Problem Set #4

Isabelle Lee
CSEP517 - Natural Language Processing

March 18, 2020

[Code] Part 1. Neural Machine Translation

1.1-1.3 Baseline implementation of seq2seq model

All sanity checks have passed for my implementation, and the baseline implementation is stored in p1 folder.

1.4 Run of the baseline implementation

There are 8701 sentences in total, in parallel English and French corpus. The default batch size is 32 for training. Therefore, there are total of 272 batches per epoch (it is the ceiling of 8701/32). In general, the perplexities decrease for both training and dev set, but sometimes it persistently increases. In those cases, patience metric goes up, and when patience hits 5, the learning rate is decayed and the best model saved is loaded. For this code, the dev set perplexity was used for increasing patience metric. In total, the code runs 49 epochs, starting at 1 epoch to complete at 49 epochs.

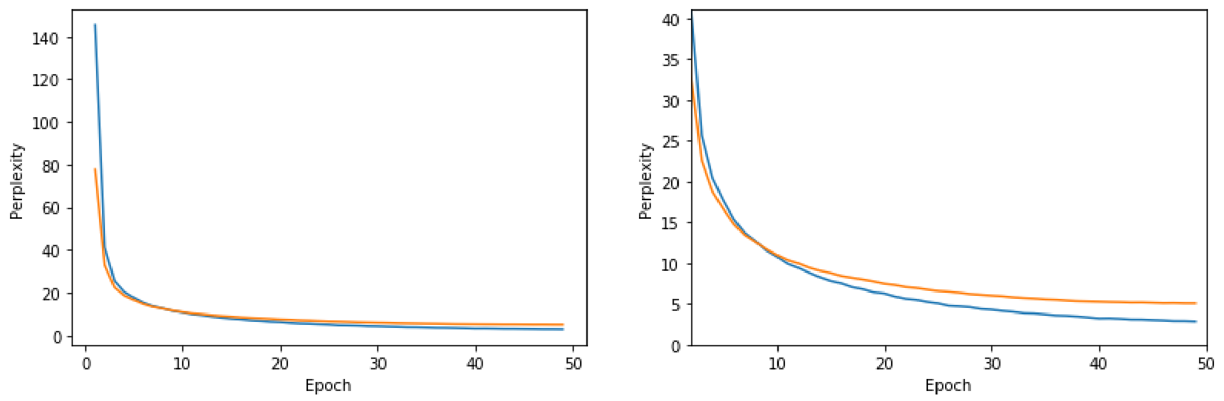

Figure 1: Perplexity curves for encoder dimension = 64 and decoder dimension = 64 (Blue = Train, Yellow = Dev); left: all 49 epochs, right: excluding the first two epochs for better frame

Overall trend for both training set and the dev set, the perplexity generally decreases as the epochs increase. As the training starts, the perplexity is around 140, and towards the end of 49th epoch, the training set reaches perplexity of 2.52. The dev set perplexity decreases as well. Initially the perplexity was a little lower than the dev set, but the perplexity curves cross at around epoch 8. Then, the dev set perplexity remains slightly higher than the training set, but generally decreases over time as well. The final perplexity of dev set is around 4.96. BLEU score for the baseline implementation is 36.34. The BLEU scores for all implementation of various attentions were organized in Table 1.

[Code] Part 2. Neural Machine Translation Experiments

2.1 Changing the hidden dimension sizes of encoders and decoders

In order to be able to set the hidden dimension separately for decoders and encoders, I modified the `__init__` attribute to include `hidden_size_en` and `hidden_size_de`. Then, the sizes of layers were changed accordingly. The projection matrices and the attention projection are all modified to include input and output dimensions, with inputs coming from encoder dimension to projecting to decoder dimension. Particularly, the dimensions of combined output projection would be modified to `(hidden_size_en*2 + hidden_size_de, hidden_size_en)`, since the decoder output vector concatenated with attention is fed into the output projection operator back to encoder dimensions.

