NANYANG TECHNOLOGICAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Assignment for SC4002 / CE4045 / CZ4045

AY2023-2024

Group ID: 25

Group members:

Name	Matric No.	Contribution
Jewel Wee Xin Yu	U2022594F	1.1 & 1.2
Saniya Nangia	U2022557J	1.3
Sim Xin Ni Clodia	U2021577B	1.3
Lim Yi Jie Jasmin	U2021622K	2
Kelsey Sim Zhi Xuan	U2022455H	2

Table of Contents

Part 1. Sequence Tagging: NER

1.1 Word Embedding

After downloading the ***word2vec*** embeddings, we can call the method ***gensim.***models***.Word2Vec.most_similar*** to find the most similar words.This method computes cosine similarity between the mean of the projection weight vectors of the given words and the vectors for each word in the model. The result is an ordered list of the top N most similar words, with their respective cosine similarities. The most similar words for "student", "Apple", and "apple" are as illustrated in Figure 1.1.1 below.

Word	Most Similar Word	Cosine Similarity
student	students	0.7295
Apple	Apple_AAPL	0.7457
apple	apples	0.7204

Figure 1.1.1: Table showing a word, its most similar word and the cosine similarity

The code output is as shown below in Figure 1.1.2.

```
# Finding most similar words
words = ['student', 'Apple', 'apple']
similarWords = {}

for word in words:
    mostSimilar = w2v.most_similar(word, topn = 1)
    similarWords[word] = mostSimilar

# Print the results
for word, similar in similarWords.items():
    print(f"Word: {word}")
    for similar_word, cosine_similarity in similar:
        print(f"  Similar Word: {similar_word}, Cosine Similarity: {cosine_similarity:.4f}")
```

✓ 1m 17.4s Python

```
Word: student
  Similar Word: students, Cosine Similarity: 0.7295
Word: Apple
  Similar Word: Apple_AAPL, Cosine Similarity: 0.7457
Word: apple
  Similar Word: apples, Cosine Similarity: 0.7204
```

Figure 1.1.2: Output of *Word2Vec.most_similar* method call.

1.2 Data

1.2.1 Describing the Datasets

The figure below shows the respective sizes of the files for CoNLL2003.

Dataset	No. of Sentences
Training	14987
Development	3466
Test	3684

Figure 1.2.1.1: Table showing the number of sentences in each file

Additionally, the complete set of all possible word labels are specified in the figure below:

Word Labels	
B-LOC	I-LOC
B-ORG	I-ORG
B-MISC	I-MISC
O	I-PER

Figure 1.2.1.2: Complete set of word labels

The CoNLL2003 dataset uses the IOB1 labelling scheme:

- Tags prefixed with “B-” (begin) indicate the beginning of an entity, only if it immediately follows another entity of the same type. It is used to separate 2 adjacent entities of the same type.
- Tags prefixed with “I-” (inside) indicate that the token is part of a named entity. Subsequent “I-” labelled tokens continue the entity.

- Tags with the abovementioned prefixes are suffixed by “LOC”, “ORG”, “MISC” or “PER”, indicating the type of entity.
- “O” (Outside) tags denote that the token does not belong to any named entity.

1.2.2 Forming Complete Named Entities from Word Labels

To retrieve entities with more than one word, we have to implement a function, ***retrieveNamedEntities()***, that searches for consecutive tokens with labels that follow the following pattern:

- Consecutive “I-” tags of the same entity type.
  E.g. I-LOC, I-LOC, I-LOC

- A “B-” tag, followed by consecutive “I-” tags, all of which are of the same type.
  E.g. B-LOC, I-LOC, I-LOC

As a result, the sentences below are retrieved (Figure 1.2.2.1):

Sentence 1:

Germany's representative to the European Union's veterinary committee Werner Zwingmann said on Wednesday consumers should buy sheepmeat from countries other than Britain until the scientific advice was clearer.

Named entities (with more than one word):

1. European Union (I-ORG, I-ORG)
2. Werner Zwingmann (I-PER, I-PER)

Sentence 2:

The television, which did not say when the security forces killed the rebels, said the four arrested men confessed details of the assassination of the French Roman Catholic Bishop Pierre Claverie.

Named entities (with more than one word):

1. Roman Catholic (B-MISC, I-MISC)
2. John Newcombe (I-PER, I-PER)

Figure 1.2.1.2: Sentences retrieved by implemented by *retrieveNamedEntities()*

1.3 Model

1.3.1 Transforming Data

Before transforming each word to its numerical representation, we tokenize the sentences into individual words. We created a data frame consisting of the words and their respective labels. Then, using the pre-trained word embeddings from Section 1.1, we generate the vectors for each word.

On top of that, we also encoded the labels such that they are mapped to classes of integer values.

```
# list of labels
unique_labels = list(set(train_df['label'].unique()) | set(dev_df['label'].unique()) | set(test_df['label'].unique()))

# Initialize the label encoder
label_encoder = LabelEncoder()

# Fit the label encoder to list of labels
label_encoder.fit(unique_labels)

# Transform labels into numerical values
train_df['label_encoded'] = label_encoder.transform(train_df['label'])
dev_df['label_encoded'] = label_encoder.transform(dev_df['label'])
test_df['label_encoded'] = label_encoder.transform(test_df['label'])

print(unique_labels)
```

```
['B-ORG', 'I-LOC', 'I-MISC', 'O', 'I-PER', 'I-ORG', 'B-MISC', 'B-LOC']
```

(q1.3a) Since there are new words in the datasets which are not found in the pre-trained word2vec dictionary, we handled them by treating them as out-of-vocabulary (OOV) words. Similar to adding a word token <UNK>, we assign the same random vector to these words. We do the same for both the training set as well as the test set.

```
vector_data = []
for word in df["word"]:
    if word in w2v:
        vector = w2v.get_vector(word)
    else:
        # unknown_vector was assigned randomly beforehand
        vector = unknown_vector
    vector_data.append(vector)
```

During training, we initially used DataLoaders to batch our data according to token batch size, and we set Shuffle = False. Shuffle was set to False to prevent entities from being shuffled up (since the input is processed word by word), and hence the data will stay sequential. However, we noticed that doing this might break certain entities into different batches. For example, “Natural Language” might be the last two tokens in one batch, while “Processing” is the first in the next batch.

To deal with this problem, we have decided to batch our data according to sentences instead of tokens. Padding tokens were added to all sentences such that they all have the same length, which we defined as the length of the longest sentence. Then, the batch size will be determined according to the number of sentences. Batch size is calculated as the number of sentences multiplied by the length of the sentence (including padding). During the training of the model, a padding mask is applied so that the model only learns the labels assigned to the words instead of the padding tokens.

1.3.2 Neural Networks

(q1.3b) To find the best neural network to predict the final label for each word, we experimented with and compared 3 types of models. They are namely: a single-layer neural network model, an LSTM (Long Short-Term Memory) Model, as well as a bi-directional LSTM model (with and without dropout). The validation accuracy refers to whether the tags of individual tokens have been identified successfully (eg “Natural”, “Language”, “Processing”), while F1 score accounts for whether the entire named entity has been tagged correctly (“Natural Language Processing”). The weighted average F1 score is computed in order to account for the differences in label frequency in the validation dataset.

The F1 score is calculated as follows:
F1 = (2PR) / (P + R) = TP / (TP + 0.5*(FP + FN)),
where P is Precision, R is Recall, TP is True Positives, FP is False Positives, and FN is False Negatives

The mathematical functions used in our models are stated in Appendix A.

The length of the final vector representation of each word to be fed into the softmax classifier is equal to the input dimension, which is 300. The output dimension is equal to the number of possible labels, which is 9 (8 NER labels and 1 padding label). The number of hidden neurons is 128 for LSTM and Bi-Directional LSTM models.

Setting of Network:
Linear Layer:

- During training, the parameters being updated include the following:
    - Weights and Biases
- Sizes of Parameters
    - fc.weight torch.Size([9, 300])
    - fc.bias torch.Size([9])
- Total Parameters = 2709

LSTM Layer:

- During training, the parameters being updated include the following:
    - Weights and Biases for Input Gate
    - Weights and Biases for Output Gate
    - Weights and Baises for Forget Gate
    - Weights and Biases for Cell State
- Sizes of Parameters
    - lstm.weight_ih_l0 torch.Size([512, 300])
    - lstm.weight_hh_l0 torch.Size([512, 128])
    - lstm.bias_ih_l0 torch.Size([512])
    - lstm.bias_hh_l0 torch.Size([512])
    - fc.weight torch.Size([9, 128])
    - fc.bias torch.Size([9])
- Total Parameters = 221321

Bidirectional LSTM Layer:

- During training, the parameters being updated include the following:
    - Weights and Biases for Input Gate (Forward and Backward)
    - Weights and Biases for Output Gate (Forward and Backward)
    - Weights and Baises for Forget Gate (Forward and Backward)
    - Weights and Biases for Cell State (Forward and Backward)
- Sizes of Parameters
    - bilstm.weight_ih_l0 torch.Size([512, 300])
    - bilstm.weight_hh_l0 torch.Size([512, 128])
    - bilstm.bias_ih_l0 torch.Size([512])
    - bilstm.bias_hh_l0 torch.Size([512])
    - bilstm.weight_ih_l0_reverse torch.Size([512, 300])
    - bilstm.weight_hh_l0_reverse torch.Size([512, 128])
    - bilstm.bias_ih_l0_reverse torch.Size([512])
    - bilstm.bias_hh_l0_reverse torch.Size([512])

    - fc.weight torch.Size([9, 256])
    - fc.bias torch.Size([9])
- Total Parameters = 442633

1.3.2.1 Single-Layer Neural Network Model

1.3.2.1.1 Experiment and Results

```python
# Define a Single Layer NN model
class SingleLinearNetwork(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(SingleLinearNetwork, self).__init__()
        self.fc = nn.Linear(input_dim, output_dim)

    def forward(self, x):
        x = self.fc(x)
        return x
```

Figure 1.3.2.1.1.1: Single-Layer Neural Network Model Architecture

Number of epochs used for training: 26/50

Run time: 16053.5379s

Results of the test set (including f1_score):

```
Predicted class: tensor([8, 8, 4,  ..., 7, 8, 8])
True class: tensor([8, 8, 4,  ..., 7, 8, 8])
Test loss: 0.2511696219444275
F1 Score: 0.6253233967733652
Test Accuracy: 94.07%
```

