Stability of deep State Space Models and their application to fuel consumption prediction in competitive racing

EÖTVÖS LORÁND UNIVERSITY

FACULTY OF SCIENCE

Author:

György Szabolcs Papp

Mathematics BSc, ELTE, TTK

III. grade

Supervisor:

Dániel Rácz

Research assistant, HUN-REN SZTAKI

PhD student, ELTE TTK

Budapest, 2025

Contents

Chapter 1

Introduction

Fuel consumption prediction is a well-known problem in scientific literature, becoming increasingly pronounced in recent years, when, due to environmental regulations and economic pressure, the imperative for enhanced fuel efficiency is on the rise. Since the transportation sector accounts for almost two thirds of all the oil used worldwide [1] and is responsible for 14% of global greenhouse gas emissions [2], accurately predicting a vehicle's fuel consumption is a necessity both for the consumers and the industry. It provides helpful data for optimizing vehicle design, aiding consumer purchase decisions, and shaping transportation policies. Accordingly, some of the most important areas taking advantage of accurate fuel consumption prediction are the transportation sector [3] and heavy-duty vehicles [4], both of which are well represented in the literature.

One of the areas that has a smaller significance factor, but utilizes fuel consumption prediction even more is competitive racing, where even slight inaccuracies can have significant ramifications for race strategy and overall performance. For instance, in Formula 1, consuming just one more deciliter of fuel than the predicted amount could result in the disqualification of the whole team for that event [5]. In endurance racing series, the time to refuel and the trade-off between carrying sufficient fuel to finish the race and minimizing vehicle weight for optimal speed is a constant strategic consideration. These topics are mostly researched by in house researchers of racing teams, but it is a popular topic in the academic literature as well [6]. Earlier literature addressed the problem using well-known tools from optimization and control theory, while more recent studies tend to exploit statistical models and machine learning.

This thesis focuses on a relatively unexplored area of study: forecasting total fuel consumption over a complete driving session using multivariate time series data measured

Figure 1.1: BME Motorsport Formula Student Racing Team's 2024 car

by various sensors of a racing car provided by the BME Motorsport Formula Student Racing Team. This is a team made up entirely of university students, including the main author of this thesis, competing in the international Formula Student Racing Series traveling to competitions all across Europe. The team members design, manufacture, test and race a combustion engine formula type racing car every year. The project is funded by various sponsors, university level supporters and the team members themselves. The car is completely built and engineered by the students, this includes the engine, the electrical telemetry system, the chassis, and all of the aerodynamic parts. Figure 1.1 shows a picture of the team's car built last year.

Based on the properties of the data available for us and motivated by the lack of work published in this area, our approach focuses on end-to-end prediction of total consumption for a given run or time segment, while the existing literature emphasizes instantaneous usage estimation. While it is not widely researched yet, in practice predicting total consumption of a given run is especially valuable for simulating realistic race conditions and strategic planning.

Motivated by both academic curiosity and the practical needs of the BME Motorsport Formula Student Racing Team, this thesis aims to explore and apply deep State-Space Models (SSMs) to fuel consumption prediction. Deep SSMs are hierarchical models, composed of building blocks of dynamical systems and neural networks. In particular, we investigate two recent SSM architectures, LRU (Linear Recurrent Unit) and Mamba, which have shown state-of-the-art performance on long-range sequence modeling tasks, but have not yet been applied to fuel consumption forecasting. To benchmark performance, we also compare these SSMs with a Transformer-based model of similar complexity, trained on the same dataset.

After systematically evaluating the models' performance on the dataset, our findings show that deep SSM architectures often outperform the considered Transformer model,

while requiring much less computational resources. Furthermore, the results suggest that the best performing models have a strong potential for practical applicability as they showed promising performance on unseen data.

Based on the promising results these stable parametrization models achieved, the question rises whether or not stability of the underlying dynamical systems can significantly impact model performance. After discussing the necessary mathematical prerequisites and introducing the model structures used in this thesis, we explore this question in more detail, as it is a prominent topic in the machine learning literature, with deep and interesting mathematical foundations.

Chapter 2

Related work

In this chapter we briefly summarize the academic literature on fuel consumption prediction focusing on its relevance, significance and similar use cases.

2.1 Relevance in competitive racing

Fuel consumption prediction of any kind can provide helpful data for consumers, racing drivers and the industry, therefore it is a hot topic in academic literature as well. Accordingly, it is best represented in the transportation sector [3], heavy-duty vehicles [4] and competitive racing [6].

Predicting fuel usage in competitive racing is a highly specialized and niche area of study, where even the smallest variables can significantly impact the outcome. According to an article published on the official NASCAR website [7], factors such as time of the day, temperature, driving style, track type, and condition, among numerous others all influence overall fuel consumption. This complexity results in the necessity for racing teams to engage in long practice sessions to collect as much data as possible. Various methods and tools are employed in competitive racing to optimize fuel strategy. It is common knowledge in the industry, that in practice, race fuel calculators are employed by race engineers to estimate the total fuel required for a run, based on factors such as race length, lap times, and fuel consumption rates. Top of the field racing teams tend to develop such tools but there are also publicly available versions, for example [8]. These tools assist teams in determining the minimum safe fuel load and establishing lap-by-lap targets for fuel consumption and energy usage, particularly in hybrid engines. Such optimization methods are the subject of extensive research both within racing teams and in academic settings. For

instance, [9] developed an optimization framework for hybrid-electric Formula 1 power units, aiming to minimize race time via optimal energy allocation strategies. Additionally, [10] proposed a stochastic lap strategy optimization algorithm that accounts for competitor behavior, enhancing energy management in hybrid race vehicles.

Top-tier racing teams rely heavily on real-time telemetry data and future consumption prediction in order to make real-time decisions based on the car's performance and the race situation. In the racing industry, it is folklore that in order to stay competitive, teams also must use a simulation software to simulate not just the vehicle dynamics of the car, but also its fuel consumption. There is a general consensus among professionals in the field that fuel consumption prediction algorithms are used to address this problem. However, the specific tools or models each team employs are considered trade secrets, in order to maintain their competitive edge.

2.2 Earlier literature

The earliest attempts to predict fuel consumption were firmly rooted in the principles of classical physics, using the driving resistance forces [11]. This approach involved developing mathematical models that accounted for factors such as aerodynamic drag, rolling resistance between the tires and the road, the force of inertia resisting changes in motion, and the impact of road gradient. The underlying idea is trying to understand the forces acting upon a moving vehicle and to quantify their effect and the energy it takes to overcome them. They also estimate the kinetic energy required for episodic accelerations, and based on these two factors they can get a prediction for the energy (and therefore fuel) consumption of the vehicle [12]. Another often used method of estimating fuel consumption that also roots in physics is via the usage of engine performance maps or fuel maps. These maps, often represented as two-dimensional lookup tables, detail the fuel consumption rate of an engine across a range of operating conditions, specifically engine speed and torque. The fuel map can be used to interpolate fuel consumption from torque and engine speed, also resulting in some form of fuel consumption prediction [13]. Despite the progress offered by analytical models and engine performance mapping, these early approaches faced significant limitations, particularly in their ability to accurately account for the dynamic and often unpredictable nature of real-world driving.

The application of control theory to the problem of fuel efficiency marked a significant step in moving beyond purely analytical predictions towards actively managing and

optimizing fuel consumption. Early control strategies involved relatively basic feedback mechanisms designed to maintain optimal engine operating parameters to ensure efficient combustion with the help of engine maps mentioned earlier. An important aspect of these studies is developing mathematical algorithms to optimize the engine output and fuel consumption of vehicles. For instance [14] and [15] both propose control methods based on mathematical models that automatically adjust the air-fuel ratio or the timing of ignition, in order to optimize efficiency and minimize consumption. Moving down the road of using automation to improve efficiency, the so-called look-ahead control systems were invented [16]. These systems utilized information about the road ahead, calculated predictions and then automatically adjusted the vehicle's speed and power delivery to minimize fuel consumption [17].

2.3 Machine learning approaches

While fuel consumption prediction is heavily researched, it can be very vague, ranging from ships and airplanes to racing cars, resulting in a wide spectrum of studies, each with their unique type of data and approach. Even if we focus on the academic literature regarding racing cars, every team tends to use its own in-house developed data collection mechanisms resulting in different types of datasets for every study, therefore no existing article perfectly matches the structure or content of our data. We work with multi-dimensional, long time series and aim to predict a scalar outcome. In contrast, much of the academic literature on fuel consumption either focuses on predicting consumption at a specific data point, or relies on more categorical data rather than continuous time series.

Recent papers on this topic tend to use statistical models and machine learning algorithms. Some of the most popular methods to predict instantaneous fuel consumption include regression models like random forest, linear regression, gradient boosting or neural networks. For example, according to [18], working on a similar problem the random forest (RF) technique produces a more accurate prediction compared to both gradient boosting and neural networks in their specific setup. In [19], SVM model is used on on-board sensor data to predict city bus fuel consumption resulting in a 0.95 R^2 value, where the R^2 is the coefficient of determination widely used in statistics. [20] also used SVM models to predict aircraft fuel consumption, showing promising results. Also, a Multi-Layer Perceptron (MLP) model achieves R^2 values around 0.94 in certain cases, according to [21]. Convolutional neural networks are also popular for this problem, as in [22] a one-

dimensional convolutional neural network is used to estimate fuel consumption of light-duty vehicles with an R^2 value of 0.99. In addition, papers [23] and [24] tackle related time-series problems, and they suggest the use of Long Short-Term Memory (LSTM) neural networks. Another article, working with CAN bus data very similar to ours, applies a transformer-based model — specifically, the Fast-Gated Attention (FGA) Transformer — and reports promising results [25]. For additional works considering different scenarios and problems, the survey paper [26] offers a comprehensive overview on fuel consumption prediction, showcasing different approaches such as the ones already mentioned or additional ones like RNN.

The usage of deep SSMs like the LRU or Mamba is not common in the academic literature, as we could not find a single article using any of the mentioned models in competitive racing, but there are some use cases for related problems in the transportation sector. Take for instance [27] which applies the deep SSM architecture S5 for fuel consumption prediction of marine vessels, and suggests the usage of the Mamba model class for further research.

Chapter 3

Preliminaries

In this chapter we, introduce the concept of a learning problem and discuss the mathematical background required to understand the deep SSM architectures used in this thesis.

3.1 Learning problem

In the following subchapter, we present the idea of a learning problem and the related mathematical and machine learning definitions, based on the framework and notation of [28].

Definition 3.1.1 ([28])**.** Let X be the set of inputs (in our case $X \subset \mathbb{R}^m$), and Y be the set of labels (typically $Y \subset \mathbb{R}$). A dataset is a set $S \subseteq X \times Y$ that is an i.i.d. sample drawn from an unknown probability distribution $\mathcal{D}$. In a **supervised learning problem** the goal is to find an $f : X \to Y$ function that describes the connection between X and Y (ideally with $f(\mathbf{x}) = y, \forall (\mathbf{x}, y) \in S$). If $Y = \mathbb{R}$ or an interval on $\mathbb{R}$ we call the problem regression and if $Y = \{1, 2, \ldots, k\}$ for some $k \in \mathbb{N}$ we call it classification.

We assume there exists a probability space $(X, \mathcal{B}(X), \mathbb{P}_{\mathcal{D}})$ over the Borel σ algebra generated by X. Unless stated otherwise the notations $\mathbb{P}_S, \mathbb{E}_S, \mathbb{E}_{(\mathbf{x},y)}$ are understood w.r.t. $\mathbb{P}_{\mathcal{D}}$. The distribution $\mathcal{D}$ is also understood over this probability space. Intuitively, $\mathbb{P}_S$ and $\mathbb{E}_S$ mean the probability and expectation over the choice of the random sample S.

In practice, the ideal representation $f(\mathbf{x}) = y$ for all $(\mathbf{x}, y) \in S$ does not necessarily exist, or it may not be possible to find it, so in order to train the model and compare the results we measure the success of a model with a loss function.

Definition 3.1.2. The **element-wise loss function**: $\ell : Y \times Y \to \mathbb{R}$ measures the discrepancy between two outputs or between an output and the corresponding label.

The *goal of the learning problem* we defined in Definition 3.1.1 is formally called the **learning objective**. The **learning objective** of a supervised learning problem is to find an $f : X \to Y$ function that minimizes the true error: $\mathcal{L}(f) := \mathbb{E}_{(\mathbf{x},y)}[\ell(f(\mathbf{x}),y)]$. Since in general $\mathcal{D}$ is unknown, in practice, instead we minimize the empirical error: $\mathcal{L}_S(f) := \frac{1}{|S|}\sum_{(\mathbf{x},y)\in S}\ell(f(\mathbf{x}),y)$. We call the difference of these two errors the **generalization error**: $\mathcal{L}(f) - \mathcal{L}_S(f)$.

Definition 3.1.3 ([28])**.** The largest gap between the true error and the empirical error of a function f is called the **representativeness of** S with respect to $\mathcal{F}$.

$$\mathrm{Rep}_{\mathcal{D}}(\mathcal{F},S) = \sup_{f\in\mathcal{F}}(\mathcal{L}(f) - \mathcal{L}_S(f))$$

In later chapters we mention model classes/architectures/families of models. By these we always mean the following definition.

Definition 3.1.4. A **model family** (or hypothesis class) $\mathcal{F}$ is a set of $f : X \to Y$ functions, which maps elements of the input set to elements of the labels set, as defined in Definition 3.1.1.

The learning objective basically means selecting the best $f \in \mathcal{F}$, the one that minimizes the empirical error.

In order to make sure that model families achieve great results on unseen data, we introduce the idea of generalization bounds. Intuitively a *generalization bound* is a theoretical upper bound on the generalization error $\mathcal{L}(f) - \mathcal{L}_S(f)$ (or its absolute value). In practice we often use PAC bounds, which are a special form of generalization bounds.

Definition 3.1.5 ([28])**.** The general shape of a **Probably Approximately Correct (PAC) generalization bound** for a hypothesis class $\mathcal{F}$ is the following:

$$\mathbb{P}_S\Big[\forall f \in \mathcal{F}:\ \mathcal{L}(f) - \mathcal{L}_S(f)\ \leq\ \varepsilon(N,\delta,\mathcal{F})\Big]\ \geq\ 1-\delta,$$

where $N = |S|$ is the number of i.i.d. training examples, $\delta \in (0,1)$ is the confidence parameter, and $\varepsilon(N,\delta,\mathcal{F})$ is a complexity term that decreases in N and δ, and increases in the "size" or capacity of $\mathcal{F}$.

In Chapter 4.4 we discuss a special case of this PAC bound definition introduced in [29].

Another metric measuring the "richness" or complexity of a function class (or model family) is called *Rademacher complexity*, with the usage of *Rademacher variables*.

Definition 3.1.6 ([28])**.** If for some σ_i variables it is true that $i \in [N]$ $\mathbb{P}(\sigma_i = 1) = \mathbb{P}(\sigma_i = -1) = 0.5$, then we call them **Rademacher variables**.

In Definition 3.1.3 we introduced the concept of representativeness. Although it appears to be a valuable metric, its practical computation requires access to the true loss, which poses a challenge since the underlying distribution $\mathcal{D}$ is typically unknown. To address this issue and estimate the representativeness of a sample S using only S, we introduce the metric of Rademacher complexity.

The intuitive motivation behind Rademacher complexity is partitioning the sample S into two disjoint subsets S_1 and S_2, each containing an equal number of elements, then estimating the representativeness of S by evaluating

$$\sup_{f \in \mathcal{F}} \left(\mathcal{L}_{S_1}(f) - \mathcal{L}_{S_2}(f) \right).$$

From this starting point, the formal definition of Rademacher complexity builds on the use of Rademacher variables and computes the expected value with respect to these variables. The formal definition is presented below.

Definition 3.1.7 ([28])**.** The **empirical Rademacher complexity** of an $A \subset \mathbb{R}^N$ set of vectors is

$$R(A) = \frac{1}{N} \mathbb{E}_\sigma \left[\sup_{\mathbf{a} \in A} \sum_{i=1}^{N} \sigma_i a_i \right] \tag{3.1}$$

where a_i is the i-th coordinate of the $\mathbf{a}$ vector

A special form of this for a function class $\mathcal{F}$ with respect to an $S = \{(\mathbf{x}_1, y_1), (\mathbf{x_2}, y_2), \ldots, (\mathbf{x}_N, y_N)\}$ dataset drawn from the probability distribution $\mathcal{D}$ is when $A = \{(f(\mathbf{x}_1), f(\mathbf{x}_2), \ldots, f(\mathbf{x}_N)) \mid f \in \mathcal{F}\}$.

$$R_S(\mathcal{F}) = \frac{1}{N} \mathbb{E}_\sigma \left[\sup_{f \in \mathcal{F}} \sum_{i=1}^{N} \sigma_i f(\mathbf{x}_i) \right] \tag{3.2}$$

where $\mathbb{E}_\sigma$ means expected value over σ Rademacher variables.

We can bound the *representativeness* of S by the *Rademacher complexity* with the following lemma.

