EECS 182 Deep Neural Networks
Fall 2022 Discussion Worksheet

Discussion 7

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.

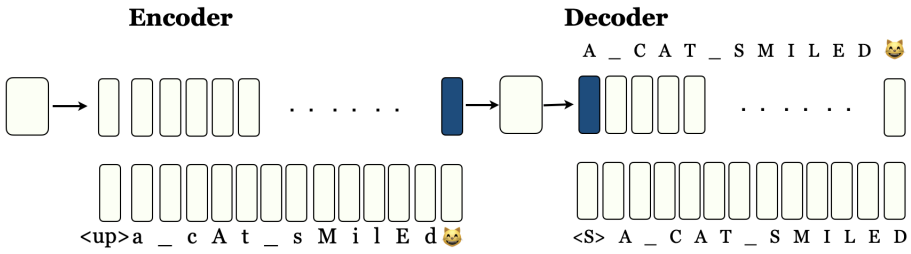

Figure 1: String Manipulation as a Sequence-to-Sequence Problem

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

 All Rights Reserved. This may not be publicly shared without explicit permission.

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. What is the qualitative nature of information captured by the weights?
iv. 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. The encoder/decoder weights are trained to generate token-specific hidden-state representations that carry sufficient information to solve the task.
iv. 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?

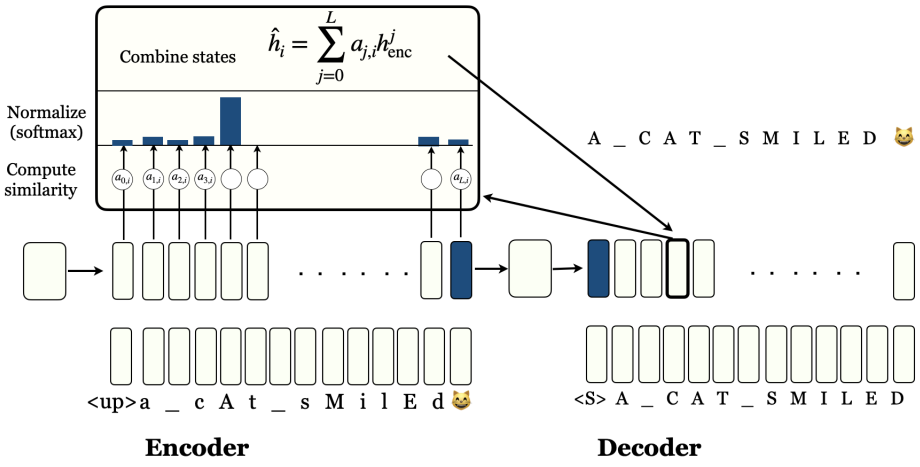

Figure 2: Attention Mechanism for Sequence Modelling with RNNs.

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

 All Rights Reserved. This may not be publicly shared without explicit permission.

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.

Staff Meta: The TA should draw out what the attention block as a soft queriable hash-table

(d) Attention & RNNs

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

Staff Meta: This is a group-discussion question.

(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.

 All Rights Reserved. This may not be publicly shared without explicit permission.

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:

$$\mathrm{PE}(t) = \begin{bmatrix} \sin\left(\frac{t}{10000^{2i/d}}\right) \\ \cos\left(\frac{t}{10000^{2i/d}}\right) \end{bmatrix} \tag{1}$$

(f) Prompt-Engineering : Alternative Designs

Based on our understanding of challenges with modelling long-range dependencies, a potential bottleneck in the previous design is that the prompt token is passed at the beginning of the input-sequence to the encoder. This means that the encoder bottleneck activation has to store information about the prompt token, and the entire input sequence. Are there any alternative ways of tokenizing the sequences? Discuss the implications for both vanilla Encoder-Decoder models, and models with attention.

Solution: One alternative design is to instead pass the prompt token as the first token to the decoder, instead of a generic <BOS> (begin of sentence) token.

i. **Vanilla RNN** : This change would allow the encoder to only store information about the input sequence, and not the prompt token. However, the decoder would still need to store information about the prompt token, and for long-sequences this might still be a bottleneck.

ii. **w/ Attention** : To fix the bottleneck that arises from seeing the prompt token only once at the beginning of the output sequence, we could additionally use attention on the decoder states to allow the decoder additional pathways to "peek" at the prompt token at each time step.

(g) Limits of Expressivity

Let's consider a different seq-to-seq task, where given a sentence with arbitrary number of spaces, we want to remove all the spaces. For example, given the sentence "I am a student", we want to output "Iamastudent". Informally, using the same architecture with attention as before, could we solve this task? Are there any simple modifications that could resolve this limitation? *(Hint : Think about modifying the output vocabulary.)*

Solution:

i. **Variable length output**: First thing to note is that unlike the previous examples, now the output sequence is of variable length, strictly shorter than the input sequence.

ii. **Need for Loops**: For a single-layer encoder-decoder architecture w/o any modifications, solving this task is not possible. Intuitively, the decoder needs to "skip" all spaces b/w two non-space tokens, and identify the correct position in the input sequence to attend to. However, since the encoder hidden-state effectively stores only token-specific information, and positional encoding, the attention score for the space tokens would be more-or-less the same with some variablity across positions. Alternative way of thinking about this, is that the decoder needs to "loop" over the input sequence, incrementing it's counter by 1, and generating a token only when it sees a non-space token. However, as we need to generate samples at every time step, this is not possible.

iii. **Modifying the output vocabulary**: One way to solve this problem is to modify the output vocabulary to include a special "NULL" token, and padding the output sequence with "NULL" tokens. This way, the decoder can learn to generate a "NULL" token when it sees a space token, and the output sequence will be of the same length as the input sequence. At post-processing time, we can simply remove all the "NULL" tokens from the output sequence.

 All Rights Reserved. This may not be publicly shared without explicit permission.

Staff Meta: Draw out an example on the board to highlight the challenge of variable length output.

Contributors:

- Kumar Krishna Agrawal.

 All Rights Reserved. This may not be publicly shared without explicit permission.