Deep Learning - Assignment 3

Carlos Vega and Marcos Alonso

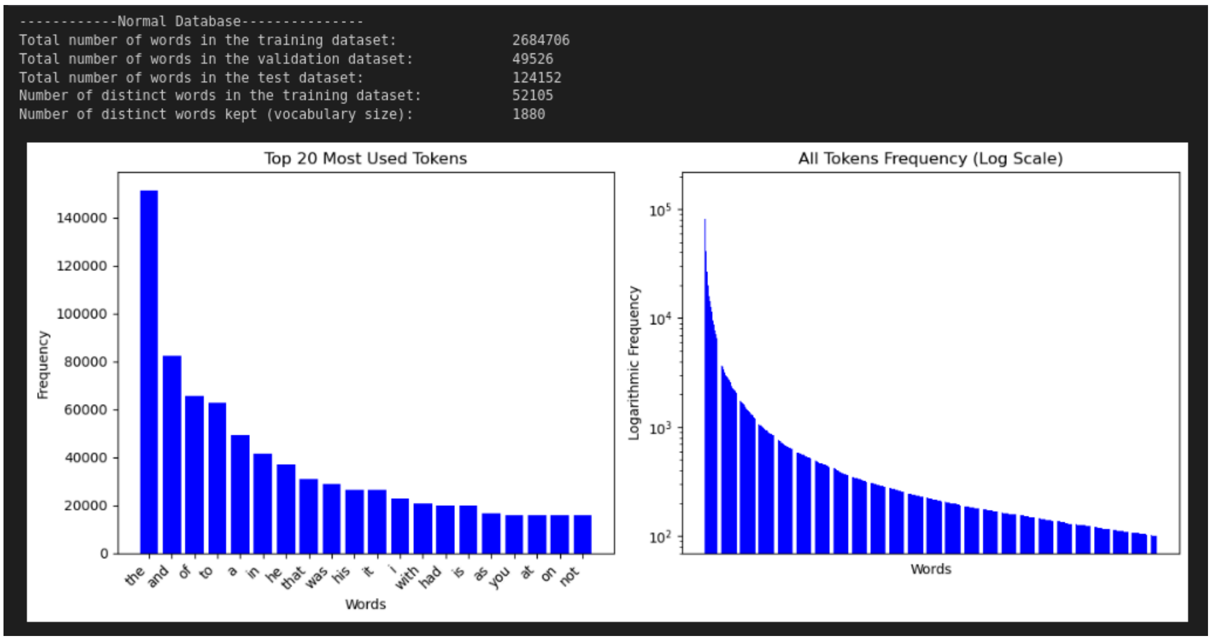
Text Prediction and Generation

Introduction

The goal of this project is to create a neural network that would be able create relations between words, telling how similar they are to each other, then be able to conjugate the verbs be and have and at the last part generate the next words given the start of a sentence. The models will be trained with different books, using the sentences in them.

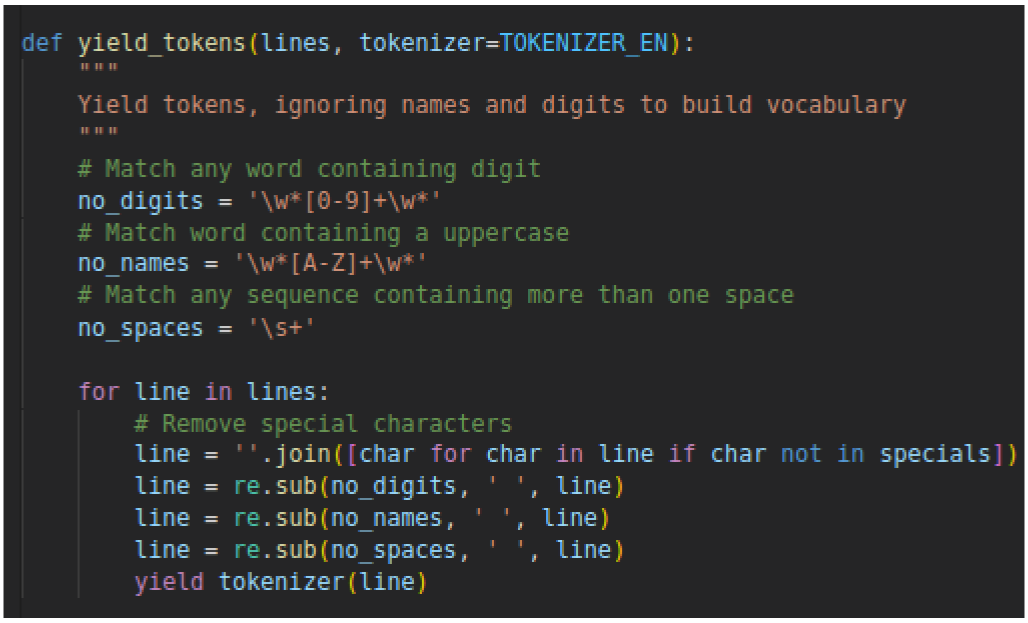
1. Word embedding

If we take a look at our database we see there are many different words and the frequency of each of them is very different as we can see at the second graph (Note the scale is logarithm).



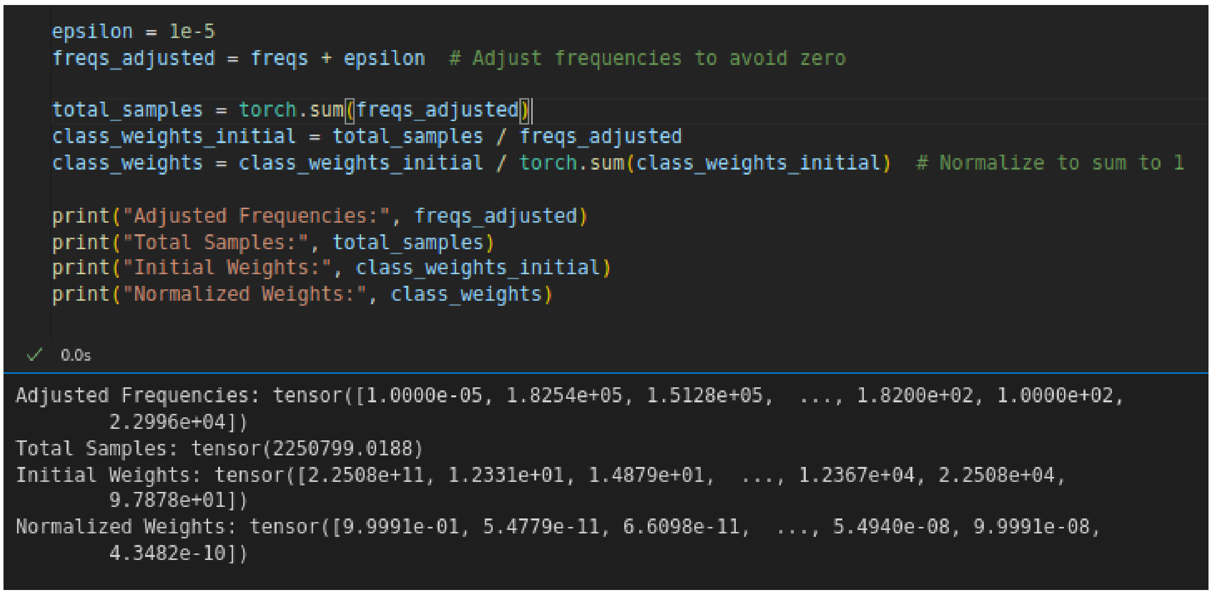
```
------------Normal Database--------------
Total number of words in the training dataset:            2684706
Total number of words in the validation dataset:          49526
Total number of words in the test dataset:                124152
Number of distinct words in the training dataset:         52105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

This is why we will only keep the words that appear at least 100 times, and we will also get rid of any words with numbers or names, keeping the vocabulary simple.



