Project 3: Sequence Models

INF265 - University of Bergen

Vebjørn Sæten Skre Martin Flo Øfstaas

Spring 2024

Division of labour: We did split this into a three part project, where we did most of the word embeddings us two, and then we split the project into Martin doing the conjugation task and Vebjørn the Text generation task.

1 Our approach and design choices

Before we go into the details of our programs, we think it is necessary to show our file structure. We first split the three tasks into three notebooks named "word_embeddings.ipynb, conjugation.ipynb" and "text_generation.ipynb". With these files together with the data already given we created the following file structure:

```
project3/
├── word_embeddings.ipynb
├── conjugation.ipynb
├── text_generation.ipynb
├── utils
├── data/
│   └── ..
└── generated/
    ├── word_embeddings/
    │   └── ..
    ├── conjugation/
    │   └── ..
    └── text_generation/
        └── ..
```

We replicated the following file structure in google colab. To make a dynamic switch between colab and testing on our own PC's we created the following code to make the paths flexible depending on the variable "using_colab" was true or not. Since Vebjørn has a slight difference in his google disk directory he added inf265 between MyDrive and project3 in the path.

```
using_colab = True # set this to false if you want to run it locally
COLAB_PATH = "/content/drive/MyDrive/project3/"
PATH_GENERATED_EMBS = "generated/word_embeddings/" if not using_colab
    else COLAB_PATH + "generated/word_embeddings/"
PATH_GENERATED_TXT_GEN = "generated/text_generation/" if not using_colab
    else COLAB_PATH + "generated/text_generation/"
PATH_GENERATED_CONJ = "generated/conjugation" if not using_colab
```

```
    else COLAB_PATH + "generated/conjugation/"
PATH_DATA = "data/" if not using_colab else COLAB_PATH + "data/"
```

We then saved all files that we trained for each specific task into their respective folders using the paths. When training our model's we trained on google colab on a TESLA T4 GPU.

For all of our models we had the same training loop. This loop was implemented by adding the models to train in a list, and then store the models stats during the training phase. Then we selected the model with the highest validation accuracy as our main model. This model was then stored for all in its respective paths. We also created our files so that we could run them all automatically without initiating the training *if* the models were already created.

1.1 Word embeddings

For our implementation of the word embeddings, we used the tutorial from our professor as the main inspiration. We created our dataset with CONTEXTSIZE = 4, meaning we used the 2 first words and the 2 preceding words in addition to the target in the middle. We then created three different models to train our embeddings, each a little more complex than the other. We made sure that all of them had the same had the same input and output layer dimension. The input layer had the dimension $embed\_dim * context\_size$ and output layer had the dimension VOCAB.

```
#This was our simple model
class simple_eTrain(nn.Module):
    def __init__(self, context_size=CONTEXT_SIZE):
        super().__init__()
        (vocab_size, embedding_dim) = (VOCAB_SIZE, 10)
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.fc1 = nn.Linear(embedding_dim*context_size, 128)
        self.fc2 = nn.Linear(128, VOCAB_SIZE)

    def forward(self, x):
        # adds a trainable embedding layer
        out = self.embedding(x)
        # out is now of shape (N, context_size, embedding_dim)
        out = F.relu(self.fc1(torch.flatten(out, 1)))
        # out is now of shape (N, context_size*embedding_dim)
        out = self.fc2(out)
        return out
```

1.1.1 Hyperparameters

For the hyperparameters we used when training our word embeddings we use Adam optimzer with learning rate 0.001. We then used a batch size of 512 and trained for 50 epochs. As the input dimensions for the models, we used embedding_dimm * context_size as the input dimensions, equaling $14 * 4$. The output layer outputted the VOCAB_SIZE. We calculated the class weights and used that as the weights in CrossEntropyLoss.

