NANYANG TECHNOLOGICAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Assignment for SC4002 / CE4045 / CZ4045

AY2023-2024

Group ID:

Group members:

Name	Matric No.	Contribution
Gordon Tian Xiao Chen	U2140820A	Q1
Min Khant Htoo	U2140545E	Q1
Alviento Adrian Nicolas Belleza	U2140615H	Q1
Yeo Zong Han	U2140665J	Q2
Wong Chu Feng	U2140100D	Q2

Question 1

Part 1.1

In order to determine the most similar words to ‘student’, ‘Apple’ and ‘apple’ respectively, we utilise the Word2Vec.most_similar() method in Gensim. This method computes cosine similarity between a simple mean of the projection weight vectors of the given words and the vectors for each word in the model. The method corresponds to the word-analogy and distance scripts in the original word2vec implementation.

ANSWER:

Word	Most Similar Word	Cosine Similarity
student	students	0.729
Apple	Apple_AAPL	0.746
apple	apples	0.720

Part 1.2a

The function, process_sets(filepath), reads a dataset file, splits it into sentences, and extracts the words and their corresponding NER tags by selecting the first and last columns of each line respectively. The returned data is organised into a 3-dimensional array, where the dimensions represent sentences, words, and word-related information (Word and NER tag).

Some lines contain `-DOCSTART- -X- O O` that is used to indicate the beginning of a new document in the dataset rather than a sentence. Hence, these lines were omitted from the data.

ANSWER:

Set	Sentences	IOB2 Tags
Training set (eng.train)	14987	`'I-ORG', 'O', 'I-MISC', 'I-PER', 'I-LOC', 'B-LOC', 'B-MISC', 'B-ORG'`
Development set (eng.testa)	3466	`'O', 'I-ORG', 'I-LOC', 'I-MISC', 'I-PER', 'B-MISC'`
Test set (eng.testb)	3684	`'O', 'I-LOC', 'I-PER', 'I-MISC', 'I-ORG', 'B-ORG', 'B-MISC', 'B-LOC'`

Complete set of tags:

```
{'B-LOC', 'B-MISC', 'B-ORG', 'I-LOC', 'I-MISC', 'I-ORG', 'I-PER', 'O'}
```

Part 1.2b

The pseudocode implemented below is designed to recognize named entities within a text string, with a specific requirement that there must be at least two named entities, and each identified entity should consist of more than one word.

```
FOR each sentence in trainset:
    named_entities = []
    previous_entity  = ‘’
    FOR each word in sentence:
        IF word’s tag is ‘O’:
            APPEND previous_entity to named_entities
            previous_entity  = ‘’
        ELIF word’s tag is ‘B’ or ‘I’:
            IF word succeeds an ‘O’ tag OR  word is first word of the sentence:
                APPEND word to previous_entity
            ELSE:
                IF word’s tag is ‘B’ or entity type is not the same as previous_entity:
                    APPEND previous_entity to named_entities
                    previous_entity = ‘’
                ELSE:
                    APPEND word to previous_entity
                IF word is last word of the sentence:
                    APPEND previous_entity to named_entities
    IF there exists at least 2 named_entities that consist of more than one word:
        BREAK
```

ANSWER:

Example sentence:

```
" What we have to be extremely careful of is how other countries are going to take Germany's lead , " Welsh National Farmers' Union ( NFU ) chairman John Lloyd Jones said on BBC radio .
```

