SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

50.040 Natural Language Processing, Fall 2025

Homework 2

Due 26 October 2025, 23:59pm

Homework 2 will be graded by Shaoyang Xu
shaoyang_xu@mymail.sutd.edu.sg

One of the core challenges in NLP is how to represent language, the nature of which lies in its sequential structure of words. To address this, various sequence modeling architectures have been proposed. The most prominent models include the early RNN (Recurrent Neural Network), LSTM (Long Short-Term Memory network), and the more recent Transformer. They differ primarily in how they processes information. RNN processes sequences step by step but struggles with long-range dependencies. LSTM introduces gating mechanisms to selectively retain information over long sequences. In contrast, Transformer relies on attention mechanisms to capture dependencies across all positions in a sequence.

This homework provides a concise review of RNN and LSTM, with particular emphasis on how they model contextual information. It then shifts focus to attention mechanism, introducing its basic operations, its applications in sequence-to-sequence learning, and its multi-head variant.

1 RNN and LSTM

The primary challenge of processing a sequence $(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T)$ lies in capturing contextual dependencies among elements. Although conventional feedforward neural networks treat each element independently, RNN introduces a hidden state $\mathbf{h}_t$ that serves as an internal memory, allowing the model to retain and utilize information from previous time steps.

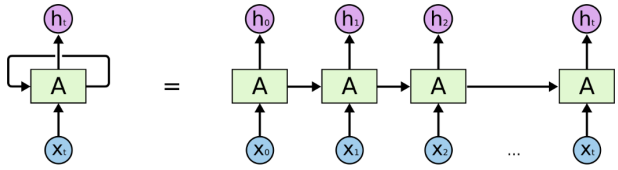

Figure 1: RNN

For a simple RNN with hidden size h and input size d, the update of $\mathbf{h}_t$ is:

$$\mathbf{h}_t = \tanh(\mathbf{x}_t \mathbf{W}_x + \mathbf{h}_{t-1} \mathbf{W}_h + \mathbf{b}) \tag{1}$$

Here, $\mathbf{x}_t \in \mathcal{R}^d$ represents the input at the current time step, which could be a word embedding or a feature vector. The term $\mathbf{h}_{t-1} \in \mathcal{R}^h$ is the hidden state from the previous step. The matrices $\mathbf{W}_x \in \mathcal{R}^{d \times h}$ and $\mathbf{W}_h \in \mathcal{R}^{h \times h}$ transform the input and the previous hidden state into the hidden space, while $\mathbf{b} \in \mathcal{R}^h$ is a bias term. The tanh introduces nonlinearity into the model, squashing the combined input to values between -1 and 1.

The hidden state of RNN naturally propagates information through time. However, when the sequence becomes long, RNN faces difficulties such as vanishing and exploding gradients, which make RNN hard to learn long-term dependencies.

To address this, LSTM was introduced. It extends RNN by adding a cell state $\mathbf{c}_t$ for long-term memory and a set of gates that control which information to remember, forget, or output.

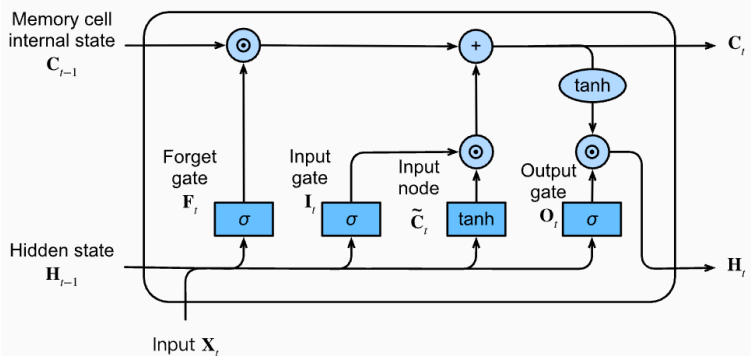

Figure 2: LSTM

The forward pass for an LSTM at a single time step updates both the hidden state $\mathbf{h}_t$ and the cell state $\mathbf{c}_t$ as follows:

$$
\begin{aligned}
\mathbf{f}_t &= \sigma\big(\mathbf{x}_t\mathbf{W}_f + \mathbf{h}_{t-1}\mathbf{U}_f + \mathbf{b}_f\big),\\
\mathbf{i}_t &= \sigma\big(\mathbf{x}_t\mathbf{W}_i + \mathbf{h}_{t-1}\mathbf{U}_i + \mathbf{b}_i\big),\\
\mathbf{o}_t &= \sigma\big(\mathbf{x}_t\mathbf{W}_o + \mathbf{h}_{t-1}\mathbf{U}_o + \mathbf{b}_o\big),\\
\tilde{\mathbf{c}}_t &= \tanh\big(\mathbf{x}_t\mathbf{W}_c + \mathbf{h}_{t-1}\mathbf{U}_c + \mathbf{b}_c\big),\\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t,\\
\mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t).
\end{aligned}
\tag{2}
$$