Lemma 3.1.1 ([28]).

$$\mathbb{E}_S[\mathrm{Rep}_{\mathcal{D}}(\mathcal{F}, S)] \leq 2 \cdot \mathbb{E}_S[R_S(\mathcal{F})] \tag{3.3}$$

Proof [28]. Let $S \neq S' = \{(\mathbf{x}'_1, y'_1), (\mathbf{x}'_2, y'_2), \ldots, (\mathbf{x}'_N, y'_N)\}$ be an i.i.d. sample drawn from $\mathcal{D}$. Since $\mathcal{L}(f)$ is the true loss, it is true that for all $f \in \mathcal{F} : \mathcal{L}(f) = \mathbb{E}_{S'}[\mathcal{L}_{S'}(f)]$, from this we obtain the following for the generalization error.

$$\mathcal{L}(f) - \mathcal{L}_S(f) = \mathbb{E}_{S'}[\mathcal{L}_{S'}(f)] - \mathcal{L}_S(f) = \mathbb{E}_{S'}[\mathcal{L}_{S'}(f) - \mathcal{L}_S(f)]. \tag{3.4}$$

After taking supremum over $f \in \mathcal{F}$ and utilizing the fact that the expected value of the supremum is an overestimation of the supremum of the expected value we obtain the following

$$\sup_{f \in \mathcal{F}} (\mathcal{L}(f) - \mathcal{L}_S(f)) \leq \mathbb{E}_{S'} \left[\sup_{f \in \mathcal{F}} (\mathcal{L}_{S'}(f) - \mathcal{L}_S(f)) \right] \tag{3.5}$$

Now taking expected value over S on both sides and applying the definition of empirical loss we obtain

$$\mathbb{E}_S \left[\sup_{f \in \mathcal{F}} (\mathcal{L}(f) - \mathcal{L}_S(f)) \right] \leq \frac{1}{N} \mathbb{E}_{S,S'} \left[\sup_{f \in \mathcal{F}} \sum_{i=1}^{N} (f(\mathbf{x}'_i) - f(\mathbf{x}_i)) \right] \tag{3.6}$$

Now, we use that $\mathbf{x}_j$ and $\mathbf{x}'_j$ are i.i.d. variables, and the definition of Rademacher variables (denoted σ_i) to note that

$$\begin{aligned}
\mathbb{E}_{S,S'} \left[\sup_{f \in \mathcal{F}} \sum_{i=1}^{N} (f(\mathbf{x}'_i) - f(\mathbf{x}_i)) \right] &= \mathbb{E}_{S,S'} \left[\sup_{f \in \mathcal{F}} \left((f(\mathbf{x}'_j) - f(\mathbf{x}_j)) + \sum_{i \neq j} (f(\mathbf{x}'_i) - f(\mathbf{x}_i)) \right) \right] = \\
&= \mathbb{E}_{S,S'} \left[\sup_{f \in \mathcal{F}} \left((f(\mathbf{x}_j) - f(\mathbf{x}'_j)) + \sum_{i \neq j} (f(\mathbf{x}'_i) - f(\mathbf{x}_i)) \right) \right] = \\
&= \mathbb{E}_{S,S'} \left[\sup_{f \in \mathcal{F}} \left(\sigma_j (f(\mathbf{x}_j) - f(\mathbf{x}'_j)) + \sum_{i \neq j} (f(\mathbf{x}'_i) - f(\mathbf{x}_i)) \right) \right]
\end{aligned}$$

Repeating the last step for all j we obtain

$$\mathbb{E}_{S,S'} \left[\sup_{f \in \mathcal{F}} \sum_{i=1}^{N} (f(\mathbf{x}'_i) - f(\mathbf{x}_i)) \right] = \mathbb{E}_{S,S',\sigma} \left[\sup_{f \in \mathcal{F}} \sum_{i=1}^{N} \sigma_i (f(\mathbf{x}'_i) - f(\mathbf{x}_i)) \right] \tag{3.7}$$

Now back to Equation (3.6) using Equation (3.7), the definition of representativeness and the properties of sup and σ we get

$$\begin{aligned}\mathbb{E}_S\left[\mathrm{Rep}_{\mathcal{D}}(\mathcal{F},S)\right] &\leq \mathbb{E}_{S,S',\sigma}\left[\sup_{f\in\mathcal{F}}\sum_{i=1}^{N}\sigma_i(f(\mathbf{x}_i')-f(\mathbf{x}_i))\right] \\ &\leq \mathbb{E}_{S,S',\sigma}\left[\sup_{f\in\mathcal{F}}\sum_{i=1}^{N}\sigma_i f(\mathbf{x}_i') + \sup_{f\in\mathcal{F}}\sum_{i=1}^{N}\sigma_i f(\mathbf{x}_i)\right]\end{aligned} \tag{3.8}$$

Finally we just use the definition of Rademacher complexity and after simplification and combination we obtain (3.3). □

While measuring the Rademacher complexity, of our function class $\mathcal{F}$ can be useful, the intuition of the definition is to use it for a new set of functions, denoted $\ell \circ \mathcal{F} = \{g_\ell : (\mathbf{x},y) \to \ell(f(\mathbf{x}),y) \mid f \in \mathcal{F}\}$. With the help of our previous lemma and *Rademacher complexity* we can get a PAC bound for the generalization error with $\ell \circ \mathcal{F}$ as the following theorem states.

Theorem 3.1.2 ([28])**.** Let $S = \{(\mathbf{x}_1,y_1),(\mathbf{x}_2,y_2),\ldots,(\mathbf{x}_N,y_N)\}$ be an i.i.d. sample drawn from the probability distribution $\mathcal{D}$. Assume that there exists a $c \in \mathbb{R}$, that for all $(\mathbf{x}_i,y_i) \in S$ and $f \in \mathcal{F}$ it is true that $|\ell(f(\mathbf{x}_i),y_i)| \leq c$. Then, with probability at least $1-\delta$, for all $f \in \mathcal{F}$:

$$\mathcal{L}(f) - \mathcal{L}_S(f) \leq 2 \cdot R_S(\ell \circ \mathcal{F}) + 4c\sqrt{\frac{2\ln(4/\delta)}{N}}$$

Proof sketch. Let us assume that the following inequality is true if our previous assumptions hold.

$$\mathcal{L}(f) - \mathcal{L}_S(f) \leq 2\,\mathbb{E}_{S'} R_{S'}(\ell \circ \mathcal{F}) + c\sqrt{\frac{2\ln(2/\delta)}{N}}. \tag{3.9}$$

Now with utilizing the following property of the union: $\mathcal{D}(A \cup B) \leq \mathcal{D}(A) + \mathcal{D}(B)$ after using McDiarmid's Inequality (Lemma 26.4 from [28]) and Inequality (3.9), we obtain

$$\mathcal{L}(f) - \mathcal{L}_S(f) \leq 2 \cdot R_S(\ell \circ \mathcal{F}) + 4c\sqrt{\frac{2\ln(4/\delta)}{N}}$$

The proof of Inequality (3.9) is a consequence of McDiarmid's Inequality, Lemma 3.3 and Definition 3.1.3. For the full proof and McDiarmid's Inequality see [28]. □

3.2 Dynamical systems

The two model families used in this work are built on dynamical systems, more precisely linear dynamical systems, so, in the following subchapter, we outline the related

mathematical definitions and theorems.

3.2.1 Introduction into dynamical systems

In this section, we adopt the notation and closely follow the definitions and theoretical framework presented in [30]. Full proofs and further details can be found in the original source.

Consider the general form of an ordinary autonomous differential equation, where $\mathbf{x} : \mathbb{R} \to \mathbb{R}^n$ is the unknown function and $f : \mathbb{R}^n \to \mathbb{R}^n$ is a given continuously differentiable function.

$$\mathbf{x}'(t) = f(\mathbf{x}(t))$$

This form of differential equations are called autonomous, because f only depends explicitly on $\mathbf{x}$ and not on the time. If a differential equation is given of the form

$$\mathbf{x}'(t) = f(t, \mathbf{x}(t))$$

it is a non-autonomous differential equation. Dynamical systems are based on autonomous differential equations. Dynamical systems can be defined in either continuous-time or discrete-time. The definition of the former is as follows.

Definition 3.2.1 ([30])**.** A $\varphi : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ continuously differentiable function is called a continuous-time dynamical system if it fulfills the following two conditions:

- $\forall p \in \mathbb{R}^n : \varphi(0, p) = p$
- $\forall p \in \mathbb{R}^n$ and $t, s \in \mathbb{R}$, it must be true that: $\varphi(t, \varphi(s, p)) = \varphi(t + s, p)$

A continuous-time dynamical system can be understood as a model of a deterministic process, where $\varphi(t, p)$ represents the state of the system after time t, starting from state p. If in 3.2.1 we limit the time dimension to the set of whole numbers instead of real numbers and we drop the requirement of differentiability for the function we get the discrete-time definition.

Definition 3.2.2 ([30])**.** A $\varphi : \mathbb{Z} \times \mathbb{R}^n \to \mathbb{R}^n$ continuous function is called a discrete-time dynamical system if it fulfills the following two conditions:

- $\forall p \in \mathbb{R}^n, \varphi(0, p) = p$

- $\forall p \in \mathbb{R}^n$ and $k, m \in \mathbb{Z}$ it must be true that: $\varphi(k, \varphi(m, p)) = \varphi(k+m, p)$

We often call a continuous-time dynamical system a flow, and a discrete-time dynamical system a mapping. Many times we just give the time set as $\mathbb{T}$ which could be $\mathbb{R}$, $\mathbb{Z}$ or other mathematical rings or fields.

3.2.2 Stability in dynamical systems

In this section, we explore some basic definitions and results on stability, following the notation and theoretical exposition of [31] and [32]. Detailed proofs and further discussions can be found in the original sources. Before we introduce the definition of stability, we need to define different types of norms. First we define the norm of a vector.

Definition 3.2.3 ([31])**.** The **norm** $\|\mathbf{x}\|$ of a vector $\mathbf{x}$ is a real valued function with the following properties:

(i) $\|\mathbf{x}\| \geq 0$ *with* $\|\mathbf{x}\| = 0$ *if and only if* $\mathbf{x} = 0$.

(ii) $\|\alpha\mathbf{x}\| = |\alpha| \, \|\mathbf{x}\|$ *for any scalar* α.

(iii) $\|\mathbf{x}+\mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$ *(triangle inequality).*

Intuitively, the norm of a vector can be interpreted as the length of the vector and $\|\mathbf{x}-\mathbf{y}\|$ is the distance between two vectors.

Now we can define the induced norm of a matrix as follows.

Definition 3.2.4 ([31])**.** *Let* $\|\cdot\|$ *be a given vector norm. For each matrix* $\mathbf{A} \in \mathbb{R}^{m\times n}$*, the quantity* $\|\mathbf{A}\|$ *is called the* **induced norm** *of* $\mathbf{A}$ *corresponding to the vector norm* $\|\cdot\|$ *and is defined by*

$$\|\mathbf{A}\| := \sup_{\substack{\mathbf{x}\neq 0 \\ \mathbf{x}\in\mathbb{R}^n}} \frac{\|\mathbf{A}\mathbf{x}\|}{\|\mathbf{x}\|} = \sup_{\|\mathbf{x}\|\leq 1} \|\mathbf{A}\mathbf{x}\| = \sup_{\|\mathbf{x}\|=1} \|\mathbf{A}\mathbf{x}\|$$

Intuitively, the induced norm of a matrix corresponding to a given vector norm represents the maximum stretching effect the matrix has on any vector in the given norm. There are also norms of matrices which are not induced by any vector, these are just called matrix or operator norms. Some of the most common vector and operator norms are summarized in Table 3.1.

We can also define corresponding norms for functions of time.

Table 3.1: **Commonly used norms**

Norms on $\mathbb{R}^n$	**Matrix norms on $\mathbb{R}^{m\times n}$**
$\|\mathbf{x}\|_\infty = \max_i \|\mathbf{x}_i\|$ (infinity norm)	$\|\mathbf{A}\|_F = \sqrt{\sum_i \sum_j \|a_{i,j}\|^2}$ (Frobenius norm)
$\|\mathbf{x}\|_1 = \sum_i \|\mathbf{x}_i\|$	$\|\mathbf{A}\|_1 = \max_j \sum_i \|a_{ij}\|$ (column sum)
$\|\mathbf{x}\|_2 = \left(\sum_i \|\mathbf{x}_i\|^2\right)^{1/2}$ (Euclidean norm)	$\|\mathbf{A}\|_2 = \sqrt{\lambda_{\max}\left(\mathbf{A}^T\mathbf{A}\right)}$ (where $\lambda_{\max}(M)$ is the biggest absolute value eigenvalue of M)

Definition 3.2.5 ([31])**.** We define the $\mathcal{L}_p(\mathbb{R}^m)$ norm for an $f : \mathbb{R} \to \mathbb{R}^m$ function of time as

$$\|f\|_{\mathcal{L}_p(\mathbb{R}^m)} = \left(\int_0^\infty \|f(\tau)\|_2^p d\tau\right)^{1/p}$$

for $p \in [1,\infty)$ and as a special case $\|f\|_{\mathcal{L}_\infty(\mathbb{R}^m)} = \sup_{t\geq 0} \|f(t)\|_\infty$. We also define $\mathcal{L}_p(\mathbb{R}^m) = \{f : \mathbb{R} \to \mathbb{R}^m \mid \|f\|_{\mathcal{L}_\infty(\mathbb{R}^m)} < \infty\}$.

The corresponding discrete time version can be defined as follows.

Definition 3.2.6. The ℓ_p norm of an $\mathbf{x} : \mathbb{N} \to \mathbb{R}^m$ function is

$$\|\mathbf{x}\|_{\ell_p(\mathbb{R}^m)} = \left(\sum_{k\in\mathbb{N}} \|\mathbf{x}(k)\|_2^p\right)^{\frac{1}{p}}$$

for $p \in [1,\infty)$ and as a special case $\|\mathbf{x}\|_{\ell_\infty(\mathbb{R}^m)} = \sup_{k\in\mathbb{N}} \|\mathbf{x}(k)\|_\infty$. We also define $\ell_p(\mathbb{R}^m) = \{\mathbf{x} : \mathbb{N} \to \mathbb{R}^m \mid \|\mathbf{x}\|_{\ell_p(\mathbb{R}^m)} < \infty\}$.

Now that we defined basic norms we can discuss the stability of a dynamical system. Consider the following linear autonomous system:

$$\mathbf{x}'(t) = \mathbf{A}\mathbf{x}(t), \quad \text{where } \mathbf{A} \in \mathbb{R}^{n\times n} \tag{3.10}$$

Definition 3.2.7 ([32])**.** A linear system shaped like (3.10) is called **stable** if all solution trajectories $\mathbf{x}$ are bounded for positive time: $\mathbf{x}(t)$ for $t > 0$ are bounded.

Definition 3.2.8 ([32])**.** Similarly a linear system shaped like (3.10) is called **asymptotically stable** if all solution trajectories go to zero as time tends to infinity: $\mathbf{x}(t) \to 0$ as $t \to \infty$.

In practice we check stability in a different way, using the following theorem:

Theorem 3.2.1 ([32])**.** The autonomous dynamical system (3.10) is

- **asymptotically stable** if and only if all eigenvalues of **A** have negative real parts. An *A* matrix like this is called *Hurwitz*,
- **stable** if and only if all eigenvalues of **A** have non-positive real parts, and, in addition, all pure imaginary eigenvalues have multiplicity of one.

Proof will be provided for the discrete time version only, discussed later in the chapter.

Now we take a look at the definition of Ljapunov stability [31]. For this we shift our attention to dynamical systems described by ordinary differential equations:

$$\mathbf{x}'(t) = f(t, \mathbf{x}(t)), \quad \mathbf{x}(t_0) = x_0 \tag{3.11}$$

where $\mathbf{x} \in \mathbb{R}^n$, $f : [t_0, \infty) \times \mathcal{B}(r) \mapsto \mathbb{R}$, and $\mathcal{B}(r) = \{\mathbf{x} \in \mathbb{R}^n \mid \|\mathbf{x}\| < r\}$. We assume that f is of such nature that for every $x_0 \in \mathcal{B}(r)$ and every $t_0 \in \mathbb{R}^+$, (3.11) possesses one and only one solution $\mathbf{x}(t; t_0, x_0)$.

In order to define Ljapunov stability we need to know what an equilibrium state of a dynamical system is:

Definition 3.2.9 ([31])**.** A state $\mathbf{x}_e \in \mathbb{R}^n$ is said to be an equilibrium state of the system described by (3.11) if:

$$f(t, \mathbf{x}_e) = 0 \quad \forall \quad t \geq t_0$$

Definition 3.2.10 ([31])**.** An equilibrium state $\mathbf{x}_e$ defined in Definition 3.2.9 is said to be **Ljapunov stable** if for arbitrary t_0 and $\varepsilon > 0$ there exists a $\delta(\varepsilon, t_0)$ such that $\|x_0 - \mathbf{x}_e\| < \delta(\varepsilon, t_0)$ implies $\|\mathbf{x}(t; t_0, x_0) - \mathbf{x}_e\| < \varepsilon$ for all $t \geq t_0$.

Definition 3.2.11 ([31])**.** An equilibrium state $\mathbf{x}_e$ is said to be **uniformly stable (u.s.)** if it is stable and if $\delta(\varepsilon, t_0)$ in Definition 3.2.10 does not depend on t_0.

Similarly general asymptotical stability in Definition 3.2.8 can be defined for an equilibrium state as follows:

Definition 3.2.12 ([31])**.** An equilibrium state $\mathbf{x}_e$ is said to be **asymptotically stable** if it is stable and there exists a $\delta(t_0)$ such that $\|x_0 - \mathbf{x}_e\| < \delta(t_0)$ implies $\lim_{t \to \infty} \|\mathbf{x}(t; t_0, x_0) - \mathbf{x}_e\| = 0$.

The models used in practice generally use stable parametrization, but the effects of stability on model performance are heavily researched. The models used in this study are based on stable discrete time systems, hence now we turn our attention to a special case of discrete-time dynamical systems, defined in Definition 3.2.2. The discrete-time version of the linear system stated in (3.10) is as follows:

$$\mathbf{x}(t+1) = \mathbf{A}\mathbf{x}(t), \quad \mathbf{A} \in \mathbb{R}^{n\times n} \tag{3.12}$$

A special form of dynamical systems are called input-output systems, more specifically discrete-time linear time-invariant (LTI) systems, where the system Σ can be identified by its parameters, namely by the tuple of matrices $(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D})$ and

$$\Sigma: \quad \mathbf{x}(k+1) = \mathbf{A}\mathbf{x}(k) + \mathbf{B}\mathbf{u}(k), \quad \mathbf{y}(k) = \mathbf{C}\mathbf{x}(k) + \mathbf{D}\mathbf{u}(k) \tag{3.13}$$

The analogue continuous-time version:

$$\Sigma: \quad \mathbf{x}'(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t), \quad \mathbf{y}(t) = \mathbf{C}\mathbf{x}(t) + \mathbf{D}\mathbf{u}(t) \tag{3.14}$$

where $\mathbf{A} \in \mathbb{R}^{n\times n}$, $\mathbf{B} \in \mathbb{R}^{n\times m}$, $\mathbf{C} \in \mathbb{R}^{p\times n}$, $\mathbf{D} \in \mathbb{R}^{p\times m}$, while n is the state, m is the input and p is the output dimension. From now on $\mathbf{x}(0) = 0$ everywhere, unless explicitly stated otherwise.

LTI systems also have *input-output (convolution)* representation [32]: $\mathbf{y} = h * \mathbf{u}$, where $h : \mathbb{R} \to \mathbb{R}^{p\times m}$ is the *impulse response* which, for discrete-time systems, takes the following shape.

$$h(k) = \begin{cases} \mathbf{C}\mathbf{A}^{k-1}\mathbf{B}, & k > 0, \\ \mathbf{D}, & k = 0, \\ \mathbf{0}, & k < 0. \end{cases} \tag{3.15}$$

The integral of the impulse response is called the *transfer function*, and can be defined as follows.

$$H(k) = \mathbf{C}(k\mathbf{I} - \mathbf{A})^{-1}\mathbf{B} + \mathbf{D}, \tag{3.16}$$

Intuitively this also means that LTI systems can map every input sequence $\mathbf{u}(1), \ldots, \mathbf{u}(T)$ to the output sequence $\mathbf{y}(1), \ldots, \mathbf{y}(T)$, where T is the length of the time series, with a linear operator we call the input-output map.

