenting various tasks in DeepRTS that RL agents are tasked to explore and act in, various datasets that were used to train DT agents and the details of the novel implementation GADT. The results of these are presented in Chapter 5, followed by a discussion of the outcome and what has been learned from the experiments. Finally, Chapter 6 concludes the thesis with a summary and reflection on the findings. [1]

[1] We have used many tools to support our writing in this such as Grammarly and ChatGPT. All the writing in this thesis is still our own, but the tools have given us grammar checks, suggestions, and inspiration.

Chapter 2

Background

This chapter presents the research domains for this thesis, namely AI, RL and RTS games. First, the transformer architecture is explained, starting from the basics of artificial neural networks and building up to the attention mechanism of a transformer. The chapter then introduces GPT, the most prominent implementation of a transformer, which is used in this thesis through the DT algorithm. Furthermore, the domain of RL and standard RL algorithms are presented. RTS games and DeepRTS is also introduced as the RL environment for this thesis.

2.1 Artificial Neural Networks

Artificial neural networks (ANNs) are machine learning algorithms inspired by the structure and function of biological neurons in the human brain. ANNs are composed of interconnected neurons organized into layers that process and analyze data to generate an output. Each neuron represent a feature of a given data sample. Each neuron has a "bias" value, and a number of "weight" values. The number of weights in a neuron is equal to the number of connections going into the neuron. In the figure 2.1 a simple ANN is shown. Interconnected neurons between layers indicates that a neuron in the first layer says something about the connected neuron in the second layer. Outputs often produce predictions to classify a data sample by the output of many interconnected neurons.

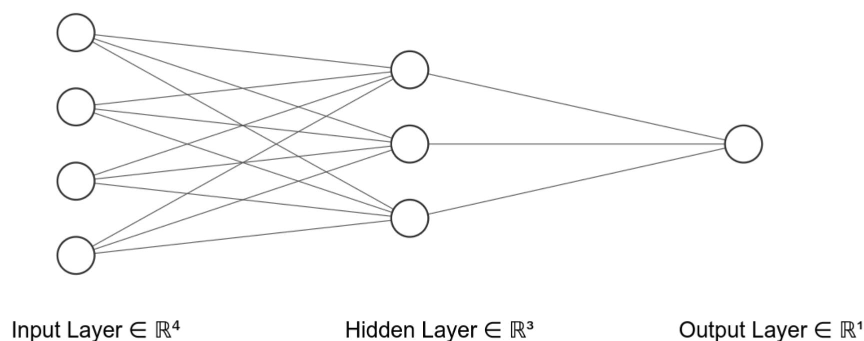


Figure 2.1: ANN vizualized.

2.1.1 Forward pass

In an ANN a "forward pass" is the action of calculating the output given a certain input. Each neuron in an ANN receives input from one or more other neurons and applies a mathematical function to the input to produce an output. The outputs from one layer of neurons become inputs to the next layer, and this process continues until the final layer produces the desired output.

Given an input into a layer, the outputs z_j of the layer is calculated by the following formula.

$$
\begin{aligned}
z_j &= \sum_{i=1}^{m} w_{ji} x_i + b_j, \quad \text{for } j = 1, \ldots, n \\
n &= \text{Number neurons in layer} \\
m &= \text{Number of elements in input sent into layer} \\
x &= \text{Inputs into layer} \\
w &= \text{Weights of the neuron} \\
b &= \text{Bias value of the neuron}
\end{aligned}
$$

2.1.2 Backpropagation

Backpropagation is a widely used algorithm for training ANNs. The goal of backpropagation is to adjust the weights of the connections between neurons in an ANN so that the output produced by the network is as close as possible to the desired output [63].
Backpropagation works by calculating the difference of output of the output layer against the wanted result, this difference is called the loss. Now using this loss the weights and biases in a given layer can be updated with the following function:

$$
w_{ij} = w_{ij} - \eta \frac{\partial \mathcal{L}}{\partial w_{ij}}, \quad \text{for } i = 1, \ldots, n \quad \text{and } j = 1, \ldots, m
$$

$$
\begin{aligned}
b_i &= b_i - \eta \frac{\partial \mathcal{L}}{\partial b_i}, \quad \text{for } i = 1, \ldots, n \\
\eta &= \text{Learning rate} \\
\mathcal{L} &= \text{Loss}
\end{aligned}
$$

The Mean Squared Error (MSE) is a popular loss function, which works by squaring the difference between the true value and the predicted value, $(truth - prediction)^2$. This has the effect of placing more emphasis on larger errors, making the model adjust its weights to minimize the loss to make predictions closer to match the truth. In other words, the MSE penalizes larger deviations from the desired outcome. The full formula for MSE is:

$$
\begin{aligned}
MSE &= \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \\
\hat{y}_i &= \text{Prediction} \\
y_i &= \text{Truth} \\
n &= \text{Number of datapoints}
\end{aligned}
$$

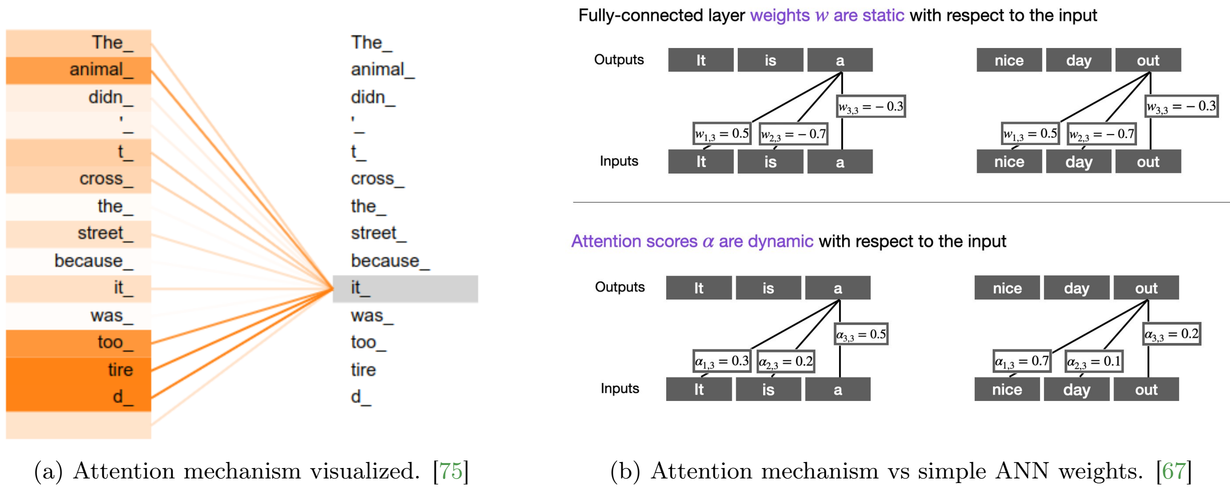
This formula will update the weights and biases of the last layer. Then it can be used again to update the previous layer and so on. It is called backpropagation because the weights calculations and updates moves backwards in the neural network for each neuron.

2.2 The Transformer

The Transformer architecture has became very popular lately with the release of large language model's (LLM) such as GPT [37], and LLaMA [38]. This architecture was initially published in 2017 by the name "Attention Is All You Need" [80], where the authors present a method of forming dynamic connections between the inputs using the attention mechanism.

2.2.1 Attention

The attention mechanism is a technique used in deep learning that enables the model to focus on specific parts of the input data when making predictions. The idea behind attention is to allow the model to dynamically weight the importance of different parts of the input. A simple illustration of this can be seen in Figure 2.2a, where the attention mechanism calculates that there is a strong connection between the word "it" and the words "animal", "too", "tired". This can be understood as when the model is creating a sentence, if the previous words are "animal" there is a high chance the word "it" will be used. This mechanism is akin to human perception, where humans can understand a sentence or image by focusing on certain features.



(a) Attention mechanism visualized. [75]

(b) Attention mechanism vs simple ANN weights. [67]

Figure 2.2: Attention visualized.

The attention mechanism starts by sending the input $x = (x_0, x_1, x_2, ..., x_T)$ through three different ANNs as shown in figure 2.1. The output dimension of these ANNs is usually called *n_embed* and can vary in size. These three outputs are called Key (K), Value (V), Query (Q). The formula [80] to calculate the attention of an input can be seen below.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V, \quad \text{where}$$

$$\text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^{N} \exp(x_j)}$$

$$d_k = \boldsymbol{K} \in \mathbb{R}^{d_k}$$

It is important to note that attention scores differ from simple ANN weights of as they depend on the input. This is shown in figure 2.2b.

2.2.2 Architecture

In the original paper [80], the Transformer model consists of an encoder and a decoder, each of which is composed of multiple layers. The encoder processes the input sequence and creates a contextualized representation of each token in the sequence. The decoder generates the output sequence based on the contextualized representation and a previously generated output.

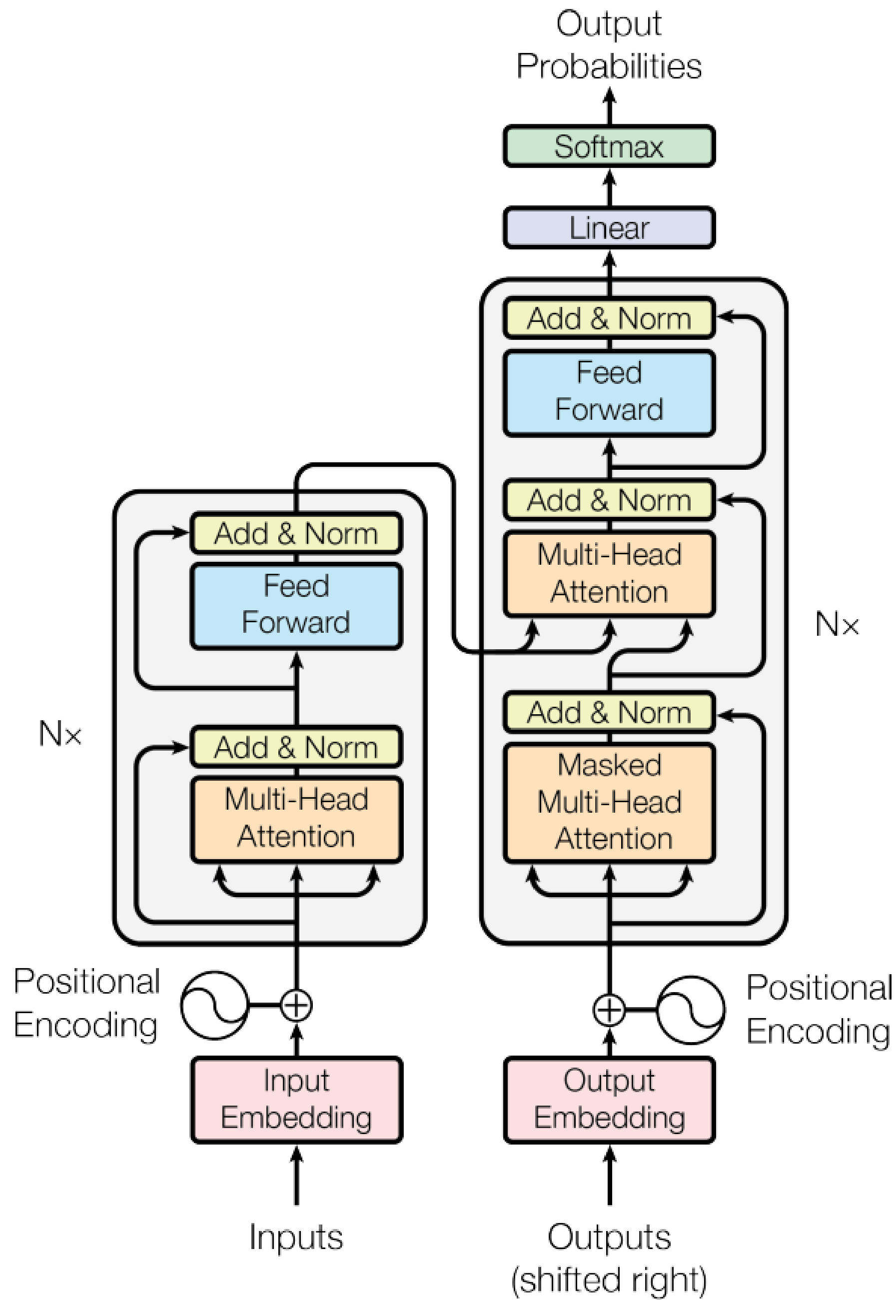


Figure 2.3: The Transformer architecure. Encoder on the left, Decoder on the right. [80]

Each layer in the Transformer model consists of an attention mechanism followed by a feed-forward neural network. The attention mechanism allows the model to weigh the importance of different parts of the input sequence, while the feed-forward neural network applies a non-linear transformation to the input. This architecture differs from recurrent neural networks and long short-term memory [35] by encoding information about all the tokens in a given input sequence, rather than concatenating the previous input to the next.

The Transformer architecture has achieved state-of-the-art performance on a wide range of NLP problems including machine translation, language modeling and question-answering. With the introduction of the original Transformer model by Vaswani et al. [80] outperformed previous state-of-the-art models like MoE [69], ConvS2S [29] and GNMT [85] [80, p. 8].

2.3 Generative Pre-trained Transformer (GPT)

GPT is a LLM created by OpenAI [17] that is based on the Transformer model. The biggest change from the transformer model to GPT is the removal of the encoder, this means GPT uses only a decoder, this can be seen in figure 2.4.

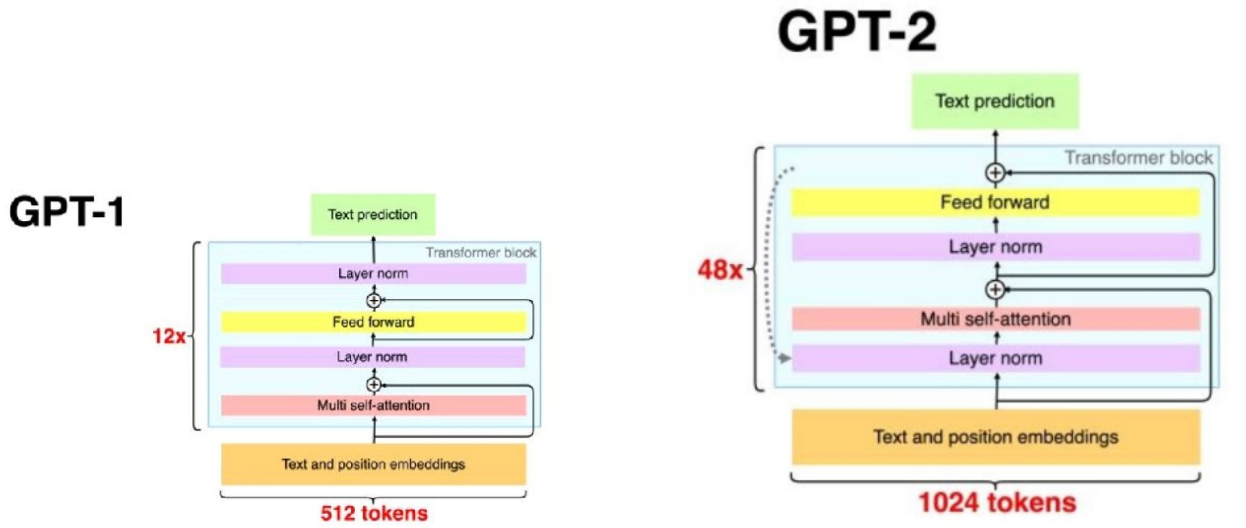


Figure 2.4: **Left:** GPT1 model. **Right:** GPT2 model. [1]

The model is called Generative pre-trained transformer because it is first trained on simply predicting the next word in a dataset. This is called pre-training, after this, the model is fine-tuned using supervised fine-tuning where the wanted output for a given input is known. The pre-training works by maximizing the sum of the log likelihood of the next wanted word x_t, given previous words $x_{t-k}, ..., x_{t-1}$. This can be written as follows.

$$\mathcal{L}(x) = \sum_{t=1}^{T} \log P(x_t | x_{t-k}, ..., x_{t-1})$$

2.3.1 GPT-2

GPT-2 is the successor of GPT [61]. GPT-2 as seen in figure 2.4 can take a larger input. It also is much bigger with 48 layers compared to 12, and lastly moved the layer normalization layers around when compared to GPT. GPT-2 is also the backbone of the Decision Transformer model, which will be explained in section 3.3.2.

2.4 Reinforcement Learning

RL considers an agent acting in an environment, and how the state of the environment changes based on the agent's actions. The purpose of RL is to provide a framework for an agent to learn the optimal behaviour in an environment, starting from zero experience and learning by trial and error. The optimal behaviour is encoded in a reward function, which acts as a scale of how close the agent is to the optimum given its actions. Hence, the goal of the agent is to maximize this reward function by exploring the environment and learning a policy for which actions gives the highest reward for a given state.

2.4.1 Markov Decision Process

RL can be described as a Markov Decision Process (MPD), presenting the process as an environment E, an agent A and a tuple (S, A, P, R) [59]:

- S: A finite set of states for an environment
- A: A finite set of discrete or continuous actions an agent can perform in an environment
- P: $P(s_{t+1} = s'|s_t, a_t)$, a function describing the probability of transition from states s to s' given an action a at timestep t.
- R: $R(s, a, s')$, a reward function which outputs the reward for transitioning between state s and s' given action a and states s, s'.