```python
def yield_tokens(lines, tokenizer=TOKENIZER_EN):
    """
    Yield tokens, ignoring names and digits to build vocabulary
    """
    # Match any word containing digit
    no_digits = '\w*[0-9]+\w*'
    # Match word containing a uppercase
    no_names = '\w*[A-Z]+\w*'
    # Match any sequence containing more than one space
    no_spaces = '\s+'

    for line in lines:
        # Remove special characters
        line = ''.join([char for char in line if char not in specials])
        line = re.sub(no_digits, ' ', line)
        line = re.sub(no_names, ' ', line)
        line = re.sub(no_spaces, ' ', line)
        yield tokenizer(line)
```

We will also make every word have a preset weight, depending of the amount of appearances it has in the text.

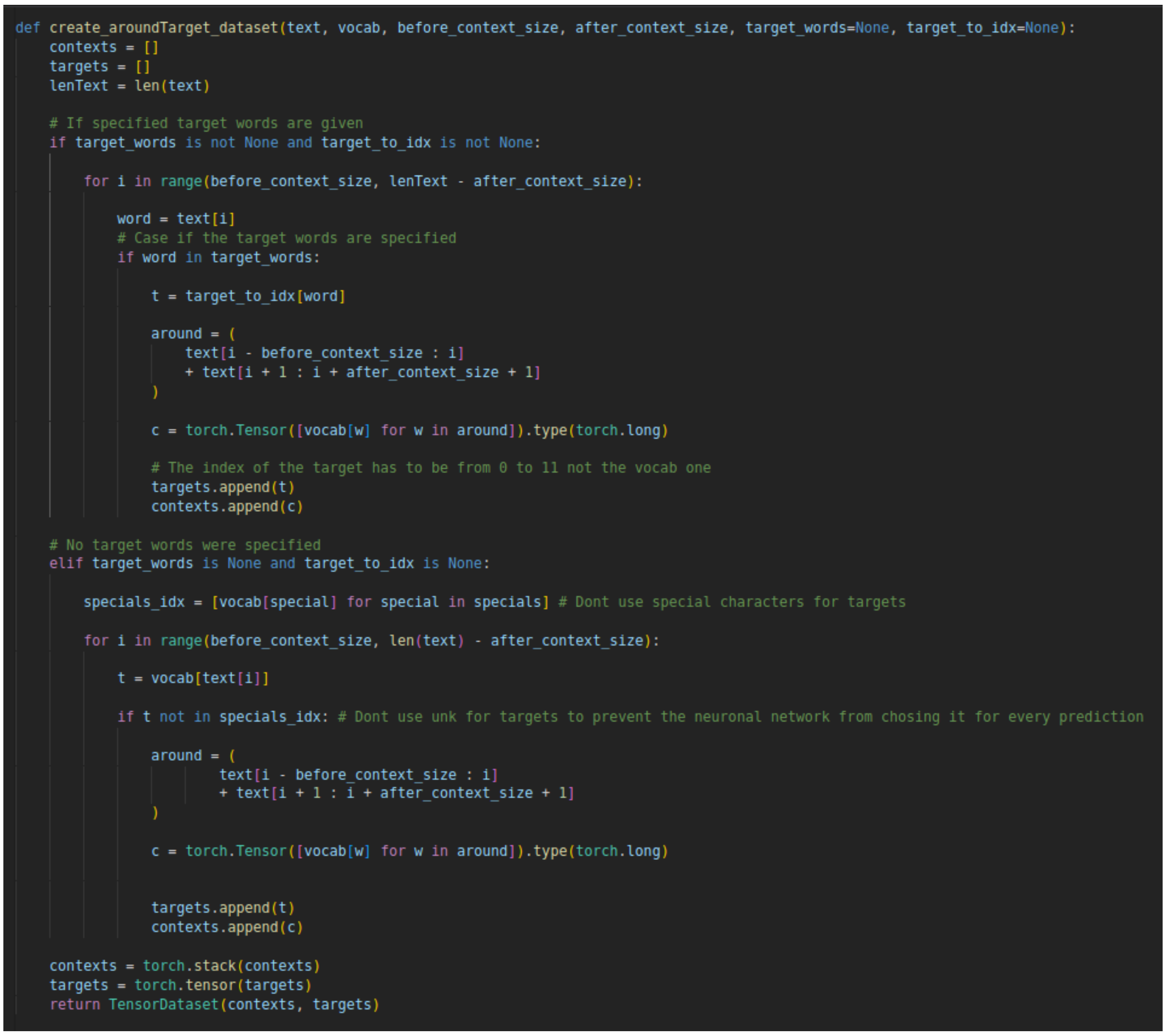


```python
epsilon = 1e-5
freqs_adjusted = freqs + epsilon  # Adjust frequencies to avoid zero

total_samples = torch.sum(freqs_adjusted)
class_weights_initial = total_samples / freqs_adjusted
class_weights = class_weights_initial / torch.sum(class_weights_initial)  # Normalize to sum to 1

print("Adjusted Frequencies:", freqs_adjusted)
print("Total Samples:", total_samples)
print("Initial Weights:", class_weights_initial)
print("Normalized Weights:", class_weights)
```

✓ 0.0s

```
Adjusted Frequencies: tensor([1.0000e-05, 1.8254e+05, 1.5128e+05,  ..., 1.8200e+02, 1.0000e+02,
        2.2996e+04])
Total Samples: tensor(2250799.0188)
Initial Weights: tensor([2.2508e+11, 1.2331e+01, 1.4879e+01,  ..., 1.2367e+04, 2.2508e+04,
        9.7878e+01])
Normalized Weights: tensor([9.9991e-01, 5.4779e-11, 6.6098e-11,  ..., 5.4940e-08, 9.9991e-08,
        4.3482e-10])
```

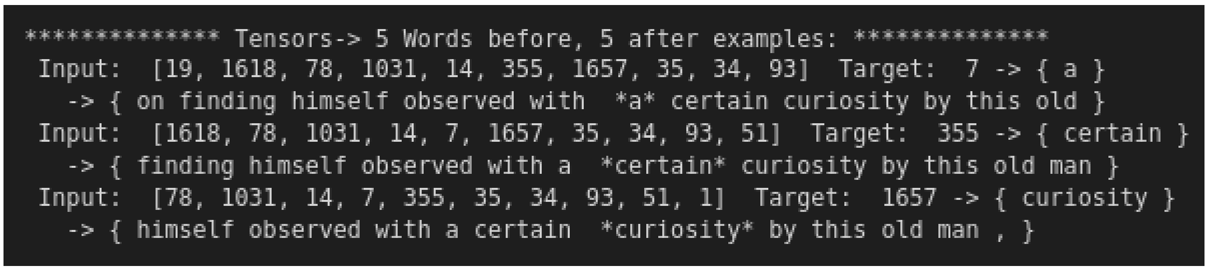
Then we will need a function that can handle all the given information inside each book and create a dataset to afterwards train the neuronal networks.



```python
def create_aroundTarget_dataset(text, vocab, before_context_size, after_context_size, target_words=None, target_to_idx=None):
    contexts = []
    targets = []
    lenText = len(text)

    # If specified target words are given
    if target_words is not None and target_to_idx is not None:

        for i in range(before_context_size, lenText - after_context_size):

            word = text[i]
            # Case if the target words are specified
            if word in target_words:

                t = target_to_idx[word]

                around = (
                    text[i - before_context_size : i]
                    + text[i + 1 : i + after_context_size + 1]
                )

                c = torch.Tensor([vocab[w] for w in around]).type(torch.long)

                # The index of the target has to be from 0 to 11 not the vocab one
                targets.append(t)
                contexts.append(c)

    # No target words were specified
    elif target_words is None and target_to_idx is None:

        specials_idx = [vocab[special] for special in specials] # Dont use special characters for targets

        for i in range(before_context_size, len(text) - after_context_size):

            t = vocab[text[i]]

            if t not in specials_idx: # Dont use unk for targets to prevent the neuronal network from chosing it for every prediction

                around = (
                        text[i - before_context_size : i]
                        + text[i + 1 : i + after_context_size + 1]
                )

                c = torch.Tensor([vocab[w] for w in around]).type(torch.long)


                targets.append(t)
                contexts.append(c)

    contexts = torch.stack(contexts)
    targets = torch.tensor(targets)
    return TensorDataset(contexts, targets)
```

This function will be able to create datasets for any word (We will not take the special characters as targets, to prevent the model to keep predicting unknown as there are many unknown words and it would have a big ratio of accuracy. We will not either take punctuation symbols to keep it simpler)

Here we can visualize some examples of how the tensors are created:

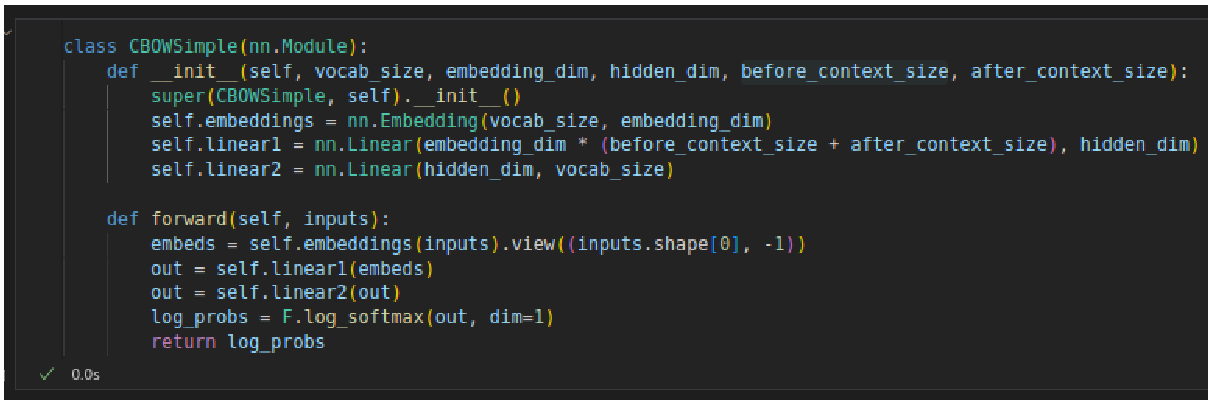


