O'REILLY®

Hands-On Machine Learning with Scikit-Learn and PyTorch

Concepts, Tools, and Techniques to Build Intelligent Systems

Aurélien Géron

Online Bonus Content for Hands-On Machine Learning with Scikit-Learn and PyTorch

Aurélien Geron

O'REILLY®

Hands-On Machine Learning with Scikit-Learn and PyTorch
by Aurélien Geron

Published by O'Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*https://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Nicole Butterfield
Develeopment Editor: Michele Cronin
Production Editor: Beth Kelly
Copyeditor: Sonia Saruba
Proofreader: Kim Cofer
Indexer: Potomac Indexing LLC
Cover Designer: Susan Brown
Cover Illustrator: José Marzan Jr.
Interior Designer: David Futato
Interior Illustrator: Kate Dullea

October 2025: First Edition

Revision History for the First Edition
2025-10-22: First Release

See *https://oreilly.com/catalog/errata.csp?isbn=9798341607989* for release details.

979-8-341-60798-9

[LSI]

APPENDIX E
State-Space Models (SSMs)

State-space models (SSMs) were proposed in 1960 by Rudolf E. Kalman,[1] but they have only recently become popular in deep learning. This led to several SSM-based neural network architectures that rival Transformers in both speed and quality, especially for very long input sequences: at inference time, these models scale linearly with the input sequence length, not quadratically like transformers, yet they manage to detect long-distance patterns over the inputs.

SSM-based neural networks are generally just called SSMs, but keep in mind that the actual SSM (in the traditional sense) is just one component of the neural network: as we will see, it serves as a memory storage and can replace attention layers (although some hybrid models combine both SSMs and attention layers).

In this appendix, we will discuss the following topics:

Traditional SSMs (1960s)
: Traditional SSMs are at the core of all modern SSM-based neural networks. An SSM models a dynamical system whose state evolves over time as a function of past states and inputs. A continuous-time SSM deals with continuous inputs (e.g., soundwaves), and it's a purely mathematical tool. However, it can be converted into a discrete-time SSM (e.g., processing discrete sound samples) with similar dynamics: in this form, the SSM is just a recurrent neural network (RNNs, introduced in Chapter 13 of the book).[2] SSM-based neural nets use *linear* discrete SSMs, where the state evolves as a linear function of the previous state and current inputs.

1 Rudolf E. Kalman, "A New Approach to Linear Filtering and Prediction Problems", Transactions of the ASME – Journal of Basic Engineering, 82(Series D): 35–45 (1960).

2 If you don't have the book, you can get it at *https://homl.info/*.

Viewing functions as vectors

To truly understand the next three topics, we will need to take a little mathematical detour through function space. We will see that functions can be viewed as vectors: you can add them, scale them, use the inner product—a generalization of the dot product—to measure a function's norm (i.e., its length), check whether two functions are orthogonal, and project a function onto another function. This will allow us to describe any continuous function as a weighted sum of basis functions, just like a 3D vector can be described as a weighted sum of basis vectors, for example $\mathbf{v} = 3\mathbf{x} - 2\mathbf{y} + 5\mathbf{z}$. The weights in this sum define the coordinates of vector **u**: (3, –2, 5). Similarly, a function's coordinates in function space are the weights of each basis function in the weighted sum. Why does this matter? Well, in the modern SSMs we will discuss, the state vector contains the coordinates in function space of an approximation of the input function (or a part of it).

The Legendre Memory Unit (2019)

The Legendre Memory Unit (LMU) is an SSM-based neural network whose SSMs are designed to have an excellent memory of the last w inputs (where w is the window size). The parameters of the SSM are not trained: they are entirely derived mathematically and remain constant. The LMU SSMs constantly update a state vector of fixed size d_{state}, ensuring that it can be used to reconstruct an approximation of the last w inputs.

The HiPPO framework (2020)

HiPPO is a theoretical framework that generalizes the ideas from the LMU, allowing for more flexibility, such as giving more weight to recent inputs, or applying uniform weights to *all* past inputs, not just a sliding window like the LMU.

The Convolutional Representation of LTI SSMs (1960s)

An LTI SSM can be converted from its usual RNN representation to a convolutional representation (convolutional neural networks—CNNs—were introduced in Chapter 12). In this convolutional form, the SSM can be parallelized along the sequence length (great during training) and it only requires $O(L \cdot \log(L))$ computations, as opposed to $O(L^2)$ for transformers (where L is the sequence length). At inference time, the RNN representation can be used: it scales with $O(L)$ as opposed to $O(L^2)$ for transformers.

The S4 model (2021)

S4 was the first SSM-based neural network to outperform transformers on tasks involving long-range dependencies. It is based on the HiPPO framework and uses advanced mathematical tricks—including the convolutional representation—to make the SSM both trainable and extremely efficient.

The Mamba model (2023)

The Mamba model is an even more powerful SSM-based neural network, which gives the SSM the ability to selectively remember some inputs and forget others (whereas the previous SSM-based models treated all inputs equally). This change makes most of the mathematical tricks in the LMU, HiPPO, and S4 unusable, so you can choose to skip the LMU, HiPPO, and S4 sections if you only want to learn about Mamba. The SSMs in Mamba are linear RNNs whose weights are input-dependent. Since they are linear, it's possible to use a parallelized algorithm that can run them in $O(\log(L))$ time during training (performing $O(L \cdot \log(L))$ computations in parallel). Mamba also proposed an I/O-aware implementation which reduces memory transfers within the GPU (like FastAttention does, as we saw in Chapter 17),[3] allowing Mamba to run slightly faster than S4.

Apart from the Mamba section, this is a math-heavy appendix. Don't worry, it's not as difficult as it seems. I will do my best to keep things as simple as I possibly can, and it will give you a much deeper understanding of SSMs.

Let's get started! First, we will dive into traditional SSMs.

Traditional SSMs

A traditional SSM describes a dynamical system using state variables, and models its evolution over time, based on its past states and inputs. For example, an SSM may describe how a car's state (e.g., its speed) evolves over time given both its current state and some inputs (e.g, sensor readings from the wheels, accelerator, and brakes). The SSM's output is also based on the state and the inputs (e.g., the output could be the throttle command for the cruise control system). SSMs are an important part of *control theory*, a field that develops models and algorithms to regulate dynamical systems.

The most common SSM is the *linear time-invariant* (LTI) SSM, defined by Equation E-1:

Equation E-1. Linear time-invariant SSM

$$\mathbf{h}'(t) = \mathbf{A}\mathbf{h}(t) + \mathbf{B}\mathbf{u}(t)$$
$$\mathbf{y}(t) = \mathbf{C}\mathbf{h}(t) + \mathbf{D}\mathbf{u}(t)$$

3 Chapter 17 is available online for free at *https://homl.info/*.

- $\mathbf{h}(t)$ is the system's state vector at time t (e.g., speed at time t). Its dimensionality is d_{state}.
- $\mathbf{h}'(t)$ is the derivative of the state vector with respect to time (e.g., acceleration at time t). Its dimensionality is also d_{state}.
- $\mathbf{u}(t)$ is the input vector at time t (e.g., wheel, accelerator, and brake sensor inputs at time t). Its dimensionality is d_{input}, which may be different from d_{state}.
- $\mathbf{y}(t)$ is the output vector at time t (e.g., the throttle command for the cruise control system). Its dimensionality is d_{output}, which may be different from d_{state} and d_{input}.
- The matrices $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, $\mathbf{D}$ are the model parameters; they don't change over time. $\mathbf{A}$ models how the current state evolves over time. $\mathbf{B}$ models how the inputs affect the state. $\mathbf{C}$ models how the output is derived from the state. Lastly, $\mathbf{D}$ models how the inputs directly influence the output (this term is often set to zero).
- The shapes of $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, $\mathbf{D}$ are respectively $d_{\text{state}} \times d_{\text{state}}$, $d_{\text{state}} \times d_{\text{input}}$, $d_{\text{output}} \times d_{\text{state}}$, and $d_{\text{output}} \times d_{\text{input}}$.

The first equation relates the function $\mathbf{h}(t)$ with its derivative $\mathbf{h}'(t)$, which makes it a Differential Equation (DE). More precisely, since there's a single independent variable t, it is an Ordinary Differential Equation (ODE).

As you can see, this model only contains linear terms, and since the matrices are constant, the system's behavior is time-invariant (hence the name).

Now there's a problem: computers can only process discrete data, not continuous inputs, so how can we actually implement a continuous-time LTI SSM? Well, we can't. It's a purely mathematical tool: we can design a continuous-time LTI SSM mathematically, but when we want to run it, we must first convert it to a discrete-time LTI SSM (see Equation E-2 and Figure E-1). As a bonus, this also opens the door to discrete data, such as text tokens.

Equation E-2. Discrete LTI SSM

$$\mathbf{h}_k = \overline{\mathbf{A}}\mathbf{h}_{k-1} + \overline{\mathbf{B}}\mathbf{u}_k$$
$$\mathbf{y}_k = \overline{\mathbf{C}}\mathbf{h}_k + \overline{\mathbf{D}}\mathbf{u}_k$$

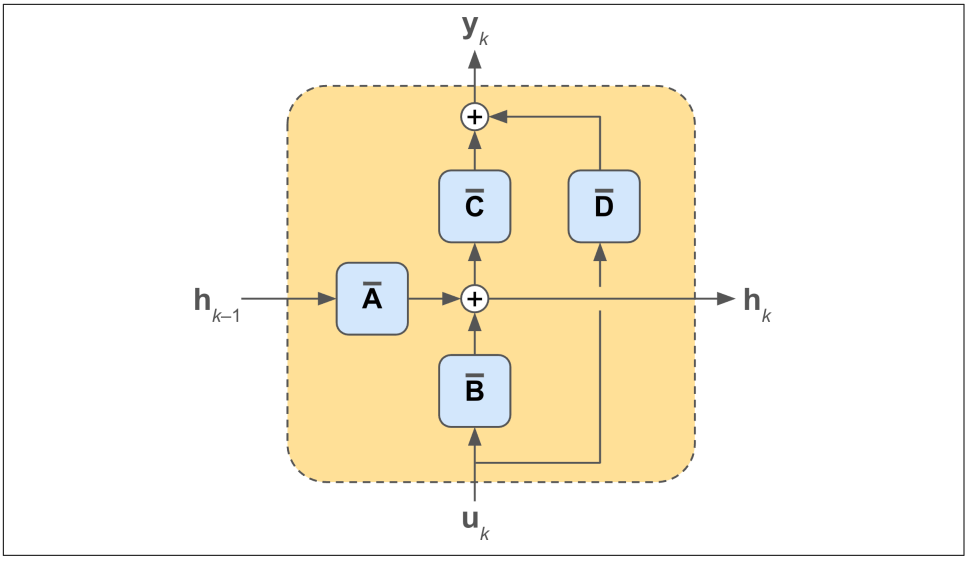

Figure E-1. Discrete-time LTI SSM[4]

This is just like the continuous-time LTI SSM, but time is indexed using steps of size Δ, so $t = k\Delta$, and we are using a different set of matrices $\overline{\mathbf{A}}$, $\overline{\mathbf{B}}$, $\overline{\mathbf{C}}$, $\overline{\mathbf{D}}$. There are various ways to convert a continuous-time LTI SSM to a discrete-time LTI SSM with approximately the same dynamics—and vice versa. In particular, if we make the simplifying assumption that the continuous inputs are constant between discrete time steps, which is called *Zero-Order Hold* (ZOH, see Figure E-2), then it's possible to derive the equations in Table E-1.

Table E-1. Converting between a continuous-time LTI SSM and a discrete-time LTI SSM using Zero-Order Hold

Continuous to discrete	Discrete to continuous
$\overline{\mathsf{A}} = \exp(\Delta\mathsf{A})$	$\mathsf{A} = \frac{1}{\Delta}\log(\overline{\mathsf{A}})$
$\overline{\mathsf{B}} = \mathsf{A}^{-1}(\exp(\Delta\mathsf{A}) - \mathsf{I})\mathsf{B}$	$\mathsf{B} = \mathsf{A}^{-1}(\overline{\mathsf{A}} - \mathsf{I})\overline{\mathsf{B}}$
$\overline{\mathsf{C}} = \mathsf{C}$	$\mathsf{C} = \overline{\mathsf{C}}$
$\overline{\mathsf{D}} = \mathsf{D}$	$\mathsf{D} = \overline{\mathsf{D}}$

4 This is the modern RNN-like formulation used in SSM-based neural networks. In the traditional SSM formulation, the state vector is shifted by one time-step in the first equation: $\mathbf{h}_{k+1} = \overline{\mathbf{A}}\mathbf{h}_k + \overline{\mathbf{B}}\mathbf{u}_k$. This makes no difference to the system's dynamics (it just shifts the state vector's index by one time step), but it *does* affect the SSM's output slightly if matrix $\overline{\mathbf{D}}$ is non-zero.