Sentence tags:

```
[['"', 'O'], ['What', 'O'], ['we', 'O'], ['have', 'O'], ['to', 'O'], ['be', 'O'], ['extremely', 'O'], ['careful', 'O'], ['of', 'O'], ['is', 'O'], ['how', 'O'], ['other', 'O'], ['countries', 'O'], ['are', 'O'], ['going', 'O'], ['to', 'O'], ['take', 'O'], ['Germany', 'I-LOC'], ["'s", 'O'], ['lead', 'O'], [',', 'O'], ['"', 'O'], ['Welsh', 'I-ORG'], ['National', 'I-ORG'], ['Farmers', 'I-ORG'], ["'", 'I-ORG'], ['Union', 'I-ORG'], ['(', 'O'], ['NFU', 'I-ORG'], [')', 'O'], ['chairman', 'O'], ['John', 'I-PER'], ['Lloyd', 'I-PER'], ['Jones', 'I-PER'], ['said', 'O'], ['on', 'O'], ['BBC', 'I-ORG'], ['radio', 'I-ORG'], ['.', 'O']]
```

Named Entities:

```
['Germany', "Welsh National Farmers' Union", 'NFU', 'John Lloyd Jones', 'BBC radio']
```

Part 1.3a

In order to reduce out of vocabulary (OOV) words, either in the training, validation or testing sets, the following preprocessing techniques were performed:

1. Token Removal: Punctuation (, !, ., ?) special characters e.g. (#, *) and pure numbers (1,2,3) are removed to simplify the text and reduce OOV size

```python
# function to remove words that are puncutation, numbers, or special characters
def remove_punc_num_special(s):
    clean_set = []
    for sentence in s:
        clean_sentence = []
        for word, tag in sentence:
            if any(c.isalpha() for c in word):
                clean_sentence.append([word, tag])
        clean_set.append(clean_sentence)
    return clean_set
```

2. Hyphen Separation: In many OOV words, a hyphen joins constituent words which may or may not be present in the pre trained word embedding dictionary and are processed accordingly.

```
def hyphen_seperation(s,w2v):
    seperated_sentences=[]
    for sentence in s:
        seperated_sentence=[]
        for word,tag in sentence:
            array_of_words = word.split('-')
            for individual_word in array_of_words:
                seperated_sentence.append([individual_word,tag])
        seperated_sentences.append(seperated_sentence)
    return seperated_sentences
```

3. Word stemming: If a base form is present in the pretrained dictionary, the word embedding is associated with all derived and inflected forms to reduce the dimensionality within word embeddings.

```
def word_stemming(s,w2v):
    ps = PorterStemmer()
    stemmed_sentences=[]
    for sentence in s:
        stemmed_sentence=[]
        for word, tag in sentence:
            if (word not in w2v.key_to_index and  ps.stem(word) in w2v.key_to_index):
                stemmed_sentence.append([ps.stem(word),tag])
            else:
                stemmed_sentence.append([word,tag])
        stemmed_sentences.append(stemmed_sentence)
    return stemmed_sentences
```

4. Stop word removal: Words like “and”, “of”, “the”, “to”, do not carry significant semantic meaning and can introduce noise into the embeddings for NER tasks and are removed.

```
def remove_stop_words(s):
    stop_words=['of','a','and','to'] #'s
    stopless_sentences=[]
    for sentence in s:
        stopless_sentence=[]
        for word,tag in sentence:
            if word not in stop_words:
                stopless_sentence.append([word,tag])

        stopless_sentences.append(stopless_sentence)
    return stopless_sentences
```

