Recurrent Linear Transformers for Reinforcement Learning

by

Subhojeet Pramanik

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

The transformer architecture is effective in processing sequential data, both because of its ability to leverage parallelism, and because of its self-attention mechanism capable of capturing long-range dependencies. However, the self-attention mechanism is slow for streaming data, that is when the input sequence is received one at a time. This limits its application in sequential decision-making problems such as online reinforcement learning. The self-attention mechanism requires past activations, that is the history, to be provided as context. As such, the inference cost, the cost of applying self-attention to a single element in a sequence, depends on the length of the input context. Increasing the context length of self-attention directly increases the inference cost. In this thesis, we present recurrent alternatives to the transformer self-attention that offer a context-independent inference cost, while also leveraging long-range dependencies. Our approaches are called the Recurrent Linear Transformer (ReLiT) and Approximate Recurrent Linear Transformer (AReLiT). We evaluate them on T-Maze, a partially observable reinforcement learning task that requires long-term memory, demonstrating their effectiveness when compared to existing RNN and transformer baselines. Additionally, we provide results in Memory Maze, a 3D pixel-based environment, and we empirically demonstrate the computational efficiency of our approach compared to standard transformer architectures.

I think; therefore I am.

– René Descartes.

To Maa and Baba, for their unwavering support and encouragement in all my endeavours.

Acknowledgements

Firstly, I would like to thank my advisors Adam White and Marlos C. Machado. Their constant support and feedback has been invaluable throughout my journey as a master's student. Their unwavering dedication to my growth and development as a scholar has been a cornerstone of my academic progress. Their guidance has not only enriched my knowledge but also instilled in me a sense of curiosity, principled inquiry, and meticulousness that I carry forward in my research pursuits. I would also like to thank Prof. Dale Schrumanns for agreeing to be a part of my thesis committee and providing valuable feedback. Next, I would like to thank my collaborator Esraa Elilemy. She has played a significant role in this project through collaborative discussions and experimental work.

I'm deeply appreciative of the sense of community, the engaging discussions, and the guidance I've received from my fellow researchers in the Reinforcement Learning and Artificial Intelligence Lab during my time as a graduate student at the University of Alberta. I would like to thank Prof. Michael Bowling for his mentorship and guidance throughout my time as a graduate student. I would like to thank Prof. Martha White for providing additional computational resources for my research. I would like to thank Prof. Richard Sutton for always being a role model and an inspiration. I would also like to thank my fellow graduate students, especially those who have been a part of the lab for a long time, for their support and encouragement. To name a few, I would like to thank Khurram Javed, Matthew Schelegel, Haseeb Shah, Edan Meyer, Jordan Coblin, Erfan Miahi, Esraa Saleh, Andrew Patterson, Shivam Garg, Gautham Vasan, Jiamin He, Justin Stevens and many others. I would like to thank my friends and family for their constant support and encouragement. Finally, I would like to thank my parents for their unwavering support and encouragement throughout my academic journey.

Contents

Abstract **ii**

Acknowledgements **v**

List of Tables **ix**

List of Figures **x**

List of Algorithms **xiii**

List of Notations **xiv**

1 Introduction **1**

1.1 Thesis Contributions 4

1.2 Thesis Outline 4

2 Background **5**

2.1 Reinforcement Learning 5

2.2 Policy Gradient Methods 6

2.2.1 Policy Gradient Theorem 6

2.2.2 Actor-Critic Algorithms 7

2.3 Partially Observable Reinforcement Learning 9

2.3.1 Recurrent Neural Networks for Dealing with Partial Observability 9

2.4 Transformers in Supervised Settings 11

2.4.1 Canonical Transformer 11
2.4.2 Transformer-XL 13
2.5 Transformers in Reinforcement Learning 14
2.5.1 Transformer-XL Self-Attention for Streaming Data 15
2.5.2 Gated Transformer-XL 17
2.6 Recurrent Attention with Linear Transformers 19
2.6.1 Limitations 19

3 Recurrent Linear Transformer **21**
3.1 Gating Mechanism to Control the Flow of Information 21
3.2 Learnable Feature Map for Self-Attention 23
3.3 Recurrent Linear Transformer (ReLiT) 24
3.3.1 Limitations 26

4 Approximate Recurrent Linear Transformer **27**
4.1 Recurrent Approximation of Memory Matrix in ReLiT 27
4.1.1 Approximation of Kronecker Delta Function 28
4.1.2 Derivation of the Approximation 30
4.2 Approximate Recurrent Linear Transformer (AReLiT) 34
4.3 Effect of r on the Quality of Approximation 36
4.4 Parallelization during Training 37

5 Results **39**
5.1 Learning to Remember with Binary Observations 39
5.1.1 The T-Maze Environment 39
5.1.2 Experiment Setup 41
5.1.3 Experiment 1: Evaluating GTrXL in T-Maze 42
5.1.4 Experiment 2: Evaluating ReLiT and AReLiT in T-Maze 43
5.2 Learning to Remember with Pixel Observations 45
5.2.1 The Memory Maze Environment 45

5.2.2 Experiment Setup 46

5.2.3 Experiment 3: Evaluating AReLiT in Memory Maze 47

5.2.4 Experiment 4: Evaluating Impact of GTrXL's Context in Memory Maze . . . 49

6 Conclusion and Future Work 51

A Additional Experiment Details 58

A.1 T-Maze Experiments 58

A.2 Memory Maze Experiments 60

List of Tables

3.1 Space and time complexity of ReLiT, Linear Transformer, and GTrXL self-attention for processing a single element in a streaming sequence. M: memory size in GTrXL, d: representation dimension, d_k feature map dimension in Linear Transformer, η: feature map hyperparameter in ReLiT, L: number of encoder layers 26

4.1 Space and time complexity of AReLiT, ReLiT, Linear Transformer and GTrXL for a processing a single element in streaming sequence. (M: memory size in GTrXL, d: representation dimension, d_k feature map dimension in Linear Transformer, η: feature map hyperparameter in ReLiT and AReLiT, r: approximation parameter in AReLiT, L: number of encoder layers) 36

5.1 Number of floating point operations and space required by the self-attention layer to process a single observation in T-Maze. The values are calculated for a single head. **Operations** represents roughly the number of floating point operations. **Space** represents the number of floating point values required to store the previous state of the self-attention layer. 44

A.1 Hyperparameters and sweeps for the T-Maze experiments. 58

A.2 Architecture configuration for LSTM, GRU, GTrXL, ReLIT and AReLiT for T-Maze experiments. 59

A.3 Hyperparameters and sweeps for Memory Maze experiments. 60

A.4 Architecture configuration for LSTM, GTrXL and AReLiT for Memory Maze experiments. 61

List of Figures

2.1 An RNN is used to encode the sequence of observations in an actor-critic algorithm. The RNN is updated using the sequence of observations and actions collected by the agent during the interaction phase. The output of the RNN is an approximate state representation $\tilde{S}_t$, which is used as the input to the actor and critic heads. The actor and critic heads are neural networks that output the policy and value function respectively. 10

2.2 The transformer architecture. The left side shows the encoder-only transformer architecture. The encoder layer consists of a multi-head self-attention layer, a feed-forward neural network, residual connections and layer normalization. Multiple encoder layers are stacked to form the transformer architecture. The right side shows the multi-head self-attention layer. The multi-head self-attention layer consists of multiple self-attention heads operating in parallel. The output of each head is concatenated and linearly transformed to obtain the final output of the self-attention layer. 12

2.3 The GTrXL architecture used as a function approximator to learn an actor and a critic in an actor-critic algorithm. The GTrXL architecture is applied sequentially to the input sequence. The recurrent state is carried forward in subsequent timesteps. Additionally, the GTrXL architecture rearranges the layer-norm and introduces a gating mechanism to stabilize the training of the self-attention layer. 18

4.1 Error in approximating the current state $\mathbf{C}_t$ for different values r and gating at $t = 100$ for randomly sampled values and keys. 37

5.1 The T-Maze environment. The agent has to remember a binary cue (denoted by green text), shown only at the beginning of the episode, in order to take the correct turn at the intersection and receive a positive reward. The figure shows two possible episodes and the optimal path an agent must take. The agent's current location is provided as gray code encoding in the observation, along with distractor signals. The corridor length could be varied to increase the difficulty of the problem. 40

5.2 Mean success rate over 50 runs for a GTrXL agent with memory sizes 128 and 256 in the T-Maze environment. The shaded region represents the standard error. Both agents have a theoretical context larger than the largest corridor length considered but GTrXL-256 has access to the entire context as input. GTrXL-128 fails on corridor lengths above 120 despite having a theoretical context length of 512. GTrXL needs access to the entire context in order to learn long-term dependencies. 42

5.3 Mean success rate over 50 runs for ReLiT, AReLiT, LSTM, GRU, and GTrXL-256 agents in the T-Maze problem environment. The shaded region represents the standard error. 44

5.4 The Memory Maze environment. On the left, we show a possible maze layout for all four Memory Maze configurations. The maze layout is randomized at each episode. On the right, we show two sample observations that the agent receives. The agent's observation at each time-step is 64×64 RGB pixels and the action space is discrete. The border color of the observation image indicates the target object color which the agent needs to find to receive a reward. After collecting the object, the border color changes, indicating the next target object. The episode lengths are fixed depending on the Memory Maze configuration, with larger configurations having longer episodes. 45

5.5 Learning curves of LSTM, GTrXL and AReLiT agents in the Memory Maze environment. The x-axis represents the number of environment steps, and the y-axis represents the total reward in an episode. Each agent is trained with 3 different random seeds. The bold lines represent the mean return across the 3 seeds, and the blurred lines represent the individual seeds. Each point is the average episodic reward over 1M environment steps. The dotted grey line represents the performance of an oracle agent that has access to the entire maze layout, target object locations and paths leading to them. 48

5.6 GPU memory usage and frames per second (FPS) comparison for GTrXL and AReLiT agents in the Memory Maze environment. The data is collected from 12 independent agents per architecture, each trained for approximately 2.5 days in different nodes, utilizing the same configuration. 49

5.7 Learning curves of GTrXL agents with different memory sizes in the Memory Maze 13×13 environment. The x-axis represents the number of environment steps, and the y-axis represents the total reward in an episode. Each agent is trained with 3 different random seeds. The bold lines represent the mean return across the 3 seeds, and the blurred lines represent the individual seeds. Each point is the average episodic reward over 1M environment steps. 50

List of Algorithms

1 Canonical Self-Attention (Batch Data) 12
2 Transformer-XL Self-Attention (Batch Data) 14
3 Transformer-XL Self-Attention (Streaming Data) 16
4 Linear Transformer Self-Attention (Streaming Data) 20
5 Recurrent Linear Transformer (ReLiT) Self-Attention (Streaming Data) 25
6 Approximate Recurrent Linear Transformer (AReLiT) Self-Attention (Streaming Data) 35

List of Notations

Notation	Description
S_t	State at time t
A_t	Action at time t
R_t	Reward at time t
γ	Discount factor
π	Policy
θ	Parameters of the actor-critic network
$\phi()$	Feature map used of the Linear Transformer self-attention
d_{in}	Dimension of each individual observation
d	Representation dimension for an RNN or a transformer
d_{hid}	Dimension of the hidden state of an RNN
d_h	Head dimension, output dimension of the self-attention layer
d_k	Output dimension of the kernel feature ϕ in the Linear Transformer self-attention
N	Length of the sequence input to the self-attention layer
M	Number of stored activations in the GTrXL self-attention layer
L	Number of layers in the transformer architecture
η	Hyperparameter controlling the output dimension of the learnable feature map in ReLiT self-attention
r	Hyperparameter controlling the quality of approximation in AReLiT self-attention
$\mathbf{X}$	Batch input passed to the entire transformer or a single self-attention layer
$\mathbf{x}_t$	Single input to the self-attention layer or RNN at time t
$\mathbf{W}_Q$	Query parameters
$\mathbf{W}_K$	Key parameters
$\mathbf{W}_V$	Value parameters

$\mathbf{W}_{p1}, \mathbf{W}_{p2}, \mathbf{W}_{p3}$	Projection parameters for the learnable feature map
$\mathit{cat}()$	Concatenation operation across the first dimension
$\mathit{split}(\mathbf{X}, N_s)$	Splits a tensor $\mathbf{X}$ into N_s chunks across the first dimension
$\mathit{softmax}()$	Softmax operation across the last dimension

Chapter 1

Introduction

The Reinforcement Learning (RL) framework comprises an agent that interacts with an environment through actions. At each time step, the agent is in a some state of the environment. Upon taking an action, the agent transitions to a new state and receives a reward. The goal of the agent is to learn a policy that maximizes the cumulative reward. This learning often happens through trial and error.

In RL, the true state of the environment is often not completely accessible to the agent, and the agent must learn to infer the state from the observations it receives. Such settings are collectively referred to as the *partially observable RL setting.* As an example, consider an agent that is learning to drive a car, and to take the correct turn at an intersection it needs to keep track of a road sign it saw a few minutes back. Since the agent cannot directly observe the road sign through its current sensory observations (e.g. a camera feed), it needs to memorize the history of interactions with the environment. A naive approach would be to simply store the entire history of observations, and use a function approximator such as a neural network to learn a policy. However, such an approach is not scalable as the history of observations can be longer than the memory available to the agent. Alternatively, the agent can learn a compressed representation of the history of observations, and use it to make decisions. Naturally, it becomes important that these representations are learned in a computationally efficient manner.

Recurrent neural network (RNN) architectures provide a framework for learning such representations due to their ability to automatically learn relationships about the past. RNNs, such as LSTMs (Hochreiter and Schmidhuber [1997]) and GRUs (Gao and Glowacka [2016]), handle sequential data by maintaining a vector of hidden states that capture short-term dependencies between consecutive elements in the sequence. RNNs have been applied to a wide range of partially observable RL environments such as Atari 2600 games (Hausknecht and Stone [2015]). Importantly, RNNs offer fast inference, as the computational complexity of processing a single element in the sequence is independent of the length of the sequence. However, empirically, RNNs such as LSTMs trained with backpropagation through time often fail to capture long-range dependencies (Khandelwal et al. [2018], Bakker [2001]); and due to their sequential nature, it is difficult to parallelize the computation over the input sequence.

The Transformer architecture (Vaswani et al. [2017]), originally designed for sequential data processing in the supervised learning setting, is an alternative to RNNs. Transformers have been successfully applied to domains such as natural language processing (e.g., Brown et al. [2020], Devlin et al. [2018]) and computer vision (e.g., Petit et al. [2021], Zhong et al. [2020]). These successes are often attributed to the transformer's self-attention mechanism. The self-attention mechanism can capture long-range dependencies and its computation is parallelizable over an input sequence. Unlike RNNs, the transformer does not maintain a hidden state, it instead processes the entire sequence in parallel. The self-attention mechanism utilizes a dot product coupled with a softmax function to learn relationships between different elements in the sequence.

Nevertheless, the transformer architecture has some limitations. First, the context length of the transformer's self-attention, which is the number of elements in the sequence that it can recall, is limited by the length of the input sequence that can be processed for a given computational budget. Further, the transformer architecture is slow for streaming data, that is when the input sequence is presented sequentially. The inference cost, that is the computational complexity of applying self-attention for a single element in the sequence, increases with the increasing context. This is in contrast to RNNs, where the inference cost does not depend on the input sequence length. As such, increasing context length and reducing the computational complexity of self-attention remains a major research topic (e.g. Dai et al. [2019], Choromanski et al. [2020], Bulatov et al. [2022]).

The issues around context length and inference cost are particularly problematic in RL. A slow inference step impacts the rate at which an agent can interact with the environment, and if the environment is partially observable, a limited context length directly impacts the agent's memory. Even relatively simple reinforcement learning problems can consist of an agent interacting with the environment for hundreds of millions of steps in episodes that are 100,000 steps long (Nair et al. [2015], Machado et al. [2018]). These numbers are already much larger than what most transformer systems can process. As the data is often processed sequentially, and the agent needs to make decisions based on the data it has seen so far. This is in contrast to natural language processing, which transformers were originally designed for. The issues around context and inference are present only during deployment and not during training. In such systems, the input sequence is known a priori during the training phase and sequences are often short; the transformer can be trained to process the entire sequence in parallel.

In this thesis, we combine the strengths of transformers and RNNs to significantly improve learning in partially observable RL problems. Our work introduces an alternative approach that can uncover relationships far in the past, both in theory and practice. It is amenable to sequential computation with a context-independent inference cost, but it is highly parallelizable if need be.

Naturally, previous work have explored the relationship between transformers and RNNs. Particularly, to reduce the computational complexity of the transformer's self-attention mechanism, Katharopoulos et al. [2020] proposed the Linear Transformer architecture which replaces the softmax function with a generic kernel function. The approach is equivalent to RNNs because the proposed self-attention function can also be calculated sequentially, without requiring the past sequence to be presented as a context. Similar to an RNN, the Linear Transformer's self-attention has a context-independent inference cost.

However, the self-attention in the Linear Transformer architecture has certain limitations. First, the Linear Transformer's self-attention mechanism naively adds positive values to the recurrent state. It can grow arbitrarily large, causing the self-attention mechanism to become unstable with increasing sequence lengths. Such a limitation prevents us from applying Linear Transformers to long sequences, where removing past data from the recurrent state becomes crucial to make room for new data. Second, the Linear Transformer's self-attention replaces the softmax with a feature map, the choice of which can significantly impact the performance of the architecture. Element-wise feature maps such as the ones used in the original Linear Transformer architecture have limited memory capacity (Schlag et al. [2021]). Lastly, the Linear Transformer's self-attention mechanism maintains a matrix as a recurrent state. Transformer architectures typically use multiple self-attention heads to capture different relationships between elements in the sequence and having a matrix as a recurrent state for each head has a high memory cost.

In this thesis, we present two approaches that extend the Linear Transformer self-attention mechanism. Our first approach, called the Recurrent Linear Transformer (ReLiT), uses the gated structure of the Gated Transformer-XL architecture (Parisotto et al. [2020]) and introduces a modified Linear Transformer self-attention, addressing issues around the recurrent state and feature map. Our second approach, called Approximate Recurrent Linear Transformer (AReLiT), introduces an approximate version of ReLiT's self-attention, eliminating the need to maintain a matrix as a recurrent state. Unlike transformers, ReLiT and AReLiT have a context-independent inference cost and do not require the past sequence to be presented as a context. Unlike RNNs, ReLiT and AReLiT are parallelizable over an input sequence.

We evaluate both approaches on partially observable RL problems and compare them to existing RNN and transformer baselines. We start with the T-Maze problem setting (Bakker [2001]), designed to test an agent's ability to remember information for long durations. The T-Maze environment features binary observations and the goal is to remember a single piece of information from several timesteps ago. We show that limiting the input context of the canonical self-attention mechanism has a detrimental effect on performance and that a large input context, albeit at the cost of increased computational complexity, is necessary for this task. We show that both ReLiT and AReLiT outperform LSTMs and GRUs, and match the performance of a much more computationally expensive transformer architecture. We then extend these results to a larger problem setting called Memory Maze (Pašukonis et al. [2023]), featuring pixel observations and multiple sources of partial observability. We find that the performance of AReLiT is close to that of LSTMs and of vanilla transformer in this environment. We highlight the computational advantages of our approaches compared to a transformer agent by empirically measuring the frames per second (FPS) and GPU memory usage.

1.1 Thesis Contributions

The main contributions of this thesis are highlighted below:

- The Recurrent Linear Transformer (ReLiT) architecture, which couples a gated structure of the GTrXL architecture with a modified Linear Transformer self-attention mechanism. It introduces the following modifications to the Linear Transformer's self-attention mechanism:
  - A Gating mechanism to control the flow of past information, which allows the architecture to process long sequences.
  - A parameterized kernel feature map that allows the architecture to learn the feature map from data.
- The Approximate Recurrent Linear Transformer (AReLiT) architecture, which is an approximate version of the ReLiT architecture. It reduces the computational complexity of ReLiT in order of the embedding dimension by approximating the outer product calculation in the ReLiT architecture and replaces a matrix recurrent state in ReLiT with a finite set of vectors.

1.2 Thesis Outline

The rest of the thesis is organized as follows. In Chapter 2 we provide a brief background on the reinforcement learning problem setting and the transformer architecture. In Chapter 3 we introduce the Recurrent Linear Transformer (ReLiT) architecture. In Chapter 4 we introduce the Approximate Recurrent Linear Transformer (AReLiT) architecture. In Chapter 5 we empirically evaluate the proposed architectures in partially observable RL problems. Finally, in Chapter 6 we conclude the thesis and discuss future directions.

