EECS 182 Deep Neural Networks
Fall 2025 Anant Sahai and Gireeja Ranade

Discussion 9

1. SSM Convolution Kernel

Consider a discrete-time linear time-invariant State-Space Model (SSM) of the form

$$\mathbf{x}_{k+1} = A\mathbf{x}_k + Bu_k, \tag{1}$$
$$y_k = C\mathbf{x}_k + Du_k, \tag{2}$$

For simplicity (and realism vis-a-vis modern state-space models), consider both u and y to be scalars while the state $\mathbf{x}$ is a vector.

(a) **Convolution Kernel and the Output Equation.** Given that the sequence length is L (input: $(u_1, \cdots, u_L)$, output: $(y_1, \cdots, y_L)$) and assuming the initial state $\mathbf{x}_0 = \mathbf{0}$, show that the output y_k can be expressed as a *convolution* of the input sequence $\{u_\ell\}_1^L$ with a kernel $K = \{K_\ell\}_1^L$:

$$y_k = \sum_{\ell=0}^{L} K_\ell u_{k-\ell},$$

where any $u_{\leq 0}$ with a negative index is set to 0 (zero-padding). Also, find K.

Solution: From the SSM recursion $x_{k+1} = Ax_k + Bu_k$, we get

$$x_1 = Bu_0, \quad x_2 = ABu_0 + Bu_1, \quad x_3 = A^2Bu_0 + ABu_1 + Bu_2, \ldots$$

Hence,

$$y_0 = Du_0, \tag{3}$$
$$y_1 = Du_1 + CBu_0, \tag{4}$$
$$y_2 = Du_2 + CBu_1 + CABu_0, \tag{5}$$
$$y_3 = Du_3 + CBu_2 + CABu_1 + CA^2Bu_0, \tag{6}$$
$$\vdots \tag{7}$$

This summation matches a discrete convolution $\{y\} = K * \{u\}$ with kernel K given by

$$K = (D, CB, CAB, CA^2B, \ldots).$$

(b) **Efficient Computation with Convolutions.** These facts will be useful for this question:

- The convolution theorem states that if we have two sequences $u[k]$ and $v[k]$ where their discrete-time Fourier transforms are $U(f)$ and $V(f)$ respectively, then the discrete-time Fourier transform of their convolution $W[f] = \mathcal{F}\{u[k] * v[k]\} = U(f)V(f)$.

- The Fast Fourier Transform (FFT) algorithm can compute the discrete Fourier transform of a length-N sequence in $O(N \log N)$ time.
- The inverse discrete Fourier transform can also be computed using FFT in $O(N \log N)$ time.

If we already know the kernel K, how much can we parallelize the computation of a scalar output sequence $\{y_k\}$ for a scalar input sequence $\{u_k\}$ of length L? What is the minimum critical path length of the parallelized computation?

What about a naive, direct computation of y_k from the recursion?

Solution: Once the convolution kernel $K = (D, CB, CAB, \ldots, CA^{L-1}B)$ is known, computing the discrete convolution $\{y\} = K * \{u\}$ can be performed by zero-padding both sequences to length $2L$ and then using an FFT-based convolution (which computes circular convolutions). If we assume maximum parallelism, the depth of the FFT computations are $\mathcal{O}(\log L)$.

In contrast, naive computation would require one sequential pass through the entire sequence. Over $k = 1, \ldots, L$, this leads to at least $\mathcal{O}(L)$ recurrent steps which cannot be parallelized, resulting in a total depth of $\mathcal{O}(L)$ (critical path length).

(c) **Efficient Kernel Computation.** Given A, B, C are matrices, how can we compute the kernel K efficiently? What are some strategies to parallelize kernel computation? You may assume $L = 2^N$ for some N for simplicity.

Solution: To fully make use of parallel computational power of GPU, we want to compute $X^L = (I, A, \cdots, A^{L-1})$ fast. We can apply divide-and-conquer idea on this:

$$X^L = (X^{L/2}, A^{L/2}X^{L/2})$$

$$A^L = A^{L/2}A^{L/2}$$

For each $L = 2^l$ with $l \in \{1, 2, \cdots, N\}$, we can compute X^L through this recurrent formula so that we can compute the whole kernel with a maximum computation depth of $O(\log n \log L)$ where $O(\log n)$ is the maximum depth of computation for matrix multiplication.

(d) **Adding structure to A.** Suppose A is a diagonal matrix where we represent each diagonal entry as $a = \exp(\lambda_R + j\lambda_I)$. How can we leverage this structure to compute the kernel K more efficiently?

Solution: If A is diagonal, we no longer need to compute matrix multiplication, so we could pre-compute all $(I, A, \cdots, A^{L-1})$ in $O(\log L)$ time (instead of $O(\log n \log L)$). In practice, the matrix multiplication is often much more expensive than the theoretical limit $O(\log n)$, so removing the need to do matrix multiplication for computing the kernel speeds up things a lot more.

(e) **Linear Time-varying SSM.** Now consider a linear time-varying SSM of the form

$$\mathbf{x}_{k+1} = A_k\mathbf{x}_k + B_k u_k, \tag{8}$$

$$y_k = C_k\mathbf{x}_k + Du_k. \tag{9}$$

As in the previous part, assume that the matrices A_k are always diagonal.

As in part (a) where $\mathbf{x}_0 = 0$, express y_k as a specific convolution-like operation of the input sequence $\{u_\ell\}_1^L$ with a position-specific kernel $K_k = \{K_\ell\}_1^L$. How would you order the necessary operations to compute all the $\{y\}$ efficiently and exploit parallelism in your hardware?

Solution: From the SSM recursion $x_{k+1} = A_k x_k + B_k u_k$, we get

$$x_1 = B_0u_0, \quad x_2 = A_1B_0u_0 + B_1u_1, \quad x_3 = A_2A_1B_0u_0 + A_2B_1u_1 + B_2u_2, \ldots$$

Hence,

$$y_0 = Du_0, \tag{10}$$
$$y_1 = Du_1 + C_1B_0u_0, \tag{11}$$
$$y_2 = Du_2 + C_2B_1u_1 + C_2A_1B_0u_0, \tag{12}$$
$$y_3 = Du_3 + C_3B_2u_2 + C_3A_2B_1u_1 + C_3A_2A_1B_0u_0, \tag{13}$$
$$\vdots \tag{14}$$

This summation matches the discrete (position-specific) convolution $y_k = K_k * \{u\}$ with kernel K_k given by

$$K_k = C_k(0, B_{k-1}, A_{k-1}B_{k-2}, A_{k-1}A_{k-2}B_{k-3}, \cdots) + (D, 0, 0, 0, \dots).$$

To compute $\{y\}$ efficiently (across all positions), we can first compute the running product $P_\ell = \prod_{i=1}^{\ell} A_i$ for $\ell = 1, \cdots, L-1$. (Or do things in log-scale by keeping track of a running sum.) This requires linear time in the length of the sequence — this is called a "Parallel Scan" in the parallel-computing literature and is a very basic (but nontrivial) design pattern. See Parallel Prefix Sum (Scan) with CUDA for lots of detail. See also Prefix Sum - Work-efficient for a high-level description. But the important thing is that these can then be shared for the next step. We can then reuse each P_ℓ to compute each kernel K_k. This is because what we need is products of the form $\prod_{\ell=i}^{j} A_\ell = \frac{P_j}{P_{i-1}}$ where the division is component-wise on the diagonal. These can be computed in constant time once we have the P_ℓ already computed and the computation of the complete (scalar) coefficient $C_k \frac{P_{k-1}}{P_j} B_j$ to multiply by u_j can also be done in parallel across j in time that depends only on the dimensions involved — not the distance between k and j.

We can't do the FFT anymore, but this is not that bad.

(f) **Convolution vs. Recurrent.** Now that you have seen two ways of representing SSM (the recurrent one and the convolution one). In this question, we explore using SSM as an autoregressive generative model. $\{x_1, \cdots, x_t, \cdots\}$ is a sequence of text tokens and you would like the SSM to output $\{y_1, \cdots, y_t, \cdots\}$ such that $y_{t+1} = x_t$. **Would you use the convolution or the recurrent computation for training? What about during evaluation time where you would like to generate the text token by token?**

Solution: Convolution for training and recurrent for autoregressive generation. During training, both inputs and outputs are available to us, so we can use convolution to compute all of them at once. When doing autoregressive generation, y_t depends on $y_{t-1} = x_t$ which means that we must use the recurrent way to generate output y one step at a time.

