INF 265 - Project 3 Report
Sander Marx, Simen Sørensen

The code discussed in this report can be found in the delivered jupyter notebook **INF265_Project3_Group_11.ipynb.** This technical report will discuss questions from section 5.

Division of labour:
Sander worked on Word embeddings, attention mechanism and Conjugating *be* and *have*
Simen worked on text generation and beam search.
Both Sander and Simen worked on the report.

This report aims to give an explanation of our approach and design choices as well as reporting the different hyperparameters and performance of our models. It is divided based on the tasks from 2.1 to 2.3. The report will answer the questions:

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 the expected results, try to give potential reasons that would explain why your code does not work and/or your results differ. Give concrete examples of cases where your model does not work as expected.

2.1 Word embeddings

For reading the text files, tokenizing them and building a vocabulary we reused some code and functions from from section 1 in the provided **06-Text_data.ipynb** file. This helped us to find the total number of words, distinct words and words that occur at least 100 times as seen in figure 1. We also found the most common words in the vocabulary and they were as expected common words like, ‘the’, ‘and’, ‘of’ etc.

```
Total number of words in the training dataset:      2684706
Total number of words in the validation dataset:    49526
Total number of words in the test dataset:          124152
Number of distinct words in the training dataset:   52105
Number of distinct words kept (vocabulary size):    1880
```

Figure 1: Word count

CBOW

For the Continuous bag of words (CBOW) model we used a simple model consisting of an embedding layer and a simple linear layer. In the forward pass we sum all the extracted embeddings along the first dimension and then return a log softmax to the linear outputs to get log probabilities so that we can use nn.NLLLoss() as our loss function.

Model Selection

We define a train() function and a compute_accuracy() function so that we can reuse the code in further sections. We chose to train 4 models with a relatively small number of epochs, this was done due to time constraints and ideally we would test more models with a bigger number of epochs. We also used a relatively low embedding dimension size of 16 as the task description asked for. Because of this our accuracies became very small.

Constant parameters used by all 4 models:
context size = 3
number of epochs: 5
Loss function: nn.NLLLoss()
Optimizer = optim.Adam()
embedding dimension = 16

Hyper parameters:

Model	Batch size	Learning rate	Validation accuracy
CBOW Model 1	32	0.1	8.4%
CBOW Model 2	32	0.01	16.9%
CBOW Model 3	64	0.1	10.7%
CBOW Model 4	64	0.01	**17.05%**

Table 1: CBOW models Hyperparameters and validation accuracy

Our selected best model was model 4 with a batch size of 64 and a learning rate of 0.01. This model had a validation accuracy of 17.05% and when we tested this model on unseen test data we got a **test accuracy of 21.14%.**

Cosine similarity matrix

After selecting the best model based on validation performance we computed the cosine similarity matrix based on the trained embeddings. We then found the 10 most similar words to the words: [me, white, man, have, be, child, yes, what]. We got some good results like for the word ‘have’ and some results that were not as good, like with for example the word ‘man’. Figure 2 shows the 10 most similar words to ‘man’ and ‘have’ including the word itself. As we can see ‘have’ produces many similar words like ‘has’, ‘had’, ‘having’, ‘hast’). While

‘man’ produces not as many similar words and gives us words like ‘dog’ and ‘mother’ which should not be among the top most similar words.

```
Words most similar to 'man':
  Nr. 1: man
  Nr. 2: child
  Nr. 3: girl
  Nr. 4: dog
  Nr. 5: woman
  Nr. 6: doctor
  Nr. 7: officer
  Nr. 8: bondes
  Nr. 9: person
  Nr. 10: mother
  Nr. 11: enemy
Words most similar to 'have':
  Nr. 1: have
  Nr. 2: has
  Nr. 3: ve
  Nr. 4: nearly
  Nr. 5: had
  Nr. 6: having
  Nr. 7: hast
  Nr. 8: produce
  Nr. 9: give
  Nr. 10: paid
  Nr. 11: make
```

Figure 2: Similar words to ‘man’ and ‘have’ based on cosine similarity

Embedding space visualized

We saved our vocabulary and embedded vectors to .tsv files and uploaded them to https://projector.tensorflow.org/ to better visualize our embeddings. Figure 3, 4 and 5 shows the visualized embedding space and cluster around the words ‘gives’, ‘two’ and ‘gazed’.

