Time-Series Generation via Deep Latent State Space Models

Part I

Table of Contents

A Derivations

A.1 Computational Complexity of Vanilla Recurrent Representation

Assuming recurrence of the simplest form in Equation 2, fulling computing matrix multiplication $\bar{\boldsymbol{A}}\bar{h}_{t_k}$ requires $\mathcal{O}(N^2)$. Fully computing all hidden states sequentially requires $\mathcal{O}(N^2L)$. In space, saving each hidden state requires $\mathcal{O}(N)$ and in total requires $\mathcal{O}(NL)$.

B Proof

To prove this result in Proposition 4.1, we first prove the following proposition.

Proposition B.1. *(Expressivity.) Given any deep autoregressive S4 model $r : (x_{<t_n}, t_n) \mapsto y_{t_n}$ evaluated at time t_n given input sequence $x_{<t_n}$, there exists a choice of θ such that $\mu_{x,n}(x_{<t_n}, 0, \theta) = r(x_{<t_n}, t_n)$.*

Proof sketch. Consider SSM of the form in Equation 6 as a building block to our generative model with parameter θ. We can choose $\boldsymbol{E} = \boldsymbol{F} = 0$ for all layers, which exactly reduces it to the SSM of an S4 model. Keeping all other hyperparameters (*e.g.* non-linearities, number of stacking layers) the same, the final model is exactly the same as a deep autoregressive S4 model. □

Now we give a proof sketch to Proposition 4.1,

Proposition 4.1. *(LS4 subsumes S4.) Given any autoregressive model $r(x)$ with conditionals $r(x_n|x_{<n})$ parameterized via deep S4 models, there exists a choice of θ, λ, ϕ such that $p_{\theta,\lambda}(x) = r(x)$ and $p_{\theta,\lambda}(z|x) = q_\phi(z|x)$, i.e. the variational lower bound (ELBO) is tight.*

Proof sketch. From Proposition B.1 we know that we can choose θ so that $p_\theta(x|z) = p_\theta(x) = r(x)$ for all z, i.e., choose a decoder that ignores the latent variables z and uses the same autoregressive structure over the observed variables as $r(x)$. This implies the posterior $p_{\theta,\lambda}(z|x)$ is equal to the prior $p_\lambda(z)$. We can then choose λ and ϕ so that $p_\lambda(z) = \mathcal{N}(0, I)$ and $q_\phi(z|x) = \mathcal{N}(0, I)$ for all x. □

Proposition 4.2. *(Efficiency.) For a SSM with H heads, an observation sequence of length L and hidden dimension N can be calculated in $\mathcal{O}(H(L+N)\log(L+N))$ time and $\mathcal{O}(H(L+N))$ space.*

Proof. Recall *SISO* SSM of the form

$$\begin{aligned} \frac{d}{dt}\mathbf{h}_t &= \boldsymbol{A}\mathbf{h}_t + \boldsymbol{B}x_t \\ y_t &= \boldsymbol{C}\mathbf{h}_t + \boldsymbol{D}x_t \end{aligned} \tag{17}$$

The calculation of $(y_{t_0}, \ldots, y_{t_L})$ involves materializing the convolution filter, which can be calculated in $\mathcal{O}((L+N)\log(L+N))$ time and $\mathcal{O}(L+N)$ space for diagonal-plus-low-rank matrices (Gu et al., 2021). Since the convolution is constant time in frequency domain, another computation cost comes from Fast Fourier Transform (FFT) and its inverse, which is $\mathcal{O}(L\log L)$ in time. The computation scales linearly with heads, Thus, a multi-input-multi-output (MIMO) SSM with H heads can be processed in $\mathcal{O}(H(L+N)\log(L+N))$ time and $\mathcal{O}(H(L+N))$ space. □

C Architecture

We parametrize our models using a similar architecture as in Goel et al. (2022), but there is no pooling operation because for general time-series the time length is hardly divisible by a reasonable factor. Before we present the full structure, we present how a multi-channel inputs are parametrized (in the case of LS4 prior layer (10):

```
def LS4_prior_layer_multi(z, psi):
    # z: (B, L, C)
    for c in range(C):
        z[:,:,c] = LS4_prior_layer(z[:,:,c], *psi.LS4_params)
    z = linear(z) # (B, L, C)  channel-wise mixing
    return z
```

The for loop is presented for demonstration purposes. In practice, the channels can be processed in parallel.