Here, $\mathbf{f}_t$, $\mathbf{i}_t$, and $\mathbf{o}_t$ are the forget, input, and output gates, respectively. The forget gate $\mathbf{f}_t$ determines how much of the previous cell state $\mathbf{c}_{t-1}$ should be retained. The input gate $\mathbf{i}_t$ controls how much of the new candidate memory $\tilde{\mathbf{c}}_t$ should be added to the cell state. The updated cell state $\mathbf{c}_t$ combines retained memory and newly selected information. The output gate $\mathbf{o}_t$ modulates the information from the cell state that will be exposed as the hidden state $\mathbf{h}_t$. The element-wise multiplication $\odot$ ensures selective gating, allowing the network to retain or suppress specific components of memory. The $\mathbf{W}_f, \mathbf{W}_i, \mathbf{W}_o, \mathbf{W}_c \in \mathcal{R}^{d\times h}$ and $\mathbf{U}_f, \mathbf{U}_i, \mathbf{U}_o, \mathbf{U}_c \in \mathcal{R}^{h\times h}$ are weight matrices of parameters conducting projections. The function σ represents the sigmoid activation, which outputs values between 0 and 1, controlling how much information flows through each gate.

Compared to a vanilla RNN, LSTM gives finer control over memory. It can remember things for a long time and forget things when they are no longer relevant. This is why it performs much better at capturing long-term dependencies.

Question 1 [code] (10 points)

Implement the single-step forward pass for both an RNN and an LSTM.

Question 2 [code] (10 points)

Implement the sequence-level forward functions that iterate through all time steps, updating the hidden state, and in the case of LSTM, also updating the cell state at each step. You should leverage the single-step functions implemented in Question 1.

2 Attention Mechanisms

Consider the following: denote by $\mathcal{D} = \{(\mathbf{k}_1, \mathbf{v}_1), \ldots, (\mathbf{k}_m, \mathbf{v}_m)\}$ a database of m tuples of *keys* and *values*. Moreover, denote by $\mathbf{q}$ a *query*. Then we can define the *attention* over $\mathcal{D}$ as

$$\text{Attention}(\mathbf{q}, \mathcal{D}) = \sum_{i=1}^{m} \alpha(\mathbf{q}, \mathbf{k}_i)\mathbf{v}_i, \tag{3}$$

where $\alpha(\mathbf{q}, \mathbf{k}_i) \in \mathcal{R}$ $(i = 1, \ldots, m)$ are scalar attention weights. This operation is commonly known as *attention pooling*. The term *attention* reflects the mechanism's ability to focus on specific elements in the dataset, assigning higher weights α to the terms in $\mathcal{D}$ that are deemed more relevant or significant. Consequently, the attention mechanism produces a weighted linear combination of the values in the database, emphasizing the most important components.

A common strategy for ensuring that the weights sum up to 1 is to normalize them via

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \frac{\alpha(\mathbf{q}, \mathbf{k}_i)}{\sum_j \alpha(\mathbf{q}, \mathbf{k}_j)}. \tag{4}$$

In particular, to ensure that the weights are also nonnegative, one can resort to exponentiation. This means that we can now pick any function $a(\mathbf{q}, \mathbf{k})$ and then apply the softmax operation used for multinomial models to it via

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \frac{\exp(a(\mathbf{q}, \mathbf{k}_i))}{\sum_j \exp(a(\mathbf{q}, \mathbf{k}_j))}. \tag{5}$$

Then, we need to keep the order of magnitude of the arguments in the exponential function under control. Assume that all the elements of the query $\mathbf{q} \in \mathcal{R}^d$ and the key $\mathbf{k}_i \in \mathcal{R}^d$ are independent and identically drawn random variables with zero mean and unit variance. The dot product between both vectors has zero mean and a variance of d. To ensure that the variance of the dot product still remains 1 regardless of vector length, we use the *scaled dot product attention* scoring function. That is, we rescale the dot product by $1/\sqrt{d}$. We thus arrive at the first commonly used attention function that is used:

$$a(\mathbf{q}, \mathbf{k}_i) = \frac{\mathbf{q}^\top \mathbf{k}_i}{\sqrt{d}}. \tag{6}$$