Chapter 2

Background

In this chapter, we provide a brief overview of the background knowledge that is relevant to this thesis. We start by introducing the reinforcement learning (RL) framework, and then we discuss the partially observable RL setting, which will be the problem setting for this thesis. Next, we introduce the actor-critic methods, which are the class of RL algorithms that we will use for our experiments. We discuss how RNNs can be combined with actor-critic methods to solve partially observable RL problems. We then introduce the background required to understand our proposed transformer approach. We start by introducing transformer architectures in supervised learning setting, and then introduce the modifications required to adapt transformers to the RL setting. Finally, we discuss the challenges in the current transformer approaches in the RL setting and introduce the Linear Transformer approach, which is the basis of our proposed approach.

2.1 Reinforcement Learning

Reinforcement learning is a problem formulation for tackling sequential decision-making problems, where an agent learns through interaction with the environment. In the RL framework, an intelligent agent interacts with an environment by taking actions, which potentially alter the environmental state. Upon taking an action in some state, the agent receives a scalar reward, and the environment transitions to a new state. The agent's goal is to maximize the sum of rewards it receives over time.

Formally, the interaction between the agent and its environment is represented as a finite Markov Decision Process (MDP). An MDP can be described as a tuple denoted by $(\mathcal{S}, \mathcal{A}, \mathcal{R}, d_0, r, p, \gamma)$. Here, $\mathcal{S}$ represents the set of all possible environmental states, and the initial state for the MDP is selected according to the starting state distribution d_0. The set $\mathcal{A}$ comprises all feasible actions. The set $\mathcal{R} \subset \mathbb{R}$ comprises all possible rewards. The reward function, denoted by $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathcal{R}$, quantifies the reward obtained when the agent takes action a in state s and transitions to state s'. The state transition dynamics, $p : \mathcal{S} \times \mathcal{A} \to \triangle(\mathcal{S})$, describe the probability of moving from one state to another after taking a specific action. Additionally, the discount factor $\gamma \in [0, 1]$ is introduced to weigh the importance of delayed rewards compared to immediate ones.

The agent-environment interaction unfolds in discrete time steps. A policy defined as $\pi : \mathcal{S} \to \triangle(\mathcal{A})$, is a mapping from states to probabilities of each possible action. At each time step t, the agent finds itself in a state $S_t \in \mathcal{S}$. Based on this state, the agent selects an action $A_t \in \mathcal{A}$ according to some policy π. Once the agent takes action A_t, the environment responds by transitioning to a new state S_{t+1} according to the state transition dynamics p and providing the agent with a reward R_{t+1} according to the reward function r. The return denoted by G_t is defined as the discounted sum of rewards obtained after time step t: $G_t \doteq \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$.

The goal of the agent is to learn a policy that maximizes the expected return over the state and actions. The expected return of being in some state $s \in \mathcal{S}$ under a policy π is characterized by the state-value function defined as $v_\pi(s) \doteq \mathbb{E}_\pi [G_t | S_t = s]$, where $\mathbb{E}_\pi$ denotes the expectation over the state-action distribution induced by the policy π. Additionally, the action-value function $q_\pi(s, a)$ is defined as the expected return of taking action a in state s and then following policy π thereafter: $q_\pi(s, a) \doteq \mathbb{E}_\pi [G_t | S_t = s, A_t = a]$. The goal of the agent is to then learn a policy that has an optimal state-value or action-value function. The optimal state-value function $v_*(s)$ is defined as the maximum state-value function over all policies: $v_*(s) \doteq \max_\pi v_\pi(s)$. Similarly, the optimal action-value function $q_*(s, a)$ is defined as the maximum action-value function over all policies: $q_*(s, a) \doteq \max_\pi q_\pi(s, a)$. The optimal policy π_* is then defined as the policy that maximizes the expected return: $\pi_*(s) \doteq \arg\max_\pi v_\pi(s)$.

2.2 Policy Gradient Methods

In this section, we briefly introduce some policy gradient algorithms. We start by introducing the policy gradient theorem, which is the basis for policy gradient methods. We discuss the REINFORCE algorithm, which is a simple policy gradient algorithm. Finally, we discuss actor-critic algorithms, which we will use as the baseline algorithms for our experiments.

2.2.1 Policy Gradient Theorem

The policy gradient theorem provides a way to learn parameterized policies in the RL setting. It states that the gradient of the expected return with respect to the policy parameters can be expressed as an expectation over the state-action distribution induced by the policy (Sutton et al. [1999]). Let π_θ denote a policy that is parameterized by θ and d_{π_θ} denote the state distribution induced by the policy π_θ. In the episodic case, the measure of performance could be characterized in terms of the policy parameters θ as: $J(\theta) \doteq \mathbb{E}_{S_t \sim d_{\pi_\theta}, A_t \sim \pi_\theta} [G_t]$. The policy gradient theorem provides a way to represent the gradient of the performance measure J with respect to the policy parameters θ and can be expressed as follows:

$$\nabla_\theta J(\theta) = \mathbb{E}_{S_t \sim d_{\pi_\theta}} \left[\sum_{a \in \mathcal{A}} q_\pi(S_t, a) \nabla_\theta \pi_\theta(a|S_t) \right] \tag{2.1}$$

2.2.2 Actor-Critic Algorithms

Actor-critic algorithms are a class of policy gradient algorithms that use a critic network to estimate the return G_t (Sutton et al. [1999]). The actor-critic algorithms consists of two parts. The actor, π_θ, is a learned parameterized policy. The critic is a value function, v_ϕ, trained to predict the return G_t from the current state S_t, typically a parameterized function learned using a Temporal Difference (TD) learning algorithm.

In one-step actor-critic, the Temporal Difference (TD) error, computed as $R_{t+1} + \gamma v_\phi(S_{t+1}) - v_\phi(S_t)$, are used for estimating advantages; these estimated advantages are then employed in formulating the loss function for both the actor and critic networks. The critic network is trained to minimize the TD error, which is the difference between the predicted return and the actual return. The loss function for the critic network is minimized using stochastic gradient descent and can be expressed as follows:

$$\mathcal{L}(\phi) = \mathbb{E}_{S_t \sim d_{\pi_\theta}, A_t \sim \pi_\theta} \left[(R_{t+1} + \gamma v_\phi(S_{t+1}) - v_\phi(S_t))^2 \right] \tag{2.2}$$

The actor network is trained to maximize the expected return, weighted by the TD error. The loss function for the actor network is the negative log-likelihood of the action taken, weighted by the advantage estimate:

$$\mathcal{L}(\theta) = -\mathbb{E}_{S_t \sim d_{\pi_\theta}, A_t \sim \pi_\theta} \left[(R_{t+1} + \gamma v_\phi(S_{t+1}) - v_\phi(S_t)) \log \pi_\theta(A_t|S_t) \right] \tag{2.3}$$

In this thesis, we utilize the actor-critic algorithms A2C and PPO for our experiments. Advantage Actor Critic (A2C) (Mnih et al. [2016]) and Proximal Policy Optimization (PPO) (Schulman et al. [2017]) are both actor-critic algorithms that have been shown to be effective for learning policies with neural network based function approximators. We briefly introduce the algorithms here. We refer the reader to the original papers for more details.

Advantage Actor Critic (A2C)

Adavantage Actor Critic (A2C) (Wu et al. [2017]) is a policy gradient algorithm that uses neural network based function approximation to learn an actor and a critic. A2C is a synchronous implementation of the A3C algorithm (Mnih et al. [2016]) that collects a batch of data by interacting with multiple environment instances in parallel and then synchronously updates the actor and the critic network parameters. The A2C algorithm alternates between update and interaction phases. During the interaction phase, the agent samples experiences (observations, actions, rewards) across multiple environment instances in parallel. Each environment is initialized with a different seed to ensure that the experiences collected are independent. The update phase is performed using stochastic gradient descent to update the actor and the critic network parameters. Unlike the TD error used in one-step actor-critic, A2C uses generalized advantage estimation (GAE) (Schulman et al. [2016]) to estimate advantages. The gradients are computed using the batch of trajectories collected in the interaction phase. Entropy regularization is used to encourage exploration.

Asynchronous Proximal Policy Optimization (Async-PPO)

The Proximal Policy Optimization (PPO) (Schulman et al. [2017]) algorithm is an on-policy algorithm that approximates the trust-region-based policy optimization problem by clipping the probability ratio between the new and old policies. Similar to A2C, PPO uses a neural network based function approximator to learn an actor and a critic. The PPO algorithm introduces a surrogate objective function that is optimized using stochastic gradient descent. The surrogate objective function is a clipped version of the policy gradient objective function. This clipping ensures that the new policy does not deviate too far from the old policy, thereby preventing the agent from taking actions that are too different from the actions it has taken in the past. Similar to A2C, PPO uses GAE to calculate the advantages, and entropy regularization to encourage exploration. The gradient updates are performed using multiple epochs of minibatch gradient descent over a batch of trajectories collected by the agent.

The asynchronous implementation of PPO (Petrenko et al. [2020]) involves running multiple agents in parallel, each interacting with its own copy of the environment. These agents asynchronously collect experience data and share their experience with a central learner process, which updates the policy parameters. This approach allows the learner to utilize the experience data more efficiently, thereby reducing the training time. The actors and learners communicate via a shared memory queue. The learner process is responsible for updating the policy and value network parameters and sending the updated policy parameters to the actors. The actors, on the other hand, are responsible for collecting experience data and sending it to the learner.

2.3 Partially Observable Reinforcement Learning

In the MDP framework it is assumed that the environment state S_t at each time step t is fully observable to the agent. However, in many real-world applications, the agent may not have access to the complete state of the environment. Instead, the agent may only receive a partial observation $O_t \in \mathcal{O}$ of the environment state S_t, where $\mathcal{O}$ is the set of all observations. In this case, the agent must learn to infer the underlying state of the environment from the partial observations it receives. This setting is known as partially observable reinforcement learning (RL).

Since the underlying state is not directly available, learning a policy to predict actions from the state is not possible. Instead, the agent then uses the history of interactions $H_t \doteq A_0, O_1, \ldots, A_{t-1}, O_t$ as a proxy for the state. The agent then learns a policy $\pi(\cdot|H_t)$ that maps the history of interactions to action distributions. The history of interaction is large and grows with time t. Alternatively, the agent can learn a summary of the history of interactions, which can be used to predict the next action.

2.3.1 Recurrent Neural Networks for Dealing with Partial Observability

In the partially observable RL setting, recurrent neural networks (RNN) provide a powerful framework for learning a summary of the history through data. RNNs are a class of neural networks that can process sequential data by maintaining a hidden state. The hidden state of the RNN is updated at each time step, and the hidden state at the current time step is a function of the hidden state at the previous time step and the current input. The hidden state of the RNN can be used as a summary of the history of interactions. For some input vector $\mathbf{x}_t \in \mathbb{R}^d$ and hidden state $\mathbf{h}_{t-1} \in \mathbb{R}^{d_{\text{hid}}}$, where d is the representation dimension and d_{hid} is the dimension of the hidden state, the RNN produces an output $\mathbf{y}_t \in \mathbb{R}^d$ and an updated hidden state $\mathbf{h}_t \in \mathbb{R}^{d_{\text{hid}}}$ using some recurrent function f:

$$\mathbf{y}_t, \mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t). \tag{2.4}$$

In actor-critic methods, which use neural networks to parameterize the policy and value function, RNNs are used to encode the sequence of observations and actions (see Bakker [2001], Onat et al. [1998]). RNNs such as Long Short Term Memory (LSTM) networks (Hochreiter and Schmidhuber [1997]) and Gated Recurrent Units (GRU) (Cho et al. [2014]) are popular choices for encoding the sequence of observations and actions, and have been shown to be effective for learning in partially observable environments (see Hausknecht and Stone [2015], Heess et al. [2015], Mnih et al. [2016], Espeholt et al. [2018]). Actor-critic algorithm that use an RNN as a function approximator typically use a shared RNN for both the policy and value function, with separate output layers for the policy and value function.

Figure 2.1 illustrates the use of an RNN to encode the sequence of observations in an actor-critic algorithm. For simplicity, we assume that the observations at each timestep is a vector of fixed dimension, $O_0, O_1, \ldots, O_t \in \mathbb{R}^{d_{\text{in}}}$, where d_{in} is the dimension of the observation vector. The action at each timestep is a discrete variable, $A_0, A_1, \ldots, A_t \in \mathcal{A}$. At a given time-step t, the observation O_t is first passed through a representation layer to generate input vector $\mathbf{x_t} \in \mathbb{R}^d$. The representation layer could be implemented using a feed-forward layer or a convolutional layer, depending on the nature of the observation. The input to the RNN at each time-step is the output of the representation layer and the previous hidden state $\mathbf{h}_{t-1}$, and it outputs an approximate state representation $\tilde{S}_t \in \mathbb{R}^d$, and an updated hidden state $\mathbf{h}_t$. The approximate state representation $\tilde{S}_t$ is then used as the input to the actor and critic heads. Typically, if the actions are discrete, the actor layer is implemented as a feedforward layer followed by a softmax, which outputs a probability distribution over the action space. The critic layer is implemented as a linear layer, which outputs a scalar value. The RNN, actor and critic parameters are updated using the sequence of observations and actions collected by the agent during the interaction phase. The RNN is trained using Truncated Backpropagation Through Time (TBPTT), which is a variant of Backpropagation Through Time (BPTT) (Werbos [1990]). TBPTT is similar to BPPT, except that the gradients are only propagated back for a fixed number of time steps.

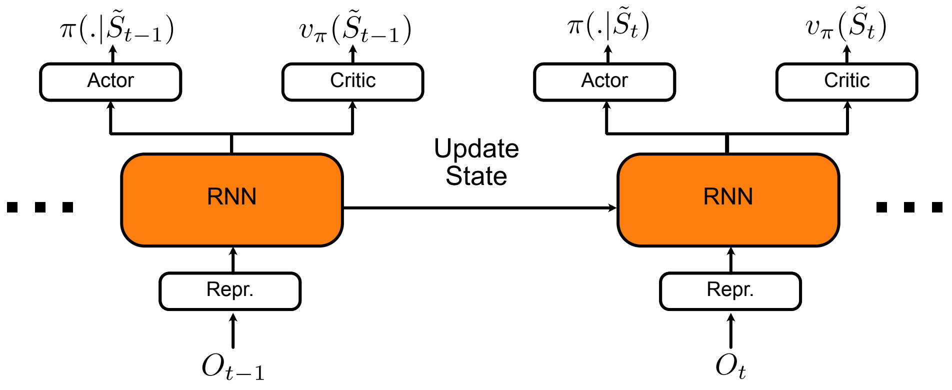

Figure 2.1: An RNN is used to encode the sequence of observations in an actor-critic algorithm. The RNN is updated using the sequence of observations and actions collected by the agent during the interaction phase. The output of the RNN is an approximate state representation $\tilde{S}_t$, which is used as the input to the actor and critic heads. The actor and critic heads are neural networks that output the policy and value function respectively.

2.4 Transformers in Supervised Settings

In this section, we delve into the core transformer architecture, beginning with an introduction to the canonical transformer architecture (Vaswani et al. [2017]) and the Transformer-XL architecture (Dai et al. [2019]), which are commonly used in supervised learning settings. We start with supervised learning because it is easier to understand the transformer architecture in the supervised setting, where these architectures were originally developed. Later, we follow up with a discussion of the challenges in applying transformers to the RL setting.

2.4.1 Canonical Transformer

The Transformer architecture was introduced by Vaswani et al. [2017] as an alternative to RNNs for processing sequential data in the supervised learning setting. The original architecture used an encoder-decoder structure. The encoder-decoder architecture is suitable for sequence-to-sequence tasks, where the input and output sequences can have different lengths, like machine translation. However, if the goal is to learn a representation of an input sequence, the encoder-only architecture introduced in Devlin et al. [2018] is sufficient.

As shown in Figure 2.2, the encoder-only Transformer architecture consists of L stacked encoder layers. Each encoder layer processes the input data as a batch, that is the entire input sequence is processed at once. The input is a matrix $\mathbf{X} \in \mathbb{R}^{N \times d}$, where N is the length of the input sequence, and d is the dimension of the input vectors. The output is also a matrix of the same dimension. The encoder layer of the transformer consists of a few key components, namely the multi-head self-attention layer, the feed-forward neural network, residual connections and layer normalization. The multi-head self-attention layer consists of multiple self-attention heads operating in parallel. We discuss the self-attention layer in detail in the next paragraph. The output of each head is concatenated and linearly transformed to obtain the final output of the self-attention layer. The feed-forward neural network consists of two linear transformations with a ReLU activation in between. It is responsible for introducing additional non-linear transformations. The residual connections (denoted by a plus), and layer normalization are introduced to stabilize the training of the self-attention layer. The layer normalization step, originally introduced by Ba et al. [2016], ensures that the token representations within the layer have consistent scales and are more amenable to training. We only provide an overview of the layer normalization, residual connections, and feed-forward neural network, and refer the reader to the original papers for more details.

Algorithm 1 Canonical Self-Attention (Batch Data)

```
Input: $\mathbf{X} \in \mathbb{R}^{N \times d}$
Parameters: $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V \in \mathbb{R}^{d \times d_h}$

1: $\mathbf{Q} \leftarrow \mathbf{X}\mathbf{W}_Q$
2: $\mathbf{K} \leftarrow \mathbf{X}\mathbf{W}_K$
3: $\mathbf{V} \leftarrow \mathbf{X}\mathbf{W}_V$
4: $\mathbf{A} \leftarrow softmax(\frac{\mathbf{Q}\mathbf{K}^\intercal}{\sqrt{d}})\mathbf{V}$
Output: $\mathbf{A} \in \mathbb{R}^{N \times d_h}$
```

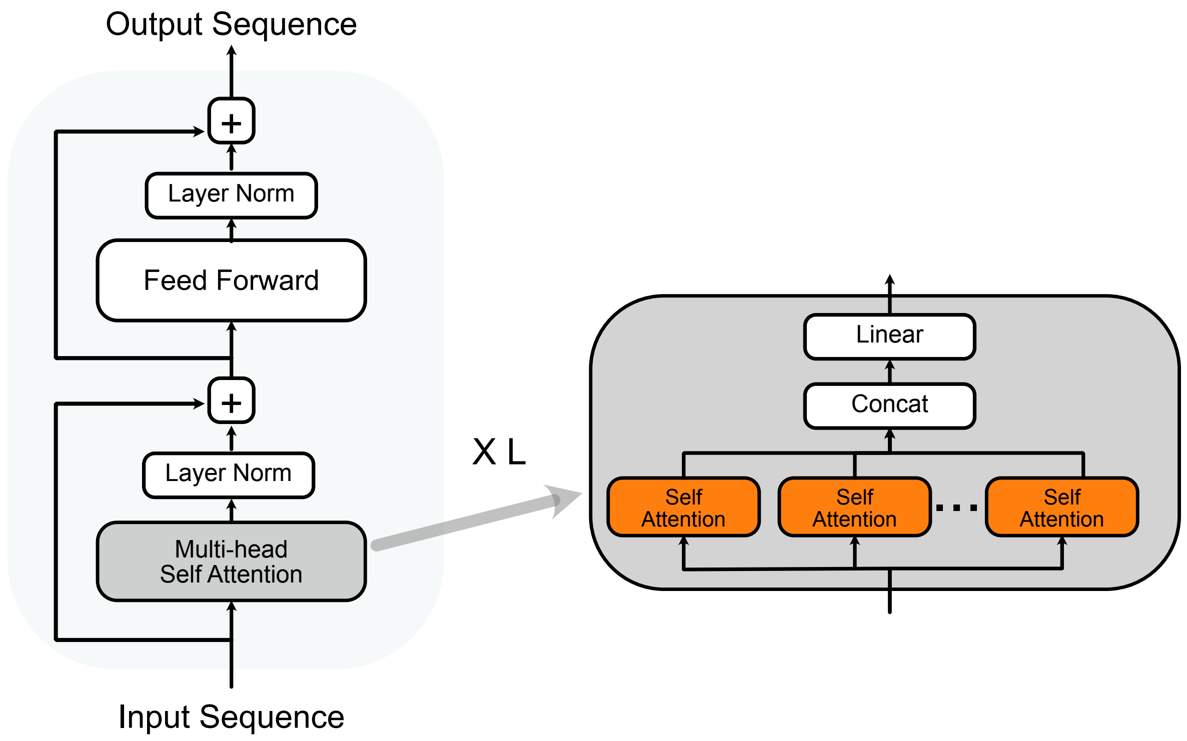

Figure 2.2: The transformer architecture. The left side shows the encoder-only transformer architecture. The encoder layer consists of a multi-head self-attention layer, a feed-forward neural network, residual connections and layer normalization. Multiple encoder layers are stacked to form the transformer architecture. The right side shows the multi-head self-attention layer. The multi-head self-attention layer consists of multiple self-attention heads operating in parallel. The output of each head is concatenated and linearly transformed to obtain the final output of the self-attention layer.

