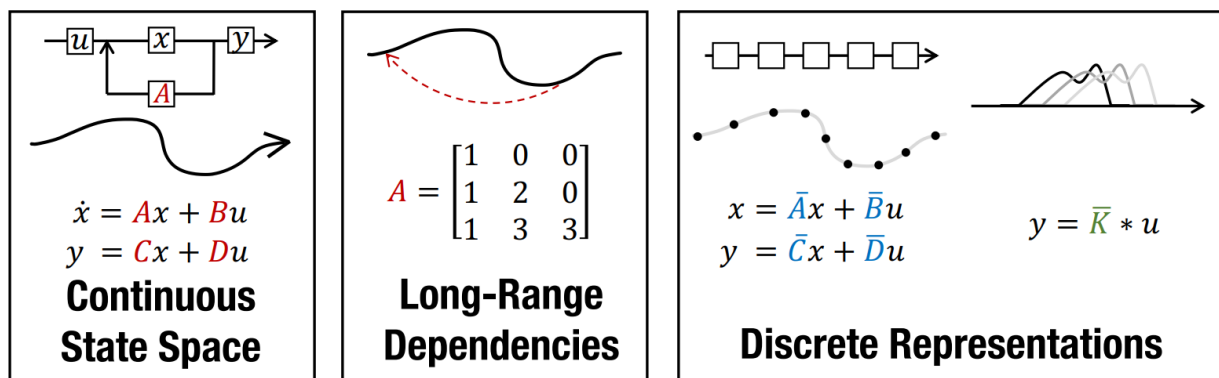


The Annotated S4

Efficiently Modeling Long Sequences with Structured State Spaces

Albert Gu, Karan Goel, and Christopher Ré.



Blog Post and Library by Sasha Rush and Sidd Karamcheti, v3

The Structured State Space for Sequence Modeling (S4) architecture is a new approach to very long-range sequence modeling tasks for vision, language, and audio, showing a capacity to capture dependencies over tens of thousands of steps. Especially impressive are the model's results on the challenging Long Range Arena benchmark, showing an ability to reason over sequences of up to 16,000+ elements with high accuracy.

MODEL	LISTOPS	TEXT	RETRIEVAL	IMAGE	PATHFINDER	PATH-X	AVG
Transformer	36.37	64.27	57.46	42.44	71.40	✗	53.66
Reformer	<u>37.27</u>	56.10	53.40	38.07	68.50	✗	50.56
BigBird	36.05	64.02	59.29	40.83	74.87	✗	54.17
Linear Trans.	16.13	<u>65.90</u>	53.09	42.34	75.30	✗	50.46
Performer	18.01	65.40	53.82	42.77	77.05	✗	51.18
FNet	35.33	65.11	59.61	38.67	<u>77.80</u>	✗	54.42
Nyströmformer	37.15	65.52	<u>79.56</u>	41.58	70.94	✗	57.46
Luna-256	37.25	64.57	79.29	<u>47.38</u>	77.72	✗	<u>59.37</u>
S4	58.35	76.02	87.09	87.26	86.05	88.10	80.48

The paper is also a refreshing departure from Transformers, taking a very different approach to an important problem-space. However, several of our colleagues have also noted privately the difficulty of gaining intuition for the model. This blog post is a first step towards this goal of gaining intuition, linking concrete code implementations with explanations from the S4 paper – very much in the style of the annotated Transformer. Hopefully this combination of code and literate explanations helps you follow the details of the model. By the end of the blog you will have an efficient working version of S4 that can operate as a CNN for training, but then convert to an efficient RNN at test time. To preview the results, you will be able to generate images from pixels and sounds directly from audio waves on a standard GPU.

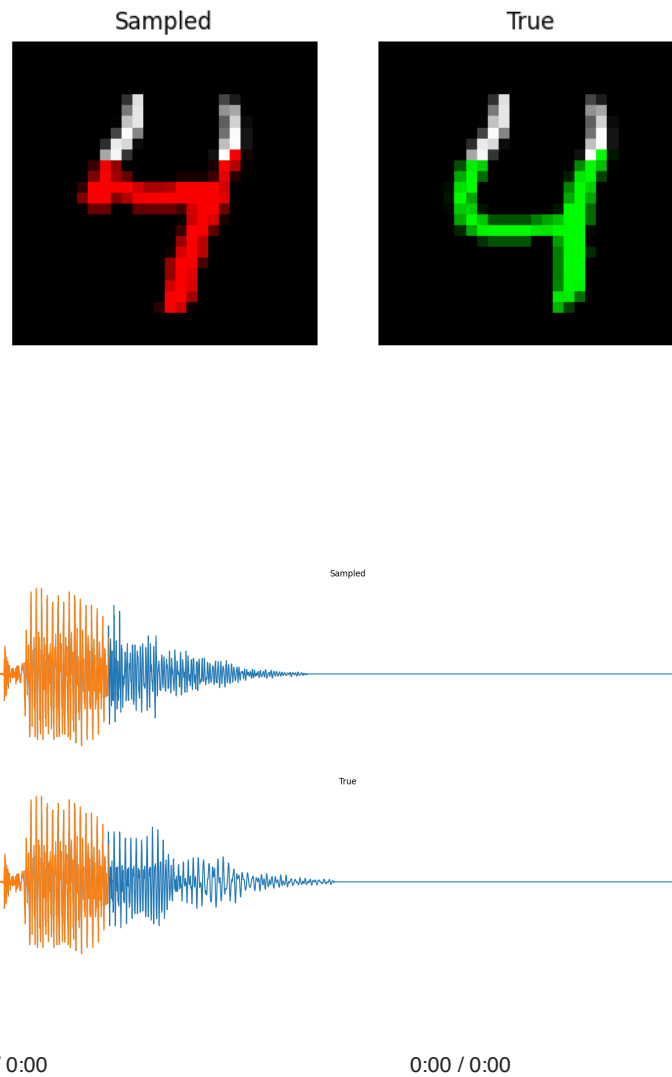


Table of Contents

- Part 1: State Space Models (Modeling)
 - Discrete-time SSM: The Recurrent Representation
 - Tangent: A Mechanics Example
 - Training SSMs: The Convolutional Representation
 - An SSM Neural Network.
- Part 1b: Addressing Long-Range Dependencies with HiPPO
- Part 2: Implementing S4 (Advanced)
 - Step 1. SSM Generating Functions
 - Step 2: Diagonal Case
 - Step 3: Diagonal Plus Low-Rank
 - Diagonal Plus Low-Rank RNN.
 - Turning HiPPO to DPLR

- Final Check
- Part 3: S4 in Practice (NN Implementation)
 - S4 CNN / RNN Layer
 - Sampling and Caching
 - Experiments: MNIST
 - Experiments: QuickDraw
 - Experiments: Spoken Digits
- Conclusion

Note that this project uses JAX with the Flax NN library. While we personally mainly use Torch, the functional nature of JAX is a good fit for some of the complexities of S4. We make heavy use of `vmap`, `scan`, their NN cousins, and most importantly `jax.jit` to compile fast and efficient S4 layers.

```
from functools import partial
import jax
import jax.numpy as np
from flax import linen as nn
from jax.nn.initializers import lecun_normal, normal
from jax.numpy.linalg import eigh, inv, matrix_power
from jax.scipy.signal import convolve

if __name__ == "__main__":
    # For this tutorial, construct a global JAX rng key
    # But we don't want it when importing as a library
    rng = jax.random.PRNGKey(1)
```

Part 1: State Space Models

Let's get started! Our goal is the efficient modeling of long sequences. To do this, we are going to build a new neural network layer based on State Space Models. By the end of this section we will be able to build and run a model with this layer. However, we are going to need some technical background. Let's work our way through the background of the paper.

The state space model is defined by this simple equation. It maps a 1-D input signal $u(t)$ to an N -D latent state $x(t)$ before projecting to a 1-D output signal $y(t)$.

$$\begin{aligned}x'(t) &= \mathbf{A}x(t) + \mathbf{B}u(t) \\ y(t) &= \mathbf{C}x(t) + \mathbf{D}u(t)\end{aligned}$$

Our goal is to simply use the SSM as a black-box representation in a deep sequence model, where \mathbf{A} , \mathbf{B} , \mathbf{C} , \mathbf{D} are parameters learned by gradient descent. For the

remainder, we will omit the parameter \mathbf{D} for exposition (or equivalently, assume $\mathbf{D} = \mathbf{0}$ because the term $\mathbf{D}u$ can be viewed as a skip connection and is easy to compute).

An SSM maps a input $u(t)$ to a state representation vector $x(t)$ and an output $y(t)$. For simplicity, we assume the input and output are one-dimensional, and the state representation is N -dimensional. The first equation defines the change in $x(t)$ over time.

Our SSMs will be defined by three matrices – \mathbf{A} , \mathbf{B} , \mathbf{C} – which we will learn. For now we begin with a random SSM, to define sizes,

```
def random_SSM(rng, N):
    a_r, b_r, c_r = jax.random.split(rng, 3)
    A = jax.random.uniform(a_r, (N, N))
    B = jax.random.uniform(b_r, (N, 1))
    C = jax.random.uniform(c_r, (1, N))
    return A, B, C
```

Discrete-time SSM: The Recurrent Representation

To be applied on a discrete input sequence (u_0, u_1, \dots) instead of continuous function $u(t)$, the SSM must be discretized by a **step size** Δ that represents the resolution of the input. Conceptually, the inputs u_k can be viewed as sampling an implicit underlying continuous signal $u(t)$, where $u_k = u(k\Delta)$.

To discretize the continuous-time SSM, we use the bilinear method, which converts the state matrix \mathbf{A} into an approximation $\overline{\mathbf{A}}$. The discrete SSM is:

$$\begin{aligned}\overline{\mathbf{A}} &= (\mathbf{I} - \Delta/2 \cdot \mathbf{A})^{-1}(\mathbf{I} + \Delta/2 \cdot \mathbf{A}) \\ \overline{\mathbf{B}} &= (\mathbf{I} - \Delta/2 \cdot \mathbf{A})^{-1} \Delta \mathbf{B} \\ \overline{\mathbf{C}} &= \mathbf{C}\end{aligned}$$

```
def discretize(A, B, C, step):
    I = np.eye(A.shape[0])
    BL = inv(I - (step / 2.0) * A)
    Ab = BL @ (I + (step / 2.0) * A)
    Bb = (BL * step) @ B
    return Ab, Bb, C
```

This equation is now a *sequence-to-sequence* map $u_k \mapsto y_k$ instead of function-to-function. Moreover the state equation is now a recurrence in x_k , allowing the discrete SSM to be computed like an RNN. Concretely, $x_k \in \mathbb{R}^N$ can be viewed as a *hidden*

state with transition matrix $\overline{\mathbf{A}}$.

$$\begin{aligned}x_k &= \overline{\mathbf{A}}x_{k-1} + \overline{\mathbf{B}}u_k \\y_k &= \overline{\mathbf{C}}x_k\end{aligned}$$

As the paper says, this “step” function does look superficially like that of an RNN. We can implement this with a scan in JAX,

```
def scan_SSM(Ab, Bb, Cb, u, x0):
    def step(x_k_1, u_k):
        x_k = Ab @ x_k_1 + Bb @ u_k
        y_k = Cb @ x_k
        return x_k, y_k

    return jax.lax.scan(step, x0, u)
```

Putting everything together, we can run the SSM by first discretizing, then iterating step by step,

```
def run_SSM(A, B, C, u):
    L = u.shape[0]
    N = A.shape[0]
    Ab, Bb, Cb = discretize(A, B, C, step=1.0 / L)

    # Run recurrence
    return scan_SSM(Ab, Bb, Cb, u[:, np.newaxis], np.zeros((N,))) [1]
```

Tangent: A Mechanics Example

To gain some more intuition and test our SSM implementation, we pause from machine learning to implement a classic example from mechanics.