```
************** Tensors-> 5 Words before, 5 after examples: **************
 Input:  [19, 1618, 78, 1031, 14, 355, 1657, 35, 34, 93]  Target:  7 -> { a }
   -> { on finding himself observed with  *a* certain curiosity by this old }
 Input:  [1618, 78, 1031, 14, 7, 1657, 35, 34, 93, 51]  Target:  355 -> { certain }
   -> { finding himself observed with a  *certain* curiosity by this old man }
 Input:  [78, 1031, 14, 7, 355, 35, 34, 93, 51, 1]  Target:  1657 -> { curiosity }
   -> { himself observed with a certain  *curiosity* by this old man , }
```

The target word is between * and the rest is the context.

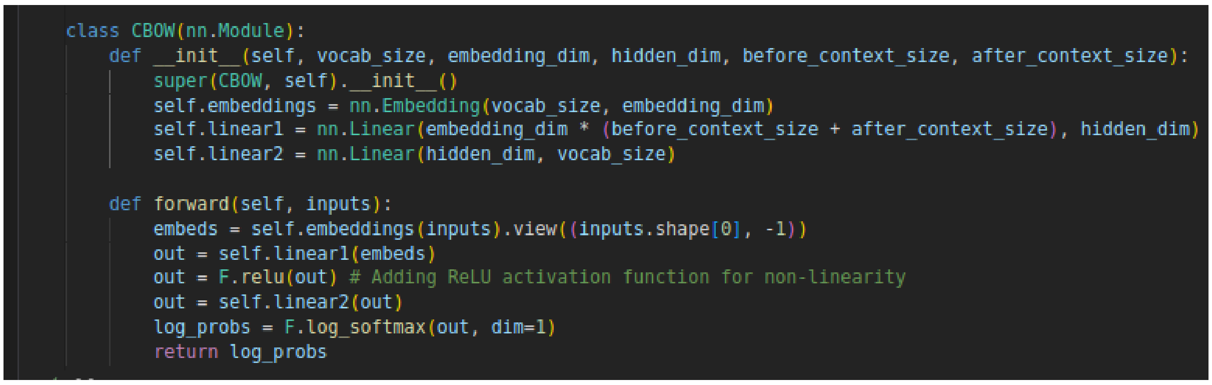
For the models we will be using 3 different ones:

- The simplest



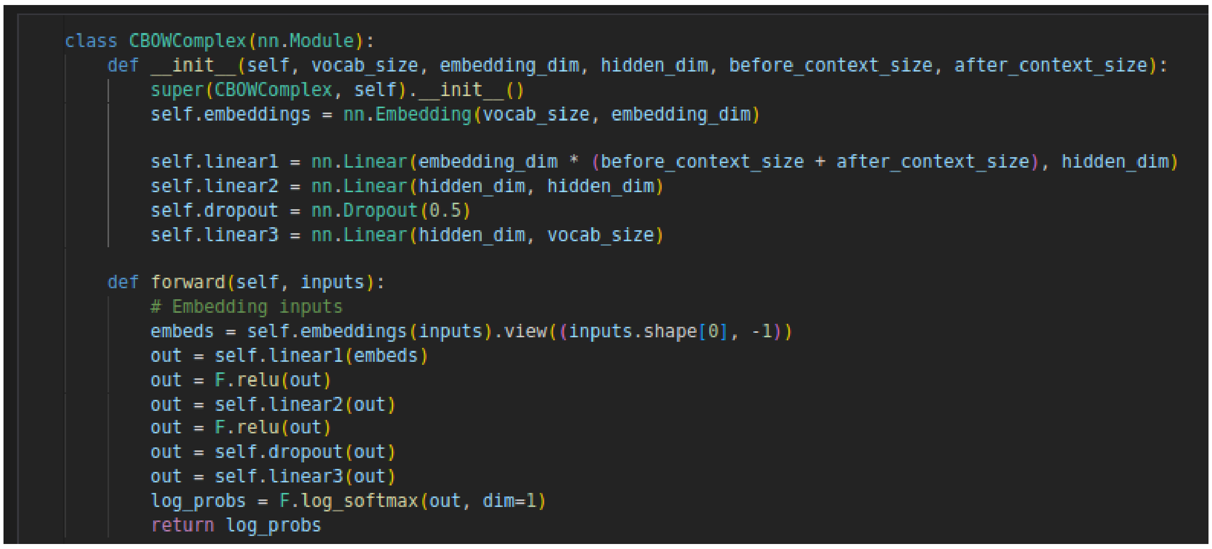
```python
class CBOWSimple(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, before_context_size, after_context_size):
        super(CBOWSimple, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.linear1 = nn.Linear(embedding_dim * (before_context_size + after_context_size), hidden_dim)
        self.linear2 = nn.Linear(hidden_dim, vocab_size)

    def forward(self, inputs):
        embeds = self.embeddings(inputs).view((inputs.shape[0], -1))
        out = self.linear1(embeds)
        out = self.linear2(out)
        log_probs = F.log_softmax(out, dim=1)
        return log_probs
```
✓ 0.0s

- The Medium complexity:



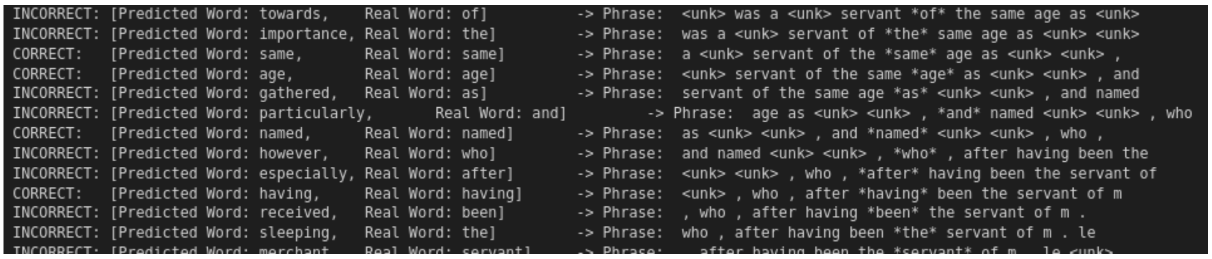
```python
class CBOW(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, before_context_size, after_context_size):
        super(CBOW, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.linear1 = nn.Linear(embedding_dim * (before_context_size + after_context_size), hidden_dim)
        self.linear2 = nn.Linear(hidden_dim, vocab_size)

    def forward(self, inputs):
        embeds = self.embeddings(inputs).view((inputs.shape[0], -1))
        out = self.linear1(embeds)
        out = F.relu(out) # Adding ReLU activation function for non-linearity
        out = self.linear2(out)
        log_probs = F.log_softmax(out, dim=1)
        return log_probs
```

- The most complex:



```python
class CBOWComplex(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, before_context_size, after_context_size):
        super(CBOWComplex, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)

        self.linear1 = nn.Linear(embedding_dim * (before_context_size + after_context_size), hidden_dim)
        self.linear2 = nn.Linear(hidden_dim, hidden_dim)
        self.dropout = nn.Dropout(0.5)
        self.linear3 = nn.Linear(hidden_dim, vocab_size)

    def forward(self, inputs):
        # Embedding inputs
        embeds = self.embeddings(inputs).view((inputs.shape[0], -1))
        out = self.linear1(embeds)
        out = F.relu(out)
        out = self.linear2(out)
        out = F.relu(out)
        out = self.dropout(out)
        out = self.linear3(out)
        log_probs = F.log_softmax(out, dim=1)
        return log_probs
```

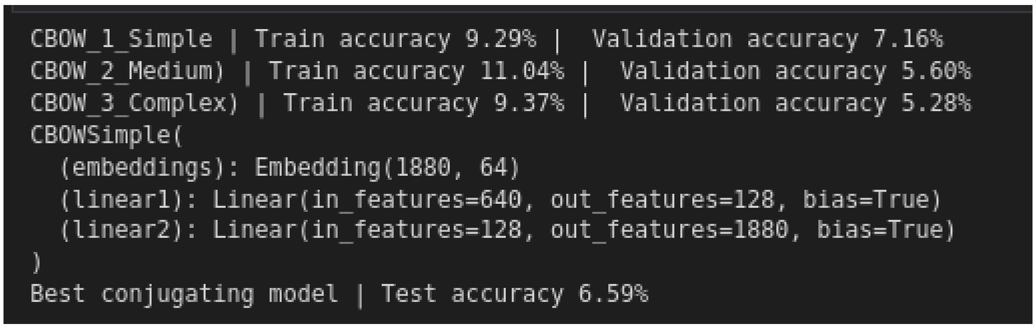
After training here we can see some examples of the predictions made.