2. Attention Mechanisms for Sequence Modelling

Sequence-to-Sequence is a powerful paradigm of formulating machine learning problems. Broadly, as long as we can formulate a problem as a mapping from a sequence of inputs to a sequence of outputs, we can use sequence-to-sequence models to solve it. For example, in machine translation, we can formulate the problem as a mapping from a sequence of words in one language to a sequence of words in another language. While some RNN architectures we previously covered possess the capability to maintain a memory of the previous inputs/outputs, to compute output, the memory states need to encompass information of many previous states, which can be difficult especially when performing tasks with long-term dependencies.

To understand the limitations of vanilla RNN architectures, we consider the task of changing the case of a sentence, given a prompt token. For example, given a mixed case sequence like "<U> I am a student", the

model should identify this as an upper-case task based on token <U>, and convert it to "I AM A STUDENT". Similarly, given "<L> I am a student", the lower-case task is to convert it to "i am a student".

We can formulate this task as a character-level sequence-to-sequence problem, where the input sequence is the mixed case sentence, and the output sequence is the desired case sentence. In this exercise, we use an encoder-decoder architecture to solve the task. The encoder is a vanilla RNN that takes the input sequence as input, and outputs a sequence of hidden states. The decoder is also a vanilla RNN that takes the last hidden state from the encoder as input, and outputs the desired case sentence.

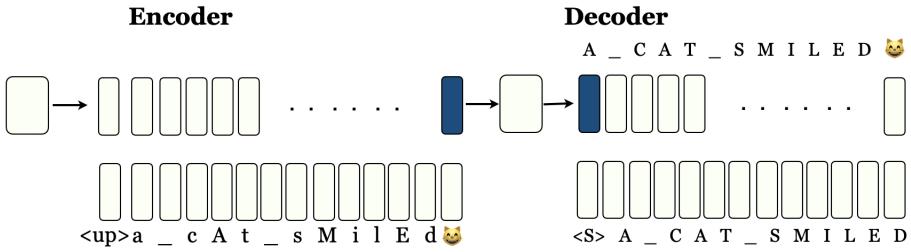

Figure 1: String Manipulation as a Sequence-to-Sequence Problem

(a) What information do RNNs store?

It is important to understand how information propagates through RNNs. Particularly in the context of sequence-to-sequence models, we want to understand what information is stored in the hidden states, and what information is stored in the weights (encoder & decoder). To understand this, we consider the different components of the RNN architecture.

- **Input sequence**: The input sequence is a sequence of T tokens.
- **Encoder Weights**: The shared learnable parameters of the encoder, W_{enc}
- **Bottleneck Activations**: The encoder hidden state at time T, that is passes to decoder.
- **Output sequence**: The output sequence is a sequence of T vectors (might be different length).
- **Decoder Weights**: The shared learnable parameters of the decoder, W_{dec}

Consider the following questions in the context of these modules:

i. Which of these components change during inference?
ii. When performing gradient based updates, how are the decoder weights trained? How is gradient propagated through the encoder?
iii. During training, what is the role of the input/output sequence?

Solution:

i. The encoder weights, and decoder weights do not change during inference. Once learned during training, they are fixed, while the input/output sequences, activations change.
ii. The output sequence with a cross-entropy loss is used to train the decoder weights. Note that the last hidden state of the encoder is used as initial state for the decoder. This allows us to compute gradients with respect to the decoder initial states, that is used for the encoder's hidden state.
iii. During training, the input sequence provides information about the *source* domain, and the output sequence provides information about the *target* domain. This information is used to learn domain specific parameters in the encoder and decoder weights.

(b) Information Bottleneck in RNNs

Consider the architecture shown in Figure 1. This is a simple encoder-decoder architecture with a single hidden layer in the encoder and decoder. The encoder takes the input sequence as input, and outputs a sequence of hidden states. The decoder takes the last hidden state from the encoder as input, and outputs the desired case sentence. **What information needs to be stored in the hidden state to perform the upper-case/lower-case task? Are there any limitations to this architecture?**