- $\mathbf{A}^{-1}$ is the inverse of matrix $\mathbf{A}$, which you can compute using `torch.linalg.inv(A)`. This assumes that $\mathbf{A}$ is invertible (otherwise you have to solve an integral).
- $\mathbf{I}$ is the identity matrix , which can be created using `torch.eye(A.size(0))`
- $\exp(\mathbf{A})$ is the *matrix exponential* of $\mathbf{A}$ (*not* the item-wise exponential). You can compute it using `torch.matrix_exp(A)`. The definition of $\exp(x)$ for real numbers is $\exp(x) = 1 + x + x^2/2! + x^3/3! + \ldots$, where $n!$ is the factorial of n. This definition can be extended to matrices: the matrix exponential of matrix $\mathbf{A}$ is defined as $\exp(\mathbf{A}) = \mathbf{I} + \mathbf{A} + \mathbf{A}^2/2! + \mathbf{A}^3/3! + \ldots$, where the powers of $\mathbf{A}$ are the result of repeated matrix multiplications; $\mathbf{A}^2 = \mathbf{AA}$, $\mathbf{A}^3 = \mathbf{AAA}$, and so on (they are *not* item-wise powers). These powers can be computed using `torch.matrix_power(A, power)`.
- $\log(\overline{\mathbf{A}})$ is the *matrix logarithm* (again, *not* the item-wise logarithm): it's the inverse of the matrix exponential.[5] PyTorch does not implement this operation yet.[6] For now, you can use `scipy.linalg.logm(A_discrete)`.

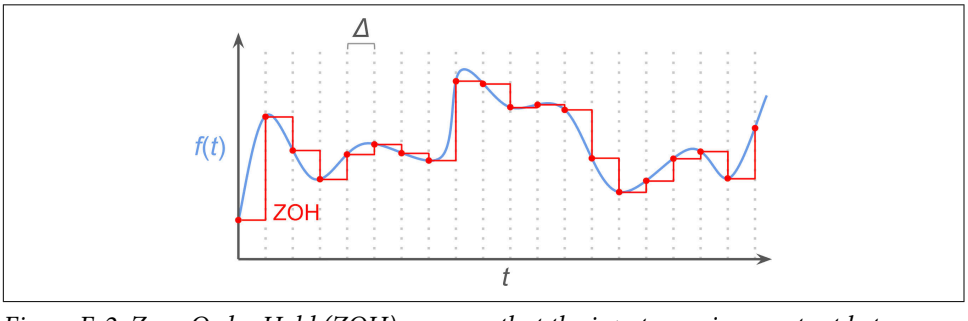

Figure E-2. Zero-Order Hold (ZOH) assumes that the input remains constant between time intervals

Here's a PyTorch implementation of ZOH (see the notebook at *https://homl.info/colab-p* for the discrete-to-continuous implementation):

```
import torch

def zoh(A, B, dt):
    d_state = A.size(-1)
    A_discrete = torch.matrix_exp(A * dt)
    Y = (A_discrete - torch.eye(d_state, device=A.device)) @ B
```

5 The matrix logarithm is not unique: several matrices can exponentiate to the same $\overline{\mathbf{A}}$. Libraries will typically compute the *principal matrix logarithm*, which uniquely selects the version whose eigenvalues have imaginary parts in the range $(-\pi, \pi]$.

6 See *https://github.com/pytorch/pytorch/issues/9983*.

```
    B_discrete = torch.linalg.solve(A, Y)  # == torch.linalg.inv(A) @ Y
    return A_discrete, B_discrete
```

Euler's method, a simple numerical ODE solver, is another common method to discretize an SSM: it approximates the rate of change of the state variables as constant over each time interval: $(\mathbf{h}_{k+1} - \mathbf{h}_k) / \Delta \approx \mathbf{h}'(k\Delta)$, which leads to $\mathbf{h}_{k+1} = (\mathbf{I} + \Delta\mathbf{A})\mathbf{h}_k + \Delta\mathbf{B}\mathbf{u}_k$. So $\overline{\mathbf{A}} = \mathbf{I} + \Delta\mathbf{A}$ and $\overline{\mathbf{B}} = \Delta\mathbf{B}$. This is a much simpler discretization equation, but the result is not as precise, especially for systems with fast dynamics or for large time steps.

You may have noticed that a discrete LTI SSM looks very much like a recurrent neural network (RNN; introduced in Chapter 13): each new state is computed based on the previous state and the inputs at the current time step, and the outputs are computed based on the current state and the inputs. The term $\mathbf{D}\mathbf{x}_k$ ensures that the inputs can directly influence the outputs; you can think of it as a flexible skip connection (not all SSMs include this term). So, yes indeed, a discrete-time LTI SSM is just a linear RNN! Here's a very basic PyTorch implementation, to make things more concrete:

```
import torch.nn as nn

class BasicSSM(nn.Module):
    def __init__(self, d_input, d_state, d_output):
        super().__init__()
        self.d_state = d_state
        self.A = nn.Parameter(torch.randn(d_state, d_state))
        self.B = nn.Parameter(torch.randn(d_state, d_input))
        self.C = nn.Parameter(torch.randn(d_output, d_state))
        self.D = nn.Parameter(torch.randn(d_output, d_input))

    def forward(self, u):
        batch_size, seq_len, _ = u.shape
        h = torch.zeros(batch_size, self.d_state, device=u.device)
        outputs = []
        for i in range(seq_len):
            u_t = u[:, i, :]  # iterate over each element in the sequence
            h = h @ self.A.T + u_t @ self.B.T
            y = h @ self.C.T + u_t @ self.D.T
            outputs.append(y)

        return torch.stack(outputs, dim=1)
```

This implementation is very basic: modern SSMs are much more efficient. Moreover, they often use non-trainable or partially trained matrices, derived mathematically to optimize the SSM's memory, as we will see. In Mamba, the matrices **B**, **C**, and **D** also depend on the inputs.

Now, before we explore modern SSMs, we must first learn to think of functions as vectors. As I mentioned in the introduction of this appendix, the state vector of modern SSMs contains the coordinates of the input function in function space (or rather the coordinates of an approximated and possibly truncated version of the input function). Let's see what all of this means.

Viewing Functions as Vectors[7]

Just like you can add two Euclidean vectors, you can also add two functions together: $(f + g)(x) = f(x) + g(x)$. Just like you can scale a vector, you can also scale a function: $(\alpha f)(x) = \alpha \cdot f(x)$. It turns out that these addition and scaling operations have all the properties required to define a *vector space*, such as associativity ($f + (g + h) = (f + g) + h$) and commutativity ($f + g = g + f$).[8] In other words, functions can be viewed as vectors; they're just not the kind we are used to.

With Euclidean vectors, a very useful operation is the dot product. For example, when the dot product of two vectors is zero, we know that these vectors are orthogonal. And the length (or norm) of a vector $\mathbf{u}$, denoted $\|\mathbf{u}\|$, is equal to the square root of $\mathbf{u} \cdot \mathbf{u}$. Similarly, two functions f and g are defined as orthogonal if their *inner product*, denoted $\langle f, g \rangle$, is equal to zero. Also, the norm of a function f is defined as the square root of $\langle f, f \rangle$.

As you can see, the inner product is a generalization of the dot product to other vector spaces, such as function spaces. There are a few variants you can choose from, depending on the vector space, but the most common inner product for real-valued functions is defined in Equation E-3.

Equation E-3. Standard inner product for a function space

$$\langle f, g \rangle = \int_a^b w(x)\, f(x)\, g(x)\, dx$$

In this equation:

- $\langle f, g \rangle$ is the inner product of functions f and g.
- a and b are the bounds we are considering for the inner product, for example $a = 0$ and $b = 1$ for the LMU, or $a = 0$ and $b = +\infty$ for some HiPPO-based SSMs such as HiPPO-LegS, as we will see.

7 Many thanks to Ulf Bissbort who reviewed this appendix and wrote this excellent post, which inspired much of this section.

8 The full list of properties required are listed at *https://homl.info/vectors*.

- $w(x)$ is a weight function. In the LMU, it is defined as $w(x) = 1$ for all x: this gives equal weight to every point in the interval $[0, 1]$. In SSMs based on HiPPO-LegS, we will use $w(x) = \exp(-x)$, giving more weight to the most recent input signal.
- The integral of some function f over a range $[a, b]$ is denoted $\int_a^b f(x)\,dx$.

If you are not familiar with integrals, you can think of $\int_a^b f(x)\,dx$ as the area under the curve defined by $y = f(x)$ between $x = a$ and $x = b$. For example, $\int_{-4}^{10} x\,dx = \frac{10^2}{2} - \frac{4^2}{2} = 42$. To understand this result, try drawing the line $y = x$ and evaluate the area between the line and the horizontal axis: any area below the horizontal axis must be subtracted rather than added.

So functions are vectors: we can add them and scale them, compute their inner products, measure their norm, and check whether two functions defined over some interval are orthogonal or not. Great, but how does this help? Well, just like we can define a set of basis vectors for a Euclidean space and use these basis vectors to define the coordinates of any vector in this space, we can also define a set of *basis functions* and use this set to define the coordinates of every function in the function space. If the basis functions are orthonormal, meaning orthogonal and of unit length, then you can find the coordinates of a function f by computing the inner product of f with each basis function.

For example, consider the space of polynomial functions, and the set of basis functions p_0, p_1, p_2, …, such that $p_i(x) = x^i$ (these functions are called monomials). The polynomial $q(x) = 2 - 3x + 7x^3$ can be described as a weighted sum of the basis functions: $q = 2p_0 - 3p_1 + 0p_2 + 7p_3 + 0p_4 + 0p_5 + \ldots$, so the coordinates of q in this basis are $(2, -3, 0, 7, 0, 0, \ldots)$. In practice, we cannot store an infinite number of coordinates, so we must truncate the coordinates at some chosen length d. Luckily, any continuous function can be approximated arbitrarily well over a bounded interval (e.g., $[0, 1]$) using a polynomial of sufficient degree—this is the Weierstrass approximation theorem—so if we choose a sufficiently large d, we can use d coordinates to approximate any continuous function over a bounded interval.

However, the monomials are not a good choice of basis functions. Indeed, although they are linearly independent—a fundamental requirement for any set of basis functions—they are not orthogonal over any bounded interval (the inner product is nonzero for any two distinct monomials). As a result, computations in this basis are numerically unstable: the coordinates are highly correlated, rounding errors are amplified, and truncating the expansion (i.e., dropping the last coordinates) can severely distort the approximation. What we need is a set of orthogonal basis functions.

Luckily, many such sets have been discovered, such as the *Legendre polynomials*. This is a sequence of polynomials of increasing degrees (the i^{th} polynomial has degree i) which are all orthogonal to each other over the range [–1, 1]. As we will see, the LMU's SSMs actually use *shifted Legendre polynomials* (see Equation E-4) which are orthogonal over the range [0, 1] instead of [–1, 1]. The first 5 shifted Legendre polynomials are represented in Figure E-3.