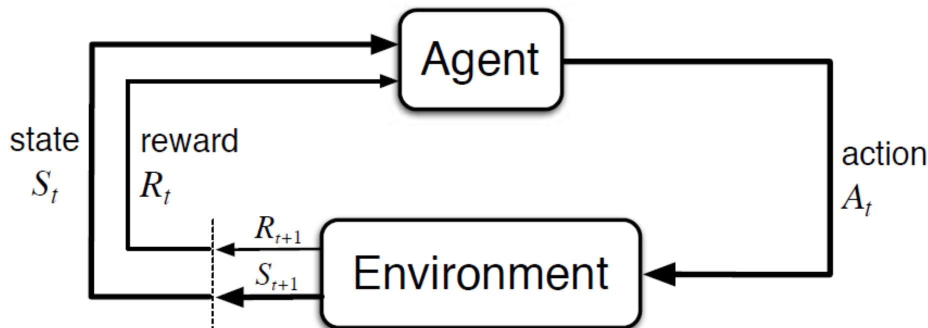


Figure 2.5: The feedback loop in an MDP, where an agent acts in an environment and collects data to learn the best actions for a given state [64].

To approximate an optimal policy, the agent has to perform actions that maximize the reward function. The agent can pick an action $a \in A$ every timestep t, where the action can be responsible for changing the environment. The agent can then record the state transitions and rewards from each action, and use this data to learn which actions get the highest reward. Based on the observed data, the agent can learn to approximate an optimal policy π by learning to predict rewards for (a, s) pairs, using a function $Q(s, a)$. Different methods exist to define this function, such as value iteration [11], policy iteration [12] and Q-learning. The latter is explained in section 2.4.3.

2.4.2 Offline & Online learning

There are two main categories of learning methods in RL: offline and online learning. In offline learning, the agent learns from a fixed dataset of state transitions and rewards. This method is suitable when there is a dataset available for a given environment, and the agent can learn from this data without directly interacting with the environment. This principle can turn RL into a supervised training process, where a ANN learns to predict an action given observations and rewards.

In contrast, online learning involves the agent interacting with the environment and learning from its own experience. In this approach, the agent learns and updates its policy based on the real-time feedback it receives from the environment. Online learning is often preferred as it allows the agent to adapt to changes in the environment. In this way, agents can also learn from their own mistakes and explore different actions by using exploration to discover new and better policies. This is normally done with the epsilon-greedy strategy, which alters between selecting actions from a policy π and using randomness, where e decreases over time to reduce randomness:

$$a = \begin{cases} \pi(a|s), & \text{if } rand() < \epsilon \\ randint(0, |\mathcal{A}|), & \text{otherwise} \end{cases}$$

2.4.3 Q-learning

Q-learning is an algorithm used to find an optimal policy in a RL environment where a reward function exists. The Q in Q-learning stands for "Quality", and the objective is to find the quality of each possible future action and choose the one with the highest quality. The Q-learning algorithm starts by making a table with $Q[s, \alpha]$ values. This means for a given state s and a given action α, what is the Q value. At the start, this table will be empty so the agent must explore the environment by making random actions using the epsilon-greedy strategy. The Q values in the table are calculated using the formula in figure 2.6. The Q-values in the table are updated over-time, and as the randomness decreases, the Q-values get more accurate, in finding the highest reward path.

Figure 2.6: Bellman equation. The equation for updating $Q[s, \alpha]$ in the table. [70]

2.4.4 Deep Q networks (DQN)

In Deep Q networks (DQN) the idea is to combine ANN's with Q-learning to learn the optimal strategy. In this algorithm, instead of using a table with Q-values, an ANN is used where the input is the s and the outputs are $Q[s, \alpha_i]$, for i = 0, ..., n where n is the number of actions. Then the action which has the highest Q value can be chosen. The network is trained by using "experience replay", in which a table of previous experiences is created, the table entries are as shown below.

$$(s_t, s_{t+1}, a_t, r_t)$$

This table can now be used to train the ANN with more efficiency by batching multiple entries. The model can now use the following loss function to update its weights.

$$\mathcal{L} = (Q^*(s_t, a_t) - Q(s_t, a_t))^2$$

$$Q^*(s_t, a_t) = r + \gamma \max_{a'} Q(s_{t+1}, a')$$

2.4.5 Double Deep Q-Network (DDQN)

Double Deep Q-Network (DDQN) improves on DQN 2.4.4 by using two ANN's. The reason for this change can be explained by looking at the loss function of DQN.

$$\mathcal{L} = (Q^*(s_t, a_t) - Q(s_t, a_t))^2$$

$$Q^*(s_t, a_t) = r + \gamma \max_{a'} Q(s_{t+1}, a')$$

In this loss function $Q^*(s_t, a_t)$ always takes the $\max_{a'} Q(s_{t+1}, a')$ such that the network will always overestimate the next Q value. This problem builds upon itself as the network will

not use the overestimated Q value in future predictions where it will again overestimate Q values.
DDQN uses two ANN's, called "target" and "online". Where the online model is used for predicting the Q value, and the target model is used to predict the next action. The target model copies its weights from the online model every n steps. The loss function is now as follows.

$$\mathcal{L} = (Q^*(s_t, a_t) - Q_{online}(s_t, a_t))^2$$

$$Q^*(s_t, a_t) = r + \gamma Q_{target}(s_{t+1}, \arg\max_{a_{t+1}} Q_{online}(s_{t+1}, a_{t+1}))$$

Now the max Q value is not always taken as the Q_{target} network may possibly have higher Q values for other actions, but it gives the Q value for the predicted action of the online model.

2.5 Real-Time Strategy Games

A Real-Time Strategy (RTS) game is a game environment where the players can act simultaneously. Because of the even playing field, RTS games require players to react to each other's actions and constantly try to capitalize on other players' mistakes. This means these games require strategic planning, which is why RTS games like Starcraft, Warcraft and DeepRTS are themed around military combat [13] [14] [7]. The RTS game genre is vastly popular worldwide, with games like DOTA 2 and League of Legends having millions of players every day [72] [58].

Figure 2.7 shows the RTS game Starcraft, presenting most of the important elements of an RTS game; multiple controllable units with different abilities, buildings that require resources to complete, and the battlefield where players move their units to attack their opponent.

Figure 2.7: The user interface for the RTS game Starcraft [27].

2.5.1 DeepRTS

DeepRTS is a RTS game made specifically for RL research [7]. While it can be played by human players in a graphical user interface, the main purpose of the game is high performance while running the game without graphics with RL agents performing the actions. As such, the game can up to output 7 million timesteps per second during simulation. Figure 2.8 shows some of the different maps of DeepRTS, containing two units, one for each player. The game can be played by multiple players, and each player starts with 1 movable unit. Units can move simultaneously with each other, and the goal of the game is to defeat the other player by attacking it and destroying all its units and buildings. Units can also gather the resources lumber, stone and gold, which can in turn be used to fund buildings. By building buildings the player can create more movable units, and the player can toggle control of these units to perform different tasks.

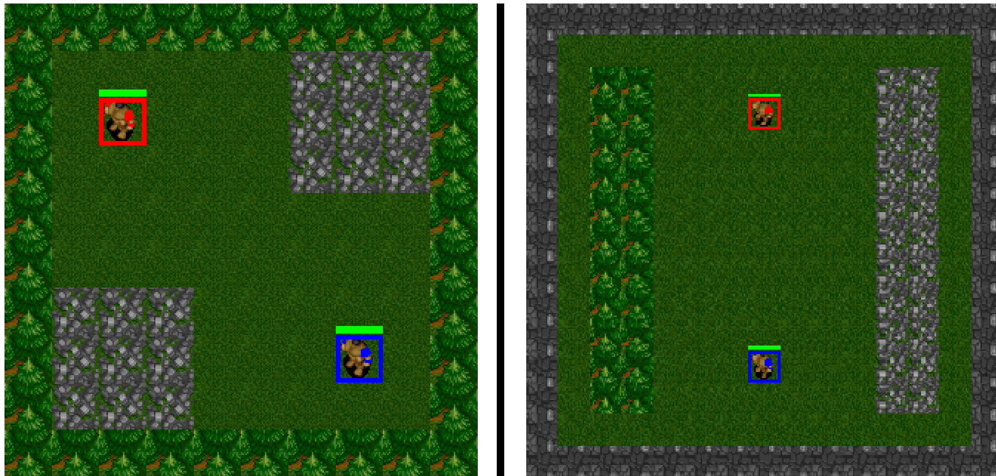


Figure 2.8: Different maps available in DeepRTS [5].

Chapter 3

State of the Art

This chapter presents the state-of-the-art within the domains of this thesis, namely RTS Environments for RL, and using Transformer models for RL.

3.1 Real-time Strategy Game Environments

RTS games challenge the player's ability to plan and strategize to get the upper hand of the opponent by exploiting their mistakes or miscalculations. This makes RTS games an interesting area of research in AI, to study how AI deals with strategy and long-term planning. Exploring RTS games can be done using RL. However, few RTS RL environments exist [8], and most of them are interfaces for high-resolution games that do not have high enough performance to be used in AI research. In addition, most popular RTS games are closed source, which makes it difficult to extract data and interact with the game through scripting. These games also include complex game logic, making RTS game development a difficult task. This section presents existing RTS environments for RL and how they compare to DeepRTS.

3.1.1 MicroRTS

MicroRTS is an RTS game made for AI research. Like DeepRTS, the game's purpose is high performance, stripping away complex game logic and high-resolution graphics, ensuring researchers can focus on AI [53]. The game is made using Java, but can also be interacted with through Python using the OpenAI Gym API [36]. There is also an annual AI competition for MicroRTS, where entrants submit their bots to play against each other in a tournament to find the best-performing AI model [52].

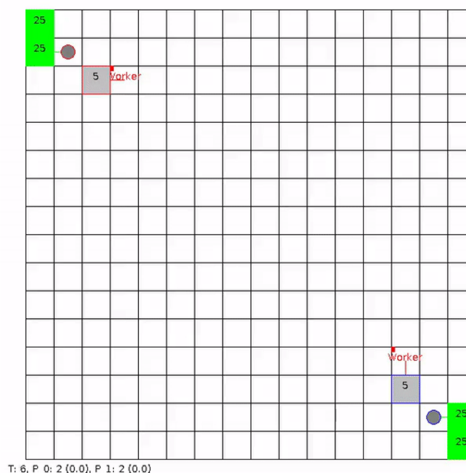


Figure 3.1: MicroRTS UI [36].

3.1.2 StarCraft II Learning Environment

StarCraft II Learning Environment (SC2LE) is a collaboration between StarCraft II developers Blizzard Entertainment and researchers at Google DeepMind to expose StarCraft II as a learning environment for AI research [25]. SC2LE targets various programming languages such as C++ through the CommandCenter project [21] and Python through PySC2. The Starcraft II base game has high resolution graphics, which in SC2LE gets encoded as feature layers to reduce observation space complexity. The large action space with approximately 10^8 combinations [83]. Testing by Vinals et al. and Andersen et al. has shown that SC2LE environment manages between 60 and 144 frames per second (FPS) [83] [7] during training. Taking these factors into account, makes StarCraft II a complex environment for RL research. Figure 3.2 shows an example of both human and machine-interpretable observations of the SC2LE through PySC2.

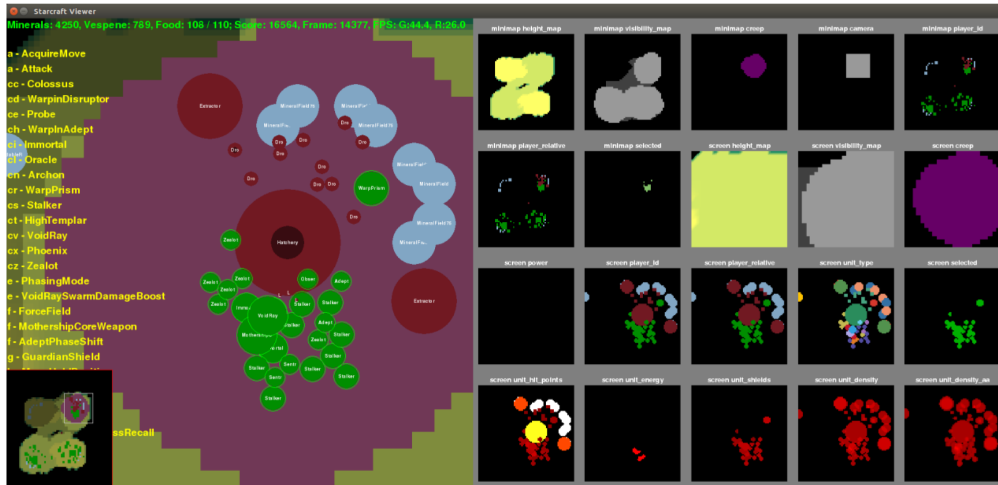


Figure 3.2: Example observations of the PySC2 environment. Left side is a simplified graphical rendering of StarCraft II, while the right side shows the feature layers used as input for AI algorithms [83].

3.1.3 TorchCraft

TorchCraft is an API between the StarCraft I game and the scientific computing framework Torch [74] [23]. TorchCraft extracts and receives StarCraft I data through a ZeroMQ (ZMQ) connection, enabling manipulation of StarCraft I through any programming language that supports the ZMQ protocol. TorchCraft has been used in RL research for years [57] [22], but as of July 2022 the project has been archived and is not in active development [2]. Lin et al. have also created StarData, which is a dataset with 65546 replays of StarCraft I for the purpose of AI research [49].

3.1.4 DeepRTS

The basics of DeepRTS is explained in section 2.5.1. Comparing DeepRTS to the state-of-the-art shows that DeepRTS outperforms other environments with its high performance. Table 3.1 shows performance across the previously stated RTS environments, where DeepRTS outperforms them all. In addition, DeepRTS has a fairly simplified action and observation space, with 16 discrete actions for simple unit movement and attacking, and 2 actions for targeting and issuing commands to units. DeepRTS is fully playable with just 16 actions, as including targeting and commanding units would increase the number of actions by $2*m*n$, where m and n is the maps width and height respectively. Furthermore, DeepRTS is in active development and the authors of this thesis were able to contribute to the project, as explained in section 4.1.4. While the DeepRTS game cannot compare to environments like StarCraft

II in terms of complex game logic, DeepRTS enables efficient computation and training of RL agents, and may perhaps be used as a pretraining tool before moving to more complex environments.

Environment	Frames per second
MicroRTS	11 500
PySC2	144
TorchCraft	2 500
DeepRTS	**7 000 000**

Table 3.1: Maximum frames per second for MicroRTS, PySC2, TorchCraft and DeepRTS [7].

3.2 Machine Learning in Video Games

The realm of gaming offers a challenging arena for Machine Learning (ML) researchers to test and compare their algorithms against human players. This provides a valuable means of evaluating the intelligence of ML algorithms, particularly within specific game environments. Over the past 25 years, ML has made remarkable advancements, progressively surpassing human capabilities in increasingly complex game scenarios.

3.2.1 Chess & DeepBlue

Chess, a well-known turn-based strategy game in the western world, serves as an exemplary domain for exploring the potential of ML. The game involves two players, each starting with 16 pieces that follow specific movement rules.

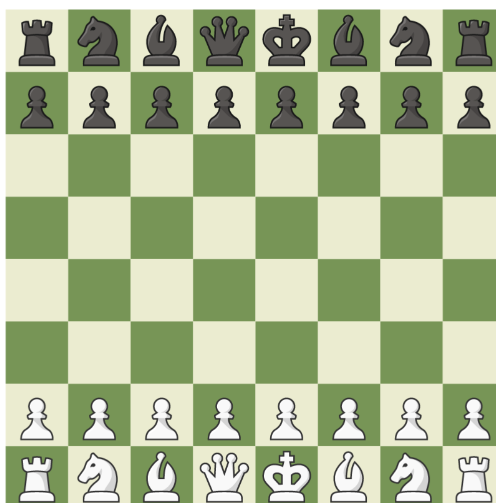


Figure 3.3: Chess [20]

With an estimated space complexity of approximately 10^{124} [71], chess boasts an astronomical number of possible game variations. To put this into perspective, a rough estimate of the number of atoms in the observable universe is 10^{80} [51]. Consequently, the game cannot be solved through brute force methods due to the immense computational requirements needed to evaluate all possible paths.
Various algorithms have been employed in attempts to defeat the world's top chess players. Notably, the first instance of a chess world champion losing to a computer occurred in 1997 during the historic match between IBM's Deep Blue and Garry Kasparov [18]. Deep Blue used a combination of hard-coded moves as well as an evaluation function to choose it's next move. The evaluation function tried to evaluate a certain game state based on hard-coded rules combined with weights which could be changed to adjust the value of the game

state. These weights were analyzed with automatic tools to fine-tune the evaluation function. This can be seen as a something similar to backpropagation in ANN's. Because Deep Blue used alot of hard-coded rules combined with a tree search algorithm aswell as a evaluation function, this meant that Deep Blue required alot of computation. To scale this up to the game Go (see next section) would be very hard, if not impossible.

3.2.2 Go & AlphaGo

Go is a turn-based strategy game which is often compared to chess, as its popularity in the east rivals the popularity of chess in the west. The space complexity of Go is a whooping 10^{360} possible game variations [71]. Because of this huge number, the method of hand-tuning and hard-coding rules would prove to be impossible for the game of Go.

