INF265
Project 3:
Sequence models

Deadline: May 5th, 23.59
Deliver here:
`https://mitt.uib.no/courses/39835/assignments/73521`

Projects are a compulsory part of the course. This project contributes a total of 15% of the final grade. **Projects have to be done in pairs. If you have good reasons not to do it in pairs, contact Pekka by email before April 21st.** Add a paragraph to your report explaining the division of labor (Note that both students will get the same grade regardless of the division of labor).

Code of conduct: You are allowed to copy-paste code from the solutions of previous weekly exercises. However, it is not allowed to copy-paste from online sources nor from other students' projects. To some extent, discussions on parts of the project with other pairs/students are tolerated. If you do so, indicate with whom and on which parts of the project you have collaborated. 3 provides some hints, if you need additional assistance, teaching assistants and group leaders are available to help you.

Grading: Grading will be based on the following qualities:

- Correctness (your answers/code are correct and clear)
- Clarity of code (documentation, naming of variables, logical formatting)
- Reporting (thoroughness and clarity of the report)

Deliverables: You should deliver the following files:

- A jupyter notebook addressing the tasks defined in this project. Cells should already be run and output visible. Alternatively, you can also have a separate notebook for each section of this project
- A PDF report addressing section 5. Note that exporting your notebook as a PDF is not what is expected here. Remember to include all the results, visualizations and comments that are expected.

If you need to provide additional files, include a `README.txt` that briefly explains the purpose of these additional files. In any case, do not include the datasets in your submission.

Late submission policy: All late submissions will get a deduction of 2 points. In addition, there is a 2-point deduction for every starting 12-hour period. That is, a project submitted at 00.01 on May 6th will get a 4-point deduction and a project submitted at 12.01 on the same day will get a 6-point deduction (and so on). (Executive summary: Submit your project on time.) There will be no possibility to re-take projects, so start working early.

1 Introduction

In this project, you will define and train recurrent neural networks to generate text sequences. Objectives include getting a better understanding of **a)** sequence data **b)** recurrent neural networks **c)** attention mechanisms **d)** word embeddings **e)** text generation.. This project involves the training of many different models, and is therefore a good opportunity to take up the habit to save and load models.

Sequence data pose new challenges to neural networks. First, input and output sequence lengths can vary between samples, which is not compatible with the types of network seen so far. Then, to learn efficiently, the neural network should be built such that its weights can share features learned across different positions of the same element in different sequences.

Text data are a type of sequence data and require different preprocessing steps from image data. For example, if each word in the vocabulary is considered as a class, then the number of classes becomes extremely high. In addition, some classes are closer to each other than others. For example, synonyms are very close, and mistaking the word *man* for *gentleman* should be acceptable. Even antonyms such as *black/white* and *happy/sad* are similar in the sense that they both describe a color or a mood. In addition, most words have different variants depending on the context, *man/men*, *child/children chair/chairs* are all different singular/plural pairs. A same concept can also yield different words depending on its role in the sentence: *luck/lucky/luckily*. Word embedding is a technique that alleviates these problems, both reducing the number of components necessary to represent a word, and bringing together similar words.

Text generation is a complex task. First, to generate a correct sentence, the neural network has to learn word semantics of a very large lexicon and syntax rules only through examples. Then, ideally, each sentence produced bears an interesting message, logically linked to the previous and next sentences.

While learning some of the semantics should be achievable using a CPU, generating sentences that carry a proper message is definitely out of the scope of this project. An easier task could be conjugating *be* and *have*. This requires the neural network to understand in which tense the rest of the sentence is, who the subject is and whether *be* or *have* makes more sense in this context. However, this assumes that the user already knows that the next word is either a variant of *be* or *have*.

2 Tasks

For this project, the train/validation/test datasets consist of free ebooks from Gutenberg project (`https://www.gutenberg.org/`) saved as `txt` files. You are free to use other books than those provided, but beware of potential additional preprocessing steps if you do so.

Section 2.1 addresses word embeddings, section 2.2 focuses on conjugating *be* and *have* and section 2.3 attempts to generate sentences. Sections 2.2 and 2.3 are independent, but both relies on section 2.1

2.1 Word embedding

1. Read `txt` files and tokenize them to obtain train/validation/test lists of words.
2. Define a vocabulary based on the training dataset. To avoid getting a too large vocabulary, a solution can be to keep only words that appear at least 100 times in the training dataset. Report the total number of words in the training dataset, the number of distinct words in the training dataset and the size the defined vocabulary. Comment your results.

3. Define a continuous bag of words model architecture based on this vocabulary that contains an embedding layer. To drastically reduce computational cost, the dimension of the embedding `emb_dim` can be very low such as 16, 12, or even 10. Of course, in a real setting a larger space would be used. You are **not allowed** to use `nn.LazyLinear` in this project.

4. Train several models, select the best one and evaluate its performance. Note that the performance here is potentially extremely low, but the real objective is not to train a good predictor, only to have a good representation of the semantic of each word in the vocabulary.