C.1 Prior Model

We specify the parametrization of $\mu_{z,n}$ and $\sigma_{z,n}$ in pseudo-code as the outputs of the following functions

```
def prior_model(z, lambda):
    # z: (B, L, z_dim) this is for time [t_0, t_{n-1}]
    z = linear(z) # (B,L,H) encoding to H

    outputs = []
    outputs.append(z)
    for i in range(lambda.num_layers1):
        z = linear(z) #  (B, L, H) -> (B, L, 2H)
        outputs.append(z)

    for i in range(lambda.num_layers2):
        z = LS4_prior_block_multi(z, *lambda.LS4_params)
                # (B, L, H) -> (B, L, H) multi-channel SSMs
        z = ResBlock(z) # this is a general 1 layer residual block
    z = z + outputs.pop()
```

```
    for i in range(lambda.num_layers1):
        z = z + outputs.pop()
        outputs.append(z)
        z = linear(z)  #  (B, L, 2H) -> (B, L, H)
        for i in range(lambda.num_layers2):
            z = LS4_prior_block_multi(z, *lambda.LS4_params)
                # (B, L, H) -> (B, L, H) multi-channel SSMs
            z = ResBlock(z) # this is a general 1 layer residual block

        z = z + outputs.pop()
    z = layernorm(z)
    z = linear(z) # (B,L,z_dim)
    mu_z = LS4_prior_block_multi(z, *lambda.LS4_params) # (B,L,z_dim)
    sigma_z = LS4_prior_block_multi(z, *lambda.LS4_params)
                                                        # (B,L,z_dim)
    return mu_z, sigma_z
```

C.2 Generative Model

```
def prior_model(x, z, theta):
    # z: (B, L, z_dim) this is for time [t_0, t_{n-1}]
    z = linear(z) # (B,L,H) encoding to H
    x = linear(x) # (B,L,H) encoding to H

    outputs_x, outputs_z = [], []
    outputs_z.append(z)
    outputs_x.append(x)
    for i in range(lambda.num_layers1):
        z = linear(z) #  (B, L, H) -> (B, L, 2H)
        x = linear(x) #  (B, L, H) -> (B, L, 2H)
        outputs_z.append(z)
        outputs_x.append(x)

    for i in range(lambda.num_layers2):
        z, x = LS4_gen_block_multi(z, *lambda.LS4_params)
                # (B, L, H) -> (B, L, H) multi-channel SSMs
        zx = ResBlock(concat(z, x))
                # this is a general 1 layer residual block
        z, x = split(z, x)
    z = z + outputs_z.pop()
    x = x + outputs_x.pop()

    for i in range(lambda.num_layers1):
        z = z + outputs_z.pop()
        x = x + outputs_x.pop()
        outputs_z.append(z)
        outputs_x.append(z)
        z = linear(z)  #  (B, L, 2H) -> (B, L, H)
        x = linear(x)  #  (B, L, 2H) -> (B, L, H)
        for i in range(lambda.num_layers2):
            x, z = LS4_gen_block_multi(x, z, *lambda.LS4_params)
                        # (B, L, H) -> (B, L, H) multi-channel SSMs
            zx = ResBlock(concat(z, x))
                # this is a general 1 layer residual block
            z, x = split(z, x)

        z = z + outputs_z.pop()
        x = x + outputs_x.pop()
    x = layernorm(x)
    z = layernorm(z)
```

```
    x = linear(concat(x, z)) # (B,L,2H) -> (B,L,x_dim)
    return mu_x
```

In practice, we find that only using z input for the entire generative model produces better generation better than including x. We hypothesize that x presents too strong of a signal for the model to reconstruct, and so the model learns to ignore signals from z in that case.