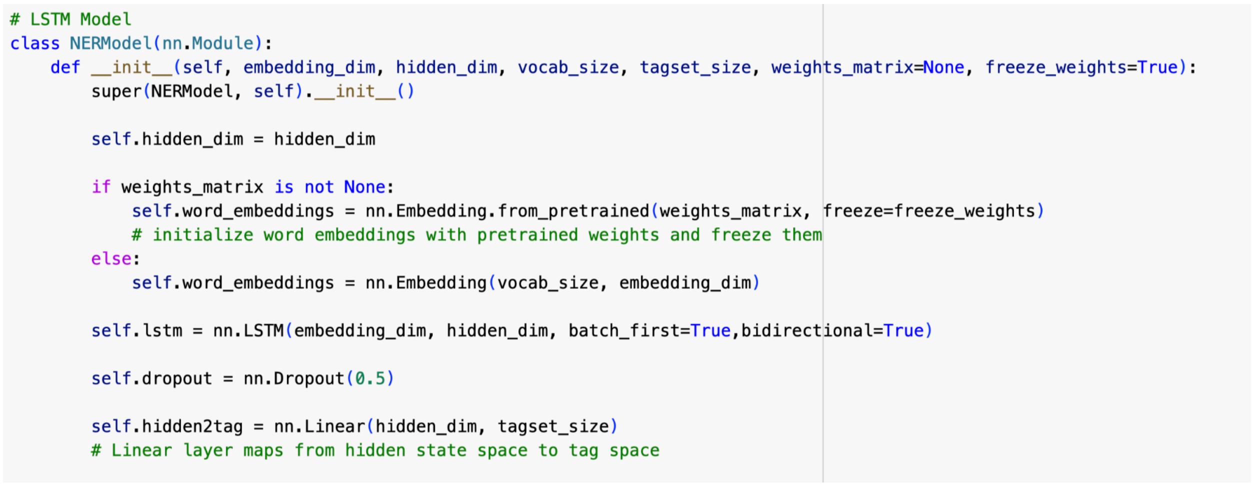
5. Case Folding: Any out-of-vocabulary (OOV) word whose lowercase equivalent is found in the pretrained dictionary is substituted with the lowercase version from the dictionary.

Any remaining OOV words are assigned the zeros vector

```
# append <UNK> token to words not in word2vec model
w2v['<UNK>'] = np.zeros(300)
```

Part 1.3b

LSTM Network:



```
# LSTM Model
class NERModel(nn.Module):
    def __init__(self, embedding_dim, hidden_dim, vocab_size, tagset_size, weights_matrix=None, freeze_weights=True):
        super(NERModel, self).__init__()

        self.hidden_dim = hidden_dim

        if weights_matrix is not None:
            self.word_embeddings = nn.Embedding.from_pretrained(weights_matrix, freeze=freeze_weights)
            # initialize word embeddings with pretrained weights and freeze them
        else:
            self.word_embeddings = nn.Embedding(vocab_size, embedding_dim)

        self.lstm = nn.LSTM(embedding_dim, hidden_dim, batch_first=True,bidirectional=True)

        self.dropout = nn.Dropout(0.5)

        self.hidden2tag = nn.Linear(hidden_dim, tagset_size)
        # Linear layer maps from hidden state space to tag space
```

