Project 3: Sequence models

Simon Vedaa Sebastian Røkholt

1 Introduction

This report aims to explain our approach and design choices for defining, training and evaluating sequence models for three language modelling tasks. Additionally, we will discuss the performance of our models and evaluate our implementation.

For general information and setup guidance, please refer to the README.

1.1 Contributions

There is some overlap, but here is a general overview of what each project member contributed with:

- **Simon Vedaa**: Word embedding and conjugation modelling, loss functions, preprocessing, model training, evaluation and selection, plotting, documentation and project report
- **Sebastion Røkholt**: Generation modelling, beam search, model evaluation, code documentation and project report

2 Task 1: Word embeddings

2.1 Approach and design choices

2.1.1 Tokenization

For tokenization we are using the basic-english tokenizer from `torchtext`. We exclude tokens which are digits, names, and spaces.

2.1.2 Dataset and vocabulary

The training dataset contains 4,384,460 tokens in total. There are 80,135 unique tokens; however, the vocabulary we have constructed for the embedding model only contains 3,110 tokens, as we decided to only include words with a training data frequency greater than 90. Since the usage frequency of words in the English language has a very long tail, 96% of all unique words become unkown tokens. This is likely to greatly affect the performance of the models.

Sequence models require the data to be in a context, target format, so we generated new training, validation and test datasets with a context size of 12. The context size refers to the number of tokens before and after the target. The generated training dataset contains 3,225,478 context-target pairs. The total context then becomes 2*context_size. We have excluded the unknown token (`<unk>`) and punctuations from the targets.

2.1.3 CBOW Architecture

We have defined two CBOW architectures, `CBOW` and `CBOWDeep`. Both take the vocab size, context size, and embedding dimension as hyperparameters. The first layer is a `nn.Embedding` for both.

`CBOW` contains two fully connected layers after the embedding layer, while `CBOWDeep` contains four. We use ReLU as the activation function for all layers except for the output layer, which uses a log_softmax function.

2.2 Training and Selection

2.2.1 Training data

In addition to the 13 books provided on MittUiB, we downloaded these 23 books from The Gutenberg Project:

Title	Author
Romeo and Juliet	William Shakespeare
The Tragedy of King Lear	William Shakespeare
Othello	William Shakespeare
Macbeth	William Shakespeare
Hamlet	William Shakespeare
The Time Machine	H. G. Wells
The Last Question	Isaac Asimov
Notes from the Underground	Fyodor Dostoyevsky
War of the Worlds	H. G. Wells
Alice's Adventures in Wonderland	Lewis Carroll
Frankenstein	Mary Wollstonecraft Shelley
Moby Dick	Herman Melville
The Importance of Being Earnest	Oscar Wilde
The Great Gatsby	F. Scott Fitzgerald
The Picture of Dorian Gray	Oscar Wilde
Metamorphosis	Franz Kafka
A Tale of Two Cities	Charles Dickens
Jane Eyre: An Autobiography	Charlotte Brontë
Treasure Island	Robert Louis Stevenson
The Hound of the Baskervilles	Arthur Conan Doyle
Gulliver's Travels	Jonathan Swift
Paradise Lost	John Milton
A Doll's House : a play	Henrik Ibsen

2.2.2 Model training and selection

For training we are using `Adam` as the optimizer, and `nn.NLLLoss` as the loss function. The weights for the vocabulary are passed to the loss function.

The function `src.utils.train` is used for training in all three tasks.

We used a batch size of 64, and trained 15 epochs for every run.

We have implemented a simple grid search, where, for each architecture, we train it for every defined hyperparameter combination. The model with the highest accuracy is chosen.

Parameters used in grid search:

Learning rate	Embedding dimension
0.001	16
0.001	20
0.008	16
0.008	20

2.3 Evaluation

These were the chosen parameters and architecture:

Architecture	Learning rate	Embedding dim
CBOWDeep	0.001	16

Figure 1: Training and validation loss of selected CBOW model

The selected model got a test accuracy of 0.76%. As the training data includes a lot of unkown words, we expected the performance to be low. When compared to random guesses, which had an accuracy of 0.032%, our embedding model performs slightly better.

