Conjugating *be* and *have* (2.2)

MLP with attention

We created a class called MLPWithAttention that inherits from the nn.Module class from pythorch, this allows us to use all the functionalities provided from the nn.Module class. Our class takes embedding_dim, hidden_dim, output_dim, max_len and num_heads as input.

The **embedding** layer maps each integer in the input to a higher dimensional vector using our pretrained embedding matrix. This returns a tensor of shape [batch_size, max_len, embedding_dim]

The **positional encoding** layer adds a positional encoding vector to each embedded word vector. This helps the model track the position of each word in the sequence. This also returns a tensor of shape [batch_size, max_len, embedding_dim].

The **multihead attention** layer applies the attention mechanism in parallel. How many is decided by the num_heads parameter. Where each head computed

-Query, key, value computation
-Attention scores
-Output values

The output value are weighted by the probabilities and summed to create an output for each head. Then the output of all heads are concatenated along the feature dimension. This also gives an output of [batch_size, max_len, embedding_dim].

Then we flatten the output to create a tensor of shape [batch_size, max_len * embedding_dim]. This is then passed though our fc1 layer which uses a linear transformation to transform out vector into a hidden layer representation. This is followed by a relu activation function which gives a tensor of shape [batch_size, hidden_dim].

Lastly, we have our fc2 layer where a final linear transformation is applied to get our desired output dimensions. The output is a tensor of shape [batch_size, 12].

Simple MLP

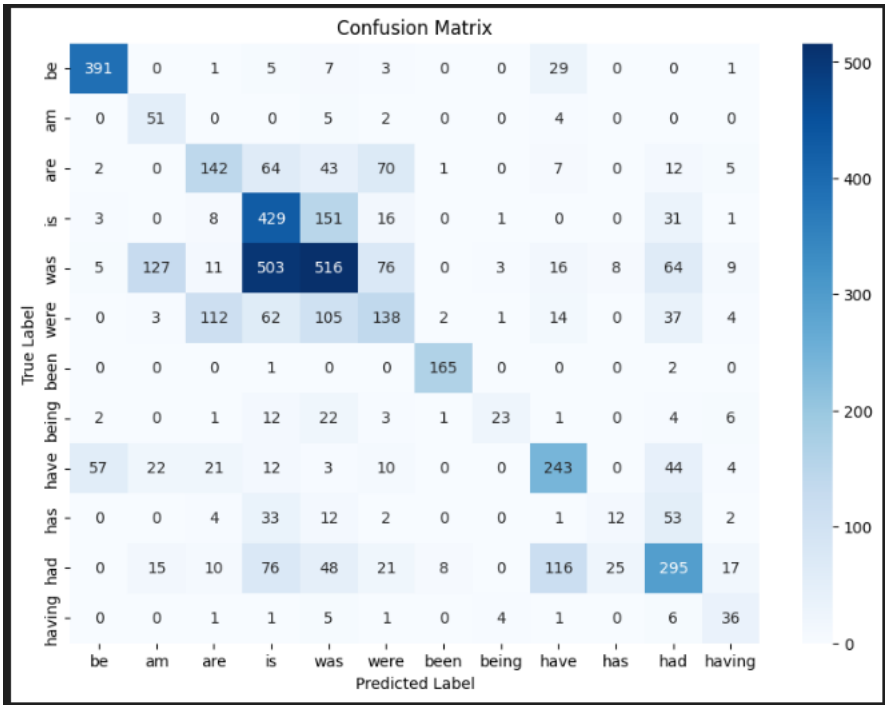
The simple MLP is just the same model has MLP with attention without positional encoding and attention layer.

RNN, LSTM

The network starts with an embedding layer which does the same as it did for the MLP model. Then we have LSTM layers using nn.LSTM . This outputs a tensor of shape [batch_size, sequence_length, hidden_dim]. Finally, we select the final output of the sequence creating a tensor of shape [batch_size, hidden_dim] which is then sendt though a fullt connected layer to get out desired output dimension which is a tensor of shape [batch_size, 12].