```
INCORRECT: [Predicted Word: towards,    Real Word: of]         -> Phrase:  <unk> was a <unk> servant *of* the same age as <unk>
INCORRECT: [Predicted Word: importance, Real Word: the]        -> Phrase:  was a <unk> servant of *the* same age as <unk> <unk>
CORRECT:   [Predicted Word: same,       Real Word: same]       -> Phrase:  a <unk> servant of the *same* age as <unk> <unk> ,
CORRECT:   [Predicted Word: age,        Real Word: age]        -> Phrase:  <unk> servant of the same *age* as <unk> <unk> , and
INCORRECT: [Predicted Word: gathered,   Real Word: as]         -> Phrase:  servant of the same age *as* <unk> <unk> , and named
INCORRECT: [Predicted Word: particularly,       Real Word: and]        -> Phrase:  age as <unk> <unk> , *and* named <unk> <unk> , who
CORRECT:   [Predicted Word: named,      Real Word: named]      -> Phrase:  as <unk> <unk> , and *named* <unk> <unk> , who ,
INCORRECT: [Predicted Word: however,    Real Word: who]        -> Phrase:  and named <unk> <unk> , *who* , after having been the
INCORRECT: [Predicted Word: especially, Real Word: after]      -> Phrase:  <unk> <unk> , who , *after* having been the servant of
CORRECT:   [Predicted Word: having,     Real Word: having]     -> Phrase:  <unk> , who , after *having* been the servant of m
INCORRECT: [Predicted Word: received,   Real Word: been]       -> Phrase:  , who , after having *been* the servant of m .
INCORRECT: [Predicted Word: sleeping,   Real Word: the]        -> Phrase:  who , after having been *the* servant of m . le
INCORRECT: [Predicted Word: merchant,   Real Word: servant]    -> Phrase:  , after having been the *servant* of m . le <unk>
```

As we can see the database has many unknown words and it is really hard to choose one word between the 1880 possibilities.

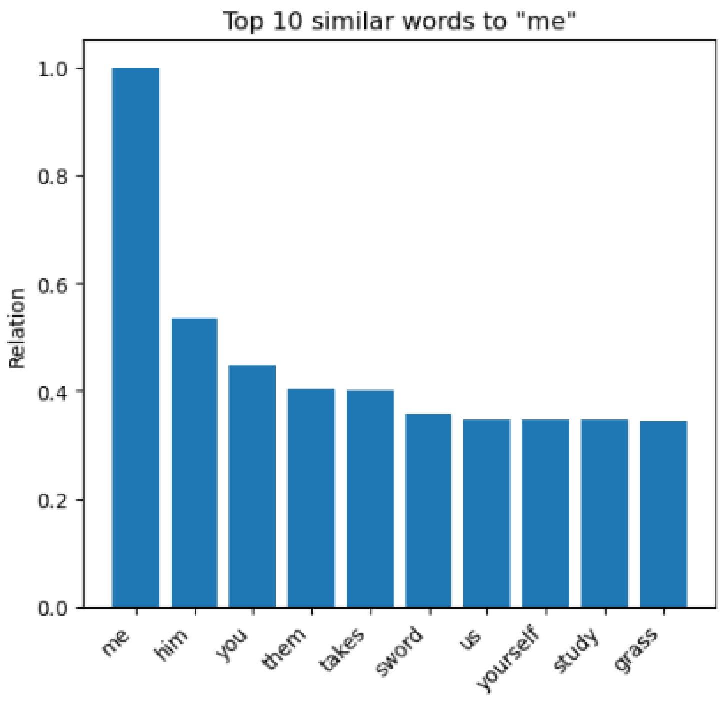
After training we got these results:



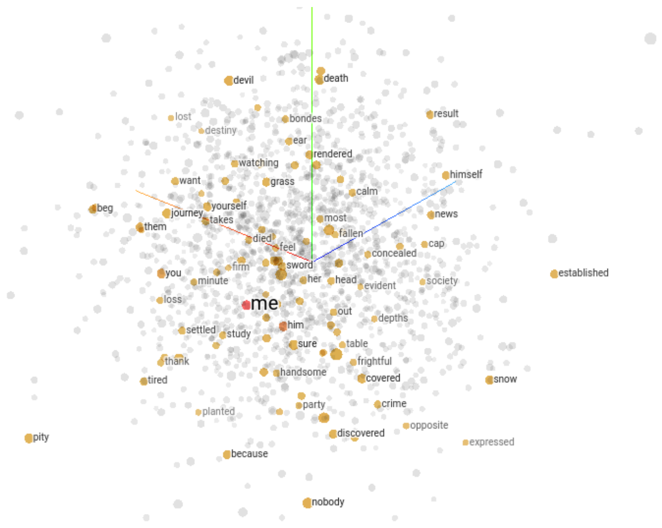
```
CBOW_1_Simple | Train accuracy 9.29% |  Validation accuracy 7.16%
CBOW_2_Medium) | Train accuracy 11.04% |  Validation accuracy 5.60%
CBOW_3_Complex) | Train accuracy 9.37% |  Validation accuracy 5.28%
CBOWSimple(
  (embeddings): Embedding(1880, 64)
  (linear1): Linear(in_features=640, out_features=128, bias=True)