Figure 1.3.2.1.1.2: Results of Single-Layer Neural Network Model on test set

Evaluation based on the development set (including f1_score):

```
Size of input:  300
Size of output:  9
Starting model training...
Epoch 1/50, Train Loss: 0.3634, Val Loss: 1.4833, Val Accuracy: 92.91%, F1 Score:  0.583539
Epoch 2/50, Train Loss: 0.2710, Val Loss: 1.4736, Val Accuracy: 93.31%, F1 Score:  0.593717
Epoch 3/50, Train Loss: 0.2632, Val Loss: 1.4704, Val Accuracy: 93.43%, F1 Score:  0.604645
Epoch 4/50, Train Loss: 0.2597, Val Loss: 1.4689, Val Accuracy: 93.66%, F1 Score:  0.611426
Epoch 5/50, Train Loss: 0.2579, Val Loss: 1.4680, Val Accuracy: 93.68%, F1 Score:  0.613817
Epoch 6/50, Train Loss: 0.2568, Val Loss: 1.4674, Val Accuracy: 93.71%, F1 Score:  0.617176
Epoch 7/50, Train Loss: 0.2561, Val Loss: 1.4670, Val Accuracy: 93.68%, F1 Score:  0.617771
Epoch 8/50, Train Loss: 0.2556, Val Loss: 1.4667, Val Accuracy: 93.67%, F1 Score:  0.617727
Epoch 9/50, Train Loss: 0.2553, Val Loss: 1.4665, Val Accuracy: 93.67%, F1 Score:  0.619322
Epoch 10/50, Train Loss: 0.2551, Val Loss: 1.4663, Val Accuracy: 93.68%, F1 Score:  0.619613
Epoch 11/50, Train Loss: 0.2549, Val Loss: 1.4662, Val Accuracy: 93.68%, F1 Score:  0.619918
Epoch 12/50, Train Loss: 0.2548, Val Loss: 1.4661, Val Accuracy: 93.69%, F1 Score:  0.620194
Epoch 13/50, Train Loss: 0.2548, Val Loss: 1.4660, Val Accuracy: 93.69%, F1 Score:  0.620116
Epoch 14/50, Train Loss: 0.2547, Val Loss: 1.4659, Val Accuracy: 93.69%, F1 Score:  0.619860
Epoch 15/50, Train Loss: 0.2547, Val Loss: 1.4658, Val Accuracy: 93.68%, F1 Score:  0.619995
Epoch 16/50, Train Loss: 0.2546, Val Loss: 1.4658, Val Accuracy: 93.69%, F1 Score:  0.620975
Epoch 17/50, Train Loss: 0.2546, Val Loss: 1.4658, Val Accuracy: 93.69%, F1 Score:  0.620879
Epoch 18/50, Train Loss: 0.2546, Val Loss: 1.4657, Val Accuracy: 93.68%, F1 Score:  0.620185
Epoch 19/50, Train Loss: 0.2546, Val Loss: 1.4657, Val Accuracy: 93.68%, F1 Score:  0.620185
Epoch 20/50, Train Loss: 0.2546, Val Loss: 1.4657, Val Accuracy: 93.68%, F1 Score:  0.620146
Epoch 21/50, Train Loss: 0.2546, Val Loss: 1.4656, Val Accuracy: 93.68%, F1 Score:  0.619944
Epoch 22/50, Train Loss: 0.2546, Val Loss: 1.4656, Val Accuracy: 93.68%, F1 Score:  0.619944
...
Epoch 25/50, Train Loss: 0.2545, Val Loss: 1.4656, Val Accuracy: 93.68%, F1 Score:  0.619944
Epoch 26/50, Train Loss: 0.2545, Val Loss: 1.4656, Val Accuracy: 93.68%, F1 Score:  0.619944
No improvement in F1 score for 10 epochs. Stopping training.
Run time:  16053.5379s
```

Figure 1.3.2.1.1.3: Results of Single-Layer Neural Network Model on development set

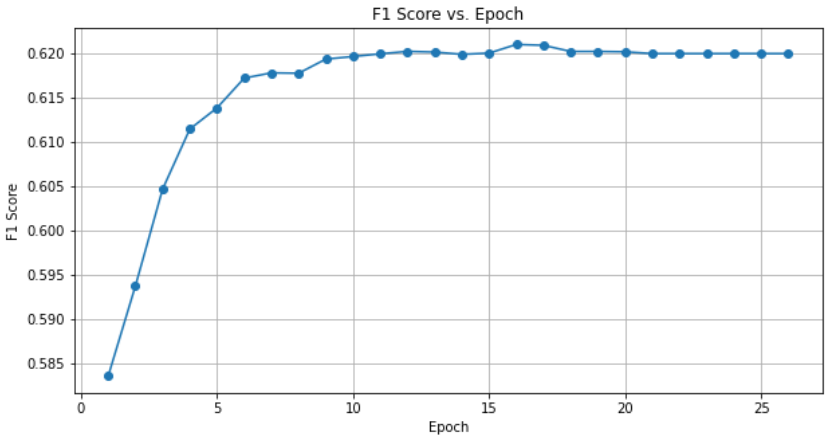


Figure 1.3.2.1.1.4: Graph of validation f1_score against epochs for Single-Layer Neural Network Model

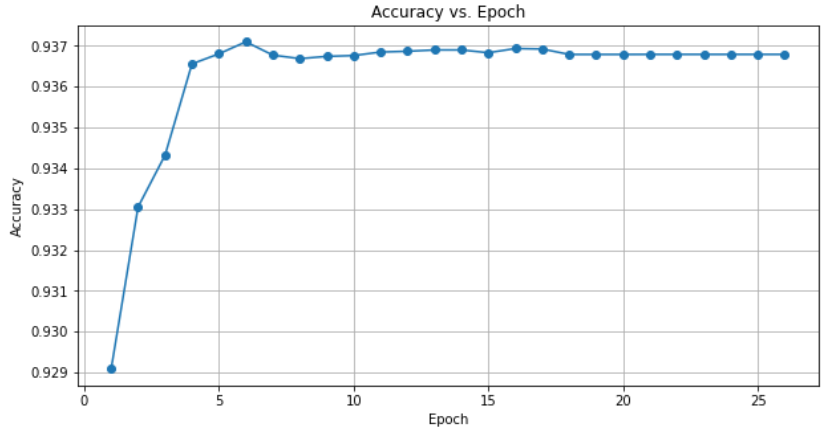


Figure 1.3.2.1.1.5: Graph of validation accuracy against epochs for Single-Layer Neural Network Model

1.3.2.1.2 Explanation

In this model, there is one single linear layer. It takes in the vector representation of each word to produce the output. Softmax activation is applied to the output to transform the raw scores into class probabilities. The class with the highest probability is taken as the predicted class.

However, we know that a single-linear layer neural network is not going to be sufficient to deal with sequential information like the sequence-based named entities. We therefore employ Recurrent Neural Networks, specifically Long Short-Term Memory architectures to handle this classification.

1.3.2.2 LSTM Model

1.3.2.2.1 Experiment and Results

```python
# Define a LSTM NN model
class LSTMClassifier(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(LSTMClassifier, self).__init__()
        self.hidden_size = hidden_size
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first = True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        out, _ = self.lstm(x)
        logits = self.fc(out)
        return logits
```

Figure 1.3.2.2.1.1: LSTM Model Architecture

Number of epochs used for training: 15/50

Run time: 10279.9130s

Results of the test set (including f1_score):

```
Predicted class: tensor([8, 8, 4,  ..., 7, 7, 8])
True class: tensor([8, 8, 4,  ..., 7, 8, 8])
Test loss: 0.27423426508903503
F1 Score: 0.7381218467981769
Test Accuracy: 95.66%
```

Figure 1.3.2.2.1.2: Results of LSTM Model on test set

Evaluation based on the development set (including f1_score):

```
Starting model training...
Epoch 1/50, Train Loss: 0.2175, Val Loss: 1.4392, Val Accuracy: 95.46%, F1 Score:  0.681924
Epoch 2/50, Train Loss: 0.1289, Val Loss: 1.4253, Val Accuracy: 96.21%, F1 Score:  0.716199
Epoch 3/50, Train Loss: 0.0983, Val Loss: 1.4184, Val Accuracy: 96.58%, F1 Score:  0.735860
Epoch 4/50, Train Loss: 0.0771, Val Loss: 1.4141, Val Accuracy: 96.96%, F1 Score:  0.751467
Epoch 5/50, Train Loss: 0.0609, Val Loss: 1.4106, Val Accuracy: 97.06%, F1 Score:  0.759858
Epoch 6/50, Train Loss: 0.0487, Val Loss: 1.4096, Val Accuracy: 96.90%, F1 Score:  0.754401
Epoch 7/50, Train Loss: 0.0397, Val Loss: 1.4079, Val Accuracy: 96.94%, F1 Score:  0.753719
Epoch 8/50, Train Loss: 0.0330, Val Loss: 1.4099, Val Accuracy: 96.71%, F1 Score:  0.743266
Epoch 9/50, Train Loss: 0.0279, Val Loss: 1.4076, Val Accuracy: 96.92%, F1 Score:  0.749843
Epoch 10/50, Train Loss: 0.0238, Val Loss: 1.4082, Val Accuracy: 96.83%, F1 Score:  0.749781
Epoch 11/50, Train Loss: 0.0208, Val Loss: 1.4079, Val Accuracy: 96.81%, F1 Score:  0.742498
Epoch 12/50, Train Loss: 0.0187, Val Loss: 1.4071, Val Accuracy: 96.77%, F1 Score:  0.749319
Epoch 13/50, Train Loss: 0.0166, Val Loss: 1.4060, Val Accuracy: 96.85%, F1 Score:  0.751160
Epoch 14/50, Train Loss: 0.0148, Val Loss: 1.4063, Val Accuracy: 96.82%, F1 Score:  0.746982
Epoch 15/50, Train Loss: 0.0142, Val Loss: 1.4069, Val Accuracy: 96.71%, F1 Score:  0.744956
No improvement in F1 score for 10 epochs. Stopping training.
Run time:  10279.9130s
```