In this example, we consider the forward position $y(t)$ of a mass attached to a wall with a spring. Over time, varying force $u(t)$ is applied to this mass. The system is parameterized by mass (m), spring constant (k), friction constant (b). We can relate these with the following differential equation:

$$my''(t) = u(t) - by'(t) - ky(t)$$

Rewriting this in matrix form yields an SSM in the following form:

$$\begin{aligned}\mathbf{A} &= \begin{bmatrix} 0 & 1 \\ -k/m & -b/m \end{bmatrix} \\ \mathbf{B} &= \begin{bmatrix} 0 \\ 1/m \end{bmatrix} & \mathbf{C} &= \begin{bmatrix} 1 & 0 \end{bmatrix}\end{aligned}$$

```
def example_mass(k, b, m):
    A = np.array([[0, 1], [-k / m, -b / m]])
    B = np.array([[0], [1.0 / m]])
    C = np.array([[1.0, 0]])
    return A, B, C
```

Looking at the \mathbf{C} , we should be able to convince ourselves that the first dimension of the hidden state is the position (since that becomes $y(t)$). The second dimension is the velocity, as it is impacted by $u(t)$ through \mathbf{B} . The transition \mathbf{A} relates these terms.

We'll set u to be a continuous function of t ,

```
@partial(np.vectorize, signature="()->()")
def example_force(t):
    x = np.sin(10 * t)
    return x * (x > 0.5)
```

Let's run this SSM through our code.

```
def example_ssm():
    # SSM
    ssm = example_mass(k=40, b=5, m=1)

    # L samples of u(t).
    L = 100
    step = 1.0 / L
    ks = np.arange(L)
    u = example_force(ks * step)

    # Approximation of y(t).
    y = run_SSM(*ssm, u)

    # Plotting ---
    import matplotlib.pyplot as plt
    import seaborn
    from celluloid import Camera

    seaborn.set_context("paper")
    fig, (ax1, ax2, ax3) = plt.subplots(3)
    camera = Camera(fig)
    ax1.set_title("Force $u_k$")
    ax2.set_title("Position $y_k$")
    ax3.set_title("Object")
    ax1.set_xticks([], [])
    ax2.set_xticks([], [])

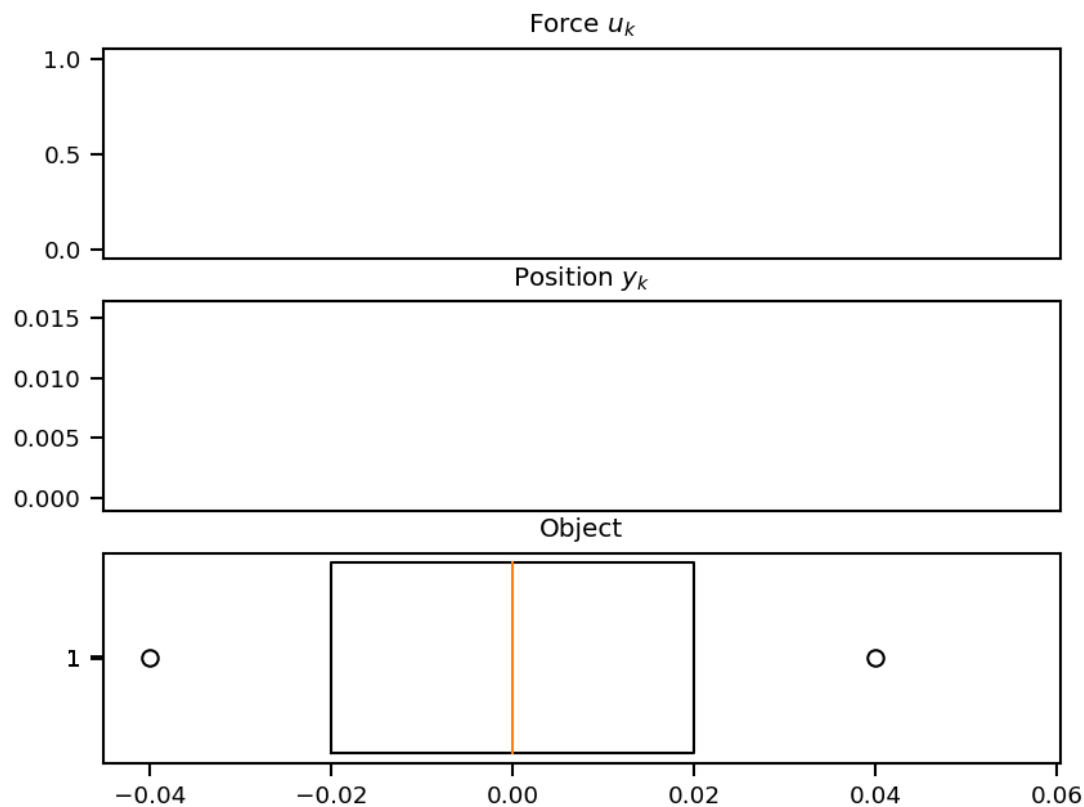
    # Animate plot over time
    for k in range(0, L, 2):
```

```

ax1.plot(ks[:k], u[:k], color="red")
ax2.plot(ks[:k], y[:k], color="blue")
ax3.boxplot(
    [[y[k, 0] - 0.04, y[k, 0], y[k, 0] + 0.04]],
    showcaps=False,
    whis=False,
    vert=False,
    widths=10,
)
camera.snap()
anim = camera.animate()
anim.save("images/line.gif", dpi=150, writer="imagemagick")

if False:
    example_ssm()

```



Neat! And that it was just 1 SSM, with 2 hidden states over 100 steps. The final model will have had 100s of stacked SSMs over thousands of steps. But first – we need to make these models practical to train.

Training SSMs: The Convolutional Representation

The punchline of this section is that we can turn the “RNN” above into a “CNN” by unrolling. Let’s go through the derivation.

The recurrent SSM is not practical for training on modern hardware due to its sequential nature. Instead, there is a well-known connection between linear time-invariant (LTI) SSMs and continuous convolutions. Correspondingly, the recurrent SSM can actually be written as a discrete convolution.

For simplicity let the initial state be $x_{-1} = 0$. Then unrolling explicitly yields:

$$\begin{aligned} x_0 &= \overline{B}u_0 & x_1 &= \overline{A}\overline{B}u_0 + \overline{B}u_1 & x_2 &= \overline{A}^2\overline{B}u_0 + \overline{A}\overline{B}u_1 + \overline{B}u_2 \\ y_0 &= \overline{C}\overline{B}u_0 & y_1 &= \overline{C}\overline{A}\overline{B}u_0 + \overline{C}\overline{B}u_1 & y_2 &= \overline{C}\overline{A}^2\overline{B}u_0 + \overline{C}\overline{A}\overline{B}u_1 + \overline{C}\overline{B}u_2 \end{aligned}$$

This can be vectorized into a convolution with an explicit formula for the convolution kernel.

$$\begin{aligned} y_k &= \overline{C}\overline{A}^k\overline{B}u_0 + \overline{C}\overline{A}^{k-1}\overline{B}u_1 + \dots + \overline{C}\overline{A}\overline{B}u_{k-1} + \overline{C}\overline{B}u_k \\ y &= \overline{K} * u \end{aligned}$$

$$\overline{K} \in \mathbb{R}^L = (\overline{C}\overline{B}, \overline{C}\overline{A}\overline{B}, \dots, \overline{C}\overline{A}^{L-1}\overline{B})$$

We call \overline{K} the SSM convolution kernel or filter.

Note that this is a *giant* filter. It is the size of the entire sequence!

```
def K_conv(Ab, Bb, Cb, L):
    return np.array(
        [(Cb @ matrix_power(Ab, l) @ Bb).reshape() for l in range(L)]
    )
```

Warning: this implementation is naive and unstable. In practice it will fail to work for more than very small lengths. However, we are going to replace it with S4 in Part 2, so for now we just keep it around as a placeholder.

We can compute the result of applying this filter either with a standard direct convolution or by using convolution theorem with Fast Fourier Transform (FFT). The discrete convolution theorem – for circular convolution of two sequences – allows us to efficiently calculate the output of convolution by first multiplying FFTs of the input sequences and then applying an inverse FFT. To utilize this theorem for non-circular convolutions as in our case, we need to pad the input sequences with zeros, and then unpad the output sequence. As the length gets longer this FFT method will be more efficient than the direct convolution,

```
def causal_convolution(u, K, nofft=False):
    if nofft:
        return convolve(u, K, mode="full")[: u.shape[0]]
```



```

else:
    assert K.shape[0] == u.shape[0]
    ud = np.fft.rfft(np.pad(u, (0, K.shape[0])))
    Kd = np.fft.rfft(np.pad(K, (0, u.shape[0])))
    out = ud * Kd
    return np.fft.irfft(out)[: u.shape[0]]

```

The CNN method and the RNN method yield (roughly) the same result,

```

def test_cnn_is_rnn(N=4, L=16, step=1.0 / 16):
    ssm = random_SSM(rng, N)
    u = jax.random.uniform(rng, (L,))
    jax.random.split(rng, 3)
    # RNN
    rec = run_SSM(*ssm, u)

    # CNN
    ssmb = discretize(*ssm, step=step)
    conv = causal_convolution(u, K_conv(*ssmb, L))

    # Check
    assert np.allclose(rec.ravel(), conv.ravel())

```

An SSM Neural Network.

We now have all of the machinery needed to build a basic SSM neural network layer. As defined above, the discrete SSM defines a map from $\mathbb{R}^L \rightarrow \mathbb{R}^L$, i.e. a 1-D sequence map. We assume that we are going to be learning the parameters B and C , as well as a step size Δ and a scalar D parameter. The HiPPO matrix is used for the transition A . We learn the step size in log space.

```

def log_step_initializer(dt_min=0.001, dt_max=0.1):
    def init(key, shape):
        return jax.random.uniform(key, shape) * (
            np.log(dt_max) - np.log(dt_min)
        ) + np.log(dt_min)

    return init

```

For the SSM layer most of the work is to build the filter. The actual call to the network is just the (huge) convolution we specified above.

Note for Torch users: setup in Flax is called each time the parameters are updated. This is similar to the Torch parameterizations.

As noted above this same layer can be used either as an RNN or a CNN. The argument `decode` determines which path is used. In the case of RNN we cache the previous state at each call in a Flax variable collection called `cache`.

```

class SSMLayer(nn.Module):
    N: int
    l_max: int
    decode: bool = False

    def setup(self):
        # SSM parameters
        self.A = self.param("A", lecun_normal(), (self.N, self.N))
        self.B = self.param("B", lecun_normal(), (self.N, 1))
        self.C = self.param("C", lecun_normal(), (1, self.N))
        self.D = self.param("D", nn.initializers.ones, (1,))

        # Step parameter
        self.log_step = self.param("log_step", log_step_initializer(), (1,))

        step = np.exp(self.log_step)
        self.ssm = discretize(self.A, self.B, self.C, step=step)
        self.K = K_conv(*self.ssm, self.l_max)

        # RNN cache for long sequences
        self.x_k_1 = self.variable("cache", "cache_x_k", np.zeros,
                                   (self.N,))

    def __call__(self, u):
        if not self.decode:
            # CNN Mode
            return causal_convolution(u, self.K) + self.D * u
        else:
            # RNN Mode
            x_k, y_s = scan_SSM(*self.ssm, u[:, np.newaxis],
                                self.x_k_1.value)
            if self.is_mutable_collection("cache"):
                self.x_k_1.value = x_k
            return y_s.reshape(-1).real + self.D * u

```

Since our SSMs operate on scalars, we make H different, stacked copies (H different SSMs!) with different parameters. Here we use the Flax vmap method to easily define these copies,

```

def cloneLayer(layer):
    return nn.vmap(
        layer,
        in_axes=1,
        out_axes=1,
        variable_axes={"params": 1, "cache": 1, "prime": 1},
        split_rngs={"params": True},
    )

