The Annotated S4 - S4 Applications

这一部分内容虽然涉及了很多工作，但实际模型非常简洁。实际上，我们只使用了以下四个函数：

1. `K_gen_DPLR`
   用于生成截断的生成函数，当矩阵 A 是 DPLR（S4部分）。
2. `conv_from_gen`
   将生成函数转换为滤波器。
3. `causal_convolution`
   执行卷积操作。
4. `discretize_DPLR`
   将状态空间模型（SSM）转换为离散形式，用于 RNN。

S4的CNN / RNN层

一个完整的 S4 层与简单的 SSM 层非常相似，唯一的区别在于 K 的计算。此外，与直接学习矩阵 C 不同，我们学习的是 $\tilde{C}$，从而避免计算矩阵 A 的幂次。

需要注意的是，在原始论文中，参数 Λ、P、Q 也是通过训练学习的。然而，为了简化，我们在此将这些参数保持固定。

```
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)
```

我们通过计算 HiPPO DPLR 初始化器来初始化模型

```
# 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)
```

采样与缓存

我们可以使用 RNN 实现从模型中采样。以下是采样代码的样子。请注意，我们保持一个运行缓存以记住 RNN 状态。

```
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]
```

为了将其整理好，我们首先为每个 S4 层预计算 RNN 的离散化版本。我们通过“prime”变量集合来完成这一点。

```
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"]
```

将这些汇总在一起，我们可以直接从模型中进行采样。

```
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)
```

实验：MNIST

现在我们已经构建了模型，可以在一些 MNIST 实验中测试它的效果。在这些实验中，我们将 MNIST 数据线性化，把每张图片视为一个像素序列。

MNIST 分类实验

我们首先进行了 MNIST 分类实验。虽然理论上这不是一个很难的问题，但将 MNIST 视为一个线性序列分类任务有些奇怪。然而，在实践中，使用隐藏层大小 ($H = 256$) 和四层的模型似乎可以轻松达到接近 99% 的准确率。

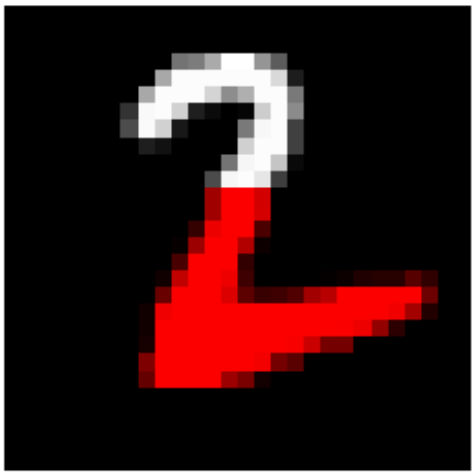
更有趣的任务：生成 MNIST 数字

一个更具视觉吸引力的任务是通过预测完整的像素序列来生成 MNIST 数字！在这个任务中，我们将一段像素序列输入模型，让它像语言建模一样预测下一个像素。经过一些调整后，我们使用隐藏层大小为 512 和 6 层（约 400 万参数）的模型，在该任务上达到了负对数似然（NLL）为 0.36 的表现。

任务指标：每维的比特数（BPD）

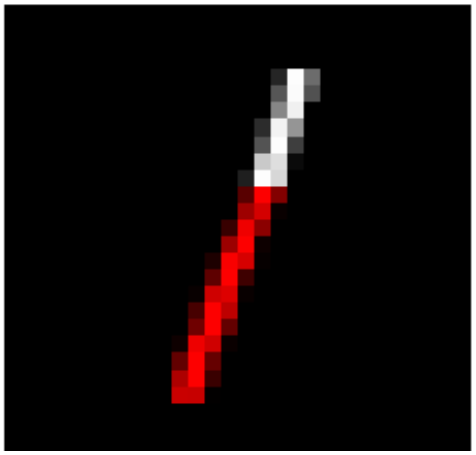
该任务通常使用 **每维的比特数（BPD）** 作为评价指标，即将 MNIST 的 NLL 换算为以 2 为底的对数。损失值 0.36 转换为约 0.52 BPD，这在 PapersWithCode 上是当前的 SOTA（最先进技术）。