  (linear2): Linear(in_features=128, out_features=1880, bias=True)
)
Best conjugating model | Test accuracy 6.59%
```

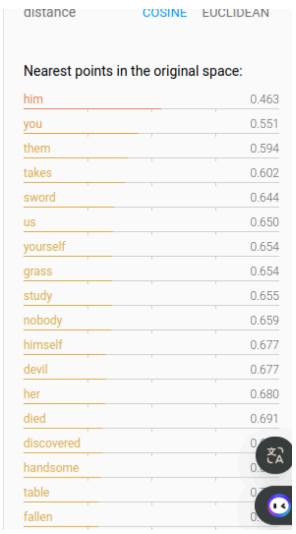
The simplest model has given us the best results, almost 7% of success in the text dataset. Even Though it looks like a really low percentage, it is very difficult even for a human to predict each word.

We then took a look of how each words is related with the rest and we got some interesting results as for example:



We can also see it in 3d in the matrix of the relation between words:




distance COSINE EUCLIDEAN

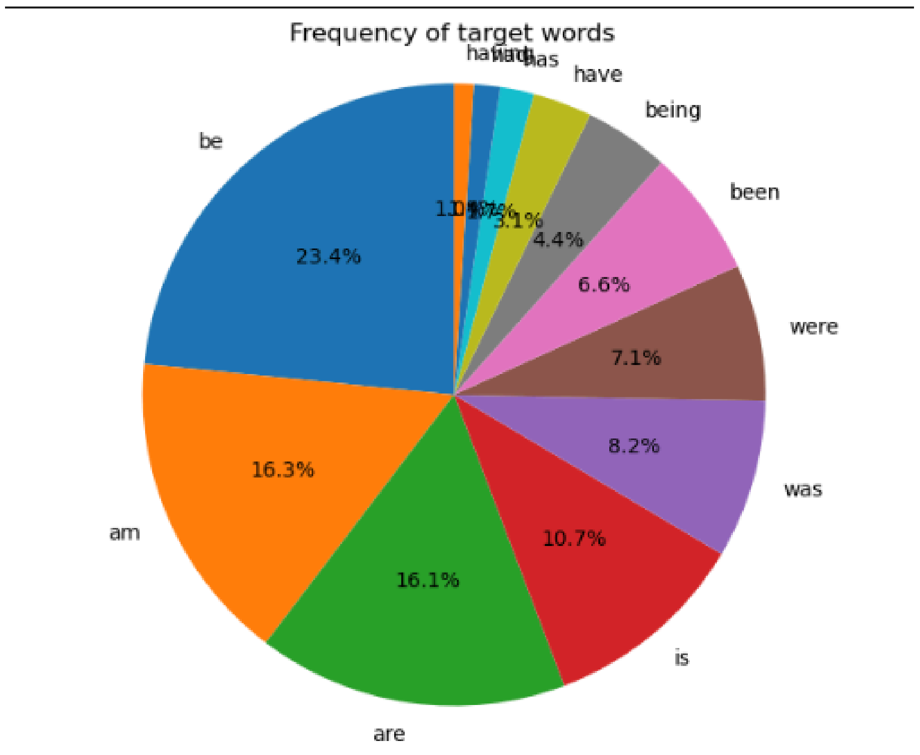
Nearest points in the original space:

him	0.463
you	0.551
them	0.594
takes	0.602
sword	0.644
us	0.650
yourself	0.654
grass	0.654
study	0.655
nobody	0.659
himself	0.677
devil	0.677
her	0.680
died	0.691
discovered	0.
handsome	0.
table	0.
fallen	0.

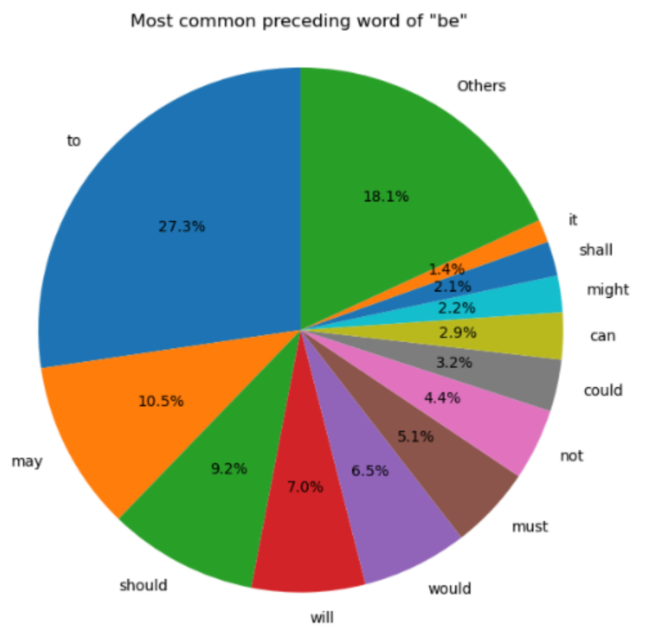
This seems to make sense for most of the words as for example me, him, you them, us… are all words used in the same content meaning wich person or people are doing an action

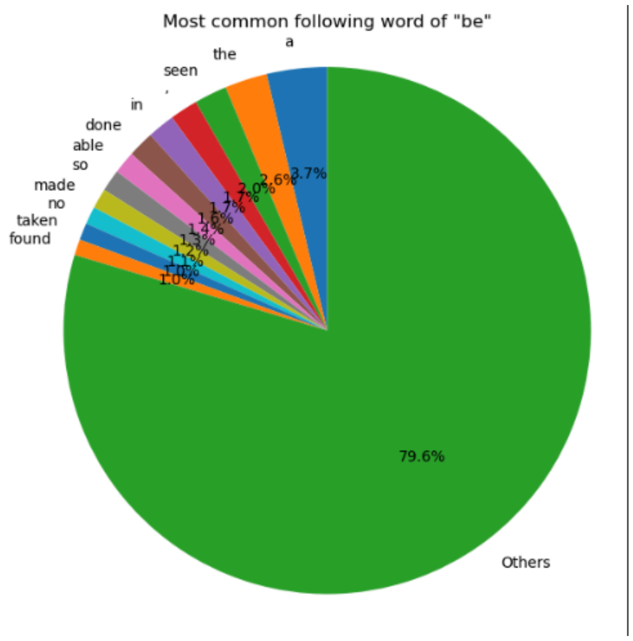
2. Conjugating be and have

We first took a look at the frequency of each conjugation and got this results.



As we can see there are some that appear much more than others.
We also took a look at the most common preceding and following word of each one of the possibilities.

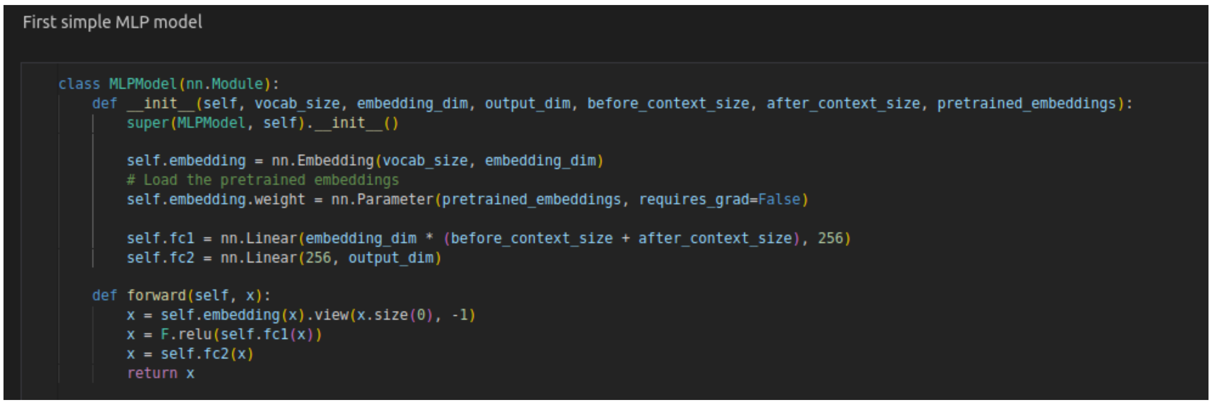




To make the conversion of the output easier we first created a vocabulary where the possible conjugations have the first index, to make the transformation for the output easier. We also had the idea of trying 2 different target models, one giving 5 words before the context and 5 after and another one given 12 before and 6to see if giving more words would cause a beneficial improvement in accuracy or might just cause overfitting, but after we ended up only using the first one. Anyway, The code is ready to train models with different amounts of context.

For the models this time 2 had 2 different ones for MLP and another 2 different ones for the RRN, using the pretrained weights of the embedding model.

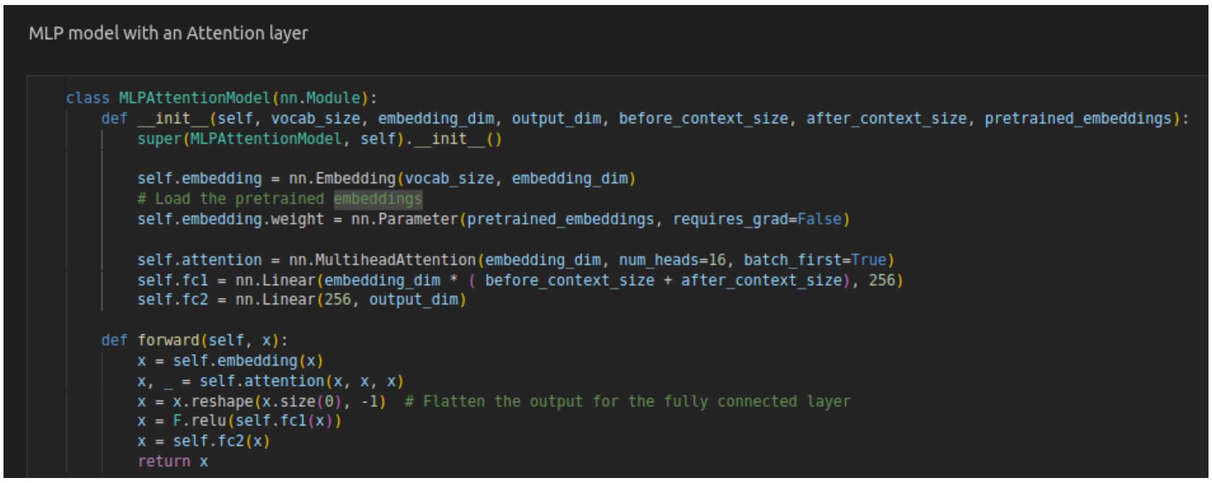
- MLP simple:



First simple MLP model