```

```
SSMLayer = cloneLayer(SSMLayer)
```

This SSM Layer can then be put into a standard NN. Here we add a block that pairs a call to an SSM with dropout and a linear projection.

```
class SequenceBlock(nn.Module):
    layer_cls: nn.Module
    layer: dict # Hyperparameters of inner layer
    dropout: float
    d_model: int
    prenorm: bool = True
    glu: bool = True
    training: bool = True
    decode: bool = False

    def setup(self):
        self.seq = self.layer_cls(**self.layer, decode=self.decode)
        self.norm = nn.LayerNorm()
        self.out = nn.Dense(self.d_model)
        if self.glu:
            self.out2 = nn.Dense(self.d_model)
        self.drop = nn.Dropout(
            self.dropout,
            broadcast_dims=[0],
            deterministic=not self.training,
        )

    def __call__(self, x):
        skip = x
        if self.prenorm:
            x = self.norm(x)
        x = self.seq(x)
        x = self.drop(nn.gelu(x))
        if self.glu:
            x = self.out(x) * jax.nn.sigmoid(self.out2(x))
        else:
            x = self.out(x)
        x = skip + self.drop(x)
        if not self.prenorm:
            x = self.norm(x)
        return x
```

We can then stack a bunch of these blocks on top of each other to produce a stack of SSM layers. This can be used for classification or generation in the standard way as a Transformer.

```
class Embedding(nn.Embed):
    num_embeddings: int
    features: int
```

```

@nn.compact
def __call__(self, x):
    y = nn.Embed(self.num_embeddings, self.features)(x[...], 0)
    return np.where(x > 0, y, 0.0)

class StackedModel(nn.Module):
    layer_cls: nn.Module
    layer: dict # Extra arguments to pass into layer constructor
    d_output: int
    d_model: int
    n_layers: int
    prenorm: bool = True
    dropout: float = 0.0
    embedding: bool = False # Use nn.Embed instead of nn.Dense encoder
    classification: bool = False
    training: bool = True
    decode: bool = False # Probably should be moved into layer_args

    def setup(self):
        if self.embedding:
            self.encoder = Embedding(self.d_output, self.d_model)
        else:
            self.encoder = nn.Dense(self.d_model)
        self.decoder = nn.Dense(self.d_output)
        self.layers = [
            SequenceBlock(
                layer_cls=self.layer_cls,
                layer=self.layer,
                prenorm=self.prenorm,
                d_model=self.d_model,
                dropout=self.dropout,
                training=self.training,
                decode=self.decode,
            )
            for _ in range(self.n_layers)
        ]

    def __call__(self, x):
        if not self.classification:
            if not self.embedding:
                x = x / 255.0 # Normalize
            if not self.decode:
                x = np.pad(x[:-1], [(1, 0), (0, 0)])
        x = self.encoder(x)
        for layer in self.layers:
            x = layer(x)
        if self.classification:
            x = np.mean(x, axis=0)

```

```
x = self.decoder(x)
return nn.log_softmax(x, axis=-1)
```

In Flax we add the batch dimension as a lifted transformation. We need to route through several variable collections which handle RNN and parameter caching (described below).

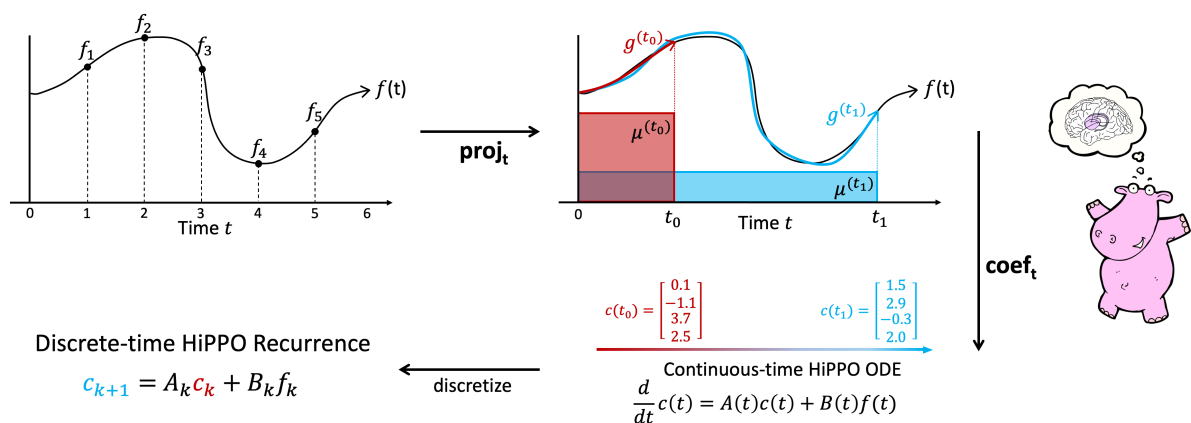
```
BatchStackedModel = nn.vmap(
    StackedModel,
    in_axes=0,
    out_axes=0,
    variable_axes={"params": None, "dropout": None, "cache": 0, "prime":
None},
    split_rngs={"params": False, "dropout": True},
)
```

Overall, this defines a sequence-to-sequence map of shape (batch size, sequence length, hidden dimension), exactly the signature exposed by related sequence models such as Transformers, RNNs, and CNNs.

Full code for training is defined in training.py.

While we now have our main model, there are *two core problems with SSMs*. First, the randomly initialized SSM actually does not perform very well. Furthermore, computing it naively like we've done so far is really slow and memory inefficient. Next, we'll complete our discussion of the modeling aspect of S4 by defining a special initialization for long-range dependencies, and then figure out how to compute this SSM Layer faster – a lot faster (Part 2)!

Part 1b: Addressing Long-Range Dependencies with HiPPO



Prior work found that the basic SSM actually performs very poorly in practice. Intuitively, one explanation is that they suffer from gradients scaling exponentially in the sequence length (i.e., the vanishing/exploding gradients problem). To address this problem, previous work developed the HiPPO theory of continuous-time memorization.

HiPPO specifies a class of certain matrices $\mathbf{A} \in \mathbb{R}^{N \times N}$ that when incorporated, allow the state $x(t)$ to memorize the history of the input $u(t)$. The most important matrix in this class is defined by the HiPPO matrix.

$$(\text{HiPPO Matrix}) \quad \mathbf{A}_{nk} = \begin{cases} (2n+1)^{1/2}(2k+1)^{1/2} & \text{if } n > k \\ n+1 & \text{if } n = k \\ 0 & \text{if } n < k \end{cases}$$

Previous work found that simply modifying an SSM from a random matrix \mathbf{A} to HiPPO improved its performance on the sequential MNIST classification benchmark from 60% to 98%.

This matrix is going to be really important, but it is a bit of magic. For our purposes we mainly need to know that: 1) we only need to calculate it once, and 2) it has a nice, simple structure (which we will exploit in part 2). Without going into the ODE math, the main takeaway is that this matrix aims to compress the past history into a state that has enough information to approximately reconstruct the history.

```
def make_HiPPO(N):
    P = np.sqrt(1 + 2 * np.arange(N))
    A = P[:, np.newaxis] * P[np.newaxis, :]
    A = np.tril(A) - np.diag(np.arange(N))
    return -A
```

Diving a bit deeper, the intuitive explanation of this matrix is that it produces a hidden state that memorizes its history. It does this by keeping track of the coefficients of a Legendre polynomial. These coefficients let it approximate all of the previous history. Let us look at an example,

```
def example_legendre(N=8):
    # Random hidden state as coefficients
    import numpy as np
    import numpy.polynomial.legendre

    x = (np.random.rand(N) - 0.5) * 2
    t = np.linspace(-1, 1, 100)
    f = numpy.polynomial.legendre.Legendre(x)(t)

    # Plot
    import matplotlib.pyplot as plt
    import seaborn

    seaborn.set_context("talk")
    fig = plt.figure(figsize=(20, 10))
    ax = fig.gca(projection="3d")
    ax.plot(
```

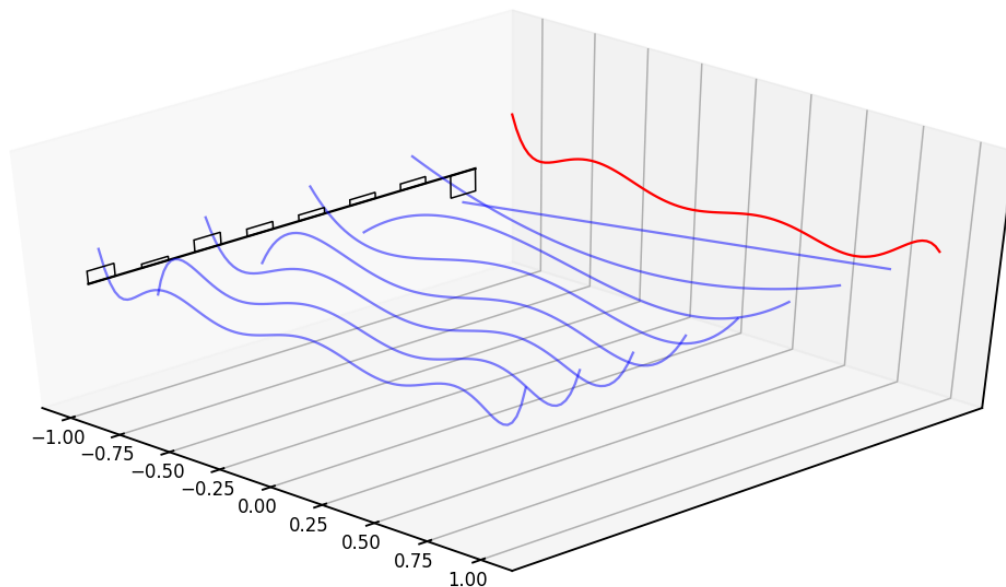
```

np.linspace(-25, (N - 1) * 100 + 25, 100),
[0] * 100,
zs=-1,
zdir="x",
color="black",
)
ax.plot(t, f, zs=N * 100, zdir="y", c="r")
for i in range(N):
    coef = [0] * N
    coef[N - i - 1] = 1
    ax.set_zlim(-4, 4)
    ax.set_yticks([])
    ax.set_zticks([])
    # Plot basis function.
    f = numpy.polynomial.legendre.Legendre(coef)(t)
    ax.bar(
        [100 * i],
        [x[i]],
        zs=-1,
        zdir="x",
        label="x%d" % i,
        color="brown",
        fill=False,
        width=50,
    )
    ax.plot(t, f, zs=100 * i, zdir="y", c="b", alpha=0.5)
ax.view_init(elev=40.0, azim=-45)
fig.savefig("images/leg.png")

if False:
    example_legendre()

```

The red line represents that curve we are approximating, while the black bars represent the values of our hidden state. Each is a coefficient for one element of the Legendre series shown as blue functions. The intuition is that the HiPPO matrix updates these coefficients each step.



Part 2: Implementing S4

Warning: this section has a lot of math. Roughly it boils down to finding a way to compute the filter from Part 1 for “HiPPO-like” matrices *really fast*. If you are interested, the details are really neat. If not, skip to Part 3 for some cool applications like MNIST completion.

Skip Button

To set the stage, recall that S4 has two main differences from a basic SSM. The first addresses a *modeling challenge* – long-range dependencies – by using a special formula for the \mathbf{A} matrix defined in the previous part. These special SSMs were considered in predecessor works to S4.

The second main feature of S4 solves the *computational challenge* of SSMs by introducing a special representation and algorithm to be able to work with this matrix!

The fundamental bottleneck in computing the discrete-time SSM is that it involves repeated matrix multiplication by $\overline{\mathbf{A}}$. For example, computing naively involves L successive multiplications by $\overline{\mathbf{A}}$, requiring $O(N^2L)$ operations and $O(NL)$ space.

Specifically, recall this function here:

```
def K_conv(Ab, Bb, Cb, L):
    return np.array(
        [(Cb @ matrix_power(Ab, l) @ Bb).reshape() for l in range(L)]
    )
```

The contribution of S4 is a stable method for speeding up this particular operation. To do this we are going to focus on the case where the SSM has special structure: specifically, Diagonal Plus Low-Rank (DPLR) in complex space.