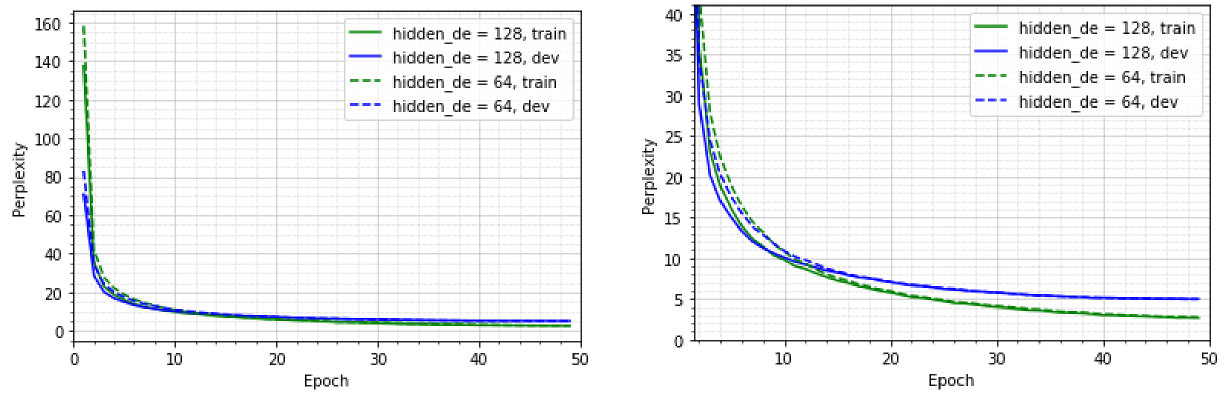

Figure 2: Perplexity curves for encoder dimension = 64 and decoder dimension = 64, 128 (Blue = Train, Green = Dev); left: all 49 epochs, right: excluding the first two epochs for better frame

The perplexity curves look similar to Figure 1, with both training and dev perplexities generally decrease with increasing epoch. The perplexities were slightly lower in the beginning for hidden decoder dimensions of 128 instead of the default of 64. However, the resulting perplexities were slightly higher for larger hidden dimensions. BLEU score was a bit lower for increased hidden dimensions at around 34.4.

Attention	Decoder Hidden Dimension Size	BLEU
Multiplicative	64	36.34085271366298
Multiplicative	128	34.40277736233369
Dot Product	128	38.98250075877358
Additive	128	41.4246363793764

Table 1: BLEU scores for various attentions

2.2 Implementing Other Types of Attention

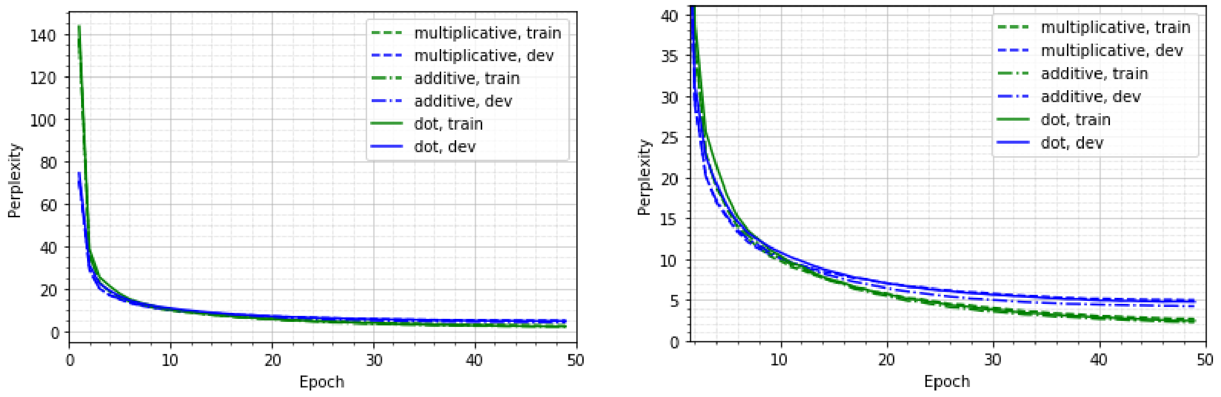

Figure 3: Perplexity curves for encoder dimension = 64 and decoder dimension = 128 for different types of attention (Blue = Train, Green = Dev); left: all 49 epochs, right: excluding the first two epochs for better frame