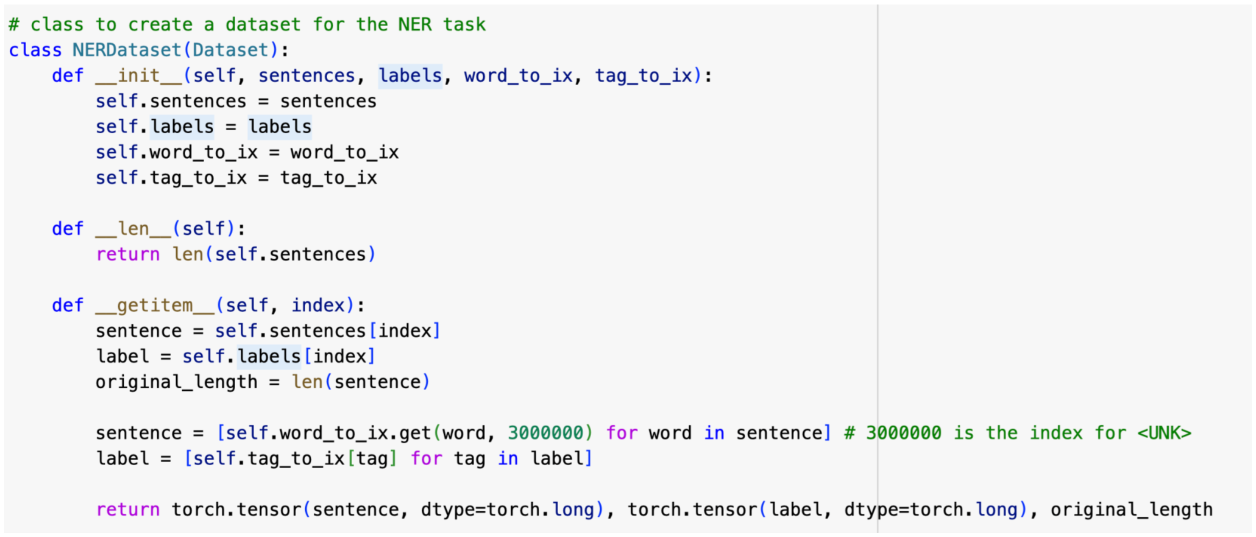
We use an LSTM network to produce the final vector representation of each word. When initialising the model, we specify the `weights_matrix` as the `w2v` pre-trained vectors. The weights are frozen. This is our embedding layer. Next, we define an LSTM layer with input size of the `embedding_dim` which is the size of the word vectors in `w2v`. It has an output size of `hidden_dim` which is the size of the LSTM hidden layer. `batch_first` is set to `True` for mini-batch training. Finally, we have a linear layer that maps from the LSTM's hidden layer to the tag space with a size equal to the `tagset_size`, i.e. the number of Named Entity labels.

```python
def forward(self, sentences):
    '''
    sentences: batch_size x max_seq_length
    tag_space: batch_size x tagset_size x max_seq_length
    '''
    embeds = self.word_embeddings(sentences) # Embed the input sentence

    # LSTM input shape: batch_size x max_seq_length x embedding_dim
    lstm_out, _ = self.lstm(embeds)

    # LSTM output shape: batch_size x max_seq_length x hidden_dim
    # reshape it for the linear layer
    lstm_out = lstm_out.contiguous().view(-1, self.hidden_dim)
    # shape: (batch_size * max_seq_length) x hidden_dim

    lstm_out=self.dropout(lstm_out)

    tag_space = self.hidden2tag(lstm_out)

    # reshape back to batch_size x max_seq_length x tagset_size
    tag_space = tag_space.contiguous().view(sentences.shape[0], sentences.shape[1], -1)

    # swap dimensions to make it batch_size x tagset_size x max_seq_length
    tag_space = tag_space.permute(0, 2, 1)

    return tag_space # Return output
```

For the forward pass, the model takes in a batch of sentences as input. We first get the `embeds` by passing it through the embedding layer. Next, we process the embeddings through the LSTM layer to get the output, `lstm_out`, with size `batch_size x max_seq_length x hidden_dim`.

We then pass this output through the linear layer with 0.5 dropout to get the score for each tag, `tag_space`. This is reshaped back to a 3D tensor with dimensions `batch_size x max_seq_length x tagset_size` which we permute to arrange it into `batch_size x tagset_size x max_seq_length` which is required for computing the loss function later on. This is then returned by the function.