Sampled

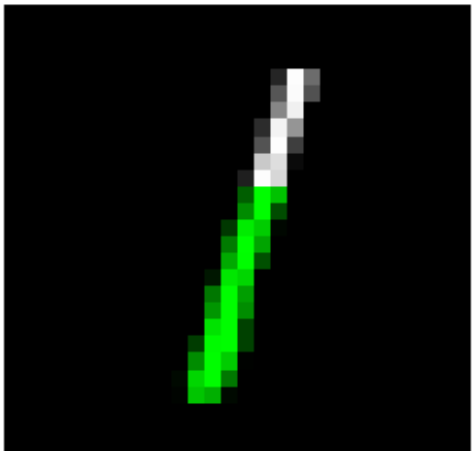


True

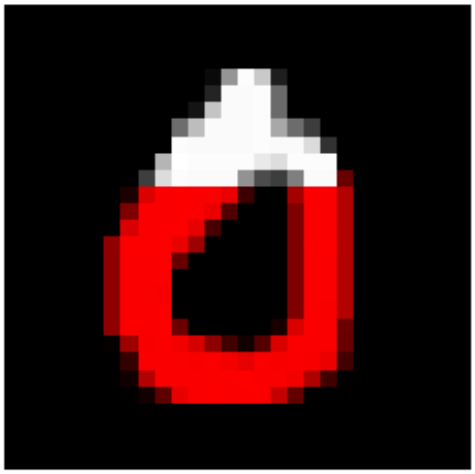
Sampled



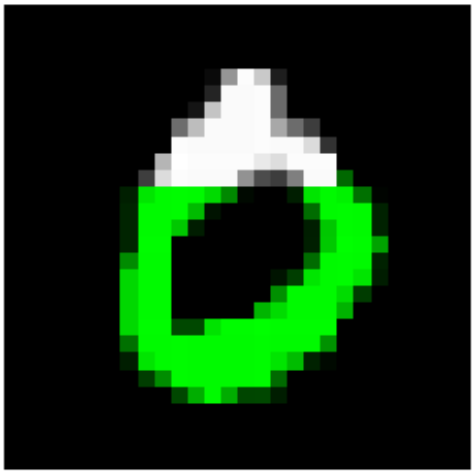
True



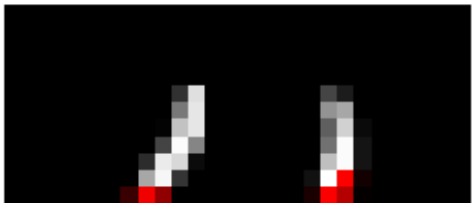
Sampled



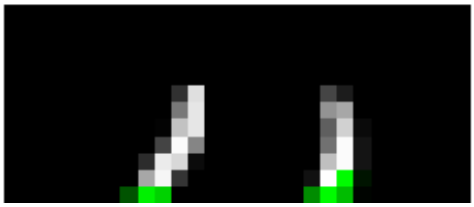
True

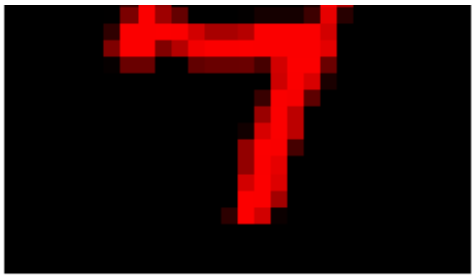