The self-attention layer learns context-aware representations at each index location in the input sequence and is the core idea behind transformers. The input to the self-attention layer is a matrix

$\mathbf{X} \in \mathbb{R}^{N \times d}$, and it generates a matrix output $\mathbf{A} \in \mathbb{R}^{N \times d_h}$, where d_h known as the head dimension is the output dimension of the self-attention layer. Algorithm 1 shows the self-attention layer in the canonical transformer. Here $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V \in \mathbb{R}^{d \times d_h}$ are learnable parameters. The self-attention layer calculates an attention operation over the input sequence. An attention operation can be defined as a process that maps a query and a set of key-value pairs to produce an output. In this process, all the components - the query, keys, values, and output - are represented as vectors. The output is calculated through a weighted sum of the values. The weight assigned to each value is determined by a compatibility function that takes into account the relationship between the query and the corresponding key. In the self-attention layer, attention weights are calculated using the dot-product of the query and key vectors. The query, key and value vectors are obtained by linearly transforming the input sequence using the learnable parameters $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V$ (lines 1-3). The attention output is then calculated as a weighted sum of the value vectors, where the weights are given by the attention weights (line 4). The *softmax* function is used to generate weightings over the elements in the input sequence. The attention output is then computed as a weighted sum of the value vectors.

The transformer architecture is trained through gradient descent using multiple pairs of input and output sequences. During each iteration, multiple input sequences are processed, generating predictions for corresponding output sequences. These predictions are compared to actual outputs using a predefined loss function, quantifying the model's performance. Gradients of the loss with respect to the model's parameters are computed and used to update the parameters.

2.4.2 Transformer-XL

The Transformer-XL (Dai et al. [2019]) architecture introduces modifications to the canonical self-attention that allows it to process longer sequences. The canonical self-attention described in Algorithm 1 assumes that the entire input sequence $\mathbf{X}$ is passed as input to the transformer. The computational complexity of the self-attention layer is quadratic in the input sequence length N. This makes it difficult to scale the transformer architecture to longer sequences where fitting the entire input sequence into memory is not feasible. To address this limitation, Dai et al. [2019] introduced the Transformer-XL architecture. Instead of processing the entire input sequence at once, the Transformer-XL architecture processes the input sequence in segments. The Transformer-XL architecture introduces a recurrence mechanism to enable the self-attention layer to capture dependencies beyond the input segment. To propagate information across segments, the self-attention layer stores the activations from the previous segment and concatenates them with the activations of the current segment. This allows the self-attention layer to capture long-range dependencies beyond the input segment.

Algorithm 2 Transformer-XL Self-Attention (Batch Data)

Input: $\mathbf{X} \in \mathbb{R}^{N \times d}$, l
Parameters: $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V \in \mathbb{R}^{d \times d_h}$

```
1: $\mathbf{S}_1, \ldots, \mathbf{S}_{N_s} \leftarrow split(\mathbf{X}, l)$
2: for $i = 1$ to $N_s - 1$ do
3:     $\tilde{\mathbf{X}} \leftarrow cat(sg(\mathbf{S}_i), (\mathbf{S}_{i+1}))$
4:     $\mathbf{Q} \leftarrow \tilde{\mathbf{X}}\mathbf{W}_Q$
5:     $\mathbf{K} \leftarrow \tilde{\mathbf{X}}\mathbf{W}_K$
6:     $\mathbf{V} \leftarrow \tilde{\mathbf{X}}\mathbf{W}_V$
7:     $\mathbf{A}_{i+1} \leftarrow softmax(\frac{\mathbf{Q}\mathbf{K}^\intercal}{\sqrt{d}})\mathbf{V}$
8: end for
9: $\mathbf{A} \leftarrow cat(\mathbf{A}_1, \ldots, \mathbf{A}_{N_s})$
```

Output: $\mathbf{A} \in \mathbb{R}^{N \times d_h}$

Algorithm 2 [1] shows the self-attention layer in the Transformer-XL architecture. Here, *sg*() denotes the stop-gradient operation and it ensures that the gradients do not flow through the activations from the previous segments. l is defined as the segment length. N_s is defined as the number of segments in the input sequence. *cat*() denotes the concatenation operation across the last dimension. $split(\mathbf{X}, l)$ splits the matrix $\mathbf{X}$ equally into l sized smaller matrices along the first dimension. The self-attention layer in the Transformer-XL architecture is similar to the self-attention layer in the canonical transformer architecture. The key difference is that the self-attention layer in the Transformer-XL architecture is unrolled across segments. The input sequence is split into segments of length l (line 1). The attention vector is then calculated sequentially for each segment (lines 3-7). The attention vector for the current segment is computed using the query, key and value vectors for the current segment and the activations from the previous segment. Since, the activations from the previous segment are concatenated to the current segment, the attention vector for the current segment can capture dependencies beyond the current segment. The attention vectors for each segment are then concatenated to form the final output of the self-attention layer (line 9).

2.5 Transformers in Reinforcement Learning

Applying transformers to the partially observable RL setting is challenging. The key challenge exists in applying the self-attention mechanism to streaming data, one input vector at a time.

[1] Algorithm 2 does not include the positional encoding mechanism used in the original Transformer-XL architecture because it is not necessary to understand the key challenges with applying existing transformers to RL. In our experiments, we do use the positional encoding mechanism with the GTrXL transformer baseline.

The self-attention mechanism expects that the entire input sequence is passed at once. Unlike the supervised learning setting, in RL, the input sequence is not available a priori and must be generated sequentially. The input sequence is generated through interaction with the environment. At each time step, the agent needs to take action to receive the next observation and reward. If the transformer architecture is used as a function approximator in an RL algorithm, the self-attention layer needs to be applied to the input sequence up to the current time step to produce an approximate state. Unlike RNN, the transformer architecture does not maintain a hidden state. Naively applying the self-attention mechanism to streaming data would require the self-attention layer to be applied to the entire input sequence up to the current time step, each time a new input vector arrives. This is computationally expensive as the number of operations required to compute the attention vector would increase linearly with the number of time steps. Alternatively, the context could be truncated to a fixed length. However, this would limit how far back in time the attention vector can remember.

To avoid the sequential nature of RL, some existing approaches formulate RL as a sequence modeling problem (see Chen et al. [2021], Laskin et al. [2022]). In these approaches, an RL algorithm is first used to generate training histories across multiple tasks. Then, an imitation approach is followed, wherein a transformer is trained end-to-end, using multiple sequences of training histories, to predict the next action at each time-step. The trained transformer could then be applied to a new task and would be able to learn in-context, i.e. without updating the transformer parameters. However, this approach is not scalable as it requires a large number of training histories to be generated across multiple tasks. Furthermore, the trained transformer does adapt online through interaction, which is a key aspect of RL, and an RL algorithm is still required to generate training histories. Therefore, we consider such approaches to be outside the scope of this thesis.

Application of transformers to the online RL setting is limited. One interesting approach is the Gated Transformer XL (GTrXL) architecture (Parisotto et al. [2020]) that introduced a modified transformer architecture as a function approximator in an RL algorithm. We consider the GTrXL architecture as a baseline in this thesis. The GTrXL architecture uses Transformer-XL self-attention as the self-attention mechanism but applies it in a streaming fashion, one input vector at a time. It does so by maintaining a finite number of inputs from the previous time step as a recurrent segment. In the following sections, we describe these approaches in detail.

2.5.1 Transformer-XL Self-Attention for Streaming Data

The Transformer-XL self-attention algorithm introduced in Algorithm 2 could be reformulated to process the input sequence in a streaming fashion by maintaining a recurrent state that is carried

Algorithm 3 Transformer-XL Self-Attention (Streaming Data)

Input: $\mathbf{x}_t \in \mathbb{R}^d$ $\mathbf{S}_{t-1} \in \mathbb{R}^{M \times d}$
Parameters: $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V \in \mathbb{R}^{d \times d_h}$

```
1: if $t = 0$ then
2:     $\mathbf{S}_0 \leftarrow \mathbf{0}$
3: end if
4: $\mathbf{S}_t \leftarrow cat(sg(\mathbf{S}_{t-1}[1:]), \mathbf{x}_t)$
5: $\mathbf{q}_t \leftarrow \mathbf{W}_Q^\intercal \mathbf{x}_t$
6: $\mathbf{K}_t \leftarrow \mathbf{S}_t \mathbf{W}_K$
7: $\mathbf{V}_t \leftarrow \mathbf{S}_t \mathbf{W}_V$
8: $\mathbf{a}_t \leftarrow softmax(\mathbf{q}_t \mathbf{K}_t^\intercal) \mathbf{V}_t$
```

Output: $\mathbf{a}_t \in \mathbb{R}^{d_h}$, $\mathbf{S}_t \in \mathbb{R}^{M \times d}$

forward in subsequent timesteps [2]. At a given time-step t, we will assume that the input passed to the self-attention mechanism is $\mathbf{x}_t \in \mathbb{R}^d$ instead of the entire input sequence $\mathbf{X} \in \mathbb{R}^{N \times d}$. Algorithm 3 presents the streaming version of the Transformer-XL self-attention. Here, [] refers to the indexing operation, and [1 :] refers to the indexing operation that returns all elements except the first element. The input to the self-attention layer at each time step is the current input $\mathbf{x}_t$ and a recurrent state $\mathbf{S}_{t-1} \in \mathbb{R}^{M \times d}$. Here M is a hyper-parameter that controls the size of the recurrent state, that is the number of stored activations. At each time step, the algorithm concatenates the current input to the previous history of activations $\mathbf{S}_{t-1}$ and then applies the self-attention layer. The output of the self-attention layer is then concatenated to the recurrent state $\mathbf{S}_{t-1}$ to form the new recurrent state $\mathbf{S}_t$. The recurrent state $\mathbf{S}_t$ is then used as the recurrent state for the next time step. The recurrent state $\mathbf{S}_t$ is initialized to zero at the start of processing each input sequence. The memory of the self-attention mechanism, number of timesteps in the history it can recall is determined by the hyper-parameter M. The self-attention mechanism can be applied to the input sequence in a streaming fashion by applying Algorithm 3 sequentially to the input sequence. The output of the self-attention layer at each time step is an attention vector $\mathbf{a}_t \in \mathbb{R}^{d_h}$ and the recurrent state $\mathbf{S}_t \in \mathbb{R}^{M \times d}$.

[2]The streaming variant of the Transformer-XL algorithm mentioned in this thesis is described as the evaluation mode of the Transformer-XL architecture in the original paper. GTrXL uses Transformer-XL self-attention in the RL setting but does not mention the details of how self-attention is applied. Nevertheless, the streaming variant of the Transformer-XL self-attention is important to discuss as it is necessary for agent-environment interaction and is the main bottleneck due to the sequential nature of the RL setting.

2.5.2 Gated Transformer-XL

The performance of the transformer architecture (Figure 2.2) in partially observable RL tasks has been found to be notably challenging to optimize, often resulting in performance comparable to a random policy (see Parisotto et al. [2020]). This optimization difficulty is not limited to RL but is also observed in the supervised learning case, where complex learning rate schedules or specialized weight initialization schemes are needed to train transformers effectively (see Vaswani et al. [2017], Dai et al. [2019]). However, these measures do not appear to be sufficient for RL, as RL algorithms often fail to converge when coupled with a transformer architecture (see Mishra et al. [2018]).

The GTrXL architecture introduced by Parisotto et al. [2020] introduces two key modifications to the encoder layers in regular transformer architecture. Figure 2.3 introduces these modifications and also shows the GTrXL architecture used as a function approximator in an actor-critic algorithm. First, the GTrXL encoder rearranges the layer normalization modules to appear before the self-attention layer instead of after the self-attention layer. The advantage of this reordering is that it allows for an identity map from the input of the transformer at the initial layer to the output of the transformer after the final layer. This differs from the canonical transformer, where a sequence of layer normalization operations non-linearly transforms the state encoding. The reordering of the layer normalization modules enables an initial Markov regime of training and is found to be critical for the success of the GTrXL architecture. Second, the GTrXL architecture introduces a GRU-based gating mechanism (Gao and Glowacka [2016]) instead of the standard residual connections, which further stabilizes the training of the self-attention layer in the reinforcement learning setting. The empirical gains of these modifications are discussed in detail in the original paper (Parisotto et al. [2020]), and the reader is referred to the original paper for more details.

Application of GTrXL as a function approximator to learn an actor and a critic in an actor-critic algorithm is shown in Figure 2.3. The GTrXL architecture uses the Transformer-XL self-attention layer, described in Algorithm 3, to process the input sequence in a streaming fashion. Similar to Figure 2.1, the GTrXL is applied sequentially to the sequence of observations. First, the input observation O_t at a time-step t is passed through a representation layer to generate an input vector of dimension d. The input vector is then passed through the GTrXL architecture to generate an approximate state $\tilde{S}_t \in \mathbb{R}^d$. The approximate state $\tilde{S}_t$ is then passed through a policy head and a value head to generate the policy and value estimates. Each head of the self-attention interdependently maintains a recurrent state $\mathbf{C}_{t-1}$ and $\mathbf{s}_{t-1}$, as a recurrent state, which is updated to the new state $\mathbf{C}_t$ and $\mathbf{s}_t$, according to Algorithm 3.

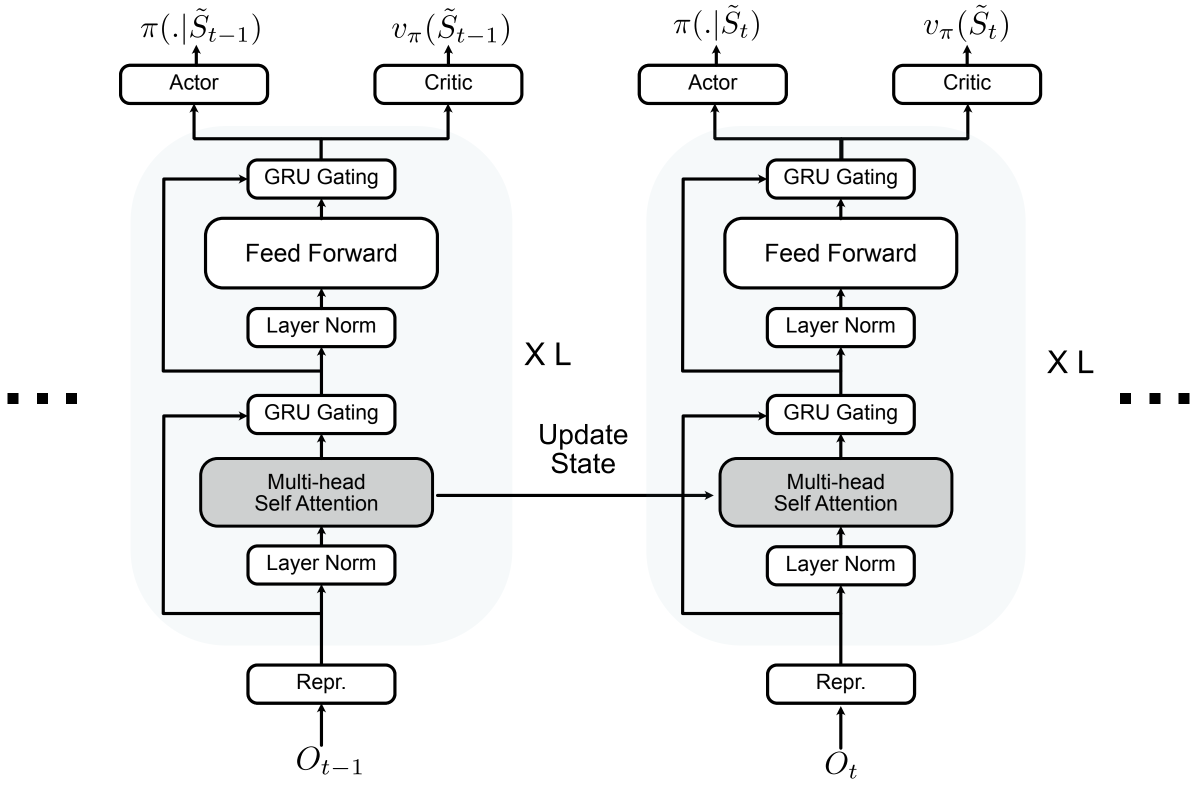

Figure 2.3: The GTrXL architecture used as a function approximator to learn an actor and a critic in an actor-critic algorithm. The GTrXL architecture is applied sequentially to the input sequence. The recurrent state is carried forward in subsequent timesteps. Additionally, the GTrXL architecture rearranges the layer-norm and introduces a gating mechanism to stabilize the training of the self-attention layer.

Applying GTrXL self-attention to a streaming input sequence is computationally challenging due to two key reasons. First, the context length of GTrXL, the history of observations the GTrXL model can recall, is determined by the size of the recurrent state M. For an L layered GTrXL model the context length is $\mathcal{O}(LM)$; the number of timesteps the GTrXL model can recall is directly proportional to the number of stored activations in the recurrent state. Second, the inference space and time complexity, the cost of applying the GTrXL self-attention for a single element in the sequence, is $\mathcal{O}(Md)$ and $\mathcal{O}(Md^2)$ respectively; the inference cost is directly proportional to the size of the recurrent state. This dependency of context length on the size of the recurrent state and the inference cost on the context length makes it challenging to scale the GTrXL model to long sequences. In problems that require long context lengths, it is necessary to have a large recurrent state, which comes at the cost of an increased inference cost. The problems are exemplified in the partially observable RL setting, where the episodes are long and a slow inference time is undesirable as it limits the agent's ability to react to changes in the environment.

2.6 Recurrent Attention with Linear Transformers

RNNs are a natural fit for processing a streaming sequence. RNNs, such as LSTMs, store information about the past in their recurrent state, which is of finite size. They do not require past activations to be stored to recall the past, and the inference cost is independent of the context length. It would interesting to explore if the self-attention mechanism in transformers can be formulated as a recurrent neural network.

Katharopoulos et al. [2020] propose an alternate way of formulating self-attention that could be updated iteratively, similar to RNNs. The approach is called the Linear Transformer, it introduces a general way of formulating attention by leveraging its equivalence to applying kernel smoothing over inputs (see Tsai et al. [2019]). They replace the *softmax* with a kernel function which is defined such that $k : \mathbb{R}^{d_h} \times \mathbb{R}^{d_h} \to \mathbb{R}^+$, where $k(\mathbf{a}, \mathbf{b}) = \phi(\mathbf{a})^\intercal \phi(\mathbf{b})$; $\phi : \mathbb{R}^{d_h} \to \mathbb{R}^{d_k}$ is a non-linear feature map, d_k is called the output dimension of the feature map ϕ. Let $\otimes$ be defined as the vector outer product operation. A single time-step of inference of the Linear Transformer is described in Algorithm 4. At a given timestep t, the Linear Transformer architecture maintains a matrix $\mathbf{C}_{t-1} \in \mathbb{R}^{d_h \times d_k}$ and a vector $\mathbf{s}_t \in \mathbb{R}^{d_k}$ as a recurrent state, which is updated iteratively using the current input vector $\mathbf{x}_t$. Different from Algorithm 3, Algorithm 4 applies the feature map ϕ to generate the query and key for a given time-step (lines 4 and 5). The original implementation of Linear Transformer used Exponential Linear Unit (ELU) + 1 (Clevert et al. [2016]) as the feature map ϕ. Instead of storing the past activations, the Linear Transformer architecture stores the outer product of value and key vectors as a recurrent matrix state $\mathbf{C}_t$ (line 7). Additionally, it also stores the sum of the key vectors as a recurrent normalization vector $\mathbf{s}_t$ (line 8). The attention output vector $\mathbf{a}_t$ is calculated by multiplying the recurrent state with the query vector, and normalizing it using the product of the normalization vector $\mathbf{s}_t$ and the query vector $\mathbf{q}_t$ (line 9). More details about how the recurrent update is derived could be found in Katharopoulos et al. [2020].

Unlike GTrXL's self-attention, the Linear Transformer's self-attention has a context-independent inference cost. Processing a single input vector using Linear Transformer's self-attention has a space and time complexity of $\mathcal{O}(dd_k)$, where d is the embedding dimensionality, and d_k is the dimensionality of the kernel feature space. Unlike vanilla self-attention, the computational complexity does not depend on the context length making it far more efficient for longer sequences. Algorithm 4 is similar to an RNN in the sense that it maintains a recurrent state allowing for a potentially unbounded context, and has a context-independent inference cost.

