Problem Set 2

Important:

- Write your name as well as your NU ID on your assignment. Please number your problems.
- Submit both results and your code.
- Give complete answers. Do not just give the final answer; instead show steps you went through to get there and explain what you are doing. Do not leave out critical intermediate steps.
- This assignment must be submitted electronically through Gradescope by October 20th 2025 (Monday) by 11:59 PM.

1 Setting Up Your Local Environment and Cloud GPU VM

1.1 Setting Up Your Local Environment

Ensure that you have conda installed. Unzip the starter code, and open the resulting directory in your favorite python integrated development environment (Visual Studio Code is a popular choice). Open a terminal and navigate (cd) to the directory that contains the env-cpu.yml and env-gpu.yml files. Create and activate the cs6180-cpu conda environment as follows:

```
conda env create --file env-cpu.yml
conda activate cs6180-cpu
```

1.2 Setting Up Your Cloud GPU Virtual Machine

BIG REMINDER: Make sure you stop your instance once you are done running your code!

For this class, you will each receive a $50 GCP coupon to use Google Compute Engine for developing and testing your implementations. When you first sign up on GCP, you will have $300 free credits. You can receive these credits by using an account that has not previously been used for GCP.

Please follow the steps in this guide to set up your Google Cloud account. When you get to "Claim CS224n GCP Credits", wou will need to use the following URL to access the Google Cloud coupon. You will be asked to provide your school email address and name. An email will be sent to you to confirm these details before a coupon is sent to you. You will be asked for a name and email address, which needs to match your school domain. A confirmation email will be sent to you with a coupon code. You can only request ONE code per unique email address. Please contact me if you have any questions or issues.

This process should take you approximately 45 minutes. Though you will need the GPU to train your model, we strongly advise that you first develop the code locally and ensure that it runs, before attempting to train it on your VM. GPU time is expensive and limited. It takes approximately 1.5 to 2 hours to train the NMT system. We don't want you to accidentally use all your GPU time for debugging your model rather than training and evaluating it. Finally, make sure that your VM is turned off whenever you are not using it. If your GCP subscription runs out of money, your VM will be temporarily locked and inaccessible. If that happens, please reach out to me describing the situation. After setting up the cloud VM, you should turn it off while you work on the programming assignment locally.

2 Neural Machine Translation with RNNs

In Machine Translation, we seek to generate a translation of a sentence from a source language to another target language. In this assignment, we will implement a sequence-to-sequence network with attention, to build a Neural Machine Translation (NMT) system which will convert sentences in Mandarin to English. This NMT model consists of a bidirectional LSTM Encoder and a unidirectional LSTM Decoder.

Note: I usually put an arrow above a variable to indicate that it is a vector. In this assignment only, I am using the arrows to distinguish between the forward and backward LSTM. To help you figure out which variables are vectors vs scalars vs matrices, I am explicitly writing the dimension of each variable.

2.1 Word Embeddings

We consider a sentence composed of m characters in Mandarin which can be represented by one-hot word vectors $x_1, x_2, ..., x_m$ such that $x_i \in \mathbb{R}^{V_s \times 1}$ where V_s is the size of the source language vocabulary. You are given a source embedding matrix $E_s \in \mathbb{R}^{d \times V_s}$ to convert the one-hot vectors into $d-$dimensional dense vectors $e_1, e_2, ..., e_m$, where $e_i = E_s x_i \in \mathbb{R}^{d \times 1}$. A different target embedding matrix E_t will be used when converting the output of the model to a vector in $\mathbb{R}^{V_t \times 1}$ where V_t is the size of the target language vocabulary.

1. The sentences that we have in our corpus may not all be of the same length. In order to apply tensor (higher dimensional matrices) operations in the model, we must ensure that the sentences in a given batch are of the same length. Thus, we must identify the longest sentence in a batch and pad others to be the same length. Implement the *pad_sents* function in *utils.py*, which shall produce these padded sentences.

2. Implement the *__init__* function in *model_embeddings.py* to initialize the:

   - source embedding.
   - target embedding.