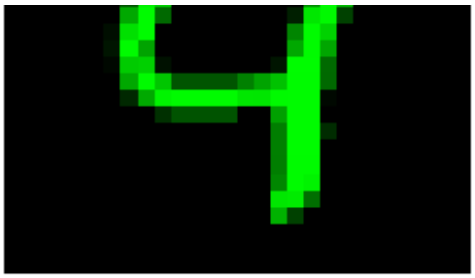
Sampled



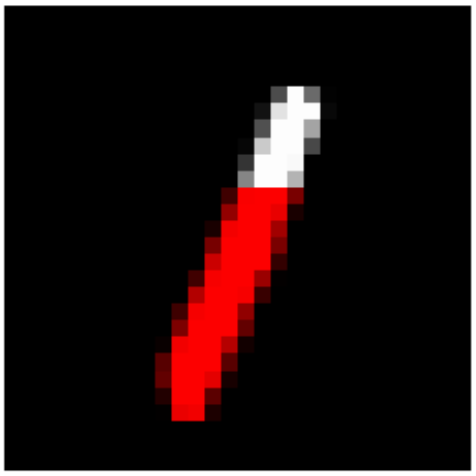
True



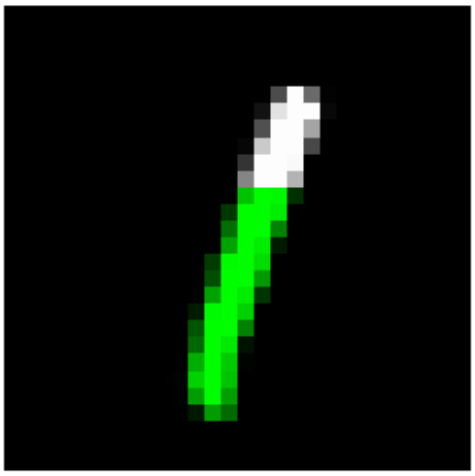


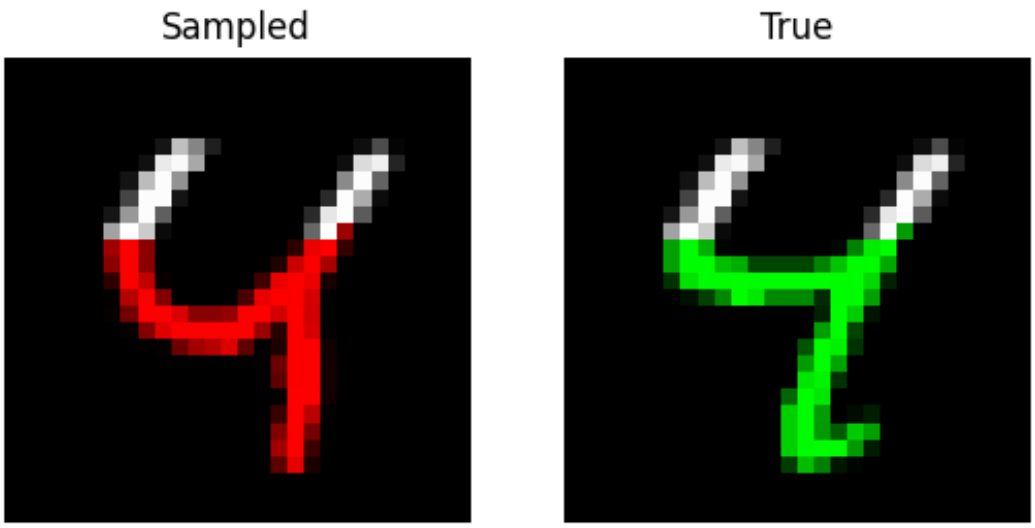


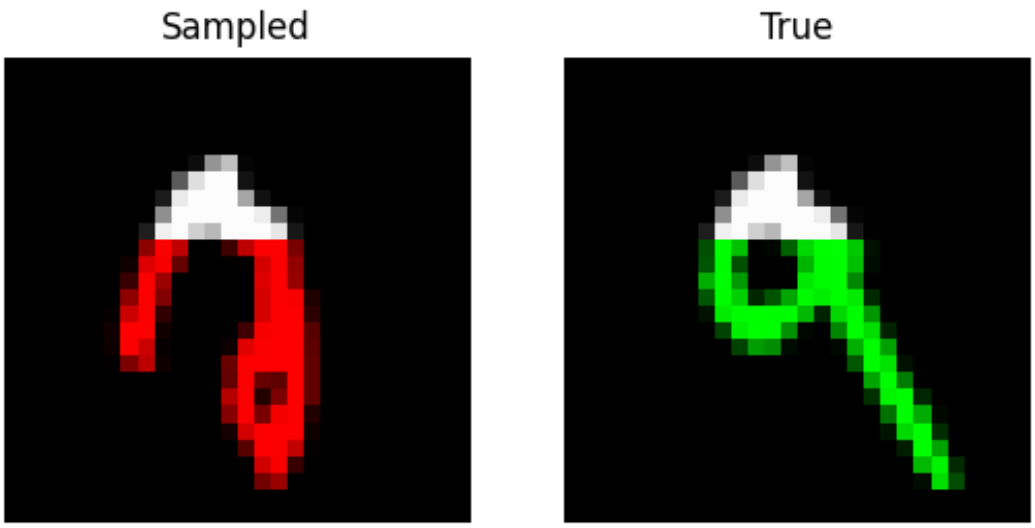
Sampled

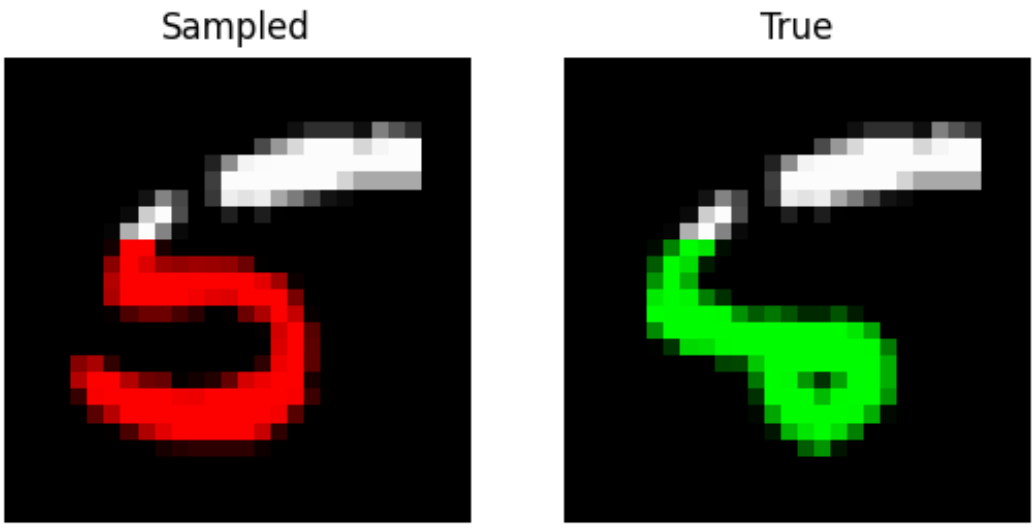


True




Sampled
True


Sampled
True