The word ‘gives’ has data points that are more spread but seem to be pointing in the same direction. The embedding shows words like ‘gave’, ‘give’, ‘offered’, ‘sent’, ‘giving’ that are very similar.

The word ‘two’ seems to have a cluster that is less spread, and we find that the cluster includes different numbers like ‘four’. ‘eight’, ‘fifteen’, ‘seven’, ‘nine’ etc. There are also words like ‘several’ and ‘many’ in this cluster, so this cluster seems to represent quantity.

The word ‘gazed’ gives a cluster more like the ‘gives’ cluster, where the points are more spread out but seem to be pointing in the same direction. Here we see similar words like ‘glanced’, ‘looked’, ‘gazing’, ‘looking’, ‘stared’, ‘glancing’ and ‘glance’.

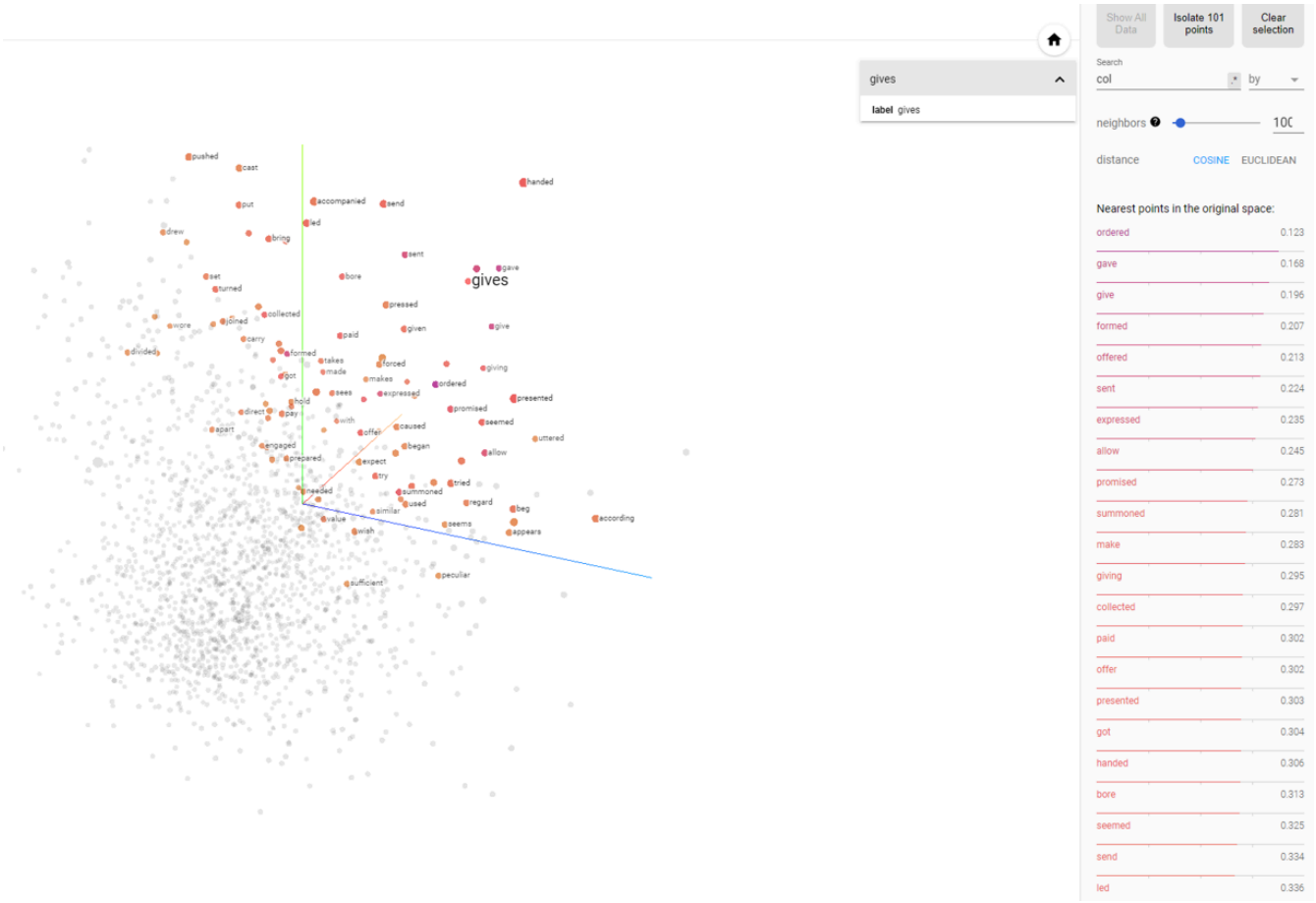

Figure 3: Visualized embedding space and cluster around the word, ‘gives’

