Junyu Yin G2101985L

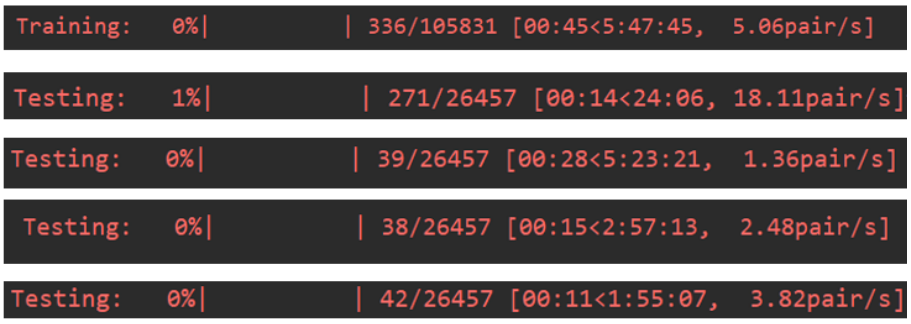
Notice

In order to shorten the training time, I set a length threshold here to filter some too long sentences in the datasets. The dataset is filtered to retain about 90% of the data. The table below shows the details.

	CS-EN	DE-EN	FR-EN	RU-EN
max length / # of sentence	113 / 146549	195 / 201288	223 / 183251	163 / 165602
length threshold / # of sentence	40 / 132288	42 / 179388	46 / 163590	50 / 147995

Table 1. This table shows the longest sentence length and data quantity in the original datasets and the datasets after setting different length thresholds.

On the above dataset, I tried several model training and testing, and found that the time required is still too long. The running screenshots shown below demonstrate this.



```
Training:   0%|          | 336/105831 [00:45<5:47:45,  5.06pair/s]
Testing:   1%|          | 271/26457 [00:14<24:06, 18.11pair/s]
Testing:   0%|          | 39/26457 [00:28<5:23:21,  1.36pair/s]
Testing:   0%|          | 38/26457 [00:15<2:57:13,  2.48pair/s]
Testing:   0%|          | 42/26457 [00:11<1:55:07,  3.82pair/s]
```

Figure 1. The estimated training time and the estimated testing time of greedy decoding and beam search decoding with beam size 5/10/20 respectively. It is worth noting that this is just for 1 of all 5 cross validations for 1 of all 4 parallel datasets.

A complete experiment consists of doing a 5-fold CV for all 4 datasets. Based on the above screenshots, I have estimated the time required to finish a complete experiment and show it in the table below.

	Greedy	Beam Search (5/10/20)
Estimated Trianing Time	> 100 hours	
Estimated Testing Time	> 10 hours	> (100/60/40) hours

Table 2. The estimated time required to finish a complete experiment.

For this assignment, at least 5 complete experiments need to be finished. If using the above settings, this assignment cannot be finished even to the end of the semester. With this in mind, I decide to randomly select 20% of the data for training in each CV. And when testing with the beam search decoding, I also randomly select 20% of the data for testing. The new estimated time required to finish a complete experiment is shown in the table below.

	Greedy	Beam Search (10)
New Estimated Trianing Time	> 20 hours	
New Estimated Testing Time	> 10 hours	> 12 hours

Table 3. The new estimated time required to finish a complete experiment.

In fact, the model or the training method used by this assignment does not fully exploit the potential of all the training data. The following training curve proves it. It can be seen from this figure that the training loss only has a significant downward trend in the early training period and keeps oscillating after that. Therefore, randomly selecting 20% of the data for training will not cause much decrease in the model performance. It is reasonable to use this method to reduce training time.

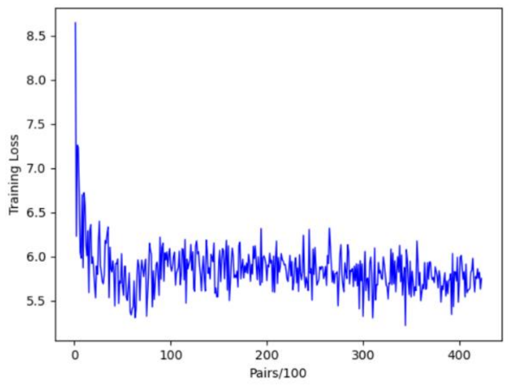


Figure 2. The training curve for 1 of all 5 cross validations for 1 of all 4 parallel datasets.

