Universiteit Leiden

# Master Computer Science

On the Application of State Space Models to Partially Observable Markov Decision Processes


| | |
|---|---|
| Name: | T.J.W. van Gelooven |
| Student ID: | s1853686 |
| Date: | 25/07/2025 |
| Specialisation: | Artificial Intelligence |
| 1st supervisor: | Dr. Thomas Moerland |
| 2nd supervisor: | Ir. A. Serra-Gómez |


Master's Thesis in Computer Science


Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

# Declaration of Authorship

I, T.J.W. van GELOOVEN, declare that this thesis titled, "On the Application of State Space Models to Partially Observable Markov Decision Processes" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: 

Date: 25/07/2025

*"We live in capitalism, its power seems inescapable but then, so did the divine right of kings. Any human power can be resisted and changed by human beings. Resistance and change often begin in art"*

Ursula Le Guin

LEIDEN UNIVERSITY

# *Abstract*

Faculty of Science
LIACS

Master of Science

**On the Application of State Space Models to Partially Observable Markov Decision Processes**

by T.J.W. van GELOOVEN

Modeling long-range temporal dependencies efficiently is a central challenge in reinforcement learning (RL), and is of particular importance in partially observable environments. Structured State Space Models (SSMs), grounded in control theory, have recently demonstrated state-of-the-art performance in sequence modeling tasks by combining low inference latency with parallelizable training. In this work, we implement several modern SSM variants, including LSSL and S5, and a SSM-inspired streamlined GRU adaptation (minGRU). We evaluate the performance of SSMs in partially observable on-policy RL tasks using Jax implementations of environments from the POPGym benchmark suite and a custom long-memory environment, *the Memory Corridor*. Our results show that S5 models match or exceed the performance of GRU-based baselines while offering improved computational efficiency and memory handling capabilities. These findings suggest that SSMs are a promising direction for sequence modeling in RL. We provide code at `https://github.com/Tom-v-G/SSMs-for-PO-RL`

# *Acknowledgements*

I would like to express my sincere gratitude to the many people who have contributed to the completion of this project, whether through guidance or through encouragement. First of all my thanks to Koen and Felix, who were instrumental in shaping the project and without whose support and patience for my (many) questions neither a line of code nor a paragraph of text would have been written. Also my thanks to Thomas, for providing insightful observations and giving constructive feedback on my work, and to Jacopo, whose presence has always brightened my day. Many thanks to the entire RL group for creating a welcoming and intellectually stimulating environment. The paper discussions were always a pleasure to attend. To my roommates, thank you for provding welcome distractions when they were needed, and for creating a home I enjoy returning to. To my parents and brother, thank you for your unconditional support which you have extended not only these past few months but for as long as I can remember. I would also like to extend my gratitude to myself, for only a few years ago I would not have had the mental fortitude to see a project of this size through to the end. And finally, thank you, for taking the time to read my work.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| **SSM** | **L**inear **S**tate **S**pace **M**odel |
| **LTI** | **L**inearly **T**ime **I**nvariant |
| **LSSL** | **L**inear **S**tate **S**pace **L**ayer |
| **LSSLf** | **F**ixed Linear **S**tate **S**pace **L**ayer |
| **HiPPO** | **H**igher **O**rder **P**olynomial **P**rojection **O**perators |
| **NPLR** | **N**ormal **P**lus **L**ow **R**ank |
| **DPLR** | **D**iagonal **P**lus **L**ow **R**ank |
| **PPO** | **P**roximal **P**olicy **O**ptimization |
| **MVM** | **M**atrix **V**ector **M**ultiplication |
| **MDP** | **M**arkov **D**ecision **P**rocess |
| **POMDP** | **P**artially **O**bservable **M**arkov **D**ecision **P**rocess |
| **FF** | **F**eed **F**orward |
| **GRU** | **G**ated **R**ecurrent **U**nit |

# List of Symbols

| | |
|---|---|
| $\mathbf{A}$ | state matrix |
| $\mathbf{B}$ | input matrix |
| $\mathbf{C}$ | emission matrix |
| $\mathbf{D}$ | feedthrough matrix |
| $\overline{\mathbf{A}}, \overline{\mathbf{B}}$ | discretized matrices |
| $\mathbf{\Lambda}$ | eigendecomposition of normal state matrix |
| $\mathbf{V}$ | eigenvalues of normal state matrix |
| $\mathbf{u}$ | input sequence |
| $\mathbf{x}$ | hidden/latent state |
| $L$ | sequence length |
| $N$ | hidden state dimension |
| $H$ | feature dimension |
| $\mathcal{S}$ | MDP State Space |
| $\mathcal{Y}$ | POMDP Observation Space |
| $\mathcal{A}$ | MDP Action Space |
| $T$ | MDP Transition Function |
| $O$ | POMDP Emission function |
| $r$ | MDP Reward Function |
| $\gamma$ | MDP Discount Factor |
| $\pi$ | RL Agent Policy |
| $\theta$ | RL Agent Parameters |
| $\epsilon$ | PPO Clipping Constant |

# Chapter 1

# Introduction

One of the large problems in machine learning at the present time is modeling long sequential datastreams in an efficient manner. Traditionally RNN's (Rumelhart, Hinton, Williams, et al., 1985), along with extensions such as LSTMs (Hochreiter and Schmidhuber, 1997) and GRU's (Cho et al., 2014), were employed. More recently, Transformers have revolutionized the field with their attention-based mechanisms, achieving state-of-the-art results on a wide range of benchmarks (Vaswani et al., 2017). Transformers do however require significant computational power and are limited in scalability to long sequence lengths. As a result, alternative architectures continue to be explored. One promising direction is found in the structured state space models (SSMs). Recent works have shown that these foundational scientific models, inspired by control theory and signal processing theory, can outperform RNN- and Transformers-based models on even the most challenging sequence modeling benchmarks, such as the Long Range Arena (Tay et al., 2021).

SSMs provide a mathematical framework, both in continuous- and discrete-time, for modeling the evolution of a system over time by propagating system information through latent states in a linearized manner. Notably, RNNs can be seen as a special case of an SSM (Gu and Dao, 2024). While standard Transformer architecture's per step runtime scales quadratically with sequence length, SSMs maintain constant memory and runtime per step, making them a more suitable option for long-sequence tasks. Although RNNs and LSTMs have this same property, they lack the parallelizable training capabilities enabled by the linear time-invariant (LTI) structure of SSMs, instead relying on (sequential) backpropagation through time (BPTT) for training. SSM based architectures might thus be poised to overtake RNNs, LSTMs and Transformers as the de facto choice for long-range sequence learning.

The constant time complexity of SSMs during inference might be of benefit in a reinforcement learning (RL) setting, where inference is repeatedly used to collect rollout trajectories from the environment. Transformers generally have poor runtime performance in RL tasks (Parisotto and Salakhutdinov, 2021). In contrast, RNNs continue to be widely used in partially observable (PO) RL tasks, where memory capacity and the ability to handle episodic boundaries via hidden state resets are important. Recent work has proposed a modification to SSMs that allows for a similar hidden state reset (Lu et al., 2023), making them suitable for RL as well.

In this work we implement and evaluate these next generation models in partially observable RL environments using the JAX implementation of the Partially Observable Process Gym (POPGym) (Lu et al., 2023), (Morad et al., 2023). Specifically, we implement several variants of the first-generation Linear State Space Layer

(LSSL) (Gu et al., 2021) and the later S5 architecture (Smith, Warrington, and Linderman, 2023). LSSL variants include the LSSLf, which updates latent states using fixed matrices optimized for online function approximation (Gu et al., 2020), regular LSSL, which allows the update matrices to be trained with gradient descent, and LSSLf-diag, a novel architecture which uses diagonalized update matrices in combination with parallel scan to efficiently compute latent states in a way similar to the S5 model. We also implement minGRU, a SSM-inspired streamlined version of the traditional gated recurrent unit (GRU) architecture that is fully parallelizable during training (Feng et al., 2024). We compare the performance of SSM-based architectures against baseline implementations to see if SSMs can replace older architectures as the de facto choice for reinforcement learning tasks. Additionally, we also implement the MemoryCorridor environment (Moerland et al., 2024) in Jax to test the memory efficiency of SSMs in comparison to baseline models such as the GRU.

Concretely, our research questions are the following.

**RQ 1** How sensitive are SSMs to hyperparameter tuning in the RL setting?

**RQ 2** How does SSM performance compare to baseline models on PO RL tasks?

**RQ 3** What is the influence of the latent space dimension on the effective context window size of the model?

We find that while LSSL-based architectures are not suitable for RL tasks due to their memory complexity, S5-based models outperform baseline models in the partially observable RL setting in both performance (FF, GRU, minGRU) and speed (GRU). When evaluating model performance, the sensitivity of the S5 architecture to hyperparameters can vary depending on the environment. While individual tuning per environment remains important, we have identified a hyperparameter configuration that allows for reasonably consistent performance on most environments. In the case of long-memory tasks we have found that performance is only limited by model latent space dimensionality in the high-update regime.

The remainder of this work is structured as follows. Chapter 2 provides an extensive background on the development of structured state space models, starting from the linear state space layer (LSSL) up to the more recent Selective SSM architecture (Mamba). This chapter also covers the adaptation of SSMs to reinforcement learning tasks. Readers who are already familiar with reinforcement learning and those seeking only the essential background may wish to focus on section 2.1.1, section 2.1.2, section 2.1.4, section 2.1.6 and section 2.2.3. Chapter 3 discusses related work on SSMs and on solving long-sequence RL problems. Next, chapter 4 presents our experimental setup and the deep neural architecture used, followed by an analysis of the experimental results in chapter 5. Finally, chapter 6 summarizes our findings, discusses the limitations of our study and suggests directions for future work.

Our contributions include implementations of LSSL, LSSLf (Gu et al., 2021), LSSLf-diag (own), S5 (Smith, Warrington, and Linderman, 2023), and minGRU (Feng et al., 2024) models using the Equinox framework (Kidger and Garcia, 2021), together with an implementation of the the MemoryCorridor environment (Moerland et al., 2024) in the Gymnax framework (Lange, 2022) [1].

[1] Code is available under the MIT license at https://github.com/Tom-v-G/SSMs-for-PO-RL

# Chapter 2

# Background

In this chapter we will explain how deep learning state space models (SSMs) can be derived from their linear control theory counterparts. We will show how the application of HiPPO matrices makes the use of SSMs a viable alternative to transformer models for sequence learning. We summarize the history of the use of SSMs in deep learning, starting from the linear state space layer, and discuss the different insights and philosophies which have led to later models such as S5 and Mamba. We will also introduce PPO, expand upon the (minor) extension of PPO to partially observable Markov Decision Processes and explain the modification one can make to adapt SSMs to the RL setting.

## 2.1 State Space Models

As mentioned above, the deep learning SSMs are derived from linear control theory. Control theory is a mathematical field studying the problem of feedback control design. Given an internally stable closed physical system, control theory tries to design methods to make the influence of external disturbance inputs as small as possible (Trentelman et al., 2002). As we will show, the linear systems used to describe problems in the control theory setting happen to hold a deep connection with recurrent neural network architectures, allowing us to leverage much of the mathematical tools developed by this field to the deep learning setting.

Control theory model systems as continuous ordinary differential equation. Similarly, we define a linear, time-invariant, finite-dimensional state-space system (LTI SSM) in the following way. Let $u \in \mathbb{R}^M$ be an input variable, $x \in \mathbb{R}^N$ be the (hidden) state variable and $y \in \mathbb{R}^P$ be the output variable. The equations of the corresponding control system are given by

$$\dot{x}(t) = \mathbf{A}x(t) + \mathbf{B}u(t) \tag{2.1}$$
$$y(t) = \mathbf{C}x(t) + \mathbf{D}u(t) \tag{2.2}$$

where $\mathbf{A} : \mathbb{R}^N \to \mathbb{R}^N$, $\mathbf{B} : \mathbb{R}^M \to \mathbb{R}^N$, $\mathbf{C} : \mathbb{R}^N \to \mathbb{R}^P$ and $\mathbf{D} : \mathbb{R}^M \to \mathbb{R}^P$ are linear maps (Trentelman et al., 2002). Note that this system is called time-invariant because the maps $\mathbf{A}, \mathbf{B}, \mathbf{C}$ and $\mathbf{D}$ do not depend on $t$.

We can view this system of equations as mapping an input function $u$ through a hidden state $x$ to an output function $y$, where equation 2.1 defines the differential equation that governs the behaviour of the hidden state variable $x$ and equation 2.2 describes the relation between the output, the state variable and the input variable.

To solve equation 2.1 and 2.2 numerically it is necessary to discretize the input, hidden state and the linear maps. This being the case, any implementation of LTI SSMs is not limited to continuous inputs, but can instead be used for any generic sequence. Discretizing the LTI control system thus allows us to use it as a black-box representation in a deep sequence model, where the linear maps are the model parameters learned via gradient descent (Gu et al., 2021).

### 2.1.1 Discretizaton Methods

Discretized we can understand $\bar{u} = (u_{t_n})_{n\in\{1,\dots,L\}}$ as a sequence of length $L$ together with a time sequence $\tau = (t_n)_{n\in\{1,\dots,L\}}$ such that $u_{t_n} = u(t_n)$. One can think of $\tau$ as points on which we sample the continuous function $u$. For the rest of this chapter we will assume that $u$ is sampled uniformly with a sampling rate of $\Delta t$ such that $u_{t_n} = u(t_0 + n \cdot \Delta t)$ given a starting sample time $t_0$. To solve our system of equations in the discretized case, we need to find valid approximations of each $x(t_n)$. Note that for any differential equation $\dot{x}(t) = f(t, x(t))$, we have $\dot{x}(t_{i+1}) = x(t_i) + \int_{t_i}^{t_{i+1}} f(s, x(s))ds$ by the fundamental theorem of calculus. The system can thus be discretized by choosing an appropriate approximation of the integral on the right hand side of this equation. When using a linear approximation scheme the discretized control system can be written as

$$x_t = \overline{\mathbf{A}} x_{t-1} + \overline{\mathbf{B}} u_t \tag{2.3}$$
$$y_t = \mathbf{C} x_t + \mathbf{D} u_t \tag{2.4}$$

where $\overline{\mathbf{A}}$ and $\overline{\mathbf{B}}$ are the (now) discretized state- and input matrices approximating the linear maps $\mathbf{A}$ and $\mathbf{B}$ over an interval $\Delta t$.

There are many different ways to discretize linear maps. Earlier SSM models used General Bilinear Transform (see appendix A). Later models use the Zero-order Hold (ZOH) method. ZOH discretization, also called "Matched Z-transform method", preserves system stability but keeps unstable poles and does not preserve the time- or frequency response of the system it discretizates. Its implementation is however fast, and it is unclear if the drawbacks of the ZOH method matter in the deep learning setting. Equations for the ZOH method are given by

$$\overline{\mathbf{A}} = \exp(\Delta t \cdot \mathbf{A}) \quad \overline{\mathbf{B}} = \mathbf{A}^{-1}(\overline{\mathbf{A}} - \mathbf{I}) \cdot \mathbf{B}. \tag{2.5}$$

where the exponent is understood to be taken element-wise. Note that, for any discretization scheme, $\overline{\mathbf{A}}$ and $\overline{\mathbf{B}}$ are the same for any $t$ as long as we assume a uniform sampling rate. Note also the $\mathbf{C}$ and $\mathbf{D}$ do not need to be discretized, as they are not part of a differential equation. Using equations 2.3 and 2.4 we have transformed our continuous time system to a recurrent set of equations. We will see that this recurrence allows us to train SSM models in a time-efficient manner thus allowing us to use them for deep learning applications.

### 2.1.2 The Linear State Space Layer

The Linear State-Space Layer (LSSL) introduced by Gu et al., 2021 was the first application of the state-space paradigm to deep learning models. They noted that the linearized structure of the discretized SSM allowed for a recurrent network architecture that could be trained efficiently without the use of backprogation through time. The LSSL functions in the intersection of three paradigms, from each of which it can benefit.

- The LSSL is based on an implicitly **continuous-time** based model family. By discretizing with different timescales $\Delta t$ the LSSL can work on differently / irregularly sampled datapoints. This is of particular note in the audio processing domain, where datasources with different sampling intervals are common.
- The LSSL is **recurrent**. Model inference can be applied efficiently and with an unbounded context window.
- The LSSL can be trained **convolutionally**. Models can be trained by convolving the LSSL-kernel with sequence data. This allows for depthwise parallel training based on local signals.