1.1.2 Results and thoughts

To get the cosine similarity of our words we implemented Pytorch's function nn.CosineSimilarity. An example output is given below:

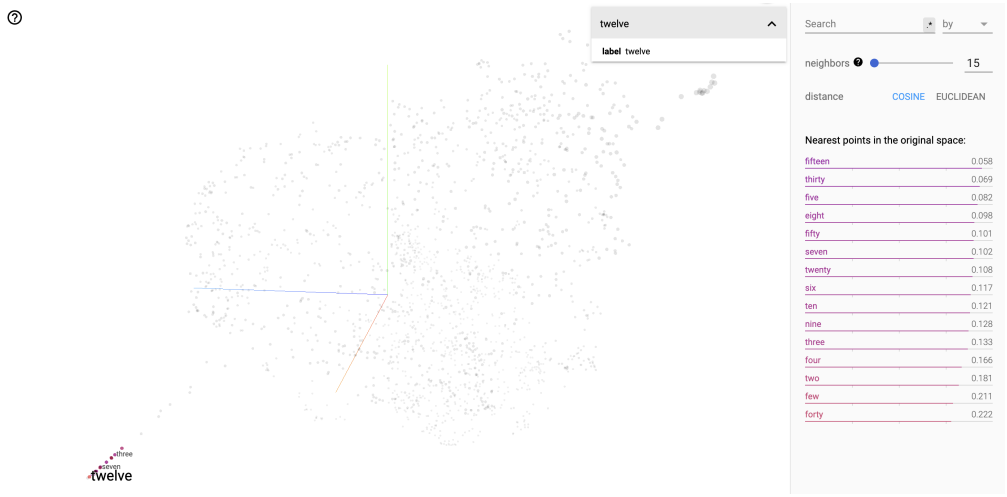


Figure 1: From Embedding projector

```
The ten most similar words to king is:
 ['earl', 'queen', 'father', 'brother', 'sister', 'mother', 'peasant',
 'commander', 'plan', 'order']

The ten most similar words to child is:
 ['heart', 'name', 'convict', 'man', 'woman', 'children', 'wound',
 'sovereign', 'dream', 'soul']

The ten most similar words to woman is:
 ['wife', 'lady', 'friend', 'fellow', 'count', 'boy', 'father', 'women',
 'grandfather', 'daughter']

The ten most similar words to man is:
 ['beast', 'person', 'hearts', 'heart', 'soul', 'child', 'woman',
 'reader', 'conscience', 'beings']

The ten most similar words to can is:
 ['could', 'might', 'will', 'll', 'would', 'may', 'cannot', 'must',
 'should', 'shall']
```

It is also interesting to look at how our word embeddings looks like when uploading them to the embedding projector Tensorflow website. Below is a picture taking from it whre we can clearly see some clusters forming in the low left and high right area.

Here we can clearly see that the clusters are forming around the other numbers, as highlighted on the table on the right. Similarly, the cluster in the upper right has great similarities with words like: *could, might, will, would, may, cannot, must, should, ..., does*. This can be seen in Figure 2. From the plots of the most similar words, it's clear that words like woman, child, king, and man clearly matches up with other similar words. We think that since we are only using 14 dimensions in our embeddings, that we are not able to pick up patters between genders, only genders in general terms.