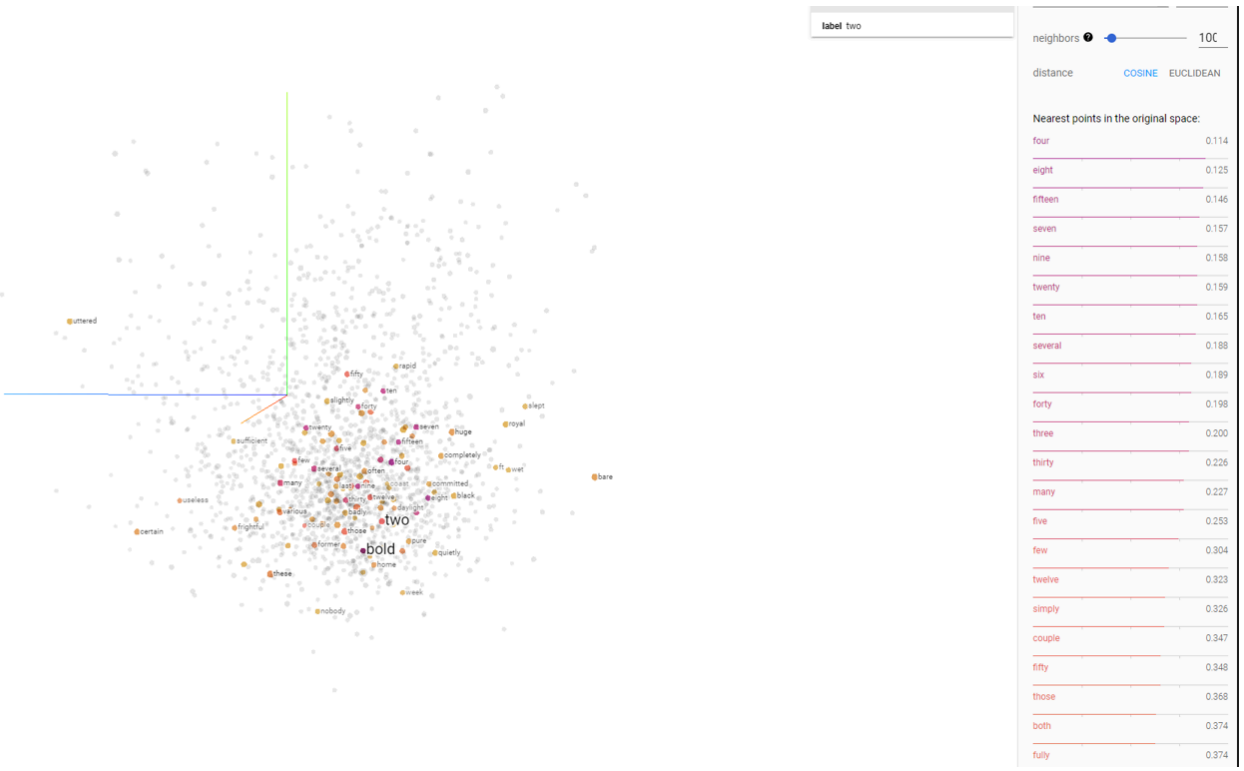

Figure 4: Visualized embedding space and cluster around the word, 'two'

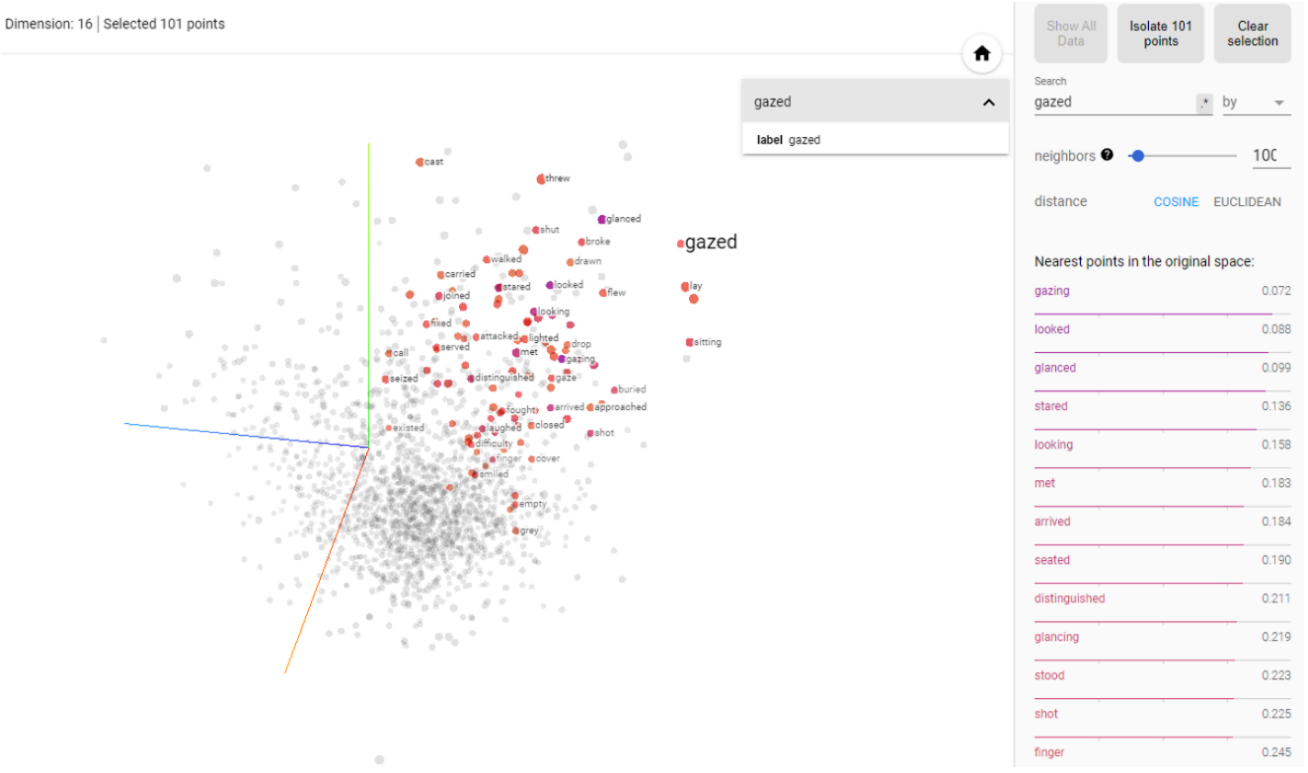

Figure 5: Visualized embedding space and cluster around the word, 'gazed'

2.2 Conjugating *be* and *have*

For this task we created context target datasets so that the targets were [be, am, are, is, was, were, been, being, have, has, had, having]. We then defined 3 different neural network models: Simple MLP, MLP with a single attention layer and RNN. In our models we made it so we could use our pretrained embeddings from 2.1 using nn.Embedding.from_pretrained() making sure we freeze the embeddings so that they do not change during training.