Figure 1.3.2.2.1.3: Results of LSTM Model on Development set

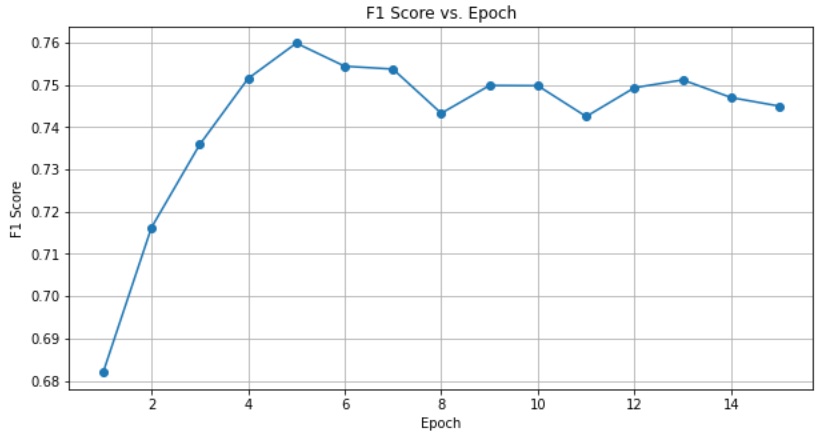


Figure 1.3.2.2.1.4: Graph of validation f1_score against epochs for LSTM Model

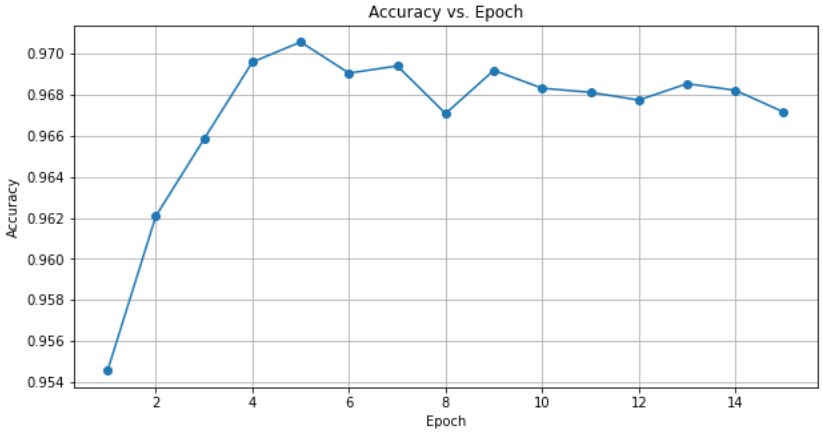


Figure 1.3.2.2.1.5: Graph of validation accuracy against epochs for LSTM Model

1.3.2.2.2 Explanation

The LSTM model is able to capture dependencies in sequences, which in this case would be named entities that are longer than 1 word (eg “Natural Language Processing”). Hence, it is able to better capture the relationships between the words that make up the named entity, as well as the context of the input, and assign a label accordingly. LSTMs can also capture long-term dependencies in data, which would be useful for longer named entities (3 or more words) or for differentiating between consecutive named entities. We chose to use LSTM instead of the traditional RNN model as LSTM addresses the vanishing gradient problem that is present in RNNs.

1.3.2.3 Bi-directional LSTM Model

```
class BiLSTMClassifier(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(BiLSTMClassifier, self).__init__()
        self.hidden_size = hidden_size
        self.bilstm = nn.LSTM(input_size, hidden_size, batch_first = True, bidirectional = True)
        self.fc = nn.Linear(2 * hidden_size, output_size)

    def forward(self, x):
        out, _ = self.bilstm(x)
        logits = self.fc(out)
        return logits
```

Figure 1.3.2.3.1.1: Bi-directional LSTM Model Architecture

1.3.2.3.1 Experiment and Results

Number of epochs used for training: 37/50

Run time: 27109.6228s

Results of the test set (including f1_score):

```
Predicted class: tensor([8, 8, 4,  ..., 7, 8, 8])
True class: tensor([8, 8, 4,  ..., 7, 8, 8])
Test loss: 0.3502257168292999
F1 Score: 0.819405389154852
Test Accuracy: 96.78%
```

Figure 1.3.2.3.1.2: Results of Bi-directional LSTM Model on test set

Evaluation based on the development set (including f1_score):

```
Starting model training...
Epoch 1/50, Train Loss: 0.1763, Val Loss: 1.4189, Val Accuracy: 96.84%, F1 Score:  0.751732
Epoch 2/50, Train Loss: 0.0864, Val Loss: 1.4064, Val Accuracy: 97.46%, F1 Score:  0.785051
Epoch 3/50, Train Loss: 0.0536, Val Loss: 1.4013, Val Accuracy: 97.60%, F1 Score:  0.793824
Epoch 4/50, Train Loss: 0.0330, Val Loss: 1.3988, Val Accuracy: 97.68%, F1 Score:  0.800050
Epoch 5/50, Train Loss: 0.0216, Val Loss: 1.3980, Val Accuracy: 97.65%, F1 Score:  0.799138
Epoch 6/50, Train Loss: 0.0157, Val Loss: 1.3969, Val Accuracy: 97.71%, F1 Score:  0.802763
Epoch 7/50, Train Loss: 0.0116, Val Loss: 1.3964, Val Accuracy: 97.77%, F1 Score:  0.804623
Epoch 8/50, Train Loss: 0.0095, Val Loss: 1.3966, Val Accuracy: 97.68%, F1 Score:  0.804301
Epoch 9/50, Train Loss: 0.0080, Val Loss: 1.3950, Val Accuracy: 97.78%, F1 Score:  0.806245
Epoch 10/50, Train Loss: 0.0062, Val Loss: 1.3951, Val Accuracy: 97.82%, F1 Score:  0.805672
Epoch 11/50, Train Loss: 0.0061, Val Loss: 1.3955, Val Accuracy: 97.75%, F1 Score:  0.801662
Epoch 12/50, Train Loss: 0.0047, Val Loss: 1.3954, Val Accuracy: 97.77%, F1 Score:  0.804392
Epoch 13/50, Train Loss: 0.0046, Val Loss: 1.3944, Val Accuracy: 97.81%, F1 Score:  0.803357
Epoch 14/50, Train Loss: 0.0046, Val Loss: 1.3961, Val Accuracy: 97.68%, F1 Score:  0.799809
Epoch 15/50, Train Loss: 0.0034, Val Loss: 1.3946, Val Accuracy: 97.81%, F1 Score:  0.802478
Epoch 16/50, Train Loss: 0.0040, Val Loss: 1.3949, Val Accuracy: 97.74%, F1 Score:  0.802753
Epoch 17/50, Train Loss: 0.0033, Val Loss: 1.3958, Val Accuracy: 97.65%, F1 Score:  0.803727
Epoch 18/50, Train Loss: 0.0029, Val Loss: 1.3949, Val Accuracy: 97.73%, F1 Score:  0.806909
Epoch 19/50, Train Loss: 0.0032, Val Loss: 1.3952, Val Accuracy: 97.70%, F1 Score:  0.803953
Epoch 20/50, Train Loss: 0.0027, Val Loss: 1.3942, Val Accuracy: 97.80%, F1 Score:  0.804679
Epoch 21/50, Train Loss: 0.0031, Val Loss: 1.3944, Val Accuracy: 97.81%, F1 Score:  0.802339
Epoch 22/50, Train Loss: 0.0024, Val Loss: 1.3944, Val Accuracy: 97.80%, F1 Score:  0.804898
Epoch 23/50, Train Loss: 0.0028, Val Loss: 1.3949, Val Accuracy: 97.74%, F1 Score:  0.803371
Epoch 24/50, Train Loss: 0.0024, Val Loss: 1.3954, Val Accuracy: 97.66%, F1 Score:  0.802634
...
Epoch 36/50, Train Loss: 0.0017, Val Loss: 1.3944, Val Accuracy: 97.75%, F1 Score:  0.802036
Epoch 37/50, Train Loss: 0.0017, Val Loss: 1.3940, Val Accuracy: 97.81%, F1 Score:  0.804443
No improvement in F1 score for 10 epochs. Stopping training.
Run time:  27109.6228s
```

Figure 1.3.2.3.1.3: Results of Bi-directional LSTM Model on development set

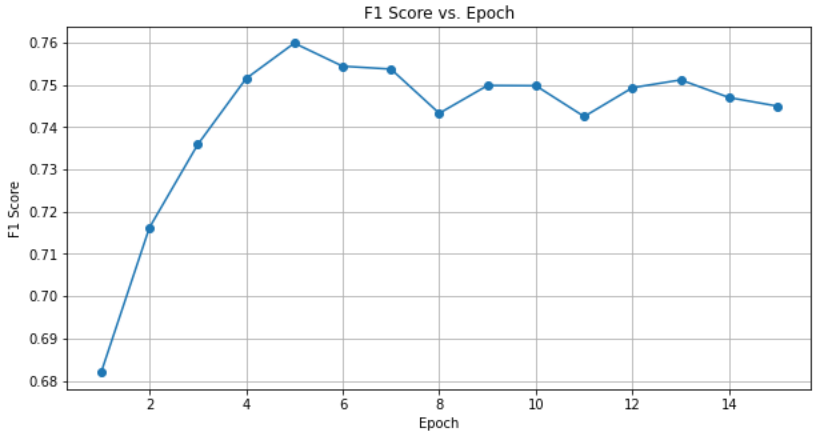


Figure 1.3.2.3.1.4: Graph of validation f1_score against epochs for Bi-directional LSTM Model

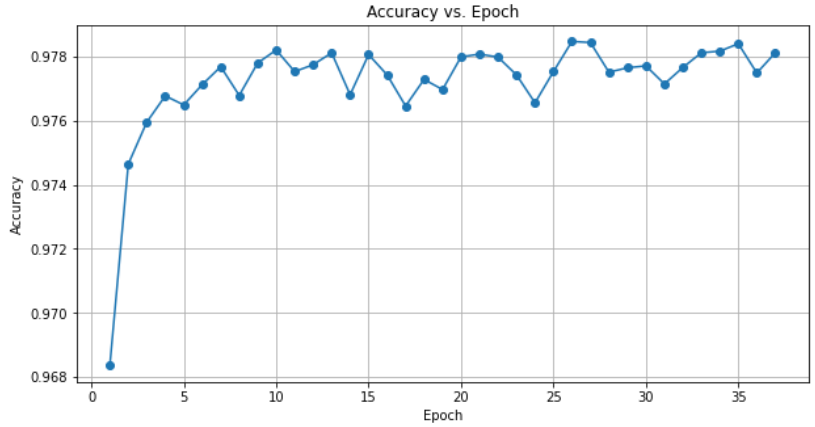


Figure 1.3.2.3.1.5: Graph of validation accuracy against epochs for Bi-directional LSTM Model

1.3.2.3.2 Explanation

Bidirectional LSTMs can process data in both the forward and backward directions. Hence, instead of simply relying on preceding words in a sequence, they can capture the context of both the previous and following words. If a named entity is longer than 1 or 2 words, this would be especially useful in understanding the boundaries of the named entities and predicting their labels. In addition, the feature representations that are learned by the model during training would be enhanced due to the contextual information from both the forward and backward direction, allowing the model to better classify named entities that consist of multiple words.