2.6.1 Limitations

Algorithm 4 has some important limitations when applying long sequences:

Algorithm 4 Linear Transformer Self-Attention (Streaming Data)

```
Input: $\mathbf{x}_t \in \mathbb{R}^d$, $\mathbf{C}_{t-1} \in \mathbb{R}^{d_h \times d_k}$, $\mathbf{s}_{t-1} \in \mathbb{R}^{d_k}$
Parameters: $\mathbf{W}_K, \mathbf{W}_Q, \mathbf{W}_V \in \mathbb{R}^{d \times d_h}$

1: if $t = 0$ then
2:     $\mathbf{s}_0 \leftarrow \mathbf{0}, \mathbf{C}_0 \leftarrow \mathbf{0}$.
3: end if
                                        {Calculate Keys, Queries, Values}
4: $\tilde{\mathbf{k}}_t \leftarrow \mathbf{W}_K^\intercal \mathbf{x}_t, \mathbf{k}_t \leftarrow \phi(\tilde{\mathbf{k}}_t)$
5: $\tilde{\mathbf{q}}_t \leftarrow \mathbf{W}_Q^\intercal \mathbf{x}_t, \mathbf{q}_t \leftarrow \phi(\tilde{\mathbf{q}}_t)$
6: $\mathbf{v}_t \leftarrow \mathbf{W}_V^\intercal \mathbf{x}_t$
                                        {Update Memory}
7: $\mathbf{C}_t \leftarrow \mathbf{C}_{t-1} + \mathbf{v}_t \otimes \mathbf{k}_t$
8: $\mathbf{s}_t \leftarrow \mathbf{s}_{t-1} + \mathbf{k}_t$
                                        {Calculate Attention Vector}
9: $\mathbf{a}_t \leftarrow (\mathbf{C}_t \mathbf{q}_t)/(\mathbf{s}_t \mathbf{q}_t)$
Output: $\mathbf{a}_t \in \mathbb{R}^{d_h}$,$\mathbf{C}_t \in \mathbb{R}^{d_h \times d_k}$, $\mathbf{s}_t \in \mathbb{R}^{d_k}$
```

1. The recurrent equations in Algorithm 4 (lines 5 and 6) naively add positive values to the recurrent state. In Reinforcement Learning, episode lengths could be long and thus the recurrent state $\mathbf{C}_t$, $\mathbf{s}_t$ could become potentially large.

2. The right choice of the kernel feature map ϕ can affect the performance of the self-attention mechanism, and element-wise functions such as the ELU+1 perform much worse than softmax (Katharopoulos et al. [2020]).

3. A matrix is used as a recurrent state for each head. Transformer architectures typically utilize multiple self-attention heads to capture different relationships between elements in the sequence and having a matrix as a recurrent state for each head has a high memory cost.

In this thesis, we explore how to mitigate these three limitations. In Chapter 3, we introduce an approach that presents modifications to the Linear Transformer self-attention mechanism to address the first two limitations. We call this architecture the Recurrent Linear Transformer (ReLiT). In Chapter 4, we introduce an approach that addresses the third limitation by approximating the self-attention mechanism. We call this architecture the Approximate Recurrent Linear Transformer (AReLiT).

Chapter 3

Recurrent Linear Transformer

In this chapter, we propose two modifications to the recurrent formulation of linear transformer self-attention described in Algorithm 4:

1. We introduce a gating mechanism to control the flow of information at each index location of the recurrent states of the Linear Transformer self-attention mechanism, potentially allowing arbitrary context lengths.
2. We introduce a parameterized feature map to calculate the key and query vectors in the Linear Transformer self-attention mechanism, eliminating the choice of kernel feature map ϕ.

We introduce our modified self-attention mechanism into the GTrXL architecture described in Figure 2.3 and call this approach the Recurrent Linear Transformer (ReLiT) architecture. We describe our proposed algorithm and further elaborate on its computational complexity. Finally, we discuss the challenges of applying ReLiT to large architectures, such as with multiple heads and layers. These issues will be addressed in the next chapter.

3.1 Gating Mechanism to Control the Flow of Information

In the Linear Transformer self-attention, at a given time-step t, Algorithm 4 adds the recurrent states $\mathbf{C}_{t-1}$ and $\mathbf{s}_{t-1}$ to the current recurrent states $\mathbf{C}_t$ and $\mathbf{s}_t$ (lines 7 and 8). Asumming that $\mathbf{C}_0$ and $\mathbf{s}_0$ are initialized to zero, the update equations for $\mathbf{C}_t$ and $\mathbf{s}_t$ are then recursively defined as follows:

$$\mathbf{C}_t \doteq \mathbf{C}_{t-1} + \mathbf{v}_t \otimes \mathbf{k}_t, \tag{3.1}$$

$$\mathbf{s}_t \doteq \mathbf{s}_{t-1} + \mathbf{k}_t. \tag{3.2}$$

Equation 3.1 and 3.2 add arbitrary positive values to the recurrent states $\mathbf{C}_{t-1}$ and $\mathbf{s}_{t-1}$ and have no mechanism to control the flow of information from the past. This is detrimental for long sequences as the values in the recurrent states could grow arbitrarily large, making prediction unstable.

Gating mechanisms can be used to control the flow of information in recurrent updates. Gating mechanisms are commonly used in RNNs such as LSTM and GRU to control the flow of information and reduce vanishing gradient problems (Hochreiter and Schmidhuber [1997]). Gating mechanisms have also been applied to the Linear Transformer architecture. Peng et al. [2021] introduced a single learned scalar parameter to control the flow of information in the Linear Transformer architecture. In their approach, at a given time-step t, the scalar parameter having a value between 0 and 1 is multiplied to the previous recurrent states $\mathbf{C}_{t-1}$ and $\mathbf{s}_{t-1}$, controlling the flow of information from the past. However, using a single learned coefficient is sub-optimal as it controls the flow of past information from each index location in a recurrent state identically. Individually controlling the flow at index location of the recurrent state could be beneficial as it allows the network to selectively update and delete information from the past.

We propose a learned outer-product based gating mechanism that decays every position of the recurrent tensors, allowing the network to learn the decay at each memory location. We introduce learnable parameters $\mathbf{W}_\beta \in \mathbb{R}^{d \times d_h}$, $\mathbf{W}_\gamma \in \mathbb{R}^{d \times d_k}$, and to learn gating vectors β_t and γ_t. Let σ_g be a sigmoid function defined as $\sigma_g(x) \doteq \frac{1}{1+e^{-x}}$, we define β_t and γ_t as follows:

$$\begin{aligned} \tilde{\beta}_t &\doteq \mathbf{W}_\beta^\intercal \mathbf{x}_t \\ \beta_t &\doteq \sigma_g(\tilde{\beta}_t), \qquad (3.3) \\ \tilde{\gamma}_t &\doteq \mathbf{W}_\gamma^\intercal \mathbf{x}_t \\ \gamma_t &\doteq \sigma_g(\tilde{\gamma}_t). \qquad (3.4) \end{aligned}$$

Let $\odot$ be the element-wise product, we use the outer product of β_t and γ_t to control the flow of past information in recurrent states $\mathbf{C}_t$ and $\mathbf{s}_t$ modifying Equations 3.1 and 3.2 as follows (changes from Equations 3.1 and 3.2 are highlighted in blue):

$$\mathbf{C}_t \doteq \big((1-\beta_t) \otimes (1-\gamma_t)\big) \odot \mathbf{C}_{t-1} + \big(\beta_t \odot \mathbf{v}_t\big) \otimes \big(\gamma_t \odot \mathbf{k}_t\big), \tag{3.5}$$

$$\mathbf{s}_t \doteq (1-\gamma_t) \odot \mathbf{s}_{t-1} + \gamma_t \odot \mathbf{k}_t. \tag{3.6}$$

We use outer products as it utilizes multiplicative interactions to learn the decay rate at each index location of the recurrent state, without requiring individual parameters for each index location. The outer product futher assume that the decay rate at each index location is independent of the decay rate at other index locations. The potential implications of such an assumption require further investigation.

3.2 Learnable Feature Map for Self-Attention

The Linear Transformer self-attention, described in Algorithm 4, uses a kernel feature map to calculate the key and query vectors. At a given time-step t, Algorithm 4 replaces the softmax function with a kernel feature map ϕ, and uses it to calculate the key and query vectors as follow:

$$\begin{aligned}
\tilde{\mathbf{k}}_t &\doteq \mathbf{W}_K^\intercal \mathbf{x}_t \\
\mathbf{k}_t &\doteq \phi(\tilde{\mathbf{k}}_t), \qquad (3.7) \\
\tilde{\mathbf{q}}_t &\doteq \mathbf{W}_Q^\intercal \mathbf{x}_t \\
\mathbf{q}_t &\doteq \phi(\tilde{\mathbf{q}}_t). \qquad (3.8)
\end{aligned}$$

Replacing the softmax with a feature map is essential for the Linear Transformer architecture as it allows for a recursive reformulation of the self-attention mechanism.

However, the choice of the feature map ϕ could have a significant impact on the overall performance (Schlag et al. [2021]). Katharopoulos et al. [2020] used *ELU+1* activation function (Clevert et al. [2016]) as a feature map. Element-wise activation functions are limited in their ability to learn complex non-linear relationships. Further, using element-wise activation functions as a feature map limits the memory capacity of the architecture (Schlag et al. [2021]). Alternatively, Peng et al. [2021] and Choromanski et al. [2020] use random feature maps to approximate a softmax function. Randomized feature maps are equivalent to softmax function in expectation, but could introduce variance in the model.

We instead consider a deterministic approach to learn the key and value vectors in the Linear Transformer self-attention mechanism. We introduce modifications to the key, query and gating vectors calculation described in Equations 3.7, 3.8, 3.3 and 3.4 respectively. We start by introducing a hyper-parameter η that allows controlling the dimensions of the feature maps used to construct the key and the query vectors. We introduce learnable parameters $\mathbf{W}_{p_1}, \mathbf{W}_{p_2}, \mathbf{W}_{p_3} \in \mathbb{R}^{d\times\eta}$. We modify the dimensions of $\mathbf{W}_\gamma$ as $\mathbf{W}_\gamma \in \mathbb{R}^{d\times d_h}$, getting rid of d_k, the kernel feature map dimension. Let $\otimes$ be the outer product notation and *flatten*() be a function that flattens a matrix into a vector, then we redefine the key and query vectors calculation defined in Equation 3.7 and 3.8 as follows:

$$\begin{aligned}
\tilde{\mathbf{k}}_t &\doteq \mathbf{W}_K^\intercal \mathbf{x}_t \\
\mathbf{k}_t &\doteq \mathit{flatten}(\mathit{relu}(\mathbf{W}_{p_1}^\intercal \mathbf{x}_t) \otimes \mathit{relu}(\tilde{\mathbf{k}}_t)) \qquad (3.9)
\end{aligned}$$

$$\begin{aligned}\tilde{\mathbf{q}}_t &\doteq \mathbf{W}_Q^\intercal \mathbf{x}_t \\ \mathbf{q}_t &\doteq \mathit{flatten}(\mathit{relu}(\mathbf{W}_{p_2}^\intercal \mathbf{x}_t) \otimes \mathit{relu}(\tilde{\mathbf{q}}_t))\end{aligned} \tag{3.10}$$

We also modify the gating vectors γ_t calculation in Equation 3.4 as follows:

$$\begin{aligned}\tilde{\gamma}_t &\doteq \mathbf{W}_\gamma^\intercal \mathbf{x}_t \\ \gamma_t &\doteq \mathit{flatten}(\sigma_g(\mathbf{W}_{p_3}^\intercal \mathbf{x}_t) \otimes \sigma_g(\tilde{\gamma}_t))\end{aligned} \tag{3.11}$$

Using the modified key, query and gating vectors, the recurrent states $\mathbf{C}_t$ and $\mathbf{s}_t$ are calculated according to Equations 3.5 and 3.6 respectively. The size of key and query vectors is controlled by the hyper-parameter η, which also controls the size of the recurrent states $\mathbf{C}_t$ and $\mathbf{s}_t$. Equations 3.9 and 3.10 utilize outer products to learn multiplicative interactions in the key and query vectors. Learning multiplicative interactions in the feature vectors could be useful as it allows us to learn complex non-linear relationships through training, instead of relying on an explicit non-linear element-wise function or a random feature map.

We the *relu* activation function to ensure that the output of the feature map is positive. Positive feature map output is neccessary as it ensures that the similarity scores produced by the underlying kernel function are positive. The choice of the *relu* activation function is arbitrary and could potentially be replaced with other activation functions which ensure positive output.

3.3 Recurrent Linear Transformer (ReLiT)

We introduce the Recurrent Linear Transformer (ReLiT) architecture, which introduces a modified Linear Transformer self-attention mechanism to the GTrXL architecture. The modifications to the Linear Transformer self-attention mechanism[1] are presented in Algorithm 5. Changes from Algorithm 4 are highlighted in blue. These modifications introduce our proposed gating and feature map approaches, described in the previous section. The algorithm introduces a hyper-parameter η that controls the size of the key and query vectors, and the size of the recurrent states $\mathbf{C}_t$ and $\mathbf{s}_t$. It also introduces additional learnable parameters $\mathbf{W}_\beta, \mathbf{W}_\gamma \in \mathbb{R}^{d \times d_h}$ and $\mathbf{W}_{p_1}, \mathbf{W}_{p_2}, \mathbf{W}_{p_3} \in \mathbb{R}^{d \times \eta}$. We then replace the self-attention mechanism in the GTrXL architecture with Algorithm 5. We refer to this modified architecture as Recurrent Linear Transformer (ReLiT).

We compare the space and time complexity of ReLiT and GTrXL in Table 3.1. We assume that in the worst case, the head dimension equals the embedding dimension, that is $d_h = d$. This

[1]Unlike Transformer-XL, ReLiT does not require any sort of positional encoding mechanism. Recency bias is introduced through the proposed gating mechanism and as such an explicit positional encoding is no longer needed.

Algorithm 5 Recurrent Linear Transformer (ReLiT) Self-Attention (Streaming Data)

Input: $\mathbf{x}_t \in \mathbb{R}^d$, $\mathbf{C}_{t-1} \in \mathbb{R}^{d_h \times \eta d_h}$, $\mathbf{s}_{t-1} \in \mathbb{R}^{\eta d_h}$
Parameters: $\mathbf{W}_K, \mathbf{W}_Q, \mathbf{W}_V, \mathbf{W}_\beta, \mathbf{W}_\gamma \in \mathbb{R}^{d \times d_h}$ and $\mathbf{W}_{p_1}, \mathbf{W}_{p_2}, \mathbf{W}_{p_3} \in \mathbb{R}^{d \times \eta}$

```
1: if $t = 0$ then
2:     $\mathbf{s}_0 \leftarrow \mathbf{0}, \mathbf{C}_0 \leftarrow \mathbf{0}$.
3: end if
                                                        {Calculate Keys}
4: $\tilde{\mathbf{k}}_t \leftarrow \mathbf{W}_K^\intercal \mathbf{x}_t$
5: $\mathbf{k}_t \leftarrow \mathit{flatten}(\mathit{relu}(\mathbf{W}_{p_1}^\intercal \mathbf{x}_t) \otimes \mathit{relu}(\tilde{\mathbf{k}}_t))$
                                                        {Calculate Queries}
6: $\tilde{\mathbf{q}}_t \leftarrow \mathbf{W}_Q^\intercal \mathbf{x}_t$
7: $\mathbf{q}_t \leftarrow \mathit{flatten}(\mathit{relu}(\mathbf{W}_{p_2}^\intercal \mathbf{x}_t) \otimes \mathit{relu}(\tilde{\mathbf{q}}_t))$
                                                        {Calculate Values}
8: $\mathbf{v}_t \leftarrow \mathbf{W}_V^\intercal \mathbf{x}_t$
                                                        {Generate Gating Vectors}
9: $\beta_t \leftarrow \sigma_g(\mathbf{W}_\beta^\intercal \mathbf{x}_t)$
10: $\tilde{\gamma}_t \leftarrow \mathbf{W}_\gamma^\intercal \mathbf{x}_t$
11: $\gamma_t \leftarrow \mathit{flatten}(\sigma_g(\mathbf{W}_{p_3}^\intercal \mathbf{x}_t) \otimes \sigma_g(\tilde{\gamma}_t))$
                                                        {Update Memory}
12: $\mathbf{C}_t \leftarrow \big((1-\beta_t) \otimes (1-\gamma_t)\big) \odot \mathbf{C}_{t-1} + \big(\beta_t \odot \mathbf{v}_t\big) \otimes \big(\gamma_t \odot \mathbf{k}_t\big)$
13: $\mathbf{s}_t \leftarrow (1-\gamma_t) \odot \mathbf{s}_{t-1} + \gamma_t \odot \mathbf{k}_t$
                                                        {Calculate Attention Vector}
14: $\mathbf{a}_t \leftarrow (\mathbf{C}_t \mathbf{q}_t)/(\mathbf{s}_t \mathbf{q}_t)$
```

Output: $\mathbf{a}_t \in \mathbb{R}^{d_h}$, $\mathbf{C}_t \in \mathbb{R}^{d_h \times \eta d_h}$, $\mathbf{s}_t \in \mathbb{R}^{\eta d_h}$

assumption is reasonable as the head dimension is typically a fraction of the embedding dimension. The computational complexity described is for processing a single element in a sequence that is presented in a streaming fashion. The space and time complexity of GTrXL is both dependent on the number of stored past activations M, which further influences the context length. Increasing the context length requires more memory and computation. In contrast, the space and time complexity of ReLiT are independent of the context length and only depend on static hyperparameters d and η.

To elaborate on the computational efficiency of our approach, we can compare the exact number operations and the space required when using typical hyperparameter values. As an estimate of the number of operations, we calculate the total number of floating point operations required to process a single element in a streaming sequence. To estimate the space required, we calculate the total number of floating point numbers required to store the recurrent state. For GTrXL self-attention we consider the hyperparameter values from Parisotto et al. [2020], that is $M = 512$, $d = 256$ and $d_h = 64$. For ReLiT, we consider $d = 256$, $d_h = 64$, and $\eta = 4$. The goal is to match the

Table 3.1: Space and time complexity of ReLiT, Linear Transformer, and GTrXL self-attention for processing a single element in a streaming sequence. M: memory size in GTrXL, d: representation dimension, d_k feature map dimension in Linear Transformer, η: feature map hyperparameter in ReLiT, L: number of encoder layers

	Space	**Time**	**Potential Context Length**
GTrXL	$\mathcal{O}(Md)$	$\mathcal{O}(M\ d^2)$	$\mathcal{O}(LM)$
Linear Transformer	$\mathcal{O}\left(d_k d\right)$	$\mathcal{O}\left(d_k d\right)$	∞
ReLiT	$\mathcal{O}\left(\eta d^2\right)$	$\mathcal{O}\left(\eta d^2\right)$	∞

hyperparameters of the GTrXL architecture as closely as possible and chose typical values for the novel hyperparameters. We chose $\eta = 4$ as we found it to work reasonably well in our experiments. ReLiT self-attention is roughly 110.95 times faster than GTrXL self-attention and uses 7.87 times less space. This improvement in time and space is only possible because the complexity of ReLiT self-attention does not depend on the context length.

3.3.1 Limitations

Algorithm 5 still has some limitations. First, it requires storing a matrix of dimension $d^2\eta$ as a recurrent hidden state. The issue is that the space complexity of this approach quadratically increases with the embedding dimension d. While this may seem trivial, it becomes an issue as transformer architectures typically use multiple heads and layers as it helps to improve stability during the training process (see Michel et al. [2019]). For example, the GTrXL architecture uses 8 heads and 12 layers, which results in a total of 96 heads. This might make it infeasible to use on low-memory devices. Another issue is that an explicit outer product and element-wise matrix sum and multiplication operations are required to calculate the updates to the recurrent state. These operations are expensive in-practice as they require significant DRAM access, which results in increased inference time.

In the next chapter, we introduce an approximation of the ReLiT self-attention mechanism that addresses these limitations.

Chapter 4

Approximate Recurrent Linear Transformer

In this chapter, we introduce an approximation of the recurrent sum defined in Equation 3.5. Our proposed approach, called Approximate Recurrent Linear Transformer (AReLiT), replaces the previous recurrent state matrix $\mathbf{C}_{t-1}$ with a set of vectors and reduces the space complexity of ReLiT by d, the embedding dimension. We start by introducing an approximation of the Kronecker delta function using sum of cosine functions. We then use this result to approximate the recurrent state matrix $\mathbf{C}_{t-1}$. We introduce the AReLiT architecture, which uses this low-rank approximation to reduce the space complexity of ReLiT. We discuss the computational complexity of AReLiT and compare it to ReLiT and GTrXL. The space complexity of AReLiT and the quality of the approximation are proportional to the introduced hyper-parameter r. We demonstrate empirically, with a synthetic example, that this hyper-parameter could be set to a small value, as the approximation error is small even for small values of r.