Results, Simple MLP

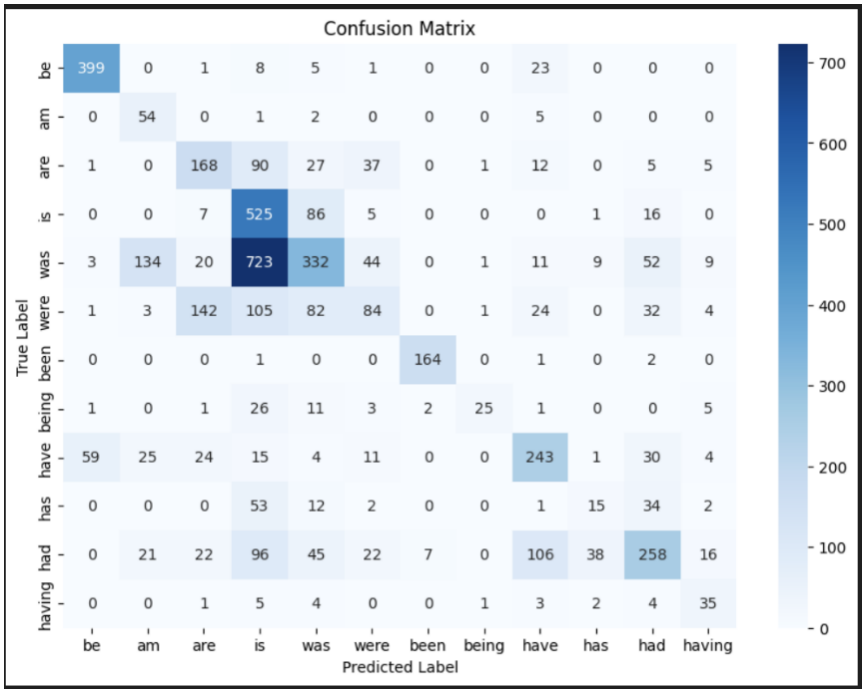
The simple MLP model gave surprisingly good results with 60% accuracy on validation data and 51% on test data. We trained a total of 8 models, where the best model had these hyperparameters: embedding_dim=100, hidden_dim=128 and lr=0.0001. Here is the confusion matrix for pred on test data.



Here we can see that this model struggles with was and is. You will see that this is a trend for all the models.

Results, MLP with attention

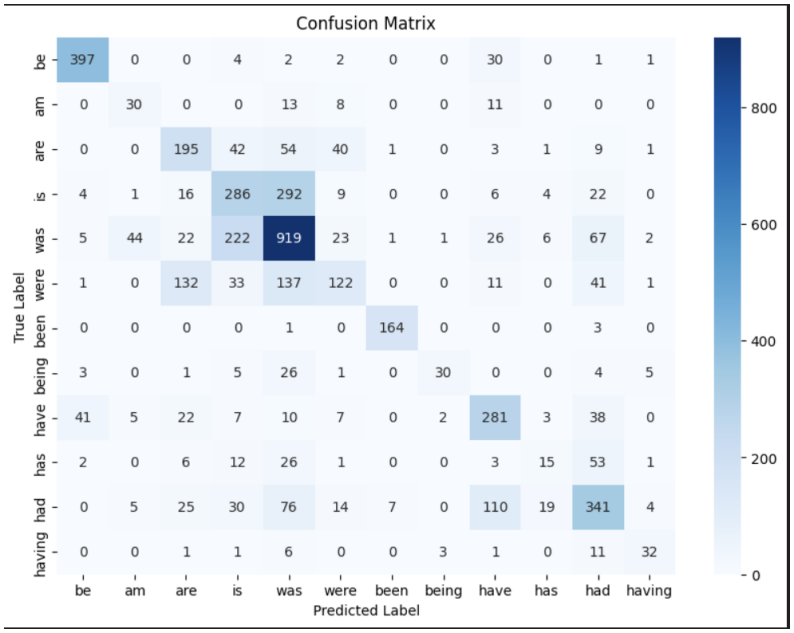
MLP with attention gave a validation accuracy of 56% with a test accuracy of 48%. We trained a total of 16 models where the best model used these hyperparameters: embedding_dim = 100, hidden_dim=128, num_heads=4 and lr=0,0001. This is not an amazing result, but it is still drastically better than random guessing. Here is the confusion matrix for MLP on test data.



Here we can see that our model did very well for all classes except is and was. If we fixed this issue, then our model would have done very well. This could potentially be fixed by checking if there was sufficient training data for is and was.

Results, LSTM

We trained a total of 36 different models for our LSTM architecture. The best model had 6 LSTM layer, 256 hidden_dim, 0,4 in dropout and 0,001 learning rate. This gave a validation accuracy of 67% and a training accuracy of 60%. These models had max_len of 6 meaning contexts consisted of 3 words in front and 3 words behind the prediction word. Here is the confusion matrix for test data.
As we can see the model does a relatively good job of choosing the correct conjugation. With a couple it struggled with, like was and is seems to be hard for the model to learn.



Comparing RNN and MLP with attention

MLP with attention and LSTM both took about 1:20 minutes per epoch. Since most of the models stopped after around 12 epochs one model took approximately 12 minutes to train. MLP without attention was a lot faster to train, but the performance was a lot worse as it lacked the mechanism to take word placement into account. It only used about 15 seconds per epoch and gave a respectable result of 60% accuracy on validation data and 51% on test data. Even though the LSTM(RNN) was the best performer of the three models it did take substantially longer to train than the simple MLP. The reason for the good performance of the simple MLP might be due to the max len being set to 6. If this was increased, then we might see that the more complex models would be even more favored.