Equation E-4. Shifted Legendre polynomials.

$$\mathscr{P}_i(x) = (-1)^i \sum_{j=0}^{i} \binom{i}{j}\binom{i+j}{j}(-x)^j$$

$$\text{where } \binom{n}{k} = \frac{n!}{k!(n-k)!}$$

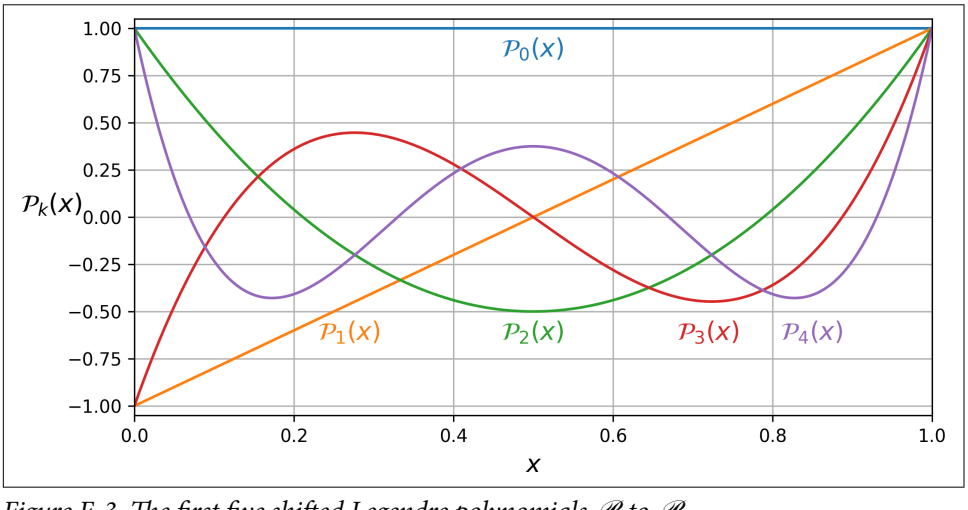

Figure E-3. The first five shifted Legendre polynomials $\mathscr{P}_0$ to $\mathscr{P}_4$

You can use the following function to compute the i^{th} shifted Legendre polynomial; the parameter **x** is a vector containing the points at which you wish to evaluate the polynomial:

```
from scipy.special import binom

def shifted_legendre_polynomial(i, x):
    coeffs = [(-1)**j * binom(i, j) * binom(i + j, i) for j in range(i + 1)]
    coeffs = (-1)**i * torch.tensor(coeffs, dtype=x.dtype)
    powers = x.unsqueeze(-1) ** torch.arange(i + 1, device=x.device)
    return (coeffs * powers).sum(dim=-1)
```

Any continuous function defined over the range [0, 1] can be approximated by using a weighted sum of the first d shifted Legendre polynomials: the larger d is, the more

precise the approximation (but the greater the memory and computational cost). Figure E-4 shows a function $f(x)$ and several approximations computed using a weighted sum of shifted Legendre polynomials. From yellow to red, the approximations use more and more polynomials in their weighted sum, from just one ($\mathscr{P}_0$, so the approximation is just a constant function) to nine ($\mathscr{P}_0$ to $\mathscr{P}_8$, so it's a degree 8 polynomial), skipping a few for readability. You can see how the approximations improve as we add more polynomials to the mix.

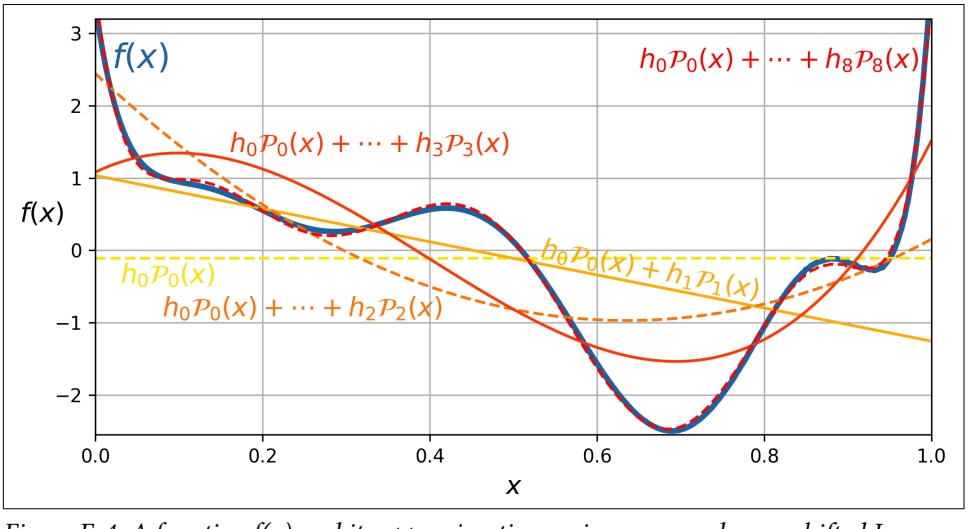

Figure E-4. A function f(x) and its approximations using more and more shifted Legendre polynomials; the term h_i is the weight of the i^{th} polynomial.

Now suppose you have a function u defined on the interval [0, 1], and a d-dimensional vector $\mathbf{h}$ containing the first d coordinates of u in the basis of shifted Legendre polynomials: you can reconstruct an approximation of u using Equation E-5.

Equation E-5. Approximating a continuous function over interval [0, 1] using shifted Legendre polynomials

$$u(x) \approx \sum_{i=0}^{d-1} h_i \mathscr{P}_i(x)$$

In this equation, h_i is the i^{th} coordinate of function u in the basis (i.e., the i^{th} element of vector $\mathbf{h}$), and $\mathscr{P}_i$ is the i^{th} shifted Legendre polynomial.

The following code can be used to compute this approximation. The weight vector $\mathbf{h}$ contains the weights of the first d shifted Legendre polynomials, and the vector $\mathbf{x}$ contains the points at which you wish to evaluate the approximation (between 0 and 1).

```
def reconstructed_function(h, x):
    Pis = [shifted_legendre_polynomial(i, x) for i in range(h.size(0))]
    return (h.unsqueeze(-1) * torch.stack(Pis)).sum(dim=0)
```

Now you know how to go from **h** to an approximation of u, but how can you go in the other direction—that is, how can you find **h** given a function u? Because the shifted Legendre polynomials are all orthogonal to each other, each coordinate h_i of u in this basis can be obtained by projecting u onto the corresponding basis function. This is done using Equation E-6 (note that we have to divide $\langle u, \mathscr{P}_i \rangle$ by $\langle \mathscr{P}_i, \mathscr{P}_i \rangle$ because the shifted Legendre polynomials are not normalized—they don't have unit norm).

Equation E-6. Finding the i^{th} coordinate h_i of a continuous function u defined over [0, 1], in the basis of shifted Legendre polynomials.

$$h_i = \frac{\langle u, \mathscr{P}_i \rangle}{\langle \mathscr{P}_i, \mathscr{P}_i \rangle} = \frac{\langle u, \mathscr{P}_i \rangle}{\frac{1}{2i+1}} = (2i+1) \int_0^1 u(x)\, \mathscr{P}_i(x)\, dx$$

Unfortunately, this integral is generally impossible to solve analytically since the function u is typically based on the SSM's inputs, which could be anything at all. Luckily, we don't need to compute **h** directly; we only need to compute the matrices **A** and **B** of the continuous-time LTI SSM, so that **h** can be updated properly at each instant. In other words, we need to find **A** and **B** such that $\mathbf{h}'(t) = \mathbf{A}\mathbf{h}(t) + \mathbf{B}u(t)$. This requires quite a bit of mathematical ingenuity, and it doesn't always work; in fact, it has only been achieved for a few well-behaved cases, such as with shifted Legendre polynomials over the interval [0, 1] using the standard inner product with $w(t) = 1$ (as in the LMU).

Congratulations on making it through this math section! You now have the main tools you need to understand modern SSMs. Let's start with the LMU.

Legendre Memory Unit (LMU)

Many researchers have attempted to combine traditional SSMs and deep learning for over a decade, particularly in the area of time series forecasting and neuroscience. A breakthrough occurred in 2019 with the *Legendre Memory Unit* (LMU): this is a powerful SSM-based RNN architecture proposed in 2019[9] by Aaron Voelker et al. from the Centre for Theoretical Neuroscience and Applied Brain Research Inc. in Canada. It demonstrated impressive performance on very long sequences.

An LMU's memory is split into a short-term memory **s** and a long-term memory **H** (see Figure E-5), much like an LSTM cell (see Chapter 13), except **H** is a matrix, not a

9 Aaron Voelker et al., "Legendre memory units: Continuous-time representation in recurrent neural networks.", *Advances in neural information processing systems 32* (2019).

vector. At time step k, the LMU's input vector $\mathbf{x}_k$, the flattened matrix $\mathbf{H}_{k-1}$, and the vector $\mathbf{s}_{k-1}$ are passed through a linear layer each, and the three results as summed. The sum is a vector $\mathbf{u}_k$, which is fed to the LMU's SSM module (we will see exactly how it works in a minute). The SSM module has two outputs: a vector $\mathbf{y}_k$, and the updated long-term memory matrix $\mathbf{H}_k$. The vectors $\mathbf{y}_k$, $\mathbf{x}_k$, and $\mathbf{s}_{k-1}$ are then passed through a linear layer each, and the three results are summed. Lastly, the sum is passed through an activation function (e.g., tanh). The final result is the updated short-term memory vector $\mathbf{s}_k$. This vector $\mathbf{s}_k$ also serves as the LMU's output.

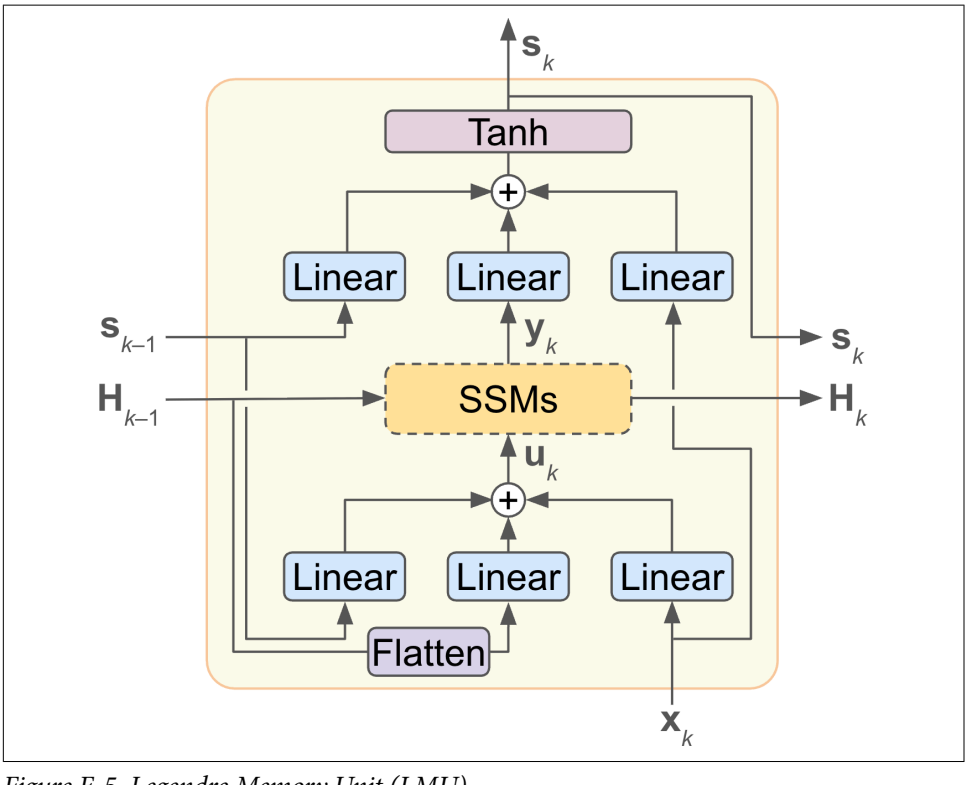

Figure E-5. Legendre Memory Unit (LMU)

I have just represented a single LMU, but you would generally stack multiple LMUs.

Now let's zoom in on the SSMs (see Figure E-6). The LMU actually uses one SSM per input dimension i in $\mathbf{u}_k$ (which I will call channel i). Each SSM has its own state vector $\mathbf{h}^{(i)}$, which corresponds to the i^{th} row of the long-term memory matrix $\mathbf{H}$. Each SSM processes a single channel of the vector $\mathbf{u}_k$, ignoring all other channels. In other

words, at time step k, the i^{th} SSM processes a scalar input $u^{(i)}{}_k$ and updates its state vector from $\mathbf{h}^{(i)}{}_{k-1}$ to $\mathbf{h}^{(i)}{}_k$ using Equation E-2. Each SSM outputs its state vector (implicitly, $\overline{\mathbf{C}}$ is the identity matrix and $\overline{\mathbf{D}}$ is zero). The LMU's output vector $\mathbf{y}_k$ is obtained by concatenating all the SSM state vectors $\mathbf{h}^{(1)}{}_k$ to $\mathbf{h}^{(d_i)}{}_k$, where d_i (short for d_{input}) is the dimensionality of $\mathbf{u}_k$ (i.e., its number of channels, which is also the number of SSMs).