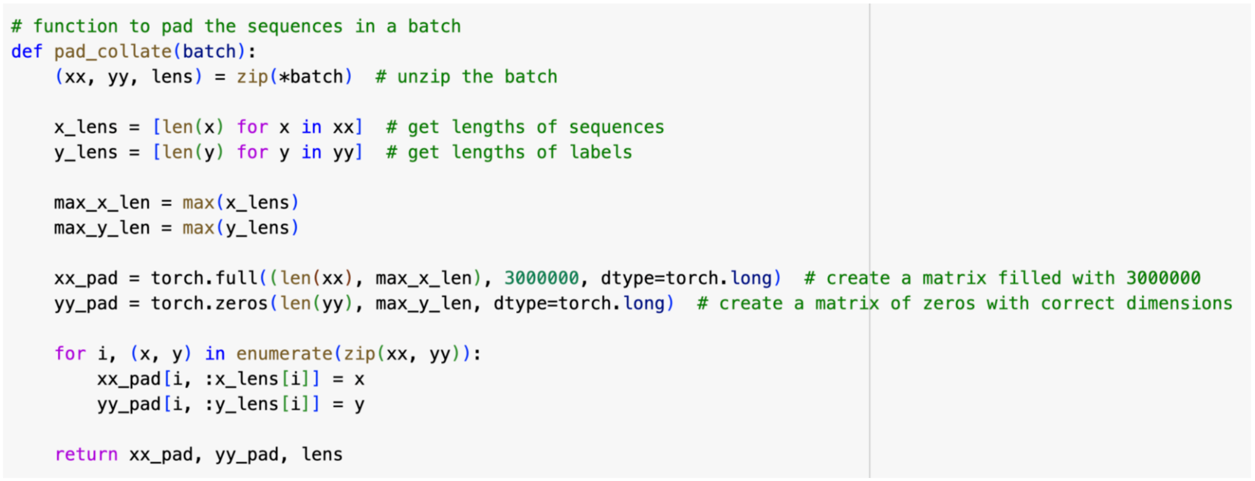
Dataset:



```python
# class to create a dataset for the NER task
class NERDataset(Dataset):
    def __init__(self, sentences, labels, word_to_ix, tag_to_ix):
        self.sentences = sentences
        self.labels = labels
        self.word_to_ix = word_to_ix
        self.tag_to_ix = tag_to_ix

    def __len__(self):
        return len(self.sentences)

    def __getitem__(self, index):
        sentence = self.sentences[index]
        label = self.labels[index]
        original_length = len(sentence)

        sentence = [self.word_to_ix.get(word, 3000000) for word in sentence] # 3000000 is the index for <UNK>
        label = [self.tag_to_ix[tag] for tag in label]

        return torch.tensor(sentence, dtype=torch.long), torch.tensor(label, dtype=torch.long), original_length
```

Here we create a custom dataset for our task. For initialisation, it takes an array of `sentences` and `labels`. `sentences` consist of arrays of sentences which consist of words. Similarly, `labels` consist of arrays of labels for each sentence, which consist of the individual tags for the words.

Following which, we have `word_to_ix` and `tag_to_ix` dictionaries. `word_to_ix` maps each word in the dataset the index to its corresponding word vector in the embedding matrix. Likewise, `tag_to_ix` maps each tag to its corresponding label index.



```python
# function to pad the sequences in a batch
def pad_collate(batch):
    (xx, yy, lens) = zip(*batch)  # unzip the batch

    x_lens = [len(x) for x in xx]  # get lengths of sequences
    y_lens = [len(y) for y in yy]  # get lengths of labels

    max_x_len = max(x_lens)
    max_y_len = max(y_lens)

    xx_pad = torch.full((len(xx), max_x_len), 3000000, dtype=torch.long)  # create a matrix filled with 3000000
    yy_pad = torch.zeros(len(yy), max_y_len, dtype=torch.long)  # create a matrix of zeros with correct dimensions

    for i, (x, y) in enumerate(zip(xx, yy)):
        xx_pad[i, :x_lens[i]] = x
        yy_pad[i, :y_lens[i]] = y

    return xx_pad, yy_pad, lens
```

This function pads all the sentences in a batch to be the same length as that of the longest sentence in that batch. Here, the paddings are assigned to the index of 300000 in the `word2vec` embeddings, which we defined to be the zero vector.

Moreover, the paddings are assigned the label index 0, which we defined as the index for the paddings. This is important as this index will be used later in the loss function to make it ignore the paddings during the training loop.

Note: This function is used as the `` `collate_fn` `` in the loading of the custom dataset in the torch dataloader.