C.3 Inference Model

```
def inference_model(x, phi):
    # x: (B, L, x_dim) this is for time [t_0, t_{n-1}]
    x = linear(x) # (B,L,H) encoding to H

    outputs = []
    outputs.append(x)
    for i in range(phi.num_layers1):
        x = linear(x) #  (B, L, H) -> (B, L, 2H)
        outputs.append(x)

    for i in range(phi.num_layers2):
        x = LS4_inf_block_multi(x, *phi.LS4_params)
                # (B, L, H) -> (B, L, H) multi-channel SSMs
        z = ResBlock(z) # this is a general 1 layer residual block
    x = x + outputs.pop()

    for i in range(phi.num_layers1):
        x = x + outputs.pop()
        outputs.append(x)
        x = linear(x)  #  (B, L, 2H) -> (B, L, H)
        for i in range(phi.num_layers2):
            x = LS4_inf_block_multi(x, *phi.LS4_params)
                    # (B, L, H) -> (B, L, H) multi-channel SSMs
            z = ResBlock(z) # this is a general 1 layer residual block

        x = x + outputs.pop()
    x = layernorm(x)
    x = linear(x) # (B,L,x_dim)
    mu_z = LS4_inf_block_multi(x, *phi.LS4_params) # (B,L,x_dim)
    sigma_z = LS4_inf_block_multi(x, *phi.LS4_params)
                                                # (B,L,x_dim)
    return mu_z, sigma_z
```

D Experiments

For all experiments we use AdamW optimizer with learning rate 0.001. We use batch size 64 and train for 7000 epochs for FRED-MD, NN5 Daily, and Solar Weekly, 1000 epochs for Temperature Rain, and 500 epochs for Physionet and USHCN. The datasets are split into 80% training data and 20% testing data.

D.1 Monash Forecasting Repository

Data. For all selected MONASH data, FRED-MD, NN5 Daily, and Solar Weekly are normalized per sequence such that each trajectory is centered at its own mean and normally distributed. We make this choice from the observation that for some datasets such as NN5 Daily the min and max can vary significantly across different data points such that normalizing sequences with dataset-wise statistics makes it difficult to learn the temporal dynamics, which would be on a widely different range. For Temperature Rain we squash each sequence into $[0, 1]$. This is due to the fact that the dataset is

always positive and lands mostly around x-axis with sharp spikes in between. For the former 3 datasets, we do not use output activation while for the last, we use sigmoid as our activation.

Hyperparameters. For all MONASH experiments, we use AdamW optimizer with learning rate 0.001 and no weight decay. For each of prior/generative/inference model, we use 4 stacks for each for loop in the pseudocode. For each LS4 block, we use 64 as the dimension of $\mathbf{h}_t$ and 64 SSM channels in parallel, same as used in S4 and SaShiMi. Each residual block consists of 2 linear layers with skip connection at the output level where the first linear layer has 2 times output size as the input size and the second layer squeezes it back to the input size of the residual block. We generally find 5-dimensional latent space gives better performance than 1, and so uses this setting throughout. We also employ EMA for model weights and use 0.999 as the lambda value, but we do not find this choice crucial. We also use 0.1 as the standard deviation for the observation as this gives better ELBO than other choices we experimented with such as $1, 0.5, 0.01$. For baselines, we reuse the code from official repo and follow their suggestions for training. To keep representation power similar, we use the same size for the latent space (for latent variable models) and the same output standard deviation for ELBO evaluation.

Evaluation. For generation evaluation. The classification model and the prediction model uses a linear encoder and linear decoder with a single S4 layer in between. The S4 layer uses 16 hidden state dimensions. For classification model, encoder maps data dimension to 16 hidden state dimension, and averages over the sequence output from S4 layer before feeding into decoder that outputs logit for binary classification. We use cross entropy loss. For prediction model, we use the same linear encoder and a decoder that maps 16 hidden dimension to data dimension. We predict $k = 10$ steps into the future. The evaluation models are trained using AdamW with 0.01 learning rate for 100 epochs with batch size 128. We generate samples equal to the number of testing data, which together are used to train the two models.

Additional discussion. We also briefly discuss the surprising result that SaShiMi does not perform as well on general time-series generation. We speculate that not using a quantization scheme to define discrete output conditionals, as standard in autoregressive models for e.g., audio and images, is the cause behind this drop in performance. LS4 does not requires quantization and sets best performance with a simple Gaussian conditional on the data space.

Additional comparisons. We additionally compare with two more relevant baselines (Fabius & Van Amersfoort, 2014) and (Li et al., 2020) present result in Table 3. We note that Latent SDE has an abnormally high classification score for Temperature Rain data, and demonstrate that this is when the classification score is not reliable. Upon visually examining generated results for Latent SDE (Figure 4) compared to ground-truths (Figure 7), one can observe that the variation is extremely noisy around the x-axis and that the selected classifier is not powerful enough to capture the distinction from real data due to the considerable noise that exists in both generated and real data, resulting in high classification score. Marginal and predictive scores are much worse in comparison and are more indicative of generation quality.