Additive Attention

For the additive attention, two new attribute layers were defined, `att_projection2` for prohecting decoder hidden layer with weights and `V` for projecting the added weighted total from both encoder and decoder hidden layers. Then, `att_projection2` has the dimensions of `(hidden_size_de, hidden_size_de)`, and `V` has the dimensions of `(hidden_size_de, 1)`. Then, to calculate the energy term, we get

$$e_i = V_t \tanh(W_1 s + W_2 h_i) \tag{1}$$

where W_2 is `att_projection2` and V_t is `V` in the code. In the plot in Figure 3, the perplexities for training and development set was lower than other types of attention. The BLEU score was highest of all other attention, with 41.42 in Table 1.

Dot Product Attention

The dot product attention was calculated directly between the encoder hidden state and the decoder hidden state. Since the encoder is a bidirectional LSTM and the decoder is a unidirectional LSTM, the decoder hidden size has to be twice the size of encoder hidden size for the vector product to compute. The perplexities were slightly lower than the multiplicative attention, but higher than the additive attention. The BLEU score reflected this as well, with a modest gain from multiplicative but not quite as high as the additive attention, at 38.98.

2.3 Further Experimentation

First, I experimented with batch size and hidden dimension sizes. The batch size seemed to peak at batch size of 32. Moreover, I wanted to look into the hidden dimension sizes, since the increase in decoder dimension in section 2.1 actually decreased the BLEU score a bit, so I wanted to investigate further. The experiments with hidden dimension sizes were shown in Table 3. The BLEU score increased for increasing encoder dimension. Since the BLEU score was drastically increased for encoder dimension of 128, encoder dimension of 128 and decoder dimension of 256 was used moving forward.

Batch Size	BLEU
16	38.75402535595097
32	38.98250075877358
64	28.76208234001384

Table 2: BLEU scores for various batch sizes. Dot product attention was used.

Encoder Hidden Dimension Size	BLEU
32	23.507112753357305
64	38.98250075877358
128	47.00847162721194

Table 3: BLEU scores for various hidden dimension sizes of encoder. The decoder hidden dimension is twice the size of encoder hidden dimension. Dot product attention was used.

For further implementation, I implemented the scaled dot product attention. Another type of attention I've calculated was to take the mean of scaled dot product attention, additive attention and multiplicative attention. The scaled dot product attention was calculated by scaling the dot product attention calculated by the square root of the length of the sentence.

$$e_t = \frac{s^T h_i}{\sqrt{d}} \tag{2}$$

and $d =$ dimension of the input, which is the length of the sentence. The averaged attention is calculated using

$$e_t = \frac{1}{3}(e_{t,scaleddot} + e_{t,additive} + e_{t,multiplicative}) \tag{3}$$

Moreover, I used backtranslation on the training set to double the training set. Essentially the french corpus was fed into the model with blank target English corpus for generation.

The best model was using scaled dot product, and this model was saved in p2 folder. The results are summarized in the table below.

Attention	Evaluation Set	BLEU
Scaled Dot Product Attention	Dev	48.43427914079657
Scaled Dot Product Attention	Test	47.32056788482256
Averaged Attention	Dev	47.13524795440857
Averaged Attention	Test	46.024690781066035

Table 4: BLEU scores for Scaled Dot Product Attention and Averaged Attention with backtranslation. Hidden dimensions = (encoder = 128, decoder = 256)

To further improve this, I tried to implement multi-headed attention with scaled dot product attention, but I got perplexity of 0 with repeating output words, i.e. "no no no", "`<pad> <pad>`", etc. This would yield low perplexity scores, but the attention score calculation was probably incorrect. I think there might have been an error with how it was implemented. Basically, I calculated the scaled dot product attention by the number of heads. Then, the resulting attention vectors were concatenated and fed into a linear layer of weights to produce final output. I think the error was in how the attention layer was concatenated before it was fed for final output. Given more time, I would separate out this portion from the decoder calculation all together for better debugging.