Parameters:

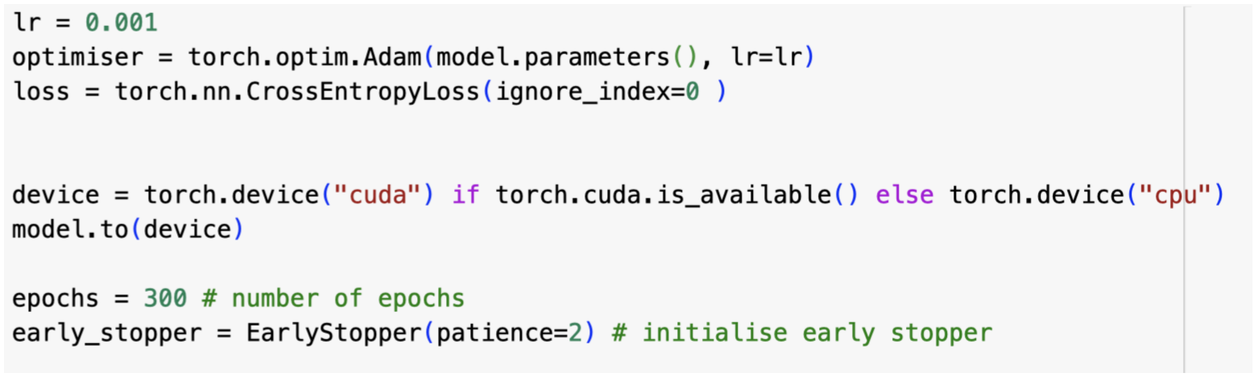
- Word input: Each word goes through the embedding layer with final embedding size as `` `embedding_dim` `` → 300
- Hidden Layer: The hidden layer has size `` `hidden_dim` `` → 1024
- Linear Layer: The linear layer has size `` `tagset_size` `` → 9
  - Thus, the length of the final vector representation is 9, a score for each label

Learnable Parameters:

- $h(t) = \phi(U^T x(t) + W^T h_i(t-1) + b)$
  - This is the output of the hidden layer at time step t
  - ϕ is the tanh hidden-layer activation function
  - $x(t)$ is the word embeddings at time step t for the entire batch, with size `` `batch_size x embedding_dim` `` → 16 x 300
  - U, W and b are the learnable parameters
    - U is the weights for the inputs, with size `` `batch_size x hidden_dim` `` → 16 x 1024
    - W is the recurrent weight matrix connecting previous hidden-layer output to hidden input, with size `` `hidden_dim x hidden_dim` `` → 1024 x 1024
    - b is the bias connected to hidden layer, with size `` `hidden_dim` `` → 1024

- $y(t) = V^T H(t) + c$
    - This is the output of the linear layer after the hidden layer, at time step t
    - V and c are the learnable parameters
        - V is the weight vector of the output layer, with size `tagset_size x hidden_dim` → 9 x 1024
        - c is the bias connected to the output layer, with size `tagset_size` → 9