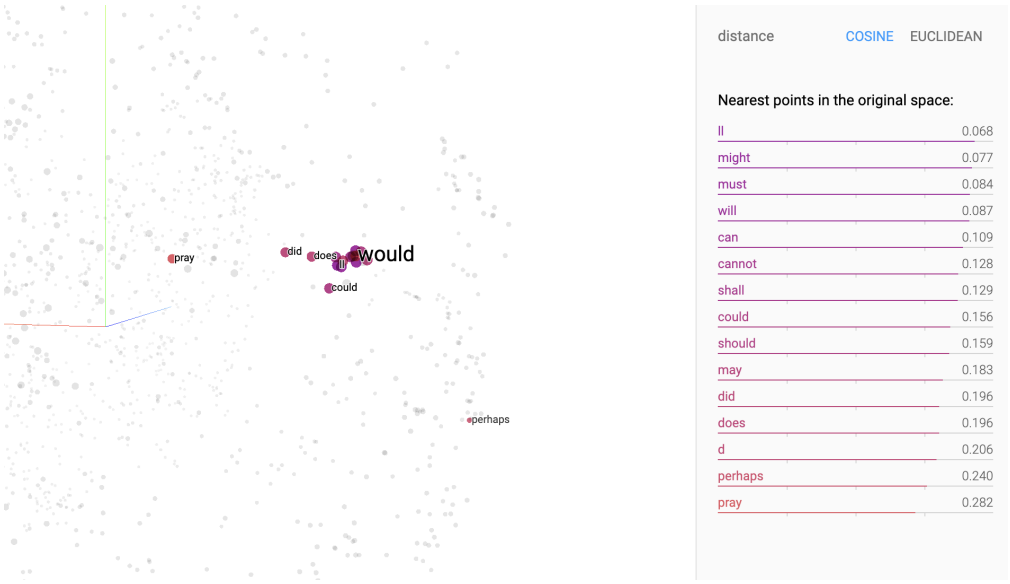


Figure 2: From Embedding projector

1.2 Conjugation

In the Conjugation task, we implemented a similar strategy as in the word embeddings for creating our datasets. We tried out different strategies in terms of context length and ended on using CONTEXT_LENGTH = 6 with 3 heads. Here our intuition was that the model should focus on 3 things in our 6 long sentence. It's necessary to say that we tried to implement with longer as well, and although that could have been better, we settled for less as our models wouldn't be too great after all. At the end, we got quite satisfied with our results given the limited complexity, time and data we used.

$$\text{Attention:} \quad Attention(K, Q, V) = Softmax\left(\frac{QV^T}{\sqrt{d_k}}\right)V \tag{1}$$

$$\text{Multi head attention:} \quad Multihead(Q, K, V) = Concat(head_1, \ldots, head_h)W^O \tag{2}$$

$$\text{where } head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$

$$\text{Positional Encoding:} \quad PE_{(pos,i)} = \begin{cases} \sin\left(\frac{pos}{10000^{i/d_{model}}}\right) & \text{if } i \mod = 0 \\ \cos\left(\frac{pos}{10000^{(i-1)/d_{model}}}\right) & \text{otherwise} \end{cases} \tag{3}$$

Above are the formulas used in our program for the attention model, also given in the report. These formulas were then converted into the classes PositionalEncoding, MLPwithAttention and MultipleAttention in addition to the embedding layer which had to be the first layer in all of our models. Then these could be used as new layers inside our final Bumblebee class (our very simple and not fully implemented transformer model). All the classes were implemented quite literally in terms of the formulas above. The output layer had 12 output features (since we are predicting between 12 words). With target mapping we could convert these numbers into our predicted words. The results will be given in the results section.

For the simple MLP model, we implemented the layers to first fit the embedding dimensions and the output to give 12 features. We also created a simple RNN with two linear layers. For activation

functions, we tried to use GELU instead of ReLu. The reason for why we chose GELU was quite random. Since the ChatGPT2 code was released this week, we looked at it by interested and saw that they used GELU. We didn't know what it was, but we did some research and GELU looked promising, which lead us to implement it as well. More on the hyperparameters in the next section.

To get the predictions, we created a TARGET_MAPPING dictionary. Then we just combined the input with the output to get the predicted words. The outputs will be given below in the results section