A DPLR SSM is $(\mathbf{A} - \mathbf{PQ}^*, \mathbf{B}, \mathbf{C})$ for some diagonal \mathbf{A} and matrices $\mathbf{P}, \mathbf{Q}, \mathbf{B}, \mathbf{C} \in \mathbb{C}^{N \times 1}$. We assume without loss of generality that the rank is 1, i.e. these matrices are vectors.

Under this DPLR assumption, S4 overcomes the speed bottleneck in three steps

1. Instead of computing $\overline{\mathbf{K}}$ directly, we compute its spectrum by evaluating its **truncated generating function**. This now involves a matrix *inverse* instead of *power*.
2. We show that the diagonal matrix case is equivalent to the computation of a **Cauchy kernel** $\frac{1}{\omega_j - \zeta_k}$.
3. We show the low-rank term can now be corrected by applying the **Woodbury identity** which reduces $(\mathbf{A} + \mathbf{PQ}^*)^{-1}$ in terms of \mathbf{A}^{-1} , truly reducing to the diagonal case.

Step 1. SSM Generating Functions

The main step will be switching from computing the sequence to computing its generating function. From the paper's appendix:

To address the problem of computing powers of $\overline{\mathbf{A}}$, we introduce another technique. Instead of computing the SSM convolution filter $\overline{\mathbf{K}}$ directly, we introduce a generating function on its coefficients and compute evaluations of it.

The *truncated SSM generating function* at node z with truncation L is

$$\hat{\mathcal{K}}_L(z; \overline{\mathbf{A}}, \overline{\mathbf{B}}, \overline{\mathbf{C}}) \in \mathbb{C} := \sum_{i=0}^{L-1} \overline{\mathbf{C}} \mathbf{A}^i \overline{\mathbf{B}} z^i$$

```
def K_gen_simple(Ab, Bb, Cb, L):
    K = K_conv(Ab, Bb, Cb, L)

    def gen(z):
        return np.sum(K * (z ** np.arange(L)))

    return gen
```

The generating function essentially converts the SSM convolution filter from the time domain to frequency domain. This transformation is also called z-transform (up to a minus sign) in control engineering literature. Importantly, it preserves the same information, and the desired SSM convolution filter can be recovered. Once the z-transform of a discrete sequence known, we can obtain the filter's discrete fourier transform from evaluations of its z-transform at the roots of unity $\Omega = \{\exp(2\pi \frac{k}{L}) :$

$k \in [L]\}$. Then, we can apply inverse fourier transformation, stably in $O(L \log L)$ operations by applying an FFT, to recover the filter.

```
def conv_from_gen(gen, L):
    # Evaluate at roots of unity
    # Generating function is (-)z-transform, so we evaluate at (-)root
    Omega_L = np.exp((-2j * np.pi) * (np.arange(L) / L))
    atRoots = jax.vmap(gen)(Omega_L)
    # Inverse FFT
    out = np.fft.ifft(atRoots, L).reshape(L)
    return out.real
```

More importantly, in the generating function we can replace the matrix power with an inverse!

$$\hat{\mathcal{K}}_L(z) = \sum_{i=0}^{L-1} \overline{\mathbf{C}} \mathbf{A}^i \overline{\mathbf{B}} z^i = \overline{\mathbf{C}} (\mathbf{I} - \overline{\mathbf{A}}^L z^L) (\mathbf{I} - \overline{\mathbf{A}} z)^{-1} \overline{\mathbf{B}} = \widetilde{\mathbf{C}} (\mathbf{I} - \overline{\mathbf{A}} z)^{-1} \overline{\mathbf{B}}$$

And for all $z \in \Omega_L$, we have $z^L = 1$ so that term is removed. We then pull this constant term into a new $\widetilde{\mathbf{C}}$. Critically, this function **does not** call `K_conv`,

```
def K_gen_inverse(Ab, Bb, Cb, L):
    I = np.eye(Ab.shape[0])
    Ab_L = matrix_power(Ab, L)
    Ct = Cb @ (I - Ab_L)
    return lambda z: (Ct.conj() @ inv(I - Ab * z) @ Bb).reshape()
```

But it does output the same values,

```
def test_gen_inverse(L=16, N=4):
    ssm = random_SSM(rng, N)
    ssm = discretize(*ssm, 1.0 / L)
    b = K_conv(*ssm, L=L)

    a = conv_from_gen(K_gen_inverse(*ssm, L=L), L)
    assert np.allclose(a, b)
```

In summary, Step 1 allows us to replace the matrix power with an inverse by utilizing a truncated generating function. However this inverse still needs to be calculated L times (for each of the roots of unity).

Step 2: Diagonal Case

The next step to assume special *structure* on the matrix \mathbf{A} to compute the inverse faster than the naive inversion. To begin, let us first convert the equation above to use the original SSM matrices. With some algebra you can expand the discretization and show:

$$\widetilde{\mathbf{C}} (\mathbf{I} - \overline{\mathbf{A}})^{-1} \overline{\mathbf{B}} = \frac{2\Delta}{1+z} \widetilde{\mathbf{C}} \left[2 \frac{1-z}{1+z} - \Delta \mathbf{A} \right]^{-1} \mathbf{B}$$

Now imagine $\mathbf{A} = \mathbf{\Lambda}$ for a diagonal $\mathbf{\Lambda}$. Substituting in the discretization formula the authors show that the generating function can be written in the following manner:

$$\hat{\mathbf{K}}_{\mathbf{\Lambda}}(z) = c(z) \sum_i \cdot \frac{\widetilde{\mathbf{C}}_i \mathbf{B}_i}{(g(z) - \mathbf{\Lambda}_i)} = c(z) \cdot k_{z,\mathbf{\Lambda}}(\widetilde{\mathbf{C}}, \mathbf{B})$$

where \mathbf{C} is a constant, and g is a function of z .

We have effectively replaced an inverse with a weighted dot product. Let's make a small helper function to compute this weight dot product for use.

```
def cauchy_dot(v, omega, lambd):
    return (v / (omega - lambd)).sum()
```

While not important for our implementation, it is worth noting that this is a Cauchy kernel and is the subject of many other fast implementations.

Step 3: Diagonal Plus Low-Rank

The final step is to relax the diagonal assumption. In addition to the diagonal term we allow a low-rank component with $\mathbf{P}, \mathbf{Q} \in \mathbb{C}^{N \times 1}$ such that:

$$\mathbf{A} = \mathbf{\Lambda} - \mathbf{P}\mathbf{Q}^*$$

The Woodbury identity tells us that the inverse of a diagonal plus rank-1 term is equal to the inverse of the diagonal plus a rank-1 term. We write it out here adding the low-rank term.

$$(\mathbf{\Lambda} + \mathbf{P}\mathbf{Q}^*)^{-1} = \mathbf{\Lambda}^{-1} - \mathbf{\Lambda}^{-1} \mathbf{P} (1 + \mathbf{Q}^* \mathbf{\Lambda}^{-1} \mathbf{P})^{-1} \mathbf{Q}^* \mathbf{\Lambda}^{-1}$$

There is a bunch of algebra in the appendix. It mostly consists of substituting this component in for \mathbf{A} , applying the Woodbury identity and distributing terms. We end up with 4 terms that all look like Step 2 above:

$$\hat{\mathbf{K}}_{DPLR}(z) = c(z) [k_{z,\mathbf{\Lambda}}(\widetilde{\mathbf{C}}, \mathbf{B}) - k_{z,\mathbf{\Lambda}}(\widetilde{\mathbf{C}}, \mathbf{P}) (1 + k_{z,\mathbf{\Lambda}}(\mathbf{q}^*, \mathbf{P}))^{-1} k_{z,\mathbf{\Lambda}}(\mathbf{q}^*, \mathbf{B})]$$

The code consists of collecting up the terms and applying 4 weighted dot products,

```
def K_gen_DPLR(Lambda, P, Q, B, C, step, unmat=False):
    aterm = (C.conj(), Q.conj())
    bterm = (B, P)

    def gen(o):
        g = (2.0 / step) * ((1.0 - o) / (1.0 + o))
        c = 2.0 / (1.0 + o)
```

```

def k(a):
    # Checkpoint this calculation for memory efficiency.
    if unmat:
        return jax.remat(cauchy_dot)(a, g, Lambda)
    else:
        return cauchy_dot(a, g, Lambda)

k00 = k(aterm[0] * bterm[0])
k01 = k(aterm[0] * bterm[1])
k10 = k(aterm[1] * bterm[0])
k11 = k(aterm[1] * bterm[1])
return c * (k00 - k01 * (1.0 / (1.0 + k11)) * k10)

return gen

```

This is our final version of the K function. Because `conv_from_gen` is always called together with a generating function (e.g. `K_gen_DPLR`), we'll fuse them into define a dedicated function to compute the DPLR SSM kernel from all of its parameters. (With fewer layers of indirection, this could also make it easier for XLA compiler to optimize.)

```

@jax.jit
def cauchy(v, omega, lambd):
    """Cauchy matrix multiplication: (n), (l), (n) -> (l)"""
    cauchy_dot = lambda _omega: (v / (_omega - lambd)).sum()
    return jax.vmap(cauchy_dot)(omega)

def kernel_DPLR(Lambda, P, Q, B, C, step, L):
    # Evaluate at roots of unity
    # Generating function is (-)z-transform, so we evaluate at (-)root
    Omega_L = np.exp((-2j * np.pi) * (np.arange(L) / L))

    aterm = (C.conj(), Q.conj())
    bterm = (B, P)

    g = (2.0 / step) * ((1.0 - Omega_L) / (1.0 + Omega_L))
    c = 2.0 / (1.0 + Omega_L)

    # Reduction to core Cauchy kernel
    k00 = cauchy(aterm[0] * bterm[0], g, Lambda)
    k01 = cauchy(aterm[0] * bterm[1], g, Lambda)
    k10 = cauchy(aterm[1] * bterm[0], g, Lambda)
    k11 = cauchy(aterm[1] * bterm[1], g, Lambda)
    atRoots = c * (k00 - k01 * (1.0 / (1.0 + k11)) * k10)
    out = np.fft.ifft(atRoots, L).reshape(L)
    return out.real

```

Now we can check whether it worked. First, let's generate a random Diagonal Plus Low Rank (DPLR) matrix,

```
def random_DPLR(rng, N):
    l_r, p_r, q_r, b_r, c_r = jax.random.split(rng, 5)
    Lambda = jax.random.uniform(l_r, (N,))
    P = jax.random.uniform(p_r, (N,))
    Q = jax.random.uniform(q_r, (N,))
    B = jax.random.uniform(b_r, (N, 1))
    C = jax.random.uniform(c_r, (1, N))
    return Lambda, P, Q, B, C
```

We can check that the DPLR method yields the same filter as computing \mathbf{A} directly,

```
def test_gen_dplr(L=16, N=4):
    I = np.eye(4)

    # Create a DPLR A matrix and discretize
    Lambda, P, B, _ = make_DPLR_HiPP0(N)
    A = np.diag(Lambda) - P[:, np.newaxis] @ P[:, np.newaxis].conj().T
    _, _, C = random_SSM(rng, N)

    Ab, Bb, Cb = discretize(A, B, C, 1.0 / L)
    a = K_conv(Ab, Bb, Cb.conj(), L=L)

    # Compare to the DPLR generating function approach.
    C = (I - matrix_power(Ab, L)).conj().T @ Cb.ravel()
    b = kernel_DPLR(Lambda, P, P, B, C, step=1.0 / L, L=L)
    assert np.allclose(a.real, b.real)
```

Diagonal Plus Low-Rank RNN.

A secondary benefit of the DPLR factorization is that it allows us to compute the discretized form of the SSM without having to invert the \mathbf{A} matrix directly. Here we return to the paper for the derivation.