```
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
```

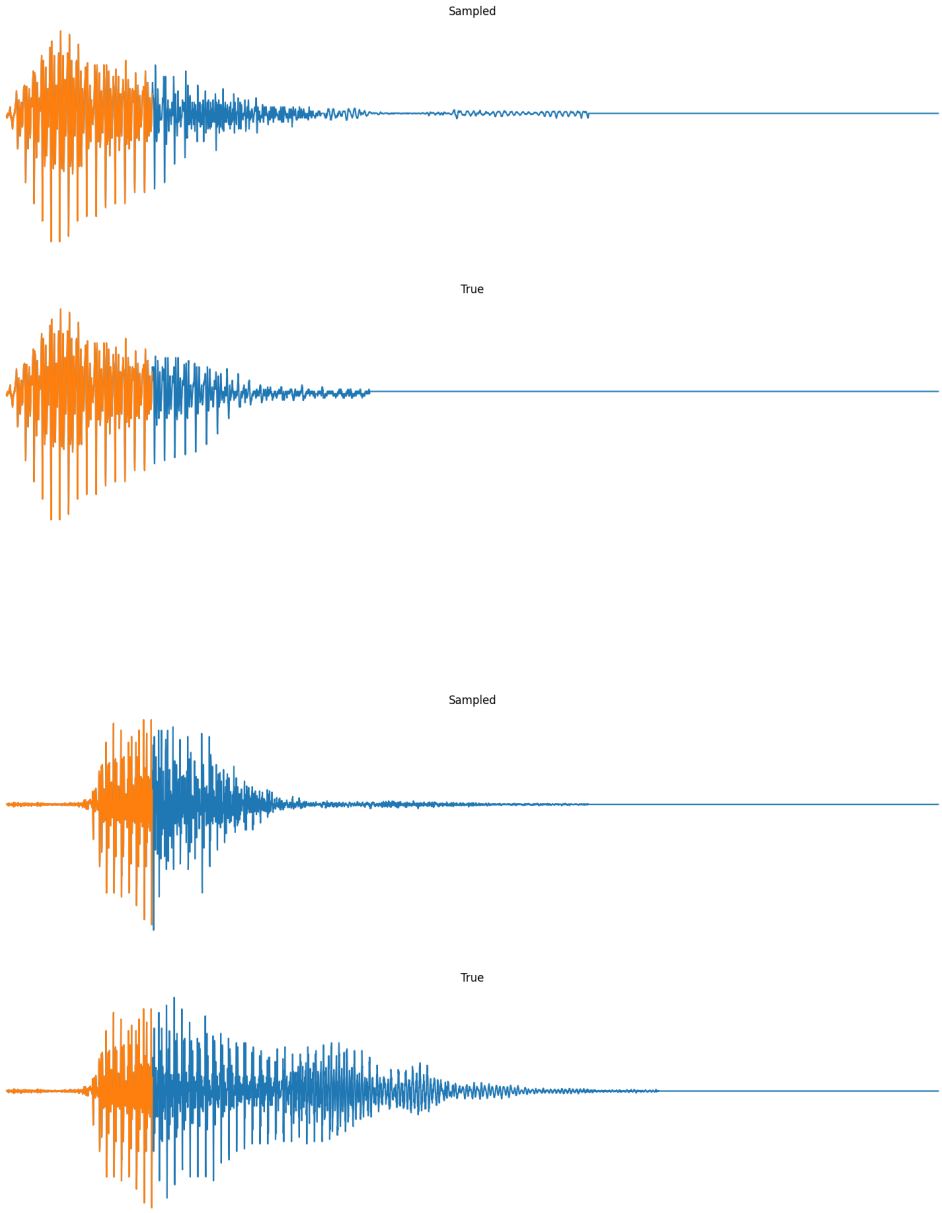
实验：QuickDraw

接下来，我们尝试训练一个模型来生成图画。为此，我们使用了QuickDraw 数据集该数据集包括一个降采样到 MNIST 大小的数据集版本，因此我们可以使用与上面大致相同的模型。然而，数据集大得多（500万张图像）且更复杂。我们只训练了1个时期。H=256一个4层模型。尽管如此，该方法能够生成相对连贯的补全。这些是给定500个像素的前缀样本。