Definition 3.2.13 ([29])**.** Let $\mathbf{u} \in \mathcal{U}$ and $\mathbf{y} \in \mathcal{Y}$, where $\mathcal{U}$ and $\mathcal{Y}$ are Banach spaces. For

a Σ LTI system there exists a $S_{\Sigma,T} : \mathcal{U} \to \mathcal{Y}$ linear map called the *input-output map* of Σ, such that $S_{\Sigma,T}(\mathbf{u})(t) = \mathbf{y}(t)$ for all $1 \leq t \leq T$.

We can also define a type of stability for these input-output systems:

Definition 3.2.14 ([32])**.** An LTI system defined in equation (3.13) is called **internally stable** if $\mathbf{u}(k) = 0 \; \forall k > 0$ implies $\mathbf{x}(k) \to 0$, for $k \to \infty \quad \forall \mathbf{x}(0)$.

Definition 3.2.15 ([32])**.** A discrete-time dynamical system defined in definition 3.2.2 is called:

- **Stable** if the absolute value of the eigenvalues of $\mathbf{A}$ are not greater than 1 and any eigenvalue on the unit circle must have a multiplicity of 1: $|\lambda(\mathbf{A})| \leq 1$
- **Asymptotically stable** if the absolute values of all the eigenvalues of $\mathbf{A}$ are smaller than 1: $|\lambda(\mathbf{A})| < 1$ which means the matrix is Schur.

Theorem 3.2.2. Internal stability of an LTI system defined in Definition 3.2.14 is equivalent to the underlying $\mathbf{A}$ matrix being Schur.

Proof. We are proving the following equivalency.

An LTI system is internally stable $\iff$ the corresponding $\mathbf{A}$ matrix is Schur.

$\implies$:

Internal stability means that if $\mathbf{u}(k) = 0 \; \forall k > 0$, then $\mathbf{x}(k) \to 0$, for $k \to \infty, \forall \mathbf{x}(0)$. If $\mathbf{u}(k) = 0 \; \forall k > 0$ then $\mathbf{x}(k+1) = \mathbf{A}\mathbf{x}(k)$. In this case $\mathbf{x}(k) \to 0$ only holds if $\lim_{k\to\infty} \mathbf{A}^k \mathbf{x}(0) = 0, \forall \mathbf{x}(0)$ which means that all eigenvalues of $\mathbf{A}$ are strictly less than 1, otherwise there exists an initial condition along the corresponding eigenvector such that $\mathbf{x}(t)$ does not converge to 0.

$\impliedby$:

If we know that $\mathbf{A}$ is Schur then all of its eigenvalues are strictly less than 1. Now let us indirectly assume that $\mathbf{u}(k) = 0 \; \forall k > 0$ does not imply $\mathbf{x}(k) \to 0$, for $k \to \infty, \forall \mathbf{x}(0)$. In this case $\mathbf{x}(k+1) = \mathbf{A}\mathbf{x}(k)$ but $\lim_{k\to\infty} \mathbf{A}^k \mathbf{x}(0) \neq 0, \forall \mathbf{x}(0)$ which contradicts $\mathbf{A}$ being Schur. □

There are other types of stability, like ℓ_p input-output stability, defined as follows.

Definition 3.2.16 ([32])**.** An LTI system defined in Equation (3.13) is called ℓ_p **input-output stable** if any ℓ_p bounded input $\mathbf{u} \in \ell_p(\mathbb{R})$ results in an ℓ_p bounded output $\mathbf{y} \in \ell_p(\mathbb{R})$

$$\mathbf{u} \in \ell_p(\mathbb{R}^m) \implies \mathbf{y} \in \ell_p(\mathbb{R}^p)$$

In the literature, the term stability usually refers to the ℓ_p input-output stability for some p. The case of $p = \infty$ is sometimes called bounded-input, bounded-output (BIBO) stability. This intuitively means that perturbing the input of the system results in only a small difference in the output.

Other than stability norms can also be defined for systems. One of the most important ones is the $\mathcal{H}_2$ norm:

Definition 3.2.17 ([32])**.** The $\mathcal{H}_2$ norm of a Σ system is defined as follows

$$\|\Sigma\|_{\mathcal{H}_2} = \sqrt{\sum_{k=0}^{\infty} \|h(k)\|_F^2}$$

For an LTI Σ system, defined in (3.13) the $\mathcal{H}_2$ norm takes the following shape [29].

$$\|\Sigma\|_{\mathcal{H}_2} := \sqrt{\|\mathbf{D}\|_F^2 + \sum_{k=0}^{\infty} \left\|\mathbf{C}\mathbf{A}^k\mathbf{B}\right\|_F^2} \tag{3.17}$$

Intuitively the $\mathcal{H}_2$ norm of a system is the maximum amplitude of the output which results from finite energy input signals.

Another useful system norm is the ℓ_1 norm [33], which takes the following form for discrete-time LTI systems [29].

$$\|\Sigma\|_{\ell_1} := \max_{1 \le i \le p} \left[\|\mathbf{D}_i\|_1 + \sum_{k=0}^{\infty} \left\|\mathbf{C}_i\mathbf{A}^k\mathbf{B}\right\|_1 \right] \tag{3.18}$$

where $\mathbf{D}_i$ and $\mathbf{C}_i$ denotes the i-th row of the corresponding matrix.

Theorem 3.2.3 ([32])**.** The internal stability of a Σ LTI system implies the BIBO stability of the system.

Proof. If an LTI system is internally stable that implies that the corresponding $\mathbf{A}$ matrix is Schur as proven before. BIBO stability means that

$$\mathbf{u} \in \ell_\infty(\mathbb{R}^m) \implies \mathbf{y} \in \ell_\infty(\mathbb{R}^p)$$

Let us assume then that A is Schur and $\mathbf{u} \in \ell_\infty(\mathbb{R}^m) \implies \|\mathbf{u}\|_\infty \le M < \infty$. If from here we obtain that $\|\mathbf{y}(k)\|_\infty < \infty$, the proof is completed. We know that $\mathbf{y} = h * \mathbf{u}$ and from here

we get $y(k) = \sum_{i=0}^{k} h(i)u(k-i)$. Now we take the $\|\cdot\|_\infty$ of $\mathbf{y}$ and use the definition of h.

$$\begin{aligned}
\|\mathbf{y}(k)\|_\infty &\leq \sum_{i=0}^{k} \|h(i)\|_\infty \|\mathbf{u}(k-i)\|_\infty \\
&\leq \sum_{i=0}^{k} \left\|\mathbf{C}\mathbf{A}^i\mathbf{B}\right\|_\infty \|\mathbf{u}(k-i)\|_\infty \\
&\leq M \|\mathbf{C}\|_\infty \|\mathbf{B}\|_\infty \sum_{i=0}^{k} \left\|\mathbf{A}^i\right\|_\infty \\
&\leq M \|\mathbf{C}\|_\infty \|\mathbf{B}\|_\infty \sum_{i=0}^{\infty} \left\|\mathbf{A}^i\right\|_\infty
\end{aligned} \tag{3.19}$$

According to Gelfand's formula $\rho(\mathbf{A}) = \lim_{k\to 0} \left\|\mathbf{A}^\mathbf{k}\right\|^{1/k}$ for any $\|\cdot\|$ matrix norm, where $\rho(\mathbf{A})$ is the *spectral radius* of $\mathbf{A}$. We know that $\mathbf{A}$ is Schur, which means that $\rho(\mathbf{A}) < 1$, hence there exists an α such that $\rho(\mathbf{A}) < \alpha < 1$. This implies that $\exists N : \forall k' > N$ it is true that $\left\|\mathbf{A}^{k'}\right\|^{1/k'} < \alpha < 1$. After taking power of k' we obtain $\left\|\mathbf{A}^{k'}\right\| < \alpha^{k'} < 1$. Let K be the maximum of the first N values of the sequence $K := \{\|\mathbf{A}\|, \left\|\mathbf{A}^2\right\|^{1/2}, \ldots \left\|\mathbf{A}^N\right\|^{1/N}\}$. From here we note that

$$\sum_{i=0}^{\infty} \left\|\mathbf{A}^i\right\| = \sum_{i=0}^{N} \left\|\mathbf{A}^i\right\| + \sum_{i=N+1}^{\infty} \left\|\mathbf{A}^i\right\| \leq NK + \sum_{i=N+1}^{\infty} \alpha^i \leq NK + \frac{1}{1-\alpha} < \infty \tag{3.20}$$

concluding the proof. □

Theorem 3.2.4 ([32])**.** For a discrete time LTI system Σ given in Equation (3.13), internal stability implies $\|\Sigma\|_{\ell_1} < \infty$ and $\|\Sigma\|_{\mathcal{H}_2} < \infty$.

Proof. The proof follows from an analogue calculation to Equation (3.19) of the previous proof as Gelfand's formula holds for any matrix norm, thus for the Frobenius and the 1-norm as well. □

Theorem 3.2.5 ([32])**.** If $\mathbf{A}$ is Schur, then for all $\mathbf{Q} \in \mathbb{R}^{n\times n}$ there exists a unique $\mathbf{P} \in \mathbb{R}^{n\times n}$ that satisfies:

$$\mathbf{A}^T\mathbf{P}\mathbf{A} + \mathbf{P} = \mathbf{Q} \tag{3.21}$$

See the corresponding references in [32] for the proof.

We remark that the converse of the last two theorems also hold under additional, mild assumption. This means that for LTI systems, the various definitions capturing stability are roughly equivalent. Note that for nonlinear systems, this is not the case in general.

Chapter 4

Deep SSM models

In this chapter we show a general overview of SSMs, present the architectures used in this thesis and discuss the question of stability and the PAC bound introduced in [29] following their framework and notation.

4.1 Deep SSM architectures and the question of stability

A State-Space Model (SSM) is a discrete-time linear dynamical system of the form defined in equation (3.13) where $\mathbf{x}(t), \mathbf{y}(t)$ and $\mathbf{u}(t)$ are the state, output and input signals respectively, with the initial state: $\mathbf{x}(0) = 0$. For a visual representation see Figure 4.1. In the academic literature, the acronym SSM sometimes can refer to the LTI system, to the Recurrent Neural Network (RNN) defined below, or even to the whole architecture but in this work it refers to the discrete-time linear dynamical system stated in equation (3.13). For the models used in this study we will focus on stable SSMs (Definition 3.2.7) for which $\mathbf{A}$ is Schur as defined in Definition 3.2.15. Intuitively, an *SSM block* takes an input $\mathbf{u}$, feeds it through the SSM (LTI system), applies the same non-linear, time invariant $g : \mathbb{R}^p \to \mathbb{R}^m$ function at every timestamp to the output $\mathbf{y}$ and then adds the input $\mathbf{u}$ to the result to complete the residual connection. More formally an SSM block maps the $\mathbf{u}(t), 1 \leq t \leq T, t \in \mathbb{Z}$ input sequence to the $f^{DTB}(\mathbf{u})(t) = g(S_{\Sigma,T}(\mathbf{u})(t)) + \alpha\mathbf{u}(t)$, where $\alpha \in \mathbb{R}$ is the residual weight and $S_{\Sigma,T}$ is the input-output map defined in Definition 3.2.13. Please note that here $\mathbf{u}$ refers to the input of the SSM block, and $\mathbf{y}$ refers to the output of the LTI system, not the deep SSM architecture. For a visual representation see Figure 4.3.

An artificial neural network is a model of computation inspired by the structure of neural networks in the brain. These neural networks can be understood as a directed graph,

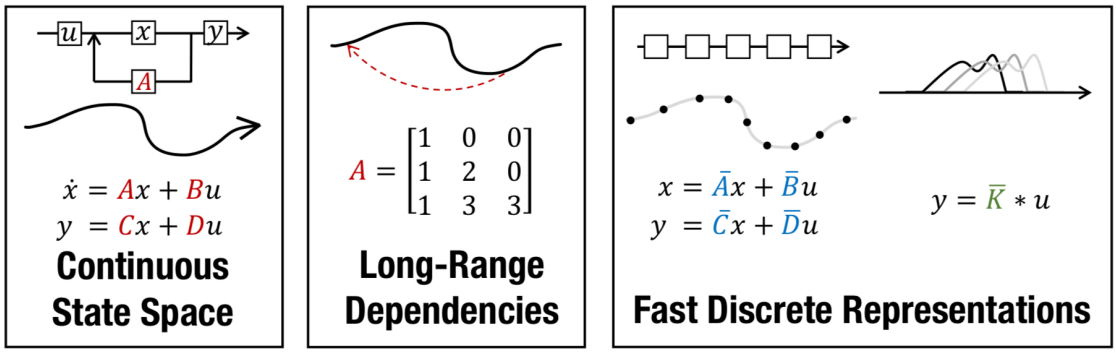


Figure 4.1: **(Left)** State Space Models (SSMs), defined by matrices A, B, C, D, transform an input sequence $u(t)$ into an output sequence $y(t)$ through a latent state $x(t)$. **(Center)** Recent advances in continuous-time memory modeling have identified specific forms of the A matrix that enable SSMs to effectively model long-range dependencies both theoretically and in practice. **(Right)** These models can be implemented either using a recurrence formulation (left) or through convolution (right). However, realizing these perspectives computationally often involves switching between distinct parameterizations (e.g., red, blue, green), which can be computationally costly. Figure and description as in [34]

where the nodes represent the neurons and the edges are the connections between them. In the network each neuron (or node) receives the weighted sum of the outputs of the nodes, which are linked to their incoming edges. For a visual representation see Figure 4.2 If the graph does not contain cycles the network is called a *feed-forward neural network*, formally defined as follows.

Definition 4.1.1 ([28])**. A feed-forward neural network** or **Multilayer perceptron (MLP)** is a function $f : \mathbb{R}^P \times \mathbb{R}^m \to \mathbb{R}^p$ s.t.

$$f(\theta, \mathbf{x}) = \mathbf{W}_L \sigma(\mathbf{W}_{L-1} \dots \sigma(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) \cdots + \mathbf{b}_{L-1}) + \mathbf{b}_L$$

where $\theta = [\mathbf{W}_1, \mathbf{b}_1, \dots, \mathbf{W}_L, \mathbf{b}_L]$, L is the number of layers with dimensions $n_1, \dots, n_L$, m is the input dimension, p is the output dimension and the dimension of θ is $P = n \cdot n_1 + n_1 + n_1 n_2 + n_2 + \cdots + n_L \cdot p$.

Some model families displayed in this work use an MLP layer. An MLP layer is a feed-forward neural network applied to the input time series at every time step.

A *Recurrent Neural Network (RNN) block* [36] operates similarly to an SSM block in that it maps an input sequence $\mathbf{u}(t) \in \mathbb{R}^m$ to an output sequence $\mathbf{y}(t) \in \mathbb{R}^p$ by maintaining a hidden state $\mathbf{x}(t) \in \mathbb{R}^n$ at each time step t. The update rules for the hidden and output states are given by the recurrence:

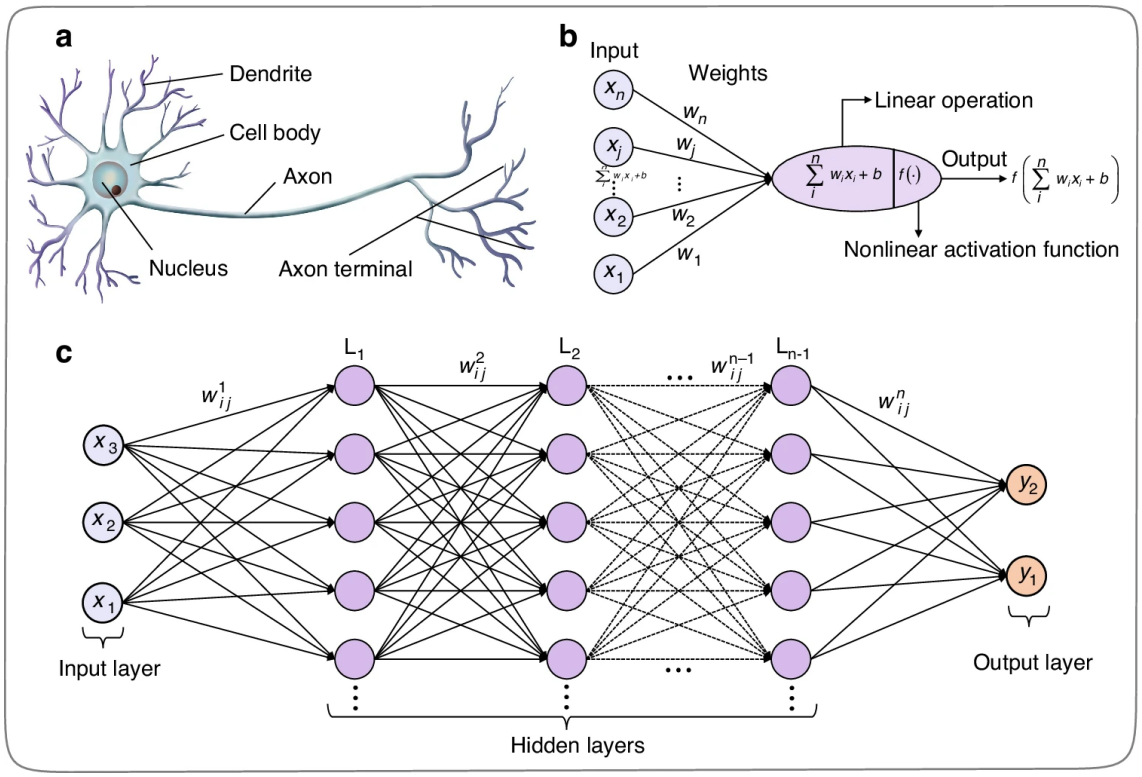


Figure 4.2: **a** Structure of biological neurons.**b** Mathematical process of artificial neurons in multi-layer perceptron. **c** Multi-layer perceptron feed-forward neural network. Figure and caption as in [35]

$$\begin{aligned} \mathbf{x}(k+1) &= \sigma(\mathbf{A}\mathbf{x}(k) + \mathbf{B}\mathbf{u}(k)), \\ \mathbf{y}(k) &= \mathbf{C}\mathbf{x}(k) + \mathbf{D}\mathbf{u}(k) \end{aligned} \tag{4.1}$$

where $\mathbf{A} \in \mathbb{R}^{n\times n}, \mathbf{B} \in \mathbb{R}^{n\times m}, \mathbf{C} \in \mathbb{R}^{p\times n}$, and $\mathbf{D} \in \mathbb{R}^{p\times m}$ are learnable parameters. Note that the second equation defining $\mathbf{y}(k)$ can also be non-linear. The initial hidden state is set to $\mathbf{x}(0) = 0$. The function σ is a non-linear activation function. If σ is the identity function, the RNN block is said to be *linear*. Note that RNN with a linear activation function results in an LTI system, stated in equation (3.13).