Data	Metric	VRNN	Latent SDE	**LS4 (Ours)**
FRED-MD	Marginal ↓	0.165	0.122	**0.0221**
	Class. ↑	0.000970	**0.687**	0.544
	Prediction ↓	0.371	1.62	**0.0373**
NN5 Daily	Marginal ↓	0.151	0.125	**0.00671**
	Class. ↑	0.00176	0.601	**0.636**
	Prediction ↓	1.22	0.957	**0.241**
Temp Rain	Marginal ↓	1.20	0.999	**0.0834**
	Class. ↑	0.479	**14.534**	0.976
	Prediction ↓	0.864	1.798	**0.521**
Solar Weekly	Marginal ↓	0.297	0.234	**0.0459**
	Class. ↑	0.00164	0.764	**0.683**
	Prediction ↓	0.964	1.01	**0.141**

Table 3: Additional generation results on FRED-MD, NN5 Daily, Temperature Rain, and Solar Weekly.

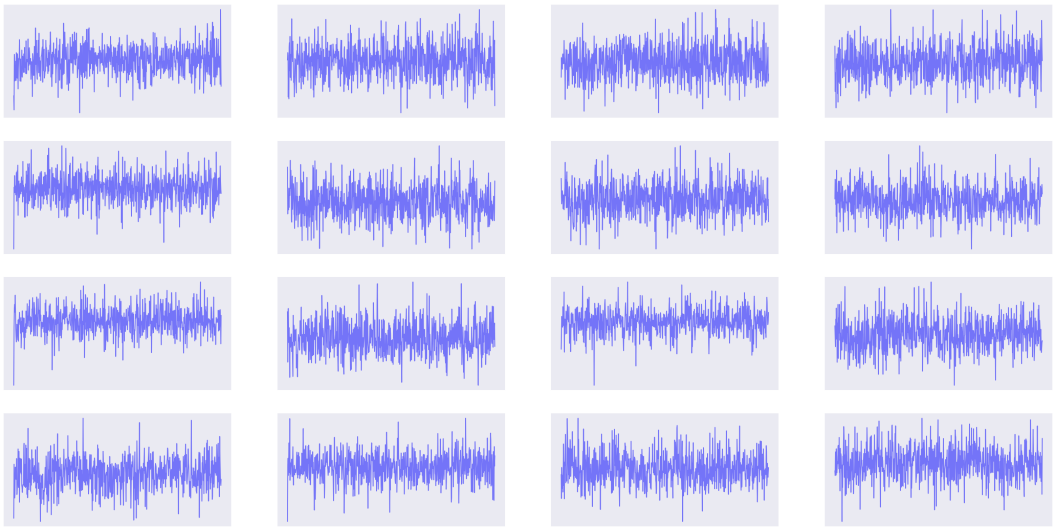

Figure 4: Latent SDE generation on Temperature Rain.

Task	Data	RNN-VAE	Latent ODE	**LS4 (Ours)**	**LS4$^{\text{IWAE}}$ (Ours)**
Interp.	Physionet	-412.8	-410.3	-669.0	**-684.3**
	USHCN	-244.9	-251.0	-312.2	**-315.6**
Extrap.	Physionet	-220.2	-168.5	-250.2	**-288.7**
	USHCN	-113.3	-110.3	-194.4	**-211.8**

Table 4: ELBO comparisons with VAE-based models.

D.2 PHYSIONET & USHCN

We follow the code provided by Rubanova et al. (2019) to process Physionet and follow the code provided by De Brouwer et al. (2019) for USHCN. For Physionet we do not use any activation to constrain the output space, and for USHCN, we use sigmoid activation for output.

We present the variational lowerbound results in Table 4.

Hyperparameters. In general we keep the hyperparameter choices the same as in MONASH, and we describe a few differences for these 2 datasets. For USHCN, we use 10 as the dimension for latent space, same as in Rubanova et al. (2019) and we use Sigmoid as the output activation with output standard deviation 0.01. For Physionet, we use no output activation and 0.05 standard deviation, and use 20-dimensional latent space, same as in baselines.

Additional Metric We also present CRPS (continuous ranked probability score) as a more appropriate metric for time-series forecasting. With the same baselines, we show CRPS result in Table 5.