Whereas the first two of these statements can be inferred directly from the previous section, the last statement requires further justification. Given a sequence $\mathbf{u}$ and initializing our hidden state to $x_{-1} = \overline{\mathbf{0}}$, equations 2.3 and 2.4 become

$$\begin{aligned} x_0 &= \overline{\mathbf{A}}x_{-1} + \overline{\mathbf{B}}u_0 = \overline{\mathbf{B}}u_0 & y_0 &= \mathbf{C}x_0 + \mathbf{D}u_0 = \mathbf{C}\overline{\mathbf{B}}u_0 + \mathbf{D}u_0 \\ x_1 &= \overline{\mathbf{A}}x_0 + \overline{\mathbf{B}}u_1 = \overline{\mathbf{A}\mathbf{B}}u_0 + \overline{\mathbf{B}}u_1 & y_1 &= \mathbf{C}x_1 + \mathbf{D}u_1 = \mathbf{C}\overline{\mathbf{A}\mathbf{B}}u_0 + \overline{\mathbf{B}}u_1 + \mathbf{D}u_1 \\ &\vdots & &\vdots \end{aligned}$$

In general the $k$-th LSSL output is given by

$$x_k = \overline{\mathbf{A}}^k\overline{\mathbf{B}}u_0 + \overline{\mathbf{A}}^{k-1}\overline{\mathbf{B}}u_1 + \cdots + \overline{\mathbf{A}\mathbf{B}}u_{k-1} + \overline{\mathbf{B}}u_k \tag{2.6}$$

$$y_k = \mathbf{C}(\overline{\mathbf{A}})^k\overline{\mathbf{B}}u_0 + \mathbf{C}(\overline{\mathbf{A}})^{k-1}\overline{\mathbf{B}}u_1 + \cdots + \mathbf{C}(\overline{\mathbf{A}\mathbf{B}})u_{k-1} + \mathbf{C}\overline{\mathbf{B}}u_k + \mathbf{D}u_k. \tag{2.7}$$

We can thus view the LSSL output as a (non-circular) convolution

$$\mathbf{y} = \mathcal{K}_L(\overline{\mathbf{A}}, \overline{\mathbf{B}}, \mathbf{C}) * \mathbf{u} + \mathbf{D}\mathbf{u} \tag{2.8}$$

where $\mathcal{K}_L(\overline{\mathbf{A}}, \overline{\mathbf{B}}, \mathbf{C}) = (\mathbf{C}\overline{\mathbf{A}}^i\overline{\mathbf{B}})_{i\in[L]} \in \mathbb{R}^L$ is called the *Krylov function*.

#### Application to Deep Learning

To use LSSLs in a deep learning model one instantiates many SSM kernels at once. Each kernel is discretized with a different sampling interval $\Delta t$ to detect different features in the input data $u$. The amount of kernels, called the feature dimension $H$, is a hyperparameter set by the user and depends on how many patterns one assumes to detect in the signal $u$. The sequence $u$ is then simply broadcast to each of the kernels. Since the main operation in each kernel consists of a convolution and a linear mapping, each kernel can be run in parallel. Non-linear activation functions are applied after the linear kernel operations. In a deep learning setting the output

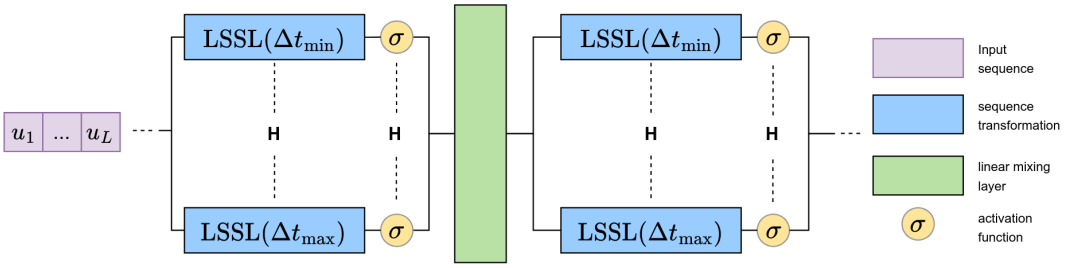



FIGURE 2.1: SISO SSM design using two LSSLs and a linear mixing layer. The input $\mathbf{u}$ is broadcast to the $H$ parallel SSM kernels each instantiated with a different sampling interval $\Delta t$. Activation functions are used to introduce non-linearity.

from the kernels can be mixed for feature interpolation. The paradigm of instantiating many parallel SSM kernels is colloquially refered to as single-input single-output (SISO) method. Multiple LSSLs to be stacked and mixed with other layers to form a deep architecture. See figure 2.1 for reference.

In theory, LSSLs can be trained efficiently, with quasi-linear time- and space complexity $\sim O(N+L)$ (Gu et al., 2021). In practice (i.e. using floating point arithmatic), LSSL algorithms can be numerically unstable and require large amounts of memory. There are two core issues. The first is the matrix-vector multiplication (MVM) with the discretized state matrix. Matrix discretization requires calculating the inverse of said matrix, which can be costly for non-specialized matrix forms. The second is that computing the Krylov function requires taking $L$-th power of the matrix $\overline{\mathbf{A}}$, which can result in unstable values and large computations.

To mitigate the issues mentioned above, Gu et al., 2021 also proposed the fixed Linear State Space Layer (LSSLf). This model fixes the state matrix $\mathbf{A}$, input matrix $B$ and sampling time $\Delta t$ such that the Krylov functions can be precomputed save for the factor $\mathbf{C}$. This results in significantly faster training and inference speed. Given proper matrix initialisations, the LSSLf model performs similarly, albeit slightly worse, to the full LSSL model on pixel-by-pixel classification problems and vital-sign prediction benchmarks.

### Matrix initialisation

In practice, initialising an LSSL with a random state matrix $\overline{\mathbf{A}}$ does not give rise to an effective sequence model (Gu et al., 2021). The gradient landscape is too complex too learn a suitable memory transition function within feasible timeframes. Gu et al., 2020 set out to solve this problem by phrasing memory retention as a the problem of *online function approximation*. Given a function $f(t) : \mathbb{R}_+ \to \mathbb{R}$, one can generate a (finite) summary of $f$ by projecting $f$ onto a (finite) family of orthogonal polynomials. The polynomial coefficients that most resemble the original function with respect to a given measure are deemed optimal. The resulting high-order polynomial projection operators (HiPPO) framework produces operators that can project arbitrary function to the orthogonal polynomial basis in a recurrent fashion, thus allowing incremental updates of the optimal polynomial coefficients on each timestep.

Gu et al., 2021 found that initialising $\mathbf{A}$ and $\mathbf{B}$ to projection matrices from the HiPPO framework drastically improves performance. Depending on the measure one chooses to optimize for, the values of the projection matrices change. A measure can, for example, assign uniform weight to the a sliding window of the function history or assign exponentially decaying weight to function history. The measure we will focus on is the **scaled Legendre measure** (Leg-S) designed by Gu et al., 2020. This measure assigns uniform weight to the entire function history. The Leg-S projection matrices are given by

$$\mathbf{A}_{nk} = \begin{cases} (2n+1)^{1/2}(2k+1)^{1/2} & \text{if } n > k \\ n+1 & \text{if } n = k \\ 0 & \text{if } n < k \end{cases} \qquad \mathbf{B}_n = (2n+1)^{1/2}. \tag{2.9}$$

### 2.1.3 Diagonalizing the state matrix

While LSSLf model fixes the numeric instability issues plaguing the LSSL model, the fact that the hidden state update equation 2.3 is no longer learnable is too large a price to pay. The main bottleneck for any implementation of equation 2.3 are the $L$ succesive multiplications with $\overline{\mathbf{A}}$. Simplifying the structure of $\overline{\mathbf{A}}$ can thus lead to more efficient algorithms. The most obvious route is to diagonalize the state matrix directly. As shown by Gu, Goel, and Ré, 2022, conjugation with a change of basis matrix $\mathbf{V}$ is an equivalence relation on SSMs $(\mathbf{A}, \mathbf{B}, \mathbf{C}) \sim (\mathbf{V}^{-1}\mathbf{A}\mathbf{V}, \mathbf{V}^{-1}\mathbf{B}, \mathbf{C}\mathbf{V})$. Diagonalizing $\mathbf{A}$ would thus not incur any penalty in expressive power. However, to diagonalize $\mathbf{A}$ one needs a change of basis matrix $\mathbf{V}$ with entries up to magnitude $2^{4N/3}$ (Gu, Goel, and Ré, 2022). Direct diagonalization would thus lead to numerical instability for any decently sized latent space dimension.

To circumvent this issue we can first construct the Normal plus Low Rank (NPLR) form of $\mathbf{A}$, which can be done for any HiPPO matrix according to theorem 1 of Gu, Goel, and Ré, 2022. This form is given by

$$\mathbf{A}_{\text{HiPPO}} = \mathbf{A}^{\text{Normal}}_{\text{HiPPO}} - \mathbf{P}\mathbf{Q}^T = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^* - \mathbf{P}\mathbf{Q}^T = \mathbf{V}(\mathbf{\Lambda} - (\mathbf{V}^*\mathbf{P})(\mathbf{V}^*\mathbf{Q})^*)\mathbf{V}^* \tag{2.10}$$

where $\mathbf{V} \in \mathbb{C}^{N\times N}$ is a unitary matrix, $\mathbf{\Lambda} \in \mathbb{C}^{N\times N}$ is a diagonal matrix and $\mathbf{P}, \mathbf{Q} \in \mathbb{R}^{N\times r}$ are low-rank factorization matrices ($r = 1$ or $r = 2$ for all HiPPO matrices). The right hand side of this equation shows that the normal matrix can be further decomposed to generate a Diagonal plus Low Rank (DPLR) representation of the HiPPO matrix. This result can be extended to show that the HiPPO-LegS matrix can be written as (Goel et al., 2022)

$$\mathbf{A}_{\text{LegS}} = \mathbf{A}^{Normal}_{\text{LegS}} - \mathbf{P}_{\text{LegS}}\mathbf{P}^T_{\text{LegS}} \tag{2.11}$$

where

$$\mathbf{A}^{\text{Normal}}_{\text{LegS}} = -\begin{cases} (n+\frac{1}{2})^{\frac{1}{2}}(k+\frac{1}{2})^{\frac{1}{2}} & n > k \\ \frac{1}{2} & n = k \\ (n+\frac{1}{2})^{\frac{1}{2}}(k+\frac{1}{2})^{\frac{1}{2}} & n < k \end{cases} \tag{2.12}$$

$$\mathbf{P}_{\text{LegS}} = (n+\frac{1}{2})^{\frac{1}{2}}. \tag{2.13}$$

Note that the eigendecomposition of $\mathbf{A}_{\text{LegS}}^{Normal}$ results in a complex diagonal state matrix $\mathbf{\Lambda}$ and change of basis matrix $\mathbf{V}$ with values that do not scale exponentially w.r.t. the latent space dimension. Gupta, Gu, and Berant, 2022 have shown that SSMs using only this diagonal component of the state matrix (disregarding the low rank corrections) perform comparatively to SSMs instantiated with the full DPLR state matrix form.

It should also be noted that the eigenvalues of the of the eigendecompositon for a diagonalizable real matrix always occur in conjugate pairs. During training this **conjugate symmetry** in $\mathbf{\Lambda}$ may be lost. Given that $\mathbf{\Lambda}$ is an approximation of the (real-entried) LegS projection matrix, we can opt to enforce conjugate symmetry on $\mathbf{\Lambda}$ during the training process effectively halving the number of used eigenvalues and the latent space dimension.

This diagonal form can be incorporated into the LSSL and has been incorporated into the later S4 model (Gupta, Gu, and Berant, 2022) with succes. The reduced memory complexity of the diagonalized state matrices also unlocked a previously unfeasible path to increasing training speeds, reducing sequence transformation time complexity of the next generation of SSMs from $O(NL)$ to $O(N \log L)$ by replacing convolutional computations with parallel scans.

### 2.1.4 Parallel Scan

Let $\circ$ be a binary associative operator i.e. $(a \circ b) \circ c = a \circ (b \circ c)$ with time complexity $T$. Given a sequence $[a_1, a_2 \ldots, a_L]$, the parallel scan algorithm computes the prefix sum

$$[a_1, a_1 \circ a_2, \ldots, (a_1 \circ a_2, \circ \cdots \circ a_L)] \tag{2.14}$$

in $O(T \log L)$ by computing intermediary results in parallel (Blelloch, 1990). Compared to a sequential computation method, with time complexity $O(TL)$, the computational speed the parallel scan offers on long sequences cannot be overstated.

Recall that we are interested in computing the hidden state $x_k$ for each sequence element $u_k$. This hidden state is given by equation 2.6, repeated here for convenience.

$$x_k = \overline{\mathbf{A}}^k \overline{\mathbf{B}} u_0 + \overline{\mathbf{A}}^{k-1} \overline{\mathbf{B}} u_1 + \cdots + \overline{\mathbf{A}\mathbf{B}} u_{k-1} + \overline{\mathbf{B}} u_k$$

One can note that the computation of the hidden state uses only associative operators (multiplications and additions with the matrices $\overline{\mathbf{A}}$ and $\overline{\mathbf{B}}$). Any combination of these operators will thus remain associative. Let us now define, for each sequence element, the tuple $c_k = (c_{k,1}, c_{k,2}) := (\overline{\mathbf{A}}, \overline{\mathbf{B}} u_k)$ consisting of the state matrix and the input matrix multiplied with $u_k$. This sequence of tuples $c_0, \ldots c_{L-1}$ will be what the parallel scan algorithm operator over. We define our operator as

$$c_i \bullet c_j := (c_{j,1} \otimes c_{i,1}, c_{j,1} \otimes c_{i,2} + c_{j,2}) \tag{2.15}$$

where $\otimes$ denotes matrix multiplication. We see that, as the parallel scan progresses, the first element in the tuple holds a power of the state matrix equal to the amount of operations already performed. The second element in the tuple contains the (intermediary) results of the hidden state computation we wish to perform. Many of these computations can be performed in parallel. Note for example that in the following

computations

$$
\begin{aligned}
r_1 = c_0 \bullet c_1 &= (\overline{\mathbf{A}}, \overline{\mathbf{B}}u_0) \bullet (\overline{\mathbf{A}}, \overline{\mathbf{B}}u_1) = (\overline{\mathbf{A}}^2, \overline{\mathbf{A}\mathbf{B}}u_0 + \overline{\mathbf{B}}u_1) \\
q_3 = c_2 \bullet c_3 &= (\overline{\mathbf{A}}, \overline{\mathbf{B}}u_2) \bullet (\overline{\mathbf{A}}, \overline{\mathbf{B}}u_3) = (\overline{\mathbf{A}}^2, \overline{\mathbf{A}\mathbf{B}}u_2 + \overline{\mathbf{B}}u_3) \\
r_3 = r_1 \bullet q_3 &= (\overline{\mathbf{A}}^2, \overline{\mathbf{A}\mathbf{B}}u_0 + \overline{\mathbf{B}}u_1) \bullet (\overline{\mathbf{A}}^2, \overline{\mathbf{A}\mathbf{B}}u_2 + \overline{\mathbf{B}}u_3) \\
&= (\overline{\mathbf{A}}^4, \overline{\mathbf{A}}^3\overline{\mathbf{B}}u_0 + \overline{\mathbf{A}}^2\overline{\mathbf{B}}u_1 + \overline{\mathbf{A}\mathbf{B}}u_2 + \overline{\mathbf{B}}u_3)
\end{aligned}
$$

result $r_1$ and intermediary result $q_3$ do not depend on one another and can be computed in parallel. One can see that, given enough processors, for a sequence of length $L$, only $\log_2 L$ intermediary steps are necessary. Note however that the construction of the the tuples over which the algorithm operates requires $O(LP)$ space to store a copy of the state matrix for each element in the sequence, where $P$ is the space complexity of the state matrix. For a general state matrix this would result in $O(LN^2)$ total space complexity, for a diagonal state matrix the space complexity would be limited to $O(LN)$ for each SSM kernel. As mentioned in 2.1.2, LSSLs are instantiated with $H$ unique SSM kernels, resulting in a $O(HLN)$ space complexity for each LSSL layer in the network. Given that feature sizes values of 128 or 256 are not uncommon, the memory requirements for a parallel scan implementation of the LSSL can quickly become unmanagable. See appendix A for a further discussion of this point. To overcome these memory issues two different approaches have been tried.

### 2.1.5 Mamba

Mamba, proposed by the same author who has conducted previous research on HiPPO-matrices and the LSSL, solves the memory issues using hardware-aware state expansion. Instead of preparing the scan input $(\overline{\mathbf{A}}, \overline{\mathbf{B}})$ in GPU high-bandwith memory, the SSM parameters $(\Delta, \mathbf{A}, \mathbf{B}, \mathbf{C})$ are loaded directly into GPU SRAM, where discretization and recurrence are performed (Gu and Dao, 2024). Only the final output of $O(LH)$ is written back into HBM. During training intermediary states are not stored, but instead recomputed in the backwards pass, which significantly reduces memory requirements.