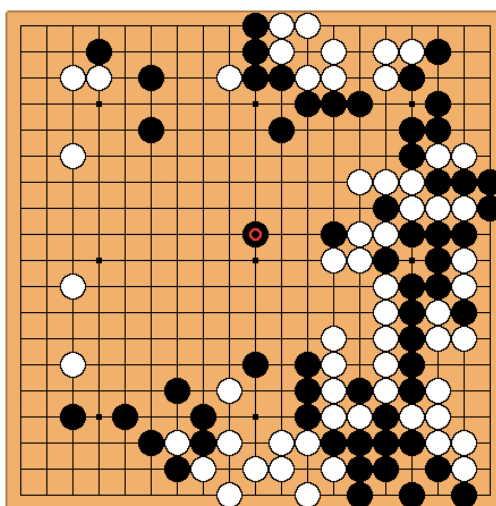


Figure 3.4: A game of Go [30]

In October 2015 for the first time ever, a computer won 5-0 against a professional player without any handicap [71]. AlphaGo (created by DeepMind) using an ANN algorithm with RL with a self play mechanism as well as learning from professional games. In 15 March 2016 AlphaGo won 4-1 against one of the best players in the world [31]. Making Go "solved" by AI in the sense that AI can match and be superior to the best humans in the Go domain.

3.2.3 Starcraft II & AlphaStar

After AlphaGo's success, DeepMind went after the RTS genre, more specifically StarCraft II (see section 3.1.2). DeepMind recognized StarCraft II as an environment that requires the management of multiple agents in real time, making it a potential representation of complex real-world problems. AlphaStar [82] is an advanced ML system created by DeepMind designed to play StarCraft II.

On December 19th [3] AlphaStar was the first computer to beat a professional StarCraft II player, the result was 5-0 in favor of AlphaStar against Team Liquid's Grzegorz "MaNa" Komincz which at the time was one of the best players in the world.

Figure 3.5: AlphaStar vs "TLO" [76]

AlphaStar had multiple handicaps to make for competitive/realistic games, such as an APM(actions per minute) limiter. A camera view limiter which only lets certain parts of the map be viewed by the agent at a given time,similarly to a human. Aswell as a 200ms request delay from choosen output of the action to the action being issued in the game.

3.3 Transformers in Reinforcement Learning

Transformer models have become the state-of-the-art when it comes to NLP, showing its scalability by learning from large datasets. Researchers seek to explore its performance in RL, by modeling environments as sequential data. Transformers have been experimented with in RL with different algorithms. In figure 3.6 the history of transformers in RL can be seen.

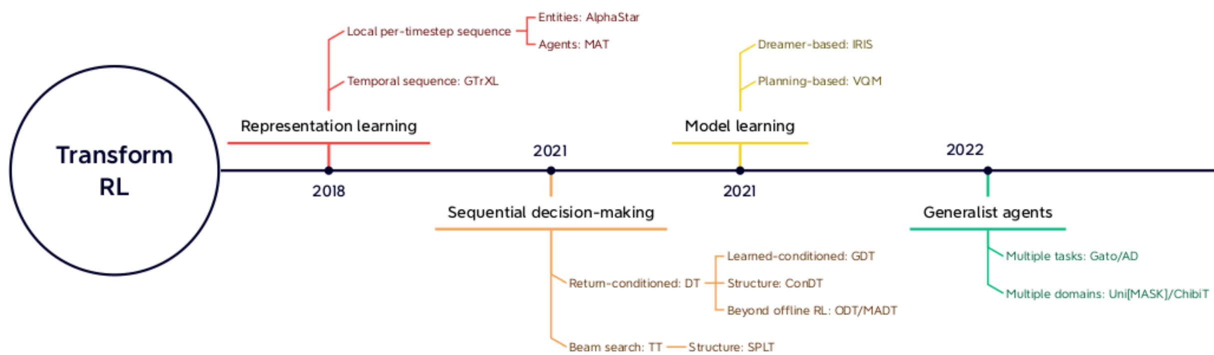


Figure 3.6: Architectures using transformers. [48]

This figure shows four different ideas implemented using transformers in RL, these ideas will be explained in the upcoming subsections.

3.3.1 Representation learning

In representation learning a model is used to create a representation of the observation. The idea is that the model can "understand" something about the environment such that it can pick what information is most necessary to keep in the representation. As seen in figure 3.6, this is most notably used by AlphaStar (see section 3.2.3).

AlphaStar

AlphaStar uses transformers as one of the many ML algorithms inside it's model. The full AlphaStar model can be seen in figure 3.7. In this model MLP's (Multilayer perceptrons) are used to connect between the different modules. But the core of the model is a Deep LSTM [35] (Long Short Term Memory) which takes input from: an Resnet [34] which encodes the visual information, a Transformer which encodes unit information, and an MLP which encodes information about units selected, and other simple values.

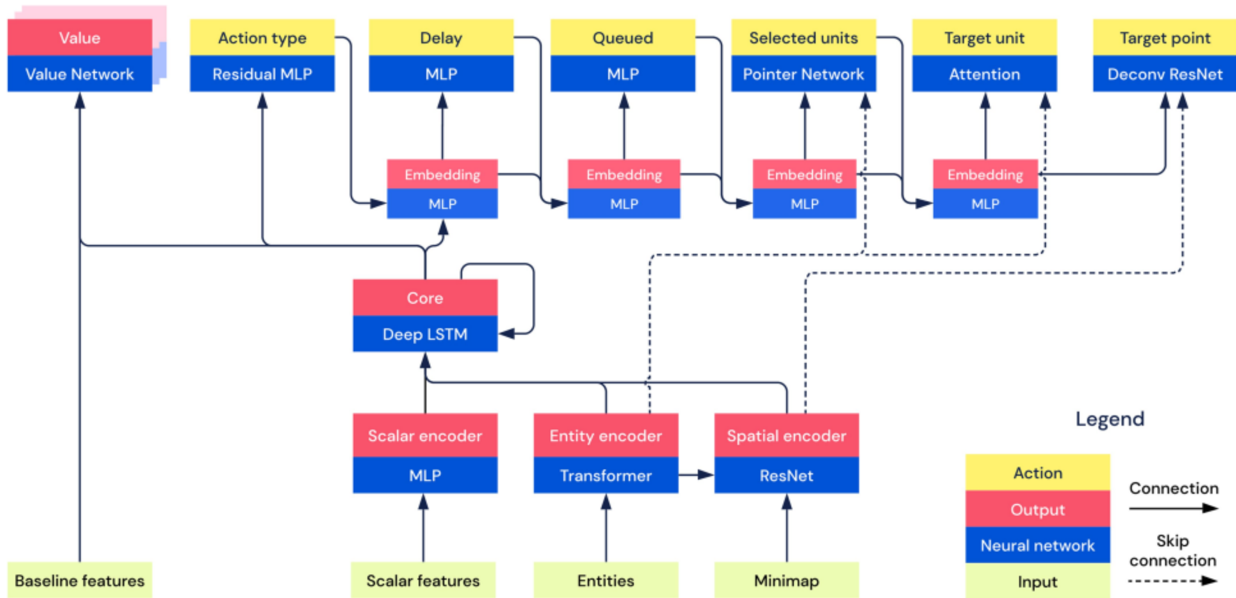


Figure 3.7: AlphaStar model. [82]

The transformer in this model is used to encode all Enitie data. Entitie data is the information about a unit. In StarCraft a unit can be used to attack enemy units, harvesting resourses and building, because of this a unit has alot of information which must be processed. This transformer takes in the following entitie data(for each entitie up to a maximum of 512 entities):

- Unit Type
- Owner
- Status
- Display type
- Cooldowns
- Attributes
- Unit atributtes
- Cargo
- Building status
- Resource status
- Order status
- Buff status

Because of the huge amount of data, a huge model is used containing a total of 139 million weights [82]. Given this huge amount of weights, the model requires a lot of training. AlphaStar solved this by creating the "League". The League is a system where agents with different variables such as APM for example, or agents which trained on different human player's data can play against each other. The League can be seen in figure 3.8. On the left an agent learns from some human data, then the agent plays against another agent chosen by an algorithm. Both agents now can learn from playing against each other which will then update their weights using RL and new agents will be created. The League also "Freezes" agents (in blue), this is to keep diversity in the League. After a certain amount of iterations, a model can be chosen by choosing the one which beats most of the other players in the League.

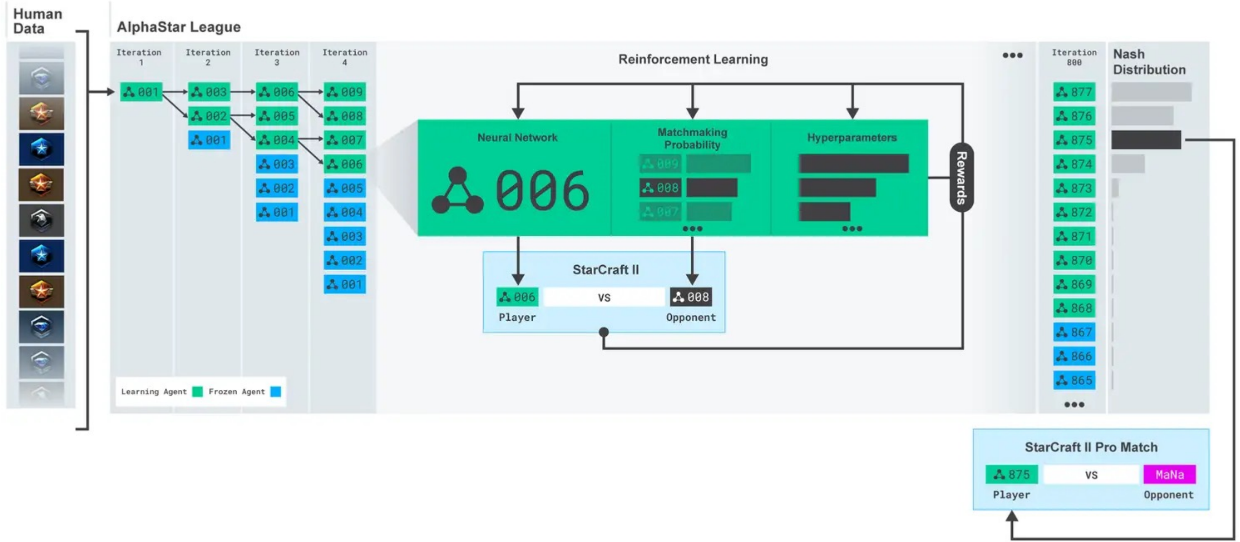


Figure 3.8: The AlphaStar League. [4]

3.3.2 Sequential decision-making

A RL environment can be thought of as a sequence by rewriting it as:

$$\tau = (s_1, a_1, r_1, s_2, a_2, r_2, ..., s_T, a_T, R_T)$$

This sequence can now be used to train a transformer, to predict what state, action or reward and in the next step. However, this will not be able to predict the best actions that result in higher rewards.

Decision Transformer

DT [19] solves sequence-modeling problems by conditioning a transformer model on returns-to-go. In DT the sequence is rewritten as:

$$\tau = (\hat{R}_1, s_1, a_1, \hat{R}_2, s_2, a_2, ..., \hat{R}_T, s_T, a_T), \quad \text{where}$$
$$\hat{R}_t = \sum_{i=t}^{T} r_i$$

$\hat{R}_t$ is called the Return-to-go where this value is the sum of all future rewards. Because this value requires to know how the episode ends, this can only be trained offline. However, this rewrite now connects the return-to-go to which actions should be taken, such that when evaluating the model, if the initial return-to-go value is very high, the model has trained to know what actions come after that. It also means that a sequence with high return-to-go at the start is valuable information as it signifies being a sequence of actions that end in a high return at the end of the sequence.

Training a DT model requires a dataset, as it is learning offline and not directly in the RL environment. This dataset should contain good and bad samples, in order to model different target reward values. The target reward value is used during evaluation, and decides what reward value the agent should try to approximate, which makes it a parameter for how optimal an agent is performing.

Q-learning Decision Transformer

A limitation of DT is that it needs good data samples to model its behaviour on. This means DT will perform poorly using only random data, but improve if the dataset includes sequences that produce higher rewards, as it needs to learn explicitly from these. Yamagata et al. [86] attempts to tackle this problem with Q-learning DT (QDT), using conservative Q-learning (CQL) [44] to learn to estimate the lower bound of the reward function. The framework augments a offline dataset by relabeling return-to-go values with on a values from a learned reward function through CQL. Essentially, this algorithm re-estimates the value of each action in a dataset, which improves the data quality. Results for QDT and DT are shown in table 3.3. Comparing QDT to DT shows small improvements for some RL environments, but not enough to consider it an overall improvement over DT.

3.3.3 World Model learning

In world model learning, an agent tries to learn how an environment changes in order to create a model of the environment. The purpose of this model is to create a simplified version of the environment that an agent can learn from in place of the actual environment. This can benefit environments where observations are presented as RGB-images to reduce the dimensionality and features of the input [41].

Model learning normally has two different parts; a model that predicts the next state, and model that predicts the reward by doing an action in that state. By doing this the second model can predict a reward for every action, and the highest reward can be chosen. This idea is famously used in the Dreamer [33] algorithm, which learns a world model and then learns by simulating the environment using imagination.

IRIS

Existing world models like Dreamer are sample inefficient, needing millions of game data observations and taking days of computation time to train a state-of-the-art model [50] [33]. Improving sample efficiency is important to the progress of reinforcement learning as it can drastically reduce training time and computational power required to train agents. This problem is explored by Micheli et al. [50] with the IRIS model. The model is composed of a discrete autoencoder and a Transformer. The discrete autoencoder [79] converts an input image to-and-from tokens with discrete values, an encoder $E : \mathbb{R}^{h \times w \times 3} \to \{1, ..., N\}^K$ and decoder $D : \{1, ..., N\}^K \to \mathbb{R}^{h \times w \times 3}$. This mechanism is trained on collected images from a given environment. The transformer learns environment dynamics by taking a sequence of tokens and actions $(x_0, a_0, x_1, a_1, ..., x_t, a_t)$ to model the probabilty of transition, reward and episode termination. During each iteration, the next frame and action are added to the sequence to autoregressively predict the probability distributions. The action and reward predictions are based on the DreamerV2 [32] actor-critic model, where the objective is to predict the reward for a given action and observations tokens.

IRIS improves on Dreamer by shortening training time by limiting data gathering. Dreamer was trained on 200 million frames per environment [50], while IRIS trains using the Atari 100k benchmark which only allow agents to do 100k actions per training iteration. Experiments show that IRIS outperforms humans in 10 out of 26 Atari games, and on average outperforms algorithms like SimPLe [40], CURL [46], DrQ [43] and SPR [66], and DreamerV3 in some cases as shown in table 3.2. This shows that IRIS performs well even with limited datasets. However, IRIS authors report 3.5 days of training per environment with 8 GPU's [50, p. 21], making the model costly to run on lesser hardware.

Environment [10]	Algorithms	
	IRIS [50]	DreamerV3 [33]
Alien	420	**959**
Assault	**1524**	706
Asterix	854	**932**
BattleZone	**13074**	12250
Boxing	70	**78**
Breakout	**84**	31
Frostbite	259	**909**
Gopher	2236	**3730**
KungFuMaster	**21760**	21420
MsPacman	999	**1327**
Pong	15	**18**
Qbert	746	**3405**
RoadRunner	9615	**15565**
Seaquest	**661**	618

Table 3.2: Results as gathered reward for IRIS [50] and DreamerV3 [33] in a range of Atari games. Best results for an environment are marked in **bold**.

3.3.4 Generalist agents

The idea of generalist agents follows the NLP domain and its success with GPT, where huge amounts of data are used to create a LLM(Large language model). In RL the idea is the same, where learning on a large dataset creates a general agent that only requires small amount of fine-tuning to succeed in a specific environment. A good example of this is Algorithm Distillation [47].

Algorithm Distillation

In Algorithm Distillation (AD) the idea as the name specifies, is to distill one or more algorithms into one model. This can be done by generating a large dataset of one or more RL algorithms containing:

$$D = \{(s_1, a_1, r_1, s_2, a_2, r_2, ..., s_T, a_T, R_T), ...\}$$

Now a Transformer can learn on this dataset using the loss function $L(\theta)$ below [47]. This loss function trains the transformer based on what action is most probable to come next based on what it has seen in the training set.

$$L(\theta) = -\sum_{n=1}^{N}\sum_{t=1}^{T-1} \log P_\theta(A = a_t^{(n)} | h_{t-1}^{(n)}, o_t^{(n)})$$

3.3.5 Comparison of Deep Reinforcement Learning Algorithms

Table 3.3 compares DT, QDT, IRIS and DreamerV3 in Atari and MuJoCo [77] environments. To summarize, DT performs better than QDT in MuJoCo with continuous actions, and outperforming IRIS in Atari where there are discrete actions. However, as DT is reliant on good data, one could argue that a lot of time was put into building high quality datasets with sequences that produce high rewards. As Yamate et al. [86] reports, the authors of QDT did not have access to the same dataset as DT, but the Q-learning integration does not seem to improve on the DT model substantially. DT performs similarly to DreamerV3 in Atari tasks, however, as shown in table 3.2, DreamerV3 performs well in a wide range of Atari tasks. DT was only evaluated on four Atari environments, which could be attributed

to the fact that each environment requires its own dataset, and data gathering is time-consuming. For this reason, DreamerV3 is considered a state-of-the-art model in RL, as it outperforms algorithms like SimPLe, CURL, DrQ and SPR in a wide range of environments [33]. However, DreamerV3 is not based on the Transformer architecture.