Figure 2: Training and validation accuracy of selected CBOW model

2.3.1 Cosine similarity

As expected, each word is most similar to itself, seen as the yellow diagonal linear through the similarity matrix in Figure 3. Additionally we can see an interesting grid pattern emerge in the similarity matrix. Some words are less similar to most words in the vocabulary, while others are similar to many words.

Words with higher frequencies appear more often in the context of target words, and could possibly explain this. Words with the lowest possible frequency of 90, rarely appear in context with other words, and thus becomes less similar to most words.

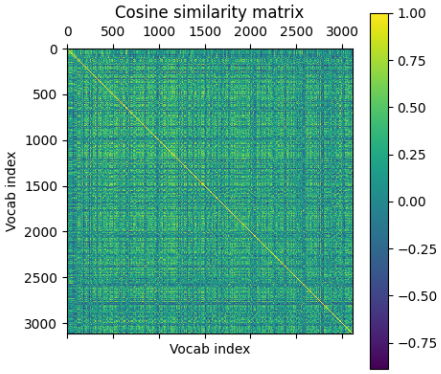

Figure 3: Cosine similarity matrix

2.3.2 Visualization of embedding space

The embedding seems to capture the word class and some semantic qualities. In Figure 4, the embeddings of man and woman seems to represent a difference in roles. The word `man` is similar to `action`, `priest`, and `position`, while `woman` is similar to words related to `family`, like `son`, `man`, `friend`, and `girls`. The verbs `be` and `speak` in Figure 5, are both grouped with other verbs, and are close to verbs of similar meaning, such as `speak` -> `say` and `think`, and `be` -> `am` and `become`. In Figure 6, we can see that the word `castle` is grouped with similar objects which could be found in a `castle` like `sword`, `bow`, and `guns`, while the word `me` is grouped with other pronouns such as `him`, `it`, and `us`.