   Detailed instructions are provided in the starter code to help you implement each component.

2.2 Bidirectional LSTM Encoder

The Bidirectional LSTM Encoder consists of a couple of components.

First, we feed the embeddings to a convolutional layer. A convolutional layer is a way for a neural network to automatically detect useful local patterns in data. You can think of it as a small sliding window that moves across a sequence (like a sentence). At each position, it looks at a few neighboring inputs together (for example, 2–3 adjacent embeddings) and produces a new feature that summarizes that local context. This allows the model to capture short-range relationships without having to memorize them explicitly. By stacking these features, the network gains richer representations of the input before passing them into the encoder. This is especially useful for Mandarin Chinese, where each character can be either a standalone word or a morpheme in a larger word. For example, 电 = "electricity", 脑 = "brain", but 电脑 = "computer".

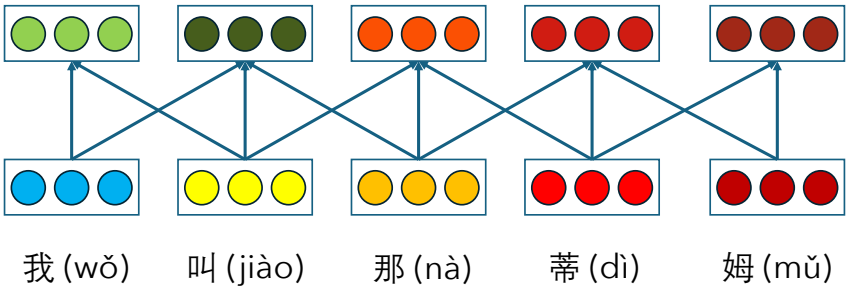

Figure 1: Convolutional layer. Notice how the outputs contain information from different characters which is useful given that my name requires three characters for example.

We denote the outputs of the convolutional layer $\tilde{e}_1, \tilde{e}_2, ..., \tilde{e}_m$. The shapes of the embeddings will remain the same, that is $\tilde{e}_i \in \mathbb{R}^{d \times 1}$.

Second, we feed the vectors $\tilde{e}_1, \tilde{e}_2, ..., \tilde{e}_m$ to the bidirectional encoder yielding hidden and cell states for both the forward ($\rightarrow$) and backward ($\leftarrow$) LSTMs.

Recall that one step of the forward LSTM is given by,

$$
\begin{aligned}
&\text{(Forget gate) } \vec{f}_t = \sigma(\vec{W}_f\ \vec{h}_{t-1} + \vec{U}_f\ \tilde{e}_t + \vec{b}_f),\\
&\text{(Input gate) } \vec{i}_t = \sigma(\vec{W}_i\ \vec{h}_{t-1} + \vec{U}_i\ \tilde{e}_t + \vec{b}_i),\\
&\text{(Output gate) } \vec{o}_t = \sigma(\vec{W}_o\ \vec{h}_{t-1} + \vec{U}_o\ \tilde{e}_t + \vec{b}_o),\\
&\text{(New cell content) } \vec{\tilde{c}}_t = \tanh\left(\vec{W}_{\tilde{c}}\ \vec{h}_{t-1} + \vec{U}_{\tilde{c}}\ \tilde{e}_t + \vec{b}_{\tilde{c}}\right),\\
&\text{(Cell State) } \vec{c}_t = \vec{f}_t \star \vec{c}_{t-1} + \vec{i}_t \star \vec{\tilde{c}}_t,\\
&\text{(Hidden State) } \vec{h}_t = \vec{o}_t \star \tanh\left(\vec{c}_t\right).
\end{aligned}
$$

Note that $\vec{h}_t \in \mathbb{R}^{k \times 1}$ and $\vec{c}_t \in \mathbb{R}^{k \times 1}$, where k is the number of hidden units.