Recall that discretization computes,

$$\begin{aligned}\overline{\mathbf{A}} &= (\mathbf{I} - \Delta/2 \cdot \mathbf{A})^{-1}(\mathbf{I} + \Delta/2 \cdot \mathbf{A}) \\ \overline{\mathbf{B}} &= (\mathbf{I} - \Delta/2 \cdot \mathbf{A})^{-1}\Delta\mathbf{B}.\end{aligned}$$

We simplify both terms in the definition of $\bar{\mathbf{A}}$ independently. The first term is:

$$\begin{aligned}\mathbf{I} + \frac{\Delta}{2}\mathbf{A} &= \mathbf{I} + \frac{\Delta}{2}(\mathbf{\Lambda} - \mathbf{P}\mathbf{Q}^*) \\ &= \frac{\Delta}{2} \left[\frac{2}{\Delta}\mathbf{I} + (\mathbf{\Lambda} - \mathbf{P}\mathbf{Q}^*) \right] \\ &= \frac{\Delta}{2}\mathbf{A}_0\end{aligned}$$

where \mathbf{A}_0 is defined as the term in the final brackets.

The second term is known as the Backward Euler's method. Although this inverse term is normally difficult to deal with, in the DPLR case we can simplify it using Woodbury's Identity as described above.

$$\begin{aligned}\left(\mathbf{I} - \frac{\Delta}{2}\mathbf{A}\right)^{-1} &= \left(\mathbf{I} - \frac{\Delta}{2}(\mathbf{\Lambda} - \mathbf{P}\mathbf{Q}^*)\right)^{-1} \\ &= \frac{2}{\Delta} \left[\frac{2}{\Delta} - \mathbf{\Lambda} + \mathbf{P}\mathbf{Q}^* \right]^{-1} \\ &= \frac{2}{\Delta} \left[\mathbf{D} - \mathbf{D}\mathbf{P}(1 + \mathbf{Q}^*\mathbf{D}\mathbf{P})^{-1}\mathbf{Q}^*\mathbf{D} \right] \\ &= \frac{2}{\Delta}\mathbf{A}_1\end{aligned}$$

where $\mathbf{D} = \left(\frac{2}{\Delta} - \mathbf{\Lambda}\right)^{-1}$ and \mathbf{A}_1 is defined as the term in the final brackets.

The discrete-time SSM becomes

$$\begin{aligned}\mathbf{x}_k &= \bar{\mathbf{A}}\mathbf{x}_{k-1} + \bar{\mathbf{B}}\mathbf{u}_k \\ &= \mathbf{A}_1\mathbf{A}_0\mathbf{x}_{k-1} + 2\mathbf{A}_1\mathbf{B}\mathbf{u}_k \\ \mathbf{y}_k &= \mathbf{C}\mathbf{x}_k.\end{aligned}$$

```
def discrete_DPLR(Lambda, P, Q, B, C, step, L):
    # Convert parameters to matrices
    B = B[:, np.newaxis]
    Ct = C[np.newaxis, :]

    N = Lambda.shape[0]
    A = np.diag(Lambda) - P[:, np.newaxis] @ Q[:, np.newaxis].conj().T
    I = np.eye(N)

    # Forward Euler
    A0 = (2.0 / step) * I + A

    # Backward Euler
```

```

D = np.diag(1.0 / ((2.0 / step) - Lambda))
Qc = Q.conj().T.reshape(1, -1)
P2 = P.reshape(-1, 1)
A1 = D - (D @ P2 * (1.0 / (1 + (Qc @ D @ P2))) * Qc @ D)

# A bar and B bar
Ab = A1 @ A0
Bb = 2 * A1 @ B

# Recover Cbar from Ct
Cb = Ct @ inv(I - matrix_power(Ab, L)).conj()
return Ab, Bb, Cb.conj()

```

Turning HiPPO to DPLR

This approach applies to DPLR matrices, but remember we would like it to also apply to the HiPPO matrix. While not DPLR in its current form, the HiPPO matrix *does have special structure*. It is Normal Plus Low-Rank (NPLR). Because normal matrices are exactly the class of matrices that are unitarily diagonalizable, NPLR matrices are essentially equivalent to DPLR matrices from the perspective of SSM models. this is just as good as DPLR for the purposes of learning an SSM network.

The S4 techniques can apply to any matrix \mathbf{A} that can be decomposed as *Normal Plus Low-Rank (NPLR)*.

$$\mathbf{A} = \mathbf{V} \mathbf{\Lambda} \mathbf{V}^* - \mathbf{P} \mathbf{Q}^\top = \mathbf{V} (\mathbf{\Lambda} - \mathbf{V}^* \mathbf{P} (\mathbf{V}^* \mathbf{Q})^*) \mathbf{V}^*$$

for unitary $\mathbf{V} \in \mathbb{C}^{N \times N}$, diagonal $\mathbf{\Lambda}$, and low-rank factorization $\mathbf{P}, \mathbf{Q} \in \mathbb{R}^{N \times r}$. An NPLR SSM is therefore unitarily equivalent to some DPLR matrix.

For S4, we need to work with a HiPPO matrix for \mathbf{A} . This requires first writing it as a normal plus low-rank term, and then diagonalizing to extract $\mathbf{\Lambda}$ from this decomposition. The appendix of the paper shows how by writing the normal part as a skew-symmetric (plus a constant times the identity matrix), which are a special class of normal matrices.

An additional simplification is that there is actually a representation that ties the low-rank components terms $\mathbf{P} = \mathbf{Q}$, which was shown in follow-up work to be important for stability.

```

def make_NPLR_HiPPO(N):
    # Make -HiPPO
    nhippo = make_HiPPO(N)

    # Add in a rank 1 term. Makes it Normal.
    P = np.sqrt(np.arange(N) + 0.5)

    # HiPPO also specifies the B matrix

```

```
B = np.sqrt(2 * np.arange(N) + 1.0)
return nhippo, P, B
```

After extracting the normal part, we can diagonalize to get out the DPLR terms. Because the normal part is actually skew-symmetric, we can extract the real and complex parts of Λ separately. This serves two purposes. First, this gives us finer-grained control over the real and imaginary parts, which can be used to improve stability. Second, this lets us use more powerful diagonalization algorithms for Hermitian matrices – in fact, the current version of JAX does not support GPU diagonalization for non-Hermitian matrices!

```
def make_DPLR_HiPP0(N):
    """Diagonalize NPLR representation"""
    A, P, B = make_NPLR_HiPP0(N)

    S = A + P[:, np.newaxis] * P[np.newaxis, :]

    # Check skew symmetry
    S_diag = np.diagonal(S)
    Lambda_real = np.mean(S_diag) * np.ones_like(S_diag)
    # assert np.allclose(Lambda_real, S_diag, atol=1e-3)

    # Diagonalize S to V \Lambda V^*
    Lambda_imag, V = eigh(S * -1j)

    P = V.conj().T @ P
    B = V.conj().T @ B
    return Lambda_real + 1j * Lambda_imag, P, B, V
```

Sanity check just to make sure those identities hold,

```
def test_nplr(N=8):
    A2, P, B = make_NPLR_HiPP0(N)
    Lambda, Pc, Bc, V = make_DPLR_HiPP0(N)
    Vc = V.conj().T
    P = P[:, np.newaxis]
    Pc = Pc[:, np.newaxis]
    Lambda = np.diag(Lambda)

    A3 = V @ Lambda @ Vc - (P @ P.T) # Test NPLR
    A4 = V @ (Lambda - Pc @ Pc.conj().T) @ Vc # Test DPLR
    assert np.allclose(A2, A3, atol=1e-4, rtol=1e-4)
    assert np.allclose(A2, A4, atol=1e-4, rtol=1e-4)
```

Final Check

This tests that everything works as planned.


```
def test_conversion(N=8, L=16):
    step = 1.0 / L
    # Compute a HiPP0 NPLR matrix.
    Lambda, P, B, _ = make_DPLR_HiPP0(N)
    # Random complex Ct
    C = normal(dtype=np.complex64)(rng, (N,))

    # CNN form.
    K = kernel_DPLR(Lambda, P, P, B, C, step, L)

    # RNN form.
    Ab, Bb, Cb = discrete_DPLR(Lambda, P, P, B, C, step, L)
    K2 = K_conv(Ab, Bb, Cb, L=L)
    assert np.allclose(K.real, K2.real, atol=1e-5, rtol=1e-5)

    # Apply CNN
    u = np.arange(L) * 1.0
    y1 = causal_convolution(u, K.real)

    # Apply RNN
    _, y2 = scan_SSM(
        Ab, Bb, Cb, u[:, np.newaxis], np.zeros((N,)).astype(np.complex64)
    )
    assert np.allclose(y1, y2.reshape(-1).real, atol=1e-4, rtol=1e-4)
```

Part 3: S4 in Practice

That was a lot of work, but now the actual model is concise. In fact we are only using four functions:

1. `K_gen_DPLR` → Truncated generating function when \mathbf{A} is DPLR (S4-part)
2. `conv_from_gen` → Convert generating function to filter
3. `causal_convolution` → Run convolution
4. `discretize_DPLR` → Convert SSM to discrete form for RNN.

S4 CNN / RNN Layer

A full S4 Layer is very similar to the simple SSM layer above. The only difference is in the computation of \mathbf{K} . Additionally instead of learning \mathbf{C} , we learn $\tilde{\mathbf{C}}$ so we avoid computing powers of \mathbf{A} . Note as well that in the original paper $\mathbf{\Lambda}$, \mathbf{P} , \mathbf{Q} are also learned. However, in this post, we leave them fixed for simplicity.

```
class S4Layer(nn.Module):
    N: int
    l_max: int
    decode: bool = False
```

```

    # Special parameters with multiplicative factor on lr and no weight
    decay (handled by main train script)
    lr = {
        "Lambda_re": 0.1,
        "Lambda_im": 0.1,
        "P": 0.1,
        "B": 0.1,
        "log_step": 0.1,
    }

    def setup(self):
        # Learned Parameters (C is complex!)
        init_A_re, init_A_im, init_P, init_B = hippo_initializer(self.N)
        self.Lambda_re = self.param("Lambda_re", init_A_re, (self.N,))
        self.Lambda_im = self.param("Lambda_im", init_A_im, (self.N,))
        # Ensure the real part of Lambda is negative
        # (described in the SaShiMi follow-up to S4)
        self.Lambda = np.clip(self.Lambda_re, None, -1e-4) + 1j *
self.Lambda_im
        self.P = self.param("P", init_P, (self.N,))
        self.B = self.param("B", init_B, (self.N,))
        # C should be init as standard normal
        # This doesn't work due to how JAX handles complex optimizers
https://github.com/deepmind/optax/issues/196
        # self.C = self.param("C", normal(stddev=1.0, dtype=np.complex64),
(self.N,))
        self.C = self.param("C", normal(stddev=0.5**0.5), (self.N, 2))
        self.C = self.C[..., 0] + 1j * self.C[..., 1]
        self.D = self.param("D", nn.initializers.ones, (1,))
        self.step = np.exp(self.param("log_step", log_step_initializer(),
(1,)))

        if not self.decode:
            # CNN mode, compute kernel.
            self.K = kernel_DPLR(
                self.Lambda,
                self.P,
                self.P,
                self.B,
                self.C,
                self.step,
                self.l_max,
            )

        else:
            # RNN mode, discretize

            # Flax trick to cache discrete form during decoding.
            def init_discrete():

```

```

        return discrete_DPLR(
            self.Lambda,
            self.P,
            self.P,
            self.B,
            self.C,
            self.step,
            self.l_max,
        )

    ssm_var = self.variable("prime", "ssm", init_discrete)
    if self.is_mutable_collection("prime"):
        ssm_var.value = init_discrete()
    self.ssm = ssm_var.value

    # RNN Cache
    self.x_k_1 = self.variable(
        "cache", "cache_x_k", np.zeros, (self.N,), np.complex64
    )

    def __call__(self, u):
        # This is identical to SSM Layer
        if not self.decode:
            # CNN Mode
            return causal_convolution(u, self.K) + self.D * u
        else:
            # RNN Mode
            x_k, y_s = scan_SSM(*self.ssm, u[:, np.newaxis],
self.x_k_1.value)
            if self.is_mutable_collection("cache"):
                self.x_k_1.value = x_k
            return y_s.reshape(-1).real + self.D * u