Definition 4.1.2 ([29])**.** In general a *deep SSM model* is a composition of an encoder, several SSM blocks, a time-pooling layer and a decoder.

$$f = f^{Dec} \circ f^{Pool} \circ f^{B_L} \circ \cdots \circ f^{B_1} \circ f^{Enc},$$

where $\circ$ represents the composition of functions, f^{B_i} is the input-output map of the i-th SSM block, f^{Dec} and f^{Enc} are linear, time invariant functions applied to the sequence at every timestamp, and f^{Pool} in our case is average pooling that averages over time, intuitively resulting in the elimination of time.

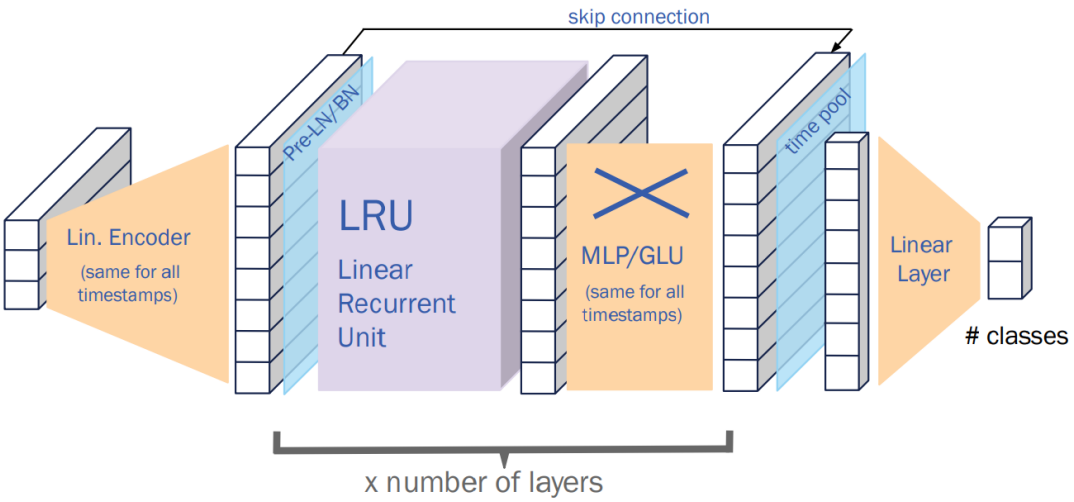


Figure 4.3: Structure of the LRU architecture as in [36]

The first architecture that showed that deep SSMs can address long-range dependencies well was the *Linear State Space Layer (LSSL)* introduced in [37], however in practice it was unusable because of memory requirements. This was improved in the paper [34], which introduced the *Structured State Space (S4)* architecture that is a special form of our definition of deep SSMs stated in Definition 4.1.2. This paper used a special parametrization of the $\mathbf{A}$ matrix and defined the g function as an activation function σ to reduce memory usage, resulting in a model class showing promising results in practice. They used a parametrization called *Normal Plus Low-Rank* that relied on the HiPPO Matrices introduced in [37]. The most important matrix in this class is called the *HiPPO matrix* and is defined as follows.

$$\textbf{(HiPPO Matrix)} \qquad \mathbf{A}_{n,k} = -\begin{cases} \sqrt{(2n+1)(2k+1)} & \text{if } n > k, \\ n+1 & \text{if } n = k, \\ 0 & \text{if } n < k. \end{cases} \tag{4.2}$$

The layers between the SSM blocks are often MLP or GLU layers. The first one defined earlier and the latter one as follows.

Definition 4.1.3 ([29])**.** A *Gated Linear Unit (GLU)* [38] layer is a function parameterized by a matrix W such that

$$f(\mathbf{u})[k] = \mathrm{GELU}(\mathbf{u}[k]) \odot \sigma\left(W \cdot \mathrm{GELU}(\mathbf{u}[k])\right),$$

where σ is the sigmoid function and GELU is the Gaussian Error Linear Unit [39].

4.2 The LRU architecture

The *LRU (Linear Recurrent Unit)* architecture, introduced by [36], is a special form of our deep SSM Definition 4.1.2 where the g function is a neural network, also using additional normalizing layers between the SSM blocks and a special exponential parametrization of the $\mathbf{A}$ matrix as follows:

$$\lambda_j = \underbrace{\exp\left(-\exp\left(\nu_j\right)\right)}_{\text{magnitude}} + i \underbrace{\exp\left(\theta_j\right)}_{\text{phase}},$$

where ν_j and θ_y are learnable parameters, and λ_j is the j-th element of the diagonal $\mathbf{A}$ matrix. As a result of this parametrization, $\mathbf{A}$ is Schur which results in internal stability as proven earlier.

The LRU model, similarly to the S4, also uses special type of matrices. The paper [36] uses several other tricks and enhancements to improve model performance. For a visual representation see Figure 4.3 and for a more detailed explanation see the original paper. The key point of the LRU paper is that if you train a time-mixing, non-linear model, you can do it by separating the time-mixing layer and the non-linearity. Basically this means that a time-mixing, linear architecture combined with a time invariant non-linear function can learn a time-mixing, non-linear model well.

In conclusion, according to [36] the LRU model outperforms SSMs of similar complexity, such as the S4, while using even less resources, hence the reason we use this structure for our thesis. For more information on the LRU architecture, see [36].

4.3 The Mamba model family

In order to understand the Mamba architecture from [40] we first introduce the concept of discretization. The discretization of a continuous LTI system defined by Equation (3.14) yields the following system.

$$\mathbf{x}(k+1) = \overline{\mathbf{A}}\mathbf{x}(k) + \overline{\mathbf{B}}\mathbf{u}(k), \quad \mathbf{y}(k) = \overline{\mathbf{C}}\mathbf{x}(k) + \overline{\mathbf{D}}\mathbf{u}(k) \tag{4.3}$$

Where $\overline{\mathbf{A}}$, $\overline{\mathbf{B}}$, $\overline{\mathbf{C}}$ and $\overline{\mathbf{D}}$ are the discretized transition matrices. For example the S4 or LRU architecture use the *zero-order hold (ZOH)* discretization. As defined in [27], this means that the system samples the latent state based on an input and maintains this value until the next input is received, which then updates the latent state. As a result, the system

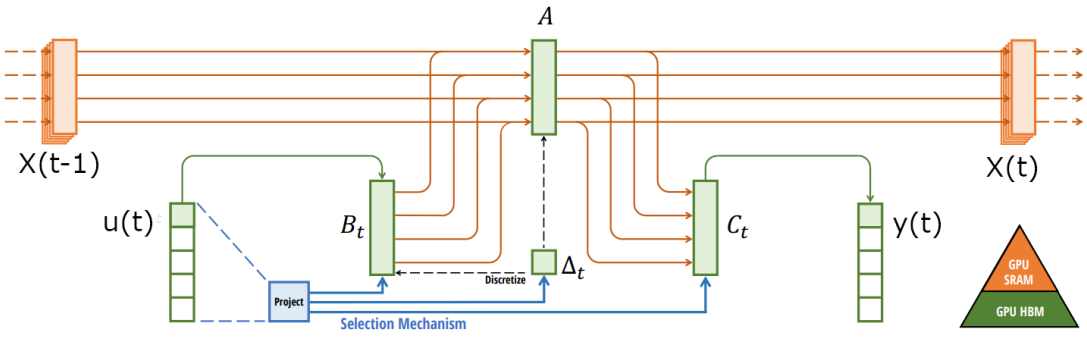


Figure 4.4: The usage of the selection mechanism in Mamba, figure as in [40]

produces continuous output and undergoes an exact discretization within the time domain of the state-space. The transition matrices for this discretization with step size $\Delta \in \mathbb{R}$ are given by:

$$\overline{\mathbf{A}} = e^{\mathbf{A}\Delta}, \quad \overline{\mathbf{B}} = \mathbf{A}^{-1}(\overline{\mathbf{A}} - \mathbf{I})\mathbf{B}, \quad \overline{\mathbf{C}} = \mathbf{C}, \quad \overline{\mathbf{D}} = \mathbf{D} \tag{4.4}$$

In case of a non-invertible $\mathbf{A}$, the input matrix instead takes the form

$$\overline{\mathbf{B}} = \left(\int_{\tau=0}^{\Delta t} e^{\tau \mathbf{A}} \, \mathrm{d}\tau \right) \mathbf{B}.$$

Probably the most important difference between the Mamba and a regular deep SSM is the discretization method the Mamba uses. The idea is that the Mamba architecture uses a discretization $\Delta(\mathbf{u}(t), t)$ that depends on the input series $\mathbf{u}(\mathbf{t})$ and the current timestamp t. This yields the following discretization of the continuous LTI system from 3.14.

$$\begin{aligned} \mathbf{x}(k+1) &= \mathbf{A}(\Delta(\mathbf{u}(k),k)\mathbf{x}(k) + \mathbf{B}(\Delta(\mathbf{u}(k),k)\mathbf{u}(k), \\ \mathbf{y}(k) &= \mathbf{C}(\Delta(\mathbf{u}(k),k)\mathbf{x}(k) + \mathbf{D}\mathbf{u}(k) \end{aligned} \tag{4.5}$$

This is done using the help of a selection mechanism. These are usually learnable linear transformations. In essence, the selection mechanism acts as a dynamic gate that decides how much of the past state to retain and how much new input to incorporate. For the details of the selection mechanism used in the Mamba model class see [40]. For a visual representation of this see Figure 4.4 and for the overall structure of the Mamba see Figure 4.5.

We mentioned earlier that we use models with stable parametrization in this thesis. For the LRU model class it is easy to show that for the discrete-time version, but with the special discretization method used in the Mamba it is not that evident. With the Mamba model we need the original continuous-time system to be stable. If that holds then the

discretized version can be written as a discrete-time LPV system, defined in [41], however the discussion of LPV systems and their stability is out of the scope of this thesis.

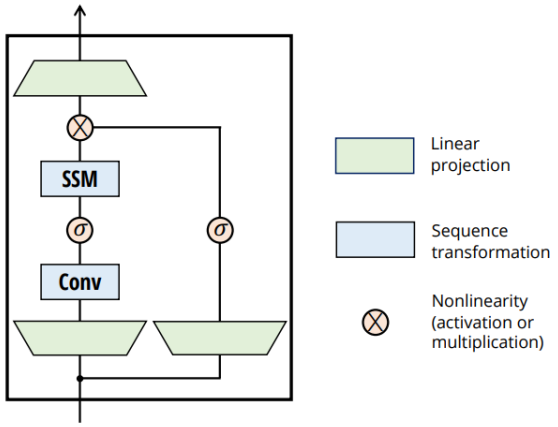


Figure 4.5: The structure of the Mamba architecture as in [40]

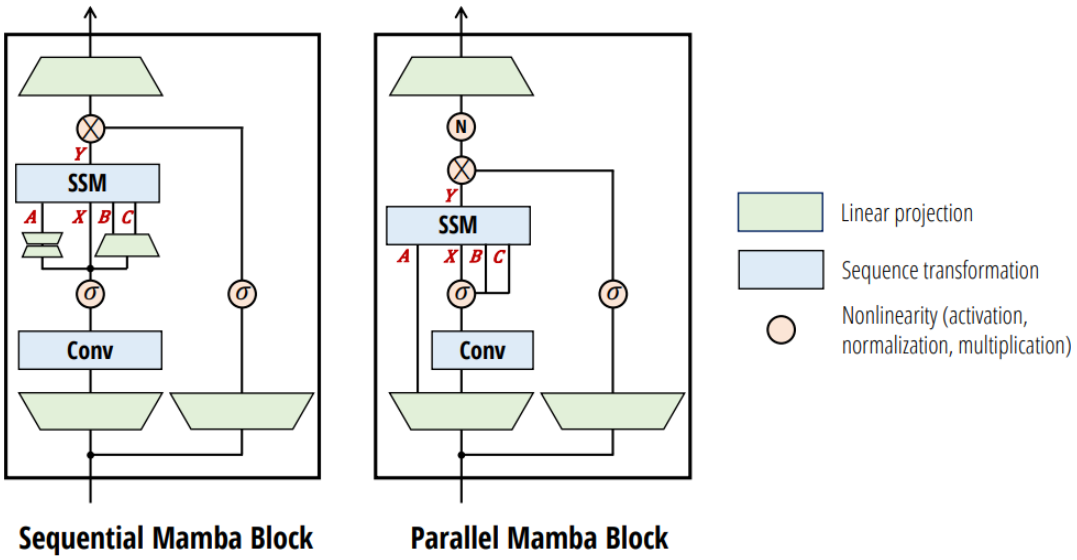


Figure 4.6: The difference between the sequential and parallel Mamba block as in [42]

Although we did not use this in our work, we have to point out the Mamba-2 architecture proposed in [42]. This model class is a refinement of the original Mamba, that is better suited for the usual computational setup which is tailored for transformer architectures, resulting in a model class that is faster, while continuing to be competitive with Transformers on language modeling. The paper [42] refines their selection mechanism based on their framework called State-Space Duality that the Mamba-2 utilizes as the inner SSM layer, together with a modified parallel Mamba block instead of the sequential one. Since other than more efficient computational resource usage and implementation of

parallel computing, there are no significant theoretical differences between the Mamba and Mamba-2, intuitively, it is not justified to test the latter one as we only utilized one GPU at a time. For the detailed structure of the Mamba-2 see Figure 4.6 and [42].

4.4 Length independent generalization bounds for deep SSM architectures

Paper [36] introduced the LRU model that had stable parametrization and achieved significantly better results on long-range dependencies than previous architectures such as the S4 [34] or LSSL [37], hence, the question rises whether or not stability of the underlying LTI systems has an effect on model performance. Intuitively it is clear that stable models can learn stable systems better but there were no mathematical proof on the effect of stability. The paper [29] explores this idea and introduces a PAC bound (Definition 3.1.5) for the generalization error of deep SSMs but in order to understand their main theorem we introduce the definition of Rademacher contraction.

Definition 4.4.1 ([29])**.** Let $\mathbf{u} \in (\mathcal{U}, \|\cdot\|_{\mathcal{U}})$ and $\mathbf{y} \in (\mathcal{Y}, \|\cdot\|_{\mathcal{Y}})$, where $\mathcal{U}$ and $\mathcal{Y}$ are Banach spaces with their corresponding norms, and let $\mu \geq 0$ and $c \geq 0$. A set of functions $\mathcal{F} = \{f : \mathcal{U} \to \mathcal{Y}\}$ is said to be (μ, c)**-Rademacher Contraction**, if for all $N \in \mathbb{N}$ and $U \subseteq \mathcal{U}^N$ we have

$$\mathbb{E}_\sigma \left[\sup_{f \in \mathcal{F}} \sup_{\{\mathbf{u}_i\}_{i=1}^N \in U} \left\| \frac{1}{N} \sum_{i=1}^N \sigma_i f(\mathbf{u}_i) \right\|_{\mathcal{Y}} \right] \leq \mu \mathbb{E}_\sigma \left[\sup_{\{\mathbf{u}_i\}_{i=1}^N \in U} \left\| \frac{1}{N} \sum_{i=1}^N \sigma_i \mathbf{u}_i \right\|_{\mathcal{U}} \right] + \frac{c}{\sqrt{N}}, \quad (4.6)$$

where σ_i are independent *Rademacher variables* (from Definition 3.1.6), and in this case $\mathbb{E}_\sigma$ means just a notation, emphasizing that the only random variable is the σ.

Now we present a simplified version of the length independent PAC bound introduced in [29].

Theorem 4.4.1 ([29])**.** Let $\mathcal{F}$ be a family of L deep, deep SSMs with the structure defined in Definition 4.1.2 with a scalar output and an L_l-Lipschitz continuous (i.e., $|\ell(y_1, y_1') - \ell(y_2, y_2')| \leq L_l(|y_1 - y_2| + |y_1' - y_2'|), \forall y_1, y_2, y_1', y_2' \in \mathbb{R}$) element-wise loss function. Furthermore we assume that the following bounds exist for all the Σ LTI components in the whole architecture

$$\|\mathbf{u}\|_{\ell^{2,2}} \leq K_u, |y| \leq K_y, \|\Sigma\|_{\ell_1} \leq K_1, \|\Sigma\|_{\mathcal{H}_2} \leq K_2.$$

(where $\|\mathbf{u}\|_{\ell^{2,2}} = \sum_{k=1}^{T} \|\mathbf{u}(k)\|_2^2$ for a sequence length T). We also assume that the encoder, decoder and non-linear layers' weight matrices have bounded operator norms. If all the assumptions hold, then

$$\mathbb{P}_S \left(\forall f \in \mathcal{F} : \mathcal{L}(f) - \mathcal{L}_S(f) \leq \frac{\mu K_u L_l + c L_l}{\sqrt{N}} + K_l \sqrt{\frac{2\log(4/\delta)}{N}} \right) \geq 1 - \delta \tag{4.7}$$

where N is the size of the random sample S, μ and c are constant scalar values, depending on the bounds K_1, K_2, K_u, K_y and the bounds on the encoder, decoder and nonlinear blocks.

Intuitively Theorem 4.4.1 means that with the help of Rademacher complexity and Rademacher contraction, we can give an upper bound for the generalization error of deep SSMs, with some constants. More specifically, if we take a stable parametrization meaning that the corresponding $\mathbf{A}$ matrix is Schur according to Theorem 3.2.2, then the system norms $\|\Sigma\|_{\ell_1}$ and $\|\Sigma\|_{\mathcal{H}_2}$ are finite according to Theorem 3.2.4. According to Theorem 4.4.1, given that $\|\Sigma\|_{\ell_1} \leq K_1$ and $\|\Sigma\|_{\mathcal{H}_2} \leq K_2$, it is possible to provide a generalization bound - stated in Equation 4.7 - that depends on (among other constants) K_1 and K_2 and does not depend on the length T of the input time series. An intuitive consequence of this is that architectures with stable parameterizations may generalize better.