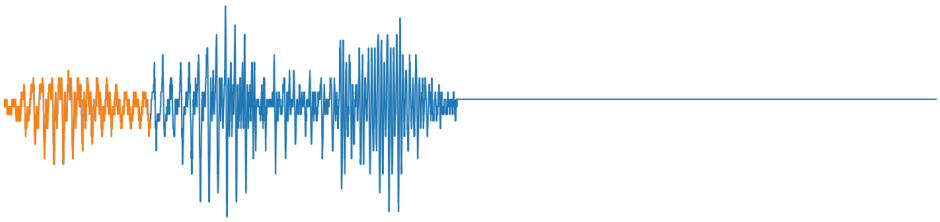
实验：口语数字

最后，我们直接玩声音波建模。为此，我们使用了免费语音数字数据集一个类似 MNIST 的数据集，各种讲述数字的说话者。我们首先训练了一个分类模型，并发现该方法能够达到97%97%准确性仅来自原始声波。接下来，我们训练了一个生成模型，以直接产生声波。使用H=512H =512该模型似乎能够相对良好地拾取数据。该数据集只有大约3000个示例，但该模型可以生成合理的（精选）延续。请注意，这些序列在8kHz

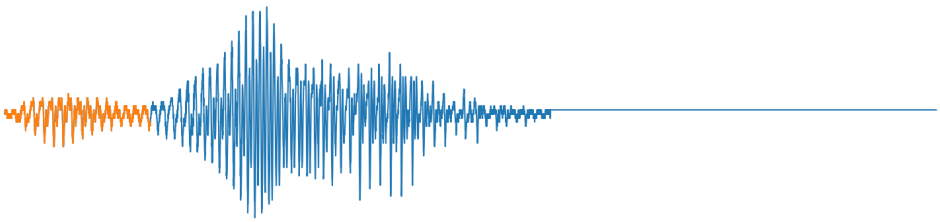
采样率下长度为6400步，离散化为256类。Mu 法编码.



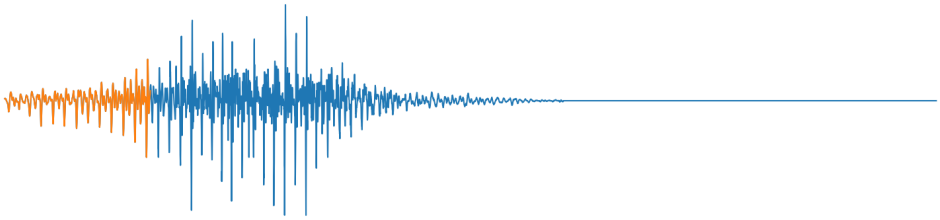
Sampled



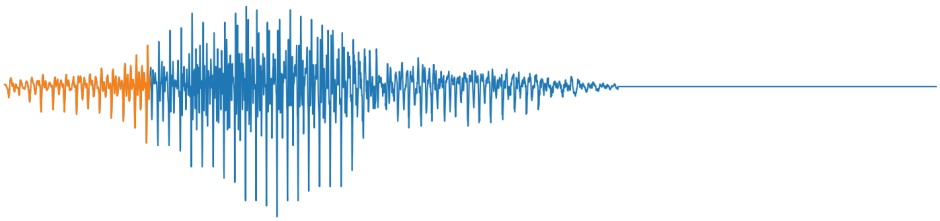
True



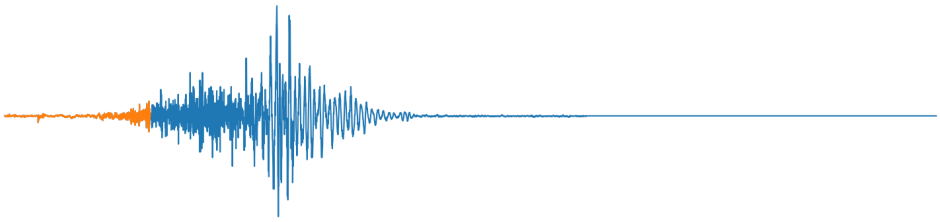
Sampled



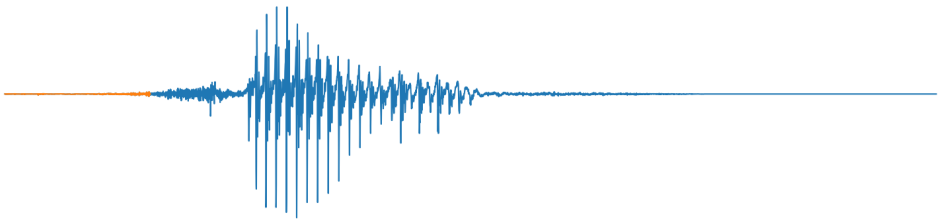
True



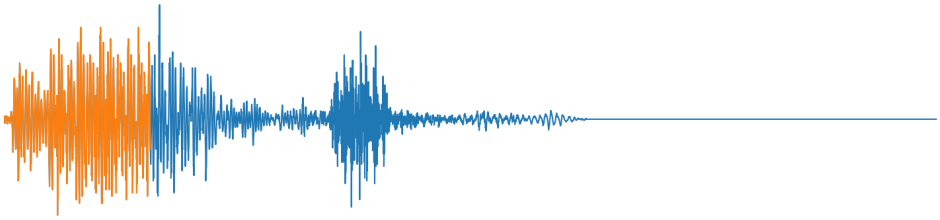
Sampled



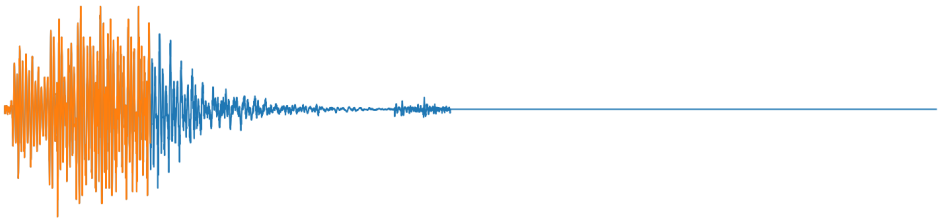
True



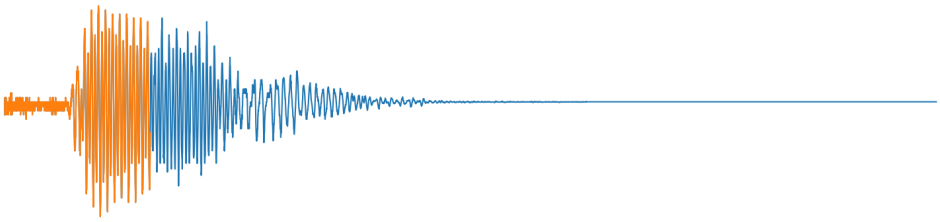