Environment	Algorithms			
	DT [19]	QDT [86]	IRIS [50]	DreamerV3 [33]
Breakout [10]	76.9 +- 27.3	-	**83.7**	31
Qbert [10]	2215.8 +- 1523.7	-	745.7	**3405**
Pong [10]	17.1 +- 2.9	-	14.6	**18**
Seaquest [10]	**1129.3 +- 189.0**	-	661.3	618
Hopper [77]	**107.6 +- 1.8**	66.5 +- 6.3	-	-
HalfCheetah [77]	**86.8 +- 1.3**	42.4 +- 0.5	-	-
Walker [77]	**108.1 +- 0.2**	67.1 +- 3.2	-	-

Table 3.3: Average rewards for different Transformer based algorithms, comparing the highest reported result for each environment. Best averages for a given environment are presented in **bold**.

Chapter 4

Methods

The computer engineering and RL research done in this master's thesis consists of the following:

- Development of RL framework to interact with DeepRTS
- Development of 4 RL environments in DeepRTS
- Development and testing of reward functions in DeepRTS
- Development, training and evaluation of intelligent agents, wherein:
  - DDQN Agent
  - DT Agent
  - GADT Agent
- Building 12 different datasets for training DT Agents

This work is necessary to fully evaluate Transformer models and expand the research base of the DeepRTS RL environment. As this is a comparative study and DeepRTS has little existing research, this work requires multiple intelligent agents to be trained in multiple environments for them to be comparable.

This chapter presents details of the work done to test the hypotheses in section 1.2.2 and achieve the goals as stated in section 1.2.1. This includes an overview of the project, a description of the DeepRTS RL environments, and their configurations wherein. Then, reward functions and OpenAI gym implementations developed to act in RL environments, before describing work to create labeled data to train Transformer models. Finally, the novel implementation GADT is introduced along with DT and DDQN agents.

4.0.1 Overview

Figure 4.1 shows an overview of how an agent acts in the DeepRTS environment, with the DT agent as an example. Agents are reliant on signals to base their actions on, being observations of the current and past game states and rewards for a given action. Observations in DeepRTS are encoded as a state matrix, rewards as floating-point numbers, and actions as integers. This game information is exposed through an OpenAI Gym [15] interface. For a given timestep t, the DT agent observes past game information and predicts the next action. This is an autoregressive process where the agent uses its own predictions to make subsequent predictions. However, the DT agent has been trained using a static dataset in advance. For a DDQN agent, the process is similar, apart from not using sequential information and selecting actions

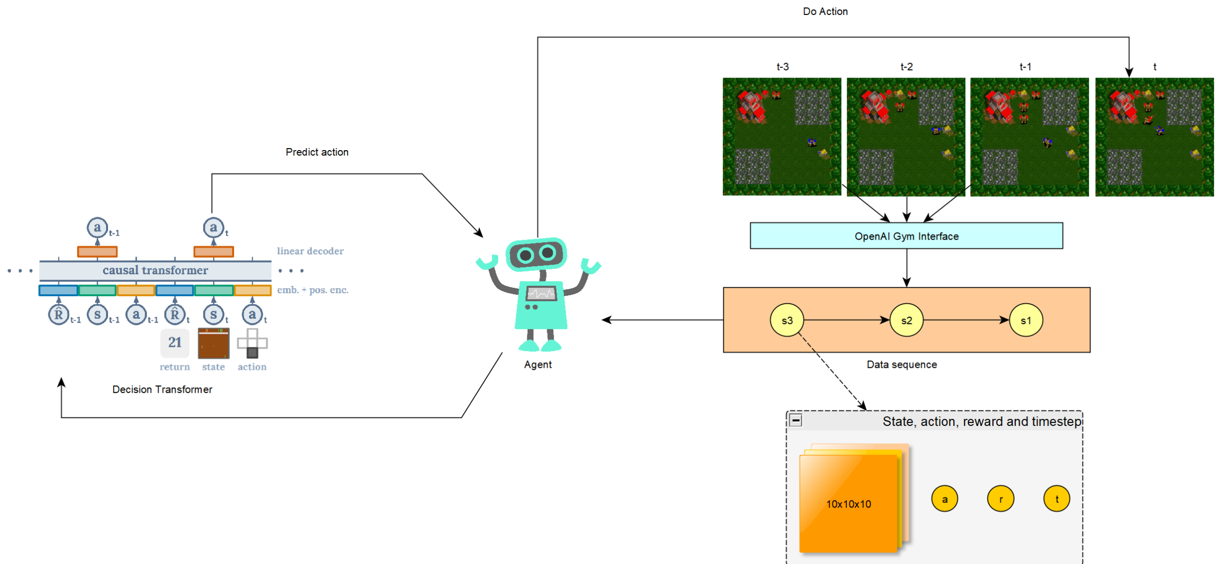


Figure 4.1: Overview of the DeepRTS DT implementation, showing how an agent predicts its actions.

4.1 DeepRTS

DeepRTS is a real-time strategy game made in C++, for the purpose of high-performance RL [7]. DeepRTS is also exposed to Python using PyBind11, a library that binds C++ types to Python, enabling Python development for C++ projects [84]. Because the Python code is compiled using a C++ compiler, DeepRTS runs with high performance in C++, while popular Deep Learning frameworks like Pytorch and Tensorflow can process data from DeepRTS in Python. As shown in section 3.1.4, DeepRTS can output up to 7 000 000 frames per second in-game simulation, which outperforms similar RTS environments like MicroRTS and TorchCraft. Furthermore, the goal of this thesis is to expand RL research using DeepRTS, as there are only a handful of projects using it [9], [65].

Agents in DeepRTS are trained iteratively in episodes, which are isolated simulations of an environment limited by a number of in-game ticks. When an episode has ended, metrics like cumulative reward, number of actions and optimizer loss can be gathered and analyzed to see if agents learn from their experiences.

4.1.1 Observation space

DeepRTS encodes game data into a 3-D matrix for the purpose of deep-learning [7, p. 5]. This matrix is rendered and changed every tick and is used as the observation input for deep-learning algorithms presented in this thesis. Figure 4.2 illustrates the observation matrix and its layers. For all experiments in this thesis each environment uses the same map with map-size $10x10$, but the implementation allows changing this map to any map that is available in DeepRTS.

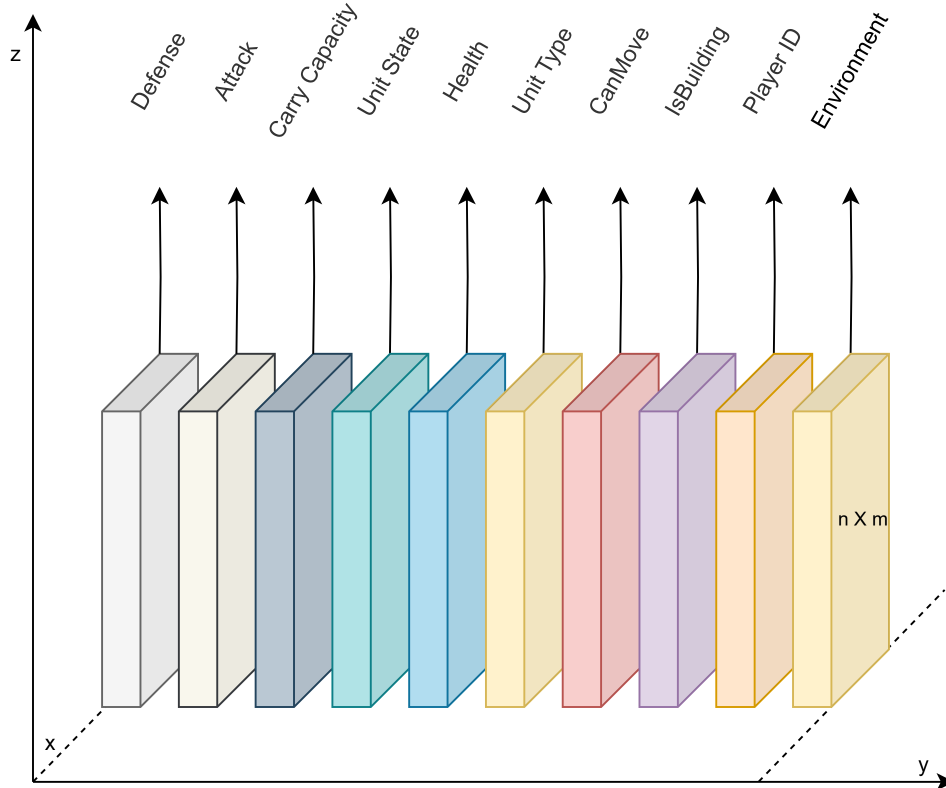


Figure 4.2: 3-D matrix with DeepRTS game data, expanded on [7] in accordance with the DeepRTS source code [6]. mXn is the size of the map.

4.1.2 Action Space

DeepRTS has a discrete action space A:

$A = \{$
$\quad PreviousUnit, NextUnit, MoveLeft, MoveRight, MoveUp, MoveDown,$
$\quad MoveUpLeft, MoveUpRight, MoveDownLeft, MoveDownRight, Attack,$
$\quad Harvest, Build0, Build1, Build2, NoAction$
$\}$

Every 10 frames each player selects an action $a \in A$, because players in DeepRTS is limited by a tick timer that ensures it takes 5-10 ticks for an action to be performed [7, p. 4]. To reduce complexity, the *Simple1v1* and *Medium1v1* environments uses a subset action space $A_1 = \{MoveLeft, MoveRight, MoveUp, MoveDown, Attack, NoAction\}, A_1 \subset A$, while *Harvest* uses the full action space A.

This work omits using left-click and right-click actions, to reduce the complexity of the learning environment. Right-click targets a tile for a selected unit to walk to, but this means that the agent needs to wait for the unit to reach its goal before selecting a new action. Having these two actions would drastically increase the number of actions, as it would depend on the map size.

4.1.3 Game Logic

Based on the action space description, this section will present how an agent can use actions to interact with the game environment.

Controlling Units

$T = \{ PreviousUnit, NextUnit \}, T \subset A$ toggles control of the players units. Moveable units must first be targeted using one of these actions before they can be controlled manually with movement controls. However, a unit may move and retaliate automatically if attacked by an opposing unit.

Since all buildings are unmoveable, the player cannot use movement controls when targeting a building. Instead, Town Hall and Barracks may be targeted using $t \in T$ and be used to construct units. The Farm unit does not construct any units but contributes to the player's food production metric, which controls how many moveable units a player can have.

Constructing Units

The subset $B = \{ Build0, Build1, Build2 \}, B \subset A$ contains the actions that build units and buildings. The actions construct units based on which unit is already selected:

- Town Hall selected:
  - Build0: Constructs a Peasant
- Peasant selected:
  - Build0: Construct Town Hall
  - Build1: Construct Farm
  - Build2: Construct Barrack
- Barrack selected:
  - Build0: Construct Footman

Figure 4.3 shows a map where players have built different units, in this case, barracks (3x3), footmen (1x1), and farms (1x1).

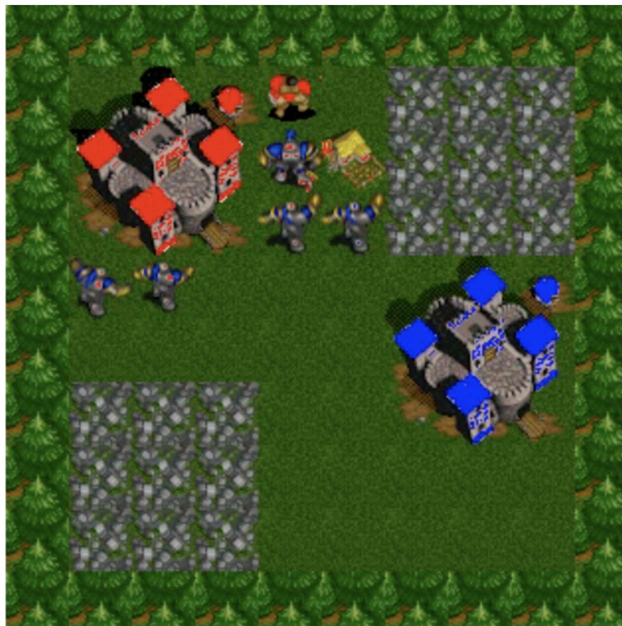


Figure 4.3: An example map screenshot showing different units in DeepRTS.

4.1.4 Improvements and Fixes

During testing, several bugs were found and fixed to improve RL development. Firstly, several uninitialized integer variables in the DeepRTS source code made it impossible to build multiple units. Each costs a certain amount of resources (gold, stone, or lumber), and when building a unit the cost is subtracted from the player's available resource pool. These costs must be initialized to some value in the source code, however, each unit's stone cost variable was not initialized, leading to the player's stone resource pool being subtracted by an undefined value, which leads to having an undefined value of stone resources. This leads to unexpected behaviour such as not being able to build units that cost stone. This was fixed by properly initializing the stone cost of each unit to its intended value [26].

Another issue was caused by uninitialized values in the state array in DeepRTS, which encodes game data as explained in section 4.1.1. The state array used during training caused the calculated Q-values and action predictions to be undefined, making learning impossible because the agent learned to predict undefined values. This issue was solved by initializing all values in the state array at the start of each game to -1, as this value is never used to represent the game state and works as a neutral value.

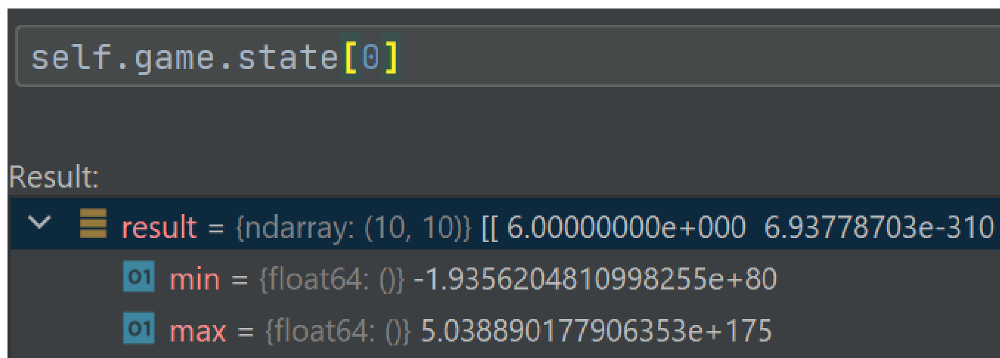


Figure 4.4: The game state array in a Python debugger. Note the min and max values, showing that the array contains undefined values.

4.1.5 Environments

This work includes the development of DeepRTS environments to train agents for specific tasks. Four different environments were created and evaluated for two primary reasons. The first reason pertains to speed, where a simpler environment with a smaller action space and reduced randomness was utilized. This allowed models to achieve noticeable results with minimal training time, facilitating the evaluation of numerous models within a short duration. Subsequently, the second reason involved scalability. The environments were scaled up to include larger action spaces and increased randomness, enabling the assessment of model performance in handling higher levels of randomness and expanded action spaces. This thesis presents the following four environments in DeepRTS:

- Player-versus-player (PVP) environments
  - Simple1v1
  - Random1v1
  - Full1v1
- Harvest

The three PVP environments consist of two players, where an RL agents controls one of them. PVP environments are designed for agents to learn how to attack and find the shortest path to

an enemy to defeat it. The next environment is *Harvest*, which is a single-player environment. Both *Harvest* and *Full1v1* focus on rewarding resource gathering and constructing buildings, being an essential part of any DeepRTS game. $Player_0$ is controlled by an RL agent.

Max Episode Lengths

Each episode in an environment is ended when one player defeats the other player, or when the number of game ticks reaches a certain limit. For *Simple1v1 and Random1v1*, the maximum episode length is 500 ticks. This is because tasks are relatively easy and specific, and testing shows that an agent can defeat the enemy in as low as 90-100 ticks. However, a higher max game tick count allows the agent to explore the observation space further to learn from previously unexplored observations.

The max length for *Harvest Full1v1* is 2000 ticks. Tasks in these environments are more complex and have multiple steps to complete, which means the agent requires more exploration as it is allowed to use the entire action space with access to more game mechanics.

Simple1v1

Simple1v1 consists of two players, where one can move freely and the other cannot initiate actions. Hence, the RL agent controls the moveable player $Player_0$, and the goal is to defeat the other player $Player_1$ by attacking it repeatedly. Since $Player_1$ cannot move, this essentially turns the game into a fully visible maze, where the RL agent must move to the correct tile using the shortest path to win the game. However, $Player_1$ may respond to attacks and move if the $Player_0$ stops its attacks. This makes it important to attack repeatedly, as this will result in reaching a terminal state, where the player who attacks repeatedly first wins.

Random1v1

Random1v1 is similar to *Simple1v1*, where the difference is that the RL agent plays against a random player. The environment also changes in that it has a moving target, and the agent must learn how to find the target and defeat it. Since actions happen in real-time and the players move at the same speed, the agent must find a way to corner the enemy and take advantage of the randomness of the opponent.