Since the beam search decoding method takes long time for testing, for problem b and c in this assignment, I choose to use the greedy decoding method. <u>In order for the code to run correctly, use the directory structure shown in the figure below.</u>

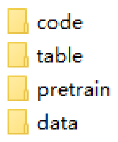


Figure 3. The directory structure of this project. Code and data need to be placed in the code and data folders respectively. The pretrain folder contains weight matrices files pretrained with glove. The test results will be placed in the table folder automatically.

a.

The experimental results are shown in the table below. Since both training and testing are extremely time-consuming, I only tried one set of hypermeters for the beam search decoding method.

	BLEU-1	BLEU-2	BLEU-3
CS-EN	0.1949	0.0544	0.0127
DE-EN	0.1852	0.0580	0.0162
FR-EN	0.1967	0.0513	0.0111
RU-EN	0.1873	0.0556	0.0146
Average	0.1910	0.0548	0.0137

Table 4. The test results with the greedy decoding method.

	BLEU-1	BLEU-2	BLEU-3
CS-EN	0.1361	0.0356	0.0050
DE-EN	0.1449	0.0411	0.0079
FR-EN	0.1588	0.0464	0.0094
RU-EN	0.1201	0.0302	0.0052
Average	0.1400	0.0383	0.0069

Table 5. The test results with the beam search decoding method.

I used qsize = 2000, beam size = 10, with length normalization and w/o max_length. From above tables, it can be seen that the beam search decoding method performs worse than the greedy decoding method. As for reasons, I think it is due to the beam search algorithm used here is different from which in the lecture. From my perspective, I'd rather say it is wrong. Suppose beam size = k here.

> Correct version: At each iteration, extend the current k nodes to k^2 hypotheses and choose k nodes with highest scores of these k^2 hypotheses. So the generated sentence will have one more word added each iteration.
>
> Tutorial version: At each iteration, extend the best node to k hypotheses and insert them all into a heap. So the heap size will increase $k-1$ at each iteration. Since there is a max size for the heap, here the bigger beam size leads to the faster testing ironically. And the length of the generated sentence may remain the same after many iterations.

So using the tutorial version of beam search will cause the generated sentences to be too short to perform well when testing.

b.

For this question, I use the official Glove implementation[1] to pre-train the word embeddings. This implementation is written in pure C and can only run under the Linux environment. Since the file size of the pretrained weighted matrices is close to 2GB, they are not included in the submitted files. As an alternative, I submit the shell file (demo.sh) needed to run the Glove implementation. You can use the command line (./demo.sh) under Linux to get the required pretrained weight matrices, then put them in the folder named pretrain in the project directory

[1] https://github.com/stanfordnlp/GloVe

to make the code run correctly.

The following table shows the test results. The results here are slightly better than which of Question a. It is reasonable since the pre-trained word embeddings contain more semantic information than randomly initialized ones.

	BLEU-1	BLEU-2	BLEU-3
CS-EN	0.2129	0.0616	0.0152
DE-EN	0.2010	0.0559	0.0120
FR-EN	0.1947	0.0512	0.0096
RU-EN	0.1962	0.0566	0.0133
Average	0.2012	0.0563	0.0125

Table 6. The test results with the pre-trained word embeddings.

c.

i.

Suppose we have encoder hidden states in $H = [\boldsymbol{h}_1, \cdots, \boldsymbol{h}_N] \in \mathbb{R}^{h \times N}$, encoder outputs in $O = [\boldsymbol{o}_1, \cdots, \boldsymbol{o}_N] \in \mathbb{R}^{h \times N}$. On timestep t, we have the decoder hidden state $\boldsymbol{s}_t \in \mathbb{R}^h$ and the embedded input $\boldsymbol{y}_t \in \mathbb{R}^h$. We abbreviate the Attention Decoder of Tutorial as ADT and the Attention Decoder of lecture as ADL.

At the beginning of timestep t, we only get the previous hidden state $\boldsymbol{s}_{t-1}$ and $\boldsymbol{y}_t$. So in the first step, we need to initialize the current state $\boldsymbol{s}_t$ with these two variables (the initial $\boldsymbol{s}_t$ does not have to be in $\mathbb{R}^h$). At this step, ADT simply concatenates $\boldsymbol{s}_{t-1}$ and $\boldsymbol{y}_t$ as the initial $\boldsymbol{s}_t$, while it is not clearly stated in the lecture how to initialize $\boldsymbol{s}_t$. In addition to concatenating, I think ADL can use $\boldsymbol{s}_{t-1}$ as the initial $\boldsymbol{s}_t$ or map from $\boldsymbol{y}_t$ and $\boldsymbol{s}_{t-1}$ to get it.

