CSCI 4800: Assignment #4

1. Neural Machine Translation with RNNs (100 points)

In Machine Translation, our goal is to convert a sentence from the *source* language (e.g. Mandarin Chinese) to the *target* language (e.g. English). In this assignment, we will implement a sequence-to-sequence (Seq2Seq) network with attention, to build a Neural Machine Translation (NMT) system. In this section, we describe the **training procedure** for the proposed NMT system, which uses a Bidirectional LSTM Encoder and a Unidirectional LSTM Decoder.

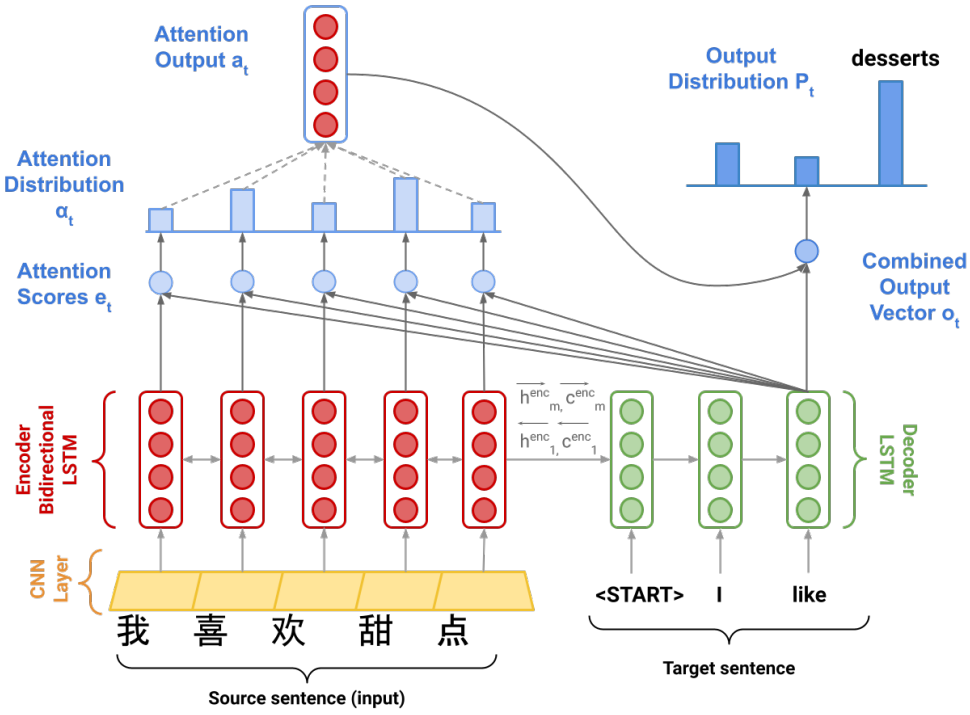

Figure 1: Seq2Seq Model with Multiplicative Attention, shown on the third step of the decoder. Hidden states $\mathbf{h}_i^{\text{enc}}$ and cell states $\mathbf{c}_i^{\text{enc}}$ are defined on the next page.

Model description (training procedure)

Given a sentence in the source language, we look up the character or word embeddings from an **embeddings matrix**, yielding $\mathbf{x}_1, \ldots, \mathbf{x}_m$ ($\mathbf{x}_i \in \mathbb{R}^{e \times 1}$), where m is the length of the source sentence and e is the embedding size. We then feed the embeddings to a **convolutional layer** [1] while maintaining their shapes. We feed the convolutional layer outputs to the **bidirectional encoder**, yielding hidden states and cell states for both the forwards ($\rightarrow$) and backwards ($\leftarrow$) LSTMs. The forwards and backwards versions are concatenated to give hidden states $\mathbf{h}_i^{\text{enc}}$ and cell states $\mathbf{c}_i^{\text{enc}}$:

$$\mathbf{h}_i^{\text{enc}} = [\overleftarrow{\mathbf{h}_i^{\text{enc}}}; \overrightarrow{\mathbf{h}_i^{\text{enc}}}] \quad \text{where} \quad \mathbf{h}_i^{\text{enc}} \in \mathbb{R}^{2h \times 1}, \overleftarrow{\mathbf{h}_i^{\text{enc}}}, \overrightarrow{\mathbf{h}_i^{\text{enc}}} \in \mathbb{R}^{h \times 1} \qquad 1 \leq i \leq m \tag{1}$$

$$\mathbf{c}_i^{\text{enc}} = [\overleftarrow{\mathbf{c}_i^{\text{enc}}}; \overrightarrow{\mathbf{c}_i^{\text{enc}}}] \quad \text{where} \quad \mathbf{c}_i^{\text{enc}} \in \mathbb{R}^{2h \times 1}, \overleftarrow{\mathbf{c}_i^{\text{enc}}}, \overrightarrow{\mathbf{c}_i^{\text{enc}}} \in \mathbb{R}^{h \times 1} \qquad 1 \leq i \leq m \tag{2}$$

[1] Checkout https://cs231n.github.io/convolutional-networks for an in-depth description for convolutional layers if you are not familiar

We then initialize the **decoder**'s first hidden state $\mathbf{h}_0^{\text{dec}}$ and cell state $\mathbf{c}_0^{\text{dec}}$ with a linear projection of the encoder's final hidden state and final cell state.[2]

$$\mathbf{h}_0^{\text{dec}} = \mathbf{W}_h[\overleftarrow{\mathbf{h}_1^{\text{enc}}}; \overrightarrow{\mathbf{h}_m^{\text{enc}}}] \text{ where } \mathbf{h}_0^{\text{dec}} \in \mathbb{R}^{h\times 1}, \mathbf{W}_h \in \mathbb{R}^{h\times 2h} \tag{3}$$

$$\mathbf{c}_0^{\text{dec}} = \mathbf{W}_c[\overleftarrow{\mathbf{c}_1^{\text{enc}}}; \overrightarrow{\mathbf{c}_m^{\text{enc}}}] \text{ where } \mathbf{c}_0^{\text{dec}} \in \mathbb{R}^{h\times 1}, \mathbf{W}_c \in \mathbb{R}^{h\times 2h} \tag{4}$$

With the decoder initialized, we must now feed it a target sentence. On the t^{th} step, we look up the embedding for the t^{th} subword, $\mathbf{y}_t \in \mathbb{R}^{e\times 1}$. We then concatenate $\mathbf{y}_t$ with the *combined-output vector* $\mathbf{o}_{t-1} \in \mathbb{R}^{h\times 1}$ from the previous timestep (we will explain what this is later down this page!) to produce $\overline{\mathbf{y}_t} \in \mathbb{R}^{(e+h)\times 1}$. Note that for the first target subword (i.e. the start token) $\mathbf{o}_0$ is a zero-vector. We then feed $\overline{\mathbf{y}_t}$ as input to the decoder.

$$\mathbf{h}_t^{\text{dec}}, \mathbf{c}_t^{\text{dec}} = \text{Decoder}(\overline{\mathbf{y}_t}, \mathbf{h}_{t-1}^{\text{dec}}, \mathbf{c}_{t-1}^{\text{dec}}) \text{ where } \mathbf{h}_t^{\text{dec}} \in \mathbb{R}^{h\times 1}, \mathbf{c}_t^{\text{dec}} \in \mathbb{R}^{h\times 1} \tag{5}$$

$$\tag{6}$$

We then use $\mathbf{h}_t^{\text{dec}}$ to compute multiplicative attention over $\mathbf{h}_1^{\text{enc}}, \ldots, \mathbf{h}_m^{\text{enc}}$:

$$\mathbf{e}_{t,i} = (\mathbf{h}_t^{\text{dec}})^T \mathbf{W}_{\text{attProj}} \mathbf{h}_i^{\text{enc}} \text{ where } \mathbf{e}_t \in \mathbb{R}^{m\times 1}, \mathbf{W}_{\text{attProj}} \in \mathbb{R}^{h\times 2h} \qquad 1 \le i \le m \tag{7}$$

$$\alpha_t = \text{softmax}(\mathbf{e}_t) \text{ where } \alpha_t \in \mathbb{R}^{m\times 1} \tag{8}$$

$$\mathbf{a}_t = \sum_{i=1}^{m} \alpha_{t,i} \mathbf{h}_i^{\text{enc}} \text{ where } \mathbf{a}_t \in \mathbb{R}^{2h\times 1} \tag{9}$$

$\mathbf{e}_{t,i}$ is a scalar, the ith element of $\mathbf{e}_t \in \mathbb{R}^{m\times 1}$, computed using the hidden state of the decoder at the tth step, $\mathbf{h}_t^{\text{dec}} \in \mathbb{R}^{h\times 1}$, the attention projection $\mathbf{W}_{\text{attProj}} \in \mathbb{R}^{h\times 2h}$, and the hidden state of the encoder at the ith step, $\mathbf{h}_i^{\text{enc}} \in \mathbb{R}^{2h\times 1}$.

We now concatenate the attention output $\mathbf{a}_t$ with the decoder hidden state $\mathbf{h}_t^{\text{dec}}$ and pass this through a linear layer, tanh, and dropout to attain the *combined-output* vector $\mathbf{o}_t$.

$$\mathbf{u}_t = [\mathbf{a}_t; \mathbf{h}_t^{\text{dec}}] \text{ where } \mathbf{u}_t \in \mathbb{R}^{3h\times 1} \tag{10}$$

$$\mathbf{v}_t = \mathbf{W}_u \mathbf{u}_t \text{ where } \mathbf{v}_t \in \mathbb{R}^{h\times 1}, \mathbf{W}_u \in \mathbb{R}^{h\times 3h} \tag{11}$$

$$\mathbf{o}_t = \text{dropout}(\tanh(\mathbf{v}_t)) \text{ where } \mathbf{o}_t \in \mathbb{R}^{h\times 1} \tag{12}$$

Then, we produce a probability distribution $\mathbf{P}_t$ over target subwords at the t^{th} timestep:

$$\mathbf{P}_t = \text{softmax}(\mathbf{W}_{\text{vocab}} \mathbf{o}_t) \text{ where } \mathbf{P}_t \in \mathbb{R}^{V_t\times 1}, \mathbf{W}_{\text{vocab}} \in \mathbb{R}^{V_t\times h} \tag{13}$$

Here, V_t is the size of the target vocabulary. Finally, to train the network we then compute the cross entropy loss between $\mathbf{P}_t$ and $\mathbf{g}_t$, where $\mathbf{g}_t$ is the one-hot vector of the target subword at timestep t:

$$J_t(\theta) = \text{CrossEntropy}(\mathbf{P}_t, \mathbf{g}_t) \tag{14}$$

Here, θ represents all the parameters of the model and $J_t(\theta)$ is the loss on step t of the decoder. Now that we have described the model, let's try implementing it for Mandarin Chinese to English translation!

[2] If it's not obvious, think about why we regard $[\overleftarrow{\mathbf{h}_1^{\text{enc}}}, \overrightarrow{\mathbf{h}_m^{\text{enc}}}]$ as the 'final hidden state' of the Encoder.

0.1 Instructions for Running the Assignment in Google Colab

0.1.1 1. Upload the Code Skeleton to Your Google Drive

1. Download the code skeleton provided with the assignment.
2. Log in to your Google Drive account.
3. Create a folder named NMT_Assignment (or any name you prefer) in your Drive.
4. Upload the code skeleton and any additional files into this folder.