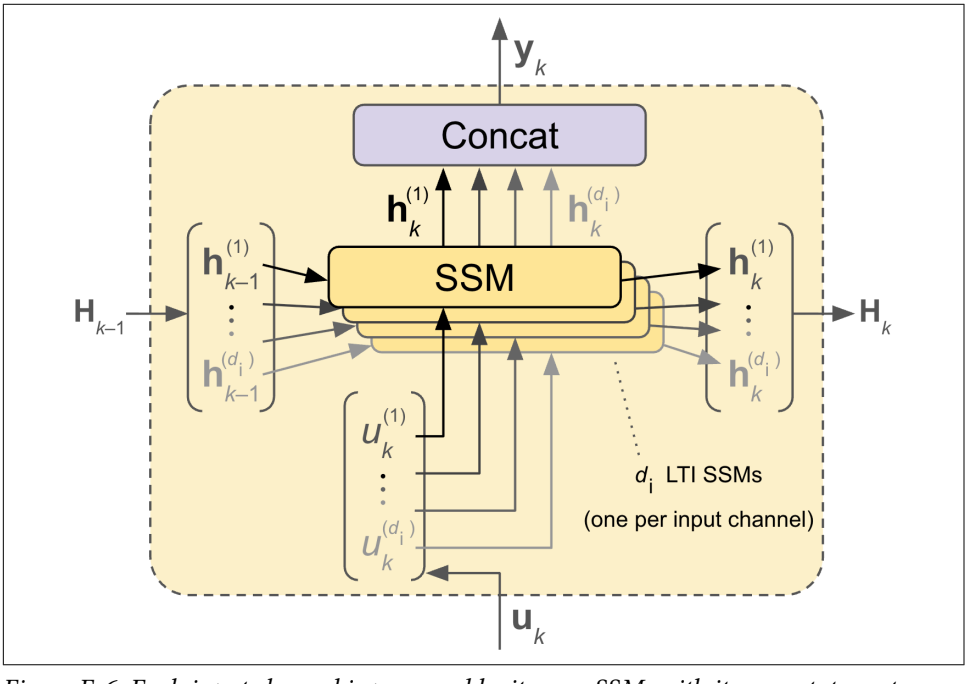

Figure E-6. Each input channel is processed by its own SSM, with its own state vector

Although LTI SSMs are capable of handling multidimensional inputs, the specific LTI SSMs used in modern SSM-based neural networks are designed to process only 1D inputs at each time step, which is why we need one SSM per input channel.

To make this more concrete, here are a few lines of PyTorch code that show how you can simultaneously update all the state vectors across all SSMs and all instances in the batch (`d_input` corresponds to d_i in Figure E-6):

```
batch_size, d_state, d_input = 64, 128, 32

A = torch.randn(1, 1, d_state, d_state)  # state-to-state matrix, shared by SSMs
B = torch.randn(1, 1, d_state, 1)  # input-to-state vector, shared by SSMs
H = torch.randn(batch_size, d_input, d_state, 1)  # state vector per inst. & SSM
u = torch.randn(batch_size, d_input, 1, 1)  # input scalar per instance & SSM
```

```
H = A @ H + B @ u  # update states of all SSMs across all instances, in parallel
y = H.view(batch_size, d_input * d_state)  # concat. outputs for all instances
```

In the rest of this discussion, I will focus on a single SSM at a time, so the input u_k will always be a scalar, but always keep in mind that there are actually multiple SSMs working in parallel, each focusing only on a single input channel.

Since there's a single SSM per input channel and they all work separately, the SSMs cannot capture any pattern *across* the input channels. Luckily, the neural network layers around the SSMs can take care of combining channels.

Each SSM in the LMU is not just any random SSM: it's a *structured SSM*. What is a structured SSM, you ask? It's an SSM whose matrices have special algebraic properties designed to give the SSM desirable features, such as speed, or stability. In the case of the LMU's SSM, the matrices $\overline{\mathbf{A}}$ and $\overline{\mathbf{B}}$ are designed to give the SSM an excellent memory.

You may be wondering how the LMU authors found these ideal matrices $\overline{\mathbf{A}}$ and $\overline{\mathbf{B}}$? Interestingly, they first designed a continuous-time SSM with the desired property (i.e., an excellent memory), then converted it to a discrete-time SSM at the very end using ZOH.

Specifically, they mathematically derived two matrices $\mathbf{A}$ and $\mathbf{B}$ for a continuous-time SSM, such that its state vector $\mathbf{h}(t)$ would always keep track of the coordinates in function space of an approximation of the sliding input window $v_{t,w}(x) = u(t - xw)$, where $0 \leq x \leq 1$, w is the width of the window, and $u(t)$ is the input signal at time t. Note that $v_{t,w}(x)$ is a function defined over the interval $[0, 1]$; $v_{t,w}(0)$ is the input signal at time t, while $v_{t,w}(1)$ is the input signal at time $t - w$. As I mentioned earlier, the coordinates are in the basis of shifted Legendre polynomials, using the standard inner product with $w(t) = 1$.

These matrices are shown in Equation E-7. For more details on the mathematical derivation, please see Aaron Voelker's 2019 PhD thesis (note that a simpler approach was proposed later by the HiPPO authors).

*Equation E-7. Matrices **A** and **B** for a continuous-time LTI SSM with Legendre memory*

$$\mathbf{A} = [a_{ij}] \in \mathbb{R}^{d \times d}, \quad a_{ij} = (2i+1)\begin{cases} -1 & i < j \\ (-1)^{i-j+1} & i \geq j \end{cases}$$

$$\mathbf{B} = [b_i] \in \mathbb{R}^{d \times 1}, \quad b_i = (2i+1)(-1)^i, \quad i,j \in [0, d-1]$$

The fact that the shifted Legendre polynomials are one-dimensional functions is the reason why LMU SSMs operate on one-dimensional inputs. In principle, mathematical tools such as multivariate Legendre expansions could extend this approach to multidimensional inputs, but the number of coefficients (and thus the state size) would grow exponentially with the number of input channels, making such an approach computationally intractable.

And that's it! After converting this continuous-time SSM to a discrete-time SSM using ZOH (Table E-1), you get an SSM with an excellent memory over the sliding input window. While a transformer's multi-head attention (MHA) layer considers all past inputs at each time step, the LMU only needs to look at the previous state and the current input. Therefore, if L is the length of the input sequence, the LMU has a computational complexity of $O(L)$, as opposed to $O(L^2)$ for the transformer. The LMU only remembers a sliding window, not the full input history, but it's possible to make the sliding window quite wide and still get good accuracy.

In conclusion, the LMU showed that a structured SSM could outperform top-tier models on tasks requiring long memory. It also showed that relying on continuous-time representations was a viable alternative to the traditional gating mechanisms used in LSTMs. But this was just the beginning. Next up: HiPPO.

The HiPPO Framework

In August 2020, Albert Gu, Tri Dao, and other researchers from Stanford University and the University at Buffalo, generalized the ideas of LMU into the *High-order Polynomial Projection Operator* (HiPPO) framework.[10] This is not a model, but a formal theory for continuously tracking and reconstructing signals using polynomial projections. HiPPO was later used to build actual models, as we will see shortly.

HiPPO reframes memory as a problem of online function approximation: given a function $u(t)$ over time, the goal is to gradually update a compressed representation $\mathbf{h}$ of the full history (denoted $u_{\leq t}$). At any time t, the representation $\mathbf{h}(t)$ can be used to approximately reconstruct $u_{\leq t}$. Just like in the LMU, the state vector $\mathbf{h}$ contains the weights of orthogonal polynomials, and the ideal transition matrices $\mathbf{A}$ and $\mathbf{B}$ are derived mathematically. However, HiPPO is more flexible; for example, it allows for other sets of orthogonal polynomials, such as Laguerre polynomials—which are orthogonal over the interval [0 to +∞)—leading to different pairs of matrices $\mathbf{A}$ and $\mathbf{B}$.

10 Albert Gu, Tri Dao, et al., "HiPPO: Recurrent Memory with Optimal Polynomial Projections", arXiv preprint arXiv:2008.07669 (2020).

HiPPO only considers single-input single-output (SISO) SSMs, meaning that the inputs and outputs at each time step are just scalars. In contrast, the LMU used single-input multiple-output SSMs, as we saw in Figure E-6. HiPPO-based neural nets contain one SSM per input channel (just like the LMU), and their scalar outputs are concatenated to form the output vector (with the same dimensionality as the inputs). For the rest of this discussion, we will focus on a single HiPPO SSM, with a scalar input $u(t)$, and a scalar output $y(t)$.

Importantly, HiPPO also proposes to evaluate how good an approximation is based on a measure of the importance of each point in time. This measure can give more weight to the recent past, while still paying attention to the distant past. The LMU implicitly uses a uniform measure $w(t) = 1$ over the sliding input window, while ignoring the past beyond the window. In contrast, HiPPO provides more flexibility: it can theoretically capture dependencies across arbitrarily long sequences.

For example, Figure E-7 shows a function $u(t)$, represented with a thin gray line, and the approximate history that a HiPPO-based model can reconstruct at times t_1 (in blue) and t_2 (in red). In this example, we use an exponentially decaying measure (represented at the bottom of the figure for both time steps): this gives more importance to the recent past but still considers the full history. This also explains why the approximations are more accurate for the recent past. As you can see, the reconstructed history keeps changing over time. Also note that the importance of a particular point in time changes over time, typically decreasing.

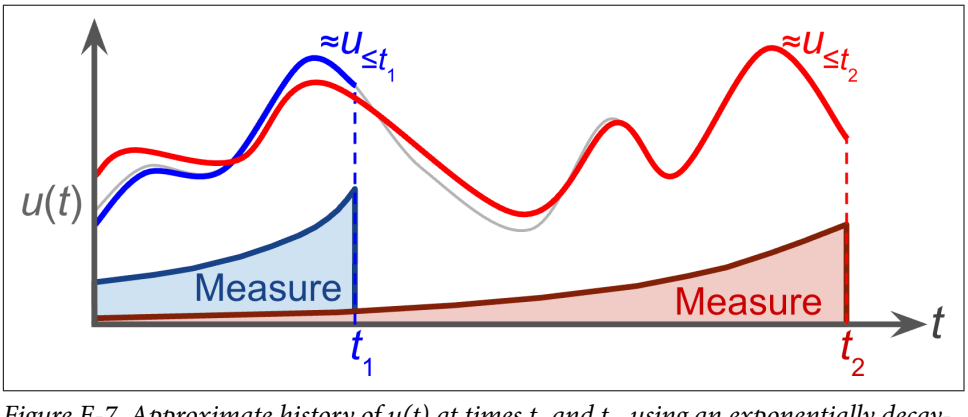

Figure E-7. Approximate history of u(t) at times t_1 and t_2, using an exponentially decaying measure

Just like the LMU authors, the HiPPO authors derived **A** and **B** mathematically (using a simpler approach than for the LMU), and they did so for a few well-behaved cases (see Figure E-8):

- *HiPPO-LegS* (Scaled Legendre): applies a uniform weight to the full history.
- *HiPPO-LegT* (Translated Legendre): assigns a uniform weight over the most recent history; it's a fixed-sized sliding window. This is equivalent to LMU.
- *HiPPO-LagT* (Translated Laguerre): uses an exponentially decaying measure over the full history.