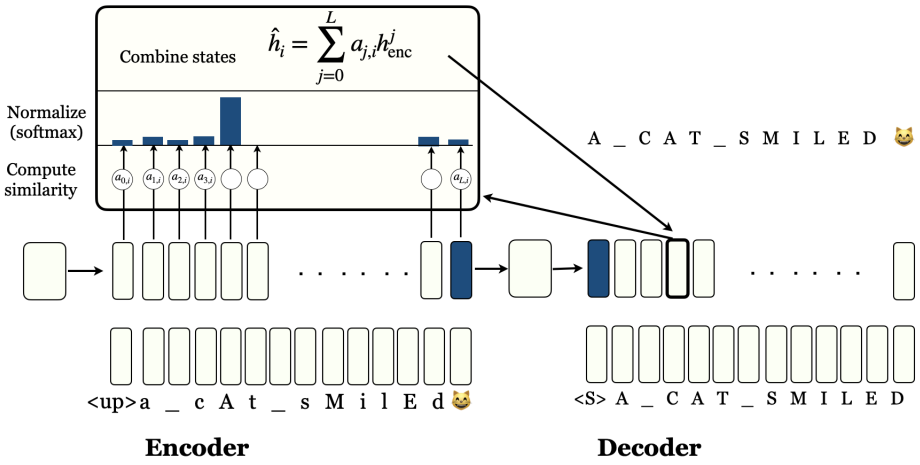

Figure 2: Attention Mechanism for Sequence Modelling with RNNs.

Solution:

i. We need to compress the entire sequence to fit in the memory of the RNN-Cell alongside the task-identifier.

ii. One major limitation of the architecture is the encoder bottleneck-activation that is passed to the decoder. This means that the hidden state at the last time step should contain information about the entire input sequence. This can be difficult especially when performing tasks with long-term dependencies (e.g. task-identifier token is at th beginning of the sentence.)

(c) Attention & RNNs

How does adding attention allow the model to bypass the information bottleneck? In particular, what information in the following modules would allow the model to perform the capitalization task ?

- **Encoder Weights**
- **Attention Scores**
- **Bottleneck Activations**
- **Decoder Weights**

Solution:

i. The encoder weights need to learn a representation of the position of a particular token in the input-sequence.

ii. The attention scores computes similarity between the decoder "query" vector and the hidden-states of the encoder. As long as it scores the token at the same index as the query vector, it can be used to perform the task.

iii. The bottleneck activation no-longer needs to store information about the entire input sequence, since we are allowed to perform a look-up with the attention scores.

iv. The decoder weights learn to count, that is used to identify which token in the output-sequence we are decoding.

3. Query-Key-Value Mechanics in Self-Attention

Self-attention is the core building block for the Transformer model, which has kickstarted the amazing progress in recent deep learning foundation models. The concept of self-attention is not restricted to Transformer exclusively though. In this question, you will be studying the detailed mechanics of the Query-Key-Value interaction in a self-attention block, in the context of RNN. Here we will be studying the case where the encoder only takes in 3 inputs, with the variables defined as:

$$h_1 = \begin{bmatrix}5\\1\end{bmatrix} \quad h_2 = \begin{bmatrix}4\\3\end{bmatrix} \quad h_3 = \begin{bmatrix}-2\\6\end{bmatrix} \quad W_q = \begin{bmatrix}1 & 0\\2 & 9\end{bmatrix} \quad W_k = \begin{bmatrix}0 & 1\\6 & 0\end{bmatrix} \quad W_v = \begin{bmatrix}4 & 3\\9 & 1\end{bmatrix}$$

where h_i denotes the hidden states at timestep i at some arbitrary self-attention layer. Each q_i, k_i, v_i is determined by $q_i = W_q h_i$, $k_i = W_k h_i$, $v_i = W_v h_i$.

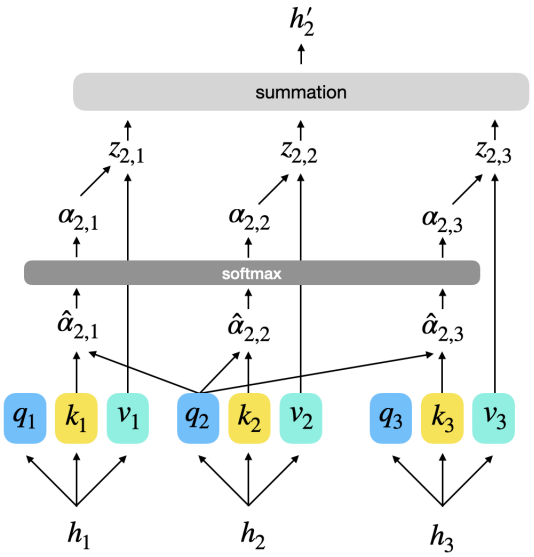

Figure 3: Self-attention of a timestep in Encoder

(a) **Compute $\hat{\alpha}_{2,1}$, $\hat{\alpha}_{2,2}$, $\hat{\alpha}_{2,3}$.**