Sketch of the proof [29]. For the sketch of the proof we first need to interpret the meaning of a (μ, c)-Rademacher contraction (RC). The intuition behind Definition 4.4.1 is that for a function class $\mathcal{F}$ we can bound its Rademacher complexity with the μ and c constants and the Rademacher complexity of its input $\mathbf{u}$. The key of the proof is to prove that every layer of our deep SSM is a (μ_i, c_i)-RC, from there, we can prove that the whole architecture is a (μ, c)-RC with the following lemma.

Lemma 4.4.2 ([29]). Let $\Phi_1 = \{\varphi_1 : X_1 \to X_2\}$ be (μ_1, c_1)-RC and $\Phi_2 = \{\varphi_2 : X_2 \to X_3\}$ be (μ_2, c_2)-RC. Then the set of compositions $\Phi_2 \circ \Phi_1 := \{\varphi_2 \circ \varphi_1 \mid \varphi_1 \in \Phi_1, \varphi_2 \in \Phi_2\}$ is $(\mu_1\mu_2, \mu_2 c_1 + c_2)$-RC.

Proof [29]. Let the Banach spaces which contain X_i be denoted by X_i for $i = 1, 2, 3$. Let $Z \subseteq X_1^N$ and

$$\widetilde{Z} = \left\{ \{\varphi_1(\mathbf{u}_i)\}_{i=1}^N \mid \varphi_1 \in \Phi_1 \right\}.$$

We have

$$\mathbb{E}_\sigma \left[\sup_{\varphi_2 \in \Phi_2} \sup_{\varphi_1 \in \Phi_1} \sup_{\{\mathbf{u}_i\}_{i=1}^N \in Z} \left\| \frac{1}{N} \sum_{i=1}^N \sigma_i \varphi_2(\varphi_1(\mathbf{u}_i)) \right\|_{X_3} \right]$$

$$= \mathbb{E}_\sigma \left[\sup_{\varphi_2 \in \Phi_2} \sup_{\{v_i\}_{i=1}^N \in \widetilde{Z}} \left\| \frac{1}{N} \sum_{i=1}^N \sigma_i \varphi_2(v_i) \right\|_{X_3} \right]$$

$$\leq \mu_2 \mathbb{E}_\sigma \left[\sup_{\{\varphi_1 \in \Phi_1\}} \sup_{\{\mathbf{u}_i\}_{i=1}^N \in Z} \left\| \frac{1}{N} \sum_{i=1}^N \sigma_i \varphi_1(\mathbf{u}_i) \right\|_{X_2} \right] + \frac{c_2}{\sqrt{N}}$$

$$\leq \mu_2 \mu_1 \mathbb{E}_\sigma \left[\sup_{\{\mathbf{u}_i\}_{i=1}^N \in Z} \left\| \frac{1}{N} \sum_{i=1}^N \sigma_i \mathbf{u}_i \right\|_{X_1} \right] + \mu_2 \frac{c_1}{\sqrt{N}} + \frac{c_2}{\sqrt{N}}.$$

□

From here, the full proof proves that every layer in the deep SSM is a (μ, c)-RC, but here we only show a proof sketch for the inner SSM blocks, so basically the corresponding LTI systems. The full proof for the other layers can be found in [29].

Lemma 4.4.3 (RC bound for a family of LTI maps). Let $\mathcal{S}$ be a collection of LTI systems Σ, each of which induces a linear input–output operator between the Banach spaces $(\mathcal{U}, \|\cdot\|_{\mathcal{U}})$ and $(\mathcal{Y}, \|\cdot\|_{\mathcal{Y}})$ as defined in Definition 3.2.13.

$$S_{\Sigma,T} : \mathcal{U} \longrightarrow \mathcal{Y}, \qquad \big(S_{\Sigma,T}(\mathbf{u})\big)(t) = y(t) \quad (1 \leq k \leq T).$$

Assume that for every $\Sigma \in \mathcal{S}$ one has

$$\| S_{\Sigma,T} \|_{\mathrm{op}} = \sup_{\|\mathbf{u}\|_{\mathcal{U}}=1} \big\| S_{\Sigma,T}(\mathbf{u}) \big\|_{\mathcal{Y}} \leq K_S.$$

If $Z \subseteq \mathcal{U}^N$ and $\{\mathbf{u}_i\} := \{\mathbf{u}_i\}_{i=1}^N \in Z$, then

$$\mathbb{E}_\sigma \left[\sup_{\Sigma \in \mathcal{S}} \sup_{\{\mathbf{u}_i\} \in Z} \left\| \tfrac{1}{N} \sum_{i=1}^N \sigma_i S_{\Sigma,T}(\mathbf{u}_i) \right\|_{\mathcal{Y}} \right] \leq K_S \, \mathbb{E}_\sigma \left[\sup_{\{\mathbf{u}_i\} \in Z} \left\| \tfrac{1}{N} \sum_{i=1}^N \sigma_i \mathbf{u}_i \right\|_{\mathcal{Y}} \right],$$

where $\{\sigma_i\}_{i=1}^N$ are i.i.d. Rademacher random variables. In particular, the function class

$$\mathcal{F} = \big\{ \mathbf{u} \mapsto S_{\Sigma,T}(\mathbf{u}) \;\big|\; \Sigma \in \mathcal{S} \big\}$$

is $(K_S, 0)$-RC.

Proof. Since each $S_{\Sigma,T}$ is a linear operator, then for any fixed choice of Rademacher signs $\{\sigma_i\}_{i=1}^N$ and any inputs $\{\mathbf{u}_i\} \in Z$, we have

$$\frac{1}{N}\sum_{i=1}^{N} \sigma_i S_{\Sigma,T}(\mathbf{u}_i) \;=\; S_{\Sigma,T}\Big(\tfrac{1}{N}\sum_{i=1}^{N} \sigma_i \mathbf{u}_i\Big).$$

Therefore,

$$\begin{aligned}
\sup_{\Sigma\in\mathcal{S}} \Big\|\frac{1}{N}\sum_{i=1}^{N} \sigma_i S_{\Sigma,T}(\mathbf{u}_i)\Big\|_{\mathcal{Y}} &= \sup_{\Sigma\in\mathcal{S}} \Big\| S_{\Sigma,T}\Big(\tfrac{1}{N}\sum_{i=1}^{N} \sigma_i \mathbf{u}_i\Big)\Big\|_{\mathcal{Y}} \\
&\leq \sup_{\Sigma\in\mathcal{S}} \|S_{\Sigma,T}\|_{\mathrm{op}} \cdot \Big\|\tfrac{1}{N}\sum_{i=1}^{N} \sigma_i \mathbf{u}_i\Big\|_{\mathcal{U}} \\
&\leq K_S \Big\|\tfrac{1}{N}\sum_{i=1}^{N} \sigma_i \mathbf{u}_i\Big\|_{\mathcal{U}}.
\end{aligned}$$

Taking expectation over the Rademacher variables and then the supremum over all choices $\{\mathbf{u}_i\} \in Z$, we obtain

$$\mathbb{E}_\sigma\left[\sup_{\Sigma\in\mathcal{S}} \sup_{\{\mathbf{u}_i\}\in Z} \Big\|\tfrac{1}{N}\sum_{i=1}^{N} \sigma_i S_{\Sigma,T}(\mathbf{u}_i)\Big\|_{\mathcal{Y}}\right] \;\leq\; K_S\, \mathbb{E}_\sigma\left[\sup_{\{\mathbf{u}_i\}\in Z} \Big\|\tfrac{1}{N}\sum_{i=1}^{N} \sigma_i \mathbf{u}_i\Big\|_{\mathcal{U}}\right].$$

By definition of Rademacher complexity, this shows that $\mathcal{F}$ is $(K_S, 0)$-RC. □

Combining this with the previous lemma and the intuition we gave for the definition of RC we obtain that the Rademacher complexity of the whole deep SSM can be bounded by the Rademacher complexity of its input and the $\mu, c \geq 0$ constants. From here we use the general definition of a PAC bound, given in Definition 3.1.5 and Theorem 3.1.2 to obtain (4.7).

□

Chapter 5

Implementation and experiments

In this chapter we discuss the handling of the data in more detail, describe the implementations of the architectures, and present the different experiments we ran and the results we obtained.

5.1 Data preparation

5.1.1 The data

Developing any machine learning algorithm requires extensively pre-processed and well-prepared data. For this study, a database provided by the BME Motorsport Formula Student Racing Team (which the author is a member of) is utilized. In this thesis, we analyze the testing data collected from the racing cars built by this team. The data was collected over a three-year period from different cars, tracks and conditions improving variability, reducing the risks of overfitting a track or weather specific bias. The data we received was originally provided in an engine management environment, called MaxxECU and contained data from more than 20 different tracks and by four different drivers. In order to validate the data, real-world tests have been conducted in cooperation with the racing team, where we measured the fuel consumption of the car manually and compared it to the virtual fuel tank data recorded by a sensor on the car. During these tests, first we reset the virtual fuel tank data to the real size of our tank, filled up the full tank, and after the run was completed we compared the logged level of the fuel tank to the measured results. After these tests it was clear that the logged fuel consumption data was accurate up to the precision of deciliters and could thus be used in this study as the label. The engine control unit on the car also calculates instantaneous consumption but the real life tests

confirmed that this data is not reliable and therefore it should not be used as labels. The inaccuracies in this data also indicated that while naive, physics-based approaches can be used to estimate fuel consumption, they heavily rely on exact measurements, supporting the usage of deep learning methods. We further explore and implement a naive, physics based method in Chapter 5.2, and compare it to our models in Chapter 5.3.

The dataset contained various modalities of data, many of which were unusable and had to be removed. Initially, we categorized the data into two main groups: full laps and samples. In the full laps group we collected data from different tracks where the car made at least one full lap around the given track, and the data where the car was running, but not on a particular track, rather just a few meters in random shapes for testing, were categorized in the samples group. The latter can be used by the learning algorithm, but some full lap data is required to truly validate the functionality of the model. Following the data cleaning and preparation process, full lap data from 15 different tracks were collected, with each track containing at least two completed laps. Additionally, over 30 sample datasets were available, with lengths ranging from several thousand to nearly one hundred thousand time frames.

5.1.2 Data cleaning, sampling and labeling

Our task involved extracting the data from the given environment into manageable CSV files, followed by data preprocessing steps such as cleaning, removal of irrelevant features, and performing final sampling and dataset partitioning. The feature selection was done with the help of BME Motorsport. The students who designed the engine used in the car, specialized in combustion engines, gave us insights into what features can significantly impact fuel consumption and based on their input and data availability we settled with the 16 dimensions displayed in Table 5.1. This selection of features provides a robust dataset that implicitly captures ambient weather conditions (e.g., Intake Air Temperature, Fuel Temperature), driver behavior and engine characteristics (e.g., Throttle Position, Front Left Wheel Speed), as well as indicators influencing fuel usage (Fuel Pulse Primary, Fuel Angle, etc). Please note that some of the features are hard to simulate and according to the intuition of engineers, some of them influence fuel consumption more than others, so in Chapter 5.3 we also explore models trained without the Fuel Duty Primary, Fuel Load Primary and Fuel Pulse Primary sequences, and models trained with those features but tested without them.

Feature	Description
Oil Temperature	Temperature of the engine oil, affecting lubrication and engine efficiency.
Fuel Temperature	Temperature of the fuel, influencing combustion behavior and density.
Oil Pressure	Pressure of the engine oil, critical for the engine proper functions and component lubrication.
Fuel Pressure	Pressure at which fuel is delivered to the injectors, impacts fuel delivery accuracy.
Coolant Temperature	Engine coolant temperature, a key indicator of engine thermal state.
Lambda	Oxygen sensor value indicating air-fuel mixture ($\lambda = 1$ is stoichiometric).
Intake Air Temperature	Temperature of air entering the engine, affects air density and combustion.
RPM	Engine speed in revolutions per minute, directly related to power output.
Throttle Position	Position of the throttle valve, indicating driver power demand.
Acceleration Enrichment	Additional fuel added during throttle changes to prevent hesitation.
Fuel Angle	Crank angle at which fuel injection begins, impacting combustion efficiency and engine performance.
Fuel Duty Primary	Injector duty cycle, representing fuel injection duration as a percentage.
Fuel Load Primary	Fuel load is the fuel amount required by the engine influencing fuel injection and ignition.
Fuel Pulse Primary	Duration of fuel injection pulses, controlling fuel volume delivered.
Total Fuel Trim	Correction applied to base fuel map to achieve desired air-fuel ratio.
Wheelspeed Front Left	Rotational speed of the front left wheel, used for vehicle dynamics control.

Table 5.1: Selected engine and vehicle parameters used as features in the fuel consumption analysis

Additionally, since all features within one time series, originate from the same vehicle and physical system, they all influence one another in some way. For a detailed overview of their correlations, refer to Figure 5.1. After the feature selection process, we had a set of time series with various length and some of them even containing more than one lap worth of data. In order to separate these time series into smaller ones containing the data for only one lap, a periodicity detection algorithm was implemented, discussed in more detail in Chapter 5.2.1. Please note however that for the robustness of the dataset we intentionally left some of these multi lap data untouched for more varied data samples.

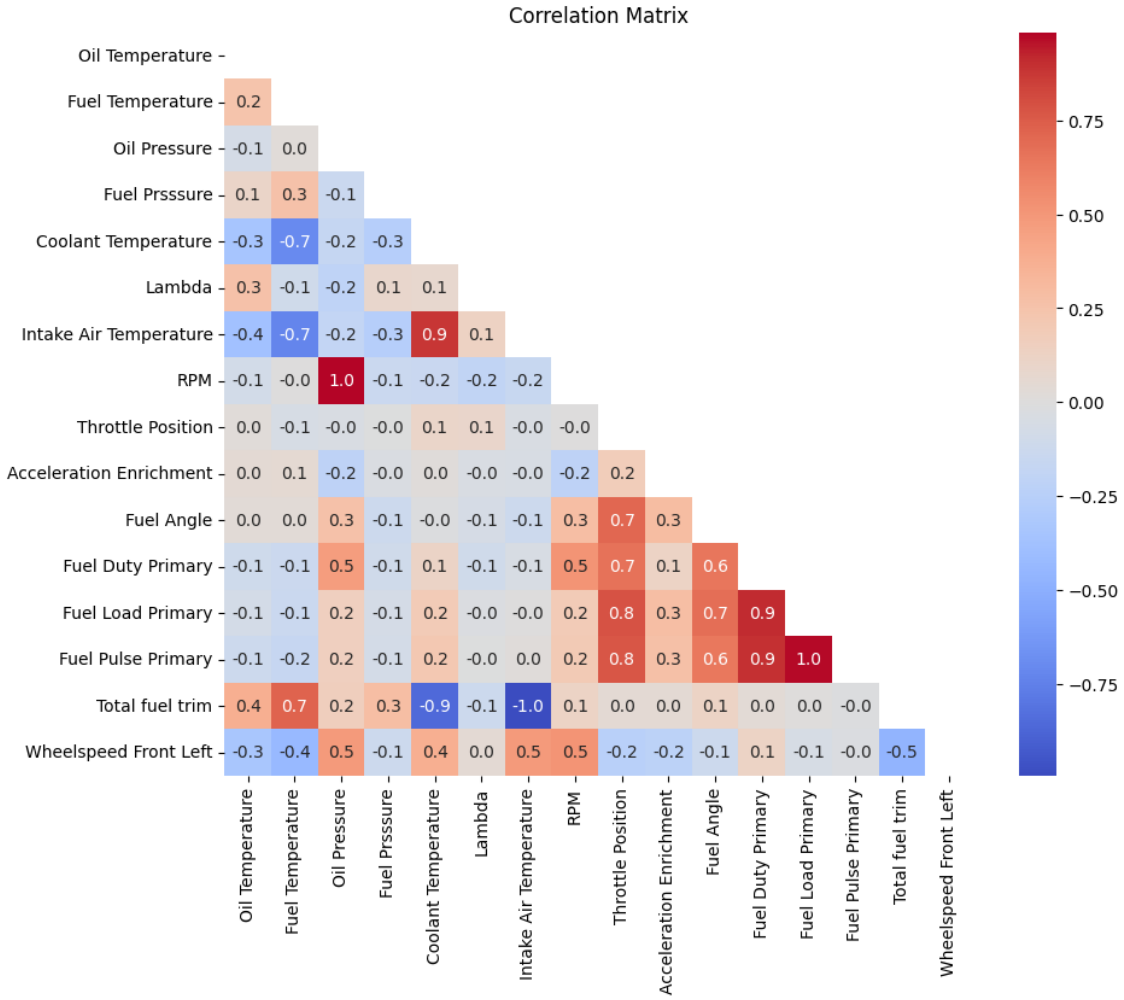
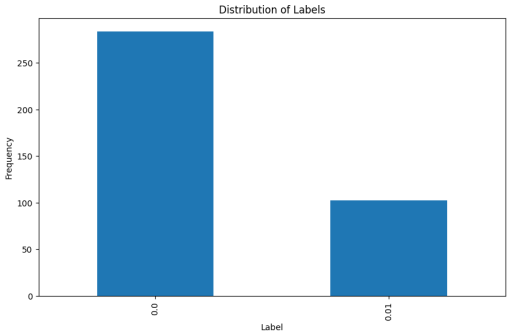


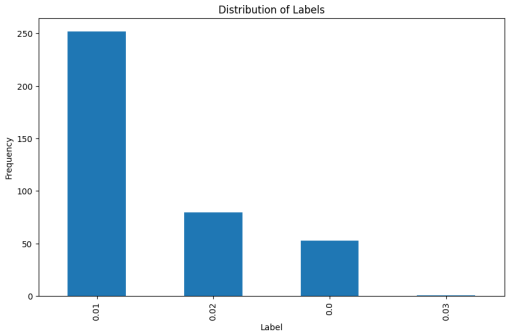
Figure 5.1: Correlation of the selected features

After the above mentioned steps we were left with separate time series varying hugely in length. In order to obtain a dataset consisting of sequences with the same length, the main author implemented a data sampling process. After experimenting with different lengths we settled with 1300 timestamp long time series in order to have a semi-optimal distribution of the labels. The graphs displaying the distribution of labels for different lengths of time series' can be seen in Figure 5.2.

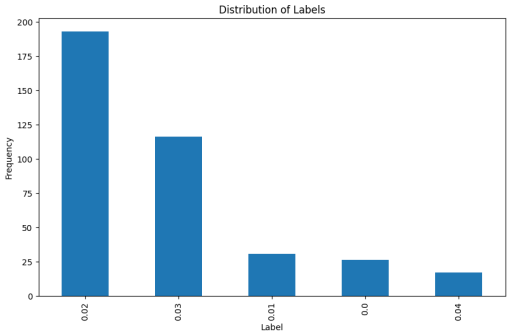
After performing all the steps explained earlier we obtained a dataset tailored to our needs. For a concrete, stratified, 80-20 split we obtained a dataset containing 175 time series in our test set and 698 in the train set. In our case stratified split means maintaining the same distribution of labels in the train and test set with the distribution of labels for length of 1300 displayed on Figure 5.2.