Full1v1

Players in *Full1v1* are able to use the entire action space A (see section 4.1.2), which enables players to construct buildings and units and toggle unit control. Each player starts with the same amount of resources, 5000 credits each of gold, lumber, and stone. This makes this environment more complex, as it simulates a full RTS game. Players can play more defensive by constructing units to gain an advantage.

Harvest

The *Harvest* environment centers around maximizing the number of harvested resources in a game. The environment is populated by a single player that is controlled by an RL agent. Each moveable unit can carry a total of 10 resources. The unit must then offload the resources in a Town Hall building, so it can gather more resources. The optimal strategy is then to build a Town Hall, build as many units as possible, and harvest different resources simultaneously.

4.2 Reward functions

The output of a reward function provides a measure of the fitness of an agent's action in an environment. The goal of an agent is to maximize this function and hence, it is essential to how an agent learns behaviour in an environment. By learning to maximize the reward function, the agent learns to perform optimally in an environment.

Reward functions can be designed to match the goal in each environment. Reward functions in PVP environments must reward defeating the opposing player as quickly as possible, while the *Harvest* environment focuses on maximizing gathered resources. Some rewards are given every tick, while others are given once each time a specific event occurs, such as when building a unit or attacking.

4.2.1 PVP Rewards

PVP rewards are centered around maximizing speed and efficiency, defeating the opposite player in as few game ticks as possible. To do this, the agent has to walk toward the enemy and attack it when it is one tile away from it. Figure 4.5 shows the first graphical frame of a DeepRTS game. The players always start in the same position and may walk freely to any tile that is not a building or a resource, eg. trees or gold. The goal of the PVP environments is then to find the shortest path to the enemy and attack it when close.

Figure 4.5: The first frame of a DeepRTS game.

Algorithm 1 shows the reward function for *Simple1v1* and *Random1v1*. The function checks the game state every tick, and outputs penalties of -1 most of the time, as the rewards are sparse and handed out when certain game events occur. The first condition is to check if the agent has won the game, which is done by repeatedly attacking the opponent. Here, the reward of 10000 is discounted by the number of current game ticks. This ensures that this reward is higher towards the start of the game and decreases as the game goes on, in an attempt to provide the agent with information about the importance of speed, as it will receive more reward if fewer game ticks have passed when initiating attacks on the enemy. Because the victory event can only occur once per episode by doing enough damage to the enemy, the reward value for this event is the highest. On the contrary to victory, the check that the agent has been defeated will penalize the agent higher later on in a game rather than earlier. This is to penalize the agent more if it does not find the shortest path and does

not win the game. However, if it does find the shortest path and still loses the game the penalty is lowered, as it completed the shortest path objective. Finally, the agents receive a reward when it is attacking the enemy, also discounted by game ticks. This event can occur multiple times during an episode, which is why the reward value is lower.

Algorithm 1 *Simple1v1* and *Random1v1* PVP reward function

```
1: if player0_won then
2:     return 10000/ticks
3: if player0_defeat then
4:     return -0.001*ticks
5: if player0_did_damage then
6:     return 1000/ticks
7: return -1
```

The reward values are chosen to weigh the fitness of an action relative to each other. Values are also chosen with respect to the maximum episode length for the environments. Because rewards are sparse and handed out only during certain events, the -1 penalty per tick provides information to the agent that it needs to minimize how many actions it uses and how many episode ticks it uses, to ensure it receives as few penalties as possible.
Full1v1 expands on Algorithm 1 by also including rewards for constructing units. The reward function for *Full1v1* is shown in Algorithm 2. Rewarding construction is done by comparing the player state between episode ticks, to see if the number of units has increased. Some buildings and units are more important than others, based on their game features. The Farm building increases movable unit capacity but is not as important as it does not produce any units by itself. Instead, the reward for building Town Hall and Barrack buildings is higher because their purpose is to construct movable units. The most valuable unit is the Footman which deals the most damage per attack, and constructing such a unit is also rewarded greatly. can Finally, the player receives a reward based on the PVP reward function as shown in Algorithm 1.

Algorithm 2 *Full1v1* reward function

```
1: player = player0_{ticks}
2: previous_player = player0_{ticks−1}
3: if player.num_farm > previous_player.num_farm then
4:     reward += 20
5: if player.num_town_hall > previous_player.num_town_hall then
6:     reward += 50
7: if player.num_peasant > previous_player.num_peasant then
8:     reward += 50
9: if player.num_barrack > previous_player.num_barrack then
10:    reward += 100
11: if player.num_footman > previous_player.num_footman then
12:    reward += 100
13: reward += simple_pvp_reward()                    ▷ see Algorithm 1
14: return reward
```

4.2.2 Harvest Reward

The reward function for the *Harvest* environment is shown in Algorithm 3. The function rewards harvesting and building units that increase harvesting capacity. This is done by comparing the player state between game ticks, to see if the number of gathered resources has increased. This environment allows for multiple units, and the reward function is designed to fit this as a sum of multiple rewards for different events. Because rewards and

penalties are less sparse because there are more actions and game mechanics enabled, rewards are not discounted by game ticks or constant negative penalties.

To provide the agent with penalties while no reward events are occurring, a penalty based on current episode ticks is given every tick. This penalty increases over time, and the agent can counteract this by gathering resources continuously. The next penalty condition is penalizing the agent if the current targeted unit is a building. This is to penalize the agent for not moving units around, and also because buildings cannot harvest resources. Due to DeepRTS's targeting system, up to one unit or building can be targeted to use its abilities. After a player has targeted a building such as a Town Hall, the player can build units using actions. However, the player can not move other units around when targeting a building, which is crucial to resource gathering.

Next, the agent is rewarded equally per resource gathered per tick. In an ideal scenario, the agent can order its units to gather a type of resource each, which will maximize rewards for harvesting. Next, the agent is rewarded if it builds a Town Hall or a Farm. Town Halls are rewarded more than Farms as it lets the agent construct movable units.

Algorithm 3 Harvest reward function

```
 1: player = player0_{ticks}
 2: previous_player = player0_{ticks-1}
 3: reward = -ticks/max_episode_ticks
 4: if player.targeted_unit.is_building == False then
 5:     reward -= 1
 6: if player.stone > previous_player.stone then
 7:     reward += 10
 8: if player.gold > previous_player.gold then
 9:     reward += 10
10: if player.lumber > previous_player.lumber then
11:     reward += 10
12: if player.num_town_hall > previous_player.num_town_hall then
13:     reward += 20
14: if player.num_farm > previous_player.num_farm then
15:     reward += 1
16: if player.num_peasant > previous_player.num_peasant then
17:     reward += 10
18: return reward
```

4.3 Implementation

In figure 4.6 an overview of the code implementation can be seen. The main.py file has the responsibility of connecting a gym and an agent depending on what is chosen, this happens by using a config.json file.

4.3.1 Gym implementation

Environments are implemented as Gyms using the OpenAI Gym library for Python. Each gym environment inherits from a generic custom Gym class, which is shown in listing 1. The *Simple1v1* child class contains environment-specific configurations, such as the action space, game configuration, and reward function. The function *step* defines what happens every game tick, where the agent and optional second player perform actions and calculates reward

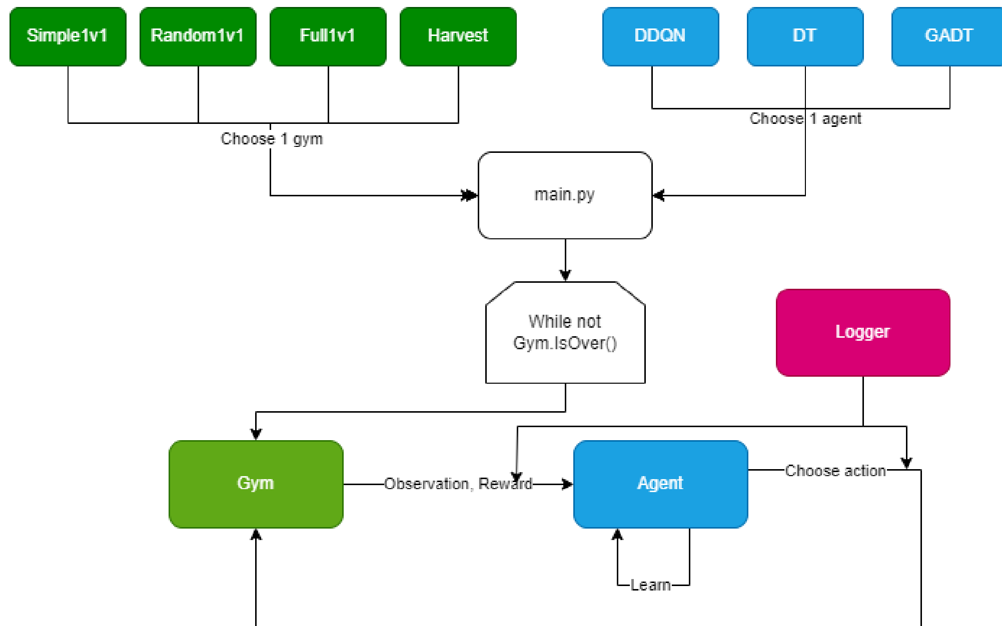


Figure 4.6: Overview of implementation.

based on the agent's actions per tick. This pattern is used to implement all environments from section 4.1.5, and specifications for each environment can be found in the appendices.

Listing 1 Generic Gym Inteface, and the *Simple1v1* class. Some code was omitted for simplicity.

```
from DeepRTS import Engine, Constants
import gym
class CustomGym(gym.Env):
    def __init__(self, max_episode_steps, shape, game_map, config):
        self.max_episode_steps = max_episode_steps
        self.elapsed_steps = None

        self.game: Engine.Game = Engine.Game(game_map, config)
        self.player0: Engine.Player = self.game.add_player()
        self.game.start()
        self.previousPlayer0 = PlayerState(self.player0)

class Simple1v1Gym(CustomGym):
    def __init__(self, max_episode_steps, shape):
        engineConfig: Engine.Config = Engine.Config().defaults()
        engineConfig.set_auto_attack(True)

        self.action_space = [3, 4, 5, 6, 11, 16] # move and attack simple

        super().__init__(max_episode_steps, shape, MAP, engineConfig) #init CustomGym

        self.player1: Engine.Player = self.game.add_player()

    def step(self, actionIndex):
        self.elapsed_steps += 1
        self.action = actionIndex

        self.player0.do_action(self.action_space[actionIndex])
        self.player1.do_action(16) # do nothing
        self.game.update()

        reward = pvp_reward()

        #return obs, reward, is_done, max_steps_reached, info
        terminal = self.game.is_terminal()
        return self._get_obs(), reward, terminal, False, self._get_info()
```

Gym environments enable a simple formulation of the training loop, where an agent interacts with the environment through the Gym interface to learn policies. A simplified version of the DDQN training loop is shown in listing 2.

Listing 2 Generic Agent Training loop using Gym. Some code has been omitted for simplicity.

```
for e in range(epochs):
    observation, info = gym.reset()
    ticks = 0
    done = False
    truncated = False
    while not done and not truncated:
        ticks += 1

        actionIndex, q_values = agent.act(observation)

        next_observation, reward, done, truncated, info = gym.step(actionIndex)

        game_finished = done or truncated
        agent.cache(observation, next_observation, actionIndex, reward, game_finished)

        # Learn
        q, loss = agent.learn()

        observation = next_observation
gym.close()
```

Environment Wrappers

OpenAI Gym provides various wrappers to alter output data during training [55]. For this work, the following wrappers are used:

- Skip Frame ($n = 10$): Each tick skip n frames, output last observation
- Time Limit: output a "game over" signal if the running environment has reached n number of ticks
- DDQN only:
  - Frame Stack ($n = 3$): Each tick take n observations and concatenate them
  - Transform Observation: For each value in observation, perform the following transformation to normalize the observation: $x/20$

4.3.2 Agent implementation

The agent implementation in this work specifically for DT utilizes the DT model from the Transformers library by Hugging Face [24]. This implementation is based on the original DT paper [19] with a small difference. In the original DT paper a minGPT by Karpathy [42] is used, while the huggingface implementation uses GPT2 [62].

4.4 Datasets

Unlike online RL where an agent learns to estimate rewards from direct experience in the environment, DT learns offline using game data D, with n number of games with $m > 0$ timesteps:

$$D = \{ G_1, G_2, ...G_n \}$$

$$G_n = \{ (s_1, a_1, t_1, r_1), (s_2, a_2, t_2, r_2), ...(s_m, a_m, t_m, r_m) \}$$

This data is generated using various policies for picking actions, namely random and using a trained DDQN agent to make actions. Each game is generated using the same policy, to make the following datasets:

- Random: 100% games with randomly generated actions
- Pretrained: 100% games with actions generated by a trained DDQN agent
- Mix: 50% Random games and 50% DDQN games

These datasets were used to explore if DT can learn the best sequence of actions from a mix of good and subpar actions. Because of the different reward functions and action spaces of the environments presented in section 4.1.5, each environment requires three datasets for a total of 12 different datasets. Furthermore, each dataset contains 1000 games.

4.5 Baseline

The DDQN agent is an online RL agent and does not rely on a static dataset. For this work, the DDQN agent is used as a baseline and basis for pretrained DT datasets, trained in each of the four environments.

4.5.1 Double Deep Q-Network

The DDQN agent used the epsilon-greedy strategy [73] to alternate between exploration and exploitation in the environment. The agent collects experience in a replay buffer with a max size, and samples mini-batches of replays to learn q-values for each action in the action space. The architecture for the DDQN model is shown in figure 4.7. The last output is a vector where N is the length of the given action space, see section 4.1.2.

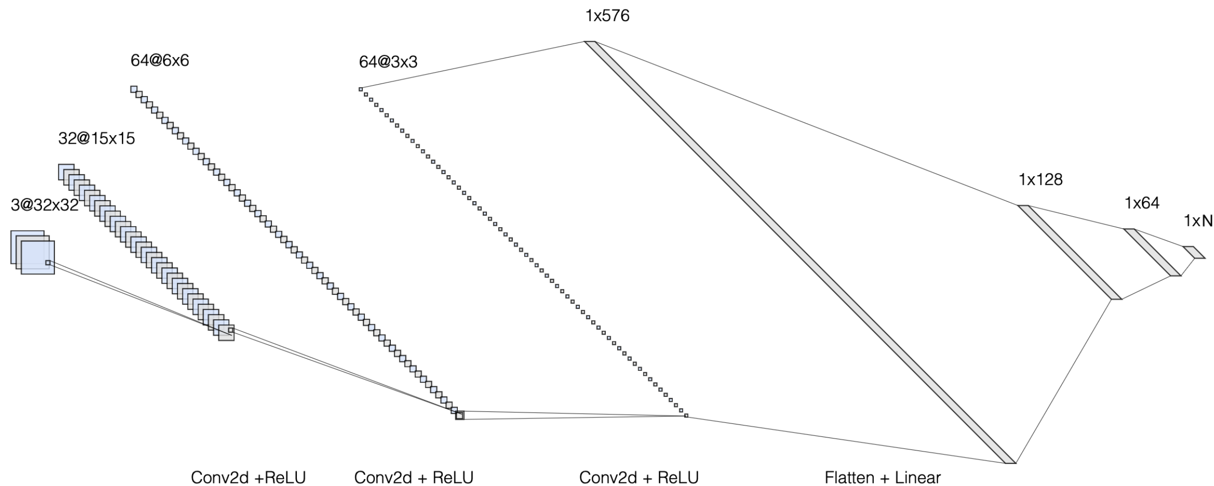


Figure 4.7: The architecture for the DDQN model used in this work

4.6 Transformers

Two Transformer algorithms are used to train agents to play DeepRTS in this work; DT and GADT. GADT is a novel implementation improving on the existing DT. All algorithms and models are implemented in Pytorch, the machine learning framework for Python [56]. Furthermore, hyperparameters applied to the different models are important for the reproduction of results and can be found in appendix A.3.

4.6.1 Decision Transformer

The architecture of the decision transformer can be seen in figure 4.8.

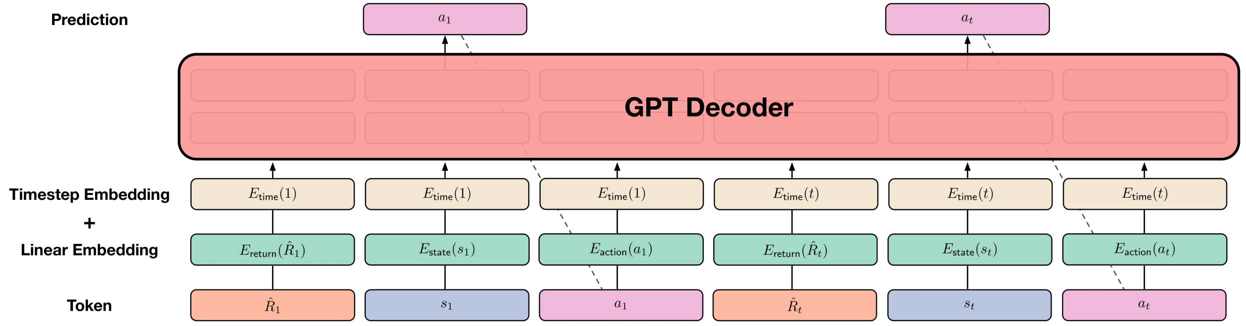


Figure 4.8: DT architecture [19]

DT learns DeepRTS through offline RL, using various static datasets and evaluating performance after training. This thesis explores how DT manages to learn using datasets with varying degrees of data quality, as presented in section 4.4.