MLP

We defined a MLP consisting of one hidden layer followed by a ReLU activation, and an output layer that maps to the number of classes considering both the left and right context around the target word, doubling the input size to the first fully connected layer. Our hidden layer consisted of 128 neurons and we had an output of 12 since we have 12 classes to choose from ranging from 0-11. Figure 6 shows the implementation.

```python
# Simple MLP Model using the pretrained embeddings
class MLP(nn.Module):
    def __init__(self, pretrained_embeddings, num_classes, context_size, hidden_dim=128):
        super(MLP, self).__init__()
        self.embeddings = nn.Embedding.from_pretrained(pretrained_embeddings, freeze=True)
        (_, embedding_dim) = pretrained_embeddings.shape
        self.fc1 = nn.Linear(embedding_dim * context_size *2, hidden_dim) #Times 2 to get context around the target
        self.fc2 = nn.Linear(hidden_dim, num_classes)
        self.relu = nn.ReLU()

    def forward(self, inputs):
        embeddings = self.embeddings(inputs).view(inputs.shape[0], -1)
        x = self.relu(self.fc1(embeddings))
        x = self.fc2(x)
        return x
```

Figure 6: Simple MLP model with a single hidden layer

MLP with attention

For the MLP with attention layer we had to define our own attention layer using pytorch's nn modules. Our attention consisted of 3 modules: SimpleAttention() , MultiHeadAttention() and finaly the MLPAttention() module. Figures 7 and 8 show the implementation.
SimpleAttention() took care of the simple head dot-product self-attention.
MultiHeadAttention() took care of the Multi-head attention.
MLPAttention() combined positional encoding with simple head dot-product self-attention and Multi-head attention to create our MLP with a single attention layer.

```python
# 4.2 Simple-head dot-product self-attention
class SimpleAttention(torch.nn.Module):
    def __init__(self, emb_dim, p):
        super(SimpleAttention, self).__init__()
        self.emb_dim = emb_dim
        self.p = p
        #weight matrices
        self.Wq = torch.nn.Parameter(torch.randn(emb_dim, p))
        self.Wk = torch.nn.Parameter(torch.randn(emb_dim, p))
        self.Wv = torch.nn.Parameter(torch.randn(emb_dim, p))

    def forward(self, Xf):
        # Compute queries, keys, values
        Q = torch.matmul(Xf, self.Wq)
        K = torch.matmul(Xf, self.Wk)
        V = torch.matmul(Xf, self.Wv)
        # Compute the dot products of Q and K for the attention scores
        scores = torch.bmm(Q, K.transpose(-2, -1)) / np.sqrt(self.p)
        # Apply softmax to get the attention weights
        attention_weights = F.softmax(scores, dim=-1)
        # Compute the output of the attention layer
        h = torch.bmm(attention_weights, V)
        return h

# 4.3 Multi-head attention
class MultiHeadAttention(nn.Module):
    def __init__(self, emb_dim, p, n_head):
        super(MultiHeadAttention, self).__init__()
        self.n_head = n_head
        self.attention_heads = nn.ModuleList([SimpleAttention(emb_dim, p) for _ in range(n_head)])
        self.Wo = nn.Linear(p * n_head, emb_dim)

    def forward(self, Xf):
        heads = [head(Xf) for head in self.attention_heads]
        heads_concat = torch.cat(heads, dim=-1)
        H = self.Wo(heads_concat) # Combine heads
        return H
```

