Chapter 9

State-Space Models and Linear Transformers

9.1 State-Space Models

9.1.1 Continuous form

A linear state space layer (Gu, Johnson, et al., 2021) takes as input a function $x : \mathbb{R} \to \mathbb{R}$ and outputs another function $y : \mathbb{R} \to \mathbb{R}$, defined by the following differential equations:

$$\begin{aligned} h'(t) &= \mathbf{A}h(t) + \mathbf{B}x(t) \\ y(t) &= \mathbf{C}h(t) \end{aligned} \tag{9.1}$$

where $h : \mathbb{R} \to \mathbb{R}^d$ and h' is its derivative. The parameters of the model are:

$$\begin{aligned} \mathbf{A} &\in \mathbb{R}^{m \times m} \\ \mathbf{B} &\in \mathbb{R}^{m \times 1} \\ \mathbf{C} &\in \mathbb{R}^{1 \times m}. \end{aligned}$$

Maybe the model is easier to think about in integral form:

$$\begin{aligned} h(t) &= \int_0^t \mathbf{A}h(\tau) + \mathbf{B}x(\tau)\ \mathrm{d}\tau \\ y(t) &= \mathbf{C}h(t). \end{aligned}$$

So $\mathbf{B}$ controls how h depends on the past input, $\mathbf{A}$ controls how h depends on its own past, and $\mathbf{C}$ controls how the output depends on h.

9.1.2 Recurrent form

Next, we want to discretize (9.1), which means that we choose a step size $\Delta > 0$ and construct a network that takes as input a sequence $x_0 = x(0), x_1 = x(\Delta), x_2 = x(2\Delta), \ldots$ and outputs a sequence $y_0 \approx y(0), y_1 \approx y(\Delta), y_2 \approx y(2\Delta), \ldots$.

There are various ways of doing this. A very simple one is the forward Euler method, which approximates x and h with rectangles. Integrate the first equation above from t to $t + \Delta$:

$$h(t+\Delta) - h(t) = \int_t^{t+\Delta} \mathbf{A}h(t) + \mathbf{B}x(t)\,\mathrm{d}t \tag{9.2}$$

$$\approx \Delta\mathbf{A}h(t) + \Delta\mathbf{B}x(t) \tag{9.3}$$

$$h(t+\Delta) \approx (\mathbf{I} + \Delta\mathbf{A})h(t) + \Delta\mathbf{B}x(t) \tag{9.4}$$

or equivalently

$$h(t+\Delta) \approx \bar{\mathbf{A}}h(t) + \bar{\mathbf{B}}x(t) \tag{9.5}$$

$$\bar{\mathbf{A}} = \mathbf{I} + \Delta\mathbf{A} \tag{9.6}$$

$$\bar{\mathbf{B}} = \Delta\mathbf{B}. \tag{9.7}$$

Then we can write this as a mapping from a discrete sequence $\{x_i\}$ to a sequence $\{y_i\}$:

$$\begin{aligned} h_i &= \bar{\mathbf{A}}h_{i-1} + \bar{\mathbf{B}}x_i \\ y_i &= \mathbf{C}h_i. \end{aligned} \tag{9.8}$$

The parameters of the model are still $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, and now Δ as well.

Real implementations use more complicated methods, like the bilinear transform a.k.a. Tustin's method (Gu, Johnson, et al., 2021), which uses trapezoids, or the zero-order hold (Gu and Dao, 2023), which approximates x (but not h) using rectangles. These methods keep Eq. (9.5) but use different $\bar{\mathbf{A}}$ and $\bar{\mathbf{B}}$.

Real implementations run d of these in parallel to give a mapping on length-n sequences of size-d vectors. But for our purposes we can continue to assume $d = 1$.

9.1.3 Convolutional form

Since Eq. (9.8) are both linear, we can write each y_i as a (really big) linear function of $x_0, \ldots, x_i$ by substituting away all the h_i's:

$$y_i = \sum_{j=0}^{i} \mathbf{C}\bar{\mathbf{A}}^j\bar{\mathbf{B}}x_{i-j}. \tag{9.9}$$

If you are familiar with convolutional neural networks, you could think of this as a (really big) convolution:

$$\bar{k}_j = \mathbf{C}\bar{\mathbf{A}}^j\bar{\mathbf{B}} \qquad j = 0, \ldots, i \tag{9.10}$$

$$\mathbf{y} = \bar{\mathbf{k}} * \mathbf{x}. \tag{9.11}$$

In some versions of SSMs, this enables efficient parallel implementations. It also plays a key role in the theoretical results below.