```python
class MLPModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, output_dim, before_context_size, after_context_size, pretrained_embeddings):
        super(MLPModel, self).__init__()

        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        # Load the pretrained embeddings
        self.embedding.weight = nn.Parameter(pretrained_embeddings, requires_grad=False)

        self.fc1 = nn.Linear(embedding_dim * (before_context_size + after_context_size), 256)
        self.fc2 = nn.Linear(256, output_dim)

    def forward(self, x):
        x = self.embedding(x).view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

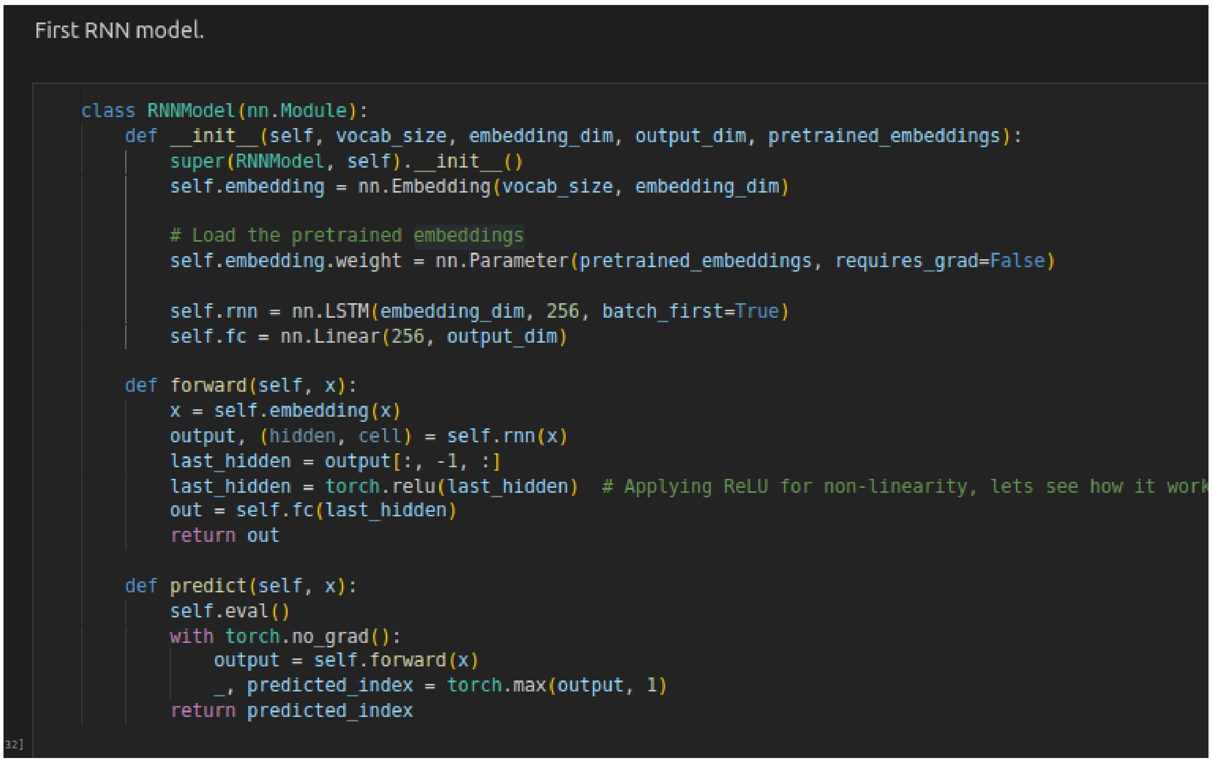
- MLP Attention model:



MLP model with an Attention layer

```python
class MLPAttentionModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, output_dim, before_context_size, after_context_size, pretrained_embeddings):
        super(MLPAttentionModel, self).__init__()

        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        # Load the pretrained embeddings
        self.embedding.weight = nn.Parameter(pretrained_embeddings, requires_grad=False)

        self.attention = nn.MultiheadAttention(embedding_dim, num_heads=16, batch_first=True)
        self.fc1 = nn.Linear(embedding_dim * ( before_context_size + after_context_size), 256)
        self.fc2 = nn.Linear(256, output_dim)

    def forward(self, x):
        x = self.embedding(x)
        x, _ = self.attention(x, x, x)
        x = x.reshape(x.size(0), -1)  # Flatten the output for the fully connected layer
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

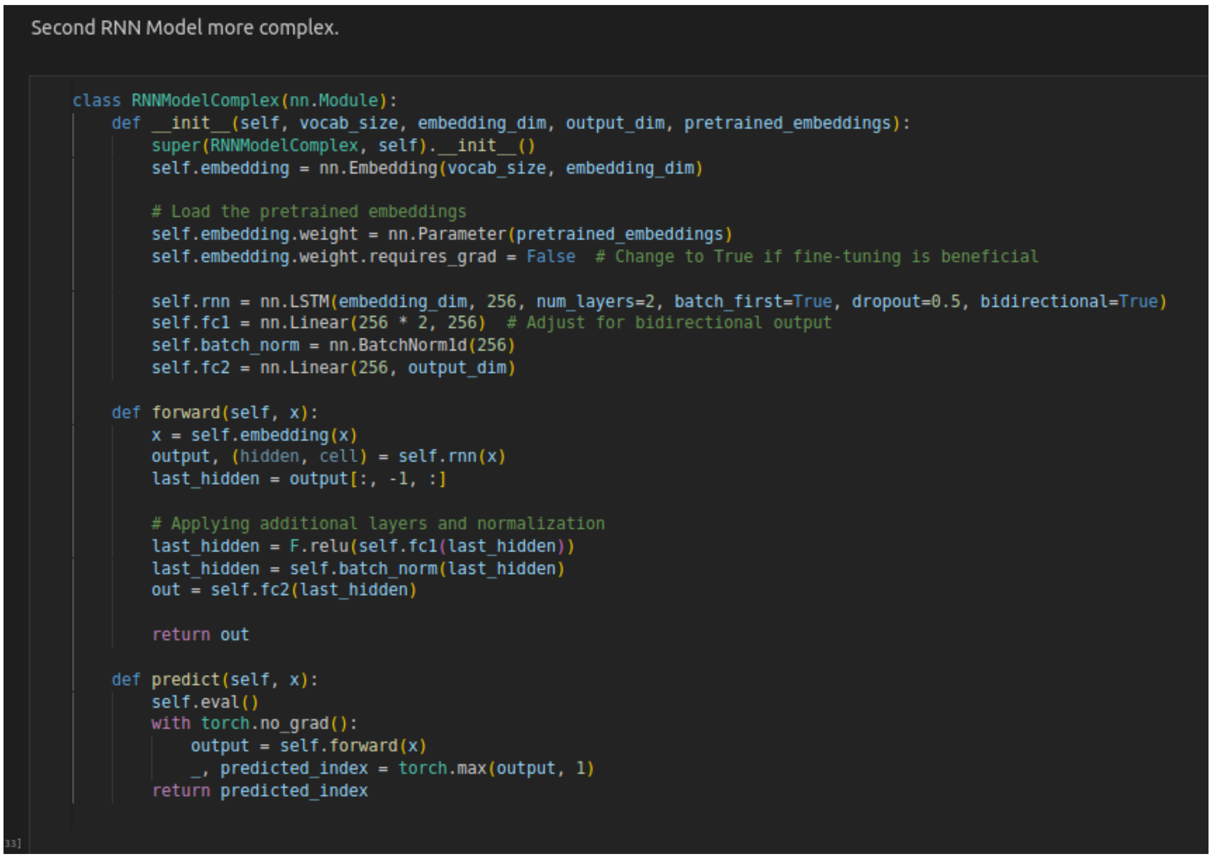
- RNN Simple:



First RNN model.

```
class RNNModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, output_dim, pretrained_embeddings):
        super(RNNModel, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)

        # Load the pretrained embeddings
        self.embedding.weight = nn.Parameter(pretrained_embeddings, requires_grad=False)

        self.rnn = nn.LSTM(embedding_dim, 256, batch_first=True)
        self.fc = nn.Linear(256, output_dim)

    def forward(self, x):
        x = self.embedding(x)
        output, (hidden, cell) = self.rnn(x)
        last_hidden = output[:, -1, :]
        last_hidden = torch.relu(last_hidden)  # Applying ReLU for non-linearity, lets see how it work
        out = self.fc(last_hidden)
        return out

    def predict(self, x):
        self.eval()
        with torch.no_grad():
            output = self.forward(x)
            _, predicted_index = torch.max(output, 1)
        return predicted_index
```

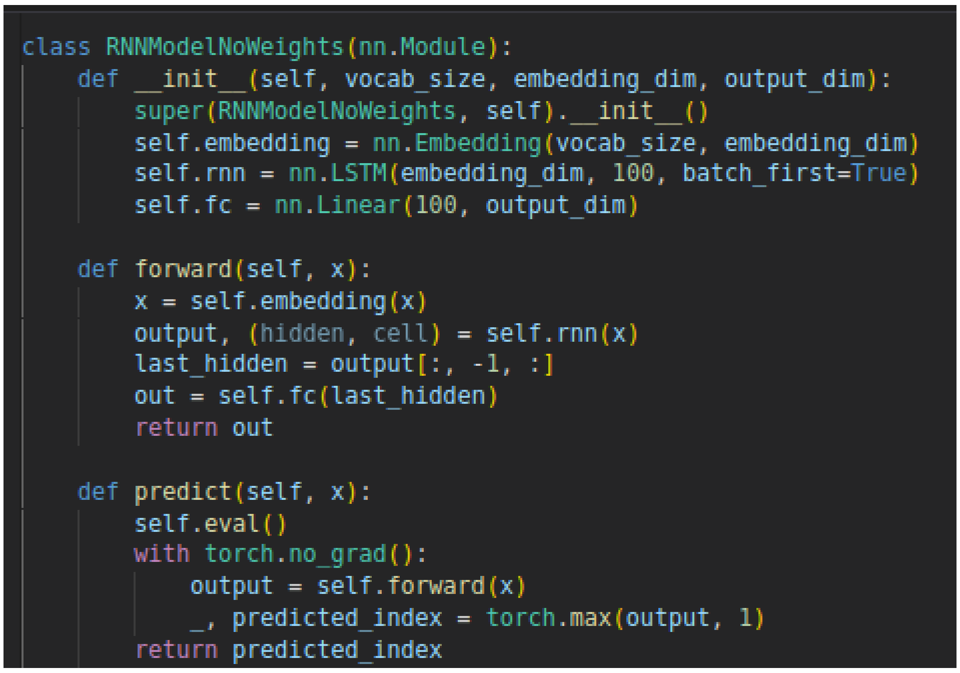
- RNN Complex:



Second RNN Model more complex.

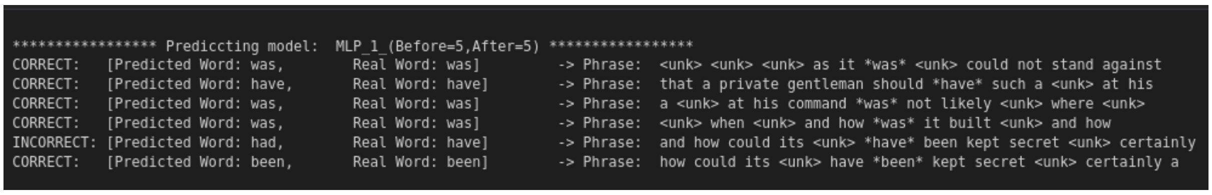
```
class RNNModelComplex(nn.Module):
    def __init__(self, vocab_size, embedding_dim, output_dim, pretrained_embeddings):
        super(RNNModelComplex, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)

