CSCE 5290 Sp 2025 Homework 3

Instruction

The homework has a total of 30 points and counts toward 7.5% of your grade. Show all steps – you can get partial credit for wrong answers with correct steps but no credit for a correct answer with no steps.

1 Attention

The goal of this exercise is to get familiar with how attention is used in different NLP models. *Some of these problems might be easier to solve if you write codes for them.* If you do it that way, please add the code to the answer.

1. We will start with the attention mechanism used in the encoder-decoder architecture for neural machine translation (NMT). Both the encoder and the decoder is an RNN.

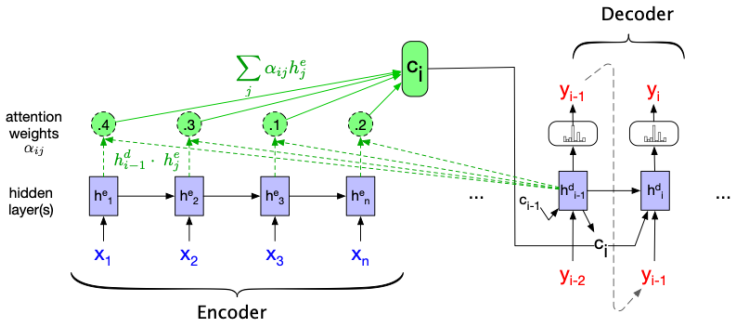

Figure 1: Attention for encoder-decoder in NMT [JM25]

The contribution ($\mathbf{c_i}$) of encoder hidden states (h_j^e) to the decoder state i is given by the following set of formulae (the particular attention mechanism is

called scaled dot product attention [Vas+17]):

$$\mathbf{c_i} = \sum_j \alpha_{ij} h_j^e$$
$$\alpha_{ij} = \text{softmax}(\text{score}(h_{i-1}^d, h_j^e)) \quad (1)$$
$$\text{score}(h_{i-1}^d, h_j^e) = \frac{h_{i-1}^d \cdot h_j^e}{\sqrt{n}} \; n = \text{dimension of } h_{i-1}^d.$$

Let's consider a `En-Fr` translation: `this is a translation` → `cést une traduction`. The following tables show the $\mathbf{x_i}$ values (embeddings for the English tokens), h_i^d and h_i^e values. Calculate the $\mathbf{c_i}$ for the French token `traduction`. **6 points**

Token	token representation	$\mathbf{h_i^e}$	$\mathbf{h_i^d}$
this	[0.12, -0.87, 0.33, 0.45]	[0.64, -0.27, 0.89, -0.12]	-
is	[0.76, 0.21, -0.34, 0.67]	[-0.45, 0.33, 0.71, 0.08]	-
a	[-0.55, 0.18, 0.29, -0.73]	[0.19, -0.94, 0.56, 0.37]	-
translation	[0.03, -0.99, 0.42, 0.11]	[0.03, 0.85, -0.41, 0.76]	-
cést	[0.76, 0.21, -0.34, 0.67]	-	[0.58, -0.13, 0.94, 0.22]
une	[-0.31, 0.66, -0.74, 0.09]	-	[0.45, 0.11, -0.88, 0.67]
traduction	[-0.92, 0.37, 0.28, -0.50]	-	-

Table 1: $\mathbf{x_i}$, $\mathbf{h_i^e}$ & $\mathbf{h_i^d}$ values

2. Next, we will consider **Single Head Self Attention**. Consider you have an English sentence "this is translation". Use the same token representations from Table 1. Compute the representation of the token "is" after passing this sentence to a single-head attention layer, which does the following operations in sequence: **6 points**

- Multiplies the data matrix with $\mathbf{W^Q}$, $\mathbf{W^K}$ and $\mathbf{W^V}$ matrices to produce the Query (Q), Key (K), and Value (V) matrices. These are the *parameters* of the `Attention Layer` that are usually learned by training the model, but **we are providing them below**.
- Computes the representations of the tokens using the following formula: $\text{softmax}(\frac{Q \cdot K^T}{\sqrt{d}}) \cdot V$ where d is the dimension of the representation, which, in this case, is 4. Note this is the same as the **scaled dot product attention** we saw before.

$$\mathbf{W^Q} = \begin{bmatrix} 0.12 & -0.87 & 0.33 & 0.45 \\ 0.76 & 0.21 & -0.34 & 0.67 \\ -0.55 & 0.18 & 0.29 & -0.73 \\ 0.03 & -0.99 & 0.42 & 0.11 \end{bmatrix} \quad \mathbf{W^K} = \begin{bmatrix} 0.64 & -0.27 & 0.89 & -0.12 \\ -0.45 & 0.33 & 0.71 & 0.08 \\ 0.19 & -0.94 & 0.56 & 0.37 \\ 0.03 & 0.85 & -0.41 & 0.76 \end{bmatrix}$$

$$\mathbf{W^V} = \begin{bmatrix} 0.58 & -0.13 & 0.94 & 0.22 \\ -0.31 & 0.66 & -0.74 & 0.09 \\ 0.45 & 0.11 & -0.88 & 0.67 \\ -0.92 & 0.37 & 0.28 & -0.50 \end{bmatrix}$$

3. Next, we will consider **Multi Head Self Attention**, similar to what is done in `BERT` [Dev+19]. Consider the same sentence as before. The multi-head attention formula is given by:

$$\text{MultiHead}(X) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\mathbf{W^O}$$

where

$$\text{head}_i = \text{Attention}(XW_i^Q, XW_i^K, XW_i^V)$$

[1] There are a couple of things to notice.