0.1.2 2. Open and Set Up Google Colab

1. Go to Google Colab.
2. Create a new notebook or open the notebook provided for the assignment.

0.1.3 3. Mount Your Google Drive in Colab

To access the code from your Google Drive, you need to mount it in your Colab environment:

1. Add the following code at the beginning of your Colab notebook:

```
from google.colab import drive
drive.mount('/content/drive')
```

2. Run this cell. You will be prompted to authorize access to your Google Drive. Follow the instructions to complete the authorization process.
3. Once mounted, navigate to your assignment folder in Drive. For example:

```
import os
# Replace 'YourFolderName' with the name of the folder where your code is stored
os.chdir('/content/drive/My␣Drive/NMT_Assignment')
```

4. Verify that your files are accessible by listing the directory:

```
!ls
```

0.1.4 4. Enable GPU for Training

To enable GPU in your Colab environment:

1. Click on **Runtime** in the Colab menu bar.
2. Select **Change runtime type**.
3. In the pop-up window:
   - Set **Hardware accelerator** to GPU.
   - Leave the other options as default.
4. Click **Save** to apply the changes.
5. Verify that GPU is available by running:

```
import torch
print("GPU␣available:", torch.cuda.is_available())
```

0.1.5 5. Best Practices for GPU Usage

- **Develop Locally First:** As noted earlier, develop and debug your code locally using a small dataset to ensure it runs without errors. Save GPU time for actual training.
- **Avoid Unnecessary GPU Usage:** GPU time is both expensive and limited. Double-check your code to ensure it' s bug-free before running intensive training processes.
- **Estimated Training Time:** Training the NMT system should take approximately **1.5 to 2 hours**. Plan your GPU usage accordingly.

After completing the assignment:

1. Save your modified code and any results in the appropriate format. For example:

```
# Example: Save your modified code or other files back to Drive
!cp modified_file.py '/content/drive/My␣Drive/NMT_Assignment/modified_file.py'
```

2. Download your updated code and required files to your local machine. This step is necessary to run the collect_submission.sh script:
   (a) Right-click on the files in Google Drive or use the following command to download them directly in Colab:

```
from google.colab import files
files.download('modified_file.py')
```

   (b) Verify that you have downloaded all required files specified in the assignment instructions.
3. Run the collect_submission.sh script on your local machine to package your submission.

Important Note: Double-check that your submission includes all the required files and adheres to the format specified in the assignment guidelines. Missing or incorrect files may result in loss of credit.

In order to run the model code on your **local** machine, please run the following command to create the proper virtual environment:

```
conda env create --file local_env.yml
```

Note that this virtual environment **will not** be needed in Google Colab.

Implementation and written questions

(a) (10 points) (coding) In order to apply tensor operations, 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.

(b) (10 points) (coding) Implement the ___init___ function in model_embeddings.py to initialize the necessary source and target embeddings.

(c) (10 points) (coding) Implement the ___init___ function in nmt_model.py to initialize the necessary model layers (LSTM, CNN, projection, and dropout) for the NMT system.

(d) (10 points) (coding) Implement the encode function in nmt_model.py. This function converts the padded source sentences into the tensor $\mathbf{X}$, generates $\mathbf{h}_1^{\text{enc}}, \ldots, \mathbf{h}_m^{\text{enc}}$, and computes the initial state $\mathbf{h}_0^{\text{dec}}$ and initial cell $\mathbf{c}_0^{\text{dec}}$ for the Decoder. You can run a non-comprehensive sanity check by executing:

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

(e) (10 points) (coding) Implement the decode function in nmt_model.py. This function constructs $\bar{\mathbf{y}}$ and runs the step function over every timestep for the input. You can run a non-comprehensive sanity check by executing:

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

(f) (20 points) (coding) Implement the step function in nmt_model.py. This function applies the Decoder's LSTM cell for a single timestep, computing the encoding of the target subword $\mathbf{h}_t^{\text{dec}}$, the attention scores $\mathbf{e}_t$, attention distribution α_t, the attention output $\mathbf{a}_t$, and finally the combined output $\mathbf{o}_t$. You can run a non-comprehensive sanity check by executing:

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

(g) (10 points) (written) The generate_sent_masks() function in nmt_model.py produces a tensor called enc_masks. It has shape (batch size, max source sentence length) and contains 1s in positions corresponding to 'pad' tokens in the input, and 0s for non-pad tokens. Look at how the masks are used during the attention computation in the step() function (lines 311-312).