In the second step, we should get the attention scores $\boldsymbol{e}_t \in \mathbb{R}^N$ and take softmax of it to get the attention distribution $\boldsymbol{\alpha}_t = softmax(\boldsymbol{e}_t) \in \mathbb{R}^N$. At this step, ADT simply pass the initial $\boldsymbol{s}_t$ to a fc layer to get $\boldsymbol{e}_t$. It doesn't use $\boldsymbol{s}_t$ to attend to anything of the encoder. While ADL uses the initial $\boldsymbol{s}_t$ to attend to the encoder hidden states H by applying a function f_{attn}. Here the function f_{attn} can be various, such as the basic dot-product attention, the multiplicative attention, the additive attention and so on.

In the third step, we should use $\boldsymbol{\alpha}_t$ as the weights to summarize all the information of the encoder to get a context vector $\boldsymbol{c}_t$. At this step, ADT gets the context vector $\boldsymbol{c}_t$ by taking a weighted sum of the encoder hidden outputs O, that is, $\boldsymbol{c}_t = O\boldsymbol{\alpha}_t$. While ADL takes a weighted sum of the encoder hidden states H, that is, $\boldsymbol{c}_t = H\boldsymbol{\alpha}_t$.

In the last step, we pass $\boldsymbol{y}_t$, $\boldsymbol{c}_t$ and $\boldsymbol{s}_{t-1}$ to the current decode unit (we use GRU here) to get the final $\boldsymbol{s}_t$ and the current output $\boldsymbol{d}_t$. At this step, ADT first concatenates $\boldsymbol{y}_t$, $\boldsymbol{c}_t$ and passes it to a fc layer followed by a relu layer to get the current input, then passes the current input as well as $\boldsymbol{s}_{t-1}$ to the GRU to get the final $\boldsymbol{s}_t$ and $\boldsymbol{d}_t$. That is, $\boldsymbol{s}_t, \boldsymbol{d}_t = gru\left(\boldsymbol{s}_{t-1}, relu\big(fc([\boldsymbol{y}_t; \boldsymbol{c}_t])\big)\right)$. While the lecture only said that $\boldsymbol{s}_t$ and $\boldsymbol{c}_t$ should be concatenated first, and the subsequent details were not discussed. One possible way is to transform the concatenated vector to a legally-shaped vector and feed it and $\boldsymbol{y}_t$ to the GRU to get the final $\boldsymbol{s}_t$ and $\boldsymbol{d}_t$, that is, $\boldsymbol{s}_t, \boldsymbol{d}_t = gru(f([\boldsymbol{s}_t; \boldsymbol{c}_t]), \boldsymbol{y}_t)$.

All the above are the differences between these two attention decoders. For more clarity, I also show the architectures of them in the following figures.