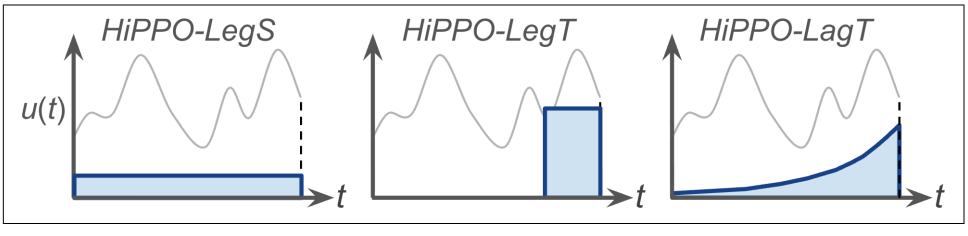

Figure E-8. Three HiPPO instances derived for different measures

The continuous and discrete time dynamics for HiPPO-LegS are shown in Equation E-8. Since the shifted Legendre polynomials are orthogonal over the range 0 to 1, the time needs to be normalized to this range, which is why we divide **A** and **B** by t. This means that we no longer have an LTI SSM, but instead we have a linear time-variant (LTV) SSM. This makes it hard to use ZOH to convert the SSM to discrete time, so the authors used Euler's method instead. The matrices **A** and **B** are shown in Equation E-9.

Equation E-8. Continuous and discrete time dynamics for HiPPO-LegS

$$\mathbf{h}'(t) = \frac{1}{t}\mathbf{A}\mathbf{h}(t) + \frac{1}{t}\mathbf{B}u(t)$$

$$\mathbf{h}_k = \left(1 + \frac{1}{k}\mathbf{A}\right)h_{k-1} + \frac{1}{k}\mathbf{B}u_k$$

*Equation E-9. Matrices **A** and **B** for HiPPO-LegS*

$$A_{nk} = \begin{cases} -\frac{\sqrt{2n+1}\sqrt{2k+1}}{n+1} & \text{if } n > k \\ -(n+1) & \text{if } n = k, \\ 0 & \text{if } n < k \end{cases} \qquad B_n = \sqrt{2n+1}$$

For HiPPO-LegT and HiPPO-LagT, the continuous time dynamics are the classical LTI SSM dynamics (see Equation E-1), and the matrices are shown in Equation E-10

and Equation E-11. You can use ZOH (see Table E-1) to convert the SSM into a discrete-time SSM.

*Equation E-10. Matrices **A** and **B** for HiPPO-LegT*

$$A_{nk} = -\frac{1}{w}\begin{cases}(-1)^{n-k}(2n+1) & \text{if } n \geq k \\ 2n+1 & \text{if } n < k\end{cases}, \quad B_n = \frac{1}{w}(2n+1)(-1)^n$$

*Equation E-11. Matrices **A** and **B** for HiPPO-LagT*

$$A_{nk} = \begin{cases}-1 & \text{if } n \geq k \\ 0 & \text{if } n < k\end{cases}, \quad B_n = 1$$

So the HiPPO recipe is:

- Choose HiPPO-LegS, HiPPO-LegT, or HiPPO-LagT—whichever best fits your task (e.g., HiPPO-LegT if only the recent past matters). Alternatively, you can derive **A** and **B** for your own measure and polynomial basis if you are a math whiz.
- Build a discrete-time LTI SSM using the corresponding matrices $\overline{\mathbf{A}}$ and $\overline{\mathbf{B}}$.
- Use it as a memory storage for your deep neural network, much like LMU did.

In short, HiPPO generalized the ideas of LMU, adding more flexibility with the measure, and offering several powerful off-the-shelf HiPPO matrices.

So far, we have seen that SSMs are great to store a compressed memory of the input history, but that's just half of the story: SSMs can also process long sequences much faster than transformers can, as we will see now.

The Convolutional Representation of LTI SSMs

The fact that an LTI SSM is entirely linear enables a neat mathematical trick known since the 1960s: it's possible to make it process entire sequences in parallel. To see how, let's unroll the SSM's state step by step using $\mathbf{h}_k = \overline{\mathbf{A}}\mathbf{h}_{k-1} + \overline{\mathbf{B}}u_k$. For simplicity, we will assume that the inputs and outputs are scalars, and the state vector is initialized to a zero vector. Let's go:

$$\mathbf{h}_0 = \overline{\mathbf{B}}u_0$$
$$\mathbf{h}_1 = \overline{\mathbf{A}}\mathbf{h}_0 + \overline{\mathbf{B}}u_1 = \overline{\mathbf{A}\mathbf{B}}u_0 + \overline{\mathbf{B}}u_1$$
$$\mathbf{h}_2 = \overline{\mathbf{A}}\mathbf{h}_1 + \overline{\mathbf{B}}u_2 = \overline{\mathbf{A}}(\overline{\mathbf{A}\mathbf{B}}u_0 + \overline{\mathbf{B}}u_1) + \overline{\mathbf{B}}u_2 = \overline{\mathbf{A}}^2\overline{\mathbf{B}}u_0 + \overline{\mathbf{A}\mathbf{B}}u_1 + \overline{\mathbf{B}}u_2$$
$$\cdots$$
$$\mathbf{h}_n = \overline{\mathbf{A}}^n\overline{\mathbf{B}}u_0 + \overline{\mathbf{A}}^{n-1}\overline{\mathbf{B}}u_1 + \ldots + \overline{\mathbf{A}\mathbf{B}}u_{n-1} + \overline{\mathbf{B}}u_n$$

We can now compute the output vector $\mathbf{y}_n = \overline{\mathbf{C}}\mathbf{h}_n$ (assuming matrix $\overline{\mathbf{D}}$ is zero). We find:

$$\mathbf{y}_n = \overline{\mathbf{C}}\overline{\mathbf{A}}^n\overline{\mathbf{B}}u_0 + \overline{\mathbf{C}}\overline{\mathbf{A}}^{n-1}\overline{\mathbf{B}}u_1 + \ldots + \overline{\mathbf{C}}\overline{\mathbf{A}}\overline{\mathbf{B}}u_{n-1} + \overline{\mathbf{C}}\overline{\mathbf{B}}u_n$$

This equation can actually be reformulated as a *discrete convolution operation*. Indeed, if we define the kernel $\mathbf{k} = (\overline{\mathbf{C}}\overline{\mathbf{B}}, \overline{\mathbf{C}}\overline{\mathbf{A}}\overline{\mathbf{B}}, \ldots, \overline{\mathbf{C}}\overline{\mathbf{A}}^{n-1}\overline{\mathbf{B}}, \overline{\mathbf{C}}\overline{\mathbf{A}}^n\overline{\mathbf{B}})$, which is a sequence of L scalars[11] (with $L = n + 1$), and $\mathbf{u}$ is the sequence of L scalar inputs $(u_0, u_1, \ldots, u_n)$, then $\mathbf{y}_n = \mathbf{k} * \mathbf{u}$, where $*$ is the discrete convolution operator (see Equation E-12). Therefore, an LTI SSM can be seen either as a recurrent model, or as a convolutional model.

Equation E-12. Discrete convolution

$$z_i = (\mathbf{m}*\mathbf{n})_i = \sum_j m_j n_{i-j}$$

This dual representation (recurrent and convolutional) makes LTI SSMs efficient both during inference and training. As a recurrent model, it's very fast at inference time, as it only considers the previous state and the current input: generating a sequence of length L has a computational cost of $O(L)$ at inference time, as opposed to $O(L^2)$ for transformers. As a convolutional model, processing a sequence of length L during training requires $O(L \cdot \log(L))$ computations (using an efficient implementation discussed shortly), versus $O(L^2)$ for transformers. Crucially, a convolutional model is parallelizable.

Note that the kernel $\mathbf{k}$ has exactly the same length as the input sequence $\mathbf{u}$. Usually, convolutional neural networks (CNN) uses a small kernel, so it's fairly easy to parallelize, but it's not the case here. Luckily, we can use yet another mathematical trick involving the Fourier transform.

The Fourier Transform

The Fourier transform breaks down a signal (e.g., a sound wave) into its individual frequencies. In other words, it changes the signal's representation from the time domain—showing how the signal's amplitude changes over time—to the frequency domain—showing the amplitude of each individual frequency. For each possible angular frequency ω (i.e., the frequency times 2π), it outputs two amplitudes: one for $\cos(\omega t)$ and one for $\sin(\omega t)$. For example, Figure E-9 shows a function that looks rather complicated in the time domain (top left), but after a Fourier transform, we

11 To be more precise, $\overline{\mathbf{C}}$ is a $d_{\text{output}} \times d_{\text{state}}$ matrix, $\overline{\mathbf{A}}$ is a $d_{\text{state}} \times d_{\text{state}}$ matrix, and $\overline{\mathbf{B}}$ is a $d_{\text{state}} \times d_{\text{input}}$ matrix, so all the elements of $\mathbf{k}$ are $d_{\text{output}} \times d_{\text{input}}$ matrices. In our case, $d_{\text{output}} = d_{\text{input}} = 1$, so each element is a 1×1 matrix, each containing a single scalar.

find that it's just composed of two frequencies (right): one cosine wave of amplitude 1.0 at 3 Hz, and one sine wave of amplitude 0.4 at 7 Hz. These two waves are shown in the lower left plot. The original wave is simply their sum.

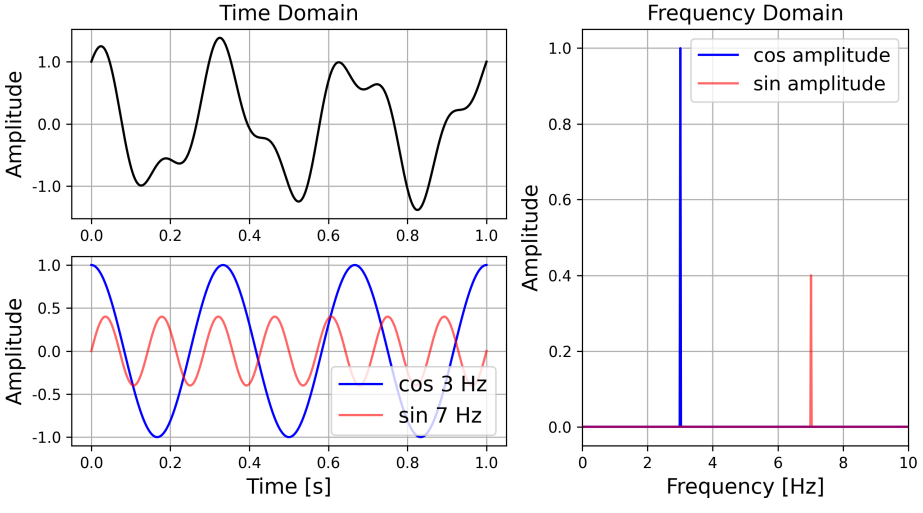

Figure E-9. A complicated-looking function (top left) and the result of its Fourier transform (right) showing that the function is just the sum of a cosine wave and a sine wave of different frequencies and amplitudes (lower left).

There's an interesting parallel with the LMU: just as the LMU SSM projects the input function onto a basis of orthogonal Legendre polynomials, the Fourier transform projects a function $f(t)$ onto a basis of orthogonal functions $\cos(\omega t)$ and $\sin(\omega t)$ of all possible angular frequencies ω, giving a coefficient for each cosine and sine function.

The Fourier transform can also be generalized to complex-valued functions: in this case, the complex-valued function $f(t)$ is projected onto the basis of functions $e^{i\omega t} = \cos(\omega t) + i \cdot \sin(\omega t)$, and the coefficient $F(\omega)$ for each frequency ω is also a complex number. Most implementations of the Fourier transform use this generalized approach, including PyTorch's implementation. In fact, you will get complex coefficients even when the input function $f(t)$ is real-valued. In this case, if you prefer to deal with coefficients of cosines and sines, then it's not too difficult to do the conversion: assuming the complex coefficient for the angular frequency ω is $F(\omega) = a + bi$, the cosine coefficient is $2a$, and the sine coefficient is $-2b$.

It's possible to show that a convolution in the time domain is equivalent to an itemwise multiplication in the frequency domain[12], so we can apply the Fast Fourier

12 See this post by Ulf Bissbort for the proof.

Transform (FFT)[13] to convert both the kernel **k** and the input sequence **u** to the frequency domain, then perform an itemwise multiplication (which is parallelizable) in the frequency domain, and finally convert the result back to the time domain using the Inverse FFT (IFFT).