9.1.4 Variations

The matrix $\mathbf{A}$ is often assumed to have a special structure, usually diagonal (Gupta et al., 2022). These models are called S4 (structured state space sequence) models.

In Mamba (Gu and Dao, 2023) – also known as S6, which stands for selective scan S4) – $\mathbf{A}$ is a learned diagonal matrix, but $\mathbf{B}$, $\mathbf{C}$, and Δ become input-dependent:

$$\begin{aligned}
\mathbf{B}_i &= lin_{\mathrm{B}}(x_i) \\
\mathbf{C}_i &= lin_{\mathrm{C}}(x_i) \\
\Delta_i &= \mathrm{softplus}(lin_{\Delta}(x_i)) \\
&= \log(1 + \exp(lin_{\Delta}(x_i))).
\end{aligned}$$

where lin_{B}, lin_{C}, lin_{Δ} are linear layers with $lin_{\mathrm{B}}.d' = lin_{\mathrm{C}}.d' = m$ and $lin_{\Delta}.d' = 1$.

9.1.5 Expressivity

Theorem 9.1 (Merrill et al., 2024)**.** *The following SSMs, using $O(\log n)$ bits of precision, are in* DLOGTIME*-uniform* TC^0*:*

1. *Input-independent* $\mathbf{A}$*,* $\mathbf{B}$*,* $\mathbf{C}$*,* Δ*.*
2. *Input-dependent diagonal* $\mathbf{A}$*, and input-dependent* $\mathbf{B}$*,* $\mathbf{C}$*,* Δ*.*

Merrill et al. (2024) only claimed L-uniformity, but their proof seems to work for DLOGTIME-uniformity as well.

Proof. If $\mathbf{A}$ is input-independent, computing the convolutional form (Eq. (9.9)) just requires matrix multiplication, powering, and iterated addition. These are all are in DLOGTIME-uniform TC^0 on $O(\mathrm{poly}(n))$-bit integers (Hesse et al., 2002; Allender et al., 2014), so we just have to show this on $O(\log n)$-bit floating-point numbers as well. Elsewhere (Theorem 7.11) we have shown this for addition, multiplication, and iterated addition, which gets us matrix multiplication and iterated addition. To compute $\bar{\mathbf{A}}^k$, let $e_{\min}$ be the minimum possible exponent. Then

$2^{-e_{\min}}\bar{\mathbf{A}}$ has integer entries with $O(\text{poly}(n))$ bits, so we can compute $(2^{-e_{\min}}\bar{\mathbf{A}})^k$ in DLOGTIME-uniform TC^0. Then we can multiply by $2^{-e_{\min}k}$ to get $\bar{\mathbf{A}}^k$.

If $\mathbf{A}_i$ is input-dependent but diagonal, then

$$\prod_{i=1}^{n} \mathbf{A}_i = \begin{bmatrix} \prod_{i=1}^{n} \mathbf{A}_i[1,1] & 0 & \cdots & 0 \\ 0 & \prod_{i=1}^{n} \mathbf{A}_i[2,2] & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \prod_{i=1}^{n} \mathbf{A}_i[m,m] \end{bmatrix}$$

and iterated product is in DLOGTIME-uniform TC^0 on $O(\text{poly}(n))$-bit integers (Hesse et al., 2002). On $O(\log n)$-bit floating-point numbers, we simply perform an iterated product on their mantissas and iterated addition on their exponents. □

9.2 Linear Transformers

9.2.1 Model

Recall that the output of an attention layer is

$$\mathbf{c}_i = \frac{\sum_j \exp(\mathbf{q}_i \cdot \mathbf{k}_j)\mathbf{v}_j}{\sum_j \exp(\mathbf{q}_i \cdot \mathbf{k}_j)}. \tag{9.12}$$

At each time step i, we have to recompute the sums over j. Not only is this time consuming, leading to $O(n^2)$ time complexity, it also means that we have to store $\mathbf{k}_j$ and $\mathbf{v}_j$ for all previous j. (This is known as the *KV cache*).

