Assignment 3: Neural Machine Translation with RNNs

Due on the date of the final exam at 11:59 pm

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

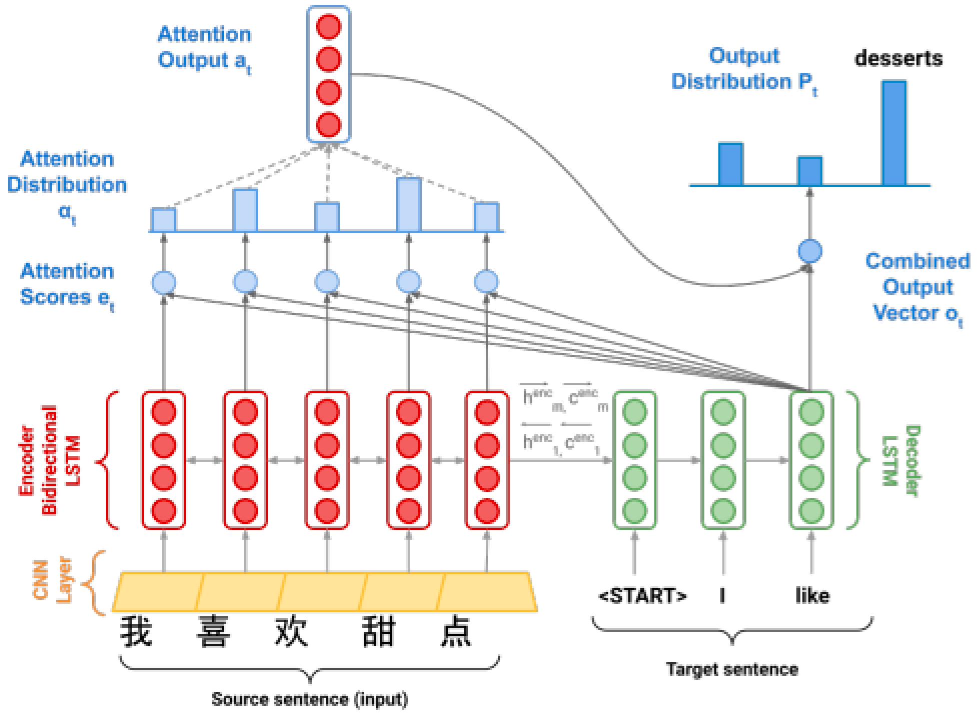

Figure 1: Seq2Seq Model with Multiplicative Attention, shown on the third step of the decoder. Hidden states $\mathbf{h}_i^{\text{enc}}$ and cell states $\mathbf{c}_i^{\text{enc}}$ are defined on the next page.

Model description (training procedure) Given a sentence in the source language, we look up the character or word embeddings from an embeddings matrix, yielding x1, . . . , xm (xi ∈ $R^{e\times 1}$), 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 (→) and backwards (←) LSTMs. The forwards and backwards versions are concatenated to give hidden states h_i^{enc} and cell states c_i^{enc}:

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

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

We then initialize the decoder's first hidden state h_0^{dec} and cell state c_0^{dec} with a linear projection of the encoder's final hidden state and final cell state.

$$\mathbf{h}_0^{\text{dec}} = \mathbf{W}_h[\overleftarrow{\mathbf{h}_1^{\text{enc}}}; \overrightarrow{\mathbf{h}_m^{\text{enc}}}] \text{ where } \mathbf{h}_0^{\text{dec}} \in \mathbb{R}^{h \times 1}, \mathbf{W}_h \in \mathbb{R}^{h \times 2h} \tag{3}$$
$$\mathbf{c}_0^{\text{dec}} = \mathbf{W}_c[\overleftarrow{\mathbf{c}_1^{\text{enc}}}; \overrightarrow{\mathbf{c}_m^{\text{enc}}}] \text{ where } \mathbf{c}_0^{\text{dec}} \in \mathbb{R}^{h \times 1}, \mathbf{W}_c \in \mathbb{R}^{h \times 2h} \tag{4}$$

With the decoder initialized, we must now feed it a target sentence. On the tth step, we look up the embedding for the tth subword, $y_t \in R^{e\times 1}$. We then concatenate y_t with the *combined-output vector* $o_{t-1} \in R^{h\times 1}$ from the previous timestep to produce $y_t \in R^{(e+h)\times 1}$. Note that for the first target subword (i.e. the start token) o_0 is a zero-vector. We then feed y_t as input to the decoder.

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

We then use h dec t to compute multiplicative attention over $h_1^{enc}, \ldots, h_m^{enc}$:

$$\mathbf{e}_{t,i} = (\mathbf{h}_t^{\text{dec}})^T \mathbf{W}_{\text{attProj}} \mathbf{h}_i^{\text{enc}} \text{ where } \mathbf{e}_t \in \mathbb{R}^{m \times 1}, \mathbf{W}_{\text{attProj}} \in \mathbb{R}^{h \times 2h} \qquad 1 \le i \le m \tag{7}$$
$$\alpha_t = \text{softmax}(\mathbf{e}_t) \text{ where } \alpha_t \in \mathbb{R}^{m \times 1} \tag{8}$$
$$\mathbf{a}_t = \sum_{i=1}^{m} \alpha_{t,i} \mathbf{h}_i^{\text{enc}} \text{ where } \mathbf{a}_t \in \mathbb{R}^{2h \times 1} \tag{9}$$

$e_{t,i}$ is a scalar, the ith element of $e_t \in R^{m\times 1}$, computed using the hidden state of the decoder at the tth step, $h_t^{dec} \in R^{h\times 1}$, the attention projection $W_{attProj} \in R^{h\times 2h}$, and the hidden state of the encoder at the ith step, $h_i^{enc} \in R^{2h\times 1}$.

We now concatenate the attention output at with the decoder hidden state h_t^{dec} and pass this through a linear layer, tanh, and dropout to attain the *combined-output vector* 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 P_t over target subwords at the tth 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 P_t and g_t, where 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.

(a) 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) Implement the __init__ function in model_embeddings.py to initialize the necessary source and target embeddings.

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

(d) Implement the encode function in nmt_model.py. This function converts the padded source sentences into the tensor X, generates h_1^{enc}, . . . , h_m^{enc} , and computes the initial state h_0^{dec} and initial cell c_0^{dec} for the Decoder. You can run a non-comprehensive sanity check by executing:

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

(e) Implement the decode function in nmt_model.py. This function constructs 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) 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 h_t^{dec}, the attention scores e_t, attention distribution α_t, the attention output a_t, and finally the combined output o_t. You can run a non-comprehensive sanity check by executing:

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

(g) 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 (https://pytorch.org/docs/stable/tensorboard.html) . 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), run it on GPU.

(h) Once your model is done training, execute the following command to test the model:

```
sh run.sh test (Windows)
```

```
run.bat test
```

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

Note: Note that the NMT system is more complicated than the neural networks we have previously constructed within this class and takes about 2 hours to train on a GPU. Thus, we strongly recommend you get started early with this assignment. 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.

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