Note that attention weights α still need normalizing. We can simplify this further via equation 5 by using the softmax operation:

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \text{softmax}(a(\mathbf{q}, \mathbf{k}_i)) = \frac{\exp(\mathbf{q}^\top \mathbf{k}_i/\sqrt{d})}{\sum_{j=1}^{m} \exp(\mathbf{q}^\top \mathbf{k}_j/\sqrt{d})}. \tag{7}$$

Question 3 [code] (15 points)

In sequence modeling, different input sequences often have different lengths. For example, consider the following batch of three sentences (each padded to the same maximum length):

Study	about	Deep	Learning
Start	by	code	`<pad>`
Hello	world	`<pad>`	`<pad>`

where the valid lengths are [4, 3, 2], and `<pad>` represents padding tokens that should not affect the computation.

To put them in the same batch and compute correctly, we usually apply a mask so that positions beyond the valid length are ignored — that is, their probabilities after softmax become zero.

Implement the function *masked_softmax*, which performs the softmax operation but masks out elements beyond the valid lengths. Then, run the sanity check cell to check your implementation.

Hints: You may find the following functions helpful (but you are not required to use them):

- `nn.functional.softmax`
- `torch.arange`

Question 4 (15 points)

In practice, we often think of minibatches for efficiency, such as computing attention for n queries and m key-value pairs, where queries and keys are of dimension d and values are of dimension v. The scaled dot product attention of queries $\mathbf{Q} \in \mathcal{R}^{n \times d}$, keys $\mathbf{K} \in \mathcal{R}^{m \times d}$, and values $\mathbf{V} \in \mathcal{R}^{m \times v}$ thus can be written as

$$\text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right)\mathbf{V} \in \mathcal{R}^{n \times v}. \tag{8}$$

Question 4.1 [written] (5 points) Write the shape of queries, keys and values during the calculation of scaled dot product attention. You should fill in the shape inside the code box.

Question 4.2 [code] (10 points) Implement function *DotProductAttention* that calculates the scaled dot product attention. Then, run the sanity check cell to check your implementation.

Hints:

- You should use the previously implemented *masked_softmax* function to handle sequences of different lengths.
- You may find the following functions helpful (but you are not required to use them): `torch.bmm`

3 Attention Seq2Seq

Attention mechanisms can be effectively integrated into encoder-decoder architectures for sequence-to-sequence learning. Traditionally, in an RNN-based approach, all relevant information from the source sequence is encoded into a fixed-dimensional state representation by the encoder. However, rather than maintaining this state—represented by the context variable $\mathbf{c}$ that summarizes the source sentence—as a fixed value, it can be dynamically updated. This update is based on both the original text (encoder hidden states $\mathbf{h}_t$) and the previously generated text (decoder hidden states $\mathbf{s}_{t'-1}$). As a result, we obtain an updated context variable $\mathbf{c}_{t'}$ after each decoding time step t'. This approach allows the model to adapt the context dynamically, even for input sequences of length T, thereby improving the ability to handle long-range dependencies and capture more nuanced information from the source sequence. In this case, the context variable is the output of attention pooling:

$$\mathbf{c}_{t'} = \sum_{t=1}^{T} \alpha(\mathbf{s}_{t'-1}, \mathbf{h}_t)\mathbf{h}_t. \tag{9}$$

We used $\mathbf{s}_{t'-1}$ as the query, and $\mathbf{h}_t$ as both the key and the value. Note that $\mathbf{c}_{t'}$ is then used to generate the state $\mathbf{s}_{t'}$ and to generate a new token.

Question 5 [code] (10 points)

Implement the RNN decoder in the *Seq2SeqAttentionDecoder* class. The decoder's state is initialized using three components: (i) the hidden states of the encoder's last layer across all time steps, which are utilized as keys and values for the attention mechanism; (ii) the hidden state of the encoder's final time step at all layers, which initializes the decoder's hidden state; and (iii) the valid length of the encoder to exclude padding tokens during attention pooling. During each decoding time step, the hidden state of the decoder's final layer from the previous step is used as the query for the attention mechanism. The attention mechanism's output is then concatenated with the input embedding to form the input for the RNN decoder, effectively guiding the generation process with context from both the source sequence and previous decoder outputs. Then, run the sanity check cell to check your implementation.