5. Compute the cosine similarity matrix of the vocabulary based on the trained embedding. For some words of your choice (e.g. *me*, *white*, *man*, *have*, *be*, *child*, *yes*, *what* etc.), report the 10 most similar words. Comment your results.

6. Visualize the embedding space on `https://projector.tensorflow.org/`. To do so, upload the vocabulary and their corresponding values in the embedding space as two `tsv` files. Try to find and select clusters. Report both plots (you can use screenshots) and their corresponding selections for some meaningful clusters. Comment your results.

2.2 Conjugating *be* and *have*

1. Use your trained word embedding and define a simple MLP architecture, an MLP architecture that has first an attention layer (see section 4), as well as a RNN architecture to predict *be* and *have* conjugation given the context *around* the missing target. Use the same context size `max_len` for both MLPs and RNNs, even though RNNs and attention layers could take a context size of arbitrary length. You are **not allowed** to use `nn.LazyLinear` in this project.

2. Train several models, select the best one and evaluate its performance. Comment the differences in terms of performances/training time between the MLP architectures (with and without the attention layer) and the RNN architecture.

2.3 Text generation

1. Use your trained word embedding and define a RNN architecture that can predict the next word given the context *before* the target.

2. Train several models, select the best one and evaluate its performance. Note that the performance here is potentially very low.

3. Implement the beam search algorithm.

4. Have fun playing with your model. Start a sentence, give it to your model, and let it complete the next n words using the beam search algorithm. Report and comment your results.

3 General hints

- You can use weights in the loss function based on each class frequency to alleviate the effects of class imbalance when training your continuous bag of words model.

- To build your different training/validation/test datasets, `(context, target)` pairs have to be defined such that `target` is a single word and `context` are the words around/before it. You can skip some words when selecting targets from the texts, such as `<unk>`, punctuation symbols, etc. and/or keep only at most a given number of occurrences for each target.

- You can (*should!*) save (and then load when needed) your vocabulary, tokenized text, weights, word embedding and models to save time.

- For this project in general, we recommend you to use the Adam optimizer.

- To save tensors as `tsv` files, you can use the `to_csv` method from Pandas library after converting your tensors to Pandas Dataframes. To match the website's requirements use the following parameters: `sep="\t", header=False, index=False`
- Your trained word embedding can be integrated in your next models in different ways. For example, you can transform inputs before feeding them to your models, or have a first layer in your models with the same weight values as your trained embedding matrix, or simply define the computations between the input and the embedding matrix in the forward function. In any case, make sure that your embedding layer doesn't get updated when training the rest of your models!
- To predict *be* and *have* conjugation, the `(contexts, targets)` datasets must be such that the targets are *be*, *am*, *are*, *is*, *was*, *were*, *been*, *being*, *have*, *has*, *had*, *having*. The output layer must match the number of classes to predict and the labels must be mapped from their original index in the vocabulary to integers between 0 and 11.
- In pytorch, dataloaders expect samples of identical shapes although RNN architectures and attention layers can take sequences of arbitrary length. For an efficient training use contexts of equal length so that dataloaders can be defined. When playing with your trained RNN model in section 2.3, use sequences of arbitrary length.

4 A simple attention layer

By a *simple* attention layer we mean:

- multihead dot product self-attention
- using sequences of fixed length (`max_len`)
- using positional encoding
- without using *masked* softmax (only a regular softmax)
- without implementing causality
- without parallel computing of heads
- without parallel computing of queries/keys/values,
- without a normalization layer

In this section, we will briefly recall a few of these concepts while trying to simplify the notations of the course material for this specific primitive attention layer. Please remember that you are **not allowed** to use `nn.LazyLinear` in this project.

4.1 Positional encoding

Positional encoding is one way to include information on the position of each element of the input sequence in their embedding. More specifically, for any sequence of length `max_len` and an embedding space of dimension `emb_dim`, the corresponding positional encoding $\mathbf{P} \in \mathbb{R}^{\texttt{max\_len} \times \texttt{emb\_dim}}$ is given by

$$\begin{aligned} \mathbf{P}[i, 2j] &= \sin\left(\frac{i}{10000^{2j/\texttt{emb\_dim}}}\right), \\ \mathbf{P}[i, 2j+1] &= \cos\left(\frac{i}{10000^{2j/\texttt{emb\_dim}}}\right). \end{aligned} \qquad \text{for } \begin{cases} i = 0, \cdots, \texttt{max\_len} - 1 \\ j = 0, \cdots, \frac{\texttt{emb\_dim}}{2} \end{cases} \tag{1}$$

To obtain the *final* embedding $\mathbf{X}_f$ of any sequence of embeddings $\mathbf{X} \in \mathbb{R}^{\texttt{max\_len} \times \texttt{emb\_dim}}$ which includes both the word embedding and the positional encoding, we simply sum them up:

$$\mathbf{X}_f = \mathbf{X} + \mathbf{P}, \qquad \text{with } \mathbf{X}_f, \mathbf{X} \text{ and } \mathbf{P} \in \mathbb{R}^{\texttt{max\_len} \times \texttt{emb\_dim}}. \tag{2}$$

If we assume that broadcasting operations are defined, Eq(2) still holds for batch computations but with $\mathbf{X}_f, \mathbf{X} \in \mathbb{R}^{N \times \texttt{max\_len} \times \texttt{emb\_dim}}$ and $\mathbf{P} \in \mathbb{R}^{1 \times \texttt{max\_len} \times \texttt{emb\_dim}}$. Note that the matrix $\mathbf{P}$ is independent of the input sequence here, so it can be computed once, and then used for all batches without being re-defined.