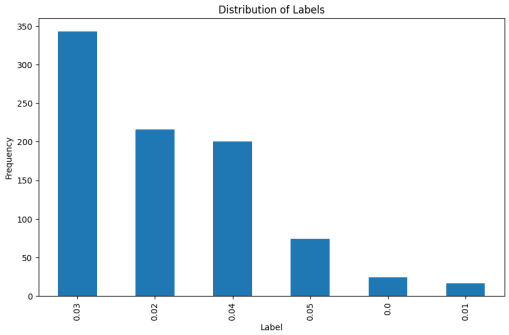
(a) Length = 130



(b) Length = 500



(c) Length = 1000



(d) Length = 1300

Figure 5.2: Distribution of fuel consumption labels for different input sequence lengths.

5.2 Implementation

5.2.1 Periodicity detection

Suppose that we have a given feature and its values over time collected into a time series. Let T be the series and $T(i)$ the corresponding value of the feature at time-stamp i. It is possible to define periodicity in multiple ways but in this study we will use segment periodicity as it is the one relatable to our data. The idea behind segment periodicity is that it focuses on the entire time series' correlation with itself, delayed by different numbers of lags [43]. An intuitive definition for segment periodicity is a pattern in a time series that occurs at regular time intervals [44]

Definition 5.2.1 ([43])**.** Let S be a similarity measure, specifically in our case: $H(u,v) = \sum_{j=0}^{m-1} \begin{cases} 1 & \text{if } u_j \neq v_j \\ 0 & \text{if } u_j = v_j \end{cases}$, $S(u,v) = 1 - H(u,v)/m$, where H is the Hamming distance. Then if a time series T of length n can be sliced into equal-length segments $T_0, T_1, \ldots, T_i, \ldots, T_N$, each of length p, where $T_i = e_{ip}, \ldots, e_{ip+p-1}$, $N = \lfloor n/p \rfloor - 1$,

$S(T_i, T_j) \geq \tau \ \ \forall i, j = 0, 1, \ldots, N$, and $0 \leq \tau \leq 1$, then T is said to be periodic with a period p with respect to periodicity threshold τ.

There are several ways to find periodicity in a time series, like the Fourier transform [44], but we used the autocorrelation method because it is easier to implement and intuitively a better fit for our case.

Correlation is a concept used mostly in probability theory and statistics. The definition of the correlation between two variables X and Y is defined as follows.

Definition 5.2.2 ([45])**.** The *correlation* of two random variables X and Y is defined as

$$\mathrm{Corr}(X,Y) = \frac{\mathrm{Cov}(X,Y)}{\mathrm{Sd}(X)\mathrm{Sd}(Y)} = \frac{\mathrm{Cov}(X,Y)}{\sqrt{\mathrm{Var}(X)\mathrm{Var}(Y)}},$$

provided $0 < \mathrm{Var}(X) < \infty$ and $0 < \mathrm{Var}(Y) < \infty$.

Where,

- $\mathrm{Cov}(X,Y) = \mathbb{E}\left[(X - \mathbb{E}(X))(Y - \mathbb{E}(Y))\right]$ is the **covariance** of X and Y;
- $\mathrm{Sd}(X) = \sqrt{\mathrm{Var}(X)}$ is the **standard deviation** of X;
- $\mathrm{Var}(X) = \mathbb{E}\left[(X - \mathbb{E}(X))^2\right]$ is the **variance** of X.

In our case, the autocorrelation function is the normalized correlation of the time series with a delayed copy of itself. Let T be a time series of length n, and define $\overline{T}$ as the mean of the series and $\mathrm{Var}(T)$ its variance. The autocorrelation at lag k is computed by the formula:

$$\mathrm{ACF}(k) = \frac{1}{(n-k) \cdot \mathrm{Var}(T)} \sum_{i=0}^{n-k-1} (T(i) - \overline{T})(T(i+k) - \overline{T})$$

To ensure stability and convergence of the autocorrelation values toward zero as the lag increases, we adjusted the denominator in the last 10% of the lag values by fixing the effective length to a constant. The implementation is summarized in the following steps:

1. Shift the series by subtracting its mean: $x - \bar{x}$.
2. Compute the autocorrelation of the normalized series with itself.
3. Retain the non-negative lags: $k \in [0, n)$.
4. Scale the result using the variance and a lag-dependent length factor to normalize the autocorrelation values.

This process was implemented using NumPy [46] as shown on Code 5.1:

```
def autocorr(x: np.ndarray) -> np.ndarray:

    """ Calculates the autocorrelation of the input array."""

    n = x.size # size of the array
    norm = (x - np.mean(x)) # normalizing the array by subtracting
        the mean
    result = np.correlate(norm, norm, mode='full') # finding the
        correlation of the array with itself
    lengths = np.concatenate((np.arange(n, round(n*0.1), -1), np.
        ones(round(n*0.1), dtype=int)*round(n*0.1))) # setting the
        last 10% of the array to a constant value so the
        autocorrelation is converging to 0
    acorr = result[n-1: 2*n] / (x.var() * lengths) # calculating
        the autocorrelation
    return acorr
```

Code 5.1: Autocorrelation function used to detect periodicity based on [47]

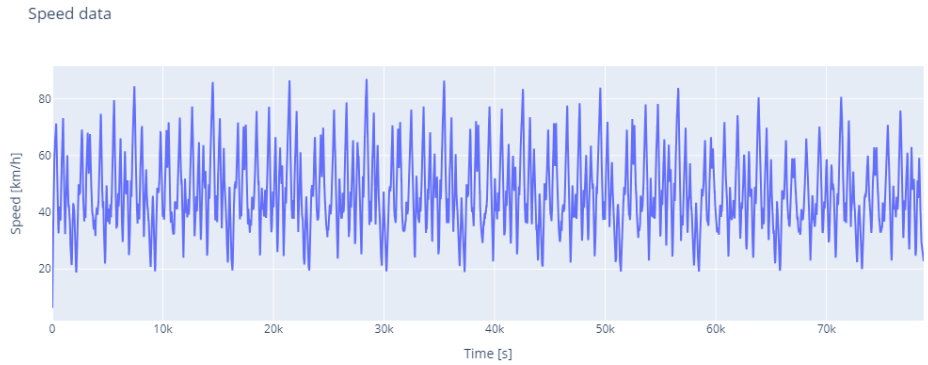
As an example of the autocorrelation function used to detect periodicity in real-world data, Figure 5.3a displays the measured speed data of our car during the completion of 10 laps.

To the human eye, it is clear that there is some periodicity in the data. Using the autocorrelation function on this data produces the plot shown in Figure 5.3b.

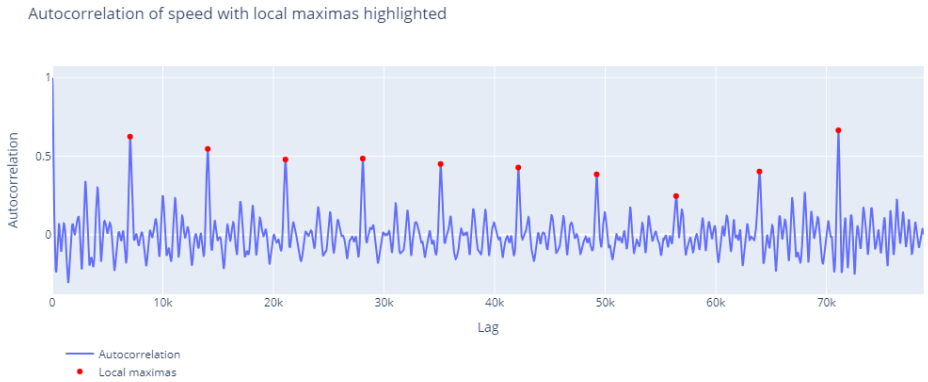
If the autocorrelation value is close to 1 for a certain lag, it means shifting the data by that many indices is very similar to the original data, meaning that the local maxima on the autocorrelation plot are the beginnings of the different laps. To find these local maxima we used a shifting window method.

5.2.2 The models

As mentioned above we implemented three different architectures, the LRU, the Mamba, and a simple Transformer. For the LRU and Mamba architectures we used open-source implementations available online in [48] and [49] with a few tweaks. Naturally we needed to incorporate our dataset into the frameworks of the model. For this we used PyTorch's [50] Dataset class, modified to our needs and extended with different kinds of normalization. We also implemented a BootstrapDataset class from scratch so the mod-



(a) Speed data over time



(b) Autocorrelation of the speed data

Figure 5.3: Speed data and its autocorrelation. Periodicity in the data becomes clearer in the autocorrelation plot.

els could learn underrepresented classes as well. For the codes of the upper mentioned datasets see Code A.2 and Code A.3 in Appendix A.

The LRU implementation we used originally only contained a classification model, so in order to use it for our regression task we implemented a regression model based on their classification one, displayed on Code A.4 in Appendix A. In addition, we implemented both L1 and L2 loss functions for the regression model, modifying the training and validation procedures accordingly. We also introduced an optional ReLU activation in place of the original GLU activation within the SequenceLayer class of the model. Furthermore, as the original code-base did not include functionality for saving model states, this feature was manually implemented. For definitions of L1/L2 loss, ReLU, GLU, and other related terms, definitions used in this chapter, please refer to Appendix B.

The Mamba codes also lacked some key features that we had to implement. The most important of these is the addition of support for stacking multiple layers, with residual

connections and layer normalizations applied between layers, fully developed by the main author and shown in Code A.1 in Appendix A. Furthermore, we applied learning rate schedulers and optimizers. We also implemented a framework that facilitates logging and enables seamless hyperparameter optimization.

The transformer model was implemented from scratch by the main author, with the usage of PyTorch and its built in layer classes. We built the model to work both for regression and classification tasks, using L1, L2 or cross-entropy losses accordingly. The class defining the model structure can be seen in Code A.5. Similarly to the Mamba model, we incorporated learning rate schedulers, optimizers, normalization, and pooling into our implementation, fitting it all in a framework designed to support comprehensive logging of model results and hyper-parameters. For more information on the structure of the Transformer architecture see [51].

As mentioned above we did experiments with both regression and classification models because of the special nature of our labels. Strictly speaking we are dealing with a regression task, however because of measurement limitations, our validation method resulted in labels that are in liter ranging between 0 and 0.05. In order to ensure numerical stability of the learning process, we multiplied the labels by 100, leaving us with labels that are integers in centiliters, making it possible to address the problem as a classification task. In the classification models the measurement of model performance was evidently in accuracy, while in the regression models we used both Mean Absolute Error (MAE) and a special form of accuracy to compare the performance of different models. In the regression tasks we defined a prediction "accurate", if it, rounded to the nearest integer, matched the label assigned to that sequence in centiliters. Accuracy of a given epoch was then simply calculated by dividing the number of correct predictions by the number of all predictions made. Occasionally, a model locally under- or overestimates the consumption, which means that instead of a simple rounding, an adaptive threshold method may achieve higher accuracy, which can be an interesting area of future research.

In order to compare the performance of the models to a physics based, naive approach, we also calculated the total fuel consumed over a lap by integrating the instantaneous injector flow rate. The calculations are presented in more detail in Appendix B. In Chapter 5.4 we evaluate the results of this calculations and compare it with the outputs of our models. The problem with this method is that it relies heavily on exact sensor measurements which are often not available or noisy. It is possible for our models to simply learn the physics based integration method we just described, so for the best architecture, chosen

based on the experiments described in the next chapter, we ran some additional training cycles, either just without the *Fuel Pulse Primary* sequence or also without all of the features heavily correlating with it (*Fuel Duty Primary, Fuel Load Primary*). Since the failure or unavailability of a given sensor is not a rare sight and some features are impossible to predict in a simulation environment, it adds to the robustness and practical usability of the model to exclude some of them.

5.3 Experiments

In order to find the optimal hyperparameters for each model, grid searches were done. We conducted many runs for each model class, using both regression and classification methods, experimenting with scaling, number of layers, initialization, initial learning rate, the type of learning rate scheduler, different loss functions, etc. We also did some experiments with bootstrapping the underrepresented labels with the help of our Bootstrap Dataset class displayed on Code A.3. After evaluating the results of the different experiments, we found that the regression model outperforms the classification model for each architecture. Taking into account the information gathered from more than 350 runs we ended up with a model structure for each architecture that seemed the most consistent and generally performed the best. For the LRU we found that the following hyperparameters performed the best overall. A regression model trained with L2 loss function, with 3 layers and batch normalization, initialized with He normal initialization, using ReLU activation, with 0.01 initial learning rate, scheduled by cosine annealing and using the Adam optimizer. The state dimension was 256, i.e. the state dimension of each LTI component of each SSM block. We used mean pooling and interestingly enough no normalization of any kind for the input data. The model has around 70 000 learnable parameters.

The Mamba model we chose had very similar complexity with around 105 000 parameters, but it consisted of 4 layers connected by residual connections and layer normalizations. For every model from now on we used an input scaling, dividing every single value in the whole dataset by the biggest absolute value of the train dataset, namely 12 437. Additionally the optimal Mamba model had a local convolution width of 4, state dimension of 256, block expansion factor of 2, that is a scalar value roughly equivalent to a scaling factor of the state space dimension, and used mean pooling with 0.01 initial learning rate, scheduled with cosine annealing and optimized by Adam. Here local con-

volution width is the size of the finite convolutional kernel that's applied locally, before or after the state-space update.

Naturally, the Transformer model had the biggest complexity, with more than 3 200 000 learnable parameters, accordingly using the most computational resources out of the three architectures. The best performing transformer model on our data had only 1 layer, 16 attention heads and 256 model dimension what is simply the dimension of the Transformer's internal vector space. We also experimented with larger models however they were struggling, indicating that we did not have enough data to properly train bigger transformers. This relatively small transformer already had more than 30 times the amount of parameters used by the other two models, while providing comparable results, showing us that the usage of bigger models may not be beneficial even if more data were available for us. The training process for the Transformer model, just like the Mamba, used the L2 loss function, a base learning rate of 0.01, a cosine anneal scheduler and the Adam optimizer.

5.4 Evaluation

After choosing a generally best model for each architecture, we evaluated and compared them with each other. For this we created 10 independent stratified train-test splits and trained and tested the chosen model structure from each model class, to see and compare the results of these models refer to Figure 5.4.

On Figure 5.4 we can see that generally the Mamba performs the best as it has the highest median accuracy, while also being more consistent than the Transformer architecture, as its box is smaller, indicating less variance between the accuracies. Based on the diagram the LRU performs the worst, but it still achieves a median accuracy of around 67% while also being consistent as it has the smallest box. This box supports our claim that deep SSMs can at least match the performance of Transformer models, with some specific architectures even outperforming them.

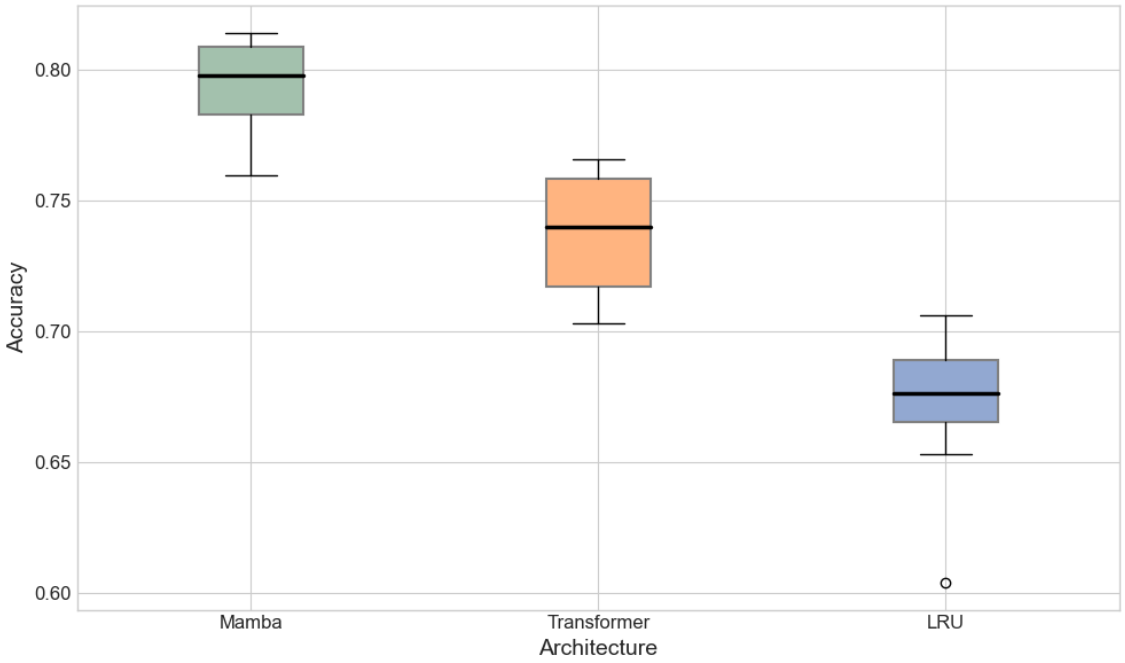


Figure 5.4: Boxplot of the accuracies achieved by the best performing models for the 10 independent train-test split

For further evaluation we took 10 laps worth of data, distinct from the train and test datasets of any previously performed experiments, each consisting of five to eight, 1300 long segments and predicted the overall consumption of each lap by predicting the consumptions on the time segments and summing them. We refer to this set of 10 laps as the evaluation set. The 10 laps were recorded at three different tracks, by three different drivers and in varying weather conditions, providing us a robust evaluation set without any clear bias. The predictions of the models, the MAEs and the standard deviations (Std. Dev) compared to each other and the naive method (described in Appendix B) are all displayed in Table 5.2.

Based on Table 5.2 we can further emphasize that in general the Mamba model performs the best, even on our evaluation set. While in some special cases (on Track B in our case) the Transformer can outperform the Mamba architecture, even then it has higher standard deviation, indicating a less consistent model. In general the LRU lags behind the other 2 models but it still achieves assessable results with relatively small deviation. The naive, physics based method also achieves results close to our models' but other than a few exceptional cases it is in general worse than our best performing model.