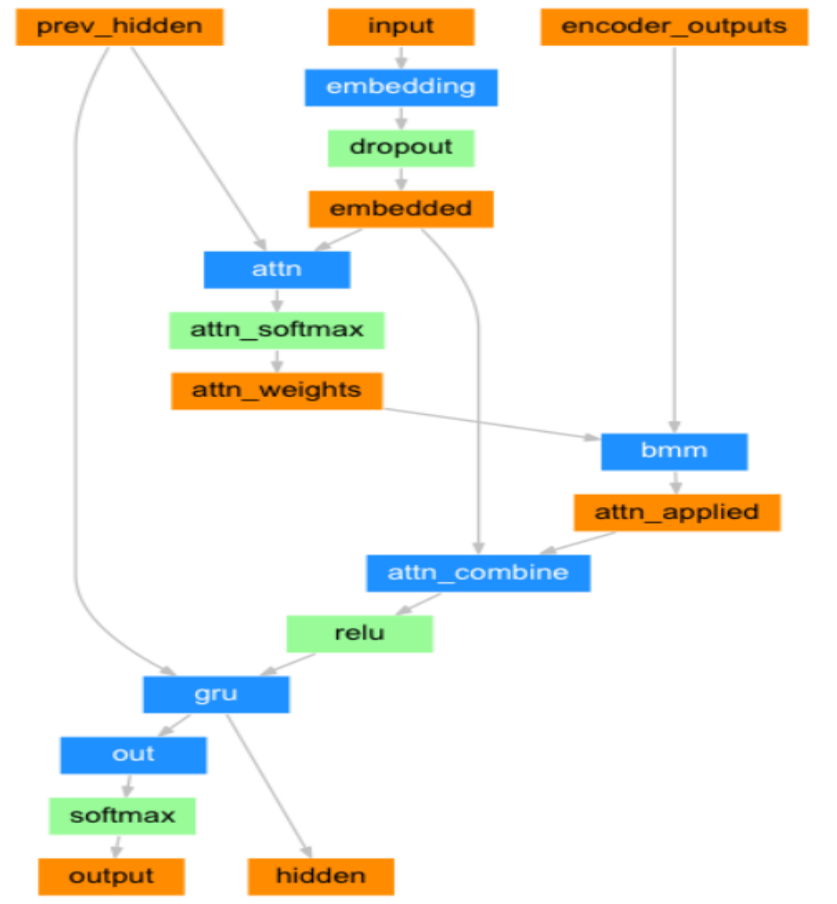


Figure 4. The architecture of the Attention Decoder of Tutorial 6, taken from here[2].

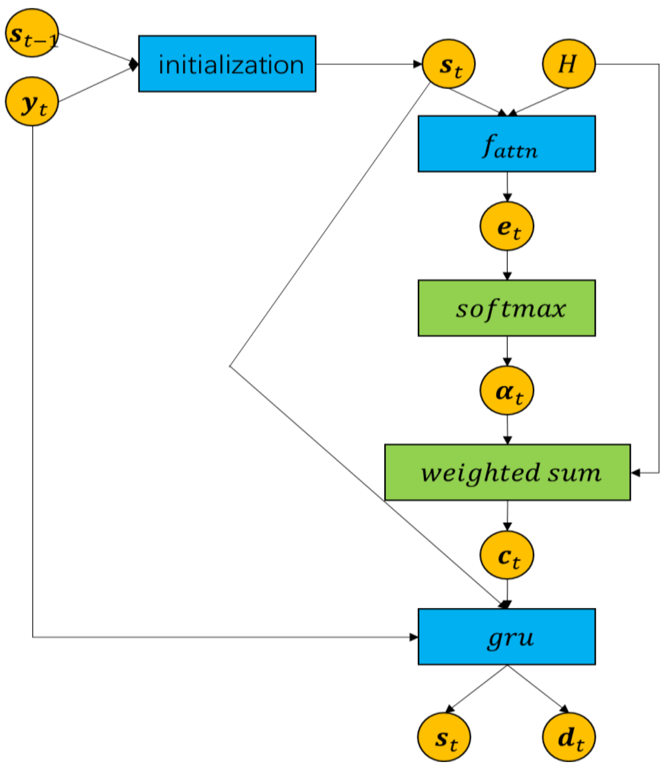


Figure 5. The architecture of the Attention Decoder of the lecture.

[2] https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html

ii.

Since the results here need to be compared with the results from Question b, I also used the same pre-trained word embeddings here. The following tables show the test results.

	BLEU-1	BLEU-2	BLEU-3
CS-EN	0.1876	0.0550	0.0137
DE-EN	0.2048	0.0580	0.0129
FR-EN	0.1839	0.0467	0.0088
RU-EN	0.1529	0.0379	0.0082
Average	0.1823	0.0494	0.0109

Table 7. The test results with the multiplicative attention.

	BLEU-1	BLEU-2	BLEU-3
CS-EN	0.1560	0.0405	0.0085
DE-EN	0.1371	0.0301	0.0016
FR-EN	0.1492	0.0337	0.0029
RU-EN	0.1434	0.0323	0.0025
Average	0.1464	0.0341	0.0039

Table 8. The test results with the additive attention.

	BLEU-1	BLEU-2	BLEU-3
Question b (Tutorial attention)	0.2012	0.0563	0.0125
Multiplicative attention	0.1823	0.0494	0.0109
Additive attention	0.1464	0.0341	0.0039

Table 9. The comparison of the results from Questions b and 2.c.ii of this assignment.

We can observe from Table 9 that both two attention variants perform worse than attention used in the Tutorial and the additive attention performs worse than multiplicative attention. In summary, the more complex the attention mechanism, the worse the performance.

As far as I'm concerned, I don't think there are some clear reasons to explain this phenomenon well. Perhaps the hyperparameters used here or subsequent part of the architecture do not match these two attention mechanisms. Since the more complex the model has the stronger the expressive power, I guess they may need more data and more training time to perform better than the simple one. Or their model capacity is too strong, thus they capture some features irrelevant of this translation task. And these features decrease test performance eventually.