4.1 Recurrent Approximation of Memory Matrix in ReLiT

Our goal is to approximate the recurrent state update in Equation 3.5 with some approximation that uses less space than $\mathcal{O}(\eta d^2)$. Recall that Equation 3.5 calculates an updated recurrent state $\mathbf{C}_t$ by adding a new outer product to the previous state $\mathbf{C}_{t-1}$:

$$\mathbf{C}_t \doteq \big((1-\beta_t)\otimes(1-\gamma_t)\big)\odot\mathbf{C}_{t-1} + \big(\beta_t\odot\mathbf{v}_t\big)\otimes\big(\gamma_t\odot\mathbf{k}_t\big) \tag{4.1}$$

The rank of the recurrent state $\mathbf{C}_{t-1}$ is ηd. The information in the recurrent state is updated as follows: First, the old state is multiplied element-wise by the outer product of the complement of the gating vectors. Second, new information is added through an element-wise matrix sum. The new information is represented as an outer product of the value and key vectors. To derive an approximation, we want to replace $\mathbf{C}_{t-1}$ with a matrix that has a lower rank. Also, we want to derive an update rule that is an approximation of Equation 4.1, but instead of updating the full-rank matrix $\mathbf{C}_{t-1}$, we update the low-rank approximation.

Existing approaches have explored incremental updates to the low-rank approximation of large matrices. Incremental Singular Value Decomposition (SVD) (Brand [2002, 2006]) provides a way to perform additive modifications to a low-rank singular value decomposition of a matrix. In this approach, a low-rank decomposition of the matrix is updated incrementally as new information arrives. Previous applications of incremental SVD in RL, however, suggest that sensitivity to the rank parameter is a significant issue (see Pan et al. [2017]). Another approach is the rank-1 trick introduced by Tallec and Ollivier [2017]. The rank-1 trick uses random numbers to approximate a Kronecker delta function and uses it to derive an unbiased approximation of a matrix represented as a sum of outer products. The use of random numbers, however, introduces variance in the approximation (see Cooijmans and Martens [2019]).

In this thesis, we consider an approximation that uses a sum of cosine functions to approximate a sum of outer products. This approximation is deterministic and does not introduce variance in the approximation, and it keeps incremental updates to the state end-to-end differentiable. Our approach is a follow-up of the rank-1 trick, but instead of using random numbers to approximate a Kronecker delta function, we utilize a trigonometric identity that relates a Kronecker delta function to an integral over cosines.

4.1.1 Approximation of Kronecker Delta Function

We start by deriving an approximation of the Kronecker delta function. The Kronecker delta function is defined for integers m and n as:

$$\delta_{mn} = \begin{cases} 1 & \text{if } m = n \\ 0 & \text{if } m \neq n \end{cases}$$

We use a trigonometric identity that is used in computing Fourier series by relating the Kronecker delta function to an integral of a product of two cosine functions (Weisstein). The identity is given by:

$$\delta_{mn} = \frac{1}{\pi} \int_0^{2\pi} \cos(mx)\ \cos(nx)\ dx. \tag{4.2}$$

We use the Trapezoidal rule to approximate the integral in Equation 4.2. The trapezoidal rule is a numerical integration method that approximates the integral of a function by dividing the interval into sub-intervals and approximating the function in each sub-interval with a straight line connecting the endpoints. For a function $f(x)$ that is integrable on the interval $[a, b]$, the trapezoidal rule is given by:

$$\int_a^b f(x)\ dx \approx \sum_{k=1}^{r} \frac{f(x_{k-1}) + f(x_k)}{2} \Delta x, \tag{4.3}$$

where $\Delta x = \dfrac{b-a}{r}$, $x_k = a + k\Delta x$, and r is the number of sub-intervals used for the integral and it controls the degree of approximation. As $r \to \infty$ the approximation becomes exact. Let $\tilde{\delta}_{mn}$ be the Trapezoidal approximation of the integral defined in Equation 4.2. We can then write $\tilde{\delta}_{mn}$ as follows:

$$\tilde{\delta}_{mn} = \frac{1}{r}\sum_{i=0}^{r-1} \cos\left(\frac{2\pi i}{r}m\right)\cos\left(\frac{2\pi i}{r}n\right) + \frac{1}{r}\sum_{i=1}^{r} \cos\left(\frac{2\pi i}{r}m\right)\cos\left(\frac{2\pi i}{r}n\right) \tag{4.4}$$

Further, in the limit we have: $\lim_{r\to\infty} \tilde{\delta}_{mn} = \delta_{mn}$.

Next, we will simplify the above equation to combine the two summations above into a single one:

$$\begin{aligned}
\tilde{\delta}_{mn} &= \frac{1}{r}\sum_{i=0}^{r-1} \cos\left(\frac{2\pi i}{r}m\right)\cos\left(\frac{2\pi i}{r}n\right) + \frac{1}{r}\sum_{i=1}^{r} \cos\left(\frac{2\pi i}{r}m\right)\cos\left(\frac{2\pi i}{r}n\right) \\
&\text{Adding and subtracting } \frac{1}{r}(\cos(0)\cos(0) + \cos(2\pi m)\cos(2\pi n)) \\
&= \frac{1}{r}\sum_{i=0}^{r-1} \cos\left(\frac{2\pi i}{r}m\right)\cos\left(\frac{2\pi i}{r}n\right) + \cos(2\pi m)\cos(2\pi n) \\
&\quad + \frac{1}{r}\sum_{i=1}^{r} \cos\left(\frac{2\pi i}{r}m\right)\cos\left(\frac{2\pi i}{r}n\right) + \cos(0)\cos(0) \\
&\quad - \frac{1}{r}(\cos(0)\cos(0) + \cos(2\pi m)\cos(2\pi n)) \\
&= \frac{1}{r}\sum_{i=0}^{r-1} \cos\left(\frac{2\pi i}{r}m\right)\cos\left(\frac{2\pi i}{r}n\right) + \cos\left(\frac{2\pi r}{r}m\right)\cos\left(\frac{2\pi r}{r}n\right) \\
&\quad + \frac{1}{r}\sum_{i=1}^{r} \cos\left(\frac{2\pi i}{r}m\right)\cos\left(\frac{2\pi i}{r}n\right) + \cos(0)\cos(0) \\
&\quad - \frac{1}{r}(\cos(0)\cos(0) + \cos(2\pi m)\cos(2\pi n))
\end{aligned}$$

$$= \frac{2}{r}\sum_{i=0}^{r}\left(\cos\left(\frac{2\pi i}{r}m\right)\cos\left(\frac{2\pi i}{r}n\right)\right) - \frac{1}{r}(\cos(0)\cos(0) + \cos(2\pi m)\cos(2\pi n))$$

Since m and n are integers

$$= \frac{2}{r}\sum_{i=0}^{r}\left(\cos\left(\frac{2\pi i}{r}m\right)\cos\left(\frac{2\pi i}{r}n\right)\right) - \frac{2}{r} \tag{4.5}$$

We will now present an approximation of the Kronecker delta function that has only the first term in the right hand side of Equation 4.5. Equation 4.5 has two terms in the left hand side. The first term is a sum of cosine functions and the second term is a constant. We want approximation of the Kronecker delta function that has only the first term. Let $\hat{\delta}_{mn}$ be an approximation of δ_{mn} that has only the first term, such that $\hat{\delta}_{mn}$ is defined as follows:

$$\hat{\delta}_{mn} \doteq \frac{2}{r}\sum_{i=0}^{r}\left(\cos\left(\frac{2\pi i}{r}m\right)\cos\left(\frac{2\pi i}{r}n\right)\right) \tag{4.6}$$

Substituting Equation 4.6 to Equation 4.5, we have:

$$\tilde{\delta}_{mn} = \hat{\delta}_{mn} - \frac{2}{r} \tag{4.7}$$

We can further show that in the limit of r, $\hat{\delta}_{mn}$ is equal to δ_{mn}. Applying limit to both sides of the above equation, we have:

$$\begin{aligned}\lim_{r\to\infty}\tilde{\delta}_{mn} &= \lim_{r\to\infty}\hat{\delta}_{mn} - \lim_{r\to\infty}\frac{2}{r} \\ &= \lim_{r\to\infty}\hat{\delta}_{mn} - 0\end{aligned} \tag{4.8}$$

Since, $\lim_{r\to\infty}\tilde{\delta}_{mn} = \delta_{mn}$, we have:

$$\lim_{r\to\infty}\hat{\delta}_{mn} = \delta_{mn} \tag{4.9}$$

4.1.2 Derivation of the Approximation

We will now use the approximation of the Kronecker delta function in Equation 4.6 to approximate the recurrent state update in Equation 4.1. We start by representing the recurrent state $\mathbf{C}_t$ as a sum of outer products. Starting with Equation 4.1:

$$\mathbf{C}_t = \big((1-\beta_t)\otimes(1-\gamma_t)\big)\odot\mathbf{C}_{t-1} + \big(\beta_t\odot\mathbf{v}_t\big)\otimes\big(\gamma_t\odot\mathbf{k}_t\big)$$

Recursively expanding $\mathbf{C}_{t-1}$

$$\begin{aligned}
&= \Big((\beta_t\odot\mathbf{v}_t)\otimes(\gamma_t\odot\mathbf{k}_t)\Big) + \Big((1-\beta_t)\otimes(1-\gamma_t)\Big)\odot\mathbf{C}_{t-1} \\
&= \Big((\beta_t\odot\mathbf{v}_t)\otimes(\gamma_t\odot\mathbf{k}_t)\Big) \\
&\quad + \Big((1-\beta_t)\otimes(1-\gamma_t)\Big)\odot\Big((\beta_{t-1}\odot\mathbf{v}_{t-1})\otimes(\gamma_{t-1}\odot\mathbf{k}_{t-1}) + \big((1-\beta_{t-1})\otimes(1-\gamma_{t-1})\big)\odot\mathbf{C}_{t-2}\Big) \\
&= \Big((\beta_t\odot\mathbf{v}_t)\otimes(\gamma_t\odot\mathbf{k}_t)\Big) + \Big((1-\beta_t)\otimes(1-\gamma_t)\Big)\odot\Big((\beta_{t-1}\odot\mathbf{v}_{t-1})\otimes(\gamma_{t-1}\odot\mathbf{k}_{t-1})\Big) \\
&\quad + \Big((1-\beta_t)\otimes(1-\gamma_t)\Big)\odot\Big((1-\beta_{t-1})\otimes(1-\gamma_{t-1})\Big)\odot\mathbf{C}_{t-2}
\end{aligned}$$

Since, $(\mathbf{a}\otimes\mathbf{b})\odot(\mathbf{c}\otimes\mathbf{d}) = (\mathbf{a}\odot\mathbf{c})\otimes(\mathbf{b}\odot\mathbf{d})$ for arbitrary vectors $\mathbf{a},\ \mathbf{b},\ \mathbf{c},\ \mathbf{d}$, we can rewrite the above equation as follows:

$$\begin{aligned}
\mathbf{C}_t &= \Big((\beta_t\odot\mathbf{v}_t)\otimes(\gamma_t\odot\mathbf{k}_t)\Big) + \Big(\big((1-\beta_t)\odot\beta_{t-1}\odot\mathbf{v}_{t-1}\big)\otimes\big((1-\gamma_t)\odot\gamma_{t-1}\odot\mathbf{k}_{t-1}\big)\Big) \\
&\quad + \Big(\big((1-\beta_t)\odot(1-\beta_{t-1})\big)\otimes\big((1-\gamma_t)\odot(1-\gamma_{t-1})\big)\Big)\odot\mathbf{C}_{t-2}
\end{aligned}$$

Recursively expanding further

$$\begin{aligned}
&= \Big((\beta_t\odot\mathbf{v}_t)\otimes(\gamma_t\odot\mathbf{k}_t)\Big) + \Big(\big((1-\beta_t)\odot\beta_{t-1}\odot\mathbf{v}_{t-1}\big)\otimes\big((1-\gamma_t)\odot\gamma_{t-1}\odot\mathbf{k}_{t-1}\big)\Big) + \\
&\quad + \Big(\big((1-\beta_t)\odot(1-\beta_{t-1})\odot\beta_{t-1}\odot\mathbf{v}_{t-2}\big)\otimes\big((1-\gamma_t)\odot(1-\gamma_{t-1})\odot\gamma_{t-2}\odot\mathbf{k}_{t-2}\big)\Big) + \dots
\end{aligned}$$

We can introduce variables $\mathbf{l}_i$ and $\mathbf{m}_i$, for $i = 0, 1, \dots, t$ to rewrite the above equation as a sum of outer products:

$$\mathbf{C}_t = \sum_{i=0}^{t} \mathbf{l}_i\otimes\mathbf{m}_i \tag{4.10}$$

where

$$\mathbf{l}_i = \prod_{j=i+1}^{t}(1-\beta_j)\odot\beta_i\odot\mathbf{v}_i \tag{4.11}$$

$$\mathbf{m}_i = \prod_{j=i+1}^{t}(1-\gamma_j)\odot\gamma_i\odot\mathbf{k}_i \tag{4.12}$$

Next, we introduce the approximate Kronecker delta function in Equation 4.6 to approximate the sum of outer products in Equation 4.10. Continuing from Equation 4.10, we have:

$$\begin{aligned}\mathbf{C}_t &= \sum_{i=0}^{t} \mathbf{l}_i \otimes \mathbf{m}_i \\ &= \sum_{j=0}^{t}\sum_{i=0}^{t} \delta_{ij} \mathbf{l}_i \otimes \mathbf{m}_j\end{aligned}$$

Replacing $\delta_{i,j}$ with $\hat{\delta}_{i,j}$ we can have an approximation $\tilde{\mathbf{C}}_t$ of $\mathbf{C}_t$ as follows:

$$\begin{aligned}\mathbf{C}_t \approx \tilde{\mathbf{C}}_t &= \sum_{j=0}^{t}\sum_{i=0}^{t} \hat{\delta}_{ij} \mathbf{l}_i \otimes \mathbf{m}_j \\ &\text{Using Equation 4.6} \\ &= \frac{2}{r}\sum_{j=0}^{t}\sum_{i=0}^{t}\sum_{k=0}^{r} \cos\left(\frac{2\pi k}{r} i\right) \cos\left(\frac{2\pi k}{r} j\right) \mathbf{l}_i \otimes \mathbf{m}_j \\ &\text{Rearranging the order of summations} \\ &= \frac{2}{r}\sum_{k=0}^{r}\sum_{j=0}^{t}\sum_{i=0}^{t} \cos\left(\frac{2\pi k}{r} i\right) \cos\left(\frac{2\pi k}{r} j\right) \mathbf{l}_i \otimes \mathbf{m}_j\end{aligned}$$

Let $\omega_k \doteq \cos\left(\frac{2\pi k}{r}\right)$, we then have:

$$\tilde{\mathbf{C}}_t = \frac{2}{r}\sum_{k=0}^{r}\sum_{j=0}^{t}\sum_{i=0}^{t} \cos(\omega_k i)\cos(\omega_k j)\, \mathbf{l}_i \otimes \mathbf{m}_j$$

Since $(ab)(\mathbf{c} \otimes \mathbf{d}) = (a\mathbf{c}) \otimes (b\mathbf{d})$ for scalars a, b and vectors $\mathbf{c}, \mathbf{d}$, we can then write:

$$\tilde{\mathbf{C}}_t = \frac{2}{r}\sum_{k=0}^{r}\sum_{j=0}^{t}\sum_{i=0}^{t} (\cos(\omega_k i)\, \mathbf{l}_i) \otimes (\cos(\omega_k j)\, \mathbf{m}_j)$$

Since $(\mathbf{a} + \mathbf{b}) \otimes \mathbf{c} = \mathbf{a} \otimes \mathbf{c} + \mathbf{b} \otimes \mathbf{c}$ for vectors $\mathbf{a}, \mathbf{b}, \mathbf{c}$, we can then write:

$$\begin{aligned}\tilde{\mathbf{C}}_t &= \frac{2}{r}\sum_{k=0}^{r}\left(\sum_{i=0}^{t} \cos(\omega_k i)\, \mathbf{l}_i\right) \otimes \left(\sum_{i=0}^{t} \cos(\omega_k i)\, \mathbf{m}_i\right) \\ &\text{Using Equation 4.11 and 4.12} \\ &= \frac{2}{r}\sum_{k=0}^{r}\left(\sum_{i=0}^{t} \cos(\omega_k i) \prod_{j=i+1}^{t} (1 - \beta_j) \odot \beta_i \odot \mathbf{v}_i\right) \otimes \left(\sum_{i=0}^{t} \cos(\omega_k i) \prod_{j=i+1}^{t} (1 - \gamma_j) \odot \gamma_i \odot \mathbf{k}_i\right)\end{aligned} \tag{4.13}$$

Next, we simplify the above equation and rewrite it in a recurrent form. Let $\tilde{\mathbf{v}}_t^k$ and $\tilde{\mathbf{k}}_t^k$ be defined as:

$$\tilde{\mathbf{v}}_t^k \doteq \sum_{i=0}^{t} \cos(\omega_k i) \prod_{j=i+1}^{t} (1-\beta_j) \odot \beta_i \odot \mathbf{v}_i \tag{4.14}$$

$$\tilde{\mathbf{k}}_t^k \doteq \sum_{i=0}^{t} \cos(\omega_k i) \prod_{j=i+1}^{t} (1-\gamma_j) \odot \gamma_i \odot \mathbf{k}_i \tag{4.15}$$

We can then rewrite Equation 4.13 in terms of $\tilde{\mathbf{v}}_t^k$ and $\tilde{\mathbf{k}}_t^k$ as follows:

$$\tilde{\mathbf{C}}_t = \frac{2}{r} \sum_{k=0}^{r} \tilde{\mathbf{v}}_t^k \otimes \tilde{\mathbf{k}}_t^k \tag{4.16}$$

It is possible to regroup the terms in the above equations and derive a recursive relationship of $\tilde{\mathbf{v}}_t^k$ and $\tilde{\mathbf{k}}_t^k$ with respect to $\tilde{\mathbf{v}}_{t-1}^k$ and $\tilde{\mathbf{k}}_{t-1}^k$ as follows:

$$\begin{aligned}
\tilde{\mathbf{v}}_t^k &= \sum_{i=0}^{t} \cos(\omega_k i) \prod_{j=i+1}^{t} (1-\beta_j) \odot \beta_i \odot \mathbf{v}_i \\
&= \cos(\omega_k t)\beta_t \odot \mathbf{v}_t + \sum_{i=0}^{t-1} \cos(\omega_k i) \prod_{j=i+1}^{t} (1-\beta_j) \odot \beta_i \odot \mathbf{v}_i \\
&\text{Taking common } (1-\beta_t) \\
&= \cos(\omega_k t)\beta_t \odot \mathbf{v}_t + (1-\beta_t) \sum_{i=0}^{t-1} \cos(\omega_k i) \prod_{j=i+1}^{t-1} (1-\beta_j) \odot \beta_i \odot \mathbf{v}_i \\
&\text{Replacing with } \tilde{\mathbf{v}}_{t-1}^i \\
&= \cos(\omega_k t)\beta_t \odot \mathbf{v}_t + (1-\beta_t) \odot \tilde{\mathbf{v}}_{t-1}^k
\end{aligned} \tag{4.17}$$

Similarly,

$$\begin{aligned}
\tilde{\mathbf{k}}_t^k &= \sum_{i=0}^{t} \cos(\omega_k i) \prod_{j=i+1}^{t} (1-\gamma_j) \odot \gamma_i \odot \mathbf{k}_i \\
&= \cos(\omega_k t)\gamma_t \odot \mathbf{k}_t + \sum_{i=0}^{t-1} \cos(\omega_k i) \prod_{j=i+1}^{t} (1-\gamma_j) \odot \gamma_i \odot \mathbf{k}_i \\
&\text{Taking common } (1-\gamma_t) \\
&= \cos(\omega_k t)\gamma_t \odot \mathbf{k}_t + (1-\gamma_t) \sum_{i=0}^{t-1} \cos(\omega_k i) \prod_{j=i+1}^{t-1} (1-\gamma_j) \odot \gamma_i \odot \mathbf{k}_i
\end{aligned}$$

Replacing with $\tilde{\mathbf{k}}^i_{t-1}$

$$= \cos(\omega_k t)\gamma_t \odot \mathbf{k}_t + (1-\gamma_t) \odot \tilde{\mathbf{k}}^k_{t-1} \tag{4.18}$$

Using recursive relationships in Equation 4.17 and 4.18, we can now present the final approximation. For a given r, we maintain recurrent states $\tilde{\mathbf{v}}^k_{t-1}$ and $\tilde{\mathbf{k}}^k_{t-1}$ for $k = 0, 1, 2, \ldots, r$. For $\omega_k \doteq \frac{2\pi k}{r}$, and assuming $\tilde{\mathbf{v}}^i_0$ and $\tilde{\mathbf{k}}^i_0$ are initialized as zeros, the recurrent updates to $\tilde{\mathbf{v}}^i_t$ and $\tilde{\mathbf{k}}^i_t$ and further the approximation to $\mathbf{C}_t$ are given by:

$$\mathbf{C}_t \approx \tilde{\mathbf{C}}_t = \frac{2}{r}\sum_{k=0}^{r} \tilde{\mathbf{v}}^{\mathbf{k}}_{\mathbf{t}} \otimes \tilde{\mathbf{k}}^k_t \tag{4.19}$$

where, for $k = 0, 1, 2, \ldots, r$ we have:

$$\tilde{\mathbf{v}}^k_t \doteq \cos(\omega_k t)\beta_t \odot \mathbf{v}_t + (1-\beta_t) \odot \tilde{\mathbf{v}}^k_{t-1} \tag{4.20}$$

$$\tilde{\mathbf{k}}^k_t \doteq \cos(\omega_k t)\gamma_t \odot \mathbf{k}_t + (1-\gamma_t) \odot \tilde{\mathbf{k}}^k_{t-1} \tag{4.21}$$

Since $\lim_{r\to\infty} \hat{\delta}_{mn} = \delta_{mn}$, it follows that $\lim_{r\to\infty} \tilde{\mathbf{C}}_t = \mathbf{C}_t$. Unlike Equation 4.1, Equation 4.20 and 4.21 define a recurrence over vectors instead of matrices, and if $r << d$, the recurrence is much more efficient in space than the recurrence in Equation 4.1. We leave it to future work to formally derive the approximation error. In Section 4.3 and further in Chapter 5 we will show that in-practice r can be chosen to be a small number, without compromising the quality of the approximation and the overall performance of the model.