Task	Data	RNN	RNN-VAE	ODE-RNN	GRU-D	Latent ODE	**LS4 (Ours)**
Interp.	Physionet	2.09	5.59	2.40	2.71	6.16	**1.25**
	USHCN	3.33	4.68	3.18	4.69	4.68	**0.438**
Extrap.	Physionet	3.30	2.17	**2.16**	13.9	2.43	2.36
	USHCN	72.1	5.01	5.09	5.01	5.04	**2.76**

Table 5: Interpolation and extrapolation CRPS ($\times 10^{-2}$) scores. Lower scores are better.

D.3 RUNTIME

We test all models on a single RTX A5000 GPU. To set up the dataset, we need to fully populate the GPU for each dataset during training for our benchmarking. For sequence lengths $\{80, 320, 1280, 5120, 20480\}$, we build dataset of length $\{102400, 25600, 6400, 1600, 400\}$ each with batch size $\{1024, 256, 64, 16, 4\}$ so that for each dataset the models are trained with 100 iterations.

E GENERATION RESULTS

We present ground-truths and generations on the two hardest selected datasets, NN5 Daily and Temperature Rain because these are the hardest to model.

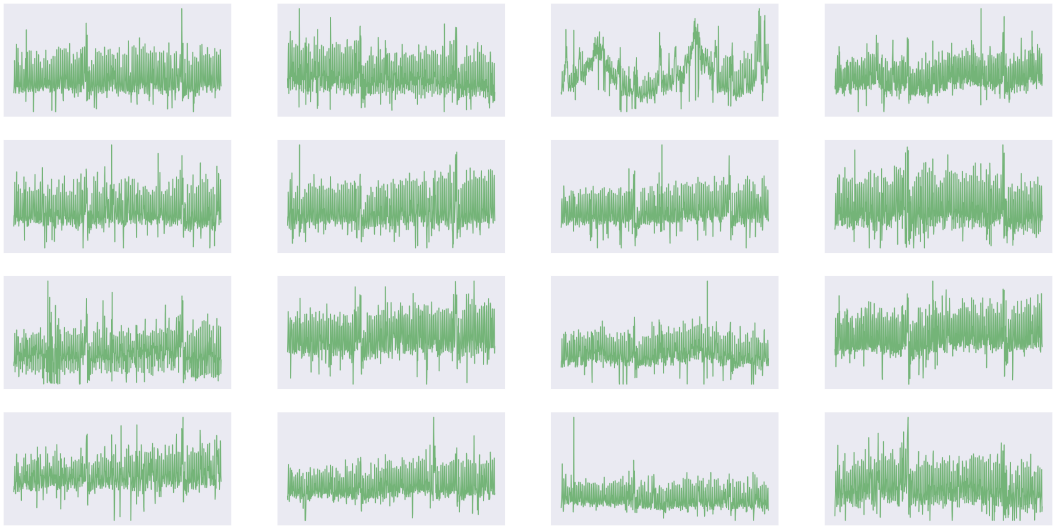

Figure 5: Normalized NN5 Daily data.

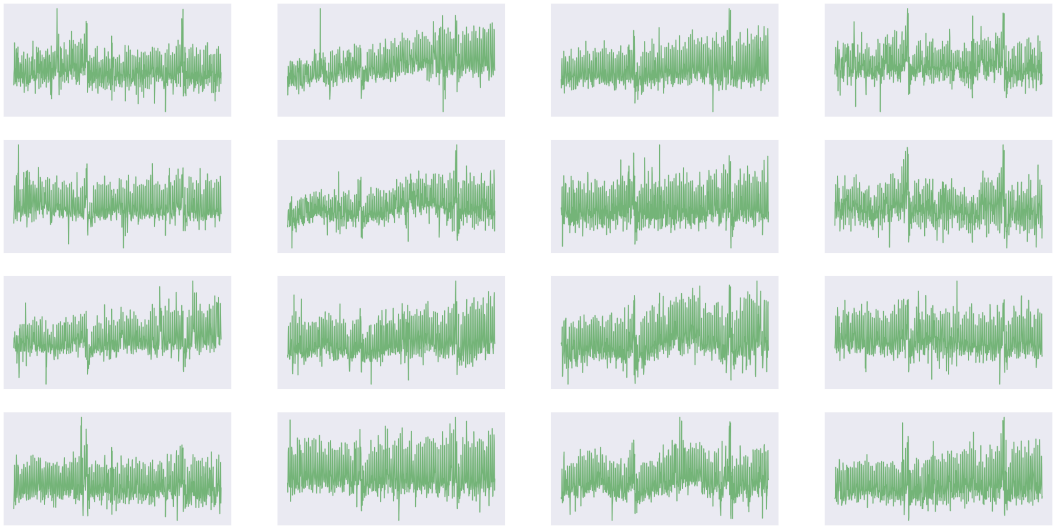

Figure 6: NN5 Daily generation.