4.2 Simple-head dot-product self-attention

The output of an attention layer is often called the *head* $\mathbf{h}$ of the layer. In this project, for a given embedded word $\mathbf{x}_{i, i \in [0, \cdots, \texttt{max\_len}-1]} \in \mathbb{R}^{\texttt{emb\_dim}}$ of a given sequence $\mathbf{X}_f \in \mathbb{R}^{\texttt{max\_len} \times \texttt{emb\_dim}}$, the head $\mathbf{h}$ of a simple-head dot-product self-attention, is obtained using

$$\mathbf{h} = f(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \in \mathbb{R}^{p_v}$$

with:

- $\mathbf{Q} = \mathbf{W}^{(q)}\mathbf{x}_i \ \in \mathbb{R}^{p_{qk}}$, with $\mathbf{W}^{(q)}$ learnable parameters in $\mathbb{R}^{p_{qk} \times \texttt{emb\_dim}}$
- $\mathbf{K} = \mathbf{W}^{(k)}\mathbf{x}_i \ \in \mathbb{R}^{p_{qk}}$, with $\mathbf{W}^{(k)}$ learnable parameters in $\mathbb{R}^{p_{qk} \times \texttt{emb\_dim}}$
- $\mathbf{V} = \mathbf{W}^{(v)}\mathbf{x}_i \ \in \mathbb{R}^{p_v}$, with $\mathbf{W}^{(v)}$ learnable parameters in $\mathbb{R}^{p_v \times \texttt{emb\_dim}}$
- $f(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{p_{qk}}}\right)\mathbf{V} \in \mathbb{R}^{p_v}$.

Note that p_{qk} and p_v are chosen arbitrarily and can thus be equal. From now on, we will use p such that $p_{qk} = p_v = p$.

In practice, computations have to be done for an entire batch of whole sequences. The use of broadcasted operations and Batch Matrix Multiplication (BMM) is then required, and additional dimensions N and `max_len` are to be taken into account.

4.3 Multi-head attention

The multi-head attention concept is rather straightforward once the simple head attention layer has been defined.

Given an embedded word $\mathbf{x}_{i, i \in [0, \cdots, \texttt{max\_len}-1]} \in \mathbb{R}^{\texttt{emb\_dim}}$ of a given sequence $\mathbf{X}_f \in \mathbb{R}^{\texttt{max\_len} \times \texttt{emb\_dim}}$,

1. Instead of one set of trainable parameters $(\mathbf{W}^{(q)}, \mathbf{W}^{(k)}, \mathbf{W}^{(v)})$ and one head $\mathbf{h}$, define `n_head` trainable sets $(\mathbf{W}_i^{(q)}, \mathbf{W}_i^{(k)}, \mathbf{W}_i^{(v)})_{i, i=0, \cdots, \texttt{n\_head}-1}$ and compute `n_head` heads $\mathbf{h}_{i, i=0, \cdots, \texttt{n\_head}-1}$.
2. Concatenate these heads into $\mathbf{h} = [\mathbf{h}_0, \cdots, \mathbf{h}_{\texttt{n\_head}-1}] \in \mathbb{R}^{(p \times \texttt{n\_head})}$
3. Define an additional trainable parameter $\mathbf{W}^{(o)} \in \mathbb{R}^{\texttt{emb\_dim} \times (p \times \texttt{n\_head})}$ and the output of the multihead attention layer is then $\mathbf{H} = \mathbf{W}^{(o)}\mathbf{h} \in \mathbb{R}^{\texttt{emb\_dim}}$.

Note that the embedding that contains information on both the word embedding and the positional encoding can be computed once and then sent to all heads.

Here again, computations have to be done for an entire batch of whole sequences. Broadcasted operations and BMM are also required there, and additional dimensions N and `max_len` are to be taken into account.

To implement multi-head attention you can use `nn.ModuleList`

5 Report

In addition to the specific comments asked in section 2, the report should address the following points:

1. Give an explanation of your approach and design choices to help us understand how your particular implementation works.
2. Report the different models and hyper-parameters you have used.
3. Report the performance of your selected model.
4. Comment your results. In case you do not get expected results, try to give potential reasons that would explain why your code does not work and/or your results differ.

and answer the following questions

1. What is the purpose of the *masked* softmax? Explain in your own words what it is.
2. How do attention layers deal with sequences of different length?
3. What is causal self-attention? When should it be used?
4. What are the advantages of *multi-head* self-attention?