```

```
S4Layer = cloneLayer(S4Layer)
```

We initialize the model by computing a HiPPO DPLR initializer

```

# Factory for constant initializer in Flax
def init(x):
    def _init(key, shape):
        assert shape == x.shape
        return x

    return _init

def hippo_initializer(N):
    Lambda, P, B, _ = make_DPLR_HiPPO(N)

```

```
return init(Lambda.real), init(Lambda.imag), init(P), init(B)
```

Sampling and Caching

We can sample from the model using the RNN implementation. Here is what the sampling code looks like. Note that we keep a running cache to remember the RNN state.

```
def sample(model, params, prime, cache, x, start, end, rng):
    def loop(i, cur):
        x, rng, cache = cur
        r, rng = jax.random.split(rng)
        out, vars = model.apply(
            {"params": params, "prime": prime, "cache": cache},
            x[:, np.arange(1, 2) * i],
            mutable=["cache"],
        )

        def update(x, out):
            p = jax.random.categorical(r, out[0])
            x = x.at[i + 1, 0].set(p)
            return x

        x = jax.vmap(update)(x, out)
        return x, rng, vars["cache"].unfreeze()

    return jax.lax.fori_loop(start, end, jax.jit(loop), (x, rng, cache))[0]
```

To get this in a good form, we first precompute the discretized version of the the RNN for each S4 layers. We do this through the “prime” collection of variables.

```
def init_recurrence(model, params, init_x, rng):
    variables = model.init(rng, init_x)
    vars = {
        "params": params,
        "cache": variables["cache"].unfreeze(),
        "prime": variables["prime"].unfreeze(),
    }
    print("[*] Priming")
    _, prime_vars = model.apply(vars, init_x, mutable=["prime"])
    return vars["params"], prime_vars["prime"], vars["cache"]
```

Putting this altogether we can sample from a model directly.

```
def sample_checkpoint(path, model, length, rng):
    from flax.training import checkpoints

    start = np.zeros((1, length, 1), dtype=int)
```

```

print("[*] Initializing from checkpoint %s" % path)
state = checkpoints.restore_checkpoint(path, None)
assert "params" in state
params, prime, cache = init_recurrence(model, state["params"], start,
rng)
print("[*] Sampling output")
return sample(model, params, prime, cache, start, 0, length - 1, rng)

```

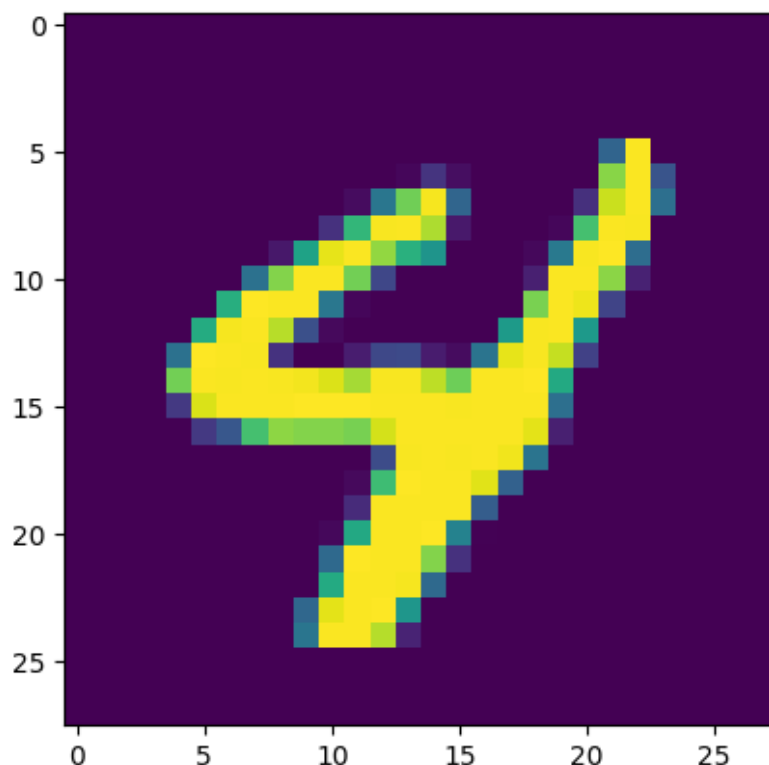
Experiments: MNIST

Now that we have the model, we can try it out on some MNIST experiments. For these experiments we linearize MNIST and just treat each image as a sequence of pixels.

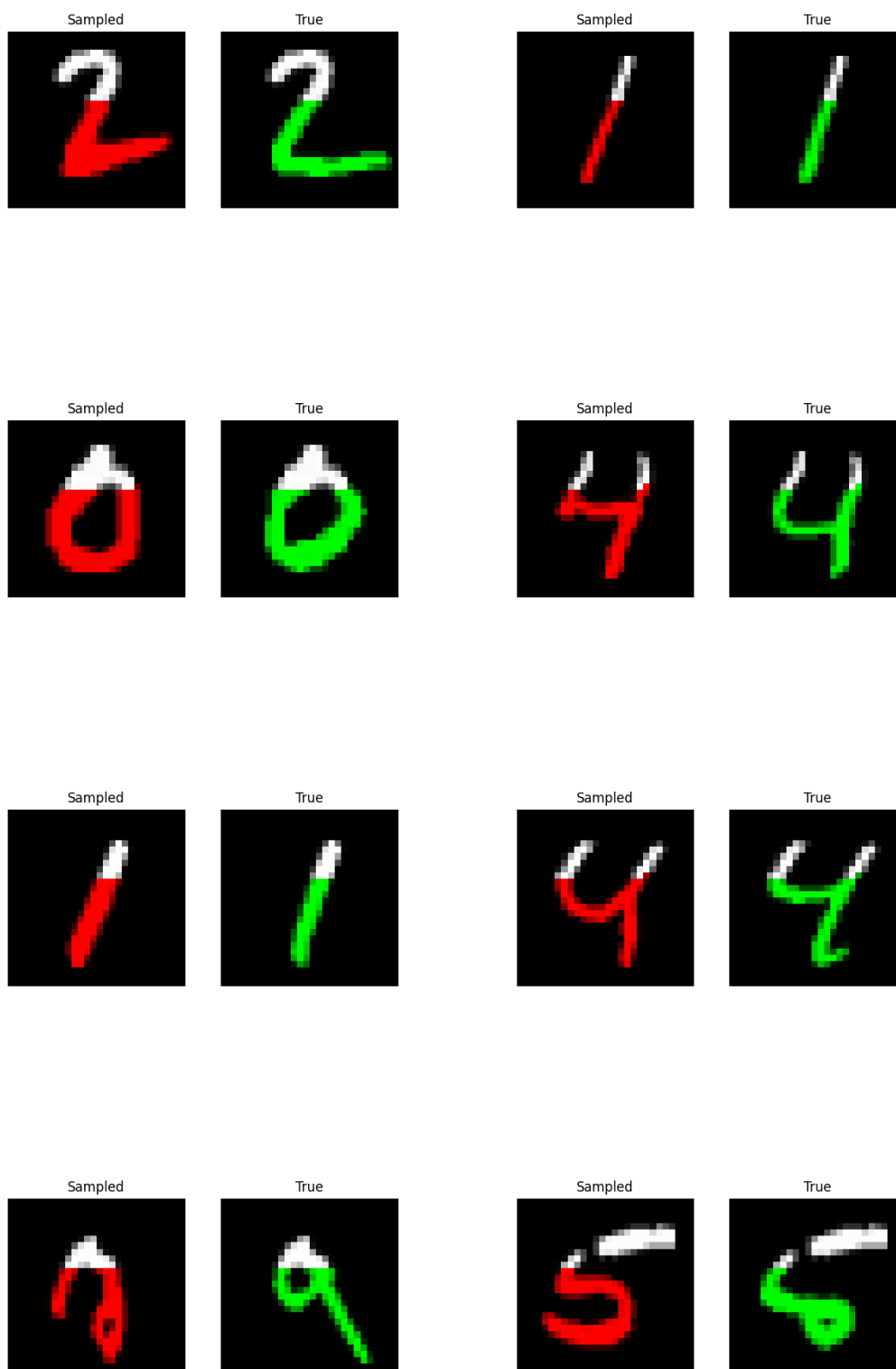
The first experiments we ran were on MNIST classification. While not in theory a hard problem, treating MNIST as a linear sequence classification task is a bit strange. However in practice, the model with $H = 256$ and four layers seems to get up near 99% right away.

A more visually interesting task is generating MNIST digits, by predicting entire sequences of pixels! Here, we simply feed in a sequence of pixels into the model and have it predict the next one like language modeling. With a little tweaking, we are able to get the model to an NLL of 0.36 on this task with size 512 and 6 layers (~4m parameters).

The metric usually used for this task is *bits per dimension* which is NLL in base 2 for MNIST. A loss of 0.36 is ~0.52 BPD which is SOTA according to PapersWithCode.



We can also do prefix-samples – given the first 300 pixels, try to complete the image. S4 is on the left, true on the right.



```
def sample_image_prefix(
    params,
    model,
    # length,
```

```

    rng,
    dataloader,
    prefix=300,
    # bsz=32,
    imshape=(28, 28),
    n_batches=None,
    save=True,
):
    """Sample a grayscale image represented as intensities in [0, 255]"""
    import matplotlib.pyplot as plt
    import numpy as onp

    # from .data import Datasets
    # BATCH = bsz
    # start = np.zeros((BATCH, length), dtype=int)
    # start = np.zeros((BATCH, length, 1), dtype=int)
    start = np.array(next(iter(dataloader))[0].numpy())
    start = np.zeros_like(start)
    # params, prime, cache = init_recurrence(model, params, start[:, :-1],
rng)
    params, prime, cache = init_recurrence(model, params, start, rng)

    BATCH = start.shape[0]
    START = prefix
    LENGTH = start.shape[1]
    assert LENGTH == onp.prod(imshape)

    # _, dataloader, _, _, _ = Datasets["mnist"](bsz=BATCH)
    it = iter(dataloader)
    for j, im in enumerate(it):
        if n_batches is not None and j >= n_batches:
            break

        image = im[0].numpy()
        image = np.pad(
            image[:, :-1, :], [(0, 0), (1, 0), (0, 0)], constant_values=0
        )
        cur = onp.array(image)
        # cur[:, START + 1 :, 0] = 0
        # cur = np.pad(cur[:, :-1, 0], [(0, 0), (1, 0)],
constant_values=256)
        cur = np.array(cur[:, :])

        # Cache the first `start` inputs.
        out, vars = model.apply(
            {"params": params, "prime": prime, "cache": cache},
            cur[:, np.arange(0, START)],
            mutable=["cache"],
        )
        cache = vars["cache"].unfreeze()

```

```

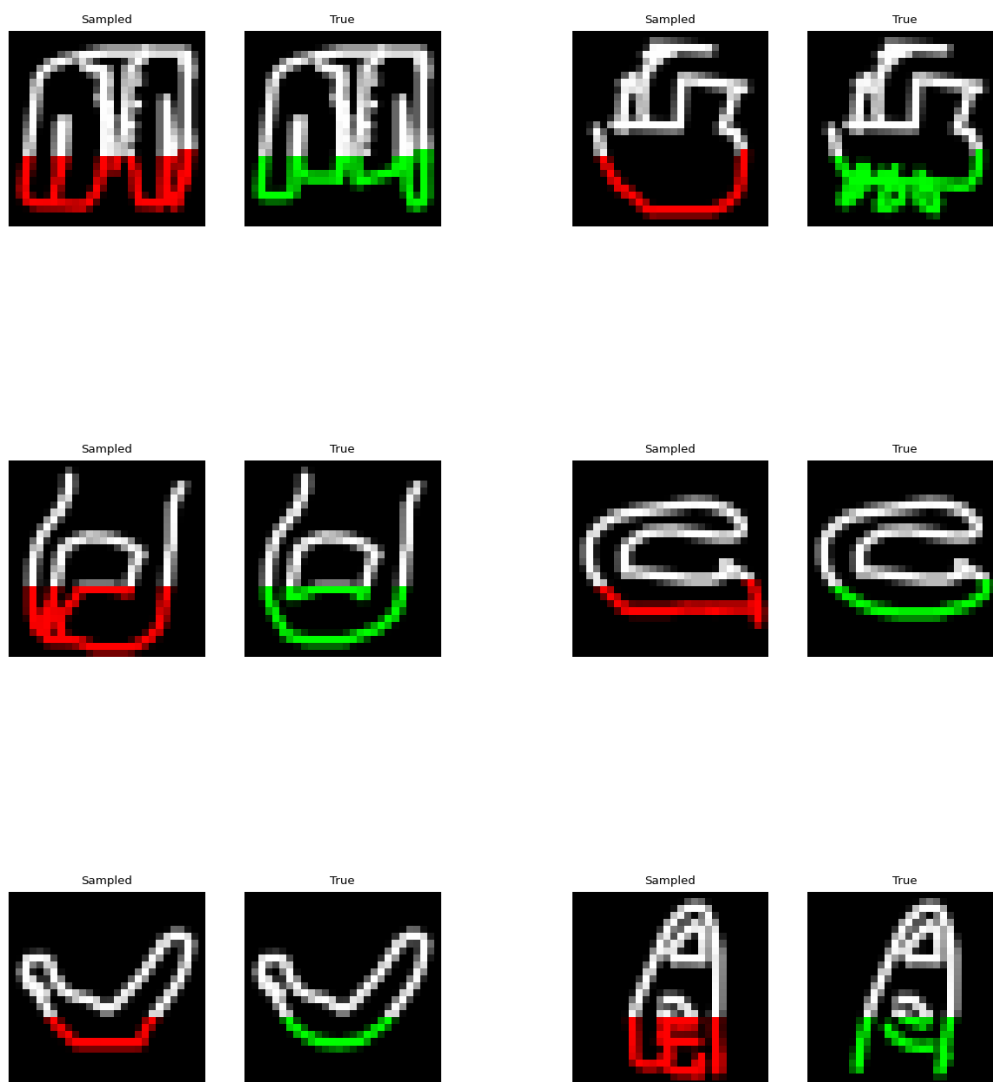
out = sample(model, params, prime, cache, cur, START, LENGTH - 1,
rng)

# Visualization
out = out.reshape(BATCH, *imshape)
final = onp.zeros((BATCH, *imshape, 3))
final2 = onp.zeros((BATCH, *imshape, 3))
final[:, :, :, 0] = out
f = final.reshape(BATCH, LENGTH, 3)
i = image.reshape(BATCH, LENGTH)
f[:, :, START, 1] = i[:, :, START]
f[:, :, START, 2] = i[:, :, START]
f = final2.reshape(BATCH, LENGTH, 3)
f[:, :, 1] = i
f[:, :, START, 0] = i[:, :, START]
f[:, :, START, 2] = i[:, :, START]
if save:
    for k in range(BATCH):
        fig, (ax1, ax2) = plt.subplots(ncols=2)
        ax1.set_title("Sampled")
        ax1.imshow(final[k] / 256.0)
        ax2.set_title("True")
        ax1.axis("off")
        ax2.axis("off")
        ax2.imshow(final2[k] / 256.0)
        fig.savefig("im%d.%d.png" % (j, k))
        plt.close()
    print(f"Sampled batch {j} image {k}")
return final, final2

```

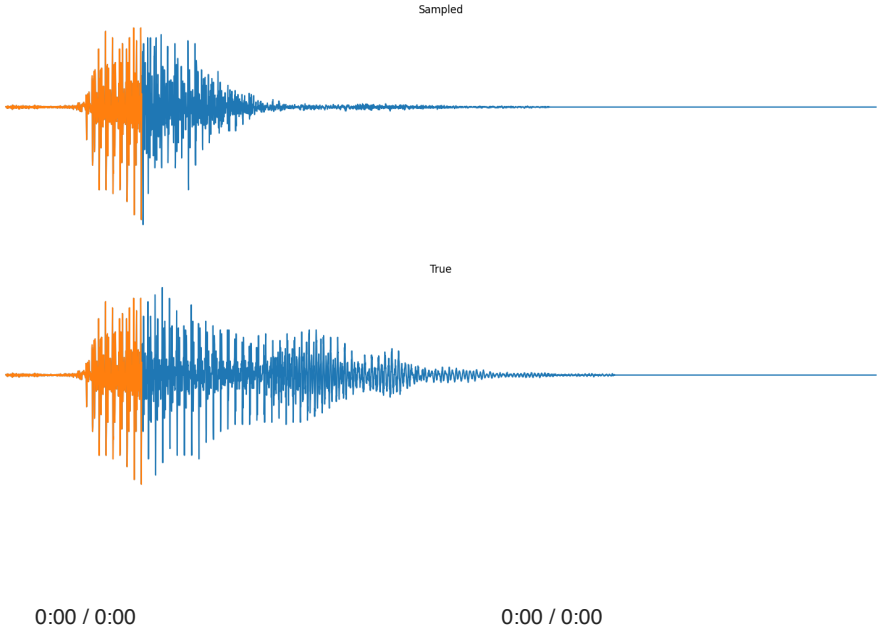
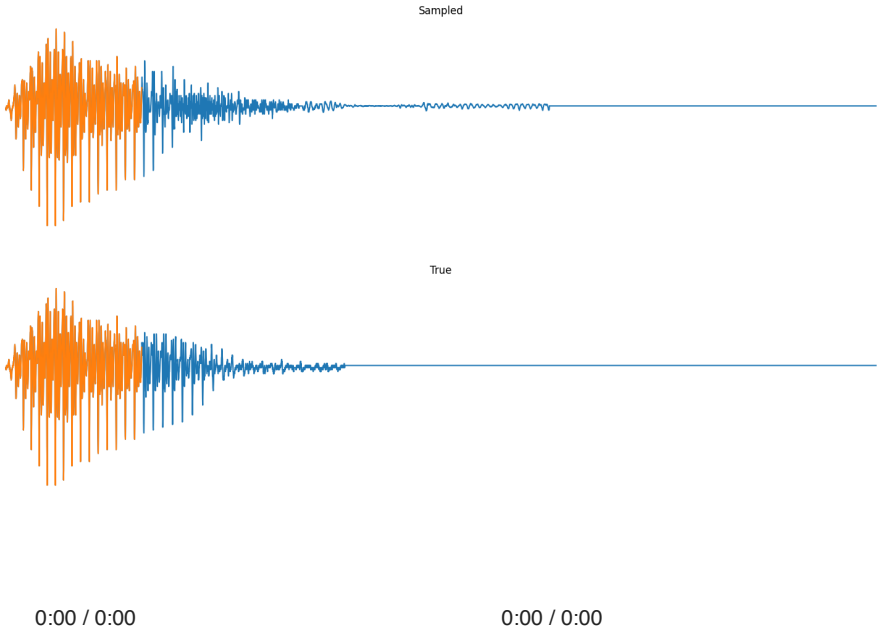
Experiments: QuickDraw

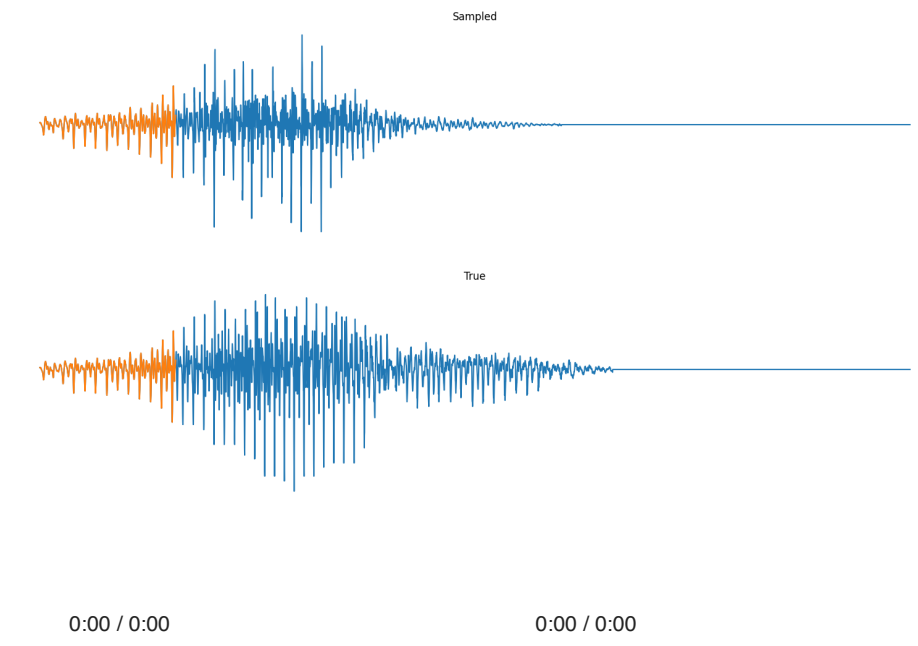
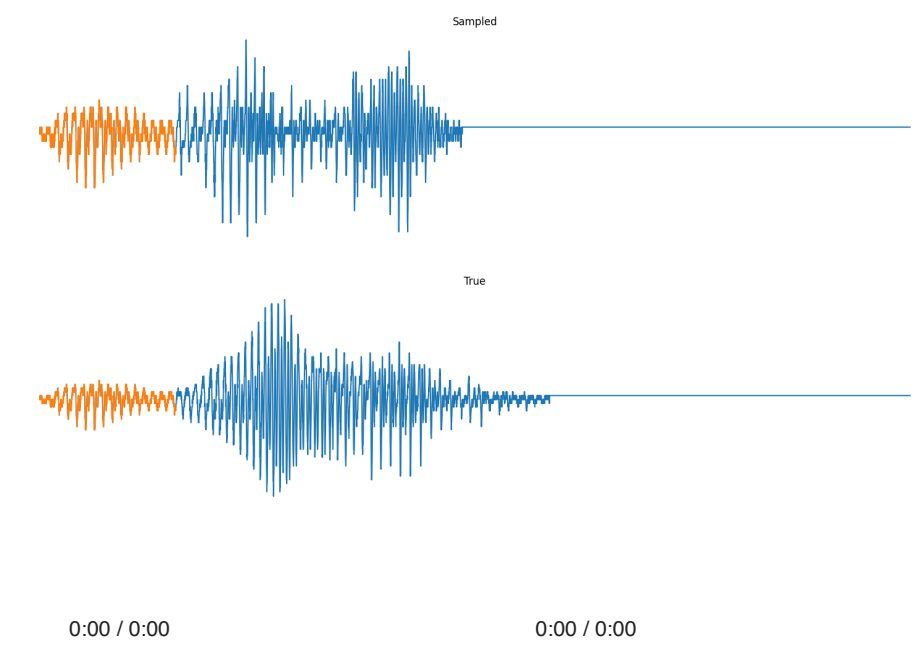
Next we tried training a model to generate drawings. For this we used the QuickDraw dataset. The dataset includes a version of the dataset downsampled to MNIST size so we can use roughly the same model as above. The dataset is much larger though (5M images) and more complex. We only trained for 1 epoch with a $H = 256$, 4 layer model. Still, the approach was able to generate relatively coherent completions. These are prefix samples with 500 pixels given.

