EECS 182 Deep Neural Networks
Spring 2023 Anant Sahai

Discussion 8

1. 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.

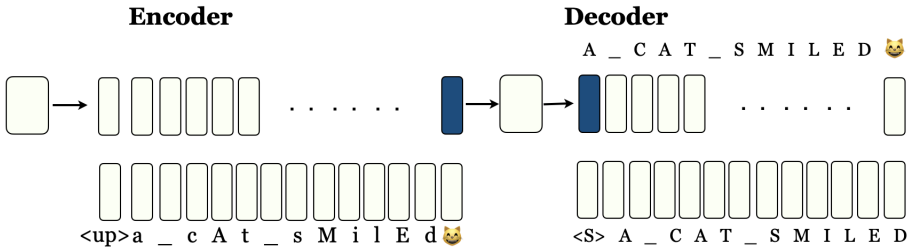

Figure 1: String Manipulation as a Sequence-to-Sequence Problem

(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, $W_{\text{enc}} \in \mathbb{R}^{d \times h}$
- **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, $W_{\text{dec}} \in \mathbb{R}^{d \times h}$

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?**

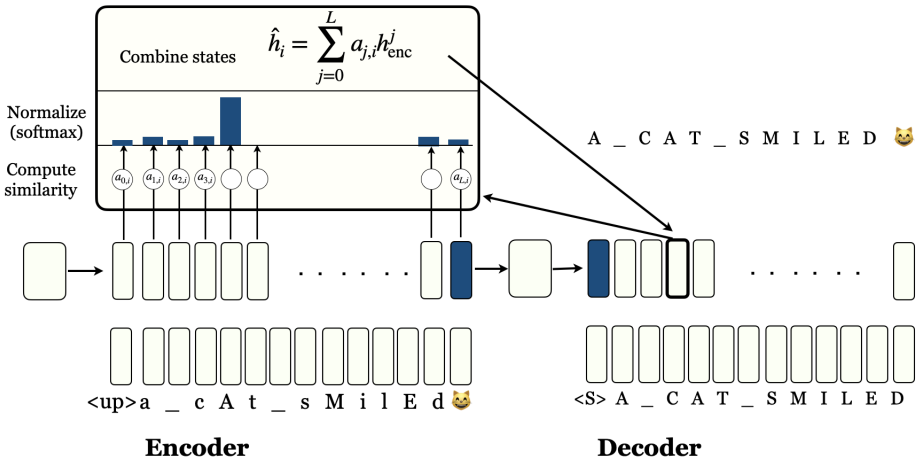

Figure 2: Attention Mechanism for Sequence Modelling with RNNs.

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) Introducing Attention

Instead of storing all the information in the hidden state, we can use attention to selectively store information. The idea of attention is to query the encoder hidden states with a query vector, and use the resulting attention weights to compute a weighted sum of the hidden states. This weighted sum is then used as the input to the readout layer that computes the output token at each time step of the decoder. **How would you modify the encoder-decoder architecture to incorporate attention?**

Solution: Discuss the figure in Figure 2, going through computing similarity, normalizing the scores, and generating the context vector.

(d) 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.

(e) Positional Encoding

As noted above, the hidden state of the encoder at position t should contain information about the position of token in the input sequence. To incorporate this information, we can add a *positional encoding* to the input tokens. **For sequences of length T discuss how you would add positional encoding to the input sequence.**

Solution:

i. **One-hot positional encoding**: Alongside the embedding of the input token, we concatenate a one-hot vector that encodes the position of the token in the sequence. For example, for a sequence of length T, the one-hot vector for the first token is $[1, 0, \ldots, 0]$, and the one-hot vector for the last token is $[0, 0, \ldots, 1]$. A disadvantage of this approach is that the positional encoding is not differentiable, and hence cannot be learned. Further, the length of the encoding grows linearly with the sequence length, which can be problematic for long sequences.

ii. **Sine positional encoding**: Alongside the embedding of the input token, we concatenate a sine positional encoding vector that encodes the position of the token in the sequence. For example,

for a sequence of length T, we can use the following sine positional encoding:

$$\text{PE}(t) = \begin{bmatrix} \sin(\omega_1 \cdot t) \\ \cos(\omega_1 \cdot t) \\ \sin(\omega_2 \cdot t) \\ \cos(\omega_2 \cdot t) \\ \vdots \\ \sin(\omega_k \cdot t) \\ \cos(\omega_k \cdot t) \end{bmatrix} \tag{1}$$

where w_k denotes the frequency, which varies at each position. One common choice of w_k is $w_k = \frac{1}{10000^{2k/d}}$ where d is the dimension of $\text{PE}(t)$. This way PE forms a geometric progression from $2\ \pi$ to $10000 \cdot 2\pi$ on the wavelengths. A few reminders for understanding the sine position embedding:

- Sine PE is defined *before* model training even starts. There's no learnable parameter in the above vector. Since different entries have different frequencies, PE explicitly appends each timestep's input with signals to capture the relative positions.
- The dimension d for sine PE is a hyperparameter, which needs not to be equal to $2T$, where T is the number of total time steps.

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

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

Solution:

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

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

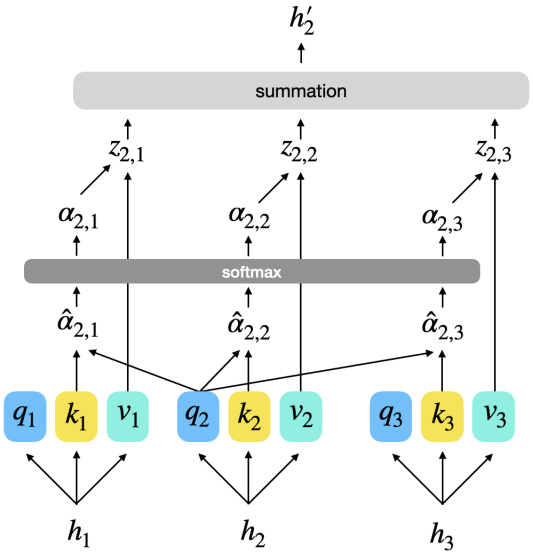

Figure 3: Self-attention of a timestep in Encoder

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

$$\tag{5}$$

(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:

- Kumar Krishna Agrawal.
- Kevin Li.