Furthermore, Mamba introduces a selection mechanism to SSM architectures which can be thought of as a linear time complexity variant of the attention mechanism in the transformer. The $\mathbf{B}$, $\mathbf{C}$ and $\Delta t$ parameters are replaced with dense input dependent linear layers

$$
\begin{aligned}
\mathbf{B} &: \mathbb{R}^L \to \mathbb{R}^L \times \mathbb{R}^N & \mathbf{u} &\mapsto s_B(\mathbf{u}) && (2.16)\\
\mathbf{C} &: \mathbb{R}^L \to \mathbb{R}^L \times \mathbb{R}^N & \mathbf{u} &\mapsto s_C(\mathbf{u}) && (2.17)\\
\Delta t &: \mathbb{R}^L \to \mathbb{R}^L \times \mathbb{R}^H & \mathbf{u} &\mapsto \tau(\theta + s_\Delta(\mathbf{u})) && (2.18)
\end{aligned}
$$

where $s_B(\mathbf{u}) = \text{Linear}_N(\mathbf{u})$, $s_C(\mathbf{u}) = \text{Linear}_N(\mathbf{u})$, $s_\Delta(\mathbf{u}) = \text{Broadcast}_H(\text{Linear}_1(\mathbf{u}))$, $\tau(x) = \ln(1 + e^x)$ (softplus) and $\theta \in \mathbb{R}^L \times \mathbb{R}^H$ a set of parameters trained directly via gradient descent. With these changes the matrices $\mathbf{B}$ and $\mathbf{C}$ used in the recurrent computations are no longer static along sequence length, but can vary depending on the provided input sequence. Together with the input dependence of $\Delta t$ the selective SSM no longer represents an LTI system. Convolutional training methods thus no longer apply. We can however use the parallel scan method introduced in section 2.1.4. The benefit of introducing the selection mechanism is that, with selection,

SSMs can handle irregularly spaced sequential data in a natural way. Also, as we will elaborate on in section 2.1.7, discretized SSMs with a selection mechanism are a generalization of the heuristic gating mechanism employed in RNNs. The selection mechanism thus allows for the ignoring of irrelevant inputs, something previous SSMs could not do.

### 2.1.6 S5

The second approach to overcoming the memory issues encountered in earlier SSMs involves densifying the parameter matrices. Where LSSL, S4 and Mamba use an ensemble of many linear single-input, single-output (SISO) SSMs together with non-linear mixing layers, Smith, Warrington, and Linderman, 2023 instead opted to use a single multi-input, multi-output (MIMO) SSM. The model used a reduced latent state size of $P < NH$, operating over an input- and output dimension of size $H$. Rather than broadcasting a single input $u$ across $H$ seperate SSMs, each sequence element is encoded into a $H$-dimensional vector on which the SSM operates as a whole. This approach resulted in the S5 layer. See figure 2.2 for reference.

The S5 layer uses a diagonalized state matrix and feedthrough matrix together with full $\mathbf{B}$ and $\mathbf{C}$ matrices. The learnable parameters of an S5 layer are thus given by $\text{diag}(\mathbf{\Lambda}) \in \mathbb{C}^P$, $\mathbf{B} \in \mathbb{C}^{P\times H}$, $\mathbf{C} \in \mathbb{C}^{H\times P}$, $\text{diag}(\mathbf{D}) \in \mathbb{R}^H$, and $\mathbf{\Delta t} \in \mathbb{R}^P$. The S5 layer uses parallel scan to compute its recurrences. First, each of the $P$ rows in $\Lambda$ and $\mathbf{B}$ are discretized with its corresponding row element from $\mathbf{\Delta t}$. We then construct a sequence of tuples $(\overline{\mathbf{\Lambda}}, \overline{\mathbf{B}}u_k)_{k\in[0..L-1]}$ and use parallel scan to compute the latent states $\mathbf{x} = (x_k)_{k\in[0..L-1]}$ given by equation 2.6 with the use of the binary operator of equation 2.15. $\mathbf{y}$ is now computed similarly to equation 2.7 by multiplying each element in the sequence of $\mathbf{x}$ with $\mathbf{C}$ and adding $\mathbf{Du}$.

Instead of $H$ seperate latent states $\mathbf{x}$ for each independent SISO SSMs, we now have only one latent state for the entire MIMO SSM. Feature mixing now occurs via the input matrix $\overline{\mathbf{B}}$ and the emission matrix $\mathbf{C}$. The final S5 layer output is given by applying a non-linearity to $\mathbf{y}$. Note that the S5 layer does not employ a selection mechanism. The matrices $\mathbf{B}$, $\mathbf{C}$ and $\mathbf{\Delta t}$ are, as before, optimized directly.

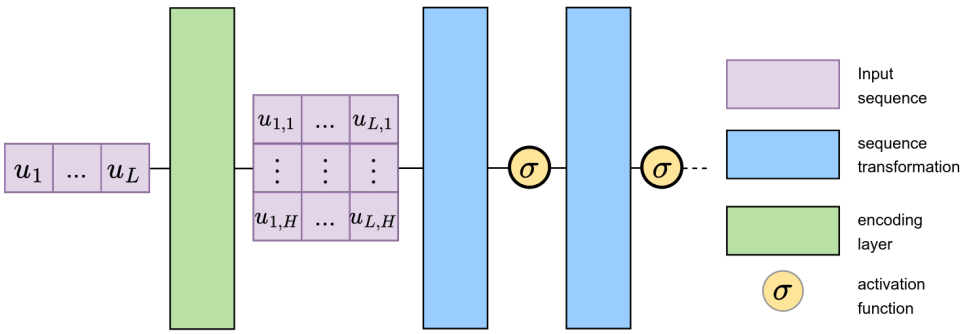



FIGURE 2.2: MIMO SSM design. Each sequence element in $\mathbf{u}$ is encoded to a $H$ dimensional feature vector on which sequence transformations are performed. Feature mixing is performed within the sequence transformations. Seperate mixing layers are not required. Activation functions are used to introduce non-linearity.

### 2.1.7 Relation to RNNs

Recurrent neural networks were the first type of sequence model to use a hidden state to capture temporal dependencies across timesteps (Elman, 1990). Given a latent space dimension of $N$, the recurrent equations capturing temporal dependency of the hidden state $x$ on the input $u$ are given by

$$x_t = g(\text{Linear}_N(x_{t-1}) + \text{Linear}_N(u_t) + b) \tag{2.19}$$

where Linear denotes a linear projection layer to $N$ dimensions, $b$ denotes a bias term and $g$ denotes an activation function (traditionally a sigmoid or tanh). Vanilla RNNs struggle with vanishing and exploding gradients, which limit their use in long-term sequence analysis. Long-short-term memory units (LSTMs) and gated recurrent units (GRUs) were developed to overcome these challenges. GRUs simplified the heuristic gating mechanism employed by LSTMs that were introduced to greatly reduced the problem of vanishing gradients (Chung et al., 2014). Equations for the GRU are given by

$$\begin{aligned}
\mathbf{z}_t &= \sigma(\text{Linear}_N([\mathbf{u}_t, \mathbf{x}_{t-1}])) \\
\mathbf{r}_t &= \sigma(\text{Linear}_N([\mathbf{u}_t, \mathbf{x}_{t-1}])) \\
\hat{\mathbf{x}}_t &= \tanh(\text{Linear}_N([\mathbf{u}_t, \mathbf{r}_t \odot \mathbf{x}_{t-1}])) \\
\mathbf{x}_t &= (1 - \mathbf{z}_t) \odot \mathbf{x}_{t-1} + \mathbf{z}_t \odot \hat{\mathbf{x}}_t
\end{aligned}$$

where the update gate $\mathbf{z}_t$ controls the balance between the preservation of previous hidden states and the inclusion of new information and the reset gate $\mathbf{r}_t$ controls which information is used in computing the to-be-added hidden state.

Due to the explicit dependence of the GRU on both the input and the previous hidden state for its gating computations, the GRU is only trainable using backpropagation through time (BPTT). Thus, training times scale (at least) linearly with sequence length, reducing the efficacy of these models for long-context tasks.

Inspired to adapt the parallel prefix scan algorithm to RNN architectures, Feng et al., 2024 created minGRU, a minimal variant of the traditional GRU architecture which foregoes the dependence of the gating mechanisms on previous hidden states. The hidden state is instead updated as

$$\mathbf{z}_t = \sigma(\text{Linear}_N(\mathbf{u}_t)) \tag{2.20}$$
$$\hat{\mathbf{x}}_t = \text{Linear}_N(\mathbf{u}_t) \tag{2.21}$$
$$\mathbf{x}_t = (1 - \mathbf{z}_t) \odot \mathbf{x}_{t-1} + \mathbf{z}_t \odot \hat{\mathbf{x}}_t \tag{2.22}$$

The minimal GRU variant is significantly more efficient in term of parameter count, training time and memory usage.

As mentioned in section 2.1.5, the discretization of selective SSMs can be seen as the principled foundation of the heuristic gating mechanism employed by RNNs. An illustrative proof was given by Gu and Dao, 2024. Suppose we are given an SSM with $N = 1$, $\mathbf{A} = -1$, $\mathbf{B} = 1$, $s_\Delta(\mathbf{u}) = \text{Linear}(\mathbf{u})$ and $\tau(x) = \ln(1 + e^x)$. The discretization step size is then given by

$$\Delta t = \ln(1 + \exp(\theta + \text{Linear}_N(\mathbf{u})) = \ln(1 + \exp(\text{Linear}_N(\mathbf{u})) \tag{2.23}$$

where in the 1-dimensional case the learnable parameter $\theta$ can be viewed as a bias of the linear projection on $\mathbf{u}$. Discretizing the state- and input matrix with ZOH discretization we find

$$\begin{aligned}\overline{\mathbf{A}} &= \exp(\Delta t \cdot \mathbf{A}) = \frac{1}{1+\exp(\text{Linear}_N(\mathbf{u}))} = \sigma(-\text{Linear}_N(\mathbf{u})) \\ &= 1 - \sigma(\text{Linear}_N(\mathbf{u})) \qquad (2.24)\\ \overline{\mathbf{B}} &= \mathbf{A}^{-1}(\overline{\mathbf{A}} - \mathbf{I}) \cdot \mathbf{B} = \sigma(\text{Linear}_N(\mathbf{u})) \qquad (2.25)\end{aligned}$$

Applying these results to equation 2.3 with single step recurrence we find that

$$\begin{aligned}g_t &= \sigma(\text{Linear}_N(u_t)) \qquad (2.26)\\ x_t &= (1-g_t)x_{t-1} + g_t \cdot u_t \qquad (2.27)\end{aligned}$$

which is exactly the hidden state update equation 2.22 of the minGRU. More generally, the relation between equation 2.19 and 2.3 is clear and, depending on the chosen selection mechanism, classic GRU and LSTM gating mechanisms can be recovered.

## 2.2 Reinforcement Learning

The field of reinforcement learning concerns itself with the study of sequential decision problems in dynamic, stochastic environments. Reinforcement learning models the sequential decision problems as Markov Decision Processes (MDPs), formally denoted by the tuple $(\mathcal{S}, \mathcal{A}, T, r, \gamma)$.

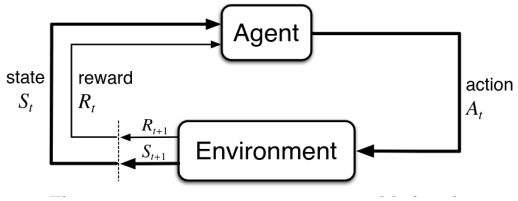



FIGURE 2.3: Agent-environment interaction loop in an MDP. Figure by Sutton, Barto, et al., 2018

The MDP, often referred to as the environment, consists of a set of states $\mathcal{S}$. Upon taking an action $a \in \mathcal{A}$ in a state, the transition function $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0,1]$ denotes the probability of reaches a state $s'$ from $s$ by taking that action. Notably, for any state $s'$, this transition probability only depends on the current state and the taken action. Previous states of the environment can be disregarded. We thus assume that the states conform to the *Markov Property*. The environment includes a reward function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ that determines the reward one receives upon transition from state $s$ to state $s'$ by taking action $a$. The solution to an MDP is the sequence of actions that maximizes the total received reward until either the environment terminates or a set numbers of steps is reached. An agent is thus tasked with optimizing a policy $\pi : \mathcal{S} \to \mathcal{A}$ that determines the best course of action in each state to maximize rewards. Note that in the reinforcement learning setting, the agent does not have explicit access to either the transition function or the reward function.

Suppose we are at timestep $t$ in an MDP. We can define the discounted return of a state as the trace of its future rewards

$$G_t = \sum_{k=t+1}^{N} \gamma^{k-t-1} r_k, \tag{2.28}$$

where N is the amount of timesteps for which the agent is active (can be infinite), $r_k$ is the reward received on step $k$, and $\gamma$ is the discount factor for future rewards. The discounted return an agent receives from a state depends both on the actions the agent takes and the (stochastic) transition function. Thus, to find an optimal policy, we should average the discounted return over all possible traces. We thus define the value function $V$ for a state as

$$V_\pi(s) = \mathbb{E}_{a\sim\pi;s'\sim T}[G_t|s_t = s].$$

where the expectation is taken over all possible transition states $s'$, given that we take actions $a$ from policy $\pi$. A similar quantity can be defined to denote expected return for each state-action pair

$$Q_\pi(s,a) = \mathbb{E}_{a\sim\pi;s'\sim T}[G_t|s_t = s, a_t = a].$$

The explicit goal of reinforcement learning is to find a policy that maximizes the average total return over an episode

$$J(\pi) = \mathbb{E}_{a\sim\pi;s'\sim T}\left[\sum_{t=0}^{N} \gamma^t r_t\right].$$

Using the Bellman equation, we can calculate the value of each state by repeatedly iterating over the state space, updating values based on the attained reward and the (discounted) values of the achievable states

$$V_\pi(s) = \mathbb{E}_{a\sim\pi;s'\sim T}[r + \gamma V_\pi(s')]. \tag{2.29}$$

The runtime of such a dynamic programming method scales exponentially with the dimension of the state space and is as such not feasible for any sufficiently complex problem. Statistical methods using monte-carlo sampling and/or temporal difference learning were required. Many different algorithms have been developed, tried and tested. Earlier value-based methods such as Q-learning and SARSA tried to directly optimize the value functions. Other policy-gradient methods such as REINFORCE instead relied on directly updating the (parametrized) policy by performing gradient ascent with respect to the total return $J$. Later methods combined both approaches to create actor-critic style algorithms, where an implicit policy is updated based on an explicit value function. PPO is such a method.

### 2.2.1 PPO

Proximal Policy Optimization (PPO) is a simplified adaptation of Trust Region Policy Optimization (TRPO) (Schulman et al., 2015) with empirically proven better sample complexity and runtime (Schulman et al., 2017). As other policy gradient methods,

PPO assumes a parametrized policy $\pi_\theta$ and computes an estimator of the policy gradient with respect to its parameters

$$\hat{g} = \hat{\mathbb{E}}_t[\nabla_\theta \log \pi_\theta(a_t|s_t)\hat{A}_t] \tag{2.30}$$

which can be used to perform stochastic gradient ascent. Note that the mathematical expectation is now replaced with an estimated expectation over time. PPO uses Generalized Advantage Estimation (GAE). In the equation above $\hat{A}_t$ is an estimator of an exponentially reweighed average of all possible N-step returns, which is computed using the temporal difference of a parametrized value function $V_\phi$

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + (\gamma\lambda)^{N-t+1}\delta_{N-1} \tag{2.31}$$

where $\delta_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t)$.

Simply using $\hat{g}$ to perform gradient updates results in destructively large policy updates (Schulman et al., 2017). Whereas TRPO constrained policy updates explicitly with respect to a surrogate objective (denoted as conservative policy iteration) given by

$$L^{\text{CPI}}(\theta) = \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}\hat{A}_t\right], \tag{2.32}$$

PPO instead clips policy changes to prevent catastrophically large updates. The main objective proposed by PPO is

$$L^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t\left[\min\left\{\begin{matrix}\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}\cdot\hat{A}_t \\ \text{clip}(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon)\cdot\hat{A}_t\end{matrix}\right\}\right] \tag{2.33}$$

where $\epsilon$ is the PPO clipping constant. We can see that the PPO objective, by taking their minimum, becomes a lower bound of the unconstrained and the clipped objective. The value function of the critic is updated using the squared error loss between the estimated value of a state and the observed cumulative reward from the collected trajectory. To compromise between variance and bias the latter of these values is often computed using the bootstrapped advantage estimate and a target network.

$$L^{\text{VF}}(\theta) = \hat{\mathbb{E}}_t\left[V_\theta(s_t) - (\hat{A}_t + V_{\theta_{\text{old}}}(s_t))\right]^2 \tag{2.34}$$

The trade-off between exploration and exploitation of the network can be influenced by adding a bonus entropy loss given by

$$L^{\mathcal{H}}(\theta) = \hat{\mathbb{E}}_t\left[\mathcal{H}(\theta)\right] = \hat{\mathbb{E}}_t\left[-\sum_a \pi_\theta(a|s_t)\log(\pi_\theta(a|s_t))\right] \tag{2.35}$$

where the summation is replaced with an integral in a continuous action environment.

Noting that we are performing gradient ascent, the total PPO loss is given by

$$\mathcal{L}_t(\theta) = L^{\text{CLIP}}(\theta) - c_1 L^{\text{VF}}(\theta) + c_2 L^{\mathcal{H}}(\theta) \tag{2.36}$$

where the value coefficient $c_1$ and the entropy coefficient $c_2$ are tunable hyperparameters. PPO is an on-policy RL algorithm. During training trajectory rollouts tuples of (at the minimum) observations, taken actions, rewards and episode boundaries $(o, a, r, d)$ are collected using the current network policy. Based on these rollouts, the network is trained using the loss function from equation 2.36. Tricks such as minibatching and repeated batch updating can be readily integrated. See algorithm box 1 for an overview.