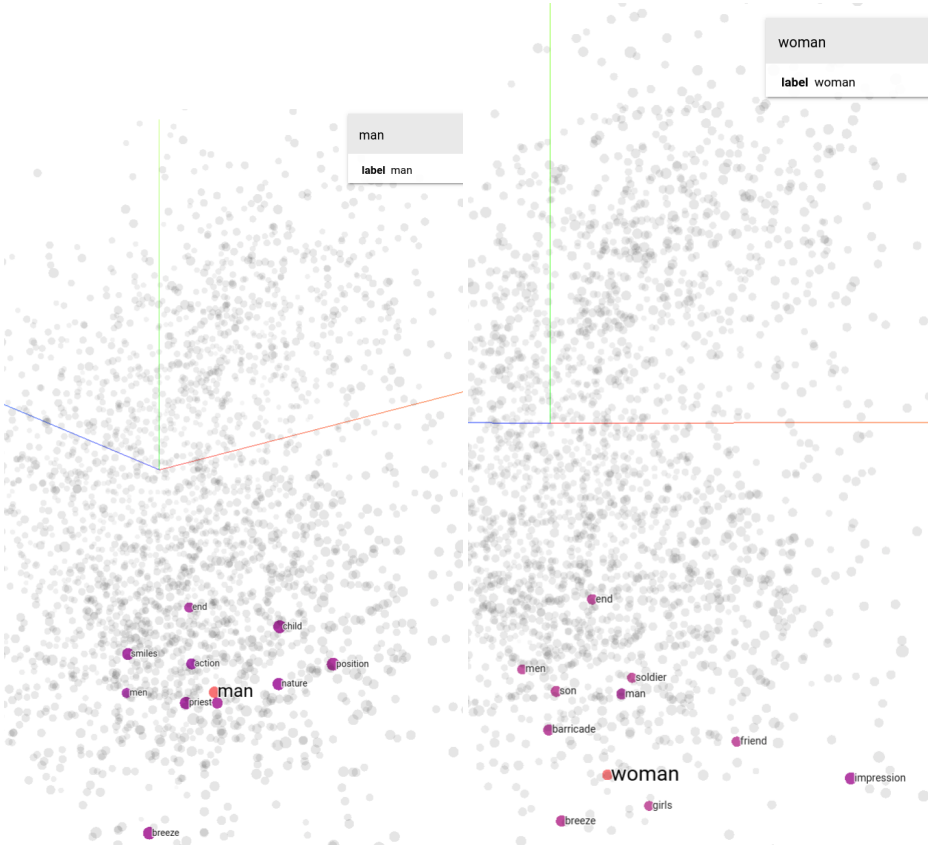

Figure 4: Embedding of man and woman, and their 10 most similar words

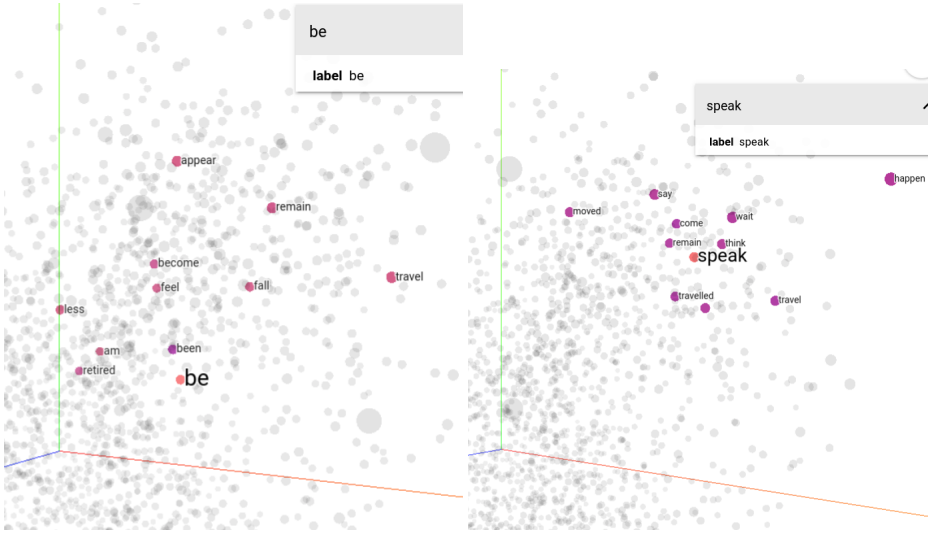

Figure 5: Embedding of be and speak, and their 10 most similar words

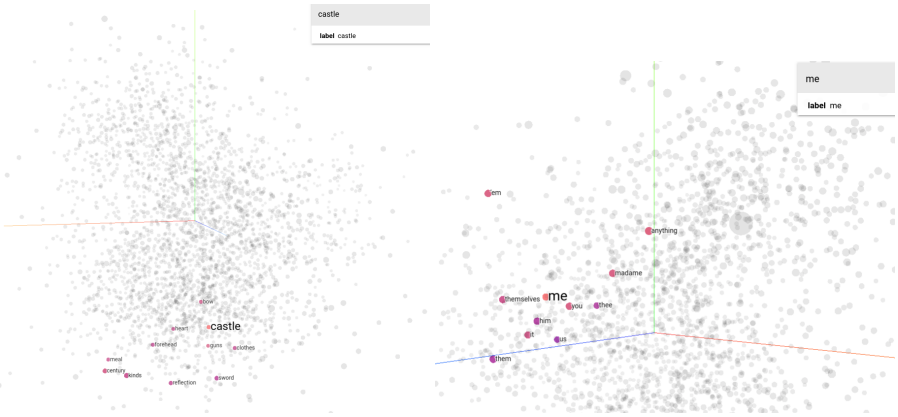

Figure 6: Embedding of castle and me, and their 10 most similar words

3 Task 2: Conjugation of *have* and *be*

3.1 Approach and design choices

3.1.1 Dataset

The conjugation dataset is a subset of the generated embedding dataset, such that we only include the words/tokens *be*, *am*, *are*, *is*, *was*, *were*, *been*, *being*, *have*, *has*, *had* and *having* as targets. The training dataset contains 190,515 context, target pairs.

In this case, the *context size/sequence length* refers to the total size of the context window.

3.1.2 Model architectures

We have defined three network architectures for the conjugation task; `SimpleMLP`, `AttentionMLP`, and `ConjugationRNN`. Each architecture has an embedding layer as its first layer, which is frozen during initalization. Additionally, they each includes a parameter for the max length of the input sequence.

`SimpleMLP` contains three fully connected layers after the embedding layer. The first linear layer takes an input of size `embedding_dim*max_len`, while the final layer has an output size of 12, corresponding to the number of possible conjugations. All sizes of the layers in between are adjustable. `ReLU` is used as activation function for the hidden layers, while the final layer simply outputs the logits without any activation function.