Lastly, since the current state $\tilde{\mathbf{C}}_t$ could be represented as a sum of outer products in a non-recurrent manner, we can avoid explicitly calculating $\tilde{\mathbf{C}}_t$ and instead calculate the attention output $\mathbf{a}_t$ as follows:

$$\mathbf{a}_t \doteq \frac{\sum_{k=0}^{r} \tilde{\mathbf{v}}^k_t \left(\left(\tilde{\mathbf{k}}^k_t\right)^\intercal \mathbf{q}_t\right)}{2r(\mathbf{s}_t^\intercal \mathbf{q}_t)} \tag{4.22}$$

4.2 Approximate Recurrent Linear Transformer (AReLiT)

We introduce the modified self-attention mechanism into the ReLiT self-attention described in Algorithm 5 and call this new algorithm Approximate Recurrent Linear Transformer (AReLiT). Changes from Algorithm 5 are highlighted in blue. The algorithm maintains a set of vectors $\tilde{\mathbf{k}}^0_{t-1}, ..., \tilde{\mathbf{k}}^r_{t-1} \in \mathbb{R}^{\eta d_h}$, $\tilde{\mathbf{v}}^0_{t-1}, ..., \tilde{\mathbf{v}}^r_{t-1} \in \mathbb{R}^{d_h}$, and $\mathbf{s}_{t-1} \in \mathbb{R}^{\eta d_h}$ as the recurrent state at a given time-step t. The number of vectors stored could be controlled by modifying the hyperparameter r, which should ideally be set to a small value. The key, query, and value vectors are calculated similar to ReLiT. The recurrent state update is modified to use the approximation in Equation 4.19. At each time-step, the recurrent vectors are updated using an element-wise vector multiplication and

Algorithm 6 Approximate Recurrent Linear Transformer (AReLiT) Self-Attention (Streaming Data)

```
Input: $\mathbf{x}_t \in \mathbb{R}^d$, $\tilde{\mathbf{k}}^0_{t-1}, ..., \tilde{\mathbf{k}}^r_{t-1} \in \mathbb{R}^{\eta d_h}$, $\tilde{\mathbf{v}}^0_{t-1}, ..., \tilde{\mathbf{v}}^r_{t-1} \in \mathbb{R}^{d_h}$, and $\mathbf{s}_{t-1} \in \mathbb{R}^{\eta d_h}$
Parameters: $\mathbf{W}_K, \mathbf{W}_Q, \mathbf{W}_V, \mathbf{W}_\beta, \mathbf{W}_\gamma \in \mathbb{R}^{d \times d_h}$ and $\mathbf{W}_{p_1}, \mathbf{W}_{p_2}, \mathbf{W}_{p_3} \in \mathbb{R}^{d \times \eta}$

1: Assume $\mathbf{s}_0 \leftarrow 0, \mathbf{C}_0 \leftarrow 0$.
                                                {Calculate Keys}
2: $\tilde{\mathbf{k}}_t \leftarrow \mathbf{W}_K^\intercal \mathbf{x}_t$
3: $\mathbf{k}_t \leftarrow \mathit{flatten}(\mathit{relu}(\mathbf{W}_{p_1}^\intercal \mathbf{x}_t) \otimes \mathit{relu}(\tilde{\mathbf{k}}_t))$
                                                {Calculate Queries}
4: $\tilde{\mathbf{q}}_t \leftarrow \mathbf{W}_Q^\intercal \mathbf{x}_t$
5: $\mathbf{q}_t \leftarrow \mathit{flatten}(\mathit{relu}(\mathbf{W}_{p_2}^\intercal \mathbf{x}_t) \otimes \mathit{relu}(\tilde{\mathbf{q}}_t))$
                                                {Calculate Values}
6: $\mathbf{v}_t \leftarrow \mathbf{W}_V^\intercal \mathbf{x}_t$
                                                {Generate Gating Vectors}
7: $\beta_t \leftarrow \sigma_g(\mathbf{W}_\beta^\intercal \mathbf{x}_t)$
8: $\tilde{\gamma}_t \leftarrow \mathbf{W}_\gamma^\intercal \mathbf{x}_t$
9: $\gamma_t \leftarrow \mathit{flatten}(\sigma_g(\mathbf{W}_{p_3}^\intercal \mathbf{x}_t) \otimes \sigma_g(\tilde{\gamma}_t))$
                                                {Update Memory}
10: for $i \leftarrow 0$ to $r$ in parallel, do
11:     $\omega_i \leftarrow (2\pi i)/r$
12:     $\tilde{\mathbf{v}}^i_t \leftarrow \tilde{\mathbf{v}}^i_{t-1} \odot (1 - \beta_t) + \cos(\omega_i t)\, (\beta_t \odot \mathbf{v}_t)$
13:     $\tilde{\mathbf{k}}^i_t \leftarrow \tilde{\mathbf{k}}^i_{t-1} \odot (1 - \gamma_t) + \cos(\omega_i t)\, (\gamma_t \odot \mathbf{k}_t)$
14: end for
15: $\mathbf{s}_t \leftarrow (1 - \gamma_t) \odot \mathbf{s}_{t-1} + \gamma_t \odot \mathbf{k}_t$
                                                {Calculate Attention Vector}
16: $\mathbf{a} \leftarrow \sum_{i=0}^{r} \tilde{\mathbf{v}}^i_t \left( \tilde{\mathbf{k}}^{i\intercal}_t \mathbf{q}_t \right)$
17: $\mathbf{b} \leftarrow 2r(\mathbf{s}_t^\intercal \mathbf{q}_t)$
18: $\mathbf{a}_t \leftarrow \mathbf{a}/\mathbf{b}$
Output: $\mathbf{a}_t \in \mathbb{R}^{d_h}$, $\tilde{\mathbf{k}}^0_t, ..., \tilde{\mathbf{k}}^r_t \in \mathbb{R}^{\eta d_h}$, $\tilde{\mathbf{v}}^0_t, ..., \tilde{\mathbf{v}}^r_t \in \mathbb{R}^{d_h}$, and $\mathbf{s}_t \in \mathbb{R}^{\eta d_h}$
```

addition operations (lines 10-14). The operation on each recurrent vector could be executed in parallel. The attention output is calculated without ever explicitly calculating $\tilde{\mathbf{C}}_t$ (lines 16-18).

The computational complexity of Algorithm 5 is compared with that of Algorithm 6 in Table 4.1. The computational complexity is for processing a single element in sequence that is presented in a streaming fashion. Unlike Algorithm 5, Algorithm 6 does not maintain matrix as a recurrent memory, reducing the space complexity to $\mathcal{O}(r\eta d)$. Further, the time complexity of Algorithm 6 is $\mathcal{O}(d^2 + r\eta d)$, which is slightly better than that of Algorithm 5 as the feature map hyperparameter η is now decoupled from the quadratic d^2 term.

We revisit the comparison presented in end of Section 3.3 and elaborate on the computational efficiency of AReLiT by comparing in terms of the number of floating point operations and space required to store the previous recurrent state. For GTrXL self-attention we consider the hyperparameter values from Parisotto et al. [2020]: $M = 512$, $d = 256$, and $d_h = 64$. For ReLiT, we can consider $d = 256$, $d_h = 64$ and $\eta = 4$. For AReLiT, we consider $d = 256$, $d_h = 64$, $\eta = 4$ and $r = 7$. We chose $r = 7$ as this is configuration we used for the Memory Maze experiments presents in Chapter 5. AReLiT self-attention is roughly 237.83 times faster than GTrXL self-attention and it uses 52.51 times less space. Further, ARELIT self-attention is roughly 2.14 times faster than ReLiT self-attention and it uses 5.9 times less space.

Table 4.1: Space and time complexity of AReLiT, ReLiT, Linear Transformer and GTrXL for a processing a single element in streaming sequence. (M: memory size in GTrXL, d: representation dimension, d_k feature map dimension in Linear Transformer, η: feature map hyperparameter in ReLiT and AReLiT, r: approximation parameter in AReLiT, L: number of encoder layers)

	Space	**Time**	**Potential Context Length**
GTrXL	$\mathcal{O}(Md)$	$\mathcal{O}(M\ d^2)$	$\mathcal{O}(LM)$
Linear Transformer	$\mathcal{O}(d_k d)$	$\mathcal{O}(d_k d)$	∞
ReLiT	$\mathcal{O}(\eta d^2)$	$\mathcal{O}(\eta d^2)$	∞
AReLiT	$\mathcal{O}(r\eta d)$	$\mathcal{O}(d^2 + r\eta d)$	∞

4.3 Effect of r on the Quality of Approximation

We empirically evaluate the effect of r on the quality of the approximation of the current state matrix $\mathbf{C}_t$. Ideally, we want to set r to a small value as the space complexity of AReLiT is directly proportional to r. We consider a synthetic example where the value $\mathbf{v}_t$ and key $\mathbf{k}_t$ at each time step are sampled randomly from a normal distribution. We set the embedding dimension d to 128 and randomly sample values and keys for 100 timesteps. Instead of using vectors γ_t and β_t for gating at every timestep, we use a constant value c. We then compare the difference between the current state matrix $\mathbf{C}_t$ computed using the exact method in Equation 3.5, with the current state matrix $\tilde{\mathbf{C}}_t$ computed using the approximate method in Equation 4.19 at the 100th time-step. We use the Frobenius norm to measure the difference between the two matrices. We repeat the experiment for different values of r and c. For each configuration, we report the mean error across 50 independent runs. Figure 4.1 shows the results of this experiment. We observe that the error in approximation decreases with increasing value of r. For most values of r and c, the approximation error is low. This is useful since it allows us to set r to a small value, thereby reducing the space complexity of the model. In fact, in the largest experiments described in this thesis, we set r to 7. Interestingly, we observe periodic bands in the error plot. It is possible that this is due to the periodicity of

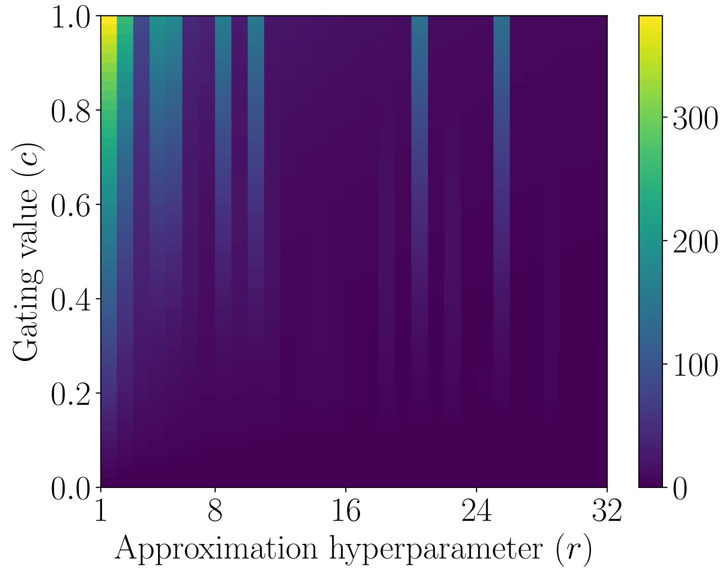

Figure 4.1: Error in approximating the current state $\mathbf{C}_t$ for different values r and gating at $t = 100$ for randomly sampled values and keys.

the cosine functions used in the attention mechanism. We leave further exploration around the theoretical nature of the error in approximation for future work.

4.4 Parallelization during Training

Transformers are naturally designed for parallelism over a sequence of input data, as the self-attention operation does not have dependencies between different parts of the input sequence. In Chapter 3 and 4, we introduced our proposed Algorithms in the scenario where the input sequence is presented in a streaming fashion. It is essential to consider the parallelizability of transformer architectures, when the input sequence is presented in a batched fashion. Such a scenario is common in practice, as most existing actor-critic approaches such as PPO and A2C (Schulman et al. [2017], Mnih et al. [2016]) estimate gradient updates to the actor and critic using batches of trajectories collected through agent-environment interactions. Furthermore, most modern hardware accelerators, such as GPUs and TPUs, excel in handling parallelizable algorithms, and parallelization is vital for effectively training large models.

Extension of Algorithm 5 and 6 to accommodate parallelization over a sequence of inputs is straightforward, depending on whether the computation has dependencies on the previous state or not. The majority of the computations in both algorithms, which involve calculating keys, queries, values, gating vectors, and the attention vector, do not depend on the previous state and can be parallelized over the sequence. The only part of the algorithm that depends on the previous state is the update of the current state. In Algorithm 5, this is done from lines 13-14, and in Algorithm 6, from lines 10-15. The update of the current state in both algorithms is implemented as a discounted sum over vectors. These operations has computational complexity of $\mathcal{O}(Nd)$, for a sequence of length N and embedding dimension d. Parallelization of these operations could be possible using associative scan (see Blelloch [1990]), but we did not explore this in our implementations and leave this as a future direction for research. Despite this remaining sequential bottleneck, we still observe a significant speedup in overall training times. We demonstrate this empirically in Section 5.2.3. Due to the parallelizability of the majority of computations, our training process achieves noteworthy speedups compared to a transformer baseline that utilizes vanilla self-attention.

Chapter 5

Results

In this chapter, we present empirical evaluations of our proposed approaches in partially observable reinforcement learning problems. We start with a diagnostic environment with a binary observation space and a single source of partial observability. We then extend our evaluation to a more complex environment with pixel observations and multiple sources of partial observability.

5.1 Learning to Remember with Binary Observations

In this section, we evaluate our proposed architectures, ReLiT and AReLiT, in a reinforcement learning setting that requires remembering a single bit of information from the past. First, we describe a diagnostic environment, called the T-Maze (Bakker [2001]). Then we evaluate several agents on their ability to learn long context dependencies in a reinforcement learning scenario. Our first experiment highlights the potential challenges of applying existing transformer approaches such as GTrXL to RL. Our second experiment compares the performance of ReLiT and AReLiT with LSTM, GRU and GTrXL. We demonstrate the effectiveness of our architecture in learning long context dependencies while being more computationally efficient than GTrXL.

5.1.1 The T-Maze Environment

The T-Maze environment is a simple environment used to evaluate an agent's ability to learn long context dependencies in a reinforcement learning scenario. In this environment, an agent is posed with the problem of remembering a single cue, which is shown only at the beginning of an episode. As shown in Figure 5.1, at the beginning of an episode, the agent starts at the bottom of a T-shaped maze. In the first timestep, the agent receives a binary cue which is never shown again throughout the episode. The agent then travels through the maze until it reaches an intersection. At the

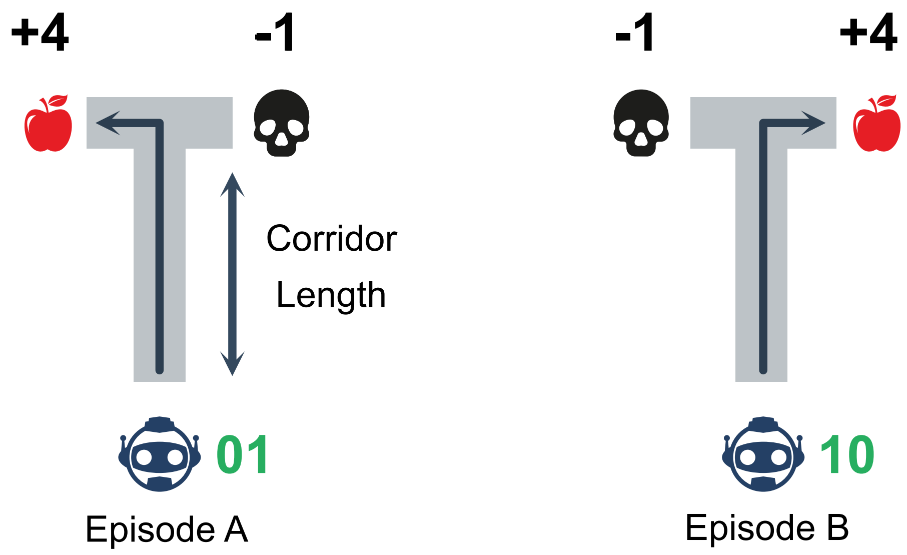

Figure 5.1: The T-Maze environment. The agent has to remember a binary cue (denoted by green text), shown only at the beginning of the episode, in order to take the correct turn at the intersection and receive a positive reward. The figure shows two possible episodes and the optimal path an agent must take. The agent's current location is provided as gray code encoding in the observation, along with distractor signals. The corridor length could be varied to increase the difficulty of the problem.

intersection, the agent has to recall the initial cue to take the correct action to receive a positive reward. The episode ends when the agent takes a turn at the intersection. The earliest version of this environment was proposed by Bakker [2001] which featured a grid-based environment with binary observations. Over the years various modifications to the T-Maze environment have been proposed, with low-dimensional vector observations (e.g. Osband et al. [2019], Morad et al. [2023]) to 3D environments featuring pixel observations (e.g. Nandy et al. [2018], Beattie et al. [2016]).

Following Bakker [2001], we consider a simpler T-Maze setting that isolates the problem of learning long-term dependencies from other challenges in the control setting. Figure 5.1 shows two possible episodes in the T-Maze environment. At each timestep, the agent receives a 16-bit binary observation. The first two bits correspond to the cue signal which is either 01 or 10 at the first timestep of an episode, depending on whether the reward is located at the left or right turn at the intersection, respectively. The cue bits are zero in all other timesteps. We consider the largest possible corridor length as 200. To encode the corridor information, the agent additionally receives 8-bit gray code encoding of its current location. The gray code encoding is zero at the beginning of an episode and is updated at each timestep. To make the problem more challenging, we added 6 noisy distractor bits to the observation. The distractor bits are sampled uniformly at random at each timestep. The agent can take one of the four possible discrete actions at each timestep:

up, down, left, or right. The agent receives a reward of -0.1 at each non-terminal timestep. At termination, the agent receives a reward of +4 for taking the correct turn and a reward of -1 for taking an incorrect turn. The reward of +4 is chosen to encourage the agent to take the correct turn at the intersection. The difficulty of this environment can be increased by increasing the corridor length. Increasing the corridor length requires the agent to remember the signal for a longer number of timesteps. Since the agent's observations include distractor bits, the agent also needs to learn to ignore the distractor bits and focus on the cue signal.

5.1.2 Experiment Setup

We describe the experiment pipeline used to evaluate the agents in the T-Maze environment. All the agents used in the T-Maze experiments are based on the A2C framework (Wu et al. [2017]), described in Section 2.2.2. We use 8 environment instances and collect data for 256 environment steps, totaling a batch size of 2048. We train each agent for 10M timesteps. To evaluate the agent's performance in this environment we calculate the success rate in the last 100K timesteps, that is the number of times the agent takes the correct turn. An agent that can remember the cue signal correctly across all trials should achieve a success rate of 1.0. On the other hand, an agent that is unable to remember the cue signal has a 50% probability of taking the incorrect turn and should achieve a score of 0.5. We evaluate each agent across 5 corridor lengths ranging from 120 to 200. We report the mean success rate across 50 seeds for each of the 5 corridor lengths.