Similarly, one step of the backward LSTM reads,

$$
\begin{aligned}
&\text{(Forget gate) } \overleftarrow{f}_t = \sigma(\overleftarrow{W}_f\ \overleftarrow{h}_{t-1} + \overleftarrow{U}_f\ \tilde{e}_t + \overleftarrow{b}_f),\\
&\text{(Input gate) } \overleftarrow{i}_t = \sigma(\overleftarrow{W}_i\ \overleftarrow{h}_{t-1} + \overleftarrow{U}_i\ \tilde{e}_t + \overleftarrow{b}_i),\\
&\text{(Output gate) } \overleftarrow{o}_t = \sigma(\overleftarrow{W}_o\ \overleftarrow{h}_{t-1} + \overleftarrow{U}_o\ \tilde{e}_t + \overleftarrow{b}_o),\\
&\text{(New cell content) } \overleftarrow{\tilde{c}}_t = \tanh\left(\overleftarrow{W}_{\tilde{c}}\ \overleftarrow{h}_{t-1} + \overleftarrow{U}_{\tilde{c}}\ \tilde{e}_t + \overleftarrow{b}_{\tilde{c}}\right),\\
&\text{(Cell State) } \overleftarrow{c}_t = \overleftarrow{f}_t \star \overleftarrow{c}_{t-1} + \overleftarrow{i}_t \star \overleftarrow{\tilde{c}}_t,\\
&\text{(Hidden State) } \overleftarrow{h}_t = \overleftarrow{o}_t \star \tanh\left(\overleftarrow{c}_t\right)
\end{aligned}
$$

We also have $\overleftarrow{h}_t \in \mathbb{R}^{k \times 1}$ and $\overleftarrow{c}_t \in \mathbb{R}^{k \times 1}$.

The hidden states, $\vec{h}_t$ and $\overleftarrow{h}_t$, as well as the cell states, $\vec{c}_t$ and $\overleftarrow{c}_t$, are concatenated resulting in,

$$
h_t^{enc} = \begin{bmatrix} \overleftarrow{h}_t \\ \vec{h}_t \end{bmatrix} \text{ and } c_t^{enc} = \begin{bmatrix} \overleftarrow{c}_t \\ \vec{c}_t \end{bmatrix},
$$

where $h_t \in \mathbb{R}^{2k \times 1}$ and $c_t \in \mathbb{R}^{2k \times 1}$. Both h_t and c_t will be used to initialize the decoder's first hidden and cell states by linearly projecting them back to the space $\mathbb{R}^{k \times 1}$ as follows,

$$
h_0^{dec} = P_h \begin{bmatrix} \overleftarrow{h}_1 \\ \vec{h}_m \end{bmatrix} \text{ and } c_0^{dec} = P_c \begin{bmatrix} \overleftarrow{c}_1 \\ \vec{c}_m \end{bmatrix},
$$

where $P_h \in \mathbb{R}^{k \times 2k}$ and $P_c \in \mathbb{R}^{k \times 2k}$ are projection matrices. Note that the final hidden and cell states of the forward LSTM are $\vec{h}_m$ and $\vec{c}_m$ whilst the the final hidden and cell states of the backward LSTM are $\overleftarrow{h}_1$ and $\overleftarrow{c}_1$.

1. Implement the *__init__* function in *nmt_model.py* to initialize the:

- convolutional layer.
- bidirectional LSTM encoder.
- P_h.
- P_c.

Detailed instructions are provided in the starter code to help you implement the model layers for the NMT system. Note that the starter code also describes variables in the decoder part of the model which you will be implementing in the next section.

2. Implement the encode function in *nmt_model.py*. This function converts the padded source sentences into the tensor X, generates $h_1^{enc}, h_2^{enc}, ..., h_m^{enc}$, and computes the initial hidden and cell states h_0^{dec} and c_0^{dec} for the Decoder.

Run a sanity check to make sure that your encode function is running correctly by executing:

```
python sanity_check.py 1d
```

2.3 Unidirectional LSTM Decoder

We are given the inputs h_0^{dec} and c_0^{dec} to the decoder. We denote one update step of the Unidirectional LSTM Decoder as,

$$h_t^{dec}, c_t^{dec} = \text{Decoder}(h_{t-1}^{dec}, c_{t-1}^{dec}),$$

to avoid rewriting all the equations involved in a LSTM.