Let's implement two functions to compute the same convolution: the first using the `F.conv1d()` function in the time domain, and the second using itemwise multiplication in the frequency domain:

```
import torch.nn.functional as F
from torch.fft import rfft, irfft

def convolution_in_time_domain(k, u):
    B, d, L = u.shape  # B = batch size, d = input/output dim, L = seq. length
    u_padded = F.pad(u, (L - 1, 0))  # pad left; shape (B, d, L + L - 1)
    k_flipped = k.flip(dims=[-1]).unsqueeze(1)  # shape (d, 1, L)
    y = F.conv1d(u_padded, k_flipped, groups=d)  # shape (d, 1, L)
    return y

def convolution_in_frequency_domain(k, u):
    B, d, L = u.shape
    n = L + L - 1
    u_f = rfft(u, n=n)  # shape (B, d, n)
    k_f = rfft(k, n=n)  # shape (B, d, n)
    y_f = u_f * k_f.unsqueeze(0)  # shape (B, d, n)
    y = irfft(y_f, n=n)[..., :L]  # shape (B, d, L)
    return y
```

There are a few things to note in this code:

- Both functions accept a kernel of shape (d, L) and an input of shape (B, d, L), where B is the batch size, d is the number of parallel SSMs (one per input channel), and L is the sequence length.
- The first function starts by padding the input sequence with $L - 1$ zeros, which ensures that the model is causal. Next, we flip the kernel: this is because the `F.conv1d()` function actually implements a mathematical operation named cross-correlation, which is just like convolution except the kernel is flipped. In regular CNNs, we can ignore this fact because the kernel is trainable, which means gradient descent can learn the appropriate flipped kernel. But when we want to perform a proper mathematical convolution using a given kernel, we must flip this kernel before calling the `F.conv1d()` function. Lastly, we set `groups=d` when calling `F.conv1d()` to ensure that each of the d sequences are processed independently (one per SSM).

13 The FFT is a fast algorithm for the discrete Fourier transform, which scales as $O(L \cdot \log(L))$ instead of $O(L^2)$ for the naive discrete Fourier Transform.

- In the second function, we start by calling the `torch.fft.rfft()` function on the input and the kernel. This performs real-valued FFT (the `torch.fft.fft()` function performs complex-valued FFT, but our inputs are real numbers). We specify the desired sequence length `n = L + L - 1`, so the `rfft()` function automatically takes care of padding the input with $L - 1$ zeros. Next, we compute the item-wise multiplication of the input and kernel in the frequency domain. Lastly, we call `torch.fft.irfft()` to perform the real-valued inverse FFT, and we crop the result to keep only the valid part of the result (the last $L - 1$ elements are not causal).

You can try generating a batch of random input sequences, and a random kernel, and verify that both functions return the same result (with only tiny differences due to floating point errors).

The most computationally expensive part of the convolutional representation of LTI SSMs is evaluating the kernel $\mathbf{k} = (\overline{\mathbf{C}}\overline{\mathbf{A}}^{n}\overline{\mathbf{B}}, \overline{\mathbf{C}}\overline{\mathbf{A}}^{n-1}\overline{\mathbf{B}}, \ldots, \overline{\mathbf{C}}\overline{\mathbf{A}}\overline{\mathbf{B}}, \overline{\mathbf{C}}\overline{\mathbf{B}})$. It's not a big problem if these matrices are fixed, since we can precompute the kernel up to the maximum input sequence length, then reuse it for any new input sequence, cropping it to the sequence length. In fact, making $\overline{\mathbf{B}}$ and $\overline{\mathbf{C}}$ trainable is doable, we just need $\overline{\mathbf{A}}$ to be fixed since the most expensive step is computing all the powers of $\overline{\mathbf{A}}$ (although storing these powers can take up a lot of RAM).

But what if we want to give the model the ability to optimize matrix $\overline{\mathbf{A}}$ during training? If we naively make $\overline{\mathbf{A}}$ trainable, then it will change during training, so we will need to recompute $\mathbf{k}$ at each training iteration: this would be so slow that it would completely cancel out the speed boost that the convolutional representation offered. Luckily, there is a solution: enter S4.

Efficiently Modeling Long Sequences with Structured SSMs (S4)

An efficient mathematical trick was found by Albert Gu, Karan Goel, and Christopher Ré and published in a 2021 paper titled "Efficiently Modeling Long **S**equences with **S**tructured **S**tate **S**paces" (or S4 for short):[14] it makes it possible to learn the parameters of the continuous-time matrix $\mathbf{A}$ (and derive the discrete-time matrix $\overline{\mathbf{A}}$ from it), while still benefiting from the convolutional speed boost.

The core idea is to express the continuous-time matrix $\mathbf{A}$ as the sum of a diagonal matrix $\mathbf{\Lambda}$ (lambda) and a low-rank matrix $\mathbf{R} = -\mathbf{P}\mathbf{Q}^{*}$. This is called the *diagonal plus*

14 Albert Gu et al., "Efficiently Modeling Long Sequences with Structured State Spaces", arXiv preprint arXiv: 2111.00396 (2021).

low rank (DPLR) technique. All three matrices $\mathbf{\Lambda}$, $\mathbf{P}$, and $\mathbf{Q}$ are complex-valued, and $\mathbf{Q}^*$ is the conjugate transpose[15] of $\mathbf{Q}$. The matrix $\mathbf{\Lambda}$ has a shape of $d_{\text{state}} \times d_{\text{state}}$ (full of zeros except on the main diagonal), while $\mathbf{P}$ and $\mathbf{Q}$ both have a shape of $d_{\text{state}} \times r$, where r is small, typically $r = 1$ or 2.

Instead of training a full $d_{\text{state}} \times d_{\text{state}}$ matrix, we only train the DPLR components $\mathbf{\Lambda}$, $\mathbf{P}$, and $\mathbf{Q}$. This ensures that the matrix $\mathbf{A}$ preserves its desirable structure, close to a diagonal matrix. This particular structure enables some mathematical shortcuts to efficiently compute the kernel $\mathbf{k}$ at each training iteration, by directly computing it in the frequency domain: this removes the need to explicitly compute all of the powers of $\overline{\mathbf{A}}$ (see part 3 of the paper for the mathematical details).

There are two important points to note:

- S4's SSM matrices are initialized using the HiPPO-LegS matrices $\mathbf{A}$ and $\mathbf{B}$: this gives the model an excellent starting point for training. To be more precise, the matrices $\mathbf{\Lambda}$, $\mathbf{P}$, and $\mathbf{Q}$ are derived from the HiPPO-LegS matrix $\mathbf{A}$ such that $\mathbf{A} = \mathbf{\Lambda} - \mathbf{P}\mathbf{Q}^*$. Note that the HiPPO-LegS matrices were designed for an LTV SSM, as we saw earlier, but we're using them in an LTI SSM. That's fine because we're only using these matrices as a good starting point for training.
- S4 converts the continuous-time matrix $\mathbf{A}$ to the discrete-time matrix $\overline{\mathbf{A}}$ using the *bilinear method*, shown in Equation E-13. Like the other discretization methods, the bilinear method involves a small time increment Δ, which the model learns during training; in fact, S4 learns one such parameter for each input channel, allowing it to learn the appropriate time scale for each.

Equation E-13. Bilinear method to discretize an SSM

$$\overline{\mathbf{A}} = \left(\mathbf{I} - \frac{\Delta}{2}\mathbf{A}\right)^{-1}\left(\mathbf{I} + \frac{\Delta}{2}\mathbf{A}\right)$$

$$\overline{\mathbf{B}} = \left(\mathbf{I} - \frac{\Delta}{2}\mathbf{A}\right)^{-1}\Delta\mathbf{B}$$

The authors implemented their ideas in the *S4 model*, which was the first SSM-based model to outperform transformers on several tasks involving long-range dependencies. In particular, it beat the state of the art on the Long Range Arena (LRA) benchmarks. Its architecture resembles a decoder-only transformer (see Figure E-10), but without positional encoding (SSMs don't need them), and replacing the multi-head attention layers with S4 blocks. An S4 block is composed of the structured SSM

15 Transpose the matrix and flip the sign of the imaginary part of each complex number in the matrix; for example $2 + 3i$ becomes $2 - 3i$.

described so far—usually with the addition of a trainable matrix **D** (i.e., a learnable skip connection)—along with a feedforward block (e.g., using a linear layer and an activation function such as GELU), and usually also some dropout, layer normalization, and skip connections (not shown in the figure). This architecture can then be used just like a decoder-only transformer; for example, you can add an embedding layer at the bottom, and a classification head at the top, and you get a model that you can use for text generation.

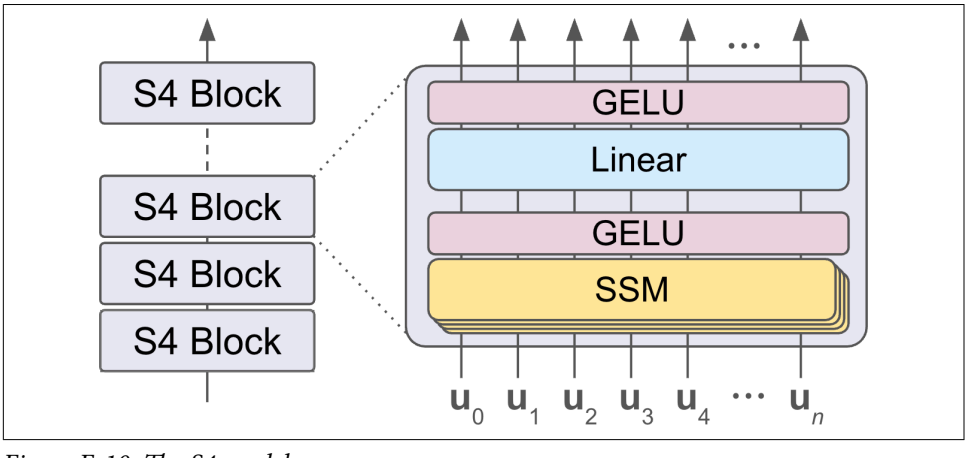

Figure E-10. The S4 model

If you want to build your own custom S4 models, you can use the `S4Block` class available in the official source code of S4.

In a later iteration of S4, the matrix **Q** was replaced with **P**, giving $\mathbf{A} = \mathbf{\Lambda} - \mathbf{P}\mathbf{P}^*$. This tweak stabilized training. Then, in March 2022, Ankit Gupta et al. published the diagonal state spaces (DSS) paper[16] showing that it's actually possible to drop the low-rank part and keep only the diagonal matrix $\mathbf{\Lambda}$. This led to the S4D model[17], which slightly outperformed S4 while being both simpler and faster.

At this point, the ML community was getting really excited about SSMs: they have a strong theoretical foundation that dates back to the 1960s, they have an excellent memory thanks to the HiPPO-LegS initialization, they are fast both during training

16 Ankit Gupta et al., "Diagonal State Spaces are as Effective as Structured State Spaces", arXiv preprint arXiv: 2203.14343 (2022).

17 Albert Gu et al., "On the Parameterization and Initialization of Diagonal State Space Models", arXiv preprint arXiv:2206.11893 (2022).

using the convolutional representation, and during inference using the recurrent representation, and they can flexibly be embedded into simple neural net architectures that resemble existing transformer architectures.

However, the SSMs we discussed so far have one major flaw: the transition matrices do not depend on the inputs. As a result, all inputs are treated equally by the SSM. In contrast, transformers can learn to ignore unimportant inputs; this gives them a significant advantage. This explains why S4 only beats transformers on tasks with long-range dependencies, where transformers struggle. For shorter input sequences, transformers still outperform S4.

This leads us to the last SSM-based neural network architecture in this appendix, which tackled this problem head-on by letting the model select the inputs to memorize.

Mamba: Linear-Time Sequence Modeling with Selective State Spaces

Tri Dao (author of FlashAttention, FlashAttention-2, and HiPPO) and Albert Gu (author of HiPPO, S4, and S4D), struck again in December 2023 with the groundbreaking SSM-based Mamba model,[18] sometimes called S6.[19] This was the first model to match or exceed the performance of state-of-the-art transformers of the same size on language modeling tasks. In some cases, it even matched transformers twice as large. Since Mamba is essentially a sophisticated recurrent neural network, it's quite a comeback for RNNs![20]

Just like S4, Mamba uses single-input single-output SSMs, each independently processing one input channel and producing one output channel. The main innovation in Mamba is the fact that the SSM matrices $\mathbf{B}$ and $\mathbf{C}$ are input-dependent: this allows the model to control which inputs should be stored in the SSM's state at each step (using $\mathbf{B}$), and which parts of the state should be extracted (using $\mathbf{C}$). Moreover, $\mathbf{\Delta}$ is also input-dependent: just like in S4, it contains one time increment Δ per input channel, and it is used to discretize the corresponding SSM; letting the model control $\mathbf{\Delta}$ allows it to choose which input channels to focus on, depending on the inputs.