Katharopoulos et al. (2020) proposed the following modification. Let ϕ be any function $\phi : \mathbb{R}^d \to \mathbb{R}^{d'}$, called the *feature map* (a term that goes back to the days of support vector machines and kernel methods). In a linear transformer, the output is

$$\mathbf{c}_i = \frac{\sum_j (\phi(\mathbf{q}_i) \cdot \phi(\mathbf{k}_j))\mathbf{v}_j}{\sum_j \phi(\mathbf{q}_i) \cdot \phi(\mathbf{k}_j)} \tag{9.13}$$

$$= \frac{\phi(\mathbf{q}_i) \cdot \sum_j \phi(\mathbf{k}_j)\mathbf{v}_j}{\phi(\mathbf{q}_i) \cdot \sum_j \phi(\mathbf{k}_j)}. \tag{9.14}$$

Now the sums over j do not depend on i. So they don't need to be recomputed for each i; instead, it can be computed incrementally. That is, in place of the KV

cache, we accumulate two partial sums:

$$\mathbf{a}_i = \mathbf{a}_{i-1} + \phi(\mathbf{k}_i)\mathbf{v}_i \tag{9.15}$$

$$\mathbf{b}_i = \mathbf{b}_{i-1} + \phi(\mathbf{k}_i) \tag{9.16}$$

$$\mathbf{c}_i = \frac{\phi(\mathbf{q}_i) \cdot \mathbf{a}_i}{\phi(\mathbf{q}_i) \cdot \mathbf{b}_i}. \tag{9.17}$$

The feature map ϕ can be anything, but in particular, ϕ can be chosen so that $\phi(\mathbf{q}_i) \cdot \phi(\mathbf{k}_j)$ is a Taylor approximation of exp (Arora et al., 2024):

$$\phi(\mathbf{q}_i) \cdot \phi(\mathbf{k}_j) = 1 + \mathbf{q}_i \cdot \mathbf{k}_j + \frac{1}{2}(\mathbf{q}_i \cdot \mathbf{k}_j)^2 \approx \exp(\mathbf{q}_i \cdot \mathbf{k}_j) \tag{9.18}$$

$$\phi(\mathbf{x}) = \begin{bmatrix} 1 \\ x[0] \\ \vdots \\ x[d-1] \\ \frac{1}{2}x[0]x[0] \\ \vdots \\ \frac{1}{2}x[0]x[d-1] \\ \vdots \\ \frac{1}{2}x[d-1]x[0] \\ \vdots \\ \frac{1}{2}x[d-1]x[d-1] \end{bmatrix} \tag{9.19}$$

The paper by Katharopoulos et al. (2020) was called "Transformers are RNNs" because whereas the standard KV cache has size $O(n)$, the partial sums $\mathbf{a}_i$ and $\mathbf{b}_i$ have size $O(1)$. So we can think of them as the state of a recurrent neural network. Later, Dao and Gu (2024) published a paper called "Transformers are SSMs" arguing that linear transformers are a special case of SSMs, since Eqs. (9.15) and (9.16) are linear in the recurrent state. (The normalization in Eq. (9.17) seems not to fit perfectly, though.)

9.2.2 In-context learning

Perhaps the most surprising finding about GPT-3 was its ability to do *in-context learning* (Brown et al., 2020): when prompted with a few training examples (as opposed to fine-tuning on them), language models can learn from those examples and make good predictions on new examples. In part, in-context learning turns the question of trainability into a question of expressivity.

Remember back in Chapter 2 that in the perceptron algorithm, the weight vector is just a weighted sum of the training examples. This is true for many

other linear models. For example, in linear regression, we have training examples

$$\mathbf{x}_0, \ldots, \mathbf{x}_{n-1} \in \mathbb{R}^d \qquad \text{inputs} \tag{9.20}$$

$$y_0, \ldots, y_{n-1} \in \mathbb{R} \qquad \text{outputs.} \tag{9.21}$$

We want to minimize the squared error,

$$\mathcal{L}(\mathbf{w}) = \sum_{i=0}^{n-1} \frac{1}{2}(\mathbf{w} \cdot \mathbf{x}_i - y_i)^2. \tag{9.22}$$

The gradient is

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \sum_{i=0}^{n-1} (\mathbf{w} \cdot \mathbf{x}_i - y_i)\mathbf{x}_i \tag{9.23}$$

so the update rule is

$$\mathbf{w}^{(\ell+1)} = \mathbf{w}^{(\ell)} - \eta \sum_{i=0}^{n-1} (\mathbf{w}^{(\ell)} \cdot \mathbf{x}_i - y_i)\mathbf{x}_i. \tag{9.24}$$

In in-context learning, the training examples become input symbols, and attention is a weighted sum of (encodings of) input symbols. So we can start to see how transformers might be able to train on examples found in the input string.