Solution:

$$\hat{\alpha}_{2,1} = q_2^T k_1 = (W_q h_2)^T (W_k h_1) = (\begin{bmatrix}1 & 0\\2 & 9\end{bmatrix}\begin{bmatrix}4\\3\end{bmatrix})^T(\begin{bmatrix}0 & 1\\6 & 0\end{bmatrix}\begin{bmatrix}5\\1\end{bmatrix}) = \begin{bmatrix}4 & 35\end{bmatrix}\begin{bmatrix}1\\30\end{bmatrix} = 1054 \quad (15)$$

$$\hat{\alpha}_{2,2} = q_2^T k_2 = (W_q h_2)^T (W_k h_2) = \begin{bmatrix}4 & 35\end{bmatrix}(\begin{bmatrix}0 & 1\\6 & 0\end{bmatrix}\begin{bmatrix}4\\3\end{bmatrix}) = \begin{bmatrix}4 & 35\end{bmatrix}\begin{bmatrix}3\\24\end{bmatrix} = 852 \quad (16)$$

$$\hat{\alpha}_{2,3} = q_2^T k_3 = (W_q h_2)^T (W_k h_3) = \begin{bmatrix}4 & 35\end{bmatrix}(\begin{bmatrix}0 & 1\\6 & 0\end{bmatrix}\begin{bmatrix}-2\\6\end{bmatrix}) = \begin{bmatrix}4 & 35\end{bmatrix}\begin{bmatrix}6\\-12\end{bmatrix} = -396 \quad (17)$$

(18)

(b) Fig 3 shows a rough sketch of how self-attention is performed in one timestep of a Transformer block. **Write out these operations in equations, ie.** $h'_2 = \textbf{\textit{SelfAttention}}(h_1, h_2, h_3)$**, what is *SelfAttention*?** You can define intermediate variables instead of expressing everything in one line.

Solution:

$$\begin{aligned} h'_2 &= \textit{SelfAttention}(h_1, h_2, h_3) \\ &= z_{2,1} + z_{2,2} + z_{2,3} = \alpha_{2,1} v_1 + \alpha_{2,2} v_2 + \alpha_{2,3} v_3 \end{aligned}$$

where $\boldsymbol{\alpha_2} = \text{softmax}(\hat{\boldsymbol{\alpha_2}})$ as computed in the previous part.

(c) For simplicity, let's use **argmax** instead of the softmax layer in the diagram. What is h'_2 in this case?

Solution:

Since we are using argmax, $\boldsymbol{\alpha_2}$ will simply be $[1, 0, 0]$ and thus $h'_2 = z_{2,1} = v_1 = W_v h_1$ which is equal to:

$$\begin{bmatrix} 4 & 3 \\ 9 & 1 \end{bmatrix} \begin{bmatrix} 5 \\ 1 \end{bmatrix} = \begin{bmatrix} 23 \\ 46 \end{bmatrix}$$

(d) In practice, it is common to have multiple self-attention operations happening in one layer. Each self-attention block is refered as a *head*, and thus the entire block is typically called *Multi-head Self-Attention (MSA)* in papers. **What's the benefit of having multiple heads?** (*Hint*: why do we want multiple kernel filters in ConvNets?)

Solution:

Having multiple heads of self-attention enables each MSA block to activate different features, which can be helpful for long range sequence modeling. Think of MSA block as multiple kernel filters, each with a receptive field that spans the entire sequence. Why can't one self-attention head capture all the features? Theoretically it could, but as we've seen before, softmax will amplify large signals and mute the smaller ones, which is why having multiple heads can ensure more features being activated.

Contributors:

- Naman Jain.
- Qiyang Li.
- Anant Sahai.
- Sultan Daniels.

- Gireeja Ranade.
- Kumar Krishna Agrawal.
- Kevin Li.