DT as described by Chen et al. [19] samples mini-batches of sequence length K with rewards-to-go, states, timesteps, and actions. The agent accepts 4 sequences with length $batch_size$, one for each token states, actions, timesteps, and rewards.

Listing 3 DT Training loop. Some code has been omitted for simplicity.

```
for e in range(epochs):
    agent.net.train()
    for game in dataLoader:
        batches = get_batches(game, batch_size=batch_size)
        for i in range(len(batches[0])):
            observations = batches[0][i]
            actions = batches[1][i]
            timesteps = batches[2][i]
            rewards = batches[3][i]

            loss, q = agent.train(observations, actions, timesteps, rewards)

    agent.net.eval()
    # after training, evaluate (code omitted)

```

Before the sequence is forwarded into the GPT-model, rewards-to-go are calculated. The loss function as described by Chen et al. [19] is used, where $loss = mean((\hat{actions} - actions)^2)$. Adding state and reward prediction loss calculation was also tested, however, this did not result in improved performance.

4.6.2 Genetic Algorithm Decision Transformer

Using the model from DT, this thesis proposes changes to DT by implementing elements of online RL and data generation, called Genetic Algorithm DT (GADT). While DT, as presented by Chen et al. [19] learns from a static dataset, GADT does not rely on a predefined dataset or pretrained models as it generates data itself and learns from it. Algorithm 4 shows the implementation of GADT. Games are generated and the best games are picked based on total cumulative reward in each game iteration. The DT itself acts as the mutator from GA,

as it generates new games based on high-quality data. This is done in an effort to improve the quality of data that the DT model is exposed to in order to find the sequences of actions that result in high rewards. In addition, this technique removes the dataset dependency of the DT model.

Algorithm 4 GADT

```
epsilon_rate, epochs, dt_epochs, max_best_data = int
max_best_data_decrease_factor, epsilon_action_decrease_rate = float
best_data = []
for e in range(epochs):
    generated_data = []

    """
        Generate data
    """
    DecisionTransformer.eval()
    epsilon = 1 - (min(e*epsilon_rate, epochs)/epochs)

    for epi in range(episodes):
        target_return = 100 # max return value
        R, s, a, t, done = [target_return], [gym.reset()], [], [1], False

        while not done:
            # epsilon-greedy strategy
            if random.random() < epsilon or e == 0:
                action = randomAction()
            else:
                a_preds = DecisionTransformer(R, s, a, t)[-1]
                action = argmax(a_preds)

            # decrease randomness
            epsilon *=epsilon_action_decrease_rate

            next_s, r, done, info = gym.step(actionIndex)

            # update sequences
            R = R + [R[-1] - r]
            s, a, t = s + [new_s], a + [action], t + [t[-1] + 1]

        generated_data = generated_data + [[R, s, a, t]] # save game data


    """
        Genetic algorithm
    """
    # sort all games based on total reward, in descending order
    best_data.sort(key=getSumOfGame, reverse=True)
    # get total reward
    old_reward_records = [getSumOfGame(game) for game in best_data]

    best_data = best_data + generated_data
    best_data.sort(key=getSumOfGame, reverse=True)

    # get only max_best_data number of games, ignore others
    best_data = best_data[0:max_best_data]
    new_reward_records = [getSumOfGame(game) for game in best_data]

    # the model hasnt produced any better game
    if old_reward_records == new_reward_records:
        max_best_data = int(max_best_data * max_best_data_decrease_factor)

        if max_best_data < 2:
            exit() # the model cant learn much more
        else:
            best_data = best_data[0:max_best_data] # deacrease best_data size

    """
        Train agent
    """
    DecisionTransformer.train()
    dataset = Dataset(best_data)
    for _ in dt_epochs:
        for (R, s, a, t) in dataset:  # dims: (batch_size, K, dim)
            a_preds = DecisionTransformer(R, s, a, t)
            loss = mean((a_preds - a)**2)  # MSE
            optimizer.zero_grad(); loss.backward(); optimizer.step()
```

Chapter 5

Results & Discussion

To evaluate DeepRTS and the implemented models, we present experiments for the four DeepRTS environments. This chapter presents the results for experiments using random, DDQN, DT and GADT agents in the DeepRTS environments *Simple1v1*, *Random1v1*, *Full1v1* and *Harvest*.

To train and evaluate agents a NVIDIA A100 GPU with 8 GB VRAM and a NVIDIA RTX 3080 10GB VRAM GPU was used. Table 5.3 shows the results for DT using various datasets, and table 5.6 shows the best results for Random, DDQN, DT and GADT agents. Furthermore, evaluation metrics were gathered from 1000 episodes across three seeds with each model, taking the average reward and deviation as a metric for performance.

5.1 Results

5.1.1 Experiment 0: Baselines

To be able to compare DT and GADT and generate datasets, we train Random and DDQN agents in all DeepRTS environments. The random agent uses a random policy to generate actions, and is meant to show the lower bound of the reward function as acting randomly in an environment will generally produce sub-optimal behaviour. The DDQN agent is vital as it can learn to approximate optimal behaviour in an environment, and hence produce data with high reward and good action sequences. By training a DDQN agent we achieve a baseline model that also can be used to generate datasets for training DT and GADT.

Random

Table 5.1 shows results for a Random agent in the four DeepRTS environments. Tasks in DeepRTS are complex and rewards are sparse, meaning agents must complete sequences of tasks to gain reward. Because of this, the random agent has bad performance and its purpose is to show the lower bound of the reward function.

Environment	Random
Simple1v1	-47.93 $\pm$ 0.136
Random1v1	-40.29 $\pm$ 0.498
Full1v1	18.17 $\pm$ 1.658
Harvest	-147.1 $\pm$ 1.884

Table 5.1: Average reward and deviation for the random agent, evaluated for 1000 episodes across three seeds.

Double Deep Q-learning Network agent

The DDQN agent is used as a baseline for performance in our four DeepRTS environments. It also serves as a way to generate high-quality data, that can be extracted as a static dataset to train the DT agent on. Its goal is to be better than the random agent by producing data with a higher total reward. As shown in table 5.2, DDQN outperforms the random agent in 3 of four environments. In the *Harvest* environment DDQN is worse than random, which might be attributed to a reward function that is difficult to learn or model overfitting.

Environment	DDQN
Simple1v1	28.89 ± 1.758
Random1v1	54.37 ± 1.236
Full1v1	28.01 ± 0.902
Harvest	-295.6 ± 13.98

Table 5.2: Average reward and deviation for the DDQN agent, evaluated for 1000 episodes across three seeds.

5.1.2 Experiment 1: Training DT on various data

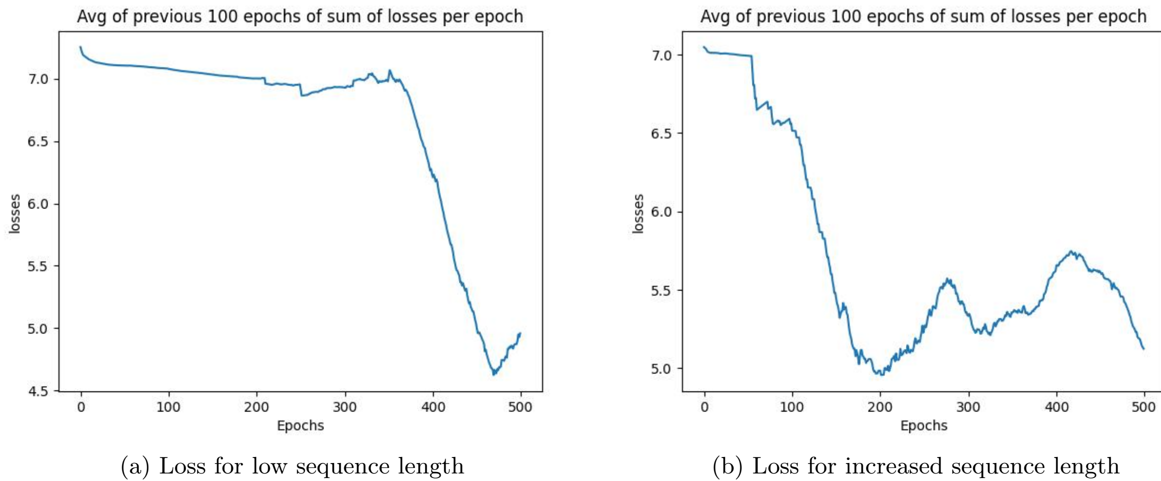
DT was trained using Random, Pretrained, and Mixed datasets as explained in section 4.4, three datasets for each of the four environments presented in section 4.1.5. The motivation for this experiment is to see if DT learns the optimal actions from various datasets. The results for DT in terms of reward are shown in table 5.3. DT performs best using pretrained data in *Simple1v1* and *Random1v1*, benefiting from replays from a pretrained DDQN agent. When it comes to *Full1v1* the DT model performed best after training on random data, which could indicate that the pretrained DDQN has been overfitted, or has not explored the environment enough to find optimal behaviour.

Dataset	Environment	DT
Random	*Simple1v1*	-50
	Random1v1	-44.65 ± 0.416
	Full1v1	**73.53 ± 1.679**
	Harvest	-142.0 ± 2.657
Mixed	*Simple1v1*	29.29 ± 12.04
	Random1v1	51.06 ± 0.892
	Full1v1	-35.28 ± 2.096
	Harvest	**-44.01 ± 61.26**
Pretrained	*Simple1v1*	**46.12 ± 0.227**
	Random1v1	**64.04 ± 0.688**
	Full1v1	-2.781 ± 4.634
	Harvest	-71.57 ± 37.29

Table 5.3: Highest average reward and deviation for DT in DeepRTS, using three different datasets. Models were evaluated for 1000 epochs across three seeds. The highest reward for a given environment is marked with **bold**.

5.1.3 Experiment 2: Improving results by increasing sequence length

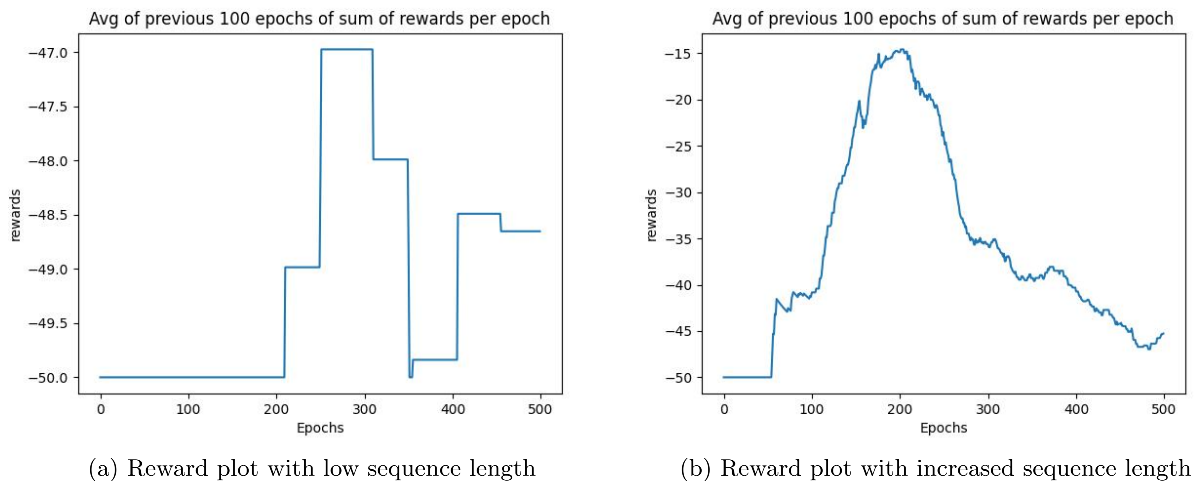
Training DT on *Simple1v1* using the random dataset resulted in subpar performance with low sequence lengths. Sequence lengths define how many tokens the DT model takes as input to the neural network to make action predictions. Increasing sequence lengths also increases memory consumption, but this means the neural network has more information about a game when making predictions. By increasing the sequence length, we saw improvements in learning from random data, as random game data had longer game lengths. Having longer sequence lengths means the model has more context of observation, action, and reward-to-go to base its predictions on. The improvements are shown in figures 5.1 and 5.2, where the reward plot shows increases from an average of -48.5 to -35 when increasing sequence lengths.



(a) Loss for low sequence length

(b) Loss for increased sequence length

Figure 5.1: Losses for training DT in Simple1v1 using Random dataset with various sequence lengths.



(a) Reward plot with low sequence length

(b) Reward plot with increased sequence length

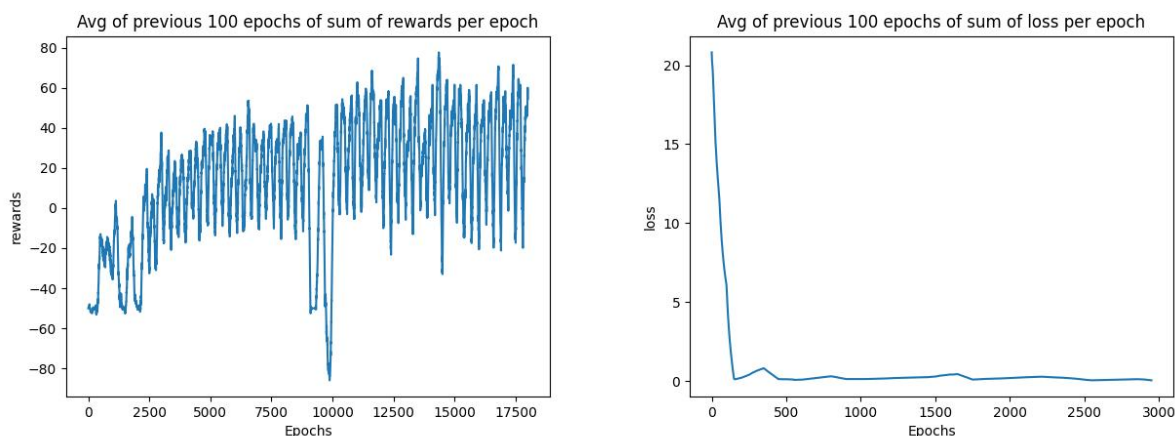
Figure 5.2: Reward for training DT in Simple1v1 using Random dataset with various sequence lengths.

5.1.4 Experiment 3: GADT

In the figures below, plots for the different environments using GADT can be seen. The oscillation in the reward plots is the reset of the epsilon value after the model has trained. This makes it so the model starts making random actions, as this value decreases again the

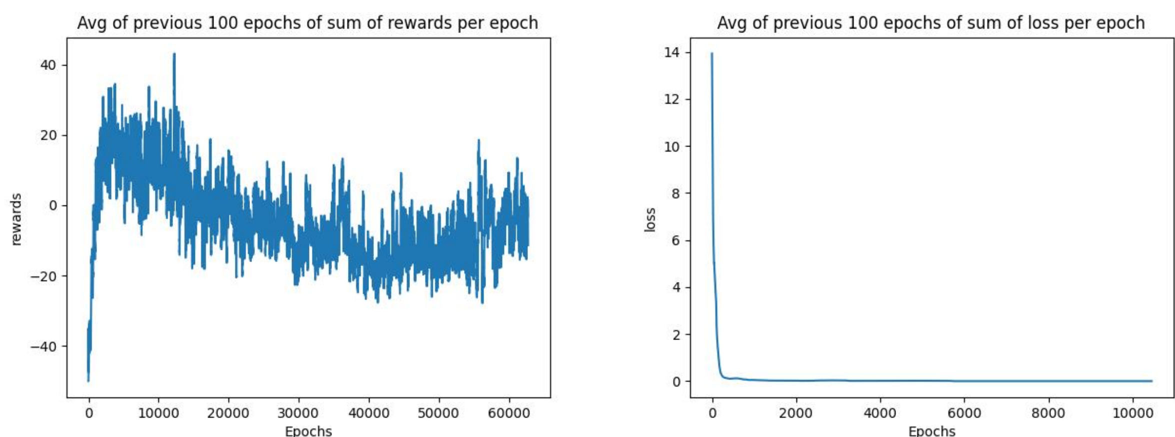
model makes most of the actions and the rewards go up. All except for Random1v1 reward plots (see figure 5.4) increase over epochs, the reason for this will be discussed in section 5.2.1.

The loss plots in all environments seem to decrease rather rapidly and then slowly stagnate at close to zero. This means the model learns to predict the following action given an return-to-go value. The loss plots have different epoch numbers when compared to the reward plots because the loss data is recorded when the agent is offline learning. While the reward data is the reward for each epoch during the generation/training loop explained in section 4.6.2.