Figure 7: SimpleAttention() and MultiHeadAttention() modules used in MLP with attention

```python
# MLP Model with attention layer
class MLPAttention(nn.Module):
    def __init__(self, pretrained_embeddings, p, n_heads, max_len, num_classes):
        super(MLPAttention, self).__init__()
        self.embeddings = nn.Embedding.from_pretrained(pretrained_embeddings, freeze=True)
        (_, emb_dim) = pretrained_embeddings.shape
        #Positonal encoding with max_len*2 for context around target
        self.pos_encoding = positional_encoding(max_len*2, emb_dim).to(device)
        #Multihead with embedding dim
        self.attention = MultiHeadAttention(emb_dim, p, n_heads)
        self.fc = nn.Linear(emb_dim, num_classes)

    def forward(self, input_ids):
        embeddings = self.embeddings(input_ids)
        # Add positional encoding
        embeddings += self.pos_encoding[:embeddings.size(1), :]
        # Apply attention
        attention_output = self.attention(embeddings)
        # Apply MLP layers
        x = F.relu((attention_output.mean(dim=1)))
        logits = self.fc(x)
        return logits
```

Figure 8: MLP with positional encodings and a single attention layer

RNN

Our RNN consisted of our pretrained embeddings, a nn.RNN() layer and then finally a linear layer. In the forward pass we pass the embeddings to the RNN layer and then use the hidden state from the last timestep to get a summarized information of our passed sequence data. We then pass this info into our fully connected layer to classify one of our 12 classes. Figure 9 shows the implementation.

```python
# RNN Model using pretrained embeddings
class RecurrentNN(nn.Module):
    def __init__(self, pretrained_embeddings, hidden_size, num_classes):
        super(RecurrentNN, self).__init__()
        (vocab_size, embedding_dim) = pretrained_embeddings.shape
        self.embeddings = nn.Embedding.from_pretrained(pretrained_embeddings, freeze=True)
        self.rnn = nn.RNN(embedding_dim, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        embedded = self.embeddings(x)
        output, hidden = self.rnn(embedded)
        # We use the hidden state from the last time step
        out = output[:,-1,:]
        out = self.fc(out)
        return out
```

Figure 9: Recurrent neural network

Model selection

We chose to train 12 models with a relatively small number of epochs, this was again done due to time constraints and ideally we would test more models with a bigger number of epochs. We trained each of the 3 models with 4 different combinations of hyperparameters, learning rate and batch size. For the MLP with attention models we kept a constant p = 32 and number of heads = 12, also for the RNN models we kept a constant hidden size = 12. Table 2 shows the value of the tested hyperparameters.

Constant parameters used by all models:
context size = 3 on both sides
number of epochs :8
Loss function: nn.CrossEntropyLoss()
Optimizer = optim.Adam()

Constant parameters used by MLP with attention model:
p = 32
number of heads = 12

Constant parameters used by RNN model:
hidden size = 12

Hyper parameters:

Model	Batch size	Learning rate	Validation accuracy
MLP Model 1	32	0.1	38.61%
MLP Model 2	32	0.01	57.22%
MLP Model 3	64	0.1	30.12%
MLP Model 4	64	0.01	**57.72%**
MLP with Attention Model 1	32	0.1	25.14%
MLP with Attention Model 2	32	0.01	43.71%
MLP with Attention Model 3	64	0.1	25.14%
MLP with Attention Model 4	64	0.01	51.43%
RNN Model 1	32	0.1	18.42%
RNN Model 2	32	0.01	50.93%
RNN Model 3	64	0.1	18.80%
RNN Model 4	64	0.01	50.69%

Table 2: MLP, MLP with attention and RNN models Hyperparameters and validation accuracy.