1.2.1 Hyperparameters

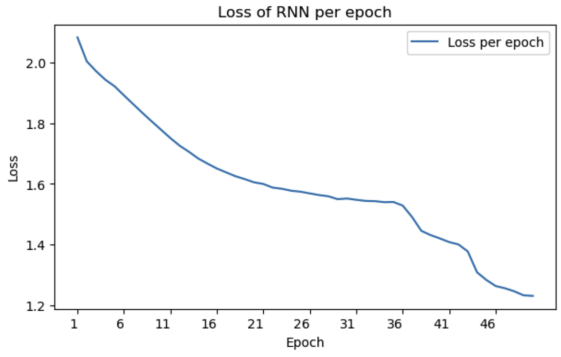
Since we trained our models on embedding dimensions 14 we had to use this as our in our implementions of the three different models. In all of our classes the embedding dim was implemented by:

```
#voca_size, embedding_dim= (1880, 14)
vocab_size, embedding_dim = embedding.weight.shape
self.embedding = nn.Embedding(vocab_size, embedding_dim)
```

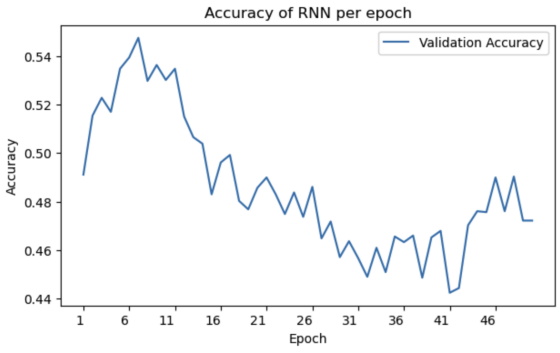
In our RNN model we used 128 as the hidden size with 2 layers. We trained all of our models with Adam optimizers with learning rate = 0.001. For the conjugation task we used quite small batch size, in comparison to the other tasks, with batch size = 32. We also calculated the class weights for the new dataset, which we used as weights inside CrossEntropyLoss to account for distribution of the target classes.

1.2.2 Results and thoughts

After training 50 epochs, we got our best model to the RNN model with 45.8% accuracy on test data. [1] The training process was a little faster than we thought it would be, totaling roughly 20 minutes on T4 GPU in google Colab. We can see that after 10 epochs, our validation accuracy significantly decreased until around 40 epochs, before it started increasing again. For us, this indicates that we could have maybe trained our model for longer and regained some of our accuracy.



(a) Loss function for conjugation



(b) Validation accuracy for conjugation

As for how our model performed, we have the full report in our jupyter notebook, but in the figure below the results can be seen. Although a more comprehensive analysis can be given by the jupyter notebook, we think that where our models performed incorrectly because of the limited context length. For example, in the first incorrectly guess from the figure, he can see the sentence:

[1] We will comment further on what we think of training for 50 epochs in our last section.

"<unk> no one *predict* yet forgotten". Here, even for us humans it would be hard to predict the correct word, simply because we haven't enough information about the sentence to predict. We designed our model to work with 3 words in the front and after, but it is clear that this is not enough from these examples. Thus, one way to improve our model, would not be to train more, but maybe more effective would have been to have longer context windows. Another way to improve this would have been to include more words in our vocabulary, this would have increased the difficulty of the training process, but as we can see in the examples above, 3/4 examples has <**unk**> in it.

```
==================== CORRECT PREDICTIONS ======================
================================================================
True               : the year <unk> was <unk> by a
predicted          : the year <unk> was <unk> by a
================================================================
True               : , vessels had been met by an
predicted          : , vessels had been met by an
====================== WRONG PREDICTIONS =======================
================================================================
True               : <unk> no one was yet forgotten .
predicted          : <unk> no one has yet forgotten .
================================================================
True               : , <unk> men are particularly <unk> .
predicted          : , <unk> men were particularly <unk> .
================================================================
Accuracy of the best conjugation model on test set:
================================================================
= 45.7922%
================================================================
```

Figure 4: Some of our model's predictions where it got the conjugation correct and wrong

1.3 Text generation

In the text generation task, we started by creating a dataset based on the texts provided in the assignment. The datasets were created with context size = 3, with the whole context window before the target word.

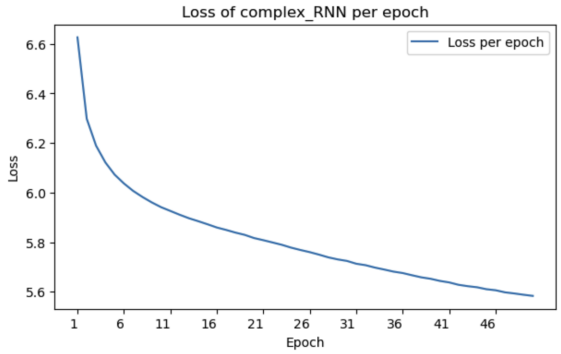
Three RNN model architectures were then defined, each with an increasing complexity. We trained all models for 50 epochs. These models were took the longest to train of all other models (around three hours). After selecting the best model, we implemented the beam search algorithm. This algorithm was used to generate the most probable sentences.

1.3.1 Hyperparameters

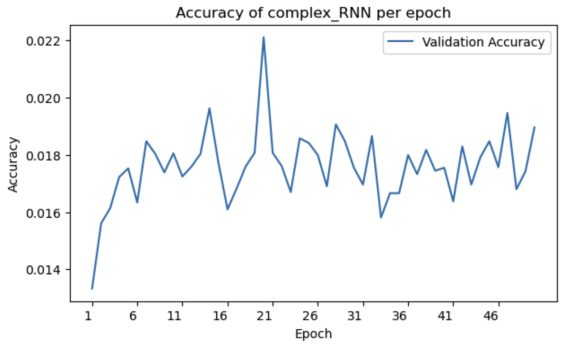
1.3.2 Results and thoughts

In this task, we created a model that could generate text. We can see that from the example sentences that our model performed pretty bad (or good, if you count in how hard it is to understand a language

on only 3 hours of training). It's clear that it has picked up on some patterns. For example, after a question, it tends to answer them with yes. It is also clear that it has trained on books where lots of "dark" things have happened. Our model often tends to say the words *cried, angrily* and *shouted*. This also makes senses, due to books often needs to have an interesting story line. Below is also some plots from our model. Here we can see that our loss went down quickly, but our models' performance on validation data didn't get too good. We think that this again is due to the limited context size in both the embeddings and our model. One could also argue that we could have considered a smaller learning rate, because of the "spikyness" of the validation accuracy between the epochs.



(a) Loss function for text generation



(b) Validation accuracy for text generation

The longer a sentence was, the more random it became. This could be due to the small context size of 3. If we had a context size of 10-15, and trained way longer, it should arguably be better able to see how a sentence is structured, and thus the sentence would not be so random. For short prompts and sentences, however, it was very pretty accurate. You can see an example of this phenomenon in the figure below:

```
Prompt: Long ago there was

Responses with k = 3 and sequence length = 5:

1 long ago there was hope managed matters everyone managed
2 long ago there was hope managed matters everyone loved
3 long ago there was something else than usual method

The most probable response was:
long ago there was something else than usual method
```

(a) Longer sentence (more random)

```
Prompt: Are you happy ?

Responses with k = 3 and sequence length = 1:

1 are you happy ? yes
2 are you happy ? asked
3 are you happy ? exclaimed

The most probable response was:
are you happy ? yes
```

(b) Shorter sentence (less random)

2 Reflecting thoughts

Overall, we are quite satisfied with our approach and solutions. The only thing that we are not so satisfied with is that we clearly trained our models for too long time. Although our loss functions clearly went down the whole time, we did not achieve the same results in terms of validation accuracy. For example, by looking at the plots from the conjugation task, it's clear that we trained for an unnecessary long time. As we discussed above, we think that this could have been because of us not having long enough context length for our models. We saw that, even for us, it could be hard to predict the correct word. Likewise, this applies to our embeddings, which were only trained on

a 4 word context. In retrospect, we think that we maybe trained too long, given these limitations. We also believe that instead of training for 50 epochs, we should instead have implemented another approach where we either implemented early stopping, or that we implemented a solution where we saved the best model in the best epoch. We think that the last method is more "stable", but will obviously take longer time.