The `AttentionMLP` architecture contains a positional encoding layer, a multi-head attention layer, and a fully connected layer. The multi-head attention layer is implemented by chaining multiple `SingleHead` layers using `nn.ModuleList`, and then concatenating their outputs and passing them through a fully connected layer. Number of heads and the size of key, query, and value matrices are adjustable.

`ConjugationRNN` contains a single RNN layer and a fully connected layer. The size and number of hidden layers in the RNN are adjustable.

3.2 Training

Here we are using `Adam` for the optimizer and `nn.CrossEntropyLoss` as the loss function.

We use the same grid search approach as in the previous task. For each model architecture we train with all hyperparameter combinations. Each architecture has model-specific hyperparameters in addition to the common ones. Additionally, we measure the average training time for each model architecture.

We used a batch size of 64 for all models, trained 30 epochs for every run, and selected the model with the highest accuracy.

`SimpleMLP`:

l1	l2
128	32
256	64
256	256

`AttentionMLP`:

n_heads	W size
4	8
8	16
16	20

`ConjugationRNN`:

num_hidden	num_layers	dropout
8	4	0
16	8	0.1
20	16	0.1

Common hyperparameters:

Learning rate
0.008
0.001
0.0005

3.3 Results

These were the selected hyperparameters and architecture:

Architecture	Learning rate	n_heads	W size
AttentionMLP	0.0005	16	20

The chosen model got a test accuracy of 61%. This is not too bad, as random guessing would yield an accuracy of 8.33%. In addition, the chosen model does not seem to overfit or underfit.

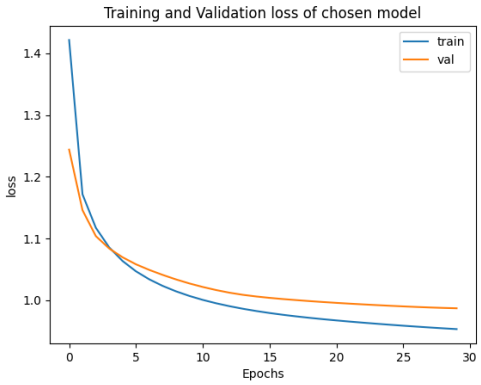

Figure 7: Training and validation loss of selected model

Figure 8: Training and validation accuracy of selected model

For each model, we time the training process and compute the loss and accuracy for both training and validation sets. As seen in Figure 9, `AttentionMLP` is by far the slowest architecture to train, followed by `ConjugationRNN`, and then `SimpleMLP`. This is largely due to the computational complexity of training attention mechanisms.

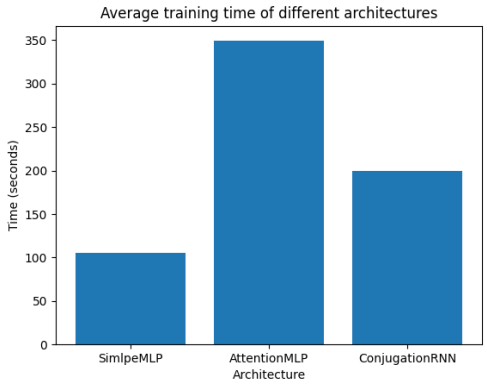

Figure 9: Average training times for architectures

4 Task 3: Text generation

4.1 Approach and design choices

4.1.1 Dataset

For the task of text generation, we generated a new dataset of context-target pairs where the context only consists of `context_size` number of tokens *before* the target. The code for creating this dataset is located in the `create_dataset` function in the generation.ipynb notebook, which takes a tokenized dataset as input.

Generation dataset type	Size (number of tokens)
Training data	3,225,469
Validation data	37,937
Test data	89,210

4.1.2 Model architectures

We wanted to compare RNN and LSTM-based architectures, because we were curious to see whether the LSTMs would outperform the "basic" RNNs for this task. Both architecture variants were similar, as they consisted of our previously trained embedding layer (with frozen weights) of size `vocab_size`*`embedding_dim`, one or multiple recurrent (RNN/LSTM) layers, and a fully connected output layer of size `num_hiddens`*`vocab_size`.

The hyperparameters were the same as for task 2; the number of hidden recurrent layers, the number of hidden units and a dropout parameter, as well as the optimizer's learning rate parameter.