First explain (in around three sentences) what effect the masks have on the entire attention computation. Then explain (in one or two sentences) why it is necessary to use the masks in this way.

First of all, if the enc_masks is equal to 1 then the value is set to -infinity for padded tokens, while the non-padded tokens are 0. This results in the padded tokens being insignificant to the computation. That is, the corresponding alpha_t from SoftMax will disfavor the padded tokens from being adjusted by setting them near 0, and by setting to -infinity it does more than just disfavoring, and essentially just disregards them from the computation entirely.

Solution:

It is essential to include the masks for a couple of reasons, namely dimensionality and effects on the computation. We can't just ignore padding because then we would have dimensionality problems where some vectors are larger than others, which doesn't work in calculations. Similarly, if we were to incorporate the padded tokens in the computation then it would result in an incorrect model where padded tokens are favored, when in reality they are insignificant.

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
(Windows) run.bat train_local
```

For a faster way to debug by training on less data, you can run the following instead:

```
sh run.sh train_debug
(Windows) run.bat debug
```

To help with monitoring and debugging, the starter code uses tensorboard to log loss and perplexity during training using TensorBoard[3]. TensorBoard provides tools for logging and visualizing training information from experiments. To open TensorBoard, run the following in your conda environment:

```
tensorboard --logdir=runs
```

You should see a significant decrease in loss during the initial iterations. Once you have ensured that your code does not crash (i.e. let it run till iter 10 or iter 20), switch to Google Colab.

Next, install necessary packages by running:

```
!pip install -r gpu_requirements.txt
```

To execute the training script, run:

```
!sh run.sh train
```

[3] https://pytorch.org/docs/stable/tensorboard.html

(h) (10 points) (written) Once your model is done training (**this should take under 2 hours**), 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.

Solution: 19.54830952420219

(i) (10 points) (written) In class, we learned about dot product attention, multiplicative attention, and additive attention. As a reminder, dot product attention is $\mathbf{e}_{t,i} = \mathbf{s}_t^T \mathbf{h}_i$, multiplicative attention is $\mathbf{e}_{t,i} = \mathbf{s}_t^T \mathbf{W} \mathbf{h}_i$, and additive attention is $\mathbf{e}_{t,i} = \mathbf{v}^T \tanh(\mathbf{W}_1 \mathbf{h}_i + \mathbf{W}_2 \mathbf{s}_t)$.

i. (5 points) Explain one advantage and one disadvantage of *dot product attention* compared to multiplicative attention.

ii. (5 points) Explain one advantage and one disadvantage of *additive attention* compared to multiplicative attention.

Solution: One advantage of dot product over multiplicative attention is that it doesn't need as much data as there are less parameters to train; one disadvantage of dot product over multiplicative attention is that the dot product is less accurate than multiplicative attention with sufficient data size.

One advantage of additive attention over multiplicative attention is that the similarity is computed non-linearly. One disadvantage of additive attention over multiplicative attention is that it is more expensive to compute by essentially adding another layer in the network.

Submission Instructions

You shall submit this assignment on Canvas as two submissions – one for "Assignment 4 [coding]" and another for 'Assignment 4 [written]":

1. Run the collect_submission.sh script to produce your assignment4.zip file.
2. Upload your assignment4.zip file to Canvas to "Assignment 4 [coding]".
3. Upload your written solutions to Canvas to "Assignment 4 [written]".