1.3.2.4 Bi-directional LSTM Model with Dropout

```python
class BiLSTMwithDropout(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, dropout_prob):
        super(BiLSTMwithDropout, self).__init__()
        self.hidden_size = hidden_size
        self.bilstm = nn.LSTM(input_size, hidden_size, batch_first = True, bidirectional = True)
        self.dropout = nn.Dropout(dropout_prob)
        self.fc = nn.Linear(2 * hidden_size, output_size)

    def forward(self, x):
        out, _ = self.bilstm(x)
        out = self.dropout(out)
        logits = self.fc(out)
        return logits
```

Figure 1.3.2.4.1.1: Bi-directional LSTM Model with Dropout Architecture

1.3.2.4.1 Experiment and Results

Number of epochs used for training: 50/50

Run time: 36925.5822

Results of the test set (including f1_score):

```
Predicted class: tensor([8, 8, 4,  ..., 7, 8, 8])
True class: tensor([8, 8, 4,  ..., 7, 8, 8])
Test loss: 0.3347415626049042
F1 Score: 0.8339593683745371
Test Accuracy: 97.00%
```

Figure 1.3.2.4.1.2: Results of Bi-directional LSTM Model with Dropout on test set

Evaluation based on the development set (including f1_score):

```
Starting model training...
Epoch 1/50, Train Loss: 0.1965, Val Loss: 1.4239, Val Accuracy: 96.37%, F1 Score:  0.738263
Epoch 2/50, Train Loss: 0.1118, Val Loss: 1.4108, Val Accuracy: 97.15%, F1 Score:  0.770790
Epoch 3/50, Train Loss: 0.0814, Val Loss: 1.4035, Val Accuracy: 97.47%, F1 Score:  0.783416
Epoch 4/50, Train Loss: 0.0636, Val Loss: 1.4011, Val Accuracy: 97.61%, F1 Score:  0.791746
Epoch 5/50, Train Loss: 0.0497, Val Loss: 1.3984, Val Accuracy: 97.78%, F1 Score:  0.800311
Epoch 6/50, Train Loss: 0.0403, Val Loss: 1.3972, Val Accuracy: 97.84%, F1 Score:  0.802253
Epoch 7/50, Train Loss: 0.0337, Val Loss: 1.3969, Val Accuracy: 97.76%, F1 Score:  0.798018
Epoch 8/50, Train Loss: 0.0284, Val Loss: 1.3965, Val Accuracy: 97.83%, F1 Score:  0.805184
Epoch 9/50, Train Loss: 0.0266, Val Loss: 1.3970, Val Accuracy: 97.75%, F1 Score:  0.804026
Epoch 10/50, Train Loss: 0.0224, Val Loss: 1.3966, Val Accuracy: 97.73%, F1 Score:  0.800320
Epoch 11/50, Train Loss: 0.0196, Val Loss: 1.3951, Val Accuracy: 97.79%, F1 Score:  0.802404
Epoch 12/50, Train Loss: 0.0178, Val Loss: 1.3946, Val Accuracy: 97.89%, F1 Score:  0.808534
Epoch 13/50, Train Loss: 0.0163, Val Loss: 1.3959, Val Accuracy: 97.72%, F1 Score:  0.801382
Epoch 14/50, Train Loss: 0.0150, Val Loss: 1.3948, Val Accuracy: 97.82%, F1 Score:  0.804357
Epoch 15/50, Train Loss: 0.0148, Val Loss: 1.3937, Val Accuracy: 97.92%, F1 Score:  0.809733
Epoch 16/50, Train Loss: 0.0131, Val Loss: 1.3943, Val Accuracy: 97.82%, F1 Score:  0.804427
Epoch 17/50, Train Loss: 0.0128, Val Loss: 1.3951, Val Accuracy: 97.76%, F1 Score:  0.802856
Epoch 18/50, Train Loss: 0.0124, Val Loss: 1.3942, Val Accuracy: 97.86%, F1 Score:  0.805437
Epoch 19/50, Train Loss: 0.0105, Val Loss: 1.3948, Val Accuracy: 97.81%, F1 Score:  0.804358
Epoch 20/50, Train Loss: 0.0129, Val Loss: 1.3941, Val Accuracy: 97.89%, F1 Score:  0.806356
Epoch 21/50, Train Loss: 0.0123, Val Loss: 1.3931, Val Accuracy: 97.92%, F1 Score:  0.807781
Epoch 22/50, Train Loss: 0.0104, Val Loss: 1.3931, Val Accuracy: 97.96%, F1 Score:  0.810502
Epoch 23/50, Train Loss: 0.0101, Val Loss: 1.3935, Val Accuracy: 97.87%, F1 Score:  0.803277
Epoch 24/50, Train Loss: 0.0108, Val Loss: 1.3929, Val Accuracy: 97.97%, F1 Score:  0.809395
...
Epoch 48/50, Train Loss: 0.0074, Val Loss: 1.3940, Val Accuracy: 97.83%, F1 Score:  0.811039
Epoch 49/50, Train Loss: 0.0067, Val Loss: 1.3936, Val Accuracy: 97.86%, F1 Score:  0.811313
Epoch 50/50, Train Loss: 0.0070, Val Loss: 1.3932, Val Accuracy: 97.90%, F1 Score:  0.811229
Run time:  36925.5822s
```

Figure 1.3.2.4.1.3: Results of Bi-directional LSTM Model with Dropout on development set

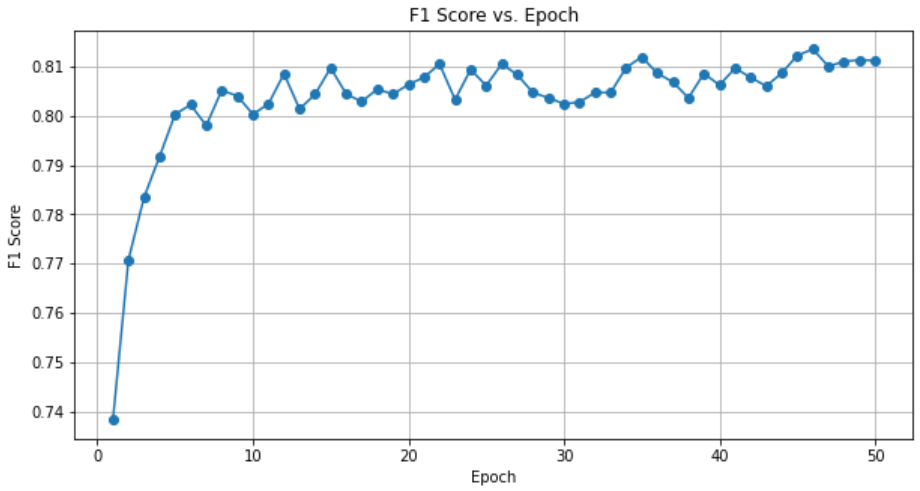


Figure 1.3.2.4.1.4: Graph of validation f1_score against epochs for Bi-directional LSTM Model with Dropout

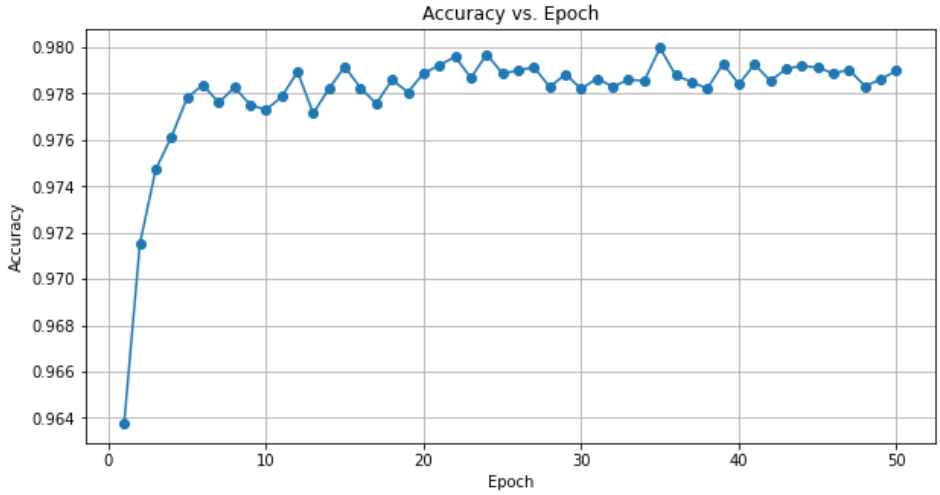


Figure 1.3.2.4.1.5: Graph of validation accuracy against epochs for Bi-directional LSTM Model with Dropout

1.3.2.4.2 Explanation

Dropout is a regularization technique that ignores a randomly selected subset of the hidden neurons on each forward pass during training of the model. The addition of dropout = 0.5 to the bidirectional LSTM model prevents overfitting of the model on the train data by increasing the "noisiness" and making the model more resilient to variations in the data. This improves the generalization of the model and is helpful in the case of limited data. For example, if certain named entity labels have a lower frequency in the training corpus, the model may memorise the

training data instead of learning their underlying patterns. The use of dropout will reduce the chances of the model overfitting on the labels that are overrepresented in the corpus.

1.3.2.5 Summary of Test Results for Various Models

	Single Linear Layer	LSTM	Bi-Directional LSTM	Bi-Directional LSTM with Dropout
Test F1 Score	0.62532	0.73812	0.81941	0.83396
Test Accuracy	94.07%	95.66%	96.78%	97.00%

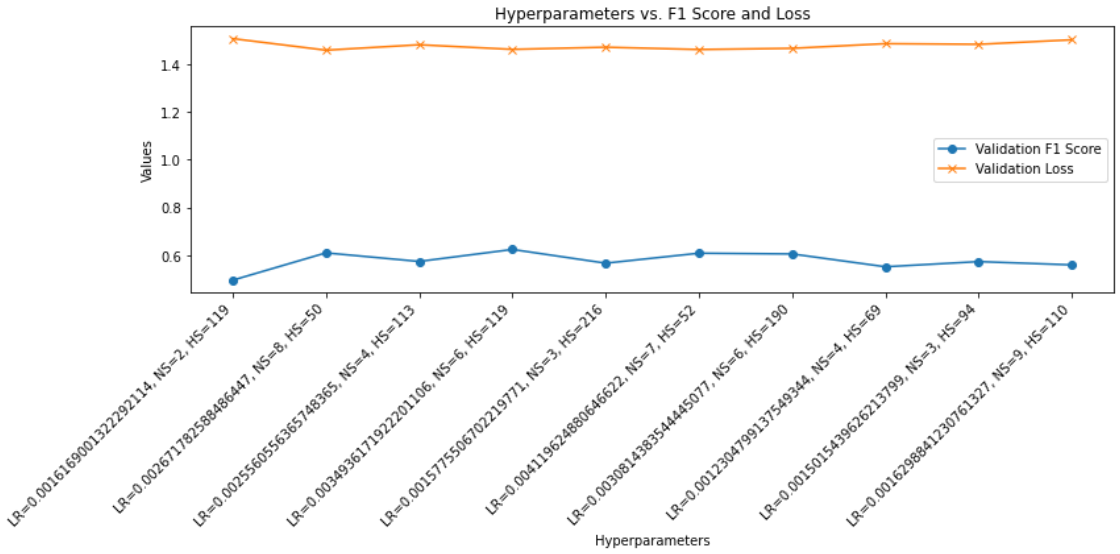
1.3.3 Hyperparameters Tuning

To find the best hyperparameters for our best model, the Bi-directional LSTM model with Dropout, we trained it against varying learning rates (lower limit = 0.001, upper limit = 0.01), batch sizes in terms of sentences (lower limit = 2, upper limit = 10) and hidden neuron numbers (lower limit = 32, upper limit = 256). We employed Optuna to find the best hyperparameters, and conducted 10 trials. The sampler chosen was TPESampler(). Due to the large dataset, and hence high run time and heavy use of computation units, we employed only 200 sentences out of the entire dataset to train the model.