Table 5.2: Results of the models across all tracks of our validation set

(a) Track A

Model	Metric	Lap 1	Lap 2	Lap 3	Lap 4
	True Label	28	30	30	28
Mamba	Rounded Pred	28.3	29.5	30.6	**27.9**
	Pred ± Std	28.579 ± 0.683	**29.684 ± 0.848**	**29.878 ± 0.646**	28.148 ± 0.261
	MAE ± Std(MAE)	0.791 ± 0.421	0.703 ± 0.569	0.470 ± 0.460	0.226 ± 0.198
Transformer	Rounded Pred	27.7	27.8	29.3	28.6
	Pred ± Std	**27.837 ± 0.646**	28.036 ± 0.978	29.019 ± 1.063	28.627 ± 0.890
	MAE ± Std(MAE)	0.583 ± 0.323	1.971 ± 0.964	1.072 ± 0.971	0.869 ± 0.656
LRU	Rounded Pred	27.3	25.7	28.6	26.7
	Pred ± Std	26.844 ± 1.306	25.248 ± 1.479	28.713 ± 1.635	26.704 ± 1.724
	MAE ± Std(MAE)	1.335 ± 1.122	4.752 ± 1.479	1.731 ± 1.154	1.879 ± 1.058
Naive method	Pred	27.770	28.244	28.782	27.866
	Error	–0.230	–1.756	–1.218	–0.134

(b) Track B

Model	Metric	Lap 5	Lap 6	Lap 7
	True Label	23	22	21
Mamba	Rounded Pred	25.5	24.2	22.7
	Pred ± Std	25.507 ± 0.621	24.246 ± 0.527	22.740 ± 0.357
	MAE ± Std(MAE)	2.507 ± 0.621	2.246 ± 0.527	1.740 ± 0.357
Transformer	Rounded Pred	24.5	**22.6**	22.6
	Pred ± Std	25.039 ± 0.847	22.864 ± 1.985	22.503 ± 0.969
	MAE ± Std(MAE)	2.039 ± 0.847	1.768 ± 1.250	1.544 ± 0.901
LRU	Rounded Pred	24.7	23.7	23.5
	Pred ± Std	24.924 ± 1.683	23.579 ± 1.372	23.981 ± 1.731
	MAE ± Std(MAE)	2.237 ± 1.238	1.879 ± 0.921	3.152 ± 1.395
Naive method	Pred	**23.911**	23.384	**22.443**
	Error	0.911	1.384	1.443

(c) Track C

Model	Metric	Lap 8	Lap 9	Lap 10
	True Label	11	8	9
Mamba	Rounded Pred	12.4	9.9	9.3
	Pred ± Std	**12.319 ± 0.571**	**9.289 ± 0.530**	**9.202 ± 0.429**
	MAE ± Std(MAE)	1.319 ± 0.571	1.289 ± 0.530	0.365 ± 0.302
Transformer	Rounded Pred	13.9	10.6	10.5
	Pred ± Std	14.323 ± 0.799	10.856 ± 2.283	10.589 ± 0.354
	MAE ± Std(MAE)	3.323 ± 0.799	3.619 ± 0.519	1.589 ± 0.354
LRU	Rounded Pred	13.9	10.9	10.1
	Pred ± Std	13.937 ± 0.428	11.412 ± 0.550	10.797 ± 0.213
	MAE ± Std(MAE)	2.937 ± 0.428	3.412 ± 0.550	1.797 ± 0.213
Naive method	Pred	12.721	9.976	10.061
	Error	1.721	1.976	1.061

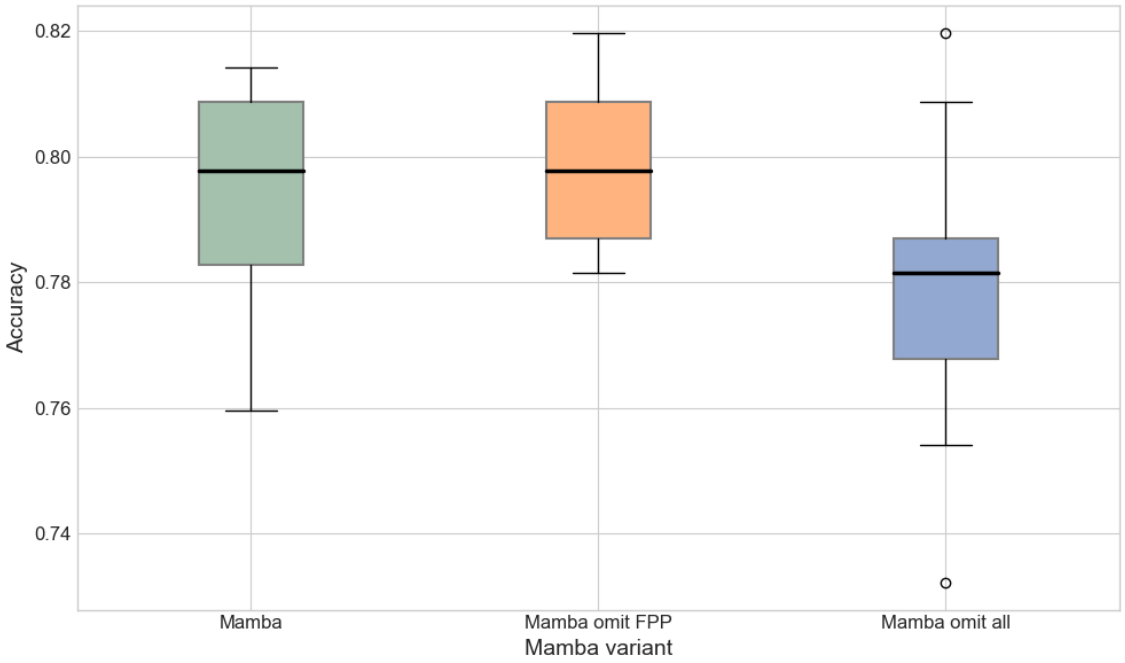


Figure 5.5: Boxplot of the accuracies of the different Mamba variants for the 10 train-test splits

The superiority of our best-performing model over the physics-based approach can be seen from the analysis of the error terms in Table 5.2. Specifically, the physics-based model consistently underestimates fuel consumption on Track A, while it tends to overestimate it on Tracks B and C. This is likely attributable to variations in ambient environmental conditions that can affect the sensor's measurement accuracy. In contrast, our machine learning models are capable of learning these environmental perturbations and integrating them into their predictions, thereby delivering more reliable and accurate estimates.

For further evaluation, we conducted more experiments with our best performing Mamba architecture. In the following Table and Figure, Mamba omit FPP means our best performing Mamba model evaluated with the *Fuel Pulse Primary* feature replaced with 0-s, while Mamba trained omit FPP and Mamba trained omit all fuel respectively means training our best model class either only without the *Fuel Pulse Primary* or also without the *Fuel Load Primary* and *Fuel Duty Primary* features. The comparison between the Mamba model and its above mentioned variants are summarized in Table 5.3 for our evaluation set, and for the accuracies of the 10 independent train-test splits refer to Figure 5.5.

As one would assume, we can see on Figure 5.5 that the removal of all features, strongly correlated with fuel injection, results in a generally worse, yet still well-

performing model. Interestingly enough from Figure 5.5 we can also obtain that on our test set on average the Mamba model trained without *Fuel Pulse Primary* performs the best. One possible reason for that can be the noise coming from the sensor or the learning process avoiding getting stuck in a local maximum.

Table 5.3 shows us an even more interesting picture that is kind of contrary to the one obtained from Figure 5.5, as here the Mamba model trained without all fuel features often outperforms everything else, in some cases even the Transformer model or the naive method. Another interesting phenomenon is the relatively bad performance of the Mamba model trained without *Fuel Pulse Primary*, as on our test set it performed the best, as displayed on Figure 5.5. One key takeaway from Table 5.3 is the great performance of the Mamba omit FPP model, promising great practical usability as according to this the model can handle missing data pretty well.

An important aspect of any machine learning model is computational cost. In Chapter 2, we claimed that deep SSMs often outperform transformers while consuming less computational resources. On our hardware (CPU: AMD EPYC 7F72, 48 cores (96 threads), 3.2 Ghz / core; GPU: Nvidia A100 40GB), one epoch for the Mamba model took 1.986s $\pm$ 0.08s averaging over 50 epoch, while one epoch for the transformer took 4.852s $\pm$ 0.113s averaging over 50 epoch. The GPU usage of the Mamba model was 2566 MB, while for the Transformer it was 4837 MB, both for the same batch size, further supporting our claim.

Table 5.3: Comparison of the mamba models

(a) Track A

Model	Metric	Lap 1	Lap 2	Lap 3	Lap 4
	True Label	28	30	30	28
Mamba	Rounded Pred	28.3	29.5	30.6	**27.9**
	Pred ± Std	28.579 ± 0.683	**29.684 ± 0.848**	**29.878 ± 0.646**	28.148 ± 0.261
	MAE ± Std(MAE)	0.791 ± 0.421	0.703 ± 0.569	0.470 ± 0.460	0.226 ± 0.198
Mamba omit FPP	Rounded Pred	**28.1**	28.8	29.7	**27.9**
	Pred ± Std	28.187 ± 0.641	29.344 ± 0.778	29.513 ± 0.596	27.768 ± 0.236
	MAE ± Std(MAE)	0.519 ± 0.420	0.716 ± 0.723	0.591 ± 0.493	0.279 ± 0.176
Mamba trained omit FPP	Rounded Pred	33.9	34.0	34.6	33.2
	Pred ± Std	33.552 ± 3.757	33.637 ± 4.409	34.796 ± 3.030	33.501 ± 3.052
	MAE ± Std(MAE)	5.560 ± 3.745	4.726 ± 3.214	4.936 ± 2.796	5.501 ± 3.052
Mamba trained omit all fuel	Rounded Pred	26.7	27.7	28.9	27.5
	Pred ± Std	27.266 ± 0.627	28.185 ± 0.720	28.547 ± 0.368	27.471 ± 0.482
	MAE ± Std(MAE)	0.794 ± 0.548	1.815 ± 0.720	1.453 ± 0.368	0.675 ± 0.237

(b) Track B

Model	Metric	Lap 5	Lap 6	Lap 7
	True Label	23	22	21
Mamba	Rounded Pred	25.5	24.2	22.7
	Pred ± Std	25.507 ± 0.621	24.246 ± 0.527	22.740 ± 0.357
	MAE ± Std(MAE)	2.507 ± 0.621	2.246 ± 0.527	1.740 ± 0.357
Mamba omit FPP	Rounded Pred	25.2	24.1	22.6
	Pred ± Std	25.161 ± 0.584	23.930 ± 0.478	22.398 ± 0.345
	MAE ± Std(MAE)	2.161 ± 0.584	1.930 ± 0.478	1.398 ± 0.345
Mamba trained omit FPP	Rounded Pred	29.2	28.1	26.8
	Pred ± Std	29.198 ± 3.707	28.218 ± 3.191	26.687 ± 2.910
	MAE ± Std(MAE)	6.278 ± 3.570	6.218 ± 3.191	5.687 ± 2.910
Mamba trained omit all fuel	Rounded Pred	**23.0**	**22.5**	**21.4**
	Pred ± Std	23.357 ± 0.848	22.684 ± 0.486	21.445 ± 0.595
	MAE ± Std(MAE)	0.737 ± 0.550	0.690 ± 0.478	0.602 ± 0.436

(c) Track C

Model	Metric	Lap 8	Lap 9	Lap 10
	True Label	11	8	9
Mamba	Rounded Pred	12.4	9.9	9.3
	Pred ± Std	12.319 ± 0.571	9.289 ± 0.530	9.202 ± 0.429
	MAE ± Std(MAE)	1.319 ± 0.571	1.289 ± 0.530	0.365 ± 0.302
Mamba omit FPP	Rounded Pred	12.2	9.8	9.3
	Pred ± Std	**12.132 ± 0.619**	9.192 ± 0.549	**9.083 ± 0.438**
	MAE ± Std(MAE)	1.134 ± 0.616	1.192 ± 0.549	0.347 ± 0.280
Mamba trained omit FPP	Rounded Pred	16.1	10.6	12.0
	Pred ± Std	16.114 ± 0.544	11.218 ± 0.848	11.733 ± 0.489
	MAE ± Std(MAE)	5.114 ± 0.544	3.218 ± 0.848	2.733 ± 0.489
Mamba trained omit all fuel	Rounded Pred	12.7	9.6	9.7
	Pred ± Std	12.661 ± 1.066	**9.172 ± 0.854**	9.378 ± 0.626
	MAE ± Std(MAE)	1.661 ± 1.066	1.226 ± 0.774	0.550 ± 0.481

Chapter 6

Conclusion and further research

Based on the systematic evaluation of the models, described in the previous, chapter our work supports the claim that deep SSMs can match and often outperform Transformers in predicting long range sequences while using much less computational power. Based on our results we can also say that deep learning models mostly outperform naive, physics based calculations based on noisy sensors, and they have the potential to be applied in practice for fuel consumption prediction. Another interesting remark is that the models we used are all built on stable parametrization, yet they performed really well on our unique, and noisy data, further emphasizing our claim that stability of the underlying dynamical systems can significantly impact the model performance.

An interesting future research idea is predicting other sequence length data for more general usability and to remove the bias the current length has on the model. However implementing it could be tricky because of uneven label distribution. Further research may include setting random features to zero for random time series in our train set for robustness and for better handling of occasional faulty sensor data. Also we are planning to integrate the models into the simulation environments used by BME Motorsport for better race strategy optimization.

Bibliography

[1] Statist. URL: https://www.statista.com/statistics/307194/top-oil-consuming-sectors-worldwide/.

[2] World Resources Institute. URL: https://www.wri.org/insights/4-charts-explain-greenhouse-gas-emissions-countries-and-sectors.

[3] Manjunath TK and 2Ashok Kumar PS. "Fuel Prediction Model for Driving Patterns Using Machine Learning Techniques". In: *Journal of Computer Science* (2023).

[4] Sasanka Katredd. "Development of Machine Learning based approach to predict fuel consumption and maintenance cost of Heavy-Duty Vehicles using diesel and alternative fuels". PhD thesis. West Virginia University, 2023.

[5] FIA. URL: https://www.fia.com/sites/default/files/fia_2025_formula_1_sporting_regulations_-_issue_1_-_2024-07-31.pdf.

[6] R. Frederick and B. Dixon. "Optimizing Performance and Fuel Efficiency for a Formula SAE Car," in: *SAE Technical Pape* (2019).

[7] NASCAR. *When fuel is all that matters*. NASCAR. 2014. URL: https://www.nascar.com/news-media/2014/07/23/when-fuel-is-all-that-matters/.

[8] Your Data Driven. URL: https://www.yourdatadriven.com/race-fuel-calculator/.

[9] Pol Duhr et al. "Minimum-Race-Time Energy Allocation Strategies for the Hybrid-Electric Formula 1 Power Unit". In: *IEEE Transactions on Vehicular Technology* (2023). ETH Zürich.

[10] Francesco Braghin, Luca Paparusso snd Manuel Riani, and Fabio Ruggeri. *Competitors-Aware Stochastic Lap Strategy Optimisation for Race Hybrid Vehicles*. Arxiv. 2022. URL: https://arxiv.org/pdf/2203.00084.

[11] Marc Ross. “Fuel efficiency and the physics of automobiles”. In: *Contemporary Physics* 38.6 (1997), pp. 381–394.

[12] Ben Michael, Efraim Shmerling, and Alon Kuperman. “Analytic Modeling of Vehicle Fuel Consumption”. In: *Energies* 6 (Jan. 2013), pp. 117–127. DOI: 10.3390/en6010117.

[13] Mengxi Wu and Gustav Norman. “Optimal Driving Strategies for Minimizing Fuel Consumption and Travelling Time”. MA thesis. KTS Stockholm, Royal Institute of Technology, 2013.

[14] Hai Yin and Zhiyuan Liu. “Fuel–air ratio control for a spark ignition engine using gain-scheduled delay-dependent approach”. In: *IET Control Theory & Applications* 9.12 (2015), pp. 1810–1820.

[15] Behrouz Ebrahimi et al. “A parameter-varying filtered PID strategy for air–fuel ratio control of spark ignition engines”. In: *Control Engineering Practice* 20.8 (2012), pp. 805–815.

[16] Valerio Turri, Bart Besselink, and Karl H Johansson. “Cooperative look-ahead control for fuel-efficient and safe heavy-duty vehicle platooning”. In: *IEEE Transactions on Control Systems Technology* 25.1 (2016), pp. 12–28.

[17] Erik Hellström, Jan Åslund, and Lars Nielsen. “Design of an efficient algorithm for fuel-optimal look-ahead control”. In: *Control Engineering Practice* 18 (Nov. 2010), pp. 1318–1327. DOI: 10.1016/j.conengprac.2009.12.008.

[18] Sandareka Wickramanayake and H.M.N. Dilum Bandara. “Fuel consumption prediction of fleet vehicles using Machine Learning: A comparative study”. In: *2016 Moratuwa Engineering Research Conference (MERCon)*. 2016.

[19] Mazhar Hussain et al. “Selection of optimal parameters to predict fuel consumption of city buses using data fusion”. In: *2022 IEEE Sensors Applications Symposium (SAS)*. 2022.

[20] Xuhui Wang and Xinfeng Chen. “A Support Vector Method for Modeling Civil Aircraft Fuel Consumption with ROC Optimization”. In: *2014 Enterprise Systems Conference*. 2014.

[21] Jarosław Ziółkowski et al. “Use of Artificial Neural Networks to Predict Fuel Consumption on the Basis of Technical Parameters of Vehicles”. In: (2021).

[22] Ngo Le Huy Hien and Ah-Lian Kor. "Analysis and Prediction Model of Fuel Consumption and Carbon Dioxide Emissions of Light-Duty Vehicles". In: (2022).

[23] Yongjie Zhu, Yi Zuo, and Tieshan Li. "Predicting ship fuel consumption based on LSTM neural network". In: *2020 7th International conference on information, cybernetics, and computational social systems (ICCSS)*. IEEE. 2020, pp. 310–313.

[24] Guanqun Wang et al. "Predictability of vehicle fuel consumption using LSTM: Findings from field experiments". In: *Journal of Transportation Engineering, Part A: Systems* 149.5 (2023), p. 04023030.

[25] Dan Yang et al. "A Multivariate Time Series Prediction Method for Automotive Controller Area Network Bus Data". In: (2024).

[26] Dengfeng Zhao et al. "A Review of the Data-Driven Prediction Method of Vehicle Fuel Consumption". In: (2023).

[27] Karl Lundgren and Erik Norlin. "Predicting Fuel Consumption of Marine Vessels-Using Boosted Regression Trees and Structured State-space Sequence Models While Considering Weather Forecasts". In: (2024).

[28] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[29] Dániel Rácz, Mihály Petreczky, and Bálint Daróczy. "Length independent generalization bounds for deep SSM architectures". In: *arXiv preprint arXiv:2405.20278* (2024).