18 Albert Gu, Tri Dao, "Mamba: Linear-Time Sequence Modeling with Selective State Spaces", arXiv preprint arXiv:2312.00752 (2023).

19 There are 6 S's: Sequences and Structured State Spaces (same as in S4), plus Selective and Scan, as we will see shortly.

20 RNNs were declared resurrected in the 2023 paper by Antonio Orvieto et al., "Resurrecting Recurrent Neural Networks for Long Sequences", arXiv preprint arXiv:2303.06349 (2023).

For discretization, Mamba computes $\overline{\mathbf{A}} = \exp(\Delta\mathbf{A})$ and $\overline{\mathbf{B}} = \Delta\mathbf{B}$. That's ZOH for $\overline{\mathbf{A}}$ and Euler's method for $\overline{\mathbf{B}}$. This hybrid approach is faster to compute and more stable during training than using ZOH for both $\overline{\mathbf{A}}$ and $\overline{\mathbf{B}}$. Moreover, since $\mathbf{A}$ is a diagonal matrix, we can just use the itemwise exponential operation when computing $\overline{\mathbf{A}}$: this gives the same result as with a matrix exponential, only much faster.

Mamba's architecture is composed of a stack of Mamba blocks, plus layer-norm before each block and a skip connection after each block (see the lefthand side of Figure E-11). There are usually extra layers before the first block (such as an embedding layer for language models), and extra layers after the last block (such as a classification head); these are not shown in the figure. Many Mamba variants replace every other Mamba block with a feedforward neural network (FFN), or with a multihead attention (MHA) layer (see the center and righthand side of Figure E-11).

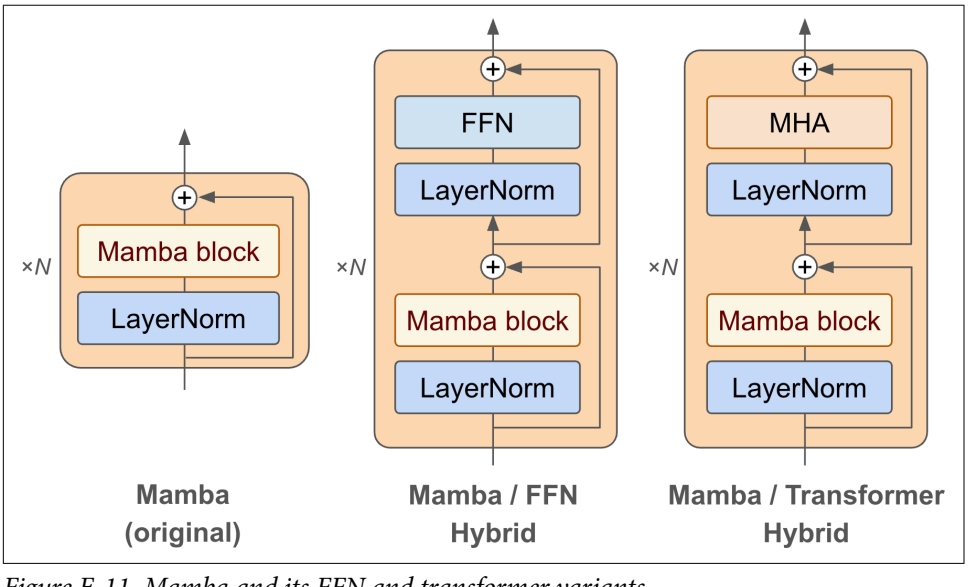

Figure E-11. Mamba and its FFN and transformer variants

Now let's zoom in on the Mamba block (see Figure E-12). In this figure, B is the batch size, L is the sequence length, d_{model} is the number of input channels to the Mamba block, d_{inner} is the number of input channels to the SSMs, and d_{state} is the state size (the last three are shortened to d_{m}, d_{i}, and d_{s} in the figure, respectively). I have shown the actual tensor shapes used in Mamba's implementation, rather than focusing on a single instance and a single time step. This should make it clear when data is shared across dimensions. For example, the tensor $\mathbf{D}$ has a shape of d_{inner} (noted d_{i} in the figure): since there's no B or L dimension, we can tell that this tensor contains one

matrix **D** (actually just a scalar) per SSM (since there's one SSM per input channel), but for each SSM this matrix **D** is shared across all instances and all time steps. Let's start with the right half of the figure:

- The input **X** represents a batch of input sequences; it's a tensor of shape [B, L, d_{model}].
- The input follows two parallel paths: the main path goes through a linear layer, a convolutional layer, the SiLU activation function, and lastly the SSM; the other path goes through a linear layer and a SiLU activation function. Then the outputs of both paths are multiplied itemwise. This is a gating mechanism similar to the one used in the SwiGLU activation function (see Chapter 11): it allows the model to control which parts of the output matter most. The result then goes through a final linear layer to produce the output.
- Notice that the input layers[21] expand the dimensionality by a factor of E (which defaults to 2), and the output layer reduces the dimensionality back to d_{model}. The inner dimensionality d_{inner} is equal to $E \times d_{\text{model}}$. This expansion gives the model more flexibility. Most of the Mamba block's parameters are in these input and output layers.
- It is important to understand that the linear layers focus only on the last dimension, just like in a transformer. In other words, all sequences in the batch and all of their tokens are treated independently and in parallel. Conversely, the SSM and the convolutional layer focus only on the sequence dimension: all instances in the batch and all input channels are treated independently and in parallel. It's as if there were an independent SSM and an independent convolutional layer for each instance and each input channel, all running in parallel, each processing a 1D input sequence.

The Mamba block is causal, since the SSM is naturally causal (it only depends on the current state and inputs), the convolutional layer uses causal padding (i.e., left padding equal to the kernel size minus one), and the other layers ignore the sequence dimension.

21 Mamba implements these two layers using a single linear layer with twice the output size, followed by a split operation: this is more efficient than running two separate linear layers. Similarly, the linear layers that produce **B** and **C**, and the bottom linear layer on the path to **Δ**, are actually all combined into one linear layer plus a split operation.

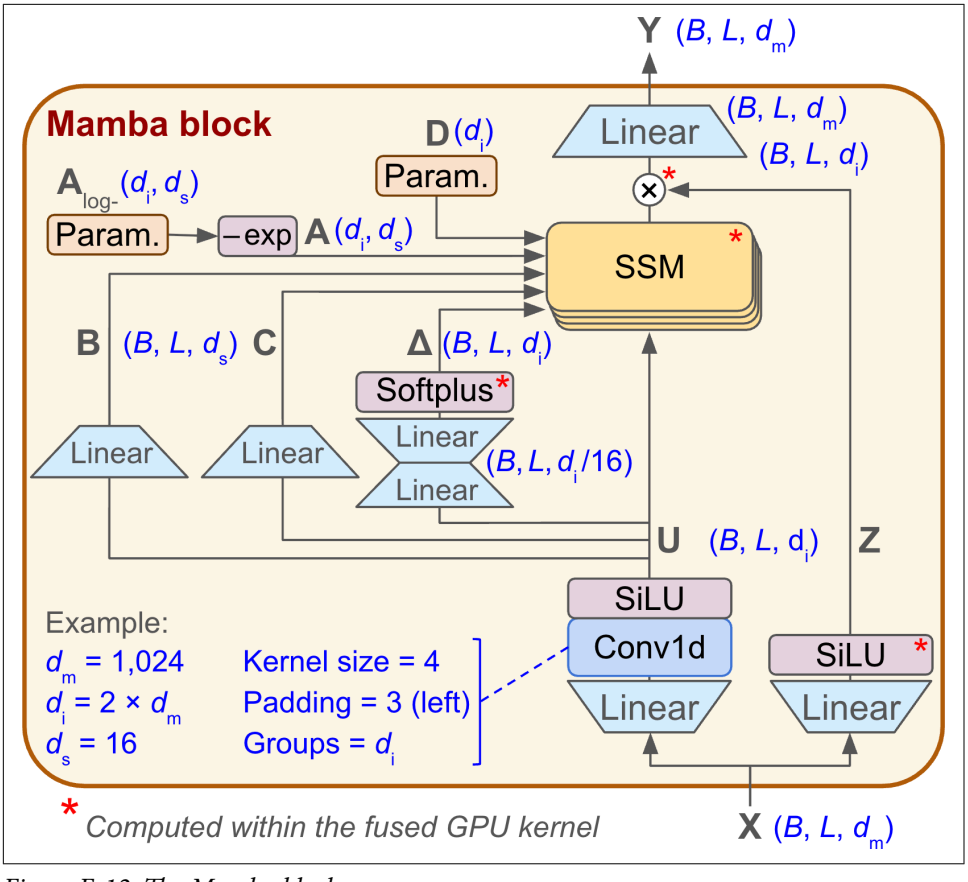

Figure E-12. The Mamba block

Now let's focus on the left part of Figure E-12, starting with **A**:

- Just like in S4D, each SSM's matrix **A** is constrained to be diagonal, so we only need to store d_{state} diagonal terms for each SSM. Since there are d_{inner} independent SSMs, we represent all of their **A** matrices using a tensor of shape $d_{inner} \times d_{state}$ (but keep in mind that in theory each matrix **A** is a diagonal matrix of shape $d_{state} \times d_{state}$).
- To ensure that the SSMs remain stable during training, we need matrix **A** to contain only negative terms on the diagonal, as this makes the state decay over time rather than explode. To guarantee that the terms remain negative during training, we don't train **A** directly: instead, we train a parameter $\mathbf{A}_{log\text{-}}$ and we compute $\mathbf{A} = -\exp(\mathbf{A}_{log\text{-}})$.
- The $\mathbf{A}_{log\text{-}}$ parameter is initialized using $[\log(1), \log(2), \log(3), \ldots, \log(d_{state})]$ for each row (i.e., for each SSM). This ensures that each diagonal matrix **A** uses the

same initialization scheme as the real-valued variant of S4D, with [-1.0, -2.0, -3.0, …, $-d_{\text{state}}$] in its diagonal. This is a simplified variant of HiPPO-LegS diagonal initialization.

Next, let's look at matrices **B**, **C**, and **Δ**, which are all input-dependent:

- The SSM's input tensor **U** is passed to linear layers that produce the SSM tensors **B**, **C**, and **Δ**. For **Δ**, instead of using a linear layer with d_{inner} inputs and d_{inner} outputs, which would be huge, we use two consecutive linear layers where the first reduces the dimensionality by a factor of 16 (it's a bottleneck layer), while the second restores the original dimensionality. Mamba still has a lot of flexibility to model **Δ**, but this technique significantly reduces the number of parameters and computations, while also acting as a regularizer and stabilizing training.
- Notice that **B**, **C**, and **Δ** have a batch dimension (B) and a sequence dimension (L): whereas previous SSMs used the same matrices at each time step, regarless of the inputs (they were linear time-invariant), Mamba is linear time-variant (LTV) and input-dependent, so we have different matrices for each instance and each time step.
- Also note that **B** and **C** are shared across input channels: their shape is [B, L, d_{state}], not [B, L, d_{inner}, d_{state}]. However, **Δ** uses a separate time scale (a scalar) per input channel.
- The time scales must be positive (time can only go forward), which is why we use the softplus activation function when outputting **Δ**.

Lastly, the parameter **D** is fully trainable, with one scalar per SSM. This matrix provides a skip connection around the SSM, with a trainable scale for each input channel.

And there you have it! You now know how to build Mamba from scratch. However, a naive implementation would be horrendously slow because we cannot use S4's convolutional representation trick. Indeed, this trick only works with linear time-invariant SSMs, but Mamba uses linear time-variant SSMs. Luckily, the Mamba authors provided an alternative solution to speed up Mamba's SSMs: they implemented an optimized GPU kernel based on the *scan* operation.