For each agent, we tune the entropy coefficient and the learning rate. We discuss the exact set of hyperparameters and the sweeps in Appendix A.1. Each hyperparameter configuration is evaluated across 5 random seeds and 5 corridor lengths. We select the hyperparameter configuration that achieves the highest mean success rate across all corridor lengths and seeds.

We use a shared representation learning layer followed by separate actor and critic heads as the network architecture for all agents. The shared layer is either an RNN or a Transformer that takes in a sequence of binary observations and outputs a sequence of vectors as representations (see Figure 2.1 and 2.3). The actor head is implemented as a 2-layer neural network with tanh activations and a softmax output. The critic head is implemented as a 2-layer neural network with Tanh activations and a linear output. The dimension of the hidden layer for both the actor's and the critic's head is 128. To make the comparison fair, we chose the RNN and transformer architecture sizes in a way such that each agent has approximately 1.6M parameters. We discuss the architecture sizes in detail in Appendix A.1.

A few additional details are worth reporting for the purposes of reproducibility. We conducted all experiments using Python and implemented the agents using the Jax library (Bradbury et al. [2018]). Each agent is trained using 16-core machine with 12GB RAM. The network weights are initialized using orthogonal initialization (Saxe et al. [2013]). A single run using the slowest architecture takes around 20 hours to complete.

5.1.3 Experiment 1: Evaluating GTrXL in T-Maze

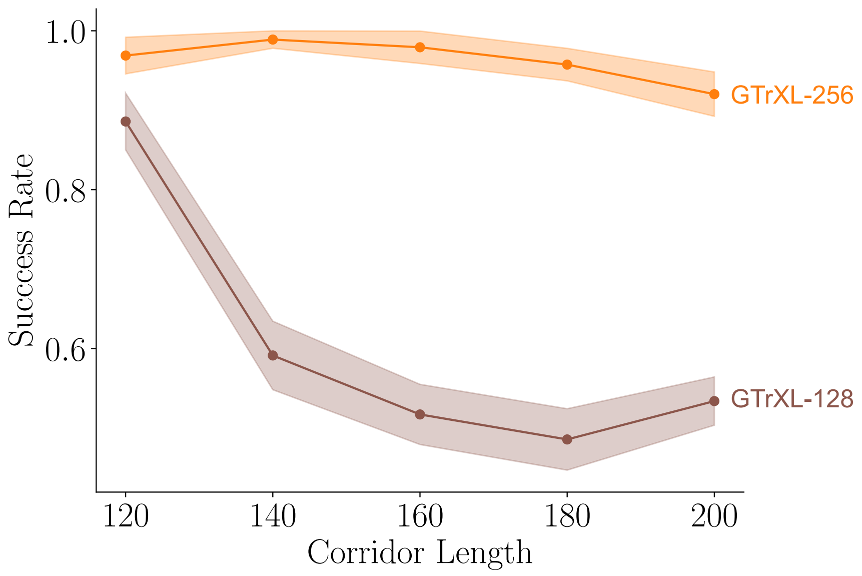

Figure 5.2: Mean success rate over 50 runs for a GTrXL agent with memory sizes 128 and 256 in the T-Maze environment. The shaded region represents the standard error. Both agents have a theoretical context larger than the largest corridor length considered but GTrXL-256 has access to the entire context as input. GTrXL-128 fails on corridor lengths above 120 despite having a theoretical context length of 512. GTrXL needs access to the entire context in order to learn long-term dependencies.

This experiment investigates how the memory size of GTrXL affects its ability to learn long-context dependencies in T-Maze. We consider two variants of GTrXL, one with $M = 128$ and the other with $M = 256$. We will refer to these variants as GTrXL-128 and GTrXL-256. Both architectures use 4 layers, which correspond to a theoretical context length above 200; for GTrXL-128 it is 512 and for GTrXL-256 it is 1024. Out of the two, however, only GTrXL-256 has M larger than the largest corridor length considered, that is it has access to the entire context as input. For corridor lengths above 120, GTrXL-128 does not have access to the entire context as input and must recover the cue signal from past activations. Our hypothesis is that GTrXL needs access to the entire context in order to achieve a high success rate in this environment. We expect GTrXL-256 to achieve a success rate close to 1.0 for all corridor lengths, while GTrXL-128 will achieve a success rate close to 1.0 for corridor lengths 120 and below, but the performance will drop for corridor lengths above 120.

The performance of the two GTrXL agents is presented in Figure 5.2. Across all corridor lengths, the GTrXL-256 achieves a success rate between 0.9-1.0. On the other hand, the GTrXL-128 achieves a success rate near 1.0 for corridor length of 120, but the success rate quickly drops to 0.5 for corridor lengths above 120.

The results presented support our hypothesis, suggesting that the GTrXL needs access to the entire context in order to learn long-term dependencies in this problem. It is also important to note that both agents have a theoretical context length above 200, which is the largest corridor length considered, suggesting that the theoretical context length is not often observed in practice.

Applying GTrXL to tasks with long context dependencies can have a significant computational cost. The maximum possible memory size of GTrXL is limited by the memory budget. Scaling to longer context lengths, however, requires increasing the memory budget. Storing a large portion of agent's history of interaction is not always feasible due to memory constraints.

5.1.4 Experiment 2: Evaluating ReLiT and AReLiT in T-Maze

This experiment highlights the effectiveness of ReLiT and AReLiT in learning long-context dependencies in T-Maze, comparing the performance of ReLiT and AReLiT with GTrXL-256, LSTM and GRU. We include LSTM and GRU because they are the most commonly used RNN architectures in reinforcement learning. We discuss the architecture sizes and the exact set of hyper-parameters in detail in Appendix A.1. For the ReLiT and AReLiT agents, we set the feature map hyperparameter η to 4. For ReLiT, we set the approximation hyper-parameter r to 1, which is the lowest possible value. We chose the lowest possible value for r because we want to evaluate the performance of AReLiT even when the approximation is poor. Our hypothesis is that ReLiT and AReLIT will perform close to a GTrXL-256 agent, achieving a success rate close to 1.0 for all corridor lengths, despite not having access to the entire context as input. Further, we expect LSTM and GRU to perform poorly for larger corridor lengths.

The results are presented in Figure 5.3. The LSTM agent achieves a success rate close to 0.5 for all the corridor lengths. The GRU agent achieves a success rate between 0.8-1.0 on corridor lengths between 120-180 but the performance starts dropping to 0.5 for corridor length above 180. The GTrXL-256 agent achieves a success rate between 0.9-1.0 for all corridor lengths. The ReLiT and AReLiT agents achieve a success rate between 0.8-1.0 for all corridor lengths while being lower than the GTrXL-256 agent.

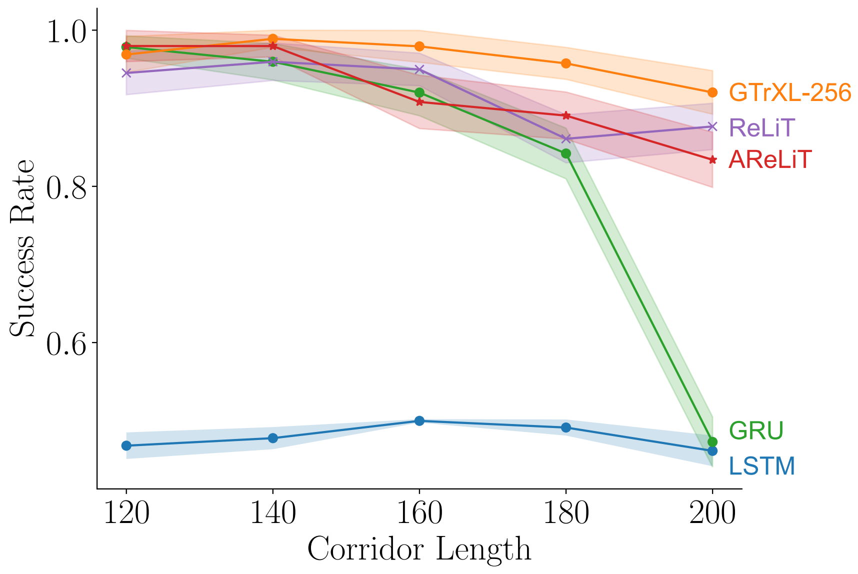

Figure 5.3: Mean success rate over 50 runs for ReLiT, AReLiT, LSTM, GRU, and GTrXL-256 agents in the T-Maze problem environment. The shaded region represents the standard error.

The results of this experiment support our hypothesis. The performance of the LSTM agent agrees with existing results in the literature (Bakker [2001]). Across all corridor lengths, ReLiT and AReLiT achieve a performance close to the GTrXL-256 agent, despite not having access to the entire context as input. The performance of ReLiT and AReLiT, however, is lower than the GTrXL-256 agent, which is not surprising as the GTrXL-256 agent has access to the entire context as input. Interestingly, even with the lowest possible approximation hyperparameter $r = 1$, AReLiT is able to achieve a performance close to ReLiT.

Table 5.1: Number of floating point operations and space required by the self-attention layer to process a single observation in T-Maze. The values are calculated for a single head. **Operations** represents roughly the number of floating point operations. **Space** represents the number of floating point values required to store the previous state of the self-attention layer.

	Operations	**Space**
GTrXL-256	8.4×10^6	32768
ReLiT	2.1×10^5	16640
AReLiT	6.7×10^4	896

AReLIT and ReLiT are both more computationally efficient than GTrXL-256 in this environment. Table 5.1 presents the total number of floating point operations required to process a single observation in T-Maze, for a single attention head. It also lists the number of floating point values required to store the previous recurrent state. For the chosen architecture configurations, ReLiT is roughly 40 times faster than GTrXL-256 self-attention and uses 1.96 times less space. On the other hand, AReLiT is roughly 125.1 times faster than GTrXL-256 self-attention and it uses 36.57 times less space.

5.2 Learning to Remember with Pixel Observations

This section investigates the capabilities of AReLiT in a more demanding setting with a pixel observation space that requires the retention of multiple pieces of past information. We use the Memory Maze (Pašukonis et al. [2023]) environment. In the previous section, we highlighted the computational advantages of our approaches in terms of the actual number of operations during a single inference step. In this section, we empirically demonstrate how these theoretical performance benefits materialize in the more complex RL frameworks, and how they translate into real-time performance.

5.2.1 The Memory Maze Environment

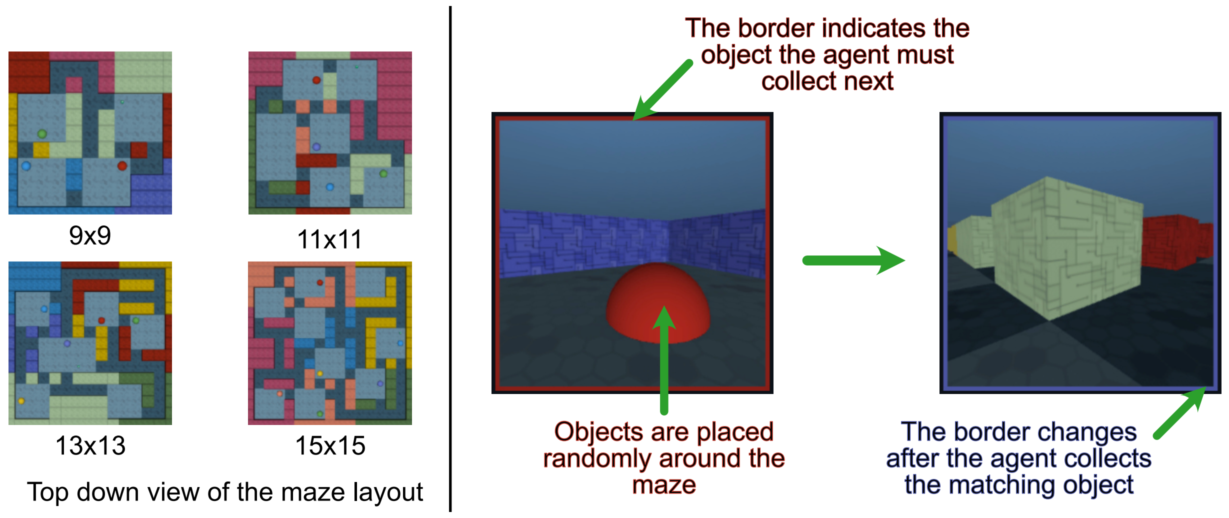
9x9
11x11
13x13
15x15
Top down view of the maze layout
The border indicates the object the agent must collect next
Objects are placed randomly around the maze
The border changes after the agent collects the matching object

Figure 5.4: The Memory Maze environment. On the left, we show a possible maze layout for all four Memory Maze configurations. The maze layout is randomized at each episode. On the right, we show two sample observations that the agent receives. The agent's observation at each time-step is 64 × 64 RGB pixels and the action space is discrete. The border color of the observation image indicates the target object color which the agent needs to find to receive a reward. After collecting the object, the border color changes, indicating the next target object. The episode lengths are fixed depending on the Memory Maze configuration, with larger configurations having longer episodes.

The Memory Maze environment evaluates an agent's long-term memory capabilities in a partially observable RL setting. Figure 5.4 illustrates this environment. The agent's observation at each time-step is an image with 64×64 RGB pixels, and the action space is discrete. In each episode, the agent starts in a randomly generated maze containing several objects of different colors. The agent's objective is to find the target object of a specific color, indicated by the border color in the observation image. Upon successfully touching the correct object, the agent receives a $+1$ reward, and the next random object is chosen as the new target. If the agent touches an object of the wrong color, there is no effect on the environment. The maze layout and object locations remain constant throughout the episode. Each episode lasts for a fixed amount of time. Since the maze layout is randomized at each episode, the agent must learn to quickly remember the maze layout, the target object locations, and the paths leading to them.

The Memory Maze environment features four different configurations. The four configurations have different maze sizes: 9×9, 11×11, 13×13, and 15×15. The sizes of the Memory Maze environments are deliberately designed to vary in difficulty, with larger environments being more challenging. The larger configurations also have a larger episode length.

5.2.2 Experiment Setup

We describe the experiment pipeline used to evaluate agents in the Memory Maze environment. All the agents used in the Memory Maze experiments are based on the Async-PPO framework (Petrenko et al. [2020]), described in Section 2.2.2. We train each agent for 100M environment steps. To measure the agent's performance, we plot the total episodic reward for the entire training duration. The total episodic reward is determined by the number of targets the agent can find within a single episode. An agent capable of remembering the maze layout, particularly the locations of the target objects and the paths leading to them, can efficiently navigate the maze and quickly reach the requested objects. We report results for 3 random seeds for each of the four Memory Maze configurations.

We use default PPO hyperparameters for the DMLab experiments by Schulman et al. [2015] and additionally tune the learning rate and the entropy coefficient. We discuss the exact set of hyperparameters and the sweeps in Appendix A.2. Each hyperparameter configuration is evaluated across 3 random seeds for 15M steps in the Memory Maze 11×11 environment. We select the hyperparameter configuration that achieves the highest mean episodic reward.

We adapt the architecture of the policy and critic networks used for the DMLab experiments described by Petrenko et al. [2020]. The architecture uses a ResNet convolutional neural network (He et al. [2016]) to extract features from the input image at a given time step. The sequence of features extracted from a trajectory of observations is then fed into an RNN or a transformer to learn a policy and a critic. We chose the architecture sizes in a way such that each agent has approximately 22M parameters. We discuss the architecture sizes in detail in Appendix A.2.

Additional experiment details are described for the purposes of reproducibility. We implemented all the agents in Python using the Sample Factory library (Petrenko et al. [2020]). Training a single agent run uses 32 parallel actors to collect experience data and a single learner process to update the policy parameters. We used a single A100 GPU for training the agent, and 12 CPU actors equipped with 80GB shared RAM to collect experiences. Training the slowest agent takes approximately 2.5 days.

5.2.3 Experiment 3: Evaluating AReLiT in Memory Maze

This experiment investigates the performance AReLiT compared to LSTM and GTrXL in Memory Maze. We follow a similar experiment pipeline as the T-Maze experiment. We consider a GTrXL agent with a memory size of 256, such that it can directly access a significant amount of past information without the need to retrieve it from a recurrent memory. For AReLiT, we consider a feature map size of $\eta = 4$ and an approximation hyperparameter of $r = 7$. We train all three agents on all four Memory Maze configurations. We use 4 maze sizes mirroring the choice of using longer and longer corridors in the T-Maze experiment. Our hypothesis is similar to the one we had in the T-Maze experiment. We expect to see differences between the methods to become more pronounced with larger mazes, just as we saw with longer corridors in T-Maze. We expect that LSTM would perform the worst for larger mazes. We expect the performance of AReLiT to be close to GTrXL, despite not having access to a significant amount of context as input.

The learning curves of the agents is presented in Figure 5.5. The asymptotic performance of all the three agents are close. The asymptotic performance of the LSTM agent is better than GTrXL in Memory Maze 9×9 and 11×11 and 13×13, while GTrXL is better than LSTM in Memory Maze 15×15. AReLiT is worser than GTrXL and LSTM in all the four configurations.

The results of this experiment do not match our hypothesis. From the results obtained in the T-Maze experiments, we expected both AReLiT and GTrXL agents to perform similarly, and outperform the LSTM agent, however, this is not the case. We speculate that the agent's are not effectively utilizing their long-term memory capabilities to solve the task.

AReLiT achieves higher training frames per second (FPS) and lower memory usage than GTrXL. These results are presented in Figure 5.6. FPS represents the number of observations processed by the learner per second, while memory usage denotes the peak GPU memory usage of the learner and worker processes. FPS and memory usage data were collected from a total of 12 agents for both AReLiT and GTrXL, where each agent was trained on the single-GPU configuration described earlier. The AReLiT agent achieves around 54% higher training FPS and 43% lower memory usage than the GTrXL agent.

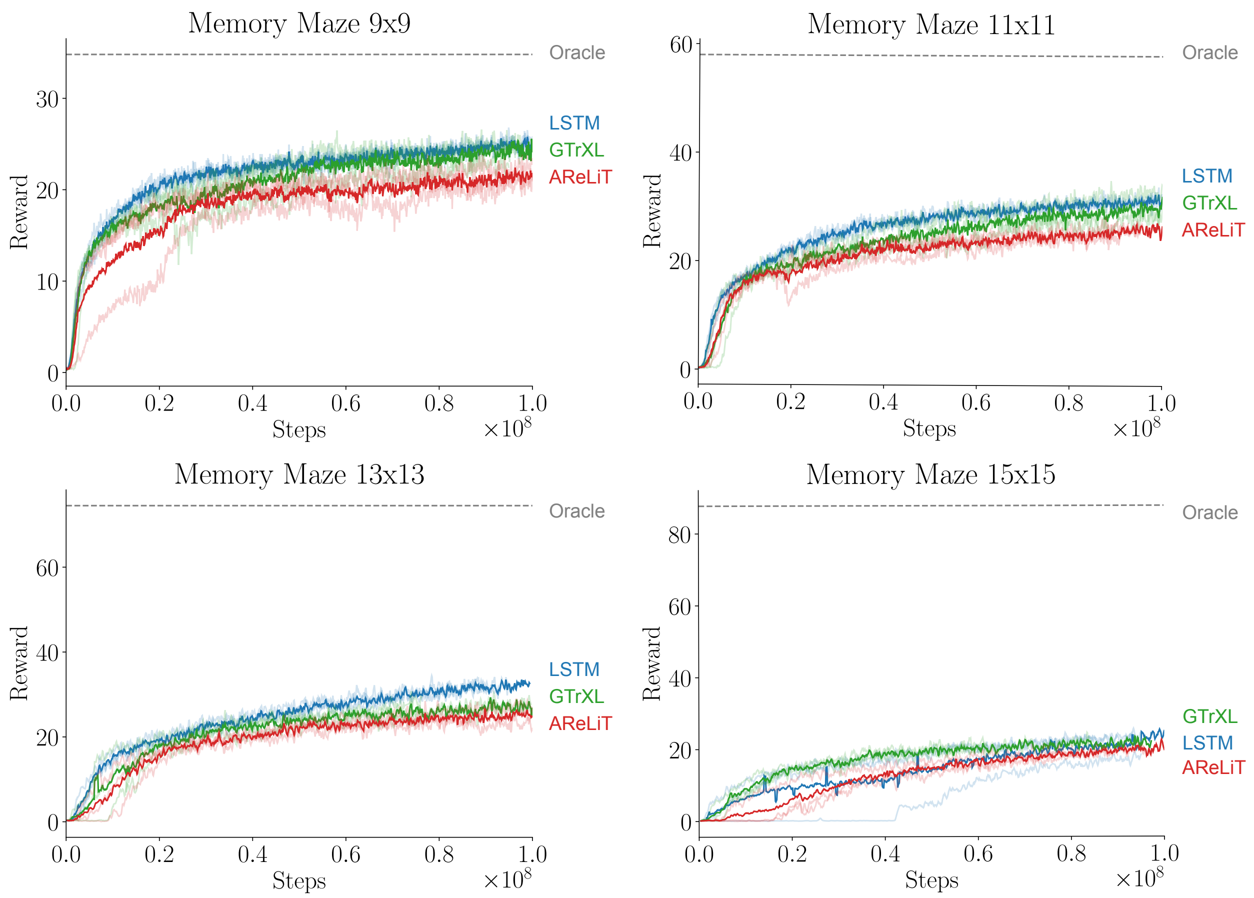

Figure 5.5: Learning curves of LSTM, GTrXL and AReLiT agents in the Memory Maze environment. The x-axis represents the number of environment steps, and the y-axis represents the total reward in an episode. Each agent is trained with 3 different random seeds. The bold lines represent the mean return across the 3 seeds, and the blurred lines represent the individual seeds. Each point is the average episodic reward over 1M environment steps. The dotted grey line represents the performance of an oracle agent that has access to the entire maze layout, target object locations and paths leading to them.