[Written] Part 3. Word2Vec

3.1 On the objective function of Word2Vec with Negative Sampling

a) Disadvantage of using the vocabulary set instead of negative sampling set.

If we modify the equation (0.1) to sum over all possible context in vocabulary instead of negative samples in the denominator, that would be a huge computational cost. Calculating the dot products are already computationally expensive, and doing so over the entire vocabulary would be hugely expensive. Moreover, calculating the gradients for updating weights would be even more time consuming.
On the other hand, the vocabulary may include common words, such as "the" or "a", and since those are ubiquitous, the dot product for the context vectors containing common words would be high. If we use unigram frequencies in the negative sampling, the frequencies of the words can be taken into consideration in terms of calculating the objective function.

b) Does SGNS depend on how negatives are sampled? If so, what would happen if we sample negatives from the vocabulary at uniform random?

If the negative samples are picked at uniformly random, then the denominator calculation could include extremely rare words and extremely common words at equal probability. Then, the normalization of the denominator may not be effective for small number of samples. For instance, consider the case where we pick 5 words for negative contexts. If we happen to pick very rare contexts for all 5, then the dot product $v_{c'} \cdot u_w$ will invariably be very small for all negative context, and the normalization of the denominator would no longer be representative of the corpus. In order for the equation to be representative of the corpus, the calculation may involve a huge number of samples, which defeats the purpose of negative sampling. Picking based on smoothed unigram distribution allows for negative sampling to pick representative group of words, since the more frequently occuring words are more likely to be picked for negative samples.

3.2 Describe the effect of window sizes for training Word2Vec on large corpuses, i.e. $win = 1, 5, 100$.

If the window size is 1, the model is equivalent to unigram features. Then, the context vector is not considered in the calculation. On the other hand, if the window size is too large, i.e. 100, the context becomes broad, like a paragraph. Some of the words would be paired with contexts that are so far away that they are not meaningful. Then picking the appropriate size of the window is important to capture context-word pairs that have meaningful relationships. Window size of 5 would be able to capture words that appear together in a sentence or a phrase.

3.3 Describe the effect of the unigram smoothing for negative sampling.

The smoothing parameter has the effect of lowering high frequencies and increasing frequencies for rare words. Consider $x^{\frac{3}{4}}$. This function increases more rapidly near 0 than 1. If we want to be sure, we can take the derivative,

$$\frac{d}{dx}x^{\frac{3}{4}} = \frac{3}{4x^{1/4}} \tag{4}$$

We can see that as $x \to 0$, this derivative will blow up. As $x \to 1$, this approaches .75. So, the smoothing parameter increases frequencies for very rare words, and decreases frequencies for very frequent words. Then, considering the dot product between words and rare contexts, the dot product will increase in general for rare context.

3.4 Error analysis

To understand failures better, we can decompose equation 0.4. Since the cosine similarities can be written as normalized dot products between vectors, we write cosine similarities in terms of dot products as below.

$$\begin{aligned} \cos(y, b-a+x) &= \frac{1}{|y||b-a+x|}[y \cdot (b-a+x)] \\ &\propto y \cdot b - y \cdot a + y \cdot x \end{aligned} \tag{5}$$

where x, y, a, and b are vectors and $|y|$ and $|b-a+x|$ are vector norms. Then, we can see what might have contributed to errors. Since the cosine similarities are proportional to a series of dot products, if the dot product of word and context compete against the comparison terms, the errors like `london:england::baghdad:mosul` types of errors can result. In this particular case, the dot product between x =baghdad and y =mosul probably was high enough to overpower the comparison dot products $y \cdot b$ and $-y \cdot a$.