4.2 Model training

The training loop for the text generation models is identical to the one we used to train the conjugation models. As before, we are using `Adam` for the optimizer and `nn.CrossEntropyLoss` for the loss function with a batch size of 64. We performed a grid search over hyperparameters where we trained the models for 20 epochs and ranked them on their target token prediction accuracy.

num_hidden	num_layers	dropout
8	4	0
16	8	0.1
20	16	0.1

Learning rate
0.008
0.001
0.01
0.0005

4.3 Model selection and evaluation

4.3.1 Comparing RNN and LSTM

The average accuracies during grid search for the RNN- and LSTM-based models were:

	RNN	LSTM
Train accuracy	9.7%	10.15%
Validation accuracy	9.6%	10.10%

The LSTM-based models performed slightly better than the RNN-based ones, which we think is likely due to the fact that LSTMs are more robust, e.g. they can handle much longer input sequences because they don't suffer from exploding gradients. In the context of natural language processing, it is very beneficial to be able to "remember" information from the start of a long input sequences.

4.3.2 Selection

Overall, the grid search determined that these were the optimal hyperparameters:

Architecture	Learning rate	num_hiddens	num_layers	dropout
GenerativeLSTM	0.0005	16	8	0.1

4.3.3 Beam search implementation

The function `beam_search` in `utils.py` is a basic implementation of beam search with length normalization. It returns an ordered list of the candidate sequences that received the best normalized scores (log probability) across all candidate sequences evaluated over the entire search process. The max_len and beam_width parameter restricts the size of the search tree. The generated sequences can therefore be of a length from `input_sequence + 1` to `input_sequence + max_len`, though a length penalty > 0.5 tends to result in sequences of maximum length.

4.3.4 Examples of generated sequences

Max generation length: 10, beam width: 3, length penalty: 0.4, generated sequences: 1

Prompt: `the cat jumped over`

Generated sequence: `the king of the country of the country of the`

Prompt: `what is the meaning of`

Generated sequence: `the`

Prompt: `i have never`

Generated sequence: `been`

Prompt: `the woman was sitting`

Generated sequence: `in the country of the country of the`

Prompt: `as i opened the`

Generated sequence: `king of the country of the country of the`

Prompt: `to be or not to be ?`

Generated sequence: `the king of the country of the`

Prompt: `a king and queen once upon a time`

Generated sequence: `of`

Prompt: `suddenly the door opened and`

Generated sequence: `to the country of the country of the`

Prompt: `in the morning we`

Generated sequence: `will not have been in the country of the`

Prompt: `the meaning of life is`

Generated sequence: `a man of the country of the country of the`

4.3.5 Evaluation

This GenerativeLSTM model achieved an accuracy of 15.92% when evaluated on the task of next-token prediction. When looking at the training and validation loss, we see that the model converged nicely, with a slightly higher validation loss that seems to plateu after 12 epochs. The validation accuracy is higher than the training accuracy, so overall, these plots indicate that the model neither overfits nor underfits on our small dataset. However, the generation results are rather poor, as the model assigns a limited set of tokens a much higher log likelihood than the rest of the vocabulary. We therefore see that the model often generates phrases like “to the”, “of the” and “the country” in loops.

Figure 10: Training and validation loss of selected text generation model

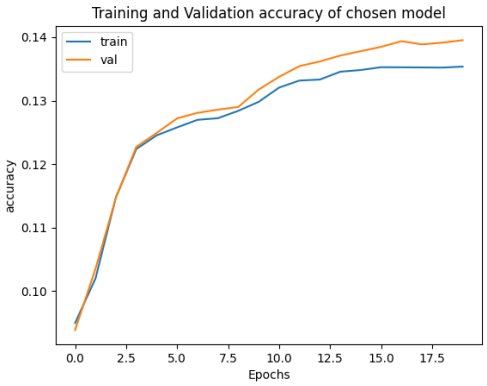

Figure 11: Training and validation accuracy of selected text generation model

As expected, the model's relatively low accuracy supports the claim that effective language models require both much larger training datasets and a significantly higher number of model parameters. However, the accuracy metric alone does not provide a good estimate of the model's ability to generate good sequences, as we are only testing to see if the model can predict the correct target word from the preceding context/prompt/input sequence. It would be more appropriate to calculate the model's Perplexity, along with the BLEU or METEOR score and perform an evaluation based on these results.