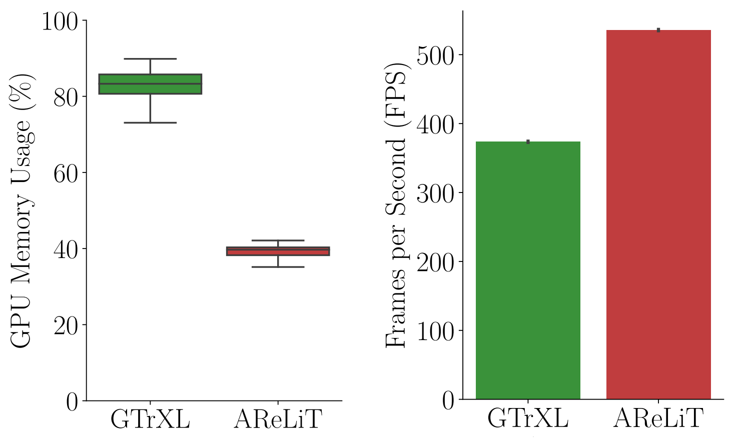

Figure 5.6: GPU memory usage and frames per second (FPS) comparison for GTrXL and AReLiT agents in the Memory Maze environment. The data is collected from 12 independent agents per architecture, each trained for approximately 2.5 days in different nodes, utilizing the same configuration.

5.2.4 Experiment 4: Evaluating Impact of GTrXL's Context in Memory Maze

This experiment evaluates the impact of GTrXL's context length in the Memory Maze environment. We showed earlier that GTrXL's performance is bottlenecked by the memory size in T-Maze. Our hypothesis is that a similar conclusion should hold in the Memory Maze environment. We expect that GTrXL with a larger memory size would outperform GTrXL with a smaller memory size. We should also be able to show that an AReLiT would outperform a GTrXL with a small memory size. To investigate this, we train two additional GTrXL agents with memory sizes of 64 and 128 in the Memory Maze 13×13 environment.

The learning curves of training the three memory sizes of GTrXL and AReLiT in the Memory Maze 13×13 environment is shown in Figure 5.7. Asymptotically, all four agents achieve similar performance. The individual learning curves, however, indicate that the GTrXL-64 agent is slower to converge than the GTrXL-128 and GTrXL-256 agents.

The results failed to provide sufficient evidence to support our hypothesis. The performance obtained by the three agents does not appear to be different. This observation leads us to the following speculation: the Memory Maze environment is too difficult for the agents to be able to utilize their long-term memory capabilities. The reward signal is sparse, which might make it difficult for the agent to learn long-term dependencies. It is also possible that learning long-term dependencies in navigation tasks is harder, in general, and longer training is necessary for the benefits of long-term memory to show.

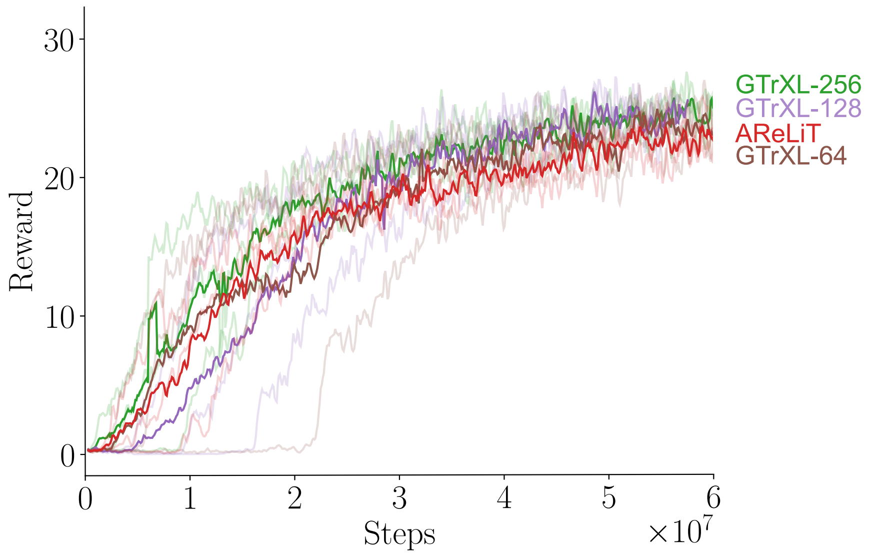

Figure 5.7: Learning curves of GTrXL agents with different memory sizes in the Memory Maze 13×13 environment. The x-axis represents the number of environment steps, and the y-axis represents the total reward in an episode. Each agent is trained with 3 different random seeds. The bold lines represent the mean return across the 3 seeds, and the blurred lines represent the individual seeds. Each point is the average episodic reward over 1M environment steps.

Chapter 6

Conclusion and Future Work

In this thesis, we introduce recurrent alternatives of the self-attention mechanism is transformers, called Recurrent Linear Transformer (ReLiT) and Approximate Recurrent Linear Transformer (AReLiT). We derived both approaches from the Linear Transformer self-attention, which was originally introduced as a way to reduce the computational complexity of canonical self-attention. Our first approach ReLiT, uses the gated structure of the GTrXL architecture and introduces a modified Linear Transformer self-attention. ReLiT introduces an outer-product-based gating mechanism into the Linear Transformer's self-attention mechanism, which selectively updates each index location of the recurrent state. Additionally, ReLiT introduces a parameterized kernel feature map that eliminates the need for an explicit feature map and learns the feature map from data. Our second approach, AReLiT, approximates the outer product calculation in ReLiT's self-attention mechanism, further reducing the computational complexity of ReLiT. AReLiT maintains a finite set of vectors as a recurrent state and replaces vector outer products with dot products, thereby reducing its computational complexity. ReLiT and AReLiT are RNNs, which means they have a cheap inference cost, but it features a self-attention mechanism that allows it to learn relationships far into the past.

We demonstrate the efficacy of both approaches in a partially observable reinforcement learning task called the T-Maze. The T-Maze environment features binary observations and the goal is to remember a single piece of information from several timesteps ago. We show that both ReLIT and AReLiT can learn to remember relationships far into the past. We compared our approach to RNNs such as LSTM and GRU, and a transformer baseline such as GTrXL. We highlight the computational advantages of our approach by showing that a limited context length in GTrXL's self-attention can be a bottleneck in such environments, and our approach is able is learn dependencies for longer durations much more efficiently.

We also provide results for our second approach, AReLiT, in a larger problem setting called the

Memory Maze, featuring pixel observations and multiple sources of partial observability. In this environment, however, we fail to demonstrate that a limited context GTrXL or an LSTM would perform worse than AReLiT. Interestingly, we observe that the performance of AReLiT, GTrXL and LSTM agents in the Memory Maze environment are close. We also varied the context length of a GTrXL agent and found that a reduced context length did not impact the performance. We speculate that the Memory Maze environment is more complex compared to the T-Maze environment, and the agents are not able to learn to utilize their context effectively. Additionally, we highlight the computational advantages of our approaches compared to a GTrXL agent, by empirically measuring the frames per second (FPS) and GPU memory usage. We show that our approach is able to achieve a higher FPS and use less GPU memory compared to a GTrXL agent.

The work presented in this thesis has the potential for several extensions:

- Adding auxiliary tasks to the training of AReLiT. Notably, recent studies that combine auxiliary tasks with reinforcement learning, such as those conducted by Rafiee et al. [2022] and Rafiee et al. [2023], are interesting options.
- Additional theoretical analysis of the approximation approach used in AReLiT. Chapter 4 provides a simple way to perform incremental updates to low rank-decompositions of matrices. It would be interesting to formally discuss this approximation approach, bound the error of the approximation, and understand its differences from other approximation techniques such as incremental SVD (Brand [2006]).
- Exploring the performance of AReLiT in other partially observable environments.
- Evaluating AReLiT in language modeling benchmarks such as the Long Range Arena (Tay et al. [2021]).
- Investigating the feasibility of deriving efficient real-time recurrent learning (Williams and Zipser [1989]) gradient updates for AReLiT. This would allow us to train AReLiT in an online fashion.

References

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv:1607.06450*, 2016.

Bram Bakker. Reinforcement learning with long short-term memory. In *Advances in neural information processing systems*, 2001.

Charles Beattie, Joel Z Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, et al. Deepmind lab. *arXiv:1612.03801*, 2016.

Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Matthew Brand. Incremental singular value decomposition of uncertain data with missing values. In *European Conference on Computer Vision*, 2002.

Matthew Brand. Fast low-rank modifications of the thin singular value decomposition. *Linear Algebra and its Applications*, 415(1):20–30, 2006. ISSN 0024-3795. Special Issue on Large Scale Linear and Nonlinear Eigenvalue Problems.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. In *Advances in neural information processing systems*, 2020.

Aydar Bulatov, Yury Kuratov, and Mikhail Burtsev. Recurrent memory transformer. In *Advances in Neural Information Processing Systems*, 2022.

Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Misha Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling. In *Advances in neural information processing systems*, 2021.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv:1406.1078*, 2014.

Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, et al. Rethinking attention with performers. *arXiv:2009.14794*, 2020.

Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). In *International Conference on Learning Representations*, 2016.

Tim Cooijmans and James Martens. On the variance of unbiased online recurrent optimization. *arXiv:1902.02405*, 2019.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv:1901.02860*, 2019.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv:1810.04805*, 2018.

Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *Proceedings of Machine Learning Research*, 2018.

Yuan Gao and Dorota Glowacka. Deep gate recurrent neural network. In *Asian conference on machine learning*, pages 350–365. PMLR, 2016.

Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. In *The Association for the Advancement of Artificial Intelligence Symposium Series*, 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Computer Vision and Pattern Recognition*, pages 770–778, 2016.

Nicolas Heess, Jonathan J Hunt, Timothy P Lillicrap, and David Silver. Memory-based control with recurrent neural networks. *arXiv:1512.04455*, 2015.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *Proceedings of Machine Learning Research*, 2020.

Urvashi Khandelwal, He He, Peng Qi, and Dan Jurafsky. Sharp nearby, fuzzy far away: How neural language models use context. In *Association for Computational Linguistics (Volume 1: Long Papers)*, 2018.

Michael Laskin, Luyu Wang, Junhyuk Oh, Emilio Parisotto, Stephen Spencer, Richie Steigerwald, DJ Strouse, Steven Hansen, Angelos Filos, Ethan Brooks, et al. In-context reinforcement learning with algorithm distillation. *arXiv:2210.14215*, 2022.

Marlos C. Machado, Marc G. Bellemare, Erik Talvitie, Joel Veness, Matthew Hausknecht, and Michael Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *J. Artif. Int. Res.*, 61(1):523–562, jan 2018. ISSN 1076-9757.

Paul Michel, Omer Levy, and Graham Neubig. In *Neural Information Processing Systems*, 2019.

Nikhil Mishra, Mostafa Rohaninejad, Xi Chen, and Pieter Abbeel. A simple neural attentive meta-learner. In *International Conference on Learning Representations*, 2018.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of Machine Learning Research*, 2016.

Steven Morad, Ryan Kortvelesy, Matteo Bettini, Stephan Liwicki, and Amanda Prorok. POP-Gym: Benchmarking partially observable reinforcement learning. In *International Conference on Learning Representations*, 2023.

Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, et al. Massively parallel methods for deep reinforcement learning. *arXiv:1507.04296*, 2015.

Abhishek Nandy, Manisha Biswas, Abhishek Nandy, and Manisha Biswas. Unity ml-agents. *Neural Networks in Unity: C# Programming for Windows 10*, pages 27–67, 2018.

A. Onat, H. Kita, and Y. Nishikawa. Recurrent neural networks for reinforcement learning: architecture, learning algorithms and internal representation. In *1998 IEEE International Joint Conference on Neural Networks Proceedings*, 1998.

Ian Osband, Yotam Doron, Matteo Hessel, John Aslanides, Eren Sezener, Andre Saraiva, Katrina McKinney, Tor Lattimore, Csaba Szepesvari, Satinder Singh, et al. Behaviour suite for reinforcement learning. *arXiv:1908.03568*, 2019.

Yangchen Pan, Adam White, and Martha White. Accelerated gradient temporal difference learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

Emilio Parisotto, Francis Song, Jack Rae, Razvan Pascanu, Caglar Gulcehre, Siddhant Jayakumar, Max Jaderberg, Raphael Lopez Kaufman, Aidan Clark, Seb Noury, et al. Stabilizing transformers for reinforcement learning. In *Proceedings of Machine Learning Research*, 2020.

Jurgis Pašukonis, Timothy P Lillicrap, and Danijar Hafner. Evaluating long-term memory in 3d mazes. In *International Conference on Learning Representations*, 2023.

Hao Peng, Nikolaos Pappas, Dani Yogatama, Roy Schwartz, Noah A Smith, and Lingpeng Kong. Random feature attention. *arXiv:2103.02143*, 2021.

Olivier Petit, Nicolas Thome, Clement Rambour, Loic Themyr, Toby Collins, and Luc Soler. U-net transformer: Self and cross attention for medical image segmentation. In *Machine Learning in Medical Imaging*, pages 267–276. Springer, 2021.

Aleksei Petrenko, Zhehui Huang, Tushar Kumar, Gaurav Sukhatme, and Vladlen Koltun. Sample factory: Egocentric 3D control from pixels at 100000 FPS with asynchronous reinforcement learning. In *Proceedings of Machine Learning Research*, 2020.

Banafsheh Rafiee, Jun Jin, Jun Luo, and Adam White. What makes useful auxiliary tasks in reinforcement learning: investigating the effect of the target policy, 2022.

Banafsheh Rafiee, Sina Ghiassian, Jun Jin, Richard S. Sutton, Jun Luo, and Adam White. Auxiliary task discovery through generate and test, 2023.

Andrew M Saxe, James L McClelland, and Surya Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv:1312.6120*, 2013.

Imanol Schlag, Kazuki Irie, and Jürgen Schmidhuber. Linear transformers are secretly fast weight programmers. In *Proceedings of Machine Learning Research*, 2021.

John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, 2015.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In *International Conference on Learning Representations*, 2016.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv:1707.06347*, 2017.

Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 1999.

Corentin Tallec and Yann Ollivier. Unbiased online recurrent optimization. *arXiv:1702.05043*, 2017.

Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. Long range arena : A benchmark for efficient transformers. In *International Conference on Learning Representations*, 2021.

Yao-Hung Hubert Tsai, Shaojie Bai, Makoto Yamada, Louis-Philippe Morency, and Ruslan Salakhutdinov. Transformer dissection: A unified understanding of transformer's attention via the lens of kernel. *arXiv:1908.11775*, 2019.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, 2017.

Eric W. Weisstein. Fourier series. From MathWorld–A Wolfram Web Resource.

P.J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

Ronald J Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989.

Yuhuai Wu, Elman Mansimov, Roger B Grosse, Shun Liao, and Jimmy Ba. Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. In *Advances in neural information processing systems*, 2017.

Huasong Zhong, Jingyuan Chen, Chen Shen, Hanwang Zhang, Jianqiang Huang, and Xian-Sheng Hua. Self-adaptive neural module transformer for visual question answering. *IEEE Transactions on Multimedia*, 23:1264–1273, 2020.

Appendix A

Additional Experiment Details

A.1 T-Maze Experiments

We include the details of the T-Maze experiments presented in Section 5.1. We trained all agents using Advantage Actor Critic (A2C) algorithm (Wu et al. [2017]). We trained 5 architectures: LSTM, GRU, GTrXL, ReLIT and AReLiT. We use the hyperparameters as described in Table A.1. We include the architecture configuration for each of the 5 architectures in Table A.2. For each architecture, we sweep the learning rate and the entropy coefficient. Our hyperparameter tuning strategy is as follows: We train 5 seeds per architecture for each corridor length in 120-200 and hyperparameter configuration for 5M steps. We identify the best hyperparameter configuration according to the best mean success rate in the last 100K steps across all corridor lengths.

Table A.1: Hyperparameters and sweeps for the T-Maze experiments.

Hyperparameter	**Value**
Learning Rate	[0.001, 0.0001 0.0005, 0.00001, 0.00005]
Discount Factor (γ)	0.99
Advantage Estimation Coefficient (λ)	0.95
Entropy Coefficient	[0.1, 0.01, 0.001, 0.0001, 0.00001]
Value Loss Coefficient	0.5
Rollout Len	256
Num of Envs	8
Batch Size (Rollout Len $\times$ Num of Envs)	2048
Actor Layer Dimension	128
Critic Layer Dimension	128

Table A.2: Architecture configuration for LSTM, GRU, GTrXL, ReLIT and AReLiT for T-Maze experiments.

Hyperparameter	**LSTM**	**GRU**	**GTrXL**	**ReLiT**	**AReLiT**
Embedding Dimension (d)	600	680	128	128	128
Hidden Dimension (d_{hid})	1200	1360	N/A	N/A	N/A
Num Heads	N/A	N/A	4	4	4
Head Dim	N/A	N/A	64	64	64
Num Layers (L)	1	1	4	4	4
Memory Size (M)	N/A	N/A	128 & 256	N/A	N/A
Projection Hyperparameter (η)	N/A	N/A	N/A	4	4
Approximation Hyperparameter (r)	N/A	N/A	N/A	N/A	1
Actor Layer Dimension	128				
Critic Layer Dimension	128				

A.2 Memory Maze Experiments

We include the details of the Memory Maze experiments presented in Section 5.2. All of the experiments in that chapter were implemented using asynchronous PPO implementation from Sample Factory library (Petrenko et al. [2020]). We started with the default hyperparameters for the DMLab lab experiments in Schulman et al. [2015], and finetuned the learning rate and entropy coefficient. For each of LSTM, GTrXL and AReLiT, to tune the learning rate and entropy coefficient, we run a sweep for three seeds for 15M steps in the Memory Maze 11×11 environment. We average the results for the last 1M steps across the three seeds and select the best hyperparameter according to total episodic reward. Using the best-identified hyperparameter, we generate the final results for 100M steps for each of the three seeds. We detail the hyperparameters along with the sweeps for the learning rate and entropy coefficient in Table A.3. We include the architecture configuration for each of the 3 architectures in Table A.4.

Table A.3: Hyperparameters and sweeps for Memory Maze experiments.

Hyperparameter	**Value**
Learning Rate	[0.0025, 0.00025, 0.000025]
Discount Factor (γ)	0.99
Advantage Estimation Coefficient (λ)	0.95
Entropy Coefficient	[0.03, 0.003, 0.0003]
Number of Epochs	1
Rollout Length	200
Sequence Length	100
Batch Size	3200
PPO Clip Ratio	0.1
PPO Clip Value	1
Max Gradient Norm	4
Value Function Coefficient	0.5
Number of Workers	32
Number of Envs per Worker	2

Table A.4: Architecture configuration for LSTM, GTrXL and AReLiT for Memory Maze experiments.

Hyperparameter	**LSTM**	**GTrXL**	**AReLiT**
Embedding Dimension (d)	768	256	256
Hidden Dimension (d_{hid})	1536	N/A	N/A
Num Heads	N/A	8	8
Head Dim	N/A	64	64
Num Layers (L)	1	4	4
Memory Size (M)	N/A	256	N/A
Projection Hyperparameter (η)	N/A	N/A	4
Approximation Hyperparameter (r)	N/A	N/A	7