- head_i refers to the representation of the tokens as you would calculate from the Single-Head Self-Attention mechanism before, but now you use $\mathbf{W_i^Q}$, which has dimensions different than the $\mathbf{W^Q}$ (also, X refers to the data matrix). In this problem, if you have **2** heads, what would be the dimensions of $\mathbf{W_i^Q}$, $\mathbf{W_i^K}$ and $\mathbf{W_i^V}$? Depending on this dimension, split the given $\mathbf{W^Q}$, $\mathbf{W^K}$, $\mathbf{W^V}$ matrices so that you can compute head_1 and head_2. Just show the split – you don't have to compute the head_i values. **3 points**
- What is the dimension of $\text{Concat}(\text{head}_1, \text{head}_2)$? **2 point**
- $\mathbf{W^O}$ is a matrix that projects the output of the `concat` operation. What are the dimensions of $\mathbf{W^O}$, assuming the multi-head self-attention operation is dimension preserving, i.e., if you send in a $T \times d$ (T = sequence length, d is the dimension of the token representation) data matrix, you get a $T \times d$ matrix as output? **2 point**
- Based on what you have done above, what would you think are the benefits of using multiple heads over a single head? **3 points**

2 Sub-word Tokenization

In class, we had seen one tokenization technique – "Byte-Pair Encoding". Here, we will look into a similar sub-word tokenization technique called **WordPiece** tokenizer. WordPiece is the tokenization algorithm used in most `BERT` models.

[1] If you look at the PyTorch API for Multi-Head Attention, you will see this is written as $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$. This is because the API considers you can send in different inputs for computing attention, but for *self*-attention, we are computing attention between the tokens of the same sequence.

However, the implementations differ in various libraries. The goal of this exercise is to be familiar with sub-word tokenization techniques in general.

The following material is taken (mostly verbatim) from The Huggingface NLP Course [Fac].

WordPiece starts with a small vocabulary, including the special tokens used by the model and the initial alphabet. The initial alphabet contains all the characters present at the beginning of a word and the characters present inside a word preceded by the WordPiece prefix (like `##` for BERT). So, the token "word" gets split like `w ##o ##r ##d`.

Then, similar to BPE, WordPiece will learn to merge. The main difference is the way the pair to be merged is selected. Instead of selecting the most frequent pair, WordPiece computes a score for each pair using the following formula:

$$\text{score}_{pair} = \frac{\text{freq}_{pair}}{\text{freq}_{element1} \times \text{freq}_{element2}}$$

By dividing the frequency of the pair by the product of the frequencies of each of its parts, the algorithm prioritizes the merging of pairs where the individual parts are less frequent in the vocabulary. For instance, it won't necessarily merge ("un", "##able") even if that pair occurs very frequently in the vocabulary because the two pairs "un" and "##able" will likely each appear in a lot of other words and have a high frequency. In contrast, a pair like ("hu", "##gging") will probably be merged faster (assuming the word "hugging" often appears in the vocabulary) since "hu" and "##gging" are likely to be less frequent individually.

Let's consider the following vocabulary (token and the number of times it appears in the corpus):

```
("hug", 10), ("pug", 5), ("pun", 12), ("bun", 4),
("hugs", 5)
```

So the initial setup is the following (if we don't consider the special tokens such as `[CLS, [SEP], [MASK]`):

```
Vocabulary: ["b", "h", "p", "##g", "##n", "##s", "##u",
"##g"]
Corpus: [
    ("h" "##u" "##g", 10),
    ("p" "##u" "##g", 5),
    ("p" "##u" "##n", 12),
    ("b" "##u" "##n", 4),
    ("h" "##u" "##g" "##s", 5)
]
```

The most frequent pair is `("##u", "##g")` (present 20 times), but the individual frequency of `"##u"` is very high. All pairs with a `"##u"` actually have that same score (1/36) (because `##u` appears in **all** words). So, the best score goes to the pair `("##g", "##s")` — the only one without a `"##u"` — at 1/20, and the first merge learned is `("##g", "##s") -> ("##gs")`. Note that when

we merge, we remove the `##` between the two tokens, so we add `"##gs"` to the vocabulary and apply the merge in the words of the corpus:

```
Vocabulary: ["b", "h", "p", "##g", "##n", "##s", "##u",
"##gs"]
Corpus: [
    ("h" "##u" "##g", 10),
    ("p" "##u" "##g", 5),
    ("p" "##u" "##n", 12),
    ("b" "##u" "##n", 4),
    ("h" "##u" "##gs", 5)
]
```

At this point, `"##u"` is in all the possible pairs, so they all end up with the same score. Let's say that in this case, the first pair is merged, so `("h", "##u") -> "hu"`. This takes us to:

```
Vocabulary: ["b", "h", "p", "##g", "##n", "##s", "##u",
"##gs", "hu"]
Corpus: [
    ("hu" "##g", 10),
    ("p" "##u" "##g", 5),
    ("p" "##u" "##n", 12),
    ("b" "##u" "##n", 4),
    ("hu" "##gs", 5)
]
```

We will repeat this process until a desired vocabulary size is reached (breaking ties randomly)

4. Show the WordPiece tokenization process (the vocabulary and corpus, as shown before) for the **first 3** steps for the following vocabulary. **8 points (2 + 3 + 3)**

```
("man", 10), ("can", 5), ("cat", 12), ("bat", 4),
("mans", 5)
```

2.1 References

[Dev+19] Jacob Devlin et al. 'BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding'. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, 2019, pp. 4171–4186.

[Fac] Hugging Face. *WordPiece tokenization*. `https://huggingface.co/learn/nlp-course/en/chapter6/6`. Accessed: 2025-01-11.

[JM25] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*. 3rd. Online manuscript released January 12, 2025. 2025.

[Vas+17] Ashish Vaswani et al. 'Attention is all you need'. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS'17. Curran Associates Inc., 2017, pp. 6000–6010.