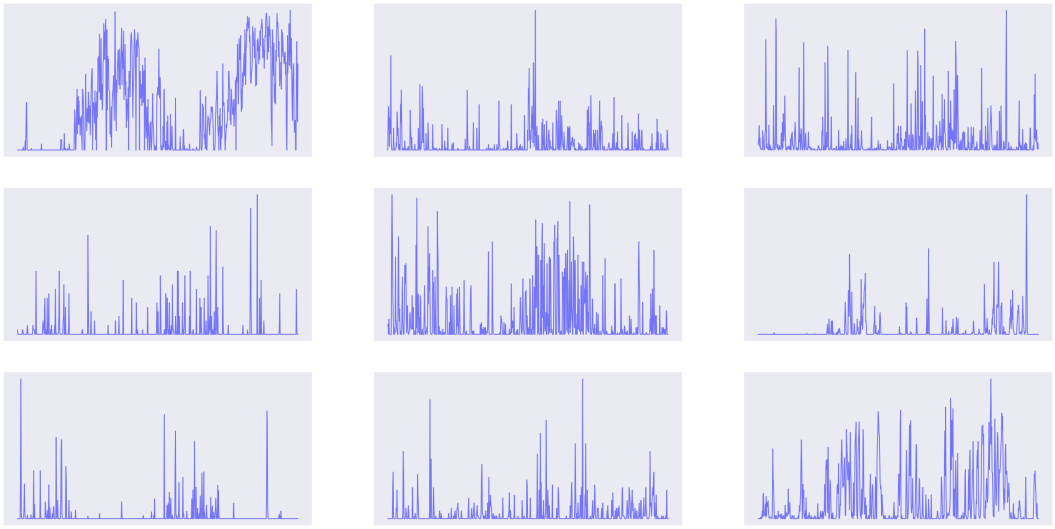

Figure 7: Normalized Temperature Rain data.

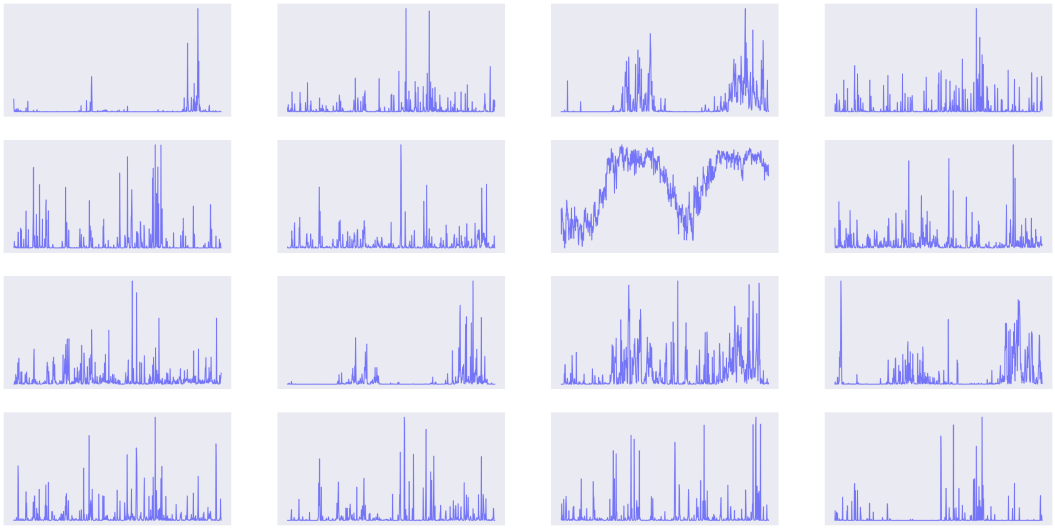

Figure 8: Temperature Rain generation.