(Von Oswald et al., 2023; Akyürek et al., 2024) analyze linear transformers with $\phi(\mathbf{x}) = \mathbf{x}$, doing in-context learning on linear regression. Construct a self-attention layer as follows. Assume that the input to the ℓ-th self-attention layer is, for $i \in [n]$:

$$\mathbf{h}_i^{(\ell)} = \begin{bmatrix} \mathbf{w}^{(\ell)} \\ \mathbf{x}_i \\ y_i \end{bmatrix}. \tag{9.25}$$

From this, compute queries, keys, and values:

$$\mathbf{q}_i = \begin{bmatrix} \mathbf{w}^{(\ell)} \\ -1 \end{bmatrix} \tag{9.26}$$

$$\mathbf{k}_j = \begin{bmatrix} \mathbf{x}_j \\ y_j \end{bmatrix} \tag{9.27}$$

$$\mathbf{v}_j = \begin{bmatrix} -\eta \mathbf{x}_j \\ \mathbf{0} \\ 0 \end{bmatrix}. \tag{9.28}$$

Then the self-attention's output (after the residual connection) is

$$\mathbf{c}_i^{(\ell+1)} = \mathbf{h}_i^{(\ell)} + \sum_{j=0}^{i} (\mathbf{q}_i \cdot \mathbf{k}_j)\mathbf{v}_j \tag{9.29}$$

$$= \begin{bmatrix} \mathbf{w}^{(\ell)} - \eta \sum_{j=0}^{i} (\mathbf{w}^{(\ell)} \cdot \mathbf{x}_j - y_j)\mathbf{x}_j \\ \mathbf{x}_i \\ y_i \end{bmatrix} \tag{9.30}$$

This is exactly equivalent to one step of gradient descent on the first i examples. The position-wise FFNN does nothing ($\mathbf{h}^{(\ell+1)} = \mathbf{c}^{(\ell+1)}$). If we repeat this layer L times, it simulates L steps of gradient descent.

Bibliography

Akyürek, Ekin, Dale Schuurmans, Jacob Andreas, Tengyu Ma, and Denny Zhou (2024). What learning algorithm is in-context learning? Investigations with linear models. In: *Proceedings of the Eleventh International Conference on Learning Representations (ICLR)*.

Allender, Eric, Nikhil Balaji, and Samir Datta (2014). Low-depth uniform threshold circuits and the bit-complexity of straight line programs. In: *Mathematical Foundations of Computer Science*. Vol. 8635. Lecture Notes in Computer Science. Springer, pp. 13–24.

Arora, Simran, Sabri Eyuboglu, Michael Zhang, Aman Timalsina, Silas Alberti, James Zou, Atri Rudra, and Christopher Re (2024). Simple linear attention language models balance the recall-throughput tradeoff. In: *Proceedings of the 41st International Conference on Machine Learning*. Vol. 235. Proceedings of Machine Learning Research, pp. 1763–1840.

Brown, Tom et al. (2020). Language models are few-shot learners. In: *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 1877–1901.

Dao, Tri and Albert Gu (2024). Transformers are SSMs: generalized models and efficient algorithms through structured state space duality. In: *Proceedings of the 41st International Conference on Machine Learning*. Vol. 235. Proceedings of Machine Learning Research, pp. 10041–10071.

Gu, Albert and Tri Dao (2023). Mamba: linear-time sequence modeling with selective state spaces. eprint: arXiv:2312.00752.

Gu, Albert, Isys Johnson, Karan Goel, Khaled Saab, Tri Dao, Atri Rudra, and Christopher Ré (2021). Combining recurrent, convolutional, and continuous-time models with linear state space layers. In: *Proc. NeurIPS*.

Gupta, Ankit, Albert Gu, and Jonathan Berant (2022). Diagonal state spaces are as effective as structured state spaces. In: *Advances in Neural Information Processing Systems*. Vol. 35, pp. 22982–22994.

Hesse, William, Eric Allender, and David A. Mix Barrington (2002). Uniform constant-depth threshold circuits for division and iterated multiplication. In: *Journal of Computer and System Sciences* 65.4, pp. 695–716.

Katharopoulos, Angelos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret (2020). Transformers are RNNs: fast autoregressive transformers with linear attention. In: *Proceedings of the 37th International Conference on Machine Learning*. Vol. 119. Proceedings of Machine Learning Research, pp. 5156–5165.

Merrill, William, Jackson Petty, and Ashish Sabharwal (2024). The illusion of state in state-space models. In: *Proc. ICML*.

Von Oswald, Johannes, Eyvind Niklasson, Ettore Randazzo, Joao Sacramento, Alexander Mordvintsev, Andrey Zhmoginov, and Max Vladymyrov (2023). Transformers learn in-context by gradient descent. In: *Proceedings of the 40th International Conference on Machine Learning (ICML)*. Vol. 202. Proceedings of Machine Learning Research, pp. 35151–35174.