To ensure that each tag type was represented in the dataset, 10 different sentences containing each tag type were first randomly selected (to ensure each tag type would appear at least 10 times in the dataset). Duplicate sentences (appear for 2 or more tag types) were removed. The remaining sentences (out of 200) were selected at random to prevent a skewed dataset, while taking care to exclude sentences that were already included in the smaller dataset.

```
0    21563
8     2648
7      167
6      149
4      138
5       94
3       20
2       11
1       10
Name: label_encoded, dtype: int64
```

1.3.3.1 Label frequencies for reduced train dataset



1.3.3.2: Graph of F1 Score and Loss for each set of Hyperparameters

The optimal hyperparameters identified were:

Learning Rate = 0.0035

Number of Sentences in Batch = 6

Hidden Layer Neurons = 119

1.3.4 Testing against Bi-directional LSTM Model with Droupout

1.3.4.1 Experiment and Results

Number of epochs used for training: 24/50

Run time: 17437.0380s

Results of the test set (including f1_score):

```
Predicted class: tensor([8, 8, 4,  ..., 7, 8, 8])
True class: tensor([8, 8, 4,  ..., 7, 8, 8])
Test loss: 0.20715296268463135
F1 Score: 0.8283925801744163
Test Accuracy: 96.98%
```

Figure 1.3.4.1: Results of Bi-directional LSTM Model with Dropout on test set

Evaluation based on the development set (including f1_score):

```
Training with best hyperparameters: {'lr': 0.0034936171922201106, 'number_of_sentences': 6, 'hidden_size': 119}
Starting model training...
Epoch 1/50, Train Loss: 0.2031, Val Loss: 1.4215, Val Accuracy: 97.15%, F1 Score:  0.832933
Epoch 2/50, Train Loss: 0.1178, Val Loss: 1.4110, Val Accuracy: 97.55%, F1 Score:  0.852264
Epoch 3/50, Train Loss: 0.0937, Val Loss: 1.4042, Val Accuracy: 97.84%, F1 Score:  0.869018
Epoch 4/50, Train Loss: 0.0760, Val Loss: 1.4015, Val Accuracy: 97.91%, F1 Score:  0.869283
Epoch 5/50, Train Loss: 0.0672, Val Loss: 1.3987, Val Accuracy: 97.89%, F1 Score:  0.867577
Epoch 6/50, Train Loss: 0.0592, Val Loss: 1.3977, Val Accuracy: 97.88%, F1 Score:  0.867261
Epoch 7/50, Train Loss: 0.0547, Val Loss: 1.3962, Val Accuracy: 97.96%, F1 Score:  0.872005
Epoch 8/50, Train Loss: 0.0517, Val Loss: 1.3968, Val Accuracy: 97.86%, F1 Score:  0.868966
Epoch 9/50, Train Loss: 0.0455, Val Loss: 1.3958, Val Accuracy: 97.96%, F1 Score:  0.874304
Epoch 10/50, Train Loss: 0.0444, Val Loss: 1.3963, Val Accuracy: 97.88%, F1 Score:  0.863054
Epoch 11/50, Train Loss: 0.0423, Val Loss: 1.3949, Val Accuracy: 97.94%, F1 Score:  0.872287
Epoch 12/50, Train Loss: 0.0390, Val Loss: 1.3948, Val Accuracy: 98.00%, F1 Score:  0.872996
Epoch 13/50, Train Loss: 0.0370, Val Loss: 1.3943, Val Accuracy: 97.93%, F1 Score:  0.873145
Epoch 14/50, Train Loss: 0.0374, Val Loss: 1.3934, Val Accuracy: 98.08%, F1 Score:  0.879081
Epoch 15/50, Train Loss: 0.0345, Val Loss: 1.3935, Val Accuracy: 98.01%, F1 Score:  0.874966
Epoch 16/50, Train Loss: 0.0342, Val Loss: 1.3940, Val Accuracy: 98.02%, F1 Score:  0.874817
Epoch 17/50, Train Loss: 0.0326, Val Loss: 1.3931, Val Accuracy: 98.07%, F1 Score:  0.877735
Epoch 18/50, Train Loss: 0.0319, Val Loss: 1.3932, Val Accuracy: 98.08%, F1 Score:  0.877342
Epoch 19/50, Train Loss: 0.0317, Val Loss: 1.3931, Val Accuracy: 98.05%, F1 Score:  0.877680
Epoch 20/50, Train Loss: 0.0313, Val Loss: 1.3932, Val Accuracy: 98.06%, F1 Score:  0.875934
Epoch 21/50, Train Loss: 0.0300, Val Loss: 1.3932, Val Accuracy: 98.04%, F1 Score:  0.874059
Epoch 22/50, Train Loss: 0.0286, Val Loss: 1.3926, Val Accuracy: 98.07%, F1 Score:  0.874261
Epoch 23/50, Train Loss: 0.0302, Val Loss: 1.3927, Val Accuracy: 98.03%, F1 Score:  0.875411
Epoch 24/50, Train Loss: 0.0285, Val Loss: 1.3933, Val Accuracy: 98.03%, F1 Score:  0.875943
No improvement in F1 score for 10 epochs. Stopping training.
Run time:  17437.0380s
```

Figure 1.3.4.2: Results of Bi-directional LSTM Model with Dropout on Development set

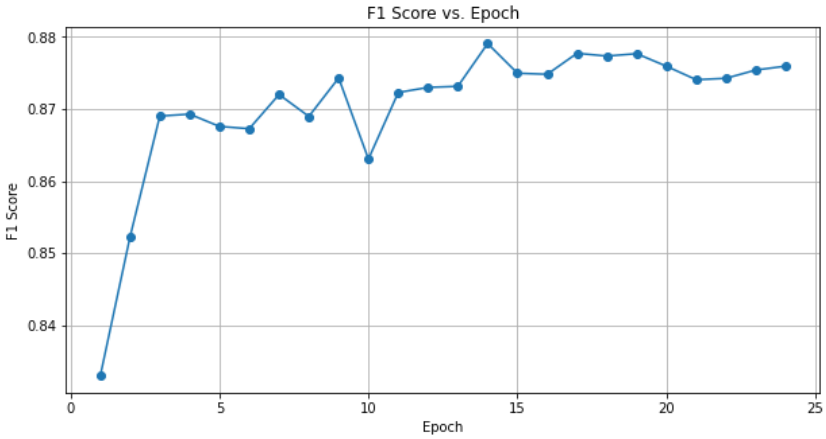


Figure 1.3.4.3: Graph of validation f1_score against epochs for Bi-directional LSTM Model with Dropout

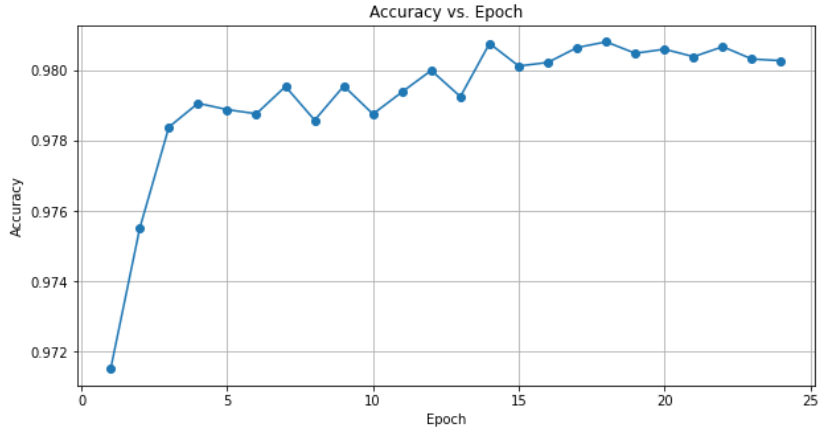


Figure 1.3.4.4: Graph of validation accuracy against epochs for Bi-directional LSTM Model with Dropout

1.3.4.2 Explanation

From the results, it can be seen that although the F1 validation scores increase significantly (to about 0.875), the model is unable to generalise to the test data as well (F1 score for test data is only 0.828). Hence, in the future, additional methods like L2 regularisation, gradient clipping, batch normalization, or varying the degree of dropout, can be explored in order to prevent the model from overfitting on the train data.

Part 2. Sentence-Level Categorization: Question Classification

2.1 Formation of New Classes

(q2a) The new classes formed after combining classes 4 and 5 into 'OTHERS' are:

- 0
- 1
- 2
- 3
- 4 (represent ‘OTHERS’)

```
    # Combine class 4 and 5 into 'OTHERS', which will be class 4

    train['label-coarse'] = train['label-coarse'].replace({4: 4, 5: 4})
    test['label-coarse'] = test['label-coarse'].replace({4: 4, 5: 4})
    val['label-coarse'] = val['label-coarse'].replace({4: 4, 5: 4})
    print(val)
```

```
     label-coarse  label-fine  \
0               4          21
1               1           2
2               0           9
3               0          12
4               3           4
..            ...         ...
495             1          37
496             4          11
497             3           5
498             4          24
499             0          12
```

Figure 2.1: Code for combining classes 4 and 5 into ‘OTHERS’

2.2 Word Embedding

We implemented word embedding using **word2vec** to transform textual data into numerical representations. The process begins by breaking down the text into individual words and then assigning **word2vec** embeddings to each word, using a zero vector for words not present in the **word2vec** model. We did this for the training, validation, and test datasets.

To ensure that the questions of varying numbers of words have the same length across all our datasets, we further applied **pad_sequence** to pad shorter sequences with zeros.
These pre-trained word embeddings will later be used as the input to all our models.

```
# Convert text data to Word2Vec embeddings
def text_to_embeddings(text, w2v_model):
    tokens = text.split()
    embeddings = [torch.tensor(w2v_model[word]) if word in w2v_model else torch.zeros(300) for word in tokens]
    return torch.stack(embeddings)

# Apply Word2Vec embeddings to data
X_train_embedded = [text_to_embeddings(text, w2v) for text in train['text']]
y_train_embedded = torch.tensor(train['label-coarse'])

X_val_embedded = [text_to_embeddings(text, w2v) for text in val['text']]
y_val_embedded = torch.tensor(val['label-coarse'])

X_test_embedded = [text_to_embeddings(text, w2v) for text in test['text']]
y_test_embedded = torch.tensor(test['label-coarse'])

# Convert the list of embedded sequences to a padded tensor
X_train_embedded= pad_sequence(X_train_embedded, batch_first=True)
X_val_embedded= pad_sequence(X_val_embedded, batch_first=True)
X_test_embedded= pad_sequence(X_test_embedded, batch_first=True)
```

Figure 2.2: Code for converting text data into word2vec embeddings

2.3 Neural Network

LSTM (Long Short-Term Memory) neural network was employed because it excels in processing sequential data by capturing long-term dependencies, which is important for understanding the semantics of the questions and improving question classification accuracy.

The layers in our neural network include:

- LSTM Layer (**nn.LSTM**)
    - Responsible for understanding the sequential patterns and long-term dependencies in the input data
    - Captures how words in a question relate to each other over time, enabling the model to understand the question's context and structure
- Linear Layer (**nn.Linear**)
    - Takes the information learned by the LSTM layer and transforms it into a format suitable for classification
    - Reduces the dimensionality of the data and prepares it for making predictions
- Softmax Layer (**nn.Softmax**)
    - Converts the transformed data into a probability distribution over different classes.