**Algorithm 1:** Proximal Policy Optimization

**input :** MDP $(\mathcal{S}, \mathcal{A}, T, r, \gamma)$, #iterations $I$, #environments $N$, #steps $M$, #minibatches $B$, update-epochs $K$, learning rate $\eta$
**output:** policy $\pi_\theta$

Initialize $\theta_{\text{old}}$
**for** $i \leftarrow 1$ **to** $I$ **do**
  **for** $n \leftarrow 1$ **to** $N$ , *using* $\theta_{old}$ **do**
    Collect trajectory rollout $(o_t, a_t, r_t, d_t)_{t \in \{1,\dots,M\}}$
    Compute advantages $\hat{A}_1, \dots \hat{A}_M$
  **end**
  Create $B$ minibatches of length $L$ such that $B \cdot L \leq N \cdot M$
  **for** $b \leftarrow 1$ **to** $B$ **do**
    **for** $k \leftarrow 1$ **to** $K$ **do**
      $\theta \leftarrow \theta + \eta \cdot \nabla_\theta \mathcal{L}_t(\theta)$
      $\theta_{\text{old}} \leftarrow \theta$
    **end**
  **end**
**end**

### 2.2.2 Partially Observable MDPs

The reinforcement learning methods mentioned above require the agent to have access to the full state of the environment. In many real world applications this is not the case. Such problems can instead be modeled as partially observable markov decision processes (POMDPs). Instead of receiving the full state, an agent receives an observation about the current state, which may be limited in scope. Note that the markov property still holds for the environment, but does not hold for the observations the agent receives.

We will define an episodic stochastic POMDP as a tuple $(\mathcal{S}, \mathcal{Y}, \mathcal{A}, T, O, r, \gamma)$ with (now latent) state space $\mathcal{S}$, observation space $\mathcal{Y}$, action space $\mathcal{A}$, transition function $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$, emission function $O : \mathcal{S} \times \mathcal{Y} \to [0, 1]$, reward function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$, and discount factor $\gamma$. The emission function maps a state-observation pair to the likelihood of receiving that observation in the current state. Note that the information an agent receives from a state is thus no longer necessarily unique, as the same observation can have a non-zero chance of being emitted in multiple states.

To extend PPO to partially observable MDPs, equations 2.30, 2.33, 2.34 and 2.35 and derivatives need to be updated to reflect their dependence on observations instead of states. While formal guarantees about the effectiveness of extending PPO to POMDPs are limited, some work has been done on extending PPO to episodic POMDPs (Azizzadenesheli, Yue, and Anandkumar, 2020). In practice, training recurrent network architectures on PO problems with PPO can give good results (Raffin et al., 2021). The only change necessary to adapt PPO for recurrent networks is to initialize and store the hidden states of the networks. During trajectory rollout the hidden state $h$ should thus be added to the tuple $(h, o, a, r, d)$. During advantage estimation the hidden states of networks should be recomputed together with the values of subsequent states.

### 2.2.3 PPO with SSMs

Most implementations of PPO use a static rollout length. A trajectory might thus contain episode boundaries. Convolutionally trained SSMs have trouble dealing with these episode boundaries. While 'done'-signals can be encoded into the observation fed to the SSM, hidden states cannot be reset midway into a sequence during training without a selection mechanism. LSSL- and S4 based architectures are thus ill equiped in the RL setting. Variable length rollouts, on the other hand, are not easily paralellizable and might incur significant runtime overhead. Lu et al., 2023 have found a way to circumvent the issue of hidden state resets by cleverly adapting the binary operator used in the parallel scan algorithm. By redefining the parallel scan tuples to include the environment done signal $c_k = (c_{k,1}, c_{k,2}, c_{k,3}) := (\overline{\mathbf{A}}, \overline{\mathbf{B}}u_k, d_k)$, the operator from equation 2.15 can be replaced with

$$c_i \bullet c_j := \begin{cases} (c_{j,1} \otimes c_{i,1}, c_{j,1} \otimes c_{i,2} + c_{j,2}, c_{i,3}) & \text{if } c_{j,3} = 0 \\ (c_{j,1}, c_{j,2}, c_{j,3}) & \text{if } c_{j,3} = 1 \end{cases} \tag{2.37}$$

After the parallel scan has concluded the second tuple element contains the hidden state associated with each sequence element. If no episode boundary is encountered the binary operator is computed as normal. If an episode boundary is encountered, the hidden state of the SSM is reset and the information from previous inputs is discarded. Suppose for example that we are given a sequence $\mathbf{u} = (u_0, u_1, u_2)$ such that we encounter an episode boundary between $u_0$ and $u_1$. The done signals from the environment would then be given by $\mathbf{d} = (0, 1, 0)$. Constructing our tuples and solving equation 2.37 yields

$$\begin{aligned} c_0 \bullet c_1 \bullet c_2 &= (\overline{\mathbf{A}}, \overline{\mathbf{B}}u_0, 0) \bullet (\overline{\mathbf{A}}, \overline{\mathbf{B}}u_1, 1) \bullet (\overline{\mathbf{A}}, \overline{\mathbf{B}}u_2, 0) \\ &= (\overline{\mathbf{A}}, \overline{\mathbf{B}}u_1, 1) \bullet (\overline{\mathbf{A}}, \overline{\mathbf{B}}u_2, 0) \\ &= (\overline{\mathbf{A}}^2, \overline{\mathbf{A}}\overline{\mathbf{B}}u_1 + \overline{\mathbf{B}}u_2, 0) \\ &= c_1 \bullet c_2. \end{aligned}$$

We can observe that all sequence information from before the episode boundary is discarded. This simple change allows any SSM using the parallel scan algorithm to work natively on RL problems.

SSMs with a selection mechanism might be able to forego this change in binary operator. As mentioned in section 2.1.5, SSMs with a selection mechanism are in

principle able to learn heuristic gating mechanism. If the done signal of an environment is provided as part of the observation, a selective SSM could in theory learn to increase the timestep parameter $\Delta t \to \infty$ such that the hidden state computed from earlier sequence elements are effectively discarded when a done signal is encountered. This allows for the handling of episode boundaries in a generic, non-hardcoded way. Selective SSMs might thus be utilized in RL with minimal changes in architecture.

# Chapter 3

# Related Work

In this chapter we cover existing research on SSMs, providing a condensed timeline of the evolution of these models. An in depth discussion of the SSMs can be found in the previous chapter. We also highlight previous work on the use of sequence models in reinforcement learning, which problems they solved and which problems are still open.

## 3.1 Structured State Space Models

The use of structed state space models in deep learning is a recent development, starting only this decade. Starting in 2020, Gu et al., 2020 introduced HiPPO matrices, formulating a family of structured state space models that can efficiently represent continuous-time dynamics in a discretized manner. Building upon this work the Linear State Space Layer was the first SSM-based architecture that could be incorporated into existing neural networks as a standalone layer (Gu et al., 2021). LSSLs can be trained with global convolutions, allowing parallel training on sequential information. Inference can be done in a recurrent manner, using constant memory and runtime per query. These properties make SSM-based architectures an interesting replacement for transformers. However, LSSLs can suffer from numerical instability and high memory usage. Two approaches to solving this issue have been tried.

The first approach used fourier transforms and a combination of three convolutional kernels together with a special class of diagonal-plus-low-rank state matrices to streamline computations, resulting in the S4 architecture (Gu, Goel, and Ré, 2022). Due to its increased efficiency and memory capacity, S4 was able to achieve state of the art results on the Long Range Arena benchmark (Tay et al., 2021) and was the first model to solve the 16k length Path-X problem. The S4 architecture has since been refined, streamlining the state matrices to a diagonal form without conceding much expressive power (Gupta, Gu, and Berant, 2022) (Gu et al., 2022).

The reduced memory usage from diagonal state matrices allowed Smith, Warrington, and Linderman, 2023 to create the S5 model, which, as an alternate approach, switched from using global convolutions during training to using a parallel scan algorithm (Blelloch, 1990). This reduced forward pass time complexity of a sequential datastream of length $L$ from $O(L)$ to $O(\log L)$. Whereas previous models used single-input-single-output (SISO) SSMs for each feature in the data, S5 used a linear encoder with a single multi-input-multi-output (MIMO) SSM, which greatly reduced memory usage as only one SSM needs to be instantiated per layer. Later work by Gu and Dao, 2024 kept the SISO design but instead loosened the imposed LTI restriction on the SSM.

The so called Selective Scan State Space Model (S6) introduced a selection mechanism which allowed the SSM parameters to vary based on the input to the model. As an example, whereas previous models discretized with the same timestep regardless of input, S6 could vary this timestep allowing for variable input spacing and context filtering. Gu and Dao, 2024 argue that the selection mechanism is the principled foundation of heuristic gating mechanism found in Recurrent Neural Networks. Together with a hardware-aware state expansion mechanism the S6 based architecture Mamba has become a popular choice for long-sequence tasks. The architecture has since been refined to for even greater speed-ups (Dao and Gu, 2024) and has been succesfully applied in a variety of domains including audio processing, text processing and medical imaging (Bansal et al., 2024).

## 3.2 Sequence models in RL

The development of efficient sequence models is of great importance for reinforcement learning, as sequence modeling has played an increasingly important role in settings where partial observability, long-term dependencies and memory are critical. Recurrent Neural Networks (RNNs) were among the earliest sequence models used in RL. Hausknecht and Stone, 2015 introduced Deep Recurrent Q-Networks (DRQNs), demonstrating that RNNs can effectively handle partial observability by maintaining a hidden memory state over time. LSTMs and GRUs have since become common in partially observable RL, particularly in domains like robotics, navigation, and dialogue systems (Heess et al., 2015). However, these recurrent approaches train via backpropagation-through-time (BPTT), which results in training times scaling with sequence length and gives rise to issues such as vanishing gradient problems. Other methods such as encoding reinforcement learning algorithms directly in the weights of an RNN (Duan et al., 2016) have seen limited success.

Transformers, originally designed for natural language processing, have recently been adopted in RL due to their ability to model long-range dependencies without recurrence. Notable examples include the decision transformer (DT) (Chen et al., 2021), which casts the problem of RL as conditional sequence modeling. DT did not use value functions or policy gradients, instead opting to output optimal actions by using a causally masked transformer conditioned on the desired reward, past states and actions. The core idea of integrating as much as possible of the trajectory optimization pipeline into the modeling problem has since been extended to diffusion models (Janner et al., 2022) and has been generalized to meta-RL (Lee et al., 2022), (Kirsch et al., 2024). Other research instead opted to use employ sequence models to learn latent state representations of the RL environments while keeping in line with traditional value- and policy based RL methods. Earlier RNN based world models (Ha and Schmidhuber, 2018) have been successfully extended to use transformers (Chen et al., 2022). The combination of world model learning and actor critic learning has allowed for general algorithms that can outperform even specialized methods across a wide variety of tasks, even being able to collect diamonds in Minecraft from scratch without human data or curriculum learning (Hafner et al., 2023).

SSMs, combining the inferential efficiency of recurrent approaches with the sequence modeling power of transformers have been tested in reinforcement learning settings with some success in both In-Context-RL (Lu et al., 2023) and offline RL (Bar-David et al., 2023). Whereas Morad et al., 2023 have found that naively using

S4 models in POMDPs did not perform well, other recent works have shown that S4 (Bar-David et al., 2023) and Mamba (Ota, 2024) based models can outperform decision transformer on a variety of benchmarks. As of now finetuning task-specific SSM models seems a necessity. Larger model scales, improvements to the parallel scan operators and selection mechanism employed by SSMs might however change this.

While SSMs continue to gain popularity in the RL domain, little is known about their performance in the single-environment on-policy setting. Effort has thus far been mostly expended on their application to meta-learning and their use as autoregressive models which can be conditioned on desired returns. In the coming chapters we will therefore study how SSMs perform on POMDPs compared to baseline methods when trained using a policy gradient method.

# Chapter 4

# Experiments

As discussed in the previous chapters, the generic sequence-to-sequence modelling capabilities of SSMs are promising for next-generation text, image, video and audio processing tools. The recurrent structure of SSMs also allows us to apply these models to POMDPs, where generalizing over multiple observations can significantly improve model performance. To study the performance of SSMs in the reinforcement learning setting we have implemented LSSL(f)-, and S5-based architectures. The LSSL(f) architectures have been implemented both in the spirit of the original implementation (Gu et al., 2021) using Krylov functions and convolutional training schemes, and in a diagonalized fashion using parallel scan. We will refer to the latter implementation as LSSLf-diag. We have also implemented a minGRU-based architecture (Feng et al., 2024) capable of using parallel scan.

To test the validity of the SSM implementations we have first implemented classification models using the same core architectures and tested their performance on the the sequential MNIST dataset. Implementation details and results can be found in appendix A. In the classification setting both LSSLf- and S5 based architectures show good generalization capacities, converging to optimal performance on the training and test data. However, the LSSLf-diag implementation was severely limited by the amount of available GPU memory. Architectures with a feature dimension $H > 32$ could not be run locally. As mentioned in section 2.2.3, the Krylov function based approach to SSM implementations will not translate well to reinforcement learning problems, since these models are not able to handle environment resets. We have thus decided to focus the experiments on the performance of the S5 architecture. Section 4.3 details the hyperparameter search conducted to find an optimal configuration. In section 4.4 this configuration is compared and contrasted with baseline models on a variety of partially observable environments. We conclude with an extensive study of the impact of the latent space dimension on the capability of S5-based models to memorize important information. First however, we will describe the deep network architecture we have adopted for these experiments and discuss the coding level decision which we have found were required to create performant models.

## 4.1 Deep network Architecture

For the experiments we use an actor-critic style architecture where different sequence transformation layers can be used as drop-in replacements. Care has been taken to design an architecture which works efficiently during both inference and training.

During the PPO training loop, many vectorized environments will be run in parallel for rollout collection. After rollout collection the data from the environments is divided into minibatches on which the model is trained sequentially.

### 4.1.1 S5

The S5 based architecture changes the interleaved broadcasting layer used in the LSSL-based models to a linear encoding layer. This layer encodes the transition minibatch observations to shape $(L, H)$ where $L$ denotes the sequence length and $H$ the encoded feature dimension. This encoded observation sequence is fed to a sequence of residual blocks (He et al., 2016) consisting of an S5 SSM, a non-linear activation function, a dropout layer, a residual connection and a (layer) normalization. Note that each block has its own S5 SSM with a separate hidden state. See figure 4.1 for a schematic overview. The residual blocks can be chained indefinitely. The output from these blocks is fed to a feedforward (FF) actor- and critic head.

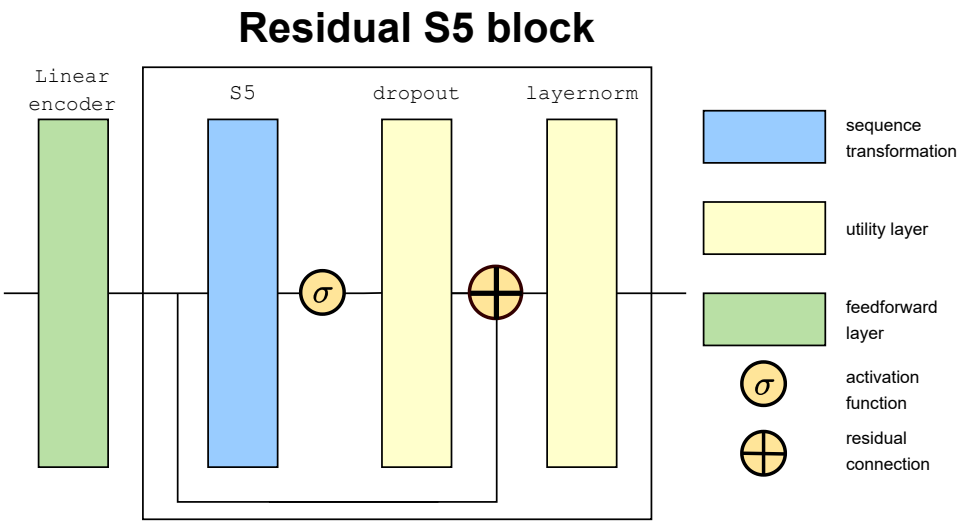



FIGURE 4.1: Architecture of a residual S5 block. The first block requires a linear encoder to cast the $\mathbf{u} \in \mathbb{R}^{|\mathcal{O}|}$ input to $\mathbb{R}^H$. Subsequent blocks do not require an encoder. Non-linearity is introduced by the activation function.

### 4.1.2 minGRU

The minGRU architecture resembles the S5 architecture. Sequences are encoded and fed to a series of residual blocks consisting of an minGRU layer, a non-linear activation function, a dropout layer, a residual connection and layer normalization. Note however that, as can be inferred from equations 2.20 to 2.22, a minGRU layer returns an output of dimensions $(L, N)$, not one of size $(L, H)$ as the S5 model would. This is accounted for in the encoding. The last hidden state computed by each minGRU block is stored and used in subsequent model calls.