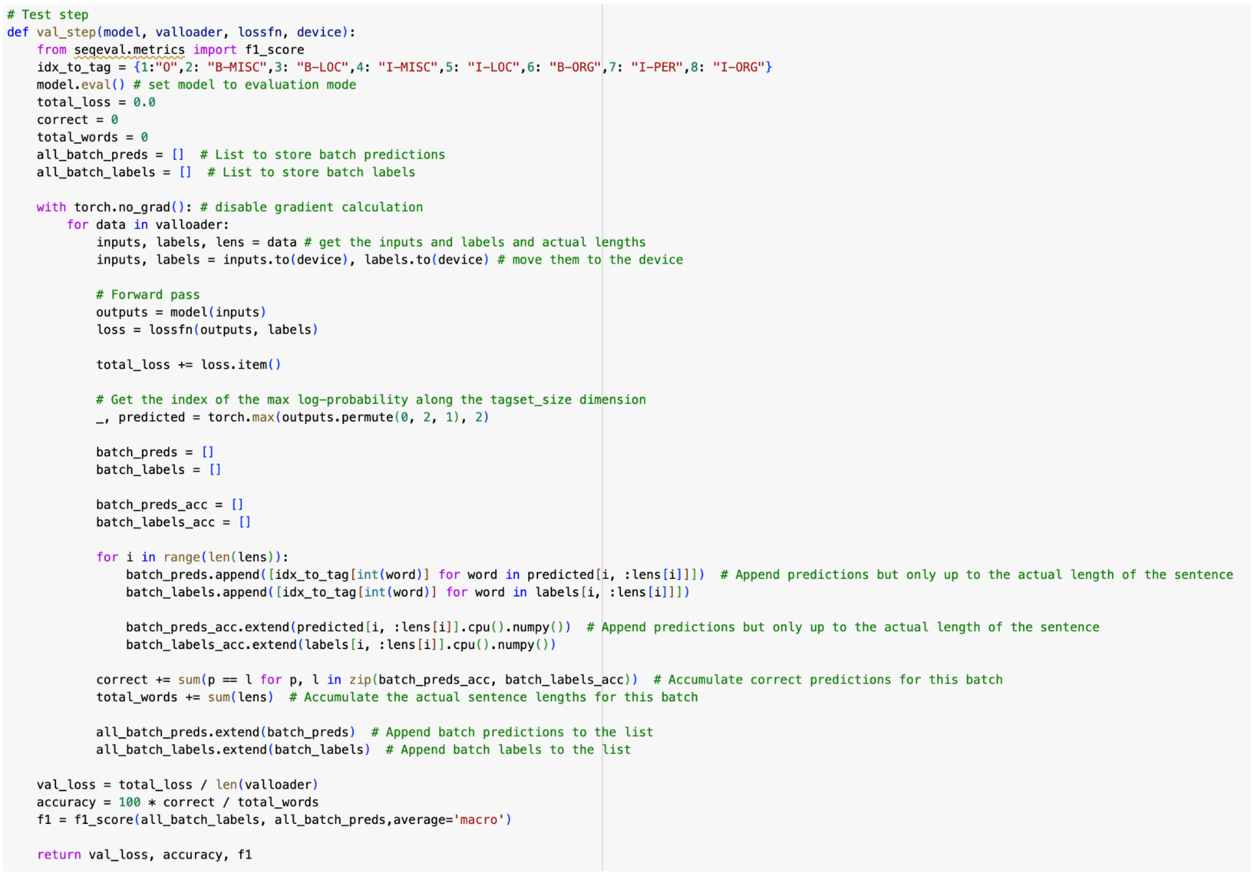
Training Parameters:



```
lr = 0.001
optimiser = torch.optim.Adam(model.parameters(), lr=lr)
loss = torch.nn.CrossEntropyLoss(ignore_index=0 )


device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
model.to(device)

epochs = 300 # number of epochs
early_stopper = EarlyStopper(patience=2) # initialise early stopper
```

We define the optimiser to be the Adam optimiser with an initial learning rate of 0.001. The loss function is the CrossEntropyLoss, and we have implemented an `early_stopper` function with a patience of 2 to automatically terminate the training loop if the validation loss doesn't improve in 2 consecutive epochs.