2.4 Different Types of Aggregation Methods (q2b)

We tried 3 types of aggregation methods to aggregate the word representations to represent each question. These aggregation methods would be used on the output vector for each word in the input sequence that is generated by the LSTM layer. They will all be used along the 1st dimension of the LSTM output, which represents the number of time steps or words in the sequence. The result will then be passed into the linear layer.

2.4.1 Averaging Over Word Representations

This method takes the average of the word representations by using **torch.mean**, which takes the mean of the vectors. This averaging operation condenses the information from all the words into a single vector, creating a sentence-level representation.

```
class AverageAggregation(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(AverageAggregation, self).__init__()
        self.lstm = nn.LSTM(input_dim, no_hidden)
        self.linear = nn.Linear(no_hidden, output_dim)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        lstm_out, _ = self.lstm(x)
        # Average over word representations
        sentence_representation = torch.mean(lstm_out, dim=1)
        # Linear layer for classification
        output = self.linear(sentence_representation)
        # Softmax activation for classification
        output = self.softmax(output)

        return output
```

Figure 2.4.1: Model that uses averaging over word representations

2.4.2 Max Pooling

This method captures the most significant features from a sequence of word-level representations by using the **torch.max** function to extract the maximum value from these vectors.

```
class MaxPoolAggre(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(MaxPoolAggre, self).__init__()
        self.lstm = nn.LSTM(input_dim, no_hidden)
        self.linear = nn.Linear(no_hidden, output_dim)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        lstm_out, _ = self.lstm(x)
        # Max pooling over word representations
        sentence_representation, _ = torch.max(lstm_out, dim=1)
        # Linear layer for classification
        output = self.linear(sentence_representation)
        # Softmax activation for classification
        output = self.softmax(output)

        return output
```

Figure 2.4.2: Model that uses max pooling

2.4.3 Taking Representation of the Last Word

This method involves selecting the word-level vector representing the final word in the sequence.

```python
class LastWordAggre(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(LastWordAggre, self).__init__()
        self.lstm = nn.LSTM(input_dim, no_hidden)
        self.linear = nn.Linear(no_hidden, output_dim)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        lstm_out, _ = self.lstm(x)
        # Use the representation of the last word
        sentence_representation = lstm_out[:, -1, :]
        # Linear layer for classification
        output = self.linear(sentence_representation)
        # Softmax activation for classification
        output = self.softmax(output)

        return output
```

Figure 2.4.3: Model for taking representation of the last word

2.5 Hyperparameter tuning for each of the aggregation methods

We performed simple hyperparameter tuning by varying the learning rate **[0.001, 0.01, 0.1]** and batch size **[32, 64, 128, 256].** This was done on all 3 aggregation models to find the model with the highest validation accuracy to be used for subsequent steps.

Performing hyperparameter tuning is a critical procedure as it helps to optimise the model's performance.

```
# finding optimal hyperparameters
avg_agreg_acc = []
avg_agreg_loss = []

for learning_rate in [0.001, 0.01, 0.1]:
    for batch_size in [32, 64, 128, 256]:
        # Set hyperparameters
        learning_rate = learning_rate
        batch_size = batch_size

        print(f'Learning Rate: {learning_rate}, Batch Size: {batch_size}')

        # Create DataLoader for training data
        train_dataset = TensorDataset(X_train_embedded, y_train_embedded)
        train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)

        aa_model = AverageAggregation(input_dim, output_dim)

        accuracy, loss = train_model(aa_model, 100, learning_rate, train_dataset, train_loader)
        avg_agreg_acc.append(accuracy)
        avg_agreg_loss.append(loss)
```

Figure 2.5: Hyperparameter tuning for max pooling aggregation model

2.5.1 Performance Evaluation of Hyperparameter Tuning for Various Aggregation Methods

Averaging over word representations has the highest validation accuracy of 0.858 and the lowest training loss of 0.943 when the *learning rate = 0.01* and *batch size = 32*

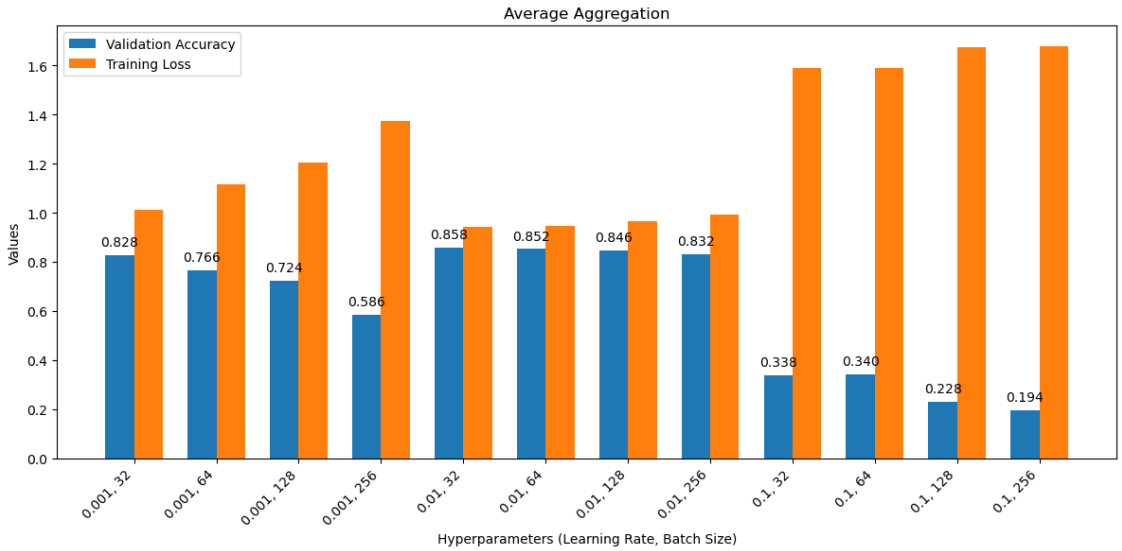


Figure 2.5.1.1: Performance comparison of hyperparameter tuning for average aggregation

Using the optimal hyperparameters (learning rate = 0.01 and batch size = 32), we train the model again to attain its optimal validation accuracy of 0.846 and training loss of 0.955 as shown in the code below.

```
learning_rate = 0.01
batch_size = 32

aa_model = AverageAggregation(input_dim, output_dim)

train_dataset = TensorDataset(X_train_embedded, y_train_embedded)
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)

train_model(aa_model, 100, learning_rate, train_dataset, train_loader)
```

✓ 21.7s

```
Early stopping after 18 epochs
Validation accuracy: 0.846
Training loss: 0.9554366969293163
```

Figure 2.5.1.2: Performance with optimal hyperparameters for average aggregation

Max pooling has the highest validation accuracy of 0.880 and the lowest training loss of 0.944 when the *learning rate = 0.01* and *batch size = 128*

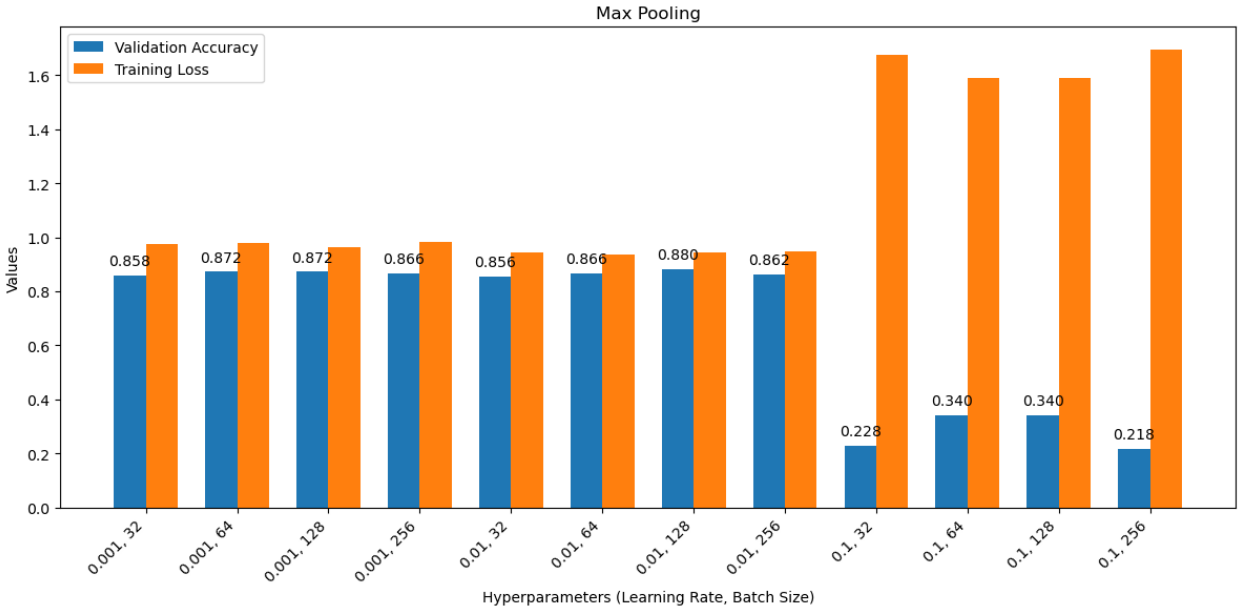


Figure 2.5.1.3: Performance comparison of hyperparameter tuning for max pooling

Using the optimal hyperparameters (learning rate = 0.01 and batch size = 128), we train the model again to attain its optimal validation accuracy of 0.87 and training loss of 0.941 as shown in the figure below.

```
# train with optimal hyperparameters (highest accuracy, lowest loss)
learning_rate = 0.01
batch_size = 128

mpa_model = MaxPoolAggre(input_dim, output_dim)

train_dataset = TensorDataset(X_train_embedded, y_train_embedded)
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)

train_model(mpa_model, 100, learning_rate, train_dataset, train_loader)
```

✓ 21.7s

```
Early stopping after 20 epochs
Validation accuracy: 0.878
Training loss: 0.941215299643003
```

Figure 2.5.1.4: Performance with optimal hyperparameters for max pooling

Taking representation of the last word has the highest validation accuracy of 0.340 and the lowest low training loss of 1.554 when the *learning rate = 0.001* and *batch size = 64*