Sampled



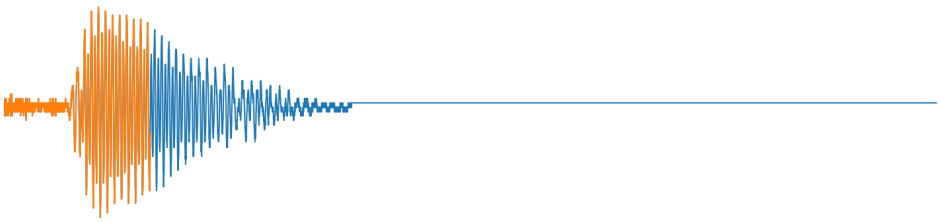
True



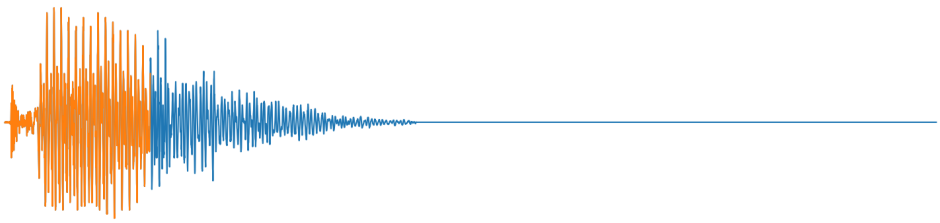
Sampled



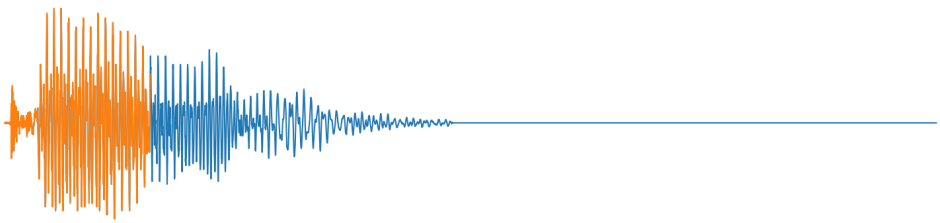
True

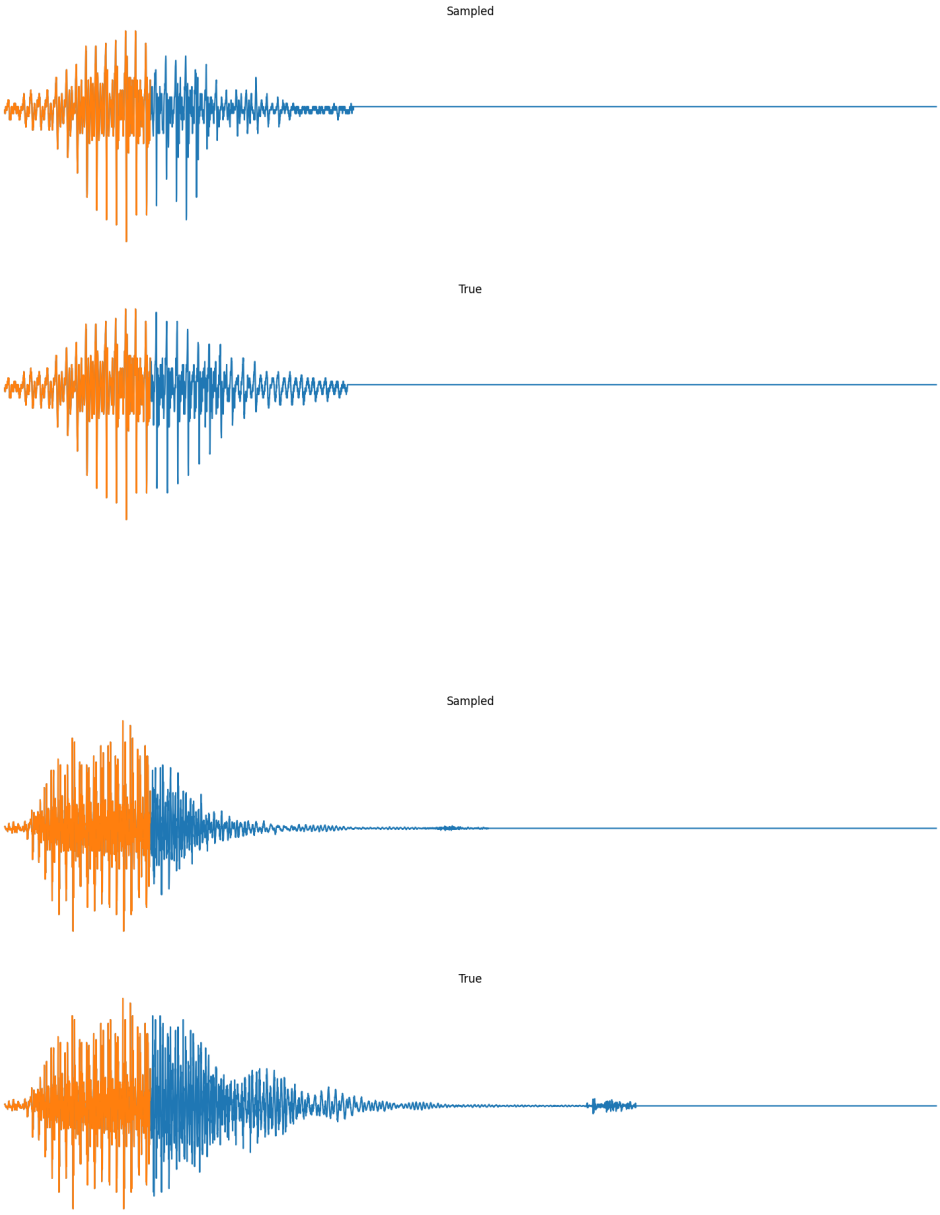


Sampled



True





完整代码库请查阅code base

结论

整理这篇文章激发了许多关于这一领域未来工作的思考。一个明显的结论是，长程模型在声学建模、基因组序列到轨迹（更不用说我们共同的NLP领域）等各类未来应用中有各种用途。另一个令人惊讶的是，线性模型在这里可以如此有效，同时也开启了一系列高效的技术。最后，从实践水平来看，JAX中的转换使实现这样的复杂模型变得非常简洁（约200行代码），并且具有相似的效率和性能！

我们最后感谢作者 Albert Gu 和 Karan Goel，他们在整理这篇文章方面给予了极大的帮助，并再次指向他们的论文和代码库。感谢 Ankit Gupta、Ekin Akyürek、Qinsheng Zhang、Nathan Yan 和 Junxiong Wang 的贡献。我们还感谢 Conner Vercellino 和 Laurel Orr 对此帖提供的有益反馈。

/ 致敬 – Sasha & Sidd