We incorporate attention to this decoder. We first need to compute the attention scores according to the product of the current decoder hidden state with every encoder hidden state. Note that $h_t^{dec} \in \mathbb{R}^{k \times 1}$ whilst $h_t^{enc} \in \mathbb{R}^{2k \times 1}$ as it includes the hidden states from both the forward and backward LSTMs. To compute a valid dot product, we project the encoder hidden states to $\mathbb{R}^{k \times 1}$ using a projection matrix $P_{att} \in \mathbb{R}^{k \times 2k}$ so that $P_{att} h_t^{enc} \in \mathbb{R}^{k \times 1}$. We can now compute the attention scores s as,

$$s_{t,i} = (h_t^{dec})^T P_{att} h_i^{enc},$$

for every $i = 1, 2, ..., m$. We can store the attention scores in a vector s_t where its i^{th} component is $s_{t,i}$.

Recall that we want the attention scores to represent weights between 0 and 1. We use the softmax function and define the scaled attention scores α as,

$$\alpha_{t,i} = \frac{\exp\left((h_t^{dec})^T P_{att} h_i^{enc}\right)}{\sum_{j=1}^{m} \exp\left((h_t^{dec})^T P_{att} h_j^{enc}\right)}.$$

Similarly, we store the scaled attention scores in a vector α_t where its i^{th} component is $\alpha_{t,i}$.

We compute the output attention vector a_t as,

$$a_t = \sum_{i=1}^{m} \alpha_{t,i} h_i^{enc}.$$

We concatenate the output attention vector a_t with the current decoder hidden state h_t^{dec} to include the attention information in the decoder so that,

$$(dec, att)_t = \begin{bmatrix} a_t \\ h_t^{dec}. \end{bmatrix}$$

The combined output $(dec, att)_t$ is in $\mathbb{R}^{3k \times 1}$ so we project it back to $\mathbb{R}^{k \times 1}$ using a projection matrix $P_{comb,out}$.

Given that $P_{comb,out} \cdot (dec, att)_t$ is obtained using a linear projection, we apply an activation function (tanh), i.e. $\tanh\left(P_{comb,out} \cdot (dec, att)_t\right)$ so that the model can still learn nonlinear combinations of decoder state and attention. Finally, during the training process of the model, we apply a dropout layer, which we looked at in HW1, to avoid overfitting as Neural Machine Translation models are very large and prone to memorization.

We obtain the combined output,

$$O_t = \text{dropout}\Big(\tanh\left(P_{comb,out} \cdot (dec, att)_t\right)\Big).$$

The final step is to produce a probability distribution p_t over the target subwords using the softmax function,

$$p_t = \text{softmax}(P_{vocab} \cdot O_t),$$

where $P_{vocab} \in \mathbb{R}^{V_t \times k}$ is a projection matrix that allows us to obtain a probability vector over the target language vocabulary.

We train the network by minimizing the cross entropy loss between p_t and y_t, the one-hot vector of the target word at step t.

The structure above is a general structure of a LSTM Decoder. In this problem, we will make one slight modification:

- To give the decoder direct access to both the target-side context and the previous attention/context signal, we concatenate e_t with the output O_{t-1}, which carries information about how the decoder attended to the source at the last step and the hidden state summary from that step. Let $\bar{e}_t = \begin{bmatrix} e_t \\ O_{t-1} \end{bmatrix}$. We then have,
$$h_t^{dec}, c_t^{dec} = \text{Decoder}(h_{t-1}^{dec}, c_{t-1}^{dec}, \bar{e}_t).$$

This whole model can be summarized in the figure below.