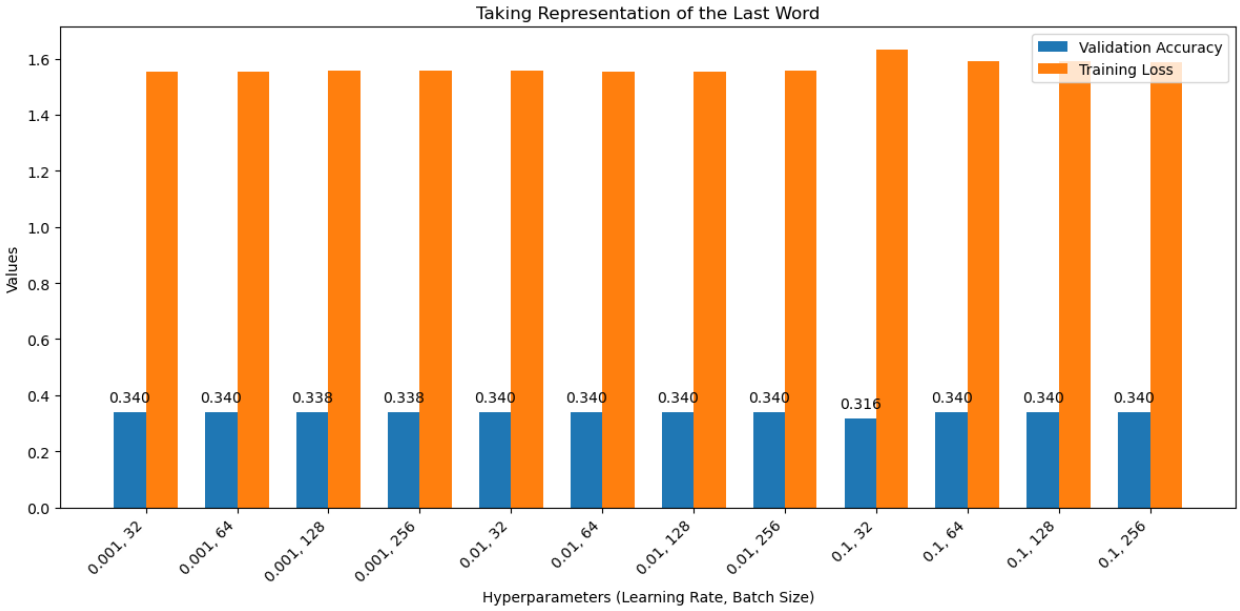


Figure 2.5.1.5: Performance comparison of hyperparameter tuning for taking representation of last word

Using the optimal hyperparameters (learning rate = 0.001 and batch size = 64), we train the model again to attain its optimal validation accuracy of 0.338 and training loss of 1.555 as shown in the figure below.

```
# train with optimal hyperparameters (highest accuracy, lowest loss)
learning_rate = 0.001
batch_size = 64

lwa_model = LastWordAggre(input_dim, output_dim)

train_dataset = TensorDataset(X_train_embedded, y_train_embedded)
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)

train_model(lwa_model, 100, learning_rate, train_dataset, train_loader)
```

✓ 6.7s

```
Early stopping after 6 epochs
Validation accuracy: 0.338
Training loss: 1.5553957942204597
```

Figure 2.5.1.6: Performance with optimal hyperparameters for taking representation of last word

The following table summarizes the validation accuracies across all three aggregation methods and their optimal learning rate and batch size.

Aggregation method	Validation accuracy	Optimal learning rate	Optimal batch size
Averaging Over Word Representations	0.846	0.01	32
Max Pooling	0.870	0.01	128
Taking Representation of the Last Word	0.338	0.001	64

2.6 Final aggregation method adopted

The table above shows that the model with max pooling has the highest validation accuracy of 0.870 hence **max pooling** will be adopted.

Possible reasons behind max pooling's performance

- Max pooling can capture the most significant features in a sentence, resulting in a more efficient sentence representation for question classification.
- It reduces noise by selecting the maximum value, which is more likely to correspond to important information and less affected by irrelevant words or noise in the data.

2.7 Neural Network Architecture & Mathematical Functions of Each Layer for Forward Computation (q2c)

The following figure shows the final neural network architecture adopted.

```
class QnsClassifier(nn.Module):
    def __init__(self, input_dim, output_dim, no_hidden):
        super(QnsClassifier, self).__init__()
        self.lstm = nn.LSTM(input_dim, no_hidden)
        self.linear = nn.Linear(no_hidden, output_dim)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        lstm_out, _ = self.lstm(x)
        # Max pooling over word representations
        sentence_representation, _ = torch.max(lstm_out, dim=1)

        output = self.linear(sentence_representation)
        # Softmax activation for classification
        output = self.softmax(output)

        return output
```

Figure 2.7.1: Structure of final model used

For mathematical functions, please refer to Appendix A.

2.8 Network Configuration

2.8.1 Updated Parameters

(q2c) The following shows the updated parameters in the network, including the weights and biases of the LSTM layer (comprising 2 weights and 2 biases) and the linear layer (comprising of 1 weight and 1 bias), along with their respective sizes. Additionally, the length of the final vector representation of each word fed into Softmax is **221**.

```
model = QnsClassifier(input_dim, output_dim, no_hidden)

for name, param in model.named_parameters():
    if param.requires_grad:
        print(name, param.shape)
```

```
lstm.weight_ih_l0 torch.Size([884, 300])
lstm.weight_hh_l0 torch.Size([884, 221])
lstm.bias_ih_l0 torch.Size([884])
lstm.bias_hh_l0 torch.Size([884])
linear.weight torch.Size([5, 221])
linear.bias torch.Size([5])
```

2.8.2 Length of final vector representation of each word fed into Softmax

2.9 Hyperparameter tuning using Optuna

Hyperparameter tuning was also performed on the final neural network architecture using Optuna as shown in the following figure.

```python
import optuna

def objective(trial):

    no_hidden = trial.suggest_int('no_hidden', 32, 256)
    learning_rate = trial.suggest_float('learning_rate', 1e-5, 1e-1, log=True)
    batch_size = trial.suggest_int('batch_size', 32, 256)

    model = QnsClassifier(input_dim, output_dim, no_hidden)

    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

    train_dataset = TensorDataset(X_train_embedded, y_train_embedded)
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)

    accuracy, loss = train_model(model, 100, learning_rate, train_dataset, train_loader)

    return loss

study = optuna.create_study(direction='minimize')
study.optimize(objective, n_trials=50)
study.best_params
```

Figure 2.9.1: Hyperparameter tuning using Optuna

A portion of the output shown during training is shown below.

```
[I 2023-11-03 14:55:33,847] A new study created in memory with name: no-name-e3409d03-da04-4731-88f3-674bb93b6785
[I 2023-11-03 15:00:42,280] Trial 0 finished with value: 1.1901525486083258 and parameters: {'no_hidden': 159, 'learning_rate': 0.0001056027041153137, 'batch_size': 246}. Best is trial 0 with value: 1.1901525486083258.
Early stopping after 68 epochs
Validation accuracy: 0.762
Training loss: 1.1901525486083258

[I 2023-11-03 15:01:07,061] Trial 1 finished with value: 0.9530293229553435 and parameters: {'no_hidden': 44, 'learning_rate': 0.05005603446110368, 'batch_size': 140}. Best is trial 1 with value: 0.9530293229553435.
Early stopping after 17 epochs
Validation accuracy: 0.856
Training loss: 0.9530293229553435

[I 2023-11-03 15:03:46,265] Trial 2 finished with value: 1.1588554514778986 and parameters: {'no_hidden': 94, 'learning_rate': 0.0001428556809960306 7, 'batch_size': 186}. Best is trial 1 with value: 0.9530293229553435.
Early stopping after 60 epochs
Validation accuracy: 0.776
Training loss: 1.1588554514778986

[I 2023-11-03 15:08:12,716] Trial 3 finished with value: 1.0406562398586954 and parameters: {'no_hidden': 182, 'learning_rate': 0.0001778508373179377, 'batch_size': 89}. Best is trial 1 with value: 0.9530293229553435.
Early stopping after 52 epochs
Validation accuracy: 0.832
Training loss: 1.0406562398586954

[I 2023-11-03 15:13:04,412] Trial 4 finished with value: 1.1006257840105005 and parameters: {'no_hidden': 253, 'learning_rate': 0.0001113956930922330 5, 'batch_size': 67}. Best is trial 1 with value: 0.9530293229553435.
Early stopping after 35 epochs
Validation accuracy: 0.796
Training loss: 1.1006257840105005

[I 2023-11-03 15:13:45,668] Trial 5 finished with value: 1.5448125749826431 and parameters: {'no_hidden': 172, 'learning_rate': 5.828084502906078e-05, 'batch_size': 209}. Best is trial 1 with value: 0.9530293229553435.
Early stopping after 8 epochs
Validation accuracy: 0.34
Training loss: 1.5448125749826431
```

Figure 2.9.2: Optuna hyperparameter tuning output

Benefits of Optuna

- It employs sophisticated algorithms for hyperparameter optimization, such as Tree-structured Parzen Estimator (TPE), which efficiently explores a wide hyperparameter space to find optimal configurations with fewer trials.
- It automates the process of hyperparameter tuning, reducing the manual effort required to find the best configuration for a machine-learning model.

Optimised hyperparameters

Shown below are the optimized hyperparameters obtained through Optuna, which will be used for subsequent steps.

```
{'no_hidden': 221, 'learning_rate': 0.011812627647641677, 'batch_size': 32}
```

However, Optuna was not employed for hyperparameter tuning in other models due to its long execution time.

2.10 Applying Trained Model on Test Set

(q2d) The number of epochs initially set was 100, however, due to early stopping with patience set to 5, the number of epochs eventually used for training was **18** and the training time was **200.76** seconds.

```
Epoch 18/100
Training Accuracy: 0.9773828756058158 | Training Loss: 0.9310145470403856
Val Accuracy: 0.864 | Val Loss: 1.0442644357681274
Test Accuracy: 0.89 | Test Loss: 1.014546275138855
Early stopping after 18 epochs. No improvement on the development set.
Number of epochs used for training: 18
Training time: 200.76 seconds
```

Figure 2.10.1: Total number of training epochs & training time

```python
# Check for early stopping
if val_accuracy > best_val_accuracy:
    best_val_accuracy = val_accuracy
    no_improvement = 0
else:
    no_improvement += 1

if no_improvement >= patience:
    end_time = time.time()
    training_time = end_time - start_time
    print(f"Early stopping after {epoch + 1} epochs. No improvement on the development set.")
    print(f"Number of epochs used for training: {epoch + 1}")
    print(f"Training time: {training_time:.2f} seconds")
    break
```

Figure 2.10.2: Code for early stopping

(q2e) The accuracies and losses for the train, development, and test sets for each epoch are shown below.