### 4.1.3 Baseline models

As baseline models we employ a Gated Recurrent Unit (GRU) (Chung et al., 2014) and a traditional feedforward actor critic network. The GRU uses the same hidden state dimension as the S5 model and an FF actor- and critic head. The fully FF model consists solely of an actor- and critic head (no sequence transformations).

## 4.2 Coding Level Decisions

As mentioned in appendix A, several concessions to generality were required to create performant S5-based SSM models. Foremost, a necessity discovered independently from Gupta, Gu, and Berant, 2022, the real values of $\Lambda$ were restricted to to the negative plane. Positive real values can incur numerical instabilities, particularly for long sequence lengths. This is due to the fact that the state dynamics governing SSM behaviour contain exponential functions. Consider that the ansatz solution to a continuous differential equation of the form $\dot{\mathbf{x}}(t) = \Lambda\mathbf{x}(t)$ is given by $\mathbf{x}(t) = e^{\Lambda t}$. Any eigenvalues with positive real parts would thus be amplified exponentially along the sequence dimension.

On the other hand, the discretization timesteps $\Delta t$, in line with their physical interpretation, were restricted to positive values. This in turn prevents undefined behaviour. In line with findings by Smith, Warrington, and Linderman, 2023, weight decay was employed on the $\mathbf{C}$ and $\mathbf{D}$ matrix. This significantly impacted model learning. Furthermore, the addition of dropout and layer normalization and the use of a residual connection were hugely beneficial in the classification setting. While layer normalization and residual connections are seen in other works on the application of SSMs in RL (Lu et al., 2023), the use of dropout is less entrenched.

One of the most influential engineering decisions, found in the works of Lu et al., 2023, was the addition of an action embedding wrapper for the tested environments. This wrapper adds the last taken action (or a flag for the start of a new episode) to the observation fed to the model. As can be seen in figure 4.2, the addition of this wrapper can be the difference between almost perfect performance and comparative underperformance.

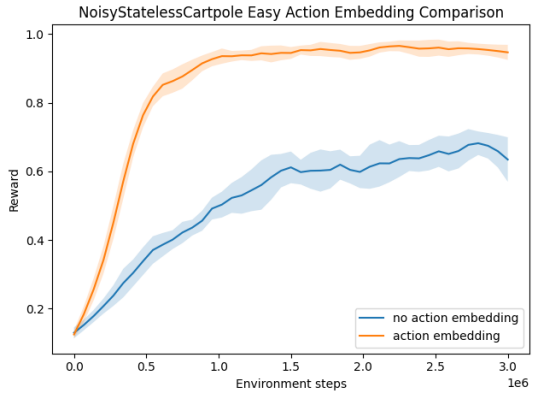



FIGURE 4.2: Mean normalized returns on the NoisyStatelessCartpoleEasy environment for a S5 RL agent using action embedding (orange curve) and an agent not using action embedding (blue curve). Results are averaged over 5 runs. The shaded region represents the standard deviation over these runs.

As mentioned in section 2.1.3, enforcing conjugate symmetry during training allows us save a large amount of memory during training. We have thus chosen to enforce this restriction. In line with the reasoning found in section 2.1.1 we have opted to use the zero-order hold discretization method. Models were trained using the adam optimizer (Kingma and Ba, 2017). Gaussian error Linear units (GeLu) were the non-linear activation function of choice.

## 4.3 Sensitivity Analysis S5

Optimal configuration of the S5 model might differ between classification- and RL tasks. As such, a good first experiment is tuning the hyperparameters of both the model and the PPO algorithm. Due to restrictions on runtime, the scope of the hyperparameter tuning was kept small. The S5 model was trained on both the StatelessCartPoleEasy (SCPE) and the harder NoisyStatelessCartpoleMedium (NSCM) environment from the popjaxrl suite for a variety of different hyperparameters. SCPE mimics the classic CartPole environment known from the Gymnasium suite (Towers et al., 2024). The environment is made partially observable by removing the velocity components of the pole from the observation space. NSCM additionally adds gaussian noise to the returned observation, obfuscating the true signal. Preliminary studies of these environments showed that SCPE can be fully solved, while even finely tuned algorithms struggle on NSCM. Comparing performance on these environments for a variety of hyperparameter configurations can therefore provide insight in the sensitivity of S5 models to their hyperparameters. The search space of the PPO hyperparameters was limited to the SSM learning rate, maximum gradient coefficient, entropy coefficient, the number of vectorized environments, the number of steps collected into the transition buffer and the number of update epochs performed on each buffer. The learning rate of other network weights was fixed to 5 times the SSM learning rate.

TABLE 4.1: Search space of the S5 model- and PPO hyperparameters on both the SCPE and the NSCM environment. Hyperparameters are either logaritmically spaced, linearly spaced or of integer type. The steps column refers to the amount of steps taken in the search space. If a hyperparameter is of integer type steps are taken by multiplying with a factor of 2.

| | Parameter | Spacing Type | Lower Bound | Higher Bound | Steps |
|---|---|---|---|---|---|
| PPO | Learning rate | logarithmic | $10^{-5}$ | $10^{-2}$ | 4 |
| | Maximum gradient coefficient | linear | 0.5 | 1 | 2 |
| | Entropy Coefficient | logarithmic | $10^{-3}$ | $10^{-1}$ | 3 |
| | Number of environments | integer | 16 | 64 | 3 |
| | Number of update steps | integer | 128 | 1024 | 4 |
| | Number of update epochs | integer | 2 | 32 | 5 |
| S5 | # residual layers | integer | 2 | 6 | 3 |
| | H | integer | 64 | 512 | 4 |
| | Actor Network hidden layer size | integer | 32 | 128 | 3 |
| | Critic Network hidden layer size | integer | 32 | 128 | 3 |

The search space of the S5 parameters was limited to the number of residual layers, the feature dimension size and the the hidden layer sizes of the actor- and critic feedforward networks. See table 4.1 for an overview of all parameters studied. The size of the latent space dimension was fixed to 256. The impact of this parameter will be studied extensively in section 4.5. In total 50 random combinations of parameters were tested. Results were averaged over three random seeds.

Note that while this experiment may give insight into well performing hyperparameter combinations for S5-based architectures, optimal hyperparameter configurations might still differ between environments.

## 4.4 Model Comparison Across Environments

To test the performance of the S5 model in the RL setting we have performed a sweep over the popjaxrl suite of partially observable MDPs (Lu et al., 2023). We have ran the S5 model, and the baseline minGRU, GRU and FF models on all environments to observe if the S5-based architecture manages to (on average) outperform baseline implementations without a significant increase in runtime. Based on the results of the hyperparameter search from section 4.3, the hyperparameters in this and the following experiments were set to the ones shown in table 4.2 unless otherwise explicitly mentioned. All models use the same set of PPO hyperparameters and the same actor- and critic head architectures. The minGRU and GRU latent space dimension are also set to size 256.

TABLE 4.2: PPO and S5 Hyperparameters used in the experiments of section 4.4 and section 4.5.

| | **Parameter** | **Value** |
|---|---|---|
| PPO | SSM Learning rate | $10^{-5}$ |
| | Learning rate | $5 \cdot 10^{-5}$ |
| | Maximum gradient coefficient | 0.5 |
| | Clip coefficient | 0.2 |
| | Entropy Coefficient | 0.0 |
| | Number of environments | 64 |
| | Number of update steps | 1024 |
| | Number of update epochs | 30 |
| S5 | # residual layers | 4 |
| | N | 256 |
| | H | 256 |
| | $\Delta t$ bounds | $(10^{-3}, 10^{-1})$ |
| | Actor Network hidden layer size | 128 |
| | Critic Network hidden layer size | 128 |

## 4.5 Memory size efficiency

One of the purported advantages of SSMs is their ability to generalize to long context windows. The application of the continuous time memory updates based on the HiPPO framework in these models should allow for efficient updates to the models hidden state. One might conclude that the ability to store information in the hidden state is therefore limited mostly by the size of the latent space dimension. To test this hypothesis we test S5 models with different hidden state sizes on the memory corridor environment.

The memory corridor is an RL environment designed to test memory retention in recurrent models (Moerland et al., 2024). The state space consists of a sequence of corridors of length $T$, where each corridor contains $N$ doors. In each corridor, only one door grants passage to the next. The first time a corridor is reached the correct

door number is provided as a one-hot encoded observation to the agent. Selecting the correct door puts the agent at the start of the corridor sequence. All previous actions need to be repeated to arrive to the next door in the sequence. Selecting any of the $N-1$ other doors transitions to a terminal state, resetting the environment. Previously seen doors provide a null-observation. Every correctly chosen door provides a reward of one (1) point. The total reward for reaching the $n$-th door thus scales as $\frac{1}{2}n(n-1)$. We have implemented the Memory Corridor environment in the Gymnax framework (Lange, 2022), inspired by PopJaxRL (Lu et al., 2023). We have also implemented an easy variant, that provides the current agent position and the correct number for the last door in the corridor as an observation each frame.

Using these environments, we test if larger latent space dimensions $N$ allow S5 models to scale to larger context windows and allow for better dynamic memory updates. We test values ranging from 8 to 256. Note that the memory corridor is an especially hard environment since the same model hidden state needs to encode different length sequences depending on the trajectory of the agent. As a comparison we also run baseline GRU models with different hidden sizes to showcase the difference in generalization capacity.

For the memory corridor environment it was found beneficial to use a gated linear unit (GLU) as the activation function in the residual layers of the S5 model. The equation for this activation function is given by

$$\mathrm{GLU}(\mathbf{y}) = \mathrm{Linear}_y(\mathrm{GeLu}(\mathbf{y})) \cdot \sigma(\mathrm{Linear}_y(\mathrm{GeLu}(\mathbf{y}))) \tag{4.1}$$

where $\mathrm{Linear}_y$ are dense linear projection layers and GeLu is the Gaussian error Linear unit.

# Chapter 5

# Results & Discussion

In this chapter we showcase and discuss the results of the experiments laid out in chapter 4. Section 5.1 discusses the results of the hyperparameter sensitivity analysis, 5.2 shows how average performance differs between S5, GRU and FF models on partially observable environments and 5.3 shows the relation between model hidden state size and memory capacity. Experiments have been performed on an NVIDIA GA104M GPU with 8 GB of VRAM. All results, unless otherwise stated, have been averaged over at least 5 seeds to mitigate the chance of outliers influencing results. When applicable, figures are smoothed with a second order Savitzky-Golay filter using a window size of 7.

## 5.1 Hyperparameter Sensitivity

As outlined in section 4.3 we conducted a random search over the hyperparameter ranges specified in table 4.1 on both the Stateless CartPole Easy- (SCPE) and the Noisy Stateless Cartpole Medium (NSCM) environment. The results of this search can be observed in figure **??**. Non-surprisingly, the performance of S5-based models is worse on the NSCM environment compared to on SCPE. The best performing models achieved 60% and 99.9% of the maximum reward respectively. More notably, we can see a difference in model performance dropoff rate and in the percentage of models achieving the (attainable) maximum reward. More than 80% of the tried S5 models achieve at least 0.8 of the maximum achieved reward on the SCPE environment. On the NSCM environment only 25% of models attain this 80% mark. Moreover we can see that the rate at which model performance declines is significantly steeper for the more complex NSCM environment. We can conclude that as environment complexity increases, the S5 model becomes more sensitive to hyperparameter tuning.

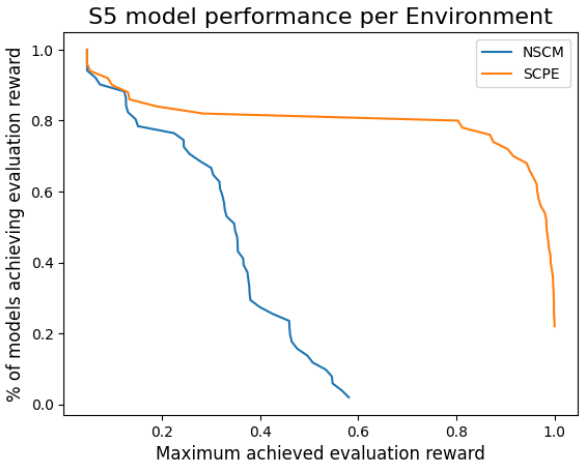
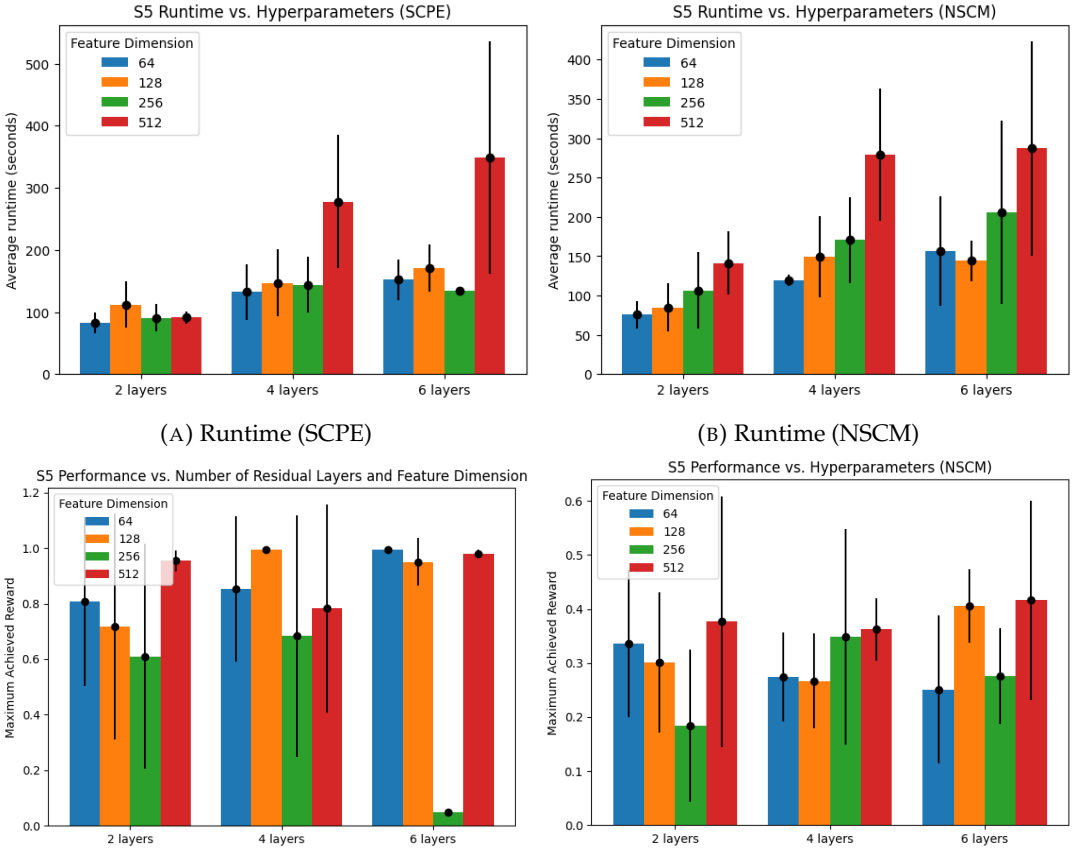



FIGURE 5.1: A comparison of S5 model performance on the NSCM (blue line) and SCPE (orange line) environment. 50 models with different hyperparameter configurations were trained on both environments. The curves show the percentage of models that are able to achieve a (normalised) reward level in their respective environments.

The results of this search were used to decide on a hyperparameter configuration for later experiments. To account for limitations in hardware, a balance between model runtime and model performance needed to be struck. Figure 5.2 shows a quantitative analysis of these metrics as a function of the most influential hyperparameters: the size of the feature dimension $H$ and the amount of residual layers in the network architecture. From figure 5.2a and 5.2b we can see that for deeper network architectures, runtime significantly increases when the feature dimension is scaled beyond $H = 256$. Below this threshold little difference is found. We can also observe that there is little cost to adding more residual layers to the network. The results from figure 5.2c and 5.2d are less clear. The average performance can vary wildly within a single configuration category. We can conclude that the two hyperparameter in the graph are, on their own, not enough to predict model performance a priori. Note also that the standard deviation for some categories is missing. This can point either to the fact that all tried configurations have achieved the same maximum reward during evaluation, or to the fact that only a single configuration within the category has been ran. These instances are observed only in the SCPE case, where three out of four instances are of models achieving the maximal attainable performance. We should be cautious with our conclusions based on this data. A more extensive hyperparameter search might achieve more substantive results.




(A) Runtime (SCPE)

(B) Runtime (NSCM)

(C) Performance (SCPE)

(D) Performance (NSCM)

FIGURE 5.2: Average S5 model runtime (top row) and maximum achieved reward during evaluation (bottom row) during hyperparameter search on the SCPE (left) and NSCM (right) environments. Both metrics are plotted as a function of the amount of residual layers in the network architecture and the dimension of the feature space $(H)$. Standard deviations of the results are displayed as black lines.

## 5.2 Comparison Across Environments

Using the results from section 5.1 to inform our hyperparameter configuration for the S5 model and the PPO configuration for all models, as found in table 4.2, we performed a sweep across all environments from the popjaxrl suite (Lu et al., 2023). As can be observed in figure 5.3, our S5 model outperforms baseline implementations. While impressive, performance between environments still differs widely (as could be expected without individual tuning). Standard deviations have thus been omitted from this graph for the sake of clarity. Graphs of runs for each individual environment can be found in appendix B. Also of note is the fact that the minGRU model, whilst underperforming compared to the S5 model, outperforms the standard GRU baseline in a fraction of the runtime. We can conclude that the removal of the reset gate and the linearization of the hidden state update are beneficial to this architecture. Figure 5.4 shows the normalized MMER achieved by each model averaged across environments. Final MMER values, together with average model runtime, can be found in table 5.1. We can see that S5 models use only a fraction of the runtime compared to BPTT trained GRUs. The runtime of the S5 model and the minGRU model are of comparable order. It should be noted that the runtime of FF models is still significantly lower. The FF models do however perform the worst of the tested models, which is to be expected in partially observable environments. We can conclude that in cases where generalization across sequence dimension is not important FF models might still be the more prudent choice compared to SSM models due to their lower runtimes. It should also be noted that while the S5, minGRU and GRU curves in figure 5.3 have not yet plateaued completely. These models might thus benefit more from longer training times and more data samples which were not achievable in our setting due to limitations in runtime.

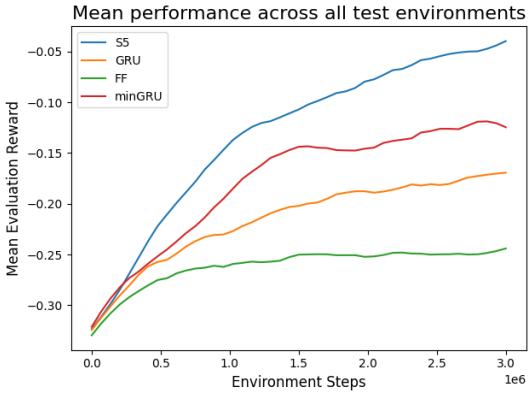



FIGURE 5.3: Mean evaluation rewards achieved by the S5, minGRU, GRU and FF models (from best performing to worst performing) as a function of environment steps achieved on the environments from the popjaxrl suite.

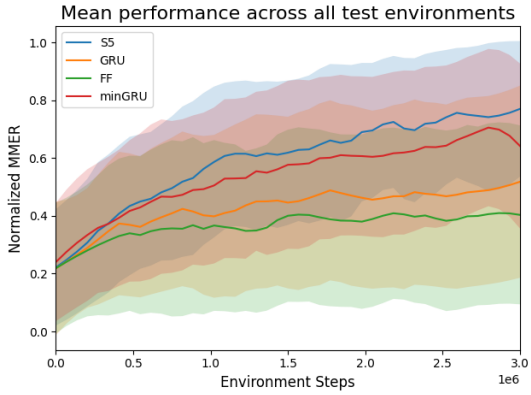



FIGURE 5.4: Normalized MMER achieved by the S5, minGRU, GRU and FF models (from best performing to worst performing) as a function of environment steps achieved on the environments from the popjaxrl suite. The colored shaded regions denote standard deviations.

| Model | Across Environment MMER | Average Runtime (seconds) |
|---|---|---|
| FF | $0.33 \pm 0.43$ | $\mathbf{25.73 \pm 8.90}$ |
| GRU | $0.53 \pm 0.33$ | $765.54 \pm 59.17$ |
| minGRU | $0.71 \pm 0.27$ | $214.31 \pm 14.15$ |
| S5 | $\mathbf{0.82 \pm 0.19}$ | $253.31 \pm 5.76$ |

TABLE 5.1: Comparison of final model performance and training runtimes on all tested environments. Best achieved results are bolded.

## 5.3 Memory Capacity

As outlined in section 4.5, S5 and baseline GRU models were tested on the Memory Corridor environment for a variety of different hidden state dimensions. Figure 5.5 shows the resulting learning curves and their standard deviations. S5 models outperform baseline GRU models, but performance does not seem dependent on the hidden state size. Final performance of all models is visualized in 5.6. We can see that there is no significant difference in performance between models with a different latent space dimension. There are a few possible reasons as to why this might be the case. The first is that due to the complexity of the environment, a more extensive hyperparameter tuning is needed. Performance in this case is not bottlenecked by the hidden state dimension, but either by the configuration of the PPO training loop or another part of the network architecture. Alternatively, the S5 models might require more environment steps to achieve a notable difference in performance. As mentioned in appendix A, S5 models can be relatively data hungry compared to other architectures. To test this hypothesis we have performed a single run of the $N = 8$ and $N = 256$ S5 architectures for $10^8$ timesteps, which is displayed in figure 5.7.

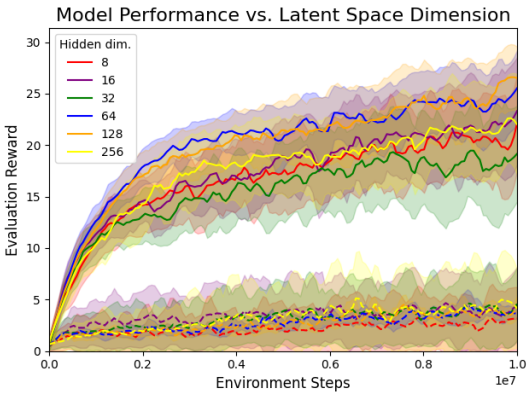



FIGURE 5.5: Comparison of S5 (full) and GRU (dashed) model performances on the Memory Corridor environment for different latent dimension sizes.

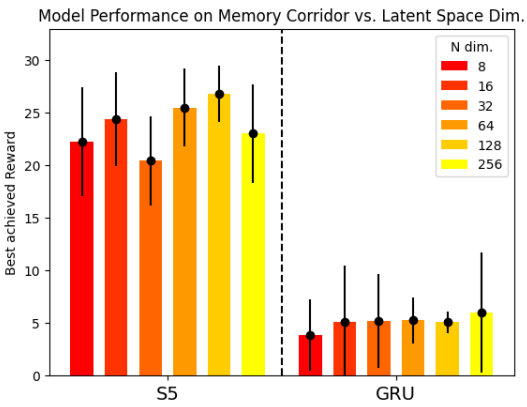



FIGURE 5.6: Maxmimum achieved model performance on the Memory Corridor environment for S5 and GRU models with different latent space dimensions.

In figure 5.7, we can see the S5 model with a hidden state dimension of $N = 256$ outperforming the $N = 8$ model, even achieving average testing scores of 45 (i.e. traversing a 10-door long corridor). Any conclusions drawn from this results should however be limited. A more extensive study of longer runtimes is unfortunately prohibited by resource constraints.

Another possible explanation can be found in the observation structure of the environment. Not all observations in the memory corridor environment are relevant for completing the task at hand. Integrating these observations into agent memory might even be detrimental to performance. Furthermore, the better an agent performs the higher the ratio between useless- and useful information becomes. Architectures employing a selection mechanism, such as the later Mamba models (Gu and Dao, 2024) could in theory prevent detrimental memory updates with irrelevant information, allowing for longer context scaling without loss in fidelity. It might be interesting to test these models on this same environment.

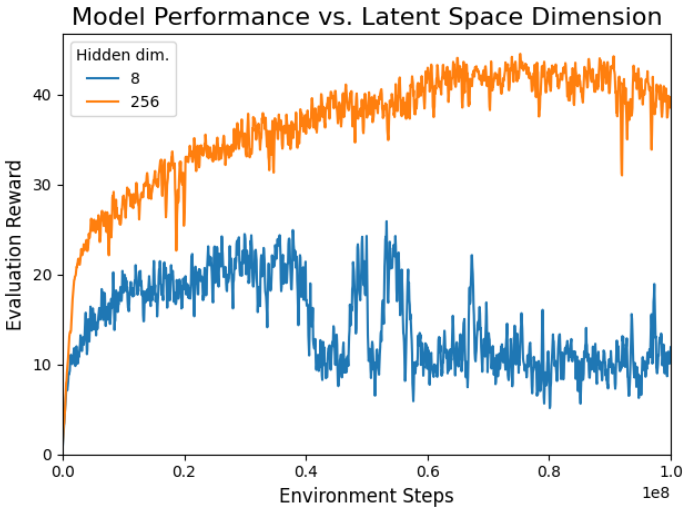



FIGURE 5.7: Comparison of S5 models with hidden state size $N = 8$ (bottom) and $N = 256$ (top) on the Memory Corridor. A single run of each model is shown.

# Chapter 6

# Conclusion

In this work we have studied the application of SSMs to partially observable reinforcement learning environments. SSMs, due to their constant time and memory usage per inference step on sequential data, have become a popular choice for long-range dependency modeling tasks. SSMs outclass transformers on asymptotic runtime and model scalability, and traditional recurrent architectures on performance benchmarks.

With the the necessary theory to implement SSM models for partially observable MDPs outlined in chapter 2 and an overview of contemporary literature in the field provided in chapter 3, chapter 4 focussed on the details of our experimental setup and the coding level decisions necessary to create functional models. Noting that implementations of LSSL-based models were not suitable for RL experiments, we focused our efforts on the S5 architecture. Based on the results discussed in chapter 5, we can answer our research questions as follows.

**RQ 1** How sensitive are SSMs to hyperparameter tuning in the RL setting?

As discussed in section 5.1, model tuning becomes increasingly important as environment complexity grows. For simpler environments, S5 models are robust to suboptimal hyperparameter settings. While no single model parameter showed a clear performance difference, we have found that a larger dimension of the feature space $H$ significantly increases model runtime. While individual tuning per environment remains important, we have identified a hyperparameter configuration that allows for reasonably consistent performance on most environments.

**RQ 2** How does SSM performance compare to baseline models on PO RL tasks?

We have demonstrated that S5 models outperform baseline models on partially observable environments and that they do so in a fraction of the runtime compared to their GRU counterpart. Their efficacy in solving PO tasks does however come at the cost of longer runtimes compared to feedforward networks, making FF architectures a contender on non-PO environments.

**RQ 3** What is the influence of the latent space dimension on the effective context window size of the model?

While preliminary results point to a positive relation between model latent space dimensionality and memory capacity, more research is needed to validate this conclusion.

## 6.1 Limitations and Future Work

There are clear limitations to the results in this work. Due to restrictions in equipment and time, all experiments had to be limited in scope. The conducted hyperparameter search was limited to a fraction of the tunable parameters, and the search spaces were limited in size. The search space was searched using random search, with no heuristically- or theoretically founded method for preferential search. A more extensive hyperparameter search might thus be interesting to conduct.

Also, the GRU architecture used as a baseline was less optimized than the S5 architecture. Different hyperparameters and a deeper network with more residual layers might achieve a better performance than the one attained in this work. Do note that GRU runtimes would quickly become unmanagable for deeper networks. It might thus be more interesting to focus on the streamlined minGRU architecture, which outperforms its predecessor in a fraction of the runtime. Observing the fact that the minGRU performance falls squarely between the S5 and GRU performance on might wonder if with better optimization the minGRU architecture could achieve performance on par with SSM architectures. As discussed in section 2.1.7, the minGRU can be seen as a special case of the general SSM model. It might thus be possible to use a similar HiPPO initialization scheme for the latent state update layer in the minGRU. Note that this initialization would only speed up initial training. In the high update regime even the current model might achieve a similar performance to the S5. This remains to be tested.

The omission of a transformer baseline from this work can be explained by limitations in scope but should nevertheless be rectified in any later works. As the current state-of-the-art sequence models their performance and runtime should be compared to that of SSMs in the RL setting.

As noted in section 5.3, results on the influence of hidden state size on memory capacity of S5 models was mostly inconclusive. Future experiments might either use a simpler environment such as the T-maze environment from Bsuite (Osband et al., 2020), or would need to devote more computational power to this experiment.

Furthermore, there were other coding level decisions whose influence has been noted but could not be studied fully. The impact of the action-embedding wrapper, explained in section 4.2, on model performance was undoubtedly significant but its impact has not been tested on all environments. The discretization method of the state- and input matrices was set to zero-order hold in line with recent works, but a full comparison of different discretization methods has not been performed. Other candidates for more extensive study include the inclusion and tuning of dropout in the residual blocks, the use of batch normalisation vs. layer normalisation and the impact of enforcing conjugate symmetry on final performance, runtime and memory usage. S5 models might also benefit from the use of the S4D-real diagonalized state-matrix in the RL domain (Gu et al., 2022) which foregoes initializing with complex eigenvalues altogether.

As noted in appendix A, S5 architectures seem to benefit from high update-to-data ratios. Another interesting adaptation of the PPO algorithm might be to let the trajectory length during rollout collection increase depending on model performance. This would allow for a higher update-to-data ratio without compromising

early training speeds. Similarly, S5 models could be tested in an offline reinforcement learning setting.

Another interesting direction for research is the inclusion of a selection mechanism in the S5 architecture. S5 outperforms S4 on a variety of benchmarks (Smith, Warrington, and Linderman, 2023), but comparisons between S5 and Mamba are lacking. Adding a selection mechanism to S5 allows us to compare SISO and MIMO design philosophies in a fair setting. Studying the influence of selection mechanisms on RL tasks can in and of itself be an interesting endeavor. As noted by Gu and Dao, 2024, the binary operator introduced by Lu et al., 2023 acts as a hardcoded selection mechanism setting $\overline{\mathbf{A}} = 0$ on episode boundaries. It might be interesting to study if selective SSMs learn to copy this behaviour with no change in binary operator.

# Bibliography


Azizzadenesheli, Kamyar, Yisong Yue, and Animashree Anandkumar (May 2020). "Policy Gradient in Partially Observable Environments: Approximation and Convergence". en. In: arXiv:1810.07900. arXiv:1810.07900 [cs]. DOI: 10.48550/arXiv.1810.07900. URL: http://arxiv.org/abs/1810.07900.

Bansal, Shubhi et al. (Oct. 2024). "A Comprehensive Survey of Mamba Architectures for Medical Image Analysis: Classification, Segmentation, Restoration and Beyond". en. In: arXiv:2410.02362. arXiv:2410.02362 [cs]. DOI: 10.48550/arXiv.2410.02362. URL: http://arxiv.org/abs/2410.02362.

Bar-David, Shmuel et al. (June 2023). "Decision S4: Efficient Sequence-Based RL via State Spaces Layers". en. In: arXiv:2306.05167. arXiv:2306.05167 [cs]. DOI: 10.48550/arXiv.2306.05167. URL: http://arxiv.org/abs/2306.05167.

Blelloch, Guy E. (Nov. 1990). *Prefix Sums and Their Applications*. Tech. rep. CMU-CS-90-190. School of Computer Science, Carnegie Mellon University.

Chen, Chang et al. (2022). "Transdreamer: Reinforcement learning with transformer world models". In: *arXiv preprint arXiv:2202.09481*.

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

Cho, Kyunghyun et al. (2014). "On the properties of neural machine translation: Encoder-decoder approaches". In: *arXiv preprint arXiv:1409.1259*.

Chung, Junyoung et al. (2014). "Empirical evaluation of gated recurrent neural networks on sequence modeling". In: *arXiv preprint arXiv:1412.3555*.

Dao, Tri and Albert Gu (2024). "Transformers are SSMs: generalized models and efficient algorithms through structured state space duality". In: *Proceedings of the 41st International Conference on Machine Learning*. ICML'24. Vienna, Austria: JMLR.org.

Duan, Yan et al. (Nov. 2016). "$RL^2$: Fast Reinforcement Learning via Slow Reinforcement Learning". en. In: arXiv:1611.02779. arXiv:1611.02779 [cs]. DOI: 10.48550/arXiv.1611.02779. URL: http://arxiv.org/abs/1611.02779.

Elman, Jeffrey L (1990). "Finding structure in time". In: *Cognitive science* 14.2, pp. 179–211.

Feng, Leo et al. (Oct. 2024). "Were RNNs All We Needed?" In: arXiv:2410.01201. arXiv:2410.01201. DOI: 10.48550/arXiv.2410.01201. URL: http://arxiv.org/abs/2410.01201.

Goel, Karan et al. (2022). *It's Raw! Audio Generation with State-Space Models*. arXiv: 2202.09729 [cs.SD]. URL: https://arxiv.org/abs/2202.09729.

Gu, Albert and Tri Dao (May 2024). "Mamba: Linear-Time Sequence Modeling with Selective State Spaces". In: arXiv:2312.00752. arXiv:2312.00752. DOI: 10.48550/arXiv.2312.00752. URL: http://arxiv.org/abs/2312.00752.

Gu, Albert, Karan Goel, and Christopher Ré (Aug. 2022). "Efficiently Modeling Long Sequences with Structured State Spaces". In: arXiv:2111.00396. arXiv:2111.00396. DOI: 10.48550/arXiv.2111.00396. URL: http://arxiv.org/abs/2111.00396.

Gu, Albert et al. (2020). "HiPPO: Recurrent Memory with Optimal Polynomial Projections". In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. Ed. by Hugo Larochelle et al. URL: https://proceedings.neurips.cc/paper/2020/hash/102f0bb6efb3a6128a3c750dd16729be-Abstract.html.