        # Load the pretrained embeddings
        self.embedding.weight = nn.Parameter(pretrained_embeddings)
        self.embedding.weight.requires_grad = False  # Change to True if fine-tuning is beneficial

        self.rnn = nn.LSTM(embedding_dim, 256, num_layers=2, batch_first=True, dropout=0.5, bidirectional=True)
        self.fc1 = nn.Linear(256 * 2, 256)  # Adjust for bidirectional output
        self.batch_norm = nn.BatchNorm1d(256)
        self.fc2 = nn.Linear(256, output_dim)

    def forward(self, x):
        x = self.embedding(x)
        output, (hidden, cell) = self.rnn(x)
        last_hidden = output[:, -1, :]

        # Applying additional layers and normalization
        last_hidden = F.relu(self.fc1(last_hidden))
        last_hidden = self.batch_norm(last_hidden)
        out = self.fc2(last_hidden)

        return out

    def predict(self, x):
        self.eval()
        with torch.no_grad():
            output = self.forward(x)
            _, predicted_index = torch.max(output, 1)
        return predicted_index
```

- RNNNoWwights (does not take the embedding weights into account)



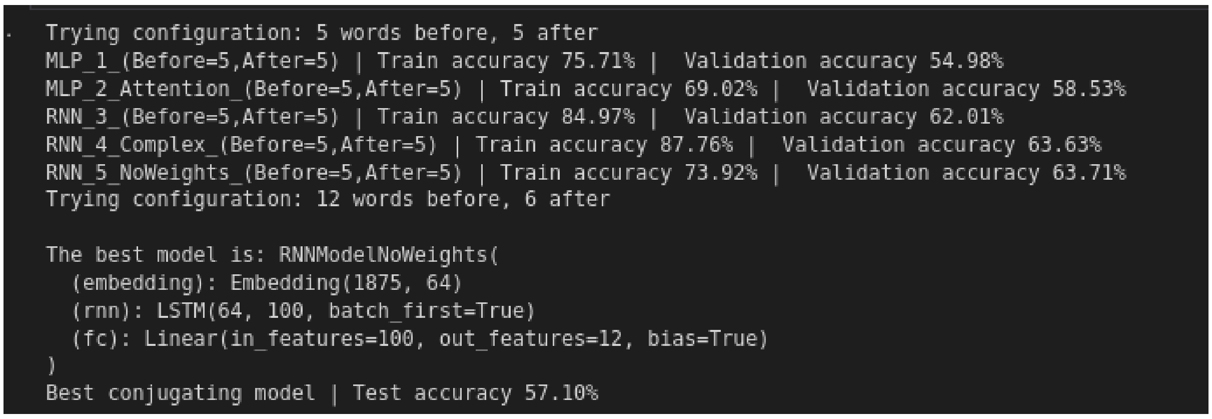
```
class RNNModelNoWeights(nn.Module):
    def __init__(self, vocab_size, embedding_dim, output_dim):
        super(RNNModelNoWeights, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.rnn = nn.LSTM(embedding_dim, 100, batch_first=True)
        self.fc = nn.Linear(100, output_dim)

    def forward(self, x):
        x = self.embedding(x)
        output, (hidden, cell) = self.rnn(x)
        last_hidden = output[:, -1, :]
        out = self.fc(last_hidden)
        return out

    def predict(self, x):
        self.eval()
        with torch.no_grad():
            output = self.forward(x)
            _, predicted_index = torch.max(output, 1)
        return predicted_index
```

We can see here an example of the predictions made by the models:



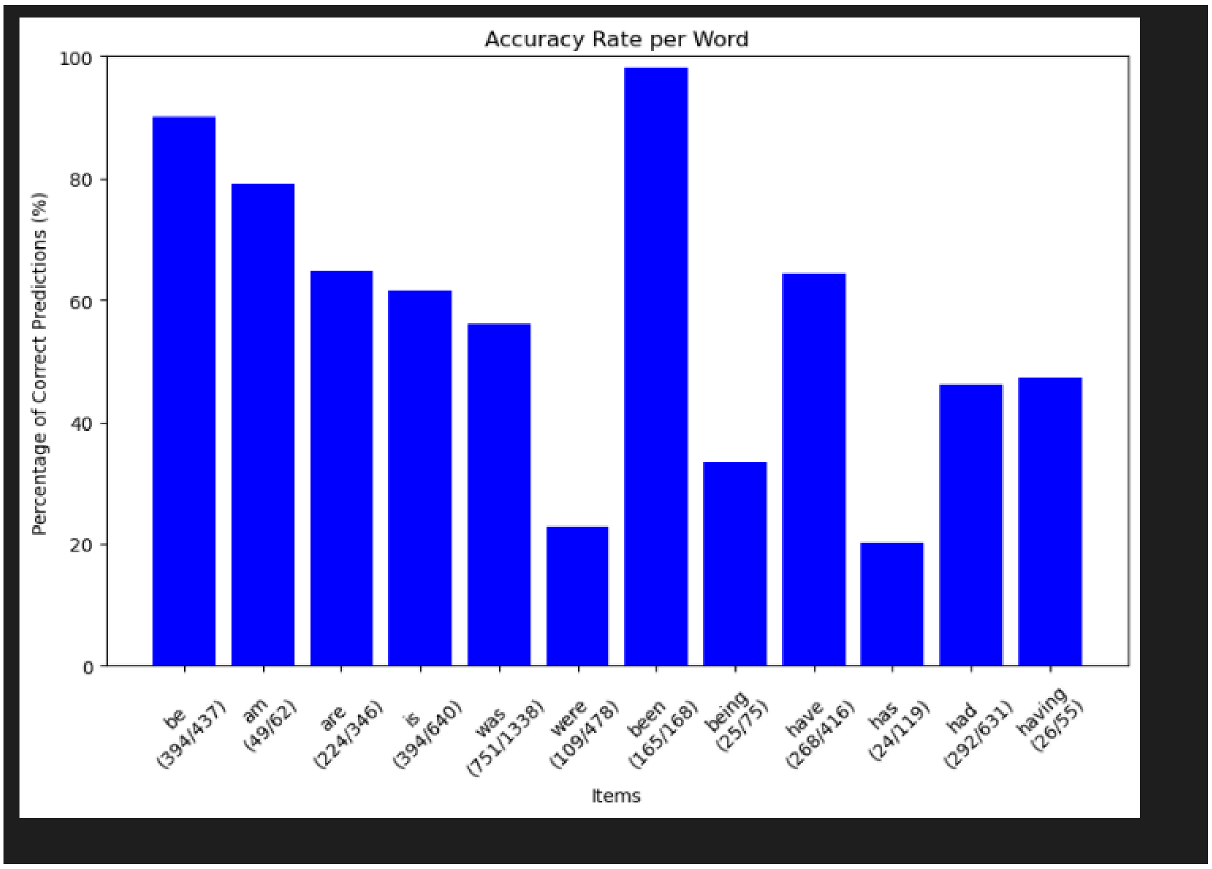
```
***************** Prediccting model:  MLP_1_(Before=5,After=5) *****************
CORRECT:   [Predicted Word: was,       Real Word: was]       -> Phrase:  <unk> <unk> <unk> as it *was* <unk> could not stand against
CORRECT:   [Predicted Word: have,      Real Word: have]      -> Phrase:  that a private gentleman should *have* such a <unk> at his
CORRECT:   [Predicted Word: was,       Real Word: was]       -> Phrase:  a <unk> at his command *was* not likely <unk> where <unk>
CORRECT:   [Predicted Word: was,       Real Word: was]       -> Phrase:  <unk> when <unk> and how *was* it built <unk> and how
INCORRECT: [Predicted Word: had,       Real Word: have]      -> Phrase:  and how could its <unk> *have* been kept secret <unk> certainly
CORRECT:   [Predicted Word: been,      Real Word: been]      -> Phrase:  how could its <unk> have *been* kept secret <unk> certainly a
```

And for the accuracy of each model:



```
Trying configuration: 5 words before, 5 after
MLP_1_(Before=5,After=5) | Train accuracy 75.71% |  Validation accuracy 54.98%
MLP_2_Attention_(Before=5,After=5) | Train accuracy 69.02% |  Validation accuracy 58.53%
RNN_3_(Before=5,After=5) | Train accuracy 84.97% |  Validation accuracy 62.01%
RNN_4_Complex_(Before=5,After=5) | Train accuracy 87.76% |  Validation accuracy 63.63%
RNN_5_NoWeights_(Before=5,After=5) | Train accuracy 73.92% |  Validation accuracy 63.71%
Trying configuration: 12 words before, 6 after

The best model is: RNNModelNoWeights(
  (embedding): Embedding(1875, 64)
  (rnn): LSTM(64, 100, batch_first=True)
  (fc): Linear(in_features=100, out_features=12, bias=True)
)
Best conjugating model | Test accuracy 57.10%
```

And for more detail we can see here the percentage of well predicted answers for each possibility:

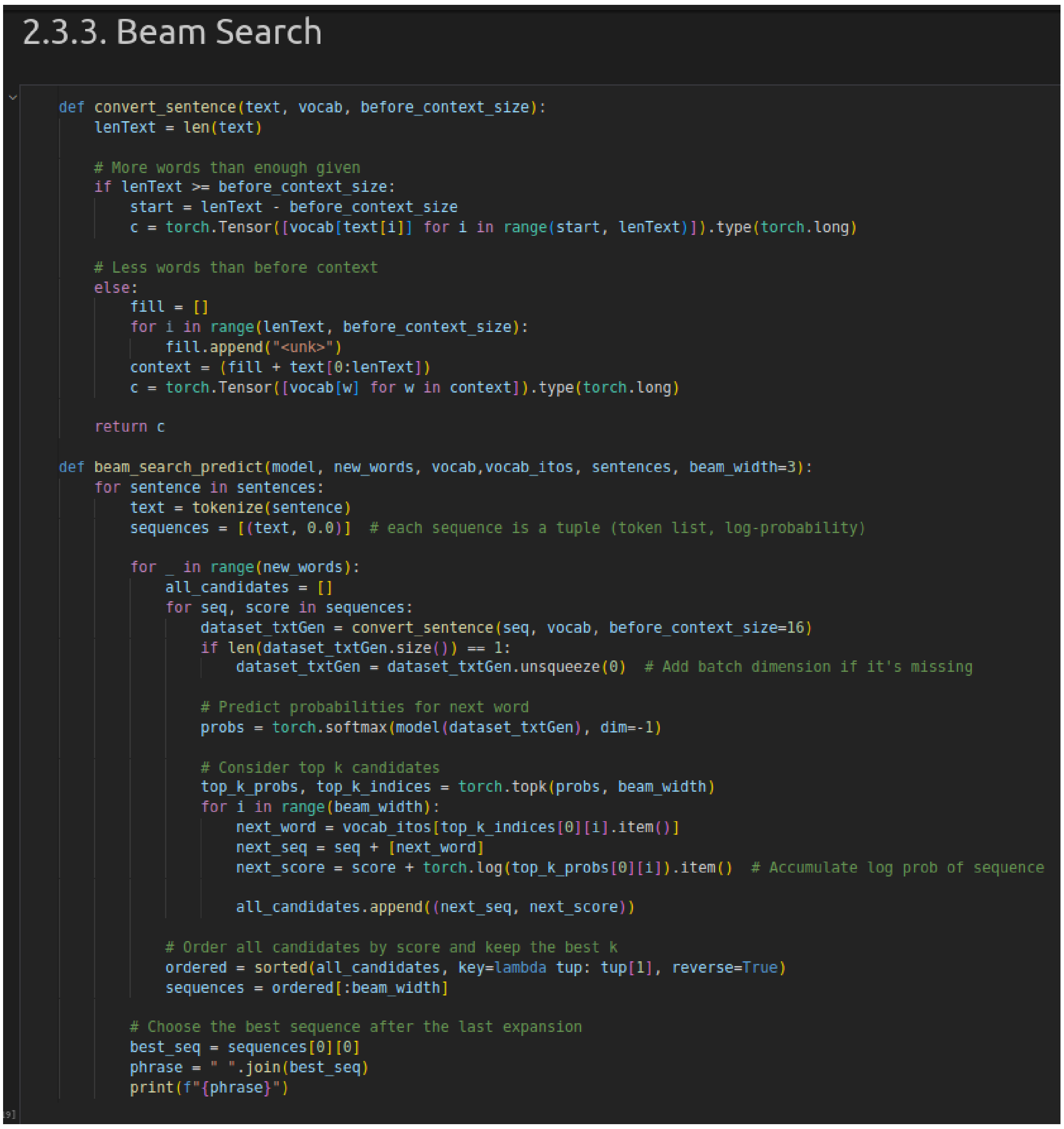


As we can see there are some conjugations that seem to be working pretty good and some others that look like maybe can be more confusing for the neuronal network.

3. Text generation

The last part was for a given sentence to predict the following words. We needed to create a special dataset for it as it would only train with the words given before the target.

We trained the RNN models we had before and used the beam search algorithm to predict the new words from a sentence.



2.3.3. Beam Search

```python
def convert_sentence(text, vocab, before_context_size):
    lenText = len(text)

    # More words than enough given
    if lenText >= before_context_size:
        start = lenText - before_context_size
        c = torch.Tensor([vocab[text[i]] for i in range(start, lenText)]).type(torch.long)

    # Less words than before context
    else:
        fill = []
        for i in range(lenText, before_context_size):
            fill.append("<unk>")
        context = (fill + text[0:lenText])
        c = torch.Tensor([vocab[w] for w in context]).type(torch.long)

    return c

def beam_search_predict(model, new_words, vocab,vocab_itos, sentences, beam_width=3):
    for sentence in sentences:
        text = tokenize(sentence)
        sequences = [(text, 0.0)]  # each sequence is a tuple (token list, log-probability)

        for _ in range(new_words):
            all_candidates = []
            for seq, score in sequences:
                dataset_txtGen = convert_sentence(seq, vocab, before_context_size=16)
                if len(dataset_txtGen.size()) == 1:
                    dataset_txtGen = dataset_txtGen.unsqueeze(0)  # Add batch dimension if it's missing

                # Predict probabilities for next word
                probs = torch.softmax(model(dataset_txtGen), dim=-1)

                # Consider top k candidates
                top_k_probs, top_k_indices = torch.topk(probs, beam_width)
                for i in range(beam_width):
                    next_word = vocab_itos[top_k_indices[0][i].item()]
                    next_seq = seq + [next_word]
                    next_score = score + torch.log(top_k_probs[0][i]).item()  # Accumulate log prob of sequence

                    all_candidates.append((next_seq, next_score))

            # Order all candidates by score and keep the best k
            ordered = sorted(all_candidates, key=lambda tup: tup[1], reverse=True)
            sequences = ordered[:beam_width]

        # Choose the best sequence after the last expansion
        best_seq = sequences[0][0]
        phrase = " ".join(best_seq)
        print(f"{phrase}")
```

The results were:

```
ORIGINAL: ['This dramatic story starts in a cold place, far from']
PREDICTION:  this dramatic story starts in a cold place , far from seed sown
returning feet apart rows wide open notice below down covering

ORIGINAL: ['It is sunny today and I feel good']
PREDICTION:  it is sunny today and i feel good pleasure alive letters alive
to-day ones sounds boots below inside beneath carefully

ORIGINAL: ['I still remember that time when I was with my friend in the park']
PREDICTION:  i still remember that time when i was with my friend in the park
sitting room closed wide shining herself open wide open wide open wide

ORIGINAL: ['I was thinking about going out tonight but']
PREDICTION:  i was thinking about going out tonight but finding matters lay
bound points trying toward around somewhere below rapidly beyond

ORIGINAL: ['Is always easy to judge when']
PREDICTION:  is always easy to judge when military human events turns
exist whence lies beyond effect produced brave brave
```