Epoch 1/100

Training Accuracy: 0.6736672051696284 | Training Loss: 1.2409739671214934

Val Accuracy: 0.812 | Val Loss: 1.1065130233764648

Test Accuracy: 0.814 | Test Loss: 1.0974669456481934

Epoch 2/100
Training Accuracy: 0.8556138933764136 | Training Loss: 1.060854919110575
Val Accuracy: 0.838 | Val Loss: 1.0715059041976929
Test Accuracy: 0.884 | Test Loss: 1.0322977304458618

Epoch 3/100
Training Accuracy: 0.8939822294022617 | Training Loss: 1.0206965358026565
Val Accuracy: 0.85 | Val Loss: 1.0630159378051758
Test Accuracy: 0.848 | Test Loss: 1.0544860363006592

Epoch 4/100
Training Accuracy: 0.9186187399030694 | Training Loss: 0.9964387739858319
Val Accuracy: 0.854 | Val Loss: 1.0505754947662354
Test Accuracy: 0.88 | Test Loss: 1.0260858535766602

Epoch 5/100
Training Accuracy: 0.9301292407108239 | Training Loss: 0.9821789107015056
Val Accuracy: 0.856 | Val Loss: 1.0496762990951538
Test Accuracy: 0.892 | Test Loss: 1.0228513479232788

Epoch 6/100
Training Accuracy: 0.9384087237479806 | Training Loss: 0.9732751996286454
Val Accuracy: 0.868 | Val Loss: 1.0463874340057373
Test Accuracy: 0.892 | Test Loss: 1.0190653800964355

Epoch 7/100
Training Accuracy: 0.9478998384491115 | Training Loss: 0.9640784844275444
Val Accuracy: 0.852 | Val Loss: 1.0522032976150513
Test Accuracy: 0.886 | Test Loss: 1.0203877687454224

Epoch 8/100
Training Accuracy: 0.9501211631663974 | Training Loss: 0.9594281631131326
Val Accuracy: 0.868 | Val Loss: 1.0387200117111206
Test Accuracy: 0.892 | Test Loss: 1.0159188508987427

Epoch 9/100
Training Accuracy: 0.9573909531502424 | Training Loss: 0.9511780238920643
Val Accuracy: 0.874 | Val Loss: 1.0374387502670288
Test Accuracy: 0.894 | Test Loss: 1.0158793926239014

Epoch 10/100
Training Accuracy: 0.9610258481421647 | Training Loss: 0.9478793874863656
Val Accuracy: 0.856 | Val Loss: 1.0480659008026123
Test Accuracy: 0.886 | Test Loss: 1.0228153467178345

Epoch 11/100
Training Accuracy: 0.965670436187399 | Training Loss: 0.9432236832957114
Val Accuracy: 0.864 | Val Loss: 1.037244439125061
Test Accuracy: 0.89 | Test Loss: 1.0168606042861938

Epoch 12/100
Training Accuracy: 0.9691033925686591 | Training Loss: 0.9395661415592317
Val Accuracy: 0.878 | Val Loss: 1.0312124490737915
Test Accuracy: 0.904 | Test Loss: 1.0036048889160156

Epoch 13/100
Training Accuracy: 0.9739499192245558 | Training Loss: 0.9339110466741747
Val Accuracy: 0.884 | Val Loss: 1.0257093906402588
Test Accuracy: 0.916 | Test Loss: 0.9931655526161194

Epoch 14/100
Training Accuracy: 0.9759693053311793 | Training Loss: 0.9311764570974534
Val Accuracy: 0.884 | Val Loss: 1.0251436233520508
Test Accuracy: 0.91 | Test Loss: 0.9998008012771606

Epoch 15/100
Training Accuracy: 0.9787964458804523 | Training Loss: 0.9289980911439465
Val Accuracy: 0.874 | Val Loss: 1.0241914987564087
Test Accuracy: 0.906 | Test Loss: 0.9991756081581116

Epoch 16/100

Training Accuracy: 0.97859450726979 | Training Loss: 0.9284598977335038

Val Accuracy: 0.884 | Val Loss: 1.0234572887420654

Test Accuracy: 0.914 | Test Loss: 0.9998930096626282

Epoch 17/100

Training Accuracy: 0.97859450726979 | Training Loss: 0.9283658969786859

Val Accuracy: 0.868 | Val Loss: 1.0354459285736084

Test Accuracy: 0.898 | Test Loss: 1.0034273862838745

Epoch 18/100

Training Accuracy: 0.9773828756058158 | Training Loss: 0.9310145470403856

Val Accuracy: 0.864 | Val Loss: 1.0442644357681274

Test Accuracy: 0.89 | Test Loss: 1.014546275138855

The epoch with the highest test accuracy and lowest test loss is epoch **13** with an accuracy of **0.916** and loss of **0.9931655526161194**.

Plotting the losses and accuracies across the training, testing, and validation sets, it becomes evident that the losses generally decrease for all datasets, while the accuracies generally improve for all datasets until the epoch 18.

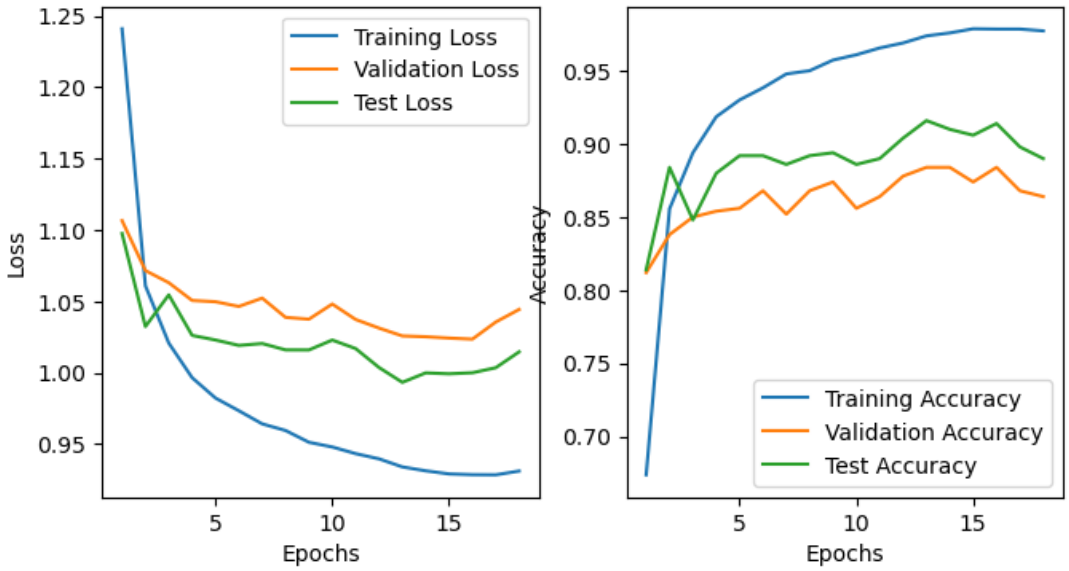


Figure 2.10.3: Accuracies and loss against number of epoch

2.11 Alternative Model

2.11.1 Linear Model

We experimented with a linear model which employs a linear layer followed by softmax activation for classification.

```
class LinearClassifier(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(LinearClassifier, self).__init__()
        self.linear = nn.Linear(input_dim, output_dim)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        linear_out = self.linear(x)
        sentence_representation, _ = torch.max(linear_out, dim=1)
        output = self.softmax(sentence_representation)
        return output

model = LinearClassifier(input_dim, output_dim)
```

Figure 2.11.1: Structure of Linear model

Plotting the losses and accuracies across the training, testing, and validation sets, we get a similar trend where the losses generally decrease and the accuracies generally improve for all datasets until epoch 27.

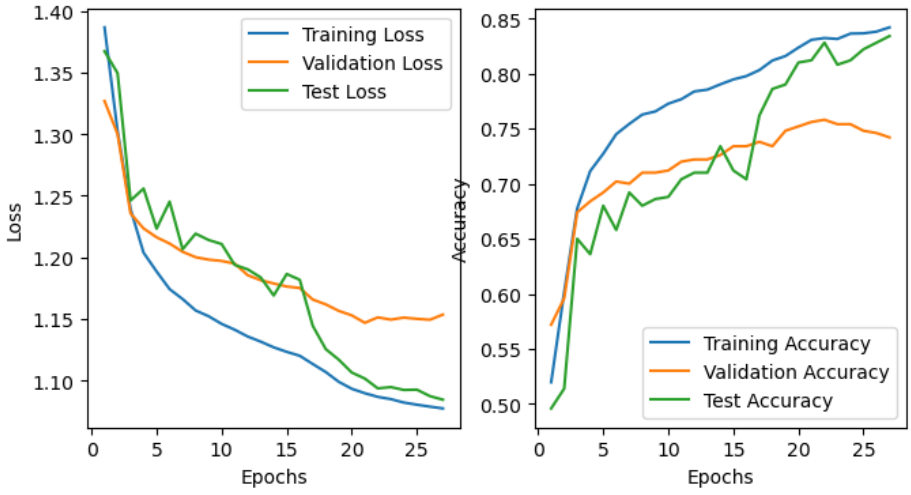


Figure 2.11.2: Accuracies and loss against number of epoch for linear model

However, this model was not employed as it has a lower test accuracy of **0.834** and a higher test loss of **1.085**.

```
Epoch 27/100
Training Accuracy: 0.8418820678513732 | Training Loss: 1.0774819743248725
Val Accuracy: 0.742 | Val Loss: 1.1536140441894531
Test Accuracy: 0.834 | Test Loss: 1.084571361541748
```

Figure 2.11.3: Performance on Training, Validation, and Test Sets

Appendix A - Mathematical Functions

1. LSTM
    a. Input Gate (i(t)): $\sigma(U_i^T x(t) + W_i^T h(t-1) + b_i)$
    b. Forget Gate (f(t)): $\sigma(U_f^T x(t) + W_f^T h(t-1) + b_f)$
    c. Ouput Gate (o(t)): $\sigma(U_o^T x(t) + W_o^T h(t-1) + b_o)$
    d. Cell State Candidate (g(t)): $\phi(U_c^T x(t) + W_c^T h(t-1), + b_c)$
    e. Cell State (c(t)): $g(t) \odot i(t) + c(t-1) \odot f(t)$
    f. Hidden State (h(t)): $\phi(c(t)) \odot o(t)$

    where

    - σ is the sigmoid activation function
    - ϕ is the tanh activation function
    - $x(t)$ is the input at time t
    - $h(t-1)$ is the previous hidden state
    - U_i^T, U_f^T, U_o^T, U_c^T are weight vectors of the input
    - W_i^T, W_f^T, W_o^T, W_c^T are weight vectors of the previous hidden state
    - b_i, b_f, b_o, b_c are bias vectors

2. Max Pooling

$$sentence\_representation[b,d] = maxt(lstm\_out[b,t,d])$$

    where

    - lstm_out is a 3D tensor with shape (batch_size, sequence_length, hidden_dim)
    - batch_size is the number of sequences in a batch
    - sequence_length is the length of the input sequence
    - hidden_dim is the dimension of the hidden representations
    - b is an element in the batch
    - d is a dimension in hidden_dim

3. Linear layer
    a. $U = XW + B$
    b. $Y = f(U) = XW + B$

    where

    - X is the input data
    - W is the weight matrix
    - B is the bias vector

4. Softmax layer
    a. $U = XW + B$
    b. $f(U) = \frac{eU}{\sum_{k=1}^{K} eUk}$
    c. $y = argmax_k f(U)$

    where

    - X is the input data
    - W is the weight matrix
    - B is the bias vector