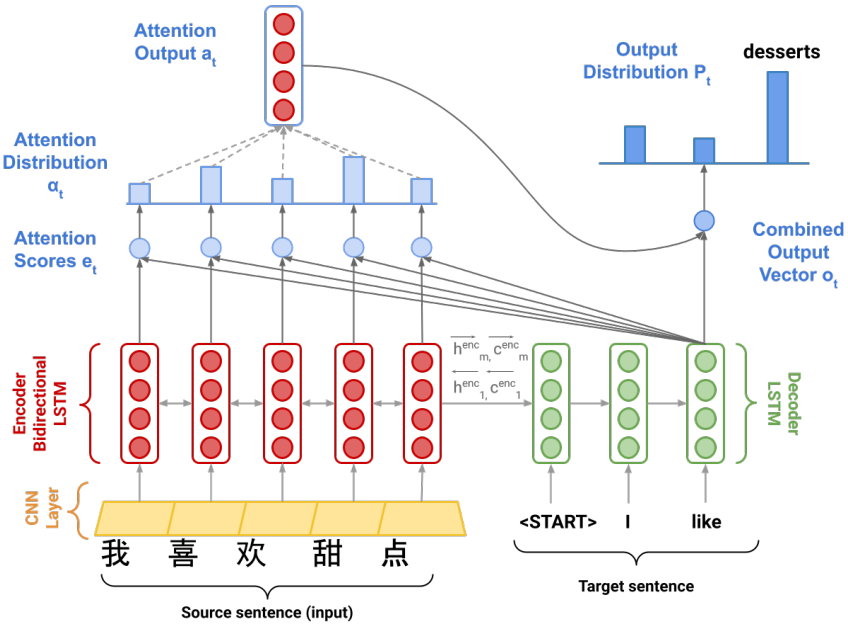

Figure 2: NMT Model. The attention scores are denoted by s_t (instead of e_t).

1. Finish implementing the *__init__* function in *nmt_model.py* by initializing:

   - unidirectional LSTM decoder.
   - P_{att}.
   - $P_{comb,out}$.
   - P_{vocab}.

   Detailed instructions are provided in the starter code to help you implement the model layers for the NMT system.

2. Implement the decode function in *nmt_model.py*. This function constructs $\bar{e}_t$ and runs the step function immediately below the decode function) over every timestep for the input.
   Detailed instructions are provided in the starter code to help you implement the decode function for the NMT system.

   Run a sanity check to make sure that your encode function is running correctly by executing:

   ```
   python sanity_check.py 1e
   ```

3. Implement the step function in *nmt_model.py*. This function applies the Decoder's LSTM cell for a single timestep, computes the encoding of the target sub-word h_t^{dec}, the attention scores vector s_t, the scaled attention vector α_t, the attention output a_t, and finally the combined output O_t.

   Run a sanity check to make sure that your encode function is running correctly by executing:

   ```
   python sanity_check.py 1f
   ```

It's finally time to get things running! Now it's time to get things running! As noted earlier, we recommend that you develop the code on your personal computer. Confirm that you are running in the proper conda environment and then execute the following command to train the model on your local machine:

```
sh run.sh train_local
```

or

```
run.bat train_local
```

for Windows machines.

The starter code uses tensorboard to monitor the log loss and perplexity during training using TensorBoard3 to help with debugging. TensorBoard provides tools for logging and visualizing training information from experiments. To open TensorBoard and monitor the training process, run the following in a separate terminal in which you have also activated the cs6180-cpu conda environment, then access tensorboard at this link.

```
tensorboard --logdir = runs --port 6006
```

You should see a significant decrease in loss during the initial iterations. Once your code runs for a few hundreds of iterations without crashing, power on your VM from the GCP Console.

Train the NMT model the VM, please refer to the How-to Appending in the guide to:

- Setup your GCP VM and Obtain SSH Connection Info.
- Connect to the VM (with SSH Tunneling setup so that you can view remote tensorboard logs).
- Copy your code from your computer to the cloud VM.
- Train the NMT System on the Cloud VM.
- Download the Gradescope Submission Package from Your Cloud VM.

2.4 After Training

1. Once your model is done training (this should take under 2 hours on the VM), execute the following command to test the model:

   ```
   sh run.sh test
   ```

   Please report the model's corpus BLEU Score. It should be larger than 18.