[30] Péter Simon. *Differenciálegyenletek és dinamikai rendszerek*. ELTE. 2012. URL: `https://simonp.web.elte.hu/files/dinrendjegyzet.pdf`.

[31] Jing Sun Petros A. Ioannou. *Robust Adaptive Control*. Prentice Hall, 1995.

[32] Athanasios C. Antoulas. *Approximation of Large-Scale Dynamical Systems (Advances in Design and Control, Series Number 6)*. Society for Industrial and Applied Mathematics, 2005.

[33] VS Chellaboina et al. "Induced convolution operator norms for discrete-time linear systems". In: *Proceedings of the 38th IEEE Conference on Decision and Control (Cat. No. 99CH36304)*. Vol. 1. IEEE. 1999, pp. 487–492.

[34] Albert Gu, Karan Goel, and Christopher Ré. "Efficiently modeling long sequences with structured state spaces". In: *arXiv preprint arXiv:2111.00396* (2021).

[35] Tingzhao Fu et al. “Optical neural networks: progress and challenges”. In: *Light: Science & Applications* 13.1 (2024), p. 263.

[36] Antonio Orvieto et al. “Resurrecting recurrent neural networks for long sequences”. In: *International Conference on Machine Learning*. PMLR. 2023, pp. 26670–26698.

[37] Albert Gu et al. “Combining recurrent, convolutional, and continuous-time models with linear state space layers”. In: *Advances in neural information processing systems* 34 (2021), pp. 572–585.

[38] Jimmy TH Smith, Andrew Warrington, and Scott W Linderman. “Simplified state space layers for sequence modeling”. In: *arXiv preprint arXiv:2208.04933* (2022).

[39] Dan Hendrycks and Kevin Gimpel. “Gaussian error linear units (gelus)”. In: *arXiv preprint arXiv:1606.08415* (2016).

[40] Albert Gu and Tri Dao. “Mamba: Linear-time sequence modeling with selective state spaces”. In: *arXiv preprint arXiv:2312.00752* (2023).

[41] Roland Tóth. *Modeling and identification of linear parameter-varying systems*. Vol. 403. Springer, 2010.

[42] Tri Dao and Albert Gu. “Transformers are ssms: Generalized models and efficient algorithms through structured state space duality”. In: *arXiv preprint arXiv:2405.21060* (2024).

[43] Mohamed G Elfeky, Walid G Aref, and Ahmed K Elmagarmid. “Periodicity detection in time series databases”. In: *IEEE Transactions on Knowledge and Data Engineering* 17.7 (2005), pp. 875–887.

[44] Tom Puech et al. “A fully automated periodicity detection in time series”. In: *Advanced Analytics and Learning on Temporal Data: 4th ECML PKDD Workshop, AALTD 2019, Würzburg, Germany, September 20, 2019, Revised Selected Papers 4*. Springer. 2020, pp. 43–54.

[45] Michael J Evans and Jeffrey S Rosenthal. *Probability and statistics: The science of uncertainty*. Macmillan, 2004.

[46] Charles R. Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[47] URL: https : / / stackoverflow . com / questions / 47351483 / autocorrelation-to-estimate-periodicity-with-numpy.

[48] URL: https://github.com/NicolasZucchet/minimal-LRU.

[49] URL: https://github.com/state-spaces/mamba.

[50] Adam Paszke et al. "Automatic differentiation in PyTorch". In: *NIPS-W*. 2017. URL: https://pytorch.org.

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

[52] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[53] Kaiming He et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.

[54] Ilya Loshchilov and Frank Hutter. "Sgdr: Stochastic gradient descent with warm restarts". In: *arXiv preprint arXiv:1608.03983* (2016).

[55] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *International conference on machine learning*. pmlr. 2015, pp. 448–456.

Acknowledgements

I would like to express my appreciation towards the BME Motorsport Formula Student Racing Team for providing us the raw data, for the validation checks they did on the labels, and for their helpful insights into our feature selection mechanism. Special thanks to my team member Dóra Belkovics for her huge help with understanding the features, and Gergely Szűcs for proposing us the problem and his useful comments. I would like to also thank the Artificial Intelligence Laboratory of HUN-REN SZTAKI for providing us the computational resources. Special thanks to my family for their continuous support. I would also like to express my huge appreciation towards my supervisor Dániel Rácz for the joint work throughout the year, the great help, useful comments and the amount of effort he put in this project.

This work was supported by the European Union project RRF-2.3.1-21-2022-00004 within the framework of the Artificial Intelligence National Laboratory Program.

Statement on the usage of artificial intelligence tools

I, the undersigned, declare that during the preparation of my thesis I used the following AI-based tools for the tasks listed below:

Task	**Tool used**	**Usage location**	**Notes**
Literature search	Google Gemini deep research	Chapter 2	Finding related works
Helping to format tables and figures	GPT-4o-mini	Chapters 3, 4 and 5	
Code completion	GitHub co-pilot	Chapter 5	Help with finishing code snippets and inserting repeated lines
LaTex code refinement	GPT-4o-mini	Whole thesis	Help with writing complicated equations into LaTex code
Grammar check	GPT-4o-mini	Whole thesis	

Table 6.1: AI-based tools used during thesis preparation

I did not use any AI-based tools other than those listed above.

Appendix A

Additional code snippets

Code snippets we used in our work that were referenced in the main text are displayed here.

```
class DeepMamba(nn.Module):

    def __init__(self, d_model, d_state, d_conv, expand, pooling,
        num_layers, n_classes, use_residual=True):
        super(DeepMamba, self).__init__()
        layers = []
        # First Mamba layer (no residual connection needed)
        layers.append(Mamba(d_model, d_state, d_conv, expand))
        self.layer_norm = nn.LayerNorm(d_model)
        # Add remaining layers with optional residual connections
        for _ in range(num_layers - 1):
            block = []
            block.append(nn.LayerNorm(d_model))
            block.append(Mamba(d_model, d_state, d_conv, expand))
            layers.append(nn.Sequential(*block))
        self.mamba_layers = nn.Sequential(*layers)
        self.pooling = Pooling(pooling) # Pooling Layer
        self.fc = nn.Linear(d_model, n_classes) # FC Output Layer

    def forward(self, x, use_residual=True):
        # Apply Mamba layers
        for layer in self.mamba_layers:
            residual = x  # Store input as residual
            if use_residual == "normed":
                residual = self.layer_norm(residual) #normalizing
```

```
            if use_residual or use_residual == "normed" or
                use_residual == "True":
                x = x + residual  # Add residual connection
        x = self.pooling(x)
        x = self.fc(x)
        return x
```

Code A.1: Implementation of the multilayer Mamba model in Python

```

class FuelDataset(Dataset):
    _name_ = "fuel"
    scaler = None  # Class-level scaler shared across instances

    def __init__(self, data_dir, type, scale=None, omit_fpp=False):
        assert type in ['train', 'test', 'validation'],
        csv_file = f'{data_dir}/{type}.csv'
        self.df = pd.read_csv(csv_file)
        if omit_fuel:
            #remove the fuel columns from the dataset
            self.df = self.df.drop(columns=['Fuel load primary
                [273]'])
            self.df = self.df.drop(columns=['Fuel duty primary [64]
                '])
            self.df = self.df.drop(columns=['Fuel pulse primary
                [63]'])
        # Initialize tensors for data and labels
        self.data = torch.empty((self.__len__(), self.__seq_len__()
            , self.__d_input__()))
        self.labels = torch.empty((self.__len__(), 1))
        # Populate data and labels tensors
        for i, id in enumerate(self.df['id'].unique()):
            self.data[i] = torch.tensor(self.df[self.df['id'] == id
                ].iloc[:, 2:].values, dtype=torch.float32)
            self.labels[i] = torch.tensor(self.df[self.df['id'] ==
                id].iloc[0, 1], dtype=torch.float32)
        # Handle scaling
        if scale is not None and scale != 'None':
            if FuelDataset.scaler is None:
                if scale == 'MinMaxScaler':
                    FuelDataset.scaler = MinMaxScaler()
```

```
27                 elif scale == 'StandardScaler':
28                     FuelDataset.scaler = StandardScaler()
29                 elif scale == 'RobustScaler':
30                     FuelDataset.scaler = RobustScaler()
31                 elif scale == 'MAX':
32                     pass
33                 elif scale == 'max':
34                     pass
35                 else:
36                     raise ValueError("Invalid scaler type")
37             if scale == 'MAX':
38                 #divide every data point by the maximum value
39                 for i in range(self.__len__()):
40                     for j in range(self.__d_input__()):
41                         self.data[i][:, j] = self.data[i][:, j] /
                            12437
42             elif scale == 'max':
43                 # Divide every data point by the max value of that
                    dimension
44                 for i in range(self.__len__()):
45                     for j in range(self.__d_input__()):
46                         max_val = torch.max(self.data[i][:, j])
47                         if max_val != 0:
48                             self.data[i][:, j] = self.data[i][:, j]
                                / max_val
49             else:
50                 if type == 'train':
51                     # Fit and transform train data
52                     for i in range(self.__len__()):
53                         for j in range(self.__d_input__()):
54                             self.data[i][:, j] = torch.tensor(
55                                 FuelDataset.scaler.fit_transform(
                                    self.data[i][:, j].reshape(-1,
                                    1)).reshape(-1)
56                             )
57                 elif type == 'test':
58                     # Transform test data using the scaler fitted
                        on train data
59                     print(f"Scaling test data using {FuelDataset.
                        scaler}")
60                     if not FuelDataset.scaler:
```

```
                raise ValueError("Scaler is not fitted yet.
                    Fit the scaler with the train dataset
                    first.")
            for i in range(self.__len__()):
                for j in range(self.__d_input__()):
                    self.data[i][:, j] = torch.tensor(
                        FuelDataset.scaler.transform(self.
                            data[i][:, j].reshape(-1, 1)).
                            reshape(-1)
                    )
        # Dataset properties
        self.seq_len = self.__seq_len__()
        self.d_output = self.__d_output__()
        self.d_input = self.__d_input__()

    def __getitem__(self, index):
        sample = self.data[index], self.labels[index]
        return sample

    def __len__(self):
        # Number of unique IDs in the dataset
        return len(self.df['id'].unique())

    def __d_input__(self):
        # Number of columns excluding the ID and label
        return self.df.shape[1] - 2

    def __d_output__(self):
        # Number of output dimensions classification is used
        return 6

    def __seq_len__(self):
        # Maximum sequence length per unique ID
        return self.df['id'].value_counts().max()
```

Code A.2: The implementation of our dataset in Python

```

class BootstrapDataset(Dataset):
    def __init__(self, dataset, target_labels):
        """
```

```
        Args:
            dataset: The original dataset to wrap.
            target_labels: List of labels to use for bootstrapping.
        """
        self.dataset = dataset
        self.target_labels = set(float(x) for x in target_labels)
        self.label_to_indices = self._create_label_index_map()

    def _create_label_index_map(self):
        """
        Create a map of label to indices for fast lookup.
        """
        label_to_indices = {}
        for idx in range(len(self.dataset)):
            _, label = self.dataset[idx]
            key = label.item() if isinstance(label, torch.Tensor)
                else float(label)
            if key not in label_to_indices:
                label_to_indices[key] = []
            label_to_indices[key].append(idx)
        return label_to_indices

    def _get_random_sample_by_label(self, label):
        """
        Fetch a random sample index with the specified label.
        """
        return random.choice(self.label_to_indices[label])

    def __getitem__(self, index):
        """
        Get the item at index. If index exceeds dataset length,
            return a bootstrap sample.
        """
        if index < len(self.dataset):
            return self.dataset[index]
        # Generate a bootstrap sample, select a random target label
        label = random.choice(list(self.target_labels)
        sample_index = self._get_random_sample_by_label(label)
        return self.dataset[sample_index]

    def __len__(self):
```

```
        """
        Extended length to include original + bootstrap samples.
        """
        return len(self.dataset) * 2  # Example: double the data
            size by adding bootstrap samples
```

Code A.3: The implementation of our dataset in Python

```

class RegressionModel(nn.Module):
    """Stacked encoder with pooling and softmax"""

    lru: nn.Module
    d_output: int # output dimension of the last decoder
    d_model: int # output dimension of the initial encoder
    n_layers: int
    dropout: float = 0.0
    training: bool = True
    norm: str = "batch"  # type of normaliztion
    multidim: int = 1  # number of outputs
    use_decoder: bool = True
    use_encoder: bool = True
    use_pooling: bool = True

    def setup(self):
        self.encoder = StackedEncoderModel(
            lru=self.lru,
            d_model=self.d_model, #output dimension of the initial
                encoder
            n_layers=self.n_layers,
            dropout=self.dropout,
            training=self.training,
            norm=self.norm,
        )
        self.decoder = None
        if self.use_decoder:
            self.decoder = nn.Dense(self.d_output,
                                    use_bias=False)

    def __call__(self, x):
        x = self.encoder(x)
```

```
        if self.use_decoder:
            x = self.decoder(x)
        if self.use_pooling:
            x = jnp.mean(x, axis=0)  # mean pooling across time
        return x
```

Code A.4: The regression model for the LRU architecture in Python

```

class TransformerModel(nn.Module):
    def __init__(self, input_dim, model_dim, num_heads, num_layers,
        num_classes=1):
        super(TransformerModel, self).__init__()
        self.encoder = nn.Linear(input_dim, model_dim)
        encoder_layer = nn.TransformerEncoderLayer(
            d_model=model_dim,
            nhead=num_heads,
            dim_feedforward=2048,
            dropout=0.1,
            activation='relu',
            batch_first=True
        )
        self.transformer_encoder = nn.TransformerEncoder(
            encoder_layer, num_layers=num_layers, norm=nn.LayerNorm(
            model_dim))
        self.pooling = Pooling("mean")
        self.fc1 = nn.Linear(model_dim, 128)
        self.fc2 = nn.Linear(128, num_classes)

    def forward(self, x):
        x = self.encoder(x)
        x = self.transformer_encoder(x)
        x = self.pooling(x)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Code A.5: The Transformer model used in the paper, implemented in Python

Appendix B

Additional definitions

B.1 Physics based naive approach

First, for each time step i we read from the CSV file the commanded injector pulse width τ_i in milliseconds and the engine speed RPM_i. We convert the pulse width to seconds,

$$\tau_i \;=\; \frac{\text{Fuel pulse primary [63]}_i}{1000}$$

and compute the number of injection events per second for a four-stroke engine as

$$E_i \;=\; N_{\text{inj}} \;\times\; \frac{\mathrm{RPM}_i}{120},$$

where N_{inj} is the number of injectors (4 in our case). The instantaneous fuel flow rate (in cm^3/s) is then

$$\dot{V}_i \;=\; \tau_i \;\times\; \left(\frac{R_{\text{inj}}}{60}\right) \;\times\; E_i,$$

where R_{inj} is the injector's calibrated flow rate in $\mathrm{cm}^3/\mathrm{min}$ (240 for our engine). Finally, we sum the fuel delivered over all time steps by multiplying by the fixed time increment Δt (in seconds) and convert cubic centimeters to liters:

$$V_{\text{total}} \;=\; \sum_i \Big(\dot{V}_i \,\Delta t\Big) \quad\Longrightarrow\quad V_{\text{total}}\,(\mathrm{L}) \;=\; \frac{1}{1000}\sum_i \big(\dot{V}_i \,\Delta t\big).$$

This yields the total fuel consumed in liters for the lap.

B.2 L1/L2 loss

L1 Loss (Mean Absolute Error). The L1 loss, also known as mean absolute error (MAE), measures the average absolute difference between predicted values $\hat{y}_i$ and true values y_i over a dataset of size N:

$$L_1 = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|.$$

This loss is less sensitive to outliers than L2 loss and encourages sparse residuals.

L2 Loss (Mean Squared Error). The L2 loss, or mean squared error (MSE), computes the average of the squared differences between $\hat{y}_i$ and y_i:

$$L_2 = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2.$$

L2 penalizes larger errors more heavily.

B.3 ReLU

The *Rectified Linear Unit (ReLU)* activation function is defined element-wise as:

$$\text{ReLU}(x) = \max(0, x).$$

It introduces non-linearity while mitigating vanishing gradients by allowing unbounded positive outputs and zeroing negatives.

B.4 Learning Rate, Scheduler and Optimizer

Learning Rate The learning rate η controls the step size at each iteration of optimization when updating model parameters.

Scheduler A learning-rate scheduler automatically adjusts η during training according to a predefined policy (e.g., step decay, cosine annealing) to improve convergence and avoid local minima.

Optimizers The Optimizer we used is the Adam optimizer [52]. This maintains adaptive estimates of first (m_t) and second (v_t) moments of gradients to compute parameter updates:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t, \quad v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2,$$
$$\hat{m}_t = m_t/(1-\beta_1^t), \quad \hat{v}_t = v_t/(1-\beta_2^t),$$
$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}.$$

B.5 Cross-Entropy Loss

For classification with C classes and true one-hot label y and predicted probability distribution $\hat{p}$, the cross-entropy loss is:

$$L_{CE} = -\sum_{i=1}^{C} y_i \log(\hat{p}_i).$$

In binary classification, this reduces to:

$$L_{BCE} = -\big(y \log(\hat{p}) + (1-y)\log(1-\hat{p})\big).$$

Cross-entropy encourages the model to assign high probability to the correct class.

B.6 He Initialization

He normal initialization [53] sets each weight w in a layer with n_{in} inputs by drawing

$$w \sim \mathcal{N}\big(0, \tfrac{2}{n_{\text{in}}}\big).$$

This preserves variance through ReLU layers.

B.7 Cosine Annealing Scheduler

A *cosine annealing* learning-rate schedule introduced in [54] decays the learning rate η from an initial η_0 toward zero over T epochs according to

$$\eta(t) = \frac{\eta_0}{2}\Big(1 + \cos\frac{\pi t}{T}\Big), \quad t = 0, 1, \ldots, T.$$

B.8 Batch Normalization

Batch normalization [55] normalizes each layer's pre-activation batchwise to zero mean and unit variance, then applies learnable scale and shift:

$$\hat{x}_i = \frac{x_i - \mu_{\text{batch}}}{\sqrt{\sigma^2_{\text{batch}} + \varepsilon}}, \quad y_i = \gamma \hat{x}_i + \beta.$$

B.9 Mean Pooling

Mean pooling aggregates a sequence $(y_1, \ldots, y_T)$ into a fixed-size vector by

$$\bar{y} = \frac{1}{T} \sum_{t=1}^{T} y_t.$$

We use it to collapse time-dependent features before the final regression head.