Handling `<PAD>`:



```
# Test step
def val_step(model, valloader, lossfn, device):
    from seqeval.metrics import f1_score
    idx_to_tag = {1:"O",2: "B-MISC",3: "B-LOC",4: "I-MISC",5: "I-LOC",6: "B-ORG",7: "I-PER",8: "I-ORG"}
    model.eval() # set model to evaluation mode
    total_loss = 0.0
    correct = 0
    total_words = 0
    all_batch_preds = []  # List to store batch predictions
    all_batch_labels = []  # List to store batch labels

    with torch.no_grad(): # disable gradient calculation
        for data in valloader:
            inputs, labels, lens = data # get the inputs and labels and actual lengths
            inputs, labels = inputs.to(device), labels.to(device) # move them to the device

            # Forward pass
            outputs = model(inputs)
            loss = lossfn(outputs, labels)

            total_loss += loss.item()

            # Get the index of the max log-probability along the tagset_size dimension
            _, predicted = torch.max(outputs.permute(0, 2, 1), 2)

            batch_preds = []
            batch_labels = []

            batch_preds_acc = []
            batch_labels_acc = []

            for i in range(len(lens)):
                batch_preds.append([idx_to_tag[int(word)] for word in predicted[i, :lens[i]]])  # Append predictions but only up to the actual length of the sentence
                batch_labels.append([idx_to_tag[int(word)] for word in labels[i, :lens[i]]])

                batch_preds_acc.extend(predicted[i, :lens[i]].cpu().numpy())  # Append predictions but only up to the actual length of the sentence
                batch_labels_acc.extend(labels[i, :lens[i]].cpu().numpy())

            correct += sum(p == l for p, l in zip(batch_preds_acc, batch_labels_acc))  # Accumulate correct predictions for this batch
            total_words += sum(lens)  # Accumulate the actual sentence lengths for this batch

            all_batch_preds.extend(batch_preds)  # Append batch predictions to the list
            all_batch_labels.extend(batch_labels)  # Append batch labels to the list

    val_loss = total_loss / len(valloader)
    accuracy = 100 * correct / total_words
    f1 = f1_score(all_batch_labels, all_batch_preds,average='macro')

    return val_loss, accuracy, f1
```

We are padding every sentence in a batch to be the same length to that of the longest sentence in that batch. As such, we have to handle these in the training and validation loop. We handle these in two steps. First, we set `ignore_index=0` in the `CrossEntropyLoss()` function, so that tokens with value of 0 will be ignored in the calculation of the loss. That suffices for the training loop since the model does not learn to predict padding tokens, which do not contribute to the real data distribution.

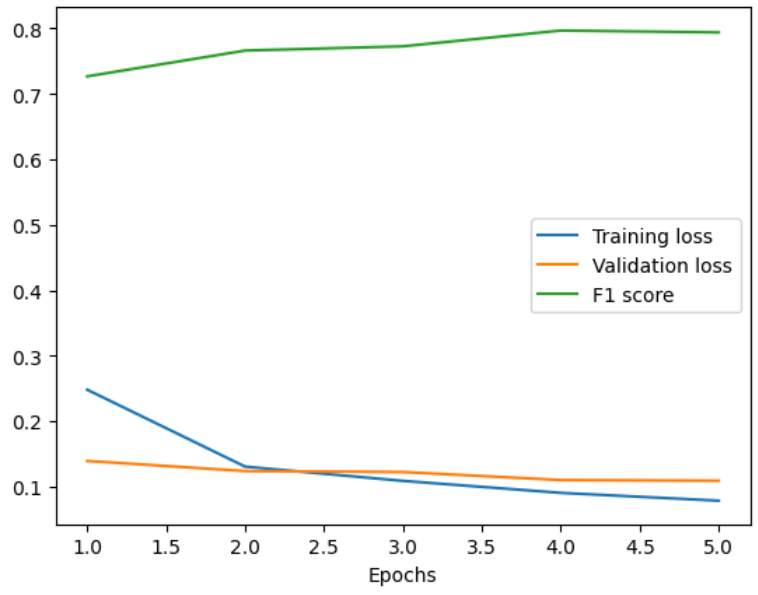
However, during validation, we still need to consider how these paddings affect our metrics. Thus, in the second step, we take into account only the predictions up to the actual length of the sentence, effectively masking out the padded tokens. Moreover `total_words` only accounts for the actual length of the sentence.

Part 1.3c / 1.3d

The model was trained for 5 epochs, as it appeared that the validation loss started to increase when the training loss continued to decrease. The runtimes, training loss, validation loss, F1-Score of each epoch are as follows:

