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 $K$.

(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?

(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.

(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?

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

$$\mathbf{x}_{k+1} = A_k\mathbf{x}_k + B_k u_k, \tag{3}$$
$$y_k = C_k\mathbf{x}_k + Du_k. \tag{4}$$

  

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?

(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?**

## 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.

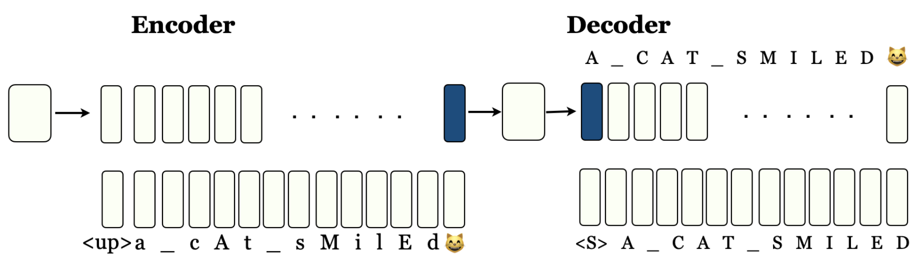


**Figure 1:** String Manipulation as a Sequence-to-Sequence Problem

### (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, $W_{\text{enc}}$
- **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, $W_{\text{dec}}$

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?

## (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?**

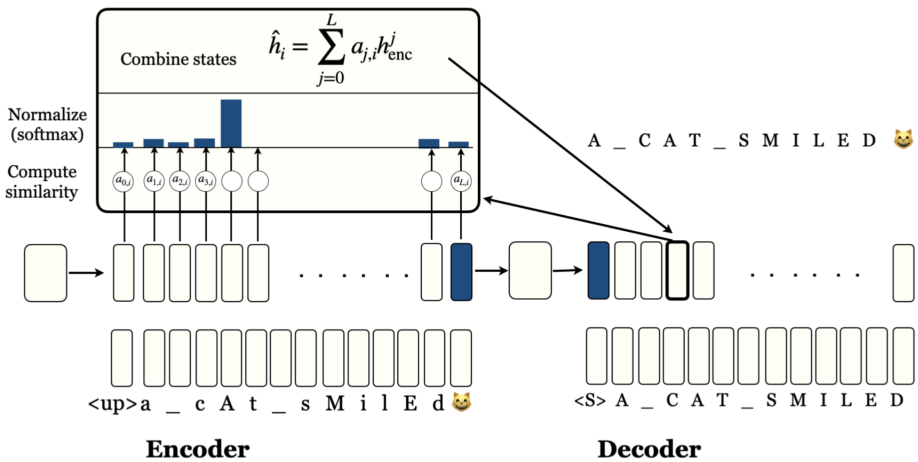


**Figure 2:** Attention Mechanism for Sequence Modelling with RNNs.

## (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**

  

## 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 $q_i = W_q h_i$, $k_i = W_k h_i$, $v_i = W_v h_i$.

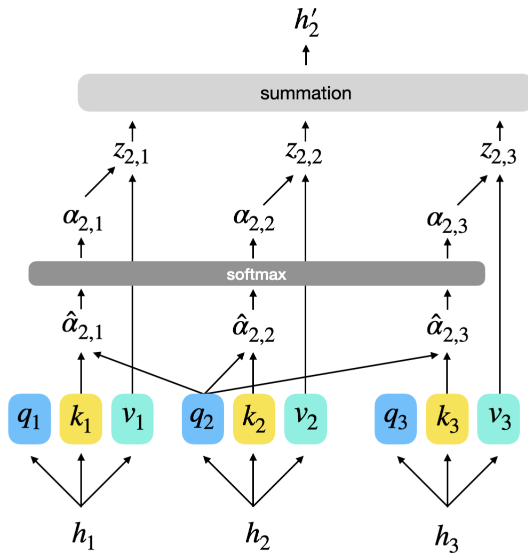


**Figure 3:** Self-attention of a timestep in Encoder

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

(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.** $h'_2 = \textit{SelfAttention}(h_1, h_2, h_3)$**, what is *SelfAttention*?** You can define intermediate variables instead of expressing everything in one line.

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

(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?)

  

**Contributors:**

- Naman Jain.
- Qiyang Li.
- Anant Sahai.
- Sultan Daniels.
- Gireeja Ranade.
- Kumar Krishna Agrawal.
- Kevin Li.

  