Gu, Albert et al. (Oct. 2021). "Combining Recurrent, Convolutional, and Continuous-time Models with Linear State-Space Layers". In: arXiv:2110.13985. arXiv:2110.13985. DOI: 10.48550/arXiv.2110.13985. URL: http://arxiv.org/abs/2110.13985.

Gu, Albert et al. (2022). "On the parameterization and initialization of diagonal state space models". In: *Advances in Neural Information Processing Systems* 35, pp. 35971–35983.

Gupta, Ankit, Albert Gu, and Jonathan Berant (2022). "Diagonal State Spaces are as Effective as Structured State Spaces". In: *Advances in Neural Information Processing Systems*. Ed. by S. Koyejo et al. Vol. 35. Curran Associates, Inc., pp. 2298222994. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/9156b0f6dfa9bbd18c79cc459ef5d61c-Paper-Conference.pdf.

Ha, David and Jürgen Schmidhuber (2018). "World models". In: *arXiv preprint arXiv:1803.10122*.

Hafner, Danijar et al. (2023). "Mastering diverse domains through world models". In: *arXiv preprint arXiv:2301.04104*.

Hausknecht, Matthew J and Peter Stone (2015). "Deep Recurrent Q-Learning for Partially Observable MDPs." In: *AAAI fall symposia*. Vol. 45, p. 141.

He, Kaiming et al. (June 2016). "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770778. DOI: 10.1109/CVPR.2016.90. URL: https://ieeexplore.ieee.org/document/7780459/.

Heess, Nicolas et al. (2015). "Memory-based control with recurrent neural networks". In: *arXiv preprint arXiv:1512.04455*.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long short-term memory". In: *Neural computation* 9.8, pp. 1735–1780.

Janner, Michael et al. (2022). *Planning with Diffusion for Flexible Behavior Synthesis*. arXiv: 2205.09991 [cs.LG]. URL: https://arxiv.org/abs/2205.09991.

Kidger, Patrick and Cristian Garcia (2021). "Equinox: neural networks in JAX via callable PyTrees and filtered transformations". In: *Differentiable Programming workshop at Neural Information Processing Systems 2021*.

Kim, Sanghyeon and Eunbyung Park (June 2023). "SMPConv: Self-Moving Point Representations for Continuous Convolution". en. In: *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Vancouver, BC, Canada: IEEE, pp. 1028910299. ISBN: 9798350301298. DOI: 10.1109/CVPR52729.2023.00992. URL: https://ieeexplore.ieee.org/document/10205071/.

Kingma, Diederik P. and Jimmy Ba (Jan. 2017). "Adam: A Method for Stochastic Optimization". en. In: arXiv:1412.6980. arXiv:1412.6980 [cs]. DOI: 10.48550/arXiv.1412.6980. URL: http://arxiv.org/abs/1412.6980.

Kirsch, Louis et al. (Jan. 2024). "General-Purpose In-Context Learning by Meta-Learning Transformers". en. In: arXiv:2212.04458. arXiv:2212.04458 [cs]. DOI: 10.48550/arXiv.2212.04458. URL: http://arxiv.org/abs/2212.04458.

Lange, Robert Tjarko (2022). *gymnax: A JAX-based Reinforcement Learning Environment Library*. Version 0.0.4. URL: http://github.com/RobertTLange/gymnax.

Lee, Kuang-Huei et al. (2022). "Multi-game decision transformers". In: *Advances in Neural Information Processing Systems* 35, pp. 27921–27936.

Lu, Chris et al. (2023). "Structured state space models for in-context reinforcement learning". In: *Advances in Neural Information Processing Systems* 36, pp. 47016–47031.

Moerland, Thomas M. et al. (2024). *EduGym: An Environment and Notebook Suite for Reinforcement Learning Education*. arXiv: 2311.10590 [cs.LG]. URL: https://arxiv.org/abs/2311.10590.

Morad, Steven et al. (2023). *POPGym: Benchmarking Partially Observable Reinforcement Learning*. arXiv: 2303.01859 [cs.LG]. URL: https://arxiv.org/abs/2303.01859.

Osband, Ian et al. (2020). "Behaviour Suite for Reinforcement Learning". In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=rygf-kSYwH.

Ota, Toshihiro (2024). "Decision mamba: Reinforcement learning via sequence modeling with selective state spaces". In: *arXiv preprint arXiv:2403.19925*.

Parisotto, Emilio and Ruslan Salakhutdinov (2021). *Efficient Transformers in Reinforcement Learning using Actor-Learner Distillation*. arXiv: 2104.01655 [cs.LG]. URL: https://arxiv.org/abs/2104.01655.

Raffin, Antonin et al. (2021). "Stable-Baselines3: Reliable Reinforcement Learning Implementations". In: *Journal of Machine Learning Research* 22.268, pp. 1–8. URL: http://jmlr.org/papers/v22/20-1364.html.

Romero, David W et al. (2021). "Ckconv: Continuous kernel convolution for sequential data". In: *arXiv preprint arXiv:2102.02611*.

Rumelhart, David E, Geoffrey E Hinton, Ronald J Williams, et al. (1985). *Learning internal representations by error propagation*.

Schulman, John et al. (2015). "Trust region policy optimization". In: *International conference on machine learning*. PMLR, pp. 1889–1897.

Schulman, John et al. (Aug. 2017). "Proximal Policy Optimization Algorithms". In: arXiv:1707.06347. arXiv:1707.06347 [cs]. DOI: 10.48550/arXiv.1707.06347. URL: http://arxiv.org/abs/1707.06347.

Smith, Jimmy T. H., Andrew Warrington, and Scott W. Linderman (Mar. 2023). "Simplified State Space Layers for Sequence Modeling". en. In: arXiv:2208.04933. arXiv:2208.04933 [cs]. DOI: 10.48550/arXiv.2208.04933. URL: http://arxiv.org/abs/2208.04933.

Sutton, Richard S, Andrew G Barto, et al. (2018). *Reinforcement learning: An introduction 2nd edition*. Vol. 1. 1. MIT press Cambridge.

Tay, Yi et al. (2021). "Long Range Arena : A Benchmark for Efficient Transformers". In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=qVyeW-grC2k.

Towers, Mark et al. (July 2024). *Gymnasium: A Standard Interface for Reinforcement Learning Environments*. DOI: 10.48550/arXiv.2407.17032.

Trentelman, Harry L et al. (2002). "Control theory for linear systems". In: *Appl. Mech. Rev.* 55.5.

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

Zhang, Guofeng, Tongwen Chen, and Xiang Chen (2007). "Performance Recovery in Digital Implementation of Analogue Systems". In: *SIAM Journal on Control and Optimization* 45.6, pp. 22072223. DOI: 10.1137/050643416.

# Appendix A

# Sequential MNIST

This section provides details on the implementation of the LSSL(f), LSSLf-diag and S5 SSM models for the use in a classification setting. All models have been trained on the sequential MNIST (sMNIST) dataset. The classic MNIST dataset consists of images of handwritten digits in a $28 \times 28$ pixel format. The sequential MNIST dataset flattens these images to a one-dimensional sequence 784 pixels long. Flattening the images from 2D to 1D removes the 2D inductive bias on that traditional convolutional neural networks (CNNs), which have classically been used to solve image recognition tasks, rely on. Instead the dataset is transformed to a purely sequential problem. Note that modern convolutional architectures, such as CKCONV (Romero et al., 2021) and SMPConv Kim and Park, 2023 can still (almost) fully solve this dataset. We will however see that SSM based architectures can achieve a similar performance with a fraction of the parameter count.

As mentioned in section 2.1.2, the full LSSL layer can be numerically unstable and can incur long runtimes due to the need to recompute the Krylov matrix each training step. We will thus focus on the architectures of the LSSLf-, the LSSLf-diag and S5-based models. The sections below contain model-specific implementation details and tables with the used training parameters. Note that the training batch size for each model was set to 50.

## A.1 Architectures

### A.1.1 LSSLf

The first architecture implemented was based on the original LSSL paper (Gu et al., 2021), using krylov functions as seen in equation 2.8. On a general level the architecture consists of a broadcasting layer, multiple residual LSSLf blocks, a mean pooling layer and linear decoders for the actor and critic heads. Given a minibatch of observations, the observations are broadcast to feature dimension $H$ with interleaving. As an example, given an 4 dimensional observation space, $u_t = (\alpha_t, \beta_t, \gamma_t, \delta_t)$ the broadcastcasting layer constructs the $L \times H$ matrix

$$\begin{pmatrix} \alpha_0 & \alpha_1 & \dots & \alpha_{L-1} \\ \beta_0 & \beta_1 & \dots & \beta_{L-1} \\ \gamma_0 & \gamma_1 & \dots & \gamma_{L-1} \\ \delta_0 & \delta_1 & \dots & \delta_{L-1} \\ \alpha_0 & \alpha_1 & \dots & \alpha_{L-1} \\ & & \vdots & \\ \alpha_0 & \alpha_1 & \dots & \alpha_{L-1} \\ \beta_0 & \beta_1 & \dots & \beta_{L-1} \\ \gamma_0 & \gamma_1 & \dots & \gamma_{L-1} \\ \delta_0 & \delta_1 & \dots & \delta_{L-1} \end{pmatrix}$$

Which will be fed to the SSMs in the residual block row-wise. Each residual block is instantiated with its own discretization parameter $\Delta t$. Initilization of these parameters, following prior works, are taken to be between 0.001 and 0.1 on a logaritmic scale. The output of the SSMs are computed in parallel. Non-linearity is applied by a ReLu activation function. Dropout is applied feature-wise before linear mixing of the SSM outputs. Following this we apply a residual connection and layer normalisation. See figure A.1 for a schematic overview. The output of these blocks is mean pooled along the sequence dimension and classified by a linear layer. See table A.1 for a full overview of the hyperparameters used during traning.

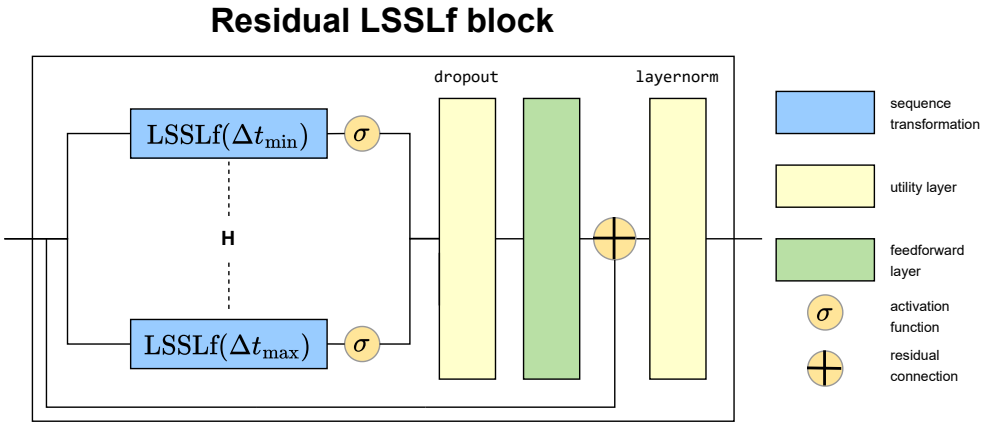



FIGURE A.1: Architecture of a residual LSSLf block.

### General Bilinear Transform

In contrast with later SSM architectures, LSSLs used General Bilinear Tranform (GBT) as their discretization method. The GBT method is a combination of the classic Euler method of integral approximation ($\alpha = 0$) and the backward Euler method ($\alpha = 1$)

$$\overline{\mathbf{A}} = (I - \alpha\Delta t \cdot \mathbf{A})^{-1}(I + (1-\alpha)\Delta t \cdot A)$$
$$\overline{\mathbf{B}} = \Delta t(I - \alpha\Delta t \cdot \mathbf{A})^{-1}\mathbf{B}$$

When $\alpha = \frac{1}{2}$, GBT preserves system stability, and can map unstable poles to stable poles (Zhang, Chen, and Chen, 2007).

### A.1.2 S5

The S5 based architecture changes the interleaved broadcasting layer used in the LSSL-based models to a linear encoding layer. This encoded observation sequence is fed to a sequence of residual blocks (He et al., 2016) consisting of an S5 SSM, a non-linear activation function (set to ReLu), a dropout layer, a residual connection and a (layer) normalization. Note that each block has its own S5 SSM with a separate hidden state. See figure A.2 for a schematic overview. The residual blocks can be chained indefinitely. The output of these blocks is mean pooled along the sequence dimension and classified by a linear layer. See table A.2 for a full overview of the hyperparameters used during traning.

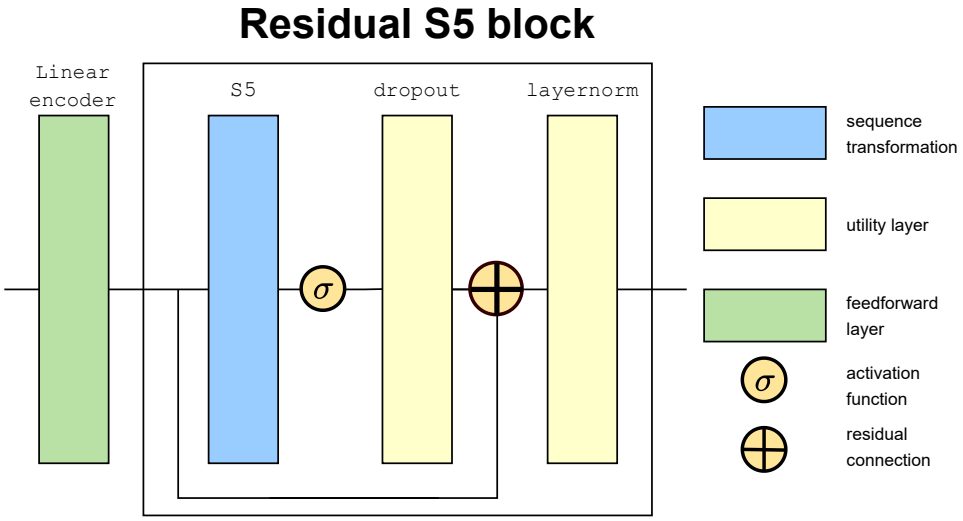



FIGURE A.2: Architecture of a residual S5 block. The first block requires a linear encoder to cast the $\mathbf{u} \in \mathbb{R}^{|\mathcal{O}|}$ input to $\mathbb{R}^H$. Subsequent blocks do not require an encoder. Non-linearity is introduced by the activation function.

### A.1.3 LSSLf-diag

The LSSLf-diag implementation combines the diagonalized SSM state matrix of S5 models with the parallel SISO kernels from the LSSLf implementation. The residual block architecture is similar to the one seen in A.1. However, LSSLf-diag uses parallel scan instead of Krylov functions to generalize over sequences of data. Due to the fact that LSSLf-diag employs SISO kernels, GPU memory is the main bottleneck of this architecture. Any scaling of the features dimension beyond 32 were unfeasible on the used machines.

## A.2 Hyperparameters

| Parameter | Value |
|---|---|
| Learning rate | 0.004 |
| Cosine Annealing | enabled |
| Epochs | 16 |
| Residual layers | 6 |
| N | 128 |
| H | 128 |
| logaritmic $\Delta t$ bounds | (-3, -1) |
| Feedthrough matrix | enabled |
| Discretization scheme | General Bilinear Transform |
| Dropout $p$ | 0.2 |
| Layer normalisation | enabled |

TABLE A.1: Hyperparameters for the training of LSSLf on the sequential MNIST dataset. Parameters were chosen to conform with the models used in (Gu et al., 2021)

| Parameter | Value |
|---|---|
| SSM learning rate | 0.002 |
| Learning rate | 0.008 |
| Cosine Annealing | enabled |
| Epochs | 150 |
| Residual layers | 4 |
| P | 128 |
| H | 96 |
| logaritmic $\Delta t$ bounds | (-3, -1) |
| Conjugate symmetry | enabled |
| Feedthrough matrix | enabled |
| Discretization scheme | Zero-order Hold |
| Dropout $p$ | 0.1 |
| Layer normalisation | enabled |

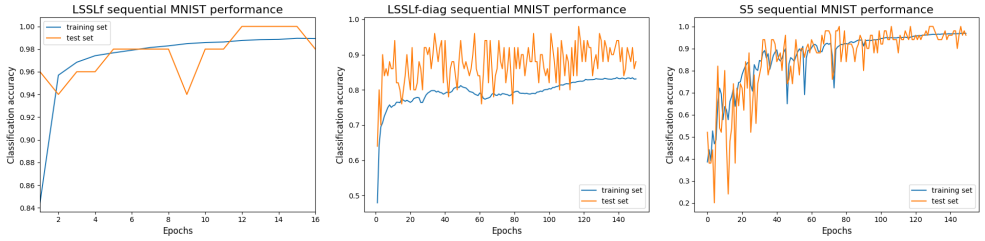
TABLE A.2: Hyperparameters for the training of S5 on the sequential MNIST dataset. Parameters were chosen to conform with the models used in (Smith, Warrington, and Linderman, 2023).

| **Parameter** | **Value** |
|---|---|
| SSM learning rate | 0.002 |
| Learning rate | 0.008 |
| Cosine Annealing | enabled |
| Epochs | 150 |
| Residual layers | 4 |
| N | 128 |
| H | 32 |
| logaritmic $\Delta t$ bounds | (-3, -1) |
| Conjugate symmetry | enabled |
| Feedthrough matrix | enabled |
| Discretization scheme | Zero-order Hold |
| Dropout $p$ | 0.1 |
| Layer normalisation | enabled |

TABLE A.3: Hyperparameters for the training of LSSLf-diag on the sequential MNIST dataset

## A.3 Results

Observe figure A.3. In this figure we can see that the LSSLf- and S5 model both achieve near perfect performance on the sMNIST dataset. The LSSLf-diag training performance stagnates around an 80 % accuracy score. Notably, the valuation score of the LSSLf-diag model is significantly higher than the achieved training scores. This is likely due to dropout, which is turned of during model testing, affecting the LSSLf-diag model more compared to the other two models due to its low feature dimension. Final (test) accuracy scores and model runtimes are displayed in table A.4 and are compared with current state of the art performance by feedforward architectures (Romero et al., 2021) (Kim and Park, 2023). One can note that while the S5 model slightly underperforms compared to the LSSLf model, the runtime is halved. Note also that only a single run was performed for each of the models. The difference in final testing accuracy between LSSLf and S5 might thus be negligible. Note also that the SSM based models can achieve a very decent performance with a significantly lower parameter count compared to their feedforward counterparts. The LSSLf-diag models parameters could not scale beyond the reported parameter count without issues in memory capacity. Even then, one can note the more than four-fold increase in runtime compared to the S5 model.