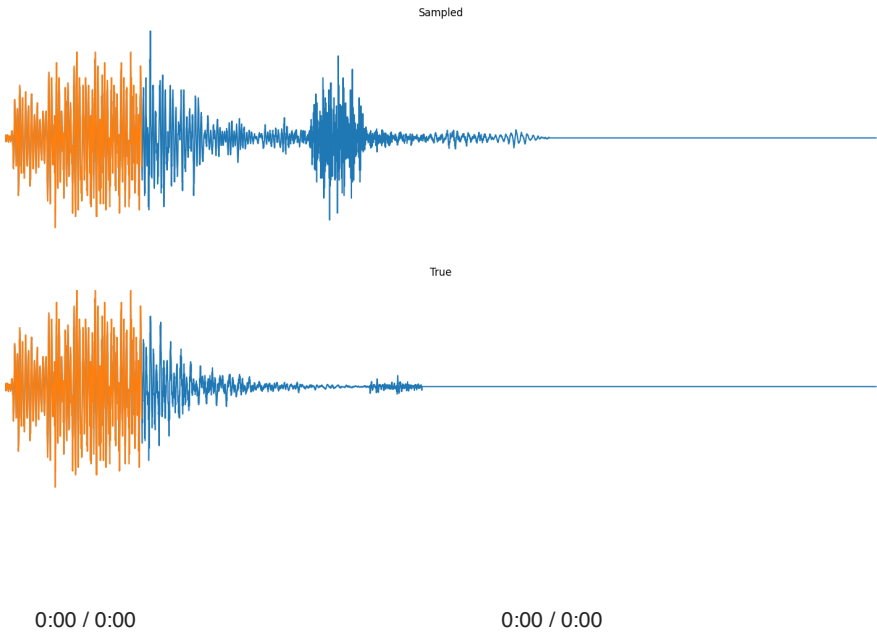
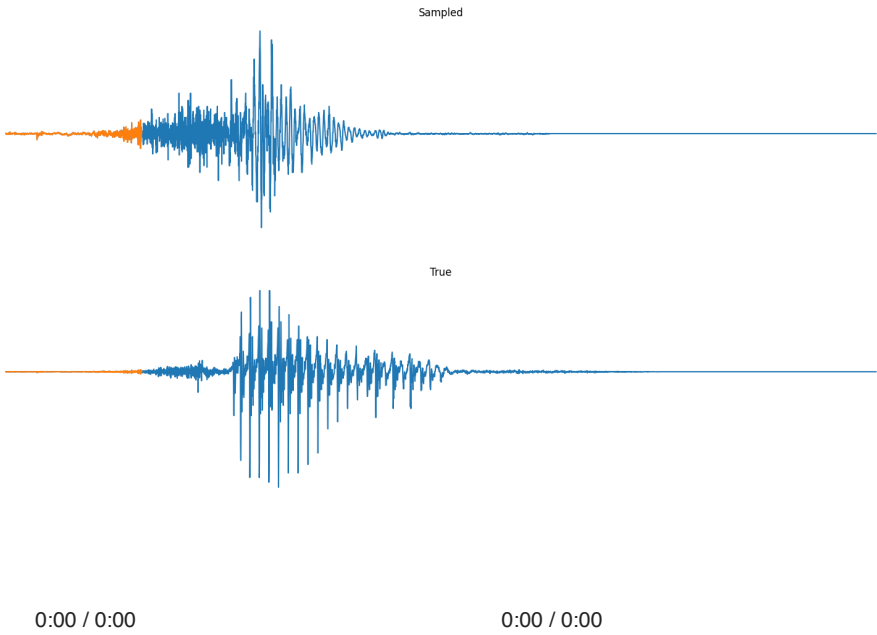


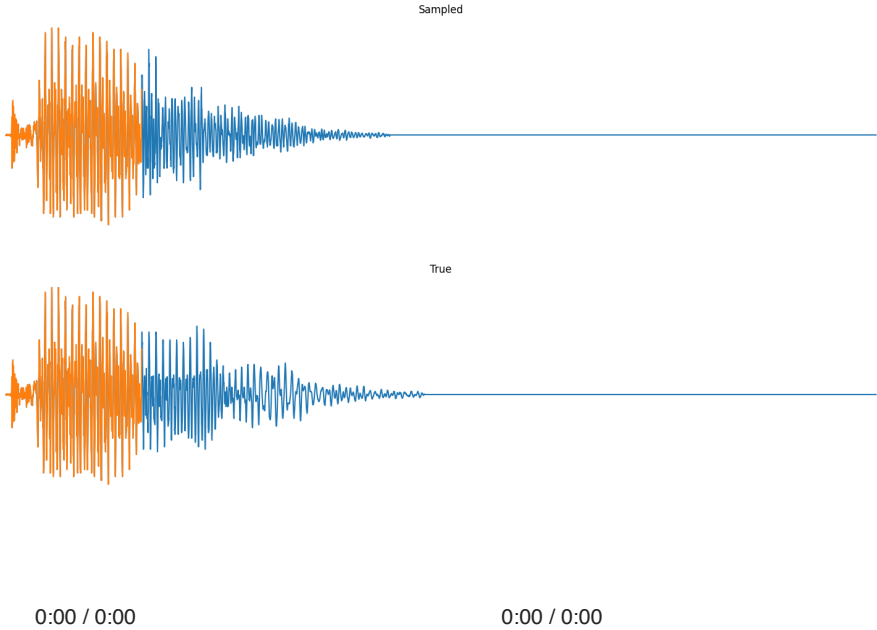
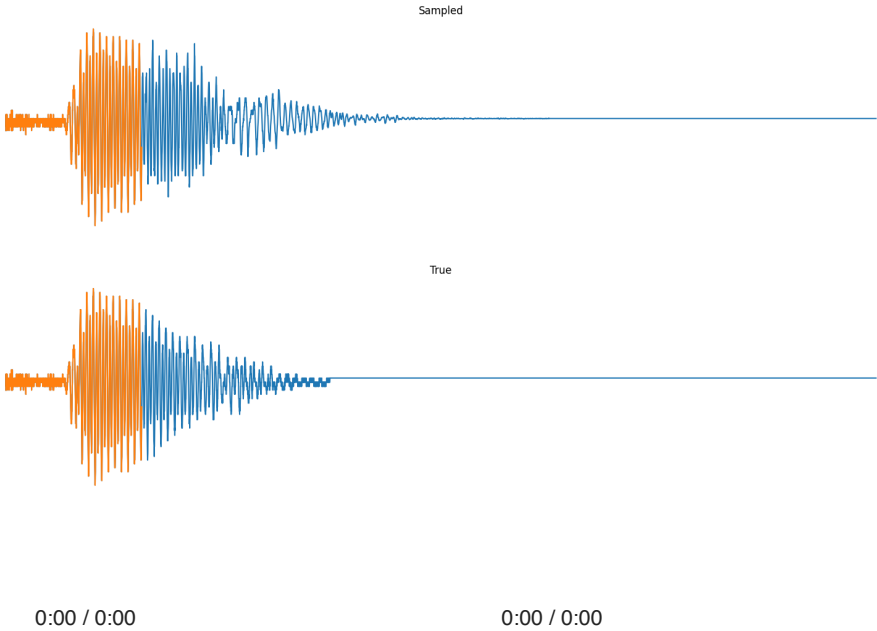
Experiments: Spoken Digits

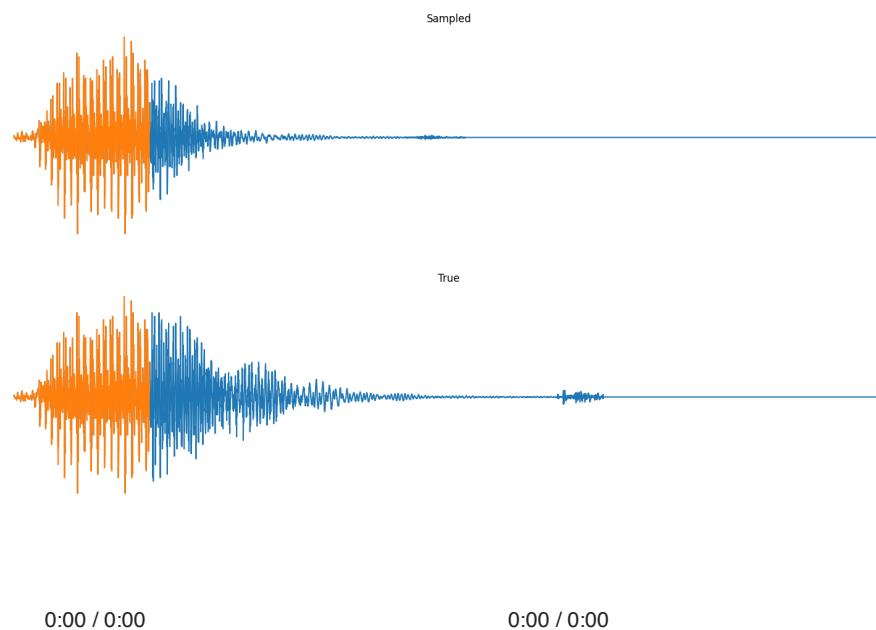
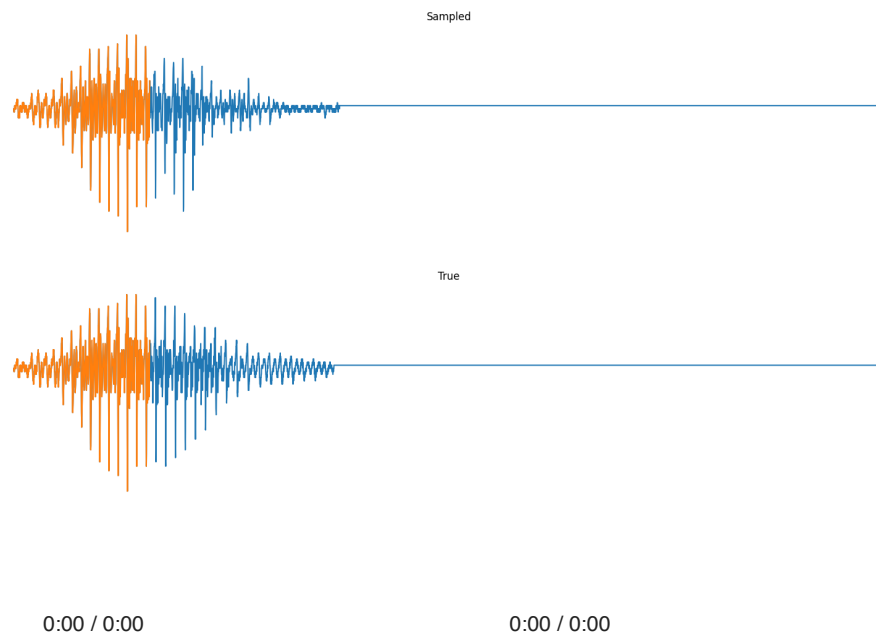
Finally we played with modeling sound waves directly. For these, we use the Free Spoken Digits Datasets an MNIST like dataset of various speakers reading off digits. We first trained a classification model and found that the approach was able to reach **97%** accuracy just from the raw soundwave. Next we trained a generation model to produce the sound wave directly. With $H = 512$ the model seems to pick up the data relatively well. This dataset only has around 3000 examples, but the model can produce reasonably good (cherry-picked) continuations. Note these sequences are 6400 steps long at an 8kHz sampling rate, discretized to 256 classes with Mu Law Encoding.











Our full code base contains more examples and infrastructure for training models for generations and classification.

Conclusion

Putting together this post inspired lots of thoughts about future work in this area. One obvious conclusion is that long-range models have all sorts of future applications from acoustic modeling to genomic sequences to trajectories (not to mention our shared area of NLP). Another is some surprise that linear models can be so effective here, while also opening up a range of efficient techniques. Finally from a practical level, the transformations in JAX make it really nice to implement complex models like this in a very concise way (~200 LoC), with similar efficiency and performance!

We end by thanking the authors Albert Gu and Karan Goel, who were super helpful in putting this together, and pointing you again to their paper and codebase. Thanks to Ankit Gupta, Ekin Akyürek, Qinsheng Zhang, Nathan Yan, and Junxiong Wang for contributions. We're also grateful for Conner Vercellino and Laurel Orr for providing helpful feedback on this post.

/ Cheers – Sasha & Sidd

Changelog

- v3
 - Major editing pass from Albert.
 - Fix bug in HiPPO calculation.
 - Added training of all S4 parameters.
 - Fix learning rate / initialization issues.
- v2
 - Added RNN decoding
 - Added Speech examples
- v1 – Original version