4 Multi-head attention

Rather than relying on a single attention pooling operation, the queries, keys, and values can be transformed through h independently learned linear projections. These h projected queries, keys, and values are then processed in parallel through attention pooling. Afterward, the h resulting attention outputs, known as *heads*, are concatenated and passed through another learned linear projection to generate the final output.

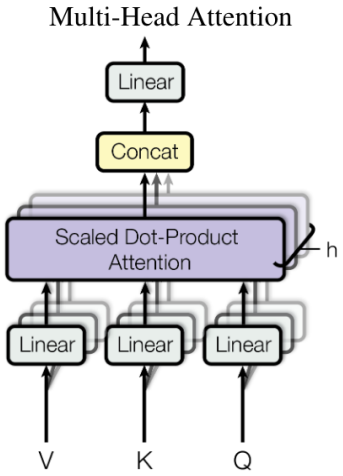

Figure 3: Multihead attention

This architecture, referred to as *multi-head attention*, allows each attention head to focus on different parts of the input, enabling the model to capture a wider range of information.

Given a query $\mathbf{q} \in \mathcal{R}^{d_q}$, a key $\mathbf{k} \in \mathcal{R}^{d_k}$, and a value $\mathbf{v} \in \mathcal{R}^{d_v}$, each attention head $\mathbf{h}_i$ $(i = 1, \ldots, h)$ is computed as:

$$\mathbf{h}_i = f(\mathbf{W}_i^{(q)}\mathbf{q}, \mathbf{W}_i^{(k)}\mathbf{k}, \mathbf{W}_i^{(v)}\mathbf{v}) \in \mathcal{R}^{p_v}, \tag{10}$$

where $\mathbf{W}_i^{(q)} \in \mathcal{R}^{p_q \times d_q}$, $\mathbf{W}_i^{(k)} \in \mathcal{R}^{p_k \times d_k}$, and $\mathbf{W}_i^{(v)} \in \mathcal{R}^{p_v \times d_v}$ are learnable parameters and f is attention pooling. The multi-head attention output is another linear transformation via learnable parameters $\mathbf{W}_o \in \mathcal{R}^{p_o \times hp_v}$ of the concatenation of h heads:

$$\mathbf{W}_o \begin{bmatrix} \mathbf{h}_1 \\ \vdots \\ \mathbf{h}_h \end{bmatrix} \in \mathcal{R}^{p_o}. \tag{11}$$

Based on this design, each head may attend to different parts of the input. More sophisticated functions than the simple weighted average can be expressed.

Question 6 [written] (10 points)

Please describe the benefits of using multi-head attention instead of single head attention.

Question 7 [code] (30 points)

In this implementation, we choose the scaled dot product attention for each head of the multi-head attention. To avoid significant growth of computational cost and parameterization cost, we set $p_q = p_k = p_v = p_o/h$. Note that h heads can be computed in parallel if we set the number of outputs of linear transformations for the query, key, and value to $p_q h = p_k h = p_v h = p_o$. In the following implementation, p_o is specified via the argument `num_hiddens`.

To allow for parallel computation of multiple heads, the *MultiHeadAttention* class uses two transposition methods *transpose_qkv* and *transpose_output*. Specifically, the *transpose_output* method reverses the operation of the *transpose_qkv* method.

Question 7.1 [code] (10 points) Implement function *transpose_qkv*, which is the transposition for parallel computation of multiple attention heads.

Question 7.2 [code] (10 points) Implement function *transpose_output* that reverse the operation of *transpose_qkv*.

Question 7.3 [code] (10 points) Complete *MultiHeadAttention* class. (Hint: you can use the two function you defined in question 7.1 and 7.2 .)

Hints: You may find the following functions helpful (but you are not required to use them):

- `torch.repeat_interleave`
- `torch.reshape`
- `torch.permute`

How to submit

1. Fill up your student ID and name in the Jupyter Notebook.
2. Click the Save button at the top of the Jupyter Notebook.
3. Select Cell - All Output - Clear. This will clear all the outputs from all cells (but will keep the content of all cells).
4. Select Cell Run All. This will run all the cells in order, and will take several minutes.
5. Once you've rerun everything, select File – Download as – PDF via LaTeX.
6. Look at the PDF file and make sure all your solutions are there, displayed correctly. **The PDF is the only thing your graders will see!**
7. Submit your PDF on eDimension.