Our selected best model was simple MLP model 4 with a batch size of 64 and a learning rate of 0.01. This model had a validation accuracy of 57.72% and when we tested this model on unseen test data we got a **test accuracy of 57.22 %.**

2.3 Text generation

For this task we created context target datasets the same way we did in task 2.1. We then defined our RNN (which has a very similar structure to RecurrentRNN in section 2.2) and trained 4 different models and chose the final model based on validation data. Note that we used nn.NLLLoss as our loss function, however using CrossEntropyLoss without the logsoftmax layer in our RNN structure would be the same.

Constant parameters used by all models:
context size = 15
number of epochs : 5
hidden_size = 12
Loss function: nn.NLLLoss()
Optimizer = optim.Adam()

Hyper parameters:

Model	Batch size	Learning rate	Validation accuracy
TextRNN Model 1	512	0.01	20.76%
TextRNN Model 2	512	0.001	20.63%
TextRNN Model 3	1024	0.01	**20.77%**
TextRNN Model 4	1024	0.001	20.52%

Table 3: RNN for text generation models Hyperparameters and validation accuracy

Ideally we would like to try more hyperparameter values, however due to time constraints we ended up with training 4 different models with batch sizes 512 and 1024, and learning rate 0.01 and 0.001. Lower batch sizes than 512 led to slower training times, so to be practical we chose 512 and 1024 as batch size values. Given more time we would experiment with lower batch sizes, other learning rate values and more epochs.

Our selected best model was model 3 with a batch size 1024 of and a learning rate of 0.01. This model had a validation accuracy of 20.77% and when we tested this model on unseen test data we got a **test accuracy of 24.61%**.

Beam search algorithm

For our selected best model, we run the beam search algorithm to generate text word for word. Starting with a list of initial words, the algorithm predicts the next words using the model. It only keeps the top choices determined by the beam width. n_words is the number of words we want to predict in addition to the initial words.

Note that we decided to exclude the “<unk>” token to get a more sensible output. When we included it, we got multiple “<unk>” which made it hard to evaluate if the sentences had some logic to them or not. Removing <unk> can lead to potentially less context for the words, but we are satisfied with the results we got.

Testing the algorithm:

```
initial_words = ["what", "to", "do"]
generated_text = beam_search(best_model, initial_words, vocab, n_words=16, beam_width=5)
print(generated_text)
```

✓ 0.0s

```
what to do you know you . i am to see it , said . he had been to
```

```
initial_words = ["white", "black", "red"]
generated_text = beam_search(best_model, initial_words, vocab, n_words=16, beam_width=5)
print(generated_text)
```

✓ 0.1s

```
white black red eyes . he had been in the room . he had been in the middle of
```

```
initial_words = ["the", "king", "and", "i"]
generated_text = beam_search(best_model, initial_words, vocab, n_words=17, beam_width=10)
print(generated_text)
```

✓ 0.1s

```
the king and i did not know that it is you know you . i am to see it , and
```

We notice that there are some logical sense to the sentences, but it struggles to keep a meaningful context.

With <unk> token:

```
initial_words = ["what", "to", "do"]
generated_text = beam_search(best_model, initial_words, vocab, n_words=16, beam_width=5)
print(generated_text)
```

✓ 0.0s

```
what to do to <unk> , and <unk> , and <unk> <unk> , and <unk> <unk> , and <unk>
```

```
initial_words = ["the", "king", "and", "i"]
generated_text = beam_search(best_model, initial_words, vocab, n_words=10, beam_width=10)
print(generated_text)
```

✓ 0.0s

```
the king and i <unk> the <unk> of the <unk> of the <unk> <unk>
```

With <unk> token included, it also struggles to make logical sense in the sentences, since it outputs <unk> most of the time with no meaning.

Conclusion and further improvement

In conclusion we managed to train several networks and do model selection for all tasks, even though the results were not the best. Our implementations are not perfect and there definitely are some improvements we could consider given enough time. Most importantly we could have tested with several different hyperparameters and more epochs to get better performance. Given enough time we could also try different model architectures.