To understand the scan operation (also called the parallel prefix or prefix sum), let's first focus on a simple use case: given a list of n numbers [z_0, z_1, z_2, …, z_{n-1}], we want to compute the cumulative sum of its elements, [z_0, $z_0 + z_1$, $z_0 + z_1 + z_2$, …, $z_0 + z_1 + z_2 + \ldots + z_{n-1}$]. For example, given the list [3, 1, 5, 9, 7, 8, 2, 4], we want to return [3, 4, 9, 18, 25, 33, 35, 39]. The scan operation can do this in just three steps (see Figure E-13):

- In the first step, it computes the sum of all consecutive pairs. Crucially, all sums can be performed in parallel.

- Once this first step is finished, the algorithm can take the resulting list and compute the sum of every pair of numbers located two indices apart.
- Again, all sums are performed in parallel. In the next step, the algorithm sums elements located four indices apart (if there were more element, we would keep doubling the distance at each step).

And we're done! This algorithm performs $2n + 1$ sums (instead of $n - 1$ in a naive sequential implementation), but assuming there are more processors than elements in the list, it only takes $O(\log(n))$ time to run, instead of $O(n)$ in the naive implementation.

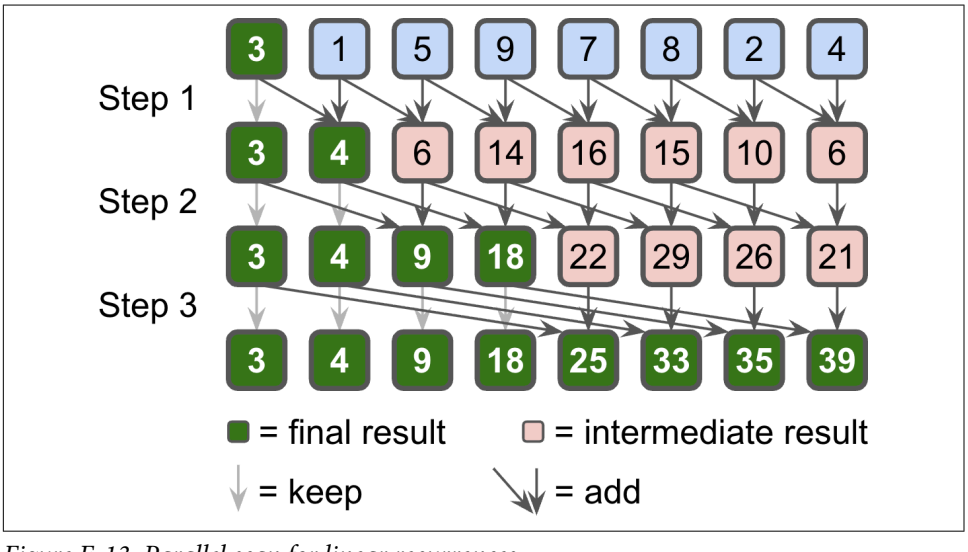

Figure E-13. Parallel scan for linear recurrences

Why does this help with SSMs? Well, the scan operation doesn't just work with addition: it also works with any binary operation $\circledcirc$, as long as it is associative, meaning that $(a \circledcirc b) \circledcirc c = a \circledcirc (b \circledcirc c)$. Now consider the following binary operation $\circledcirc$ which takes two pairs as input, $(\overline{\mathbf{A}}_0, \mathbf{v}_0)$ and $(\overline{\mathbf{A}}_1, \mathbf{v}_1)$, and outputs the pair $(\overline{\mathbf{A}}_1\overline{\mathbf{A}}_0, \overline{\mathbf{A}}_1\mathbf{v}_0 + \mathbf{v}_1)$. It's not too hard to verify that this operation is associative:

$$
\begin{aligned}
&((\overline{\mathbf{A}}_0, \mathbf{v}_0) \circledcirc (\overline{\mathbf{A}}_1, \mathbf{v}_1)) \circledcirc (\overline{\mathbf{A}}_2, \mathbf{v}_2) \\
&= (\overline{\mathbf{A}}_1\overline{\mathbf{A}}_0, \overline{\mathbf{A}}_1\mathbf{v}_0 + \mathbf{v}_1) \circledcirc (\overline{\mathbf{A}}_2, \mathbf{v}_2) \\
&= (\overline{\mathbf{A}}_2\overline{\mathbf{A}}_1\overline{\mathbf{A}}_0, \overline{\mathbf{A}}_2(\overline{\mathbf{A}}_1\mathbf{v}_0 + \mathbf{v}_1) + \mathbf{v}_2) \\
&= (\overline{\mathbf{A}}_2\overline{\mathbf{A}}_1\overline{\mathbf{A}}_0, \overline{\mathbf{A}}_2\overline{\mathbf{A}}_1\mathbf{v}_0 + \overline{\mathbf{A}}_2\mathbf{v}_1 + \mathbf{v}_2) \\
&= (\overline{\mathbf{A}}_2\overline{\mathbf{A}}_1\overline{\mathbf{A}}_0, (\overline{\mathbf{A}}_2\overline{\mathbf{A}}_1)\mathbf{v}_0 + (\overline{\mathbf{A}}_2\mathbf{v}_1 + \mathbf{v}_2)) \\
&= (\overline{\mathbf{A}}_0, \mathbf{v}_0) \circledcirc (\overline{\mathbf{A}}_2\overline{\mathbf{A}}_1, \overline{\mathbf{A}}_2\mathbf{v}_1 + \mathbf{v}_2) \\
&= (\overline{\mathbf{A}}_0, \mathbf{v}_0) \circledcirc ((\overline{\mathbf{A}}_1, \mathbf{v}_1) \circledcirc (\overline{\mathbf{A}}_2, \mathbf{v}_2))
\end{aligned}
$$

Now let's see what happens if we apply a scan operation on a sequence of pairs $[(\overline{\mathbf{A}}_0, \mathbf{v}_0), (\overline{\mathbf{A}}_1, \mathbf{v}_1), \ldots, (\overline{\mathbf{A}}_n, \mathbf{v}_n)]$, using the binary operator $\circledcirc$ we just defined. We get the following output pairs:

$$\mathbf{o}_0 = (\overline{\mathbf{A}}_0, \mathbf{v}_0)$$
$$\mathbf{o}_1 = \mathbf{o}_0 \circledcirc (\overline{\mathbf{A}}_1, \mathbf{v}_1) = (\overline{\mathbf{A}}_1\overline{\mathbf{A}}_0, \overline{\mathbf{A}}_1\mathbf{v}_0 + \mathbf{v}_1)$$
$$\mathbf{o}_2 = \mathbf{o}_1 \circledcirc (\overline{\mathbf{A}}_2, \mathbf{v}_2) = (\overline{\mathbf{A}}_2\overline{\mathbf{A}}_1\overline{\mathbf{A}}_0, \overline{\mathbf{A}}_2\overline{\mathbf{A}}_1\mathbf{v}_0 + \overline{\mathbf{A}}_2\mathbf{v}_1 + \mathbf{v}_2)$$
$$\ldots$$
$$\mathbf{o}_n = \mathbf{o}_{n-1} \circledcirc (\overline{\mathbf{A}}_n, \mathbf{v}_n) = (\overline{\mathbf{A}}_n\overline{\mathbf{A}}_{n-1}\ldots\overline{\mathbf{A}}_1\overline{\mathbf{A}}_0, \overline{\mathbf{A}}_n\overline{\mathbf{A}}_{n-1}\ldots\overline{\mathbf{A}}_2\overline{\mathbf{A}}_1\mathbf{v}_0 + \overline{\mathbf{A}}_n\overline{\mathbf{A}}_{n-1}\ldots\overline{\mathbf{A}}_3\overline{\mathbf{A}}_2\mathbf{v}_1 + \ldots + \overline{\mathbf{A}}_n\mathbf{v}_{n-1} + \mathbf{v}_n$$

The second item of each $\mathbf{o}_k$ looks very much like the output of an SSM; in fact, we only need to multiply the inputs by $\overline{\mathbf{B}}$ and the outputs by $\mathbf{C}$, then add $\mathbf{D}$, and we're done. So here's the full process to efficiently compute the SSMs' outputs:

1. Generate $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, $\mathbf{D}$, $\boldsymbol{\Delta}$ as explained earlier (see Figure E-12). The tensors $\mathbf{B}$, $\mathbf{C}$, $\boldsymbol{\Delta}$ contain $\mathbf{B}_k$, $\mathbf{C}_k$, $\boldsymbol{\Delta}_k$ for each time step k, for each SSM, and each instance in the batch. The following steps are run in parallel for all SSMs and all instances.
2. Discretize $\mathbf{A}$ and $\mathbf{B}$ using $\boldsymbol{\Delta}$. As explained earlier, we can use ZOH to get $\overline{\mathbf{A}}$, and Euler's method to get $\overline{\mathbf{B}}$. The tensors $\overline{\mathbf{A}}$ and $\overline{\mathbf{B}}$ contain $\overline{\mathbf{A}}_k$ and $\overline{\mathbf{B}}_k$ for each time step k.
3. Compute $\mathbf{v}_k = \overline{\mathbf{B}}_k u_k$ for each time step k, all in parallel.
4. Run the scan operation using the sequence of pairs $[(\overline{\mathbf{A}}_0, \mathbf{v}_0), (\overline{\mathbf{A}}_1, \mathbf{v}_1), \ldots, (\overline{\mathbf{A}}_n, \mathbf{v}_n)]$ to obtain the cumulative result $\mathbf{o}_k$ for each time step k.
5. Drop the first element of each $\mathbf{o}_k$ and compute the SSM's outputs: $\mathbf{y}_k = \mathbf{C}_k\mathbf{o}_k + \mathbf{D}$ for each time step k (in parallel).

The Mamba authors developed an optimized GPU kernel that fuses steps 2 to 5 (and it also takes care of adding the bias term of the second linear layer that produces $\boldsymbol{\Delta}$): this avoids materializing large intermediate tensors and drastically reduces the amount of memory transfers, much like FastAttention (see Chapter 17). If the input sequence doesn't fit in the GPU's high-performance memory, then it is split into chunks. This optimized GPU kernel runs 20 to 40 times faster than a pure PyTorch implementation.

Mamba-2[22] was released in May 2024, improving Mamba's speed by a factor of 2 to 8.

Using Mamba

If you want to try out SSMs, many pretrained models are available via the Hugging Face Transformers library and can be used just like regular transformers (see Chapter 15). For example, the following code downloads a small version of Mamba and uses it to generate some text:

```
from transformers import AutoTokenizer, AutoModelForCausalLM

model_id = "state-spaces/mamba-790m-hf"  # or mamba-2.8b if it fits on your GPU
model = AutoModelForCausalLM.from_pretrained(model_id)
tokenizer = AutoTokenizer.from_pretrained(model_id)
input_text = "State space models are"
inputs = tokenizer(input_text, return_tensors="pt")
outputs = model.generate(**inputs, max_length=50)  # the usual generation API
print(tokenizer.decode(outputs[0], skip_special_tokens=True))
```

This code will default to a slow sequential implementation, unless you install the causal-conv1d and mamba-ssm libraries, which provide GPU kernels for the causal 1D convolution and for Mamba's selective SSM (discussed earlier), respectively. At the time of writing, this only works on Linux using an Nvidia GPU with CUDA 11.6+.

If you want to create your own custom Mamba model and train it from scratch, you can create a custom `MambaConfig` object and create a `MambaForCausalLM` model based on this config, for example:

```
from transformers import MambaConfig, MambaForCausalLM

custom_config = MambaConfig(  # a tiny toy Mamba model
    vocab_size=1000,
    hidden_size=64,  # d_model
    num_hidden_layers=4,  # number of Mamba blocks
    state_size=8,  # d_state
    conv_kernel_size=4,  # the Conv1d layer's kernel size
    expansion_factor=2,  # the expansion factor E
)
tiny_mamba = MambaForCausalLM(config=custom_config)
```

22 Tri Dao, Albert Gu, "Transformers are SSMs: Generalized Models and Efficient Algorithms Through Structured State Space Duality", arXiv preprint arXiv:2405.21060 (2024).

If you have installed the mamba-ssm library, then you can even build your own custom models based on the Mamba or Mamba2 block:

```
from mamba_ssm import Mamba  # or Mamba2

mamba_block = Mamba(d_model=32, d_state=16, d_conv=4, expand=2)
```

Congratulations, you have made it through this difficult appendix! Don't worry if you didn't grasp everything on the first read. It took me quite a while to really understand SSMs: just persevere and things will gradually fall into place.