(A) LSSlf (B) LSSLf-diag (C) S5

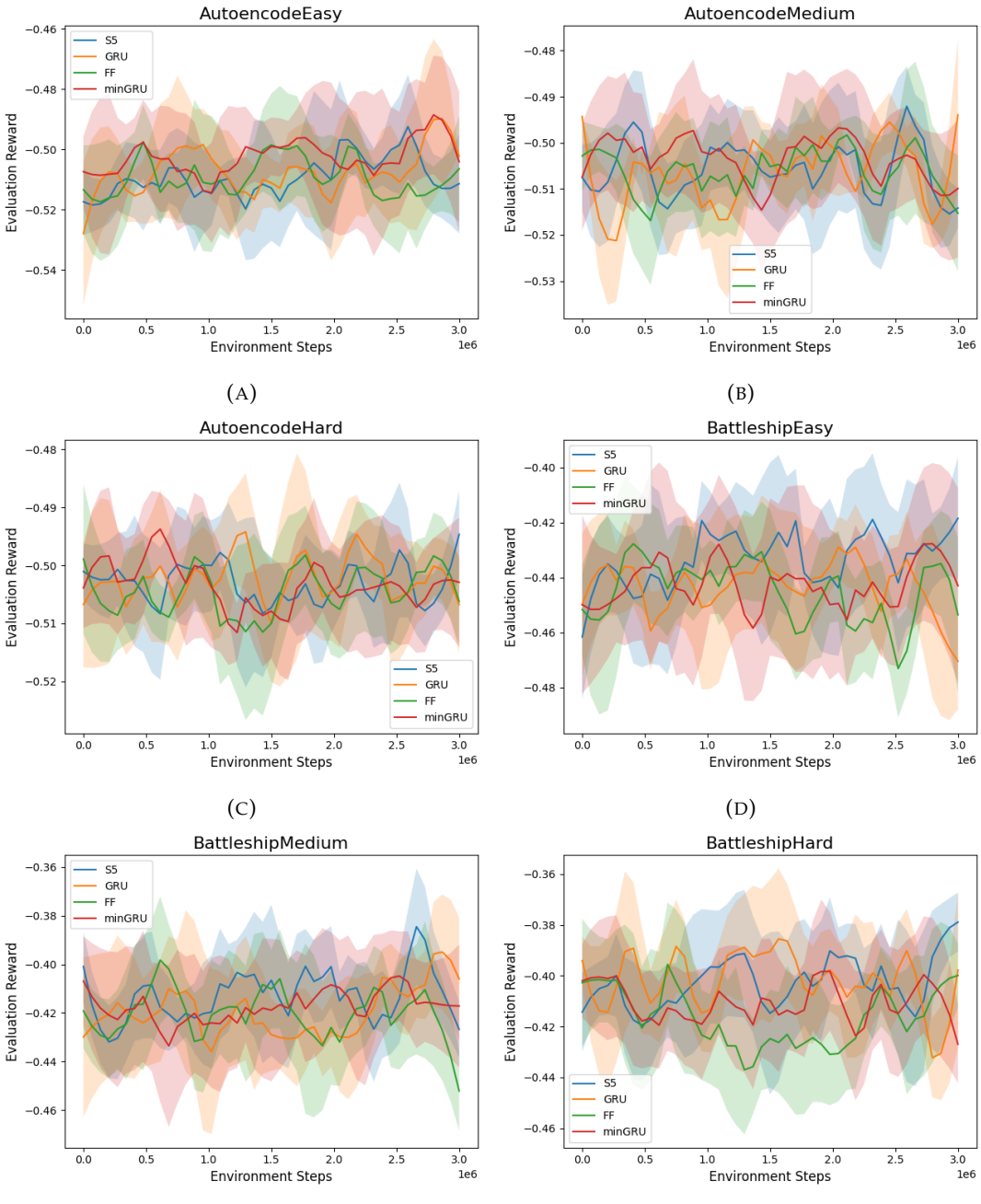
FIGURE A.3: SSM model training and valuation performance on the sMNIST dataset plotted as a function of epochs.

| **Model** | Parameter Count (↓) | **Test Accuracy** (↑) | **Runtime (H:M:S)** (↓) |
|---|---|---|---|
| LSSLf | 21322 | 0.98 | 3:34:12 |
| LSSLf-diag | **13130** | 0.90 | 9:15:31 |
| S5 | 26122 | 0.97 | **1:50:51** |
| CKCONV | 1M | 0.9932 | - |
| SMPConv | 373k | **0.9975** | - |

TABLE A.4: Model performance and runtime on the sMNIST dataset. Bold values are the best achieved results. Models below the dashed line represent the current state of the art.

# Appendix B

# Model Comparison on Individual Environments

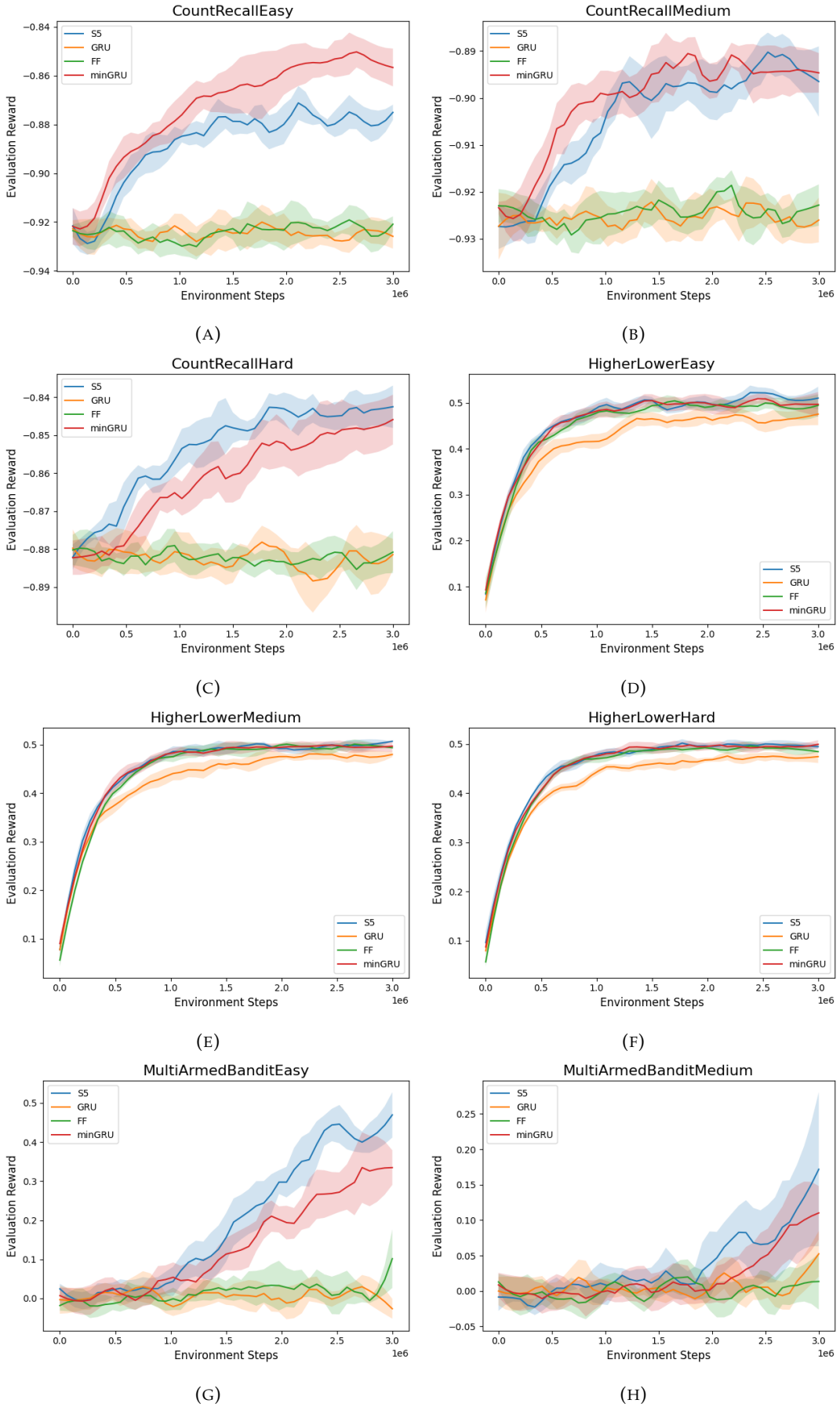



(A) (B)

(C) (D)

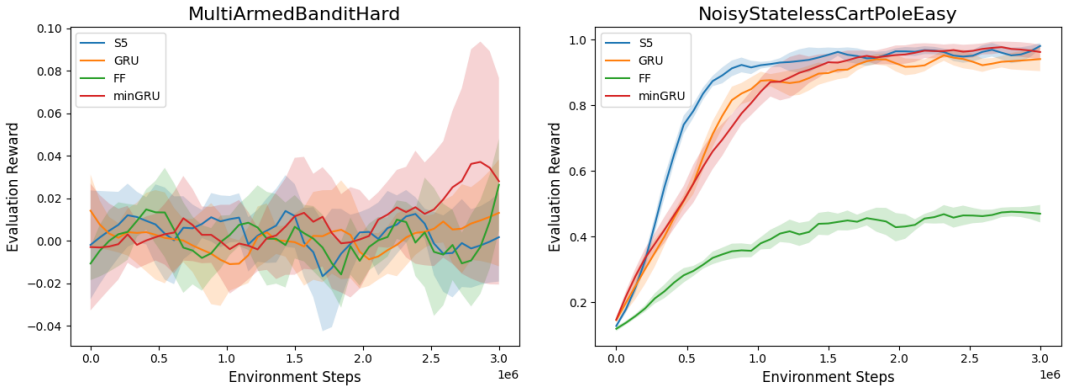
(E) (F)

FIGURE B.1: Results on the Auto Encode and Battleship environments




(A) (B) (C) (D) (E) (F) (G) (H)

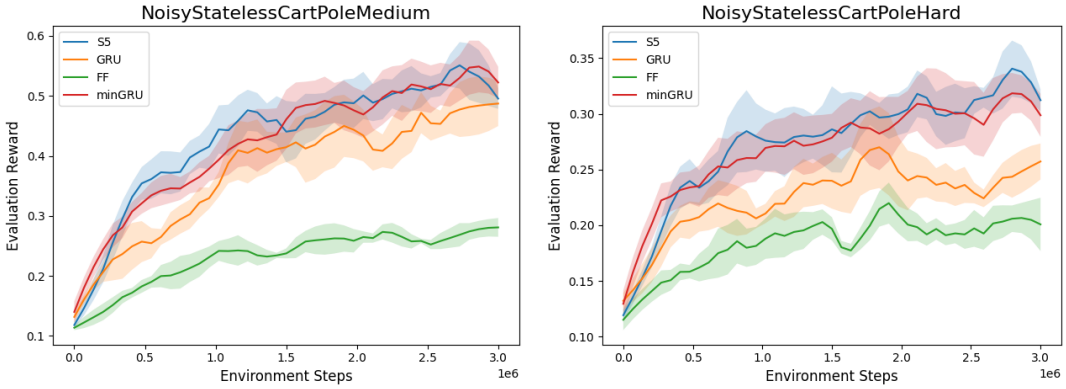
FIGURE B.2: Results on the Count Recall, Higher Lower and Multi Armed Bandit environments




(A)



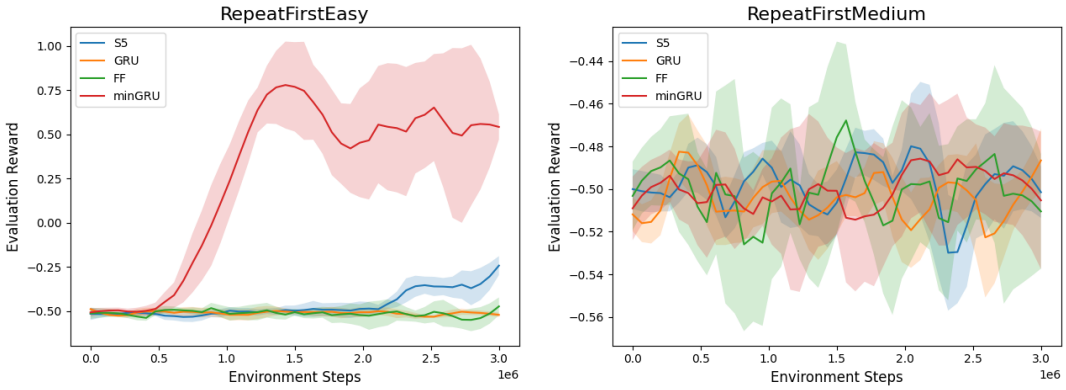

(B)




(C)



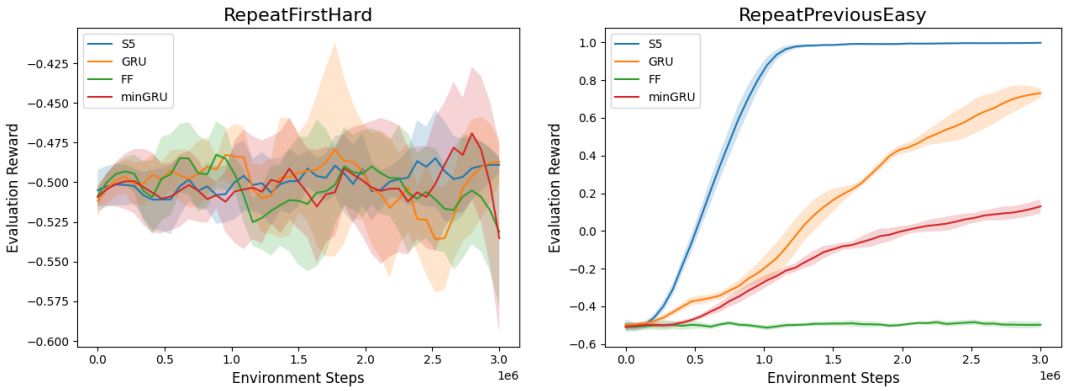

(D)




(E)




(F)

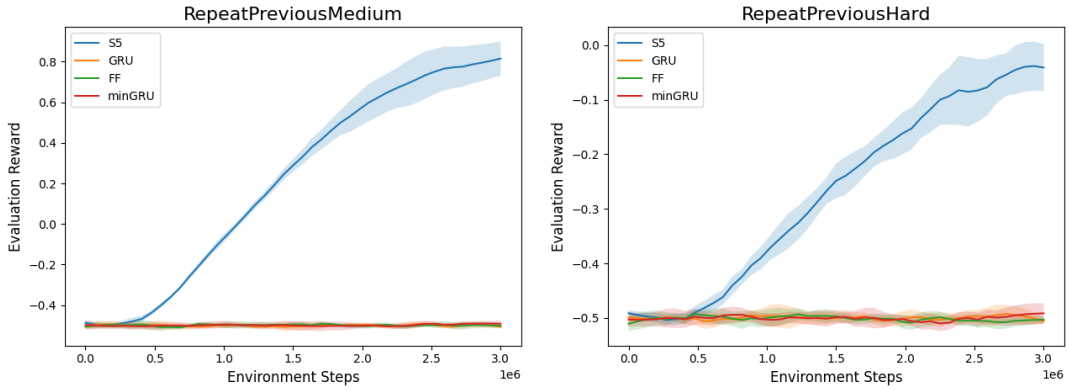



(G)




(H)

FIGURE B.3: Results on the Multi Armed Bandit, Noisy Stateless Cart-Pole, Repeat First and Repeat Previous environments.




(A) (B)

FIGURE B.4: Results on the Repreat Previous environments