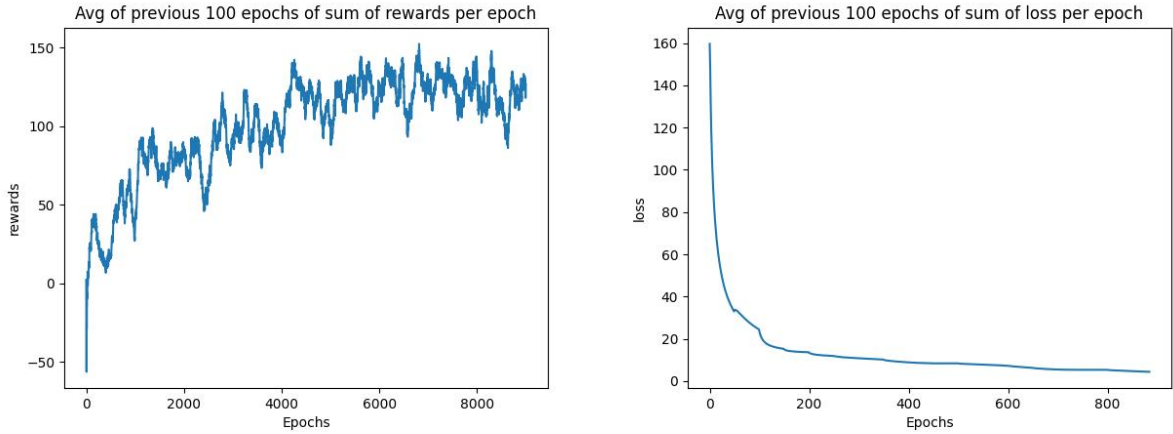
(a) Moving average of 100 previous total reward per epoch (b) Moving average of 100 previous total loss per epoch

Figure 5.3: Simple1v1 plots



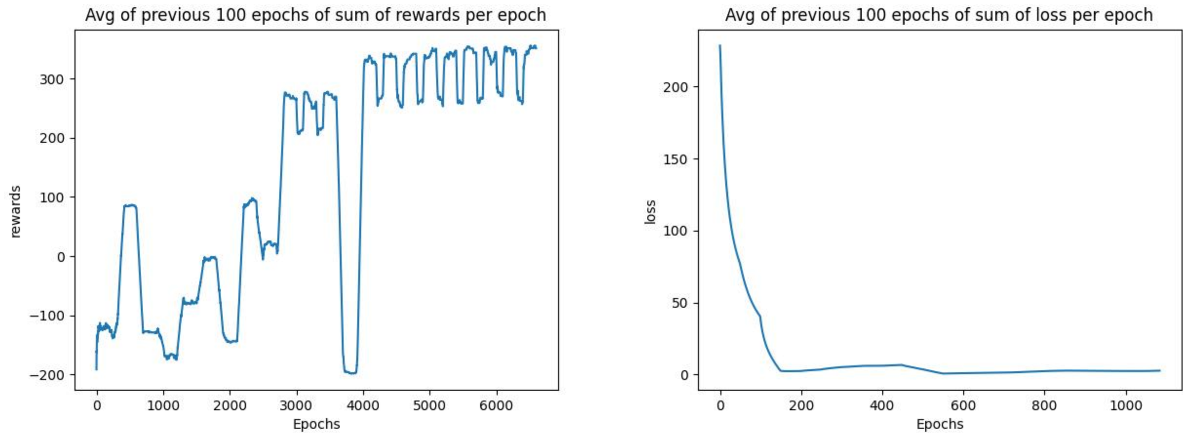
(a) Moving average of 100 previous total reward per epoch (b) Moving average of 100 previous total loss per epoch

Figure 5.4: Random1v1 plots



(a) Moving average of 100 previous total reward per epoch (b) Moving average of 100 previous total loss per epoch

Figure 5.5: Full1v1 plots



(a) Moving average of 100 previous total reward per epoch (b) Moving average of 100 previous total loss per epoch

Figure 5.6: Harvest plots

5.1.5 Win rate in PVP environments

To compare our results to existing research, we look at work from Şahin and Yücesoy [65], which also evaluates multiple agents in DeepRTS. To this end, we present win rates in *Simple1v1*, *Random1v1* and *Full1v1* environments. The win rate is evaluated by running 1000 episodes across three seeds, and recording how many times the agent defeated the opposite player. However, the values for our models and [65], as we use different environments. There can be many differences in configurations, such as different action spaces, observation spaces and reward functions. Nonetheless, The average win rate for DDQN, DT and GADT combined is slightly higher than the combined win rate of [65] by 0.15%.

Agent	Win Rate	Averages across experiments
DDQN Simple1v1 (Ours)	0.7	
DDQN Random1v1 (Ours)	0.78	
DDQN Full1v1 (Ours)	0.13	
DT Simple1v1 (Ours)	0.82	
DT Random1v1 (Ours)	0.94	**0.56**
DT Full1v1 (Ours)	0.28	
GADT Simple1v1 (Ours)	0.83	
GADT Random1v1 (Ours)	0.27	
GADT Full1v1 (Ours)	0.32	
PPO [65]	0.29	
Sunrise [65]	0.65	
Curl [65]	0.86	0.545
Rainbow [65]	0.71	
Rule-based Offensive [65]	0.46	
Rule-based Defensive [65]	0.3	

Table 5.4: Win rate for our own models versus [65], across different environments. Results are not directly comparable, as different environments with different objectives were used.

5.1.6 Harvest rate

Since *Harvest* does not have a PVP aspect, the best measure for performance is resources harvested. Table 5.5 shows results for harvest rates for DDQN, DT and GADT. The rate shows the average number of resources that is gathered per episode using the given model. To compare results, we looked at work from [9], which performs experiments where the objective is to gather as many resources as possible. However, the reported results are not enough to compare to our work, as the authors only report average rewards and not resources gathered. Furthermore, their experiments were done using different limits in episode length.

Agent	Harvest Rate
DDQN	1.2 ± 0.14
DT	19.53 ± 8.048
GADT	$\mathbf{71.99 \pm 6.402}$

Table 5.5: Harvest rate for DDQN, DT and GADT. Results were gathered from 1000 episodes across three seeds. Highest value is marked in **bold**.

5.1.7 Summary

Table 5.6 summarizes the performance of all agents in DeepRTS. Results show that DT outperforms random and DDQN agents in all four environments. While the data that DT is trained on comes from random and DDQN agents, it still increases the average reward substantially compared to them, which shows its ability to learn the best sequences from a dataset of both sub-optimal and high-quality data.

For our novel implementation GADT, it outperforms all agents in three of four environments. Even without a static dataset, it is able to build its own dataset and learn from it. However, GADT performs worse tha both DDQN and DT in *Random1v1*, and possible reasons for this are discussed in section 5.2.1.

Environment	Random	DDQN	DT	**GADT (Ours)**
Simple1v1	-47.93 ± 0.136	28.89 ± 1.758	46.12 ± 0.227	**63.20 ± 0.852**
Random1v1	-40.29 ± 0.498	54.37 ± 1.236	**64.04 ± 0.688**	-2.296 ± 0.886
Full1v1	18.17 ± 1.658	28.01 ± 0.902	73.53 ± 1.679	**123.3 ± 3.377**
Harvest	-147.1 ± 1.884	-295.6 ± 13.98	-44.01 ± 61.26	**280.5 ± 63.24**
Averages	-54.29	-46.08	34.92	**116.2**

Table 5.6: Highest reward in each of the four environments, regardless of dataset. Best results are marked in **bold**.

5.2 Discussion

From the presented results we see that both DT and GADT perform well in DeepRTS. This chapter is a discussion of the results shown in the previous section, including rationales for the results and possibilities for future work.

5.2.1 Genetic Algorithm Decision Transformer results

GADT shows good results on Simple1v1, Full1v1 and Harvest, while showing subpar results in Random1v1. The reason for this can be explained by understanding the genetic algorithm part of GADT, when the games that DT will learn on are being chosen, only the games with the highest total reward are used in the dataset. This creates a bias such that games where the enemy agent has made mistakes (and therefore given our agent a better reward) are chosen to train on. This makes the model learn what to do when the random agent does something bad, but the model will not learn what to do in other situations. This could be potentially solved by feeding poor data to the agent or by simply training on a bigger percentage of the data generated.

5.2.2 Data Dependency

DT as presented by Chen et al. [19] uses predefined datasets for RL environments like the Atari benchmark [10] and D4RL [28]. For DeepRTS we must rely on randomly generated data and data from pretrained RL agents, as human replay data would be too time consuming to record. However, we used this fact to implement GADT, which does not rely on any dataset before training starts, as it generates and improves the data by exploring the environment. This data dependency is both a blessing and a curse, depending on if existing datasets are available.

5.2.3 Decision Transformer scalability

DT has great scalability potential, but this also means a lot of data is required. Data also needs to show good examples of gameplay for the model to learn how to play optimally. DT cannot stitch multiple sequences together to learn from both, and requires exact data samples that achieves the desired reward. This means it performs worse with smaller datasets where the observation space has been less explored. This is shown with DT trained on just random data, where it the data quality is not enough to achieve optimal behaviour. Had there been more data in the random dataset, the chance that the dataset contains a sequence with high reward would increase, which the DT model can learn from. With our proposal of GADT, we can train agents using smaller datasets which saves resources and computing power.

5.2.4 Using DeepRTS for reinforcement learning research

DeepRTS is a high-performance RL environment that is customizable and easily accessible through Python bindings. As explained in section 3.1.4 DeepRTS outperforms state-of-the-

art RTS environments like MicroRTS, which seemed impressive and a perfect fit for this thesis. However, the environment has not been extensively tested for RL research, and several bugs caused unexpected behaviour during agent training. After fixing these issues as explained in section 4.1.4, we found DeepRTS to provide everything we needed for this thesis. High FPS in headless mode made it easy and fast to test reward functions, game mechanics, and how models reacted to changes in hyperparameters. In addition, we could create our own tasks and environments in DeepRTS due to its configurability, which meant we could train agents to learn specific parts of the game.

5.2.5 Hypothesis Review

Looking back at our hypotheses proposed in section 1.2.2, we can now evaluate them based on the results obtained from our experiments.

- **Hypothesis 1:** A DT agent will outperform a DDQN agent in terms of total reward in DeepRTS

  Our results in Table 5.6 confirm this hypothesis. DT consistently outperforms both random and DDQN agents in all four environments. The average total rewards achieved by DT are substantially higher compared to the other agents.

- **Hypothesis 2:** A DT agent will learn the best sequences of actions from a dataset with mixed policies for action generation, and outperform a DQN agent in terms of cumulative reward in DeepRTS

  According to table 5.3 DT learns best from good pretrained data and as (mentioned in the hypothesis above), will outperform DDQN.

- **Hypothesis 3:** Training a DT using only randomly generated actions will outperform a random agent in terms of reward

  This can be checked by comparing table 5.3 and table 5.6. It seems that although DT will consistently outperform a random agent in all four environments by training with random data, the improvement is negligible.

- **Hypothesis 4:** Gathering game data using the Genetic Algorithm and using the epsilon-greedy strategy will improve on Decision Transformer and allow an agent to explore the environment and avoid dependencies on labeled data

  According to table 5.3, GADT managed to outperform three out of the four environments, when compared to DT and DDQN.

- **Hypothesis 5:** Decision Transformer will be dependent on sequence length for performance during training

  In figure 5.2 the reward is almost doubled in all epochs by just using a higher sequence length as input to DT, this was highly expected, as the higher the sequence length, the more information the DT has action per prediction.

Overall, our hypotheses were largely confirmed positively by the experimental results. DT demonstrated its superiority over random and DDQN agents, while GADT showcased promising performance, outperforming other agents in most environments. However, DT struggles to learn from sub-optimal data, which was not predicted.

5.3 Future work

As we near the conclusion of our research, we present proposals to further improve agent performance, data quality and our DeepRTS environments.

5.3.1 Data quality

A lot of improvements could be made to the data used to train DT. Data was generated randomly and through a pretrained DDQN agent, and the DT model would improve a lot if the pretrained agent performed better in terms of reward. This would mean that the data that DT trains on contains games where the cumulative reward is higher, and therefore learn which actions result in the highest reward. Improving the pretrained agent boils down to improving its neural network and tuning hyperparameters, or changing the action space to make learning easier.

5.3.2 Data gathering

Another improvement is to increase dataset sizes. DT and other Transformer models require labeled data to learn specific policies, and DeepRTS does not have publicly available datasets for the game and as such, a lot of time was spent on generating and improving data to in turn improve the supervised training of the Transformer models. Furthermore, as a contribution to the DeepRTS research environment the data used in this work is made available open-source, see appendix A.1. We invite fellow AI researchers and gamers to contribute to data gathering by playing the game and training RL agents to play DeepRTS and recording games as labeled data, as this would greatly improve the corpus of data for this RL environment.

5.3.3 Increasing Action Space

DeepRTS supports more complex actions like selecting and issuing commands to units through left and right-clicking map tiles. These actions were omitted in this work, to keep the action space limited to single actions that affect the environment directly. However, using left and right-click actions could make the environment more interactive, as it allows for an agent to issue automatic commands like attacking a specific unit or harvesting a specific resource.

Macro actions

Implementing command actions could be done by introducing macro actions, where an action in the action space performs a sequence of actions. For example, a macro action "Build peasant" could be done by performing the following actions in order:

1. Select Town Hall (build Town Hall if not built)
2. Build Peasant
3. Select Peasant unit

Using the current DeepRTS framework, this sequence is more difficult to do as it requires sequential action-making, where the agent has to either select the correct tile for the Town Hall or cycle through each unit until it has targeted the Town Hall. Then, the agent has to select the correct build action, before selecting the peasant unit.

The agent could also cancel macro actions at any time with an empty action (see section 4.1.2. Otherwise, the macro actions would run until completed.

5.3.4 Multi-Agent

DeepRTS has the possibility of adding as many players as wanted into an environment. For future work, agents with different decision algorithms could be tested against each other. This would be interesting as it is ultimately a PVP match of two algorithms, to see which is

superior. The agents could be trained by playing against each other, or by playing against a copy of themselves to improve. As the agent is playing against a similarly intelligent agent, this experiment would be much harder for any agent than the environments we have introduced, and could make the agent learn how to strategize and react to the opposing player's actions.

A League system as explained in 3.3.1 which was implemented in AlphaStar could be implemented in DeepRTS to train agents against each other. While we have implemented agents using different algorithms for decision-making, we trained them using the same player instance in each environment. In PVP environments, the agent is trained by always controlling $player_0$, and is not able to play as $player_1$ unless trained to control this player. This is because players have different spawn positions at the start of an episode, and is visually distinguishable when using image data instead of the state representation.

5.3.5 Combining Agents

Our agents can play specific parts of DeepRTS well, but needs improvements in *Full1v1*. This environment is more complex because the agent has to learn two policies; constructing units at the start of the game and defeating the enemy after. This policy is difficult to learn, as rewards change based on which timestep the game is currently in. To solve this, we could train agents on specific parts within each game, for instance, having one agent focusing on military expansion at the start of the game, and an agent focusing on defeating the enemy towards the end of the game. In practice this could be done by training two neural networks, each using separate reward functions, and limiting which timesteps each network is used to predict actions.

Chapter 6

Conclusions

In conclusion, this thesis has presented a comprehensive exploration and evaluation of Transformer models applied in Real-Time Strategy game environments. The thesis utilized Deep-RTS as the reinforcement learning environment and implemented various sub-environments with different objectives and complexity levels to facilitate training speed and enable the comparison of deep learning algorithms.

In general, the experimental results aligned with the proposed hypotheses, validating the effectiveness of Transformer models, particularly Decision Transformer, in RTS game environments. The study also introduced the Genetic Algorithm Decision Transformer as a novel approach to data generation in reinforcement learning environments. These findings contribute to advancing the application of Transformer models in real-world scenarios, specifically in the context of reinforcement learning and game-based learning environments. Further research in this area holds promise for developing intelligent agents capable of learning and strategizing in complex real-time environments.

Appendix A

Appendices

A.1 Implementation

The implementation for *Mastering DeepRTS with Transformers* is available publically and open-source at https://github.com/lixado/Age-of-transformers. This also includes download links to datasets that were used for training the DT models.

A.2 DeepRTS Configurations

DeepRTS Game mechanics can be adjusted through various configurations. *Simple1v1* & *Random1v1* uses the default configurations as shown in table A.1, but *Full1v1* and *Harvest* has some alterations.

Option	**Value**
Spawn with Town Hall	False
Enable Build Instantly	True
Enable Auto Harvesting	True
Enable Auto Attack	True
Food Limit	100
Enable Farm	True
Enable Barrack	False
Enable Footman	True
Enable Archer	False
Gold	1500
Lumber	750
Stone	0
Max Episode Steps	50

Table A.1: Default DeepRTS config. Each environment uses these by default, unless specified otherwise.

A.2.1 *Full1v1*

Option	Value
Enable Auto Harvesting	False
Enable Barrack	True
Gold	5000
Lumber	5000
Stone	5000
Max Episode Steps	200

Table A.2: Config for *Full1v1*. Any options that are not specified is set to default, as shown in table A.1.

A.2.2 *Harvest*

Option	Value
Enable Auto Harvesting	False
Enable Footman	False
Gold	5000
Lumber	5000
Stone	5000
Max Episode Steps	200

Table A.3: Config for *Harvest*. Any options that are not specified are set to default, as shown in table A.1.

A.3 Training Configuration

A.3.1 DDQN

Hyperparameter	Value
Exploration Rate ϵ	1
Exploration Rate Decay	0.99999
Exploration Rate Minimum	0.001
Replay Buffer Size N	100 000
Batch Size	512
Discount Parameter γ	0.9
Learning Rate	0.00025
Burnin	10 000
Learn Every x Step	3
Sync Networks Every x Step	10 000

Table A.4: Default hyperparameters for DDQN.

Simple1v1

Simple1v1 uses the default parameters as shown in table A.4.

Random1v1

Random1v1 uses the default parameters as shown in table A.4, with the only alteration being the exploration rate decay = 0.99998.

Full1v1

Full1v1 uses the default parameters as shown in table A.4, with the only alteration being the exploration rate decay = 0.999997.

Harvest

Harvest uses the default parameters as shown in table A.4, with the only alteration being the exploration rate decay = 0.9999991.

A.3.2 Vanilla DT

Hyperparameter	**Value**
normalize	false
epochs	100
batchSize	2048
learning_rate	0.00025
TargetReturn	10

Table A.5: Default hyperparameters for Vanilla DT trained on a dataset.

Simple1v1

Simple1v1 uses the default parameters as shown in table A.5.

Random1v1

Random1v1 uses the default parameters as shown in table A.5.

Full1v1

Full1v1 uses the default parameters as shown in table A.5.

Harvest

Harvest uses the default parameters as shown in table A.5.

A.3.3 GADT

Hyperparameter	**Value**
normalize	false
stepsMax	50
batchSize	2048
exploration_rate	1
exploration_rate_decay	0.999
exploration_rate_min	0.001
learning_rate	0.00025
DTDataMaxSize	100
DTTrainEpochs	50
DTEpisodesGenerate	300
DTEpsilonRate	3
TargetReturn	10,000

Table A.6: Default hyperparameters for GADT.

Simple1v1

Simple1v1 uses the default parameters as shown in table A.6.

Random1v1

Random1v1 uses the default parameters as shown in table A.6.

Full1v1

Full1v1 uses the default parameters as shown in table A.6, with the only alteration being the stepsMax = 200.

Harvest

Harvest uses the default parameters as shown in table A.6, with the only alteration being the stepsMax = 200.

Bibliography

[1] *(5) Post | LinkedIn*. https://www.linkedin.com/posts/ingliguori_gpt1-gpt2-gpt3-activity-7028774382193774592-xdoj/?originalSubdomain=na. (Accessed on 05/06/2023).

[2] Synnaeve et al. *Torch - A scientific computing framework for LuaJIT*. URL: http://torch.ch/ (visited on 05/07/2023).

[3] *AlphaStar: Mastering the real-time strategy game StarCraft II*. https://www.deepmind.com/blog/alphastar-mastering-the-real-time-strategy-game-starcraft-ii. (Accessed on 05/26/2023).

[4] *AlphaStar: Mastering the real-time strategy game StarCraft II*. URL: https://www.deepmind.com/blog/alphastar-mastering-the-real-time-strategy-game-starcraft-ii (visited on 05/30/2023).

[5] Per Arne Andersen. *DeepRTS - A Real-Time-Strategy game for Deep Learning research*. URL: https://github.com/cair/deep-rts (visited on 04/09/2023).

[6] Per Arne Andersen. *Git commit - Updated state representation*. URL: https://github.com/cair/deep-rts/commit/8352f4a4a69d9347be692281ef5025b8b9207c1d (visited on 04/09/2023).

[7] Per Arne Andersen. Andersen, M. Goodwin, and O. Granmo. "Deep RTS: A Game Environment for Deep Reinforcement Learning in Real-Time Strategy Games." In: *2018 IEEE Conference on Computational Intelligence and Games (CIG)*. Aug. 2018, pp. 1–8. DOI: 10.1109/CIG.2018.8490409.

[8] Per-Arne Andersen, Morten Goodwin, and Ole-Christoffer Granmo. "Towards safe reinforcement-learning in industrial grid-warehousing." In: *Information Sciences* 537 (2020), pp. 467–484.

[9] Marco Antônio Silva Araújo. "Beyond Star: um modelo de arquitetura de aprendizado para generalização de estratégias em jogos RTS." MA thesis. Universidade Federal do Rio Grande do Norte, 2020.

[10] Marc G Bellemare et al. "The arcade learning environment: An evaluation platform for general agents." In: *Journal of Artificial Intelligence Research* 47 (2013), pp. 253–279.

[11] Richard Bellman. "A Markovian Decision Process." In: *Indiana Univ. Math. J.* 6 (4 1957), pp. 679–684. ISSN: 0022-2518.

[12] Dimitri P Bertsekas et al. "Dynamic programming and optimal control 3rd edition, volume ii." In: *Belmont, MA: Athena Scientific* (2011).

[13] Blizzard. *Starcraft II*. URL: https://starcraft2.com/en-us/ (visited on 04/09/2023).

[14] Blizzard. *World of Warcraft III*. URL: https://worldofwarcraft.blizzard.com/en-us/story/timeline/chapter-5 (visited on 04/09/2023).

[15] Greg Brockman et al. "Openai gym." In: *arXiv preprint arXiv:1606.01540* (2016).

[16] Tom Brown et al. "Language models are few-shot learners." In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[17] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL].

[18] Murray Campbell, A.Joseph Hoane, and Feng-hsiung Hsu. “Deep Blue.” In: *Artificial Intelligence* 134.1 (2002), pp. 57–83. ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(01)00129-1. URL: https://www.sciencedirect.com/science/article/pii/S0004370201001291.

[19] Lili Chen et al. “Decision transformer: Reinforcement learning via sequence modeling.” In: *Advances in neural information processing systems* 34 (2021), pp. 15084–15097.

[20] *chess - Google Search*. https://www.google.com/search?q=chess&sxsrf=APwXEdcAZuP90pFX98eQLH8L7v 1684850773012&source=lnms&tbm=isch&sa=X&ved=2ahUKEwiHqMyezov_AhWEjYsKHXDeA74Q_AUoAnoECAEQBA&biw=923&bih=941&dpr=1#imgrc=pF53nxXztftyyM. (Accessed on 05/23/2023).

[21] Dave Churchill. *CommandCenter: AI Bot for Broodwar and Starcraft II*. URL: https://github.com/davechurchill/commandcenter (visited on 05/07/2023).

[22] David Churchill, Zeming Lin, and Gabriel Synnaeve. “An analysis of model-based heuristic search techniques for StarCraft combat scenarios.” In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. Vol. 13. 2. 2017, pp. 8–14.

[23] Torch contributors. *Torch - A scientific computing framework for LuaJIT*. URL: http://torch.ch/ (visited on 05/07/2023).

[24] *Decision Transformer*. URL: https://huggingface.co/docs/transformers/model_doc/decision_transformer#transformers.DecisionTransformerModel (visited on 06/03/2023).

[25] Google DeepMind. *PySC2*. URL: https://www.deepmind.com/open-source/pysc2 (visited on 05/07/2023).

[26] Andreas Eike. *Git commit - Fixed stonecost on unit create*. URL: https://github.com/cair/deep-rts/pull/51/commits/80a5ead76a9a3db496c0f9172cf55f5549d3dcda (visited on 05/05/2023).

[27] Endgadget. *'StarCraft: Remastered' upgrades a real-time strategy classic*. URL: https://www.engadget.com/2017-03-26-starcraft-remastered.html (visited on 04/11/2023).

[28] Justin Fu et al. “D4rl: Datasets for deep data-driven reinforcement learning.” In: *arXiv preprint arXiv:2004.07219* (2020).

[29] Jonas Gehring et al. “Convolutional sequence to sequence learning.” In: *International conference on machine learning*. PMLR. 2017, pp. 1243–1252.

[30] *Go at Sensei's Library*. https://senseis.xmp.net/?Go. (Accessed on 05/24/2023).

[31] *Google's AI AlphaGo to take on world No 1 Lee Se-dol in live broadcast | Artificial intelligence (AI) | The Guardian*. https://www.theguardian.com/technology/2016/feb/05/google-ai-alphago-world-no-1-lee-se-dol-live-broadcast. (Accessed on 05/24/2023).

[32] Danijar Hafner et al. “Mastering atari with discrete world models.” In: *arXiv preprint arXiv:2010.02193* (2020).

[33] Danijar Hafner et al. “Mastering Diverse Domains through World Models.” In: *arXiv preprint arXiv:2301.04104* (2023).

[34] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[35] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-term Memory.” In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[36] Shengyi Huang et al. “Gym-μRTS: toward affordable full game real-time strategy games research with deep reinforcement learning.” In: *2021 IEEE Conference on Games (CoG)*. IEEE. 2021, pp. 1–8.

[37] *Introducing ChatGPT*. https://openai.com/blog/chatgpt. (Accessed on 05/01/2023).

[38] *Introducing LLaMA: A foundational, 65-billion-parameter language model*. https://ai.facebook.com/blog/large-language-model-llama-meta-ai/. (Accessed on 05/01/2023).

[39] Michael Janner, Qiyang Li, and Sergey Levine. "Offline reinforcement learning as one big sequence modeling problem." In: *Advances in neural information processing systems* 34 (2021), pp. 1273–1286.

[40] Lukasz Kaiser et al. "Model-based reinforcement learning for atari." In: *arXiv preprint arXiv:1903.00374* (2019).

[41] Maximilian Karl et al. "Deep variational bayes filters: Unsupervised learning of state space models from raw data." In: *arXiv preprint arXiv:1605.06432* (2016).

[42] *karpathy/minGPT: A minimal PyTorch re-implementation of the OpenAI GPT (Generative Pretrained Transformer) training*. URL: https://github.com/karpathy/minGPT (visited on 06/03/2023).

[43] Ilya Kostrikov, Denis Yarats, and Rob Fergus. "Image augmentation is all you need: Regularizing deep reinforcement learning from pixels." In: *arXiv preprint arXiv:2004.13649* (2020).

[44] Aviral Kumar et al. "Conservative q-learning for offline reinforcement learning." In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 1179–1191.

[45] Lambda Labs. *OpenAI's GPT-3 Language Model: A Technical Overview*. URL: https://lambdalabs.com/blog/demystifying-gpt-3 (visited on 05/21/2023).

[46] Michael Laskin, Aravind Srinivas, and Pieter Abbeel. "Curl: Contrastive unsupervised representations for reinforcement learning." In: *International Conference on Machine Learning*. PMLR. 2020, pp. 5639–5650.

[47] Michael Laskin et al. *In-context Reinforcement Learning with Algorithm Distillation*. 2022. arXiv: 2210.14215 [cs.LG].

[48] Wenzhe Li et al. *A Survey on Transformers in Reinforcement Learning*. 2023. arXiv: 2301.03044 [cs.LG].

[49] Zeming Lin et al. "Stardata: A starcraft ai research dataset." In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. Vol. 13. 1. 2017, pp. 50–56.

[50] Vincent Micheli, Eloi Alonso, and François Fleuret. "Transformers are sample efficient world models." In: *arXiv preprint arXiv:2209.00588* (2022).

[51] *Number of atoms in the universe - Oxford Education Blog*. https://educationblog.oup.com/secondary/maths/numbers-of-atoms-in-the-universe. (Accessed on 05/24/2023).

[52] Santiago Ontanón. *MicroRTS AI Competition*. URL: https://sites.google.com/site/micrortsaicompetition/ (visited on 05/07/2023).

[53] Santiago Ontanón. "The combinatorial multi-armed bandit problem and its application to real-time strategy games." In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. Vol. 9. 1. 2013, pp. 58–64.

[54] OpenAI. *GPT-4 Technical Report*. 2023. arXiv: 2303.08774 [cs.CL].

[55] OpenAI. *Gym Wrappers*. URL: https://www.gymlibrary.dev/api/wrappers/ (visited on 05/12/2023).

[56] Adam Paszke et al. "Pytorch: An imperative style, high-performance deep learning library." In: *Advances in neural information processing systems* 32 (2019).

[57] Peng Peng et al. "Multiagent bidirectionally-coordinated nets: Emergence of human-level coordination in learning to play starcraft combat games." In: *arXiv preprint arXiv:1703.10069* (2017).

[58] PlayerCounter. *League of Legends Player Count*. URL: https://playercounter.com/league-of-legends/ (visited on 05/07/2023).

[59] Martin L. Puterman. "Chapter 8 Markov decision processes." In: *Stochastic Models*. Vol. 2. Handbooks in Operations Research and Management Science. Elsevier, 1990, pp. 331–434. DOI: https://doi.org/10.1016/S0927-0507(05)80172-0. URL: https://www.sciencedirect.com/science/article/pii/S0927050705801720.

[60] Alec Radford et al. "Improving language understanding by generative pre-training." In: (2018).

[61] Alec Radford et al. "Language Models are Unsupervised Multitask Learners." In: *OpenAI Blog* (2019).

[62] Alec Radford et al. "Language models are unsupervised multitask learners." In: ().

[63] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors." In: *nature* 323.6088 (1986), pp. 533–536.

[64] Sreenath S. *Getting to Grips with Reinforcement Learning via Markov Decision Process*. URL: https://editor.analyticsvidhya.com/uploads/30519mdp.png (visited on 05/04/2023).

[65] Safa Onur Şahin and Veysel Yücesoy. "Reinforcement Learning Algorithms Performance Comparison on the Game of DeepRTS." In: *2021 29th Signal Processing and Communications Applications Conference (SIU)*. IEEE. 2021, pp. 1–4.

[66] Max Schwarzer et al. "Data-efficient reinforcement learning with self-predictive representations." In: *arXiv preprint arXiv:2007.05929* (2020).

[67] *Sebastian Raschka on Twitter: "A question that often comes up when introducing colleagues to the attention mechanism: how are attention scores different from weights in a fully-connected layer? Conceptually, attention mechanisms allow transformers to attend to different parts of a sequence or image. On the... https://t.co/XyF3DwUVrE" / Twitter*. https://twitter.com/rasbt/status/1629884953965068288/photo/1. (Accessed on 05/10/2023).

[68] Jinghuan Shang and Michael S Ryoo. "StARformer: Transformer with State-Action-Reward Representations." In: *arXiv preprint arXiv:2110.06206* (2021).

[69] Noam Shazeer et al. "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer." In: *arXiv preprint arXiv:1701.06538* (2017).

[70] Chathurangi Shyalika. *A Beginners Guide to Q-Learning. Model-Free Reinforcement Learning*. Nov. 2019. URL: https://towardsdatascience.com/a-beginners-guide-to-q-learning-c3e2a30a653c (visited on 05/01/2022).

[71] David Silver et al. "Mastering the game of Go with deep neural networks and tree search." In: *Nature* 529.7587 (Jan. 2016), pp. 484–489. ISSN: 1476-4687. DOI: 10.1038/nature16961. URL: https://doi.org/10.1038/nature16961.

[72] SteamDB. *Dota 2 Steam Charts · SteamDB*. URL: https://steamdb.info/app/570/charts/ (visited on 05/07/2023).

[73] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[74] Gabriel Synnaeve et al. "TorchCraft: a Library for Machine Learning Research on Real-Time Strategy Games." In: *arXiv preprint arXiv:1611.00625* (2016).

[75] *Tensor2Tensor Intro - Colaboratory*. https://colab.research.google.com/github/tensorflow/tensor2tensor/blob/master/tensor2tensor/notebooks/hello_t2t.ipynb#scrollTo=OJKU36QAfqOC. (Accessed on 05/06/2023).

[76] *The First Match of Deepmind Starcraft 2: AlphaStar (AI) vs TLO - YouTube*. https://www.youtube.com/watch?v=DpRPfidTjDA. (Accessed on 05/26/2023).

[77] Emanuel Todorov, Tom Erez, and Yuval Tassa. "MuJoCo: A physics engine for model-based control." In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2012, pp. 5026–5033. DOI: 10.1109/IROS.2012.6386109.

[78] Hugo Touvron et al. "Llama: Open and efficient foundation language models." In: *arXiv preprint arXiv:2302.13971* (2023).

[79] Aaron Van Den Oord, Oriol Vinyals, et al. "Neural discrete representation learning." In: *Advances in neural information processing systems* 30 (2017).

[80] Ashish Vaswani et al. *Attention Is All You Need*. 2017. arXiv: 1706.03762 [cs.CL].

[81] Ashish Vaswani et al. "Attention is all you need." In: *Advances in neural information processing systems* 30 (2017).

[82] Oriol Vinyals et al. "Grandmaster level in StarCraft II using multi-agent reinforcement learning." In: *Nature* 575.7782 (Nov. 2019), pp. 350–354. ISSN: 1476-4687. DOI: 10.1038/s41586-019-1724-z. URL: https://doi.org/10.1038/s41586-019-1724-z.

[83] Oriol Vinyals et al. "Starcraft ii: A new challenge for reinforcement learning." In: *arXiv preprint arXiv:1708.04782* (2017).

[84] Jakob Wenzel. *pybind11 — Seamless operability between C++11 and Python*. URL: https://github.com/pybind/pybind11 (visited on 02/17/2023).

[85] Yonghui Wu et al. "Google's neural machine translation system: Bridging the gap between human and machine translation." In: *arXiv preprint arXiv:1609.08144* (2016).

[86] Taku Yamagata, Ahmed Khalil, and Raul Santos-Rodriguez. "Q-learning decision transformer: Leveraging dynamic programming for conditional sequence modelling in offline rl." In: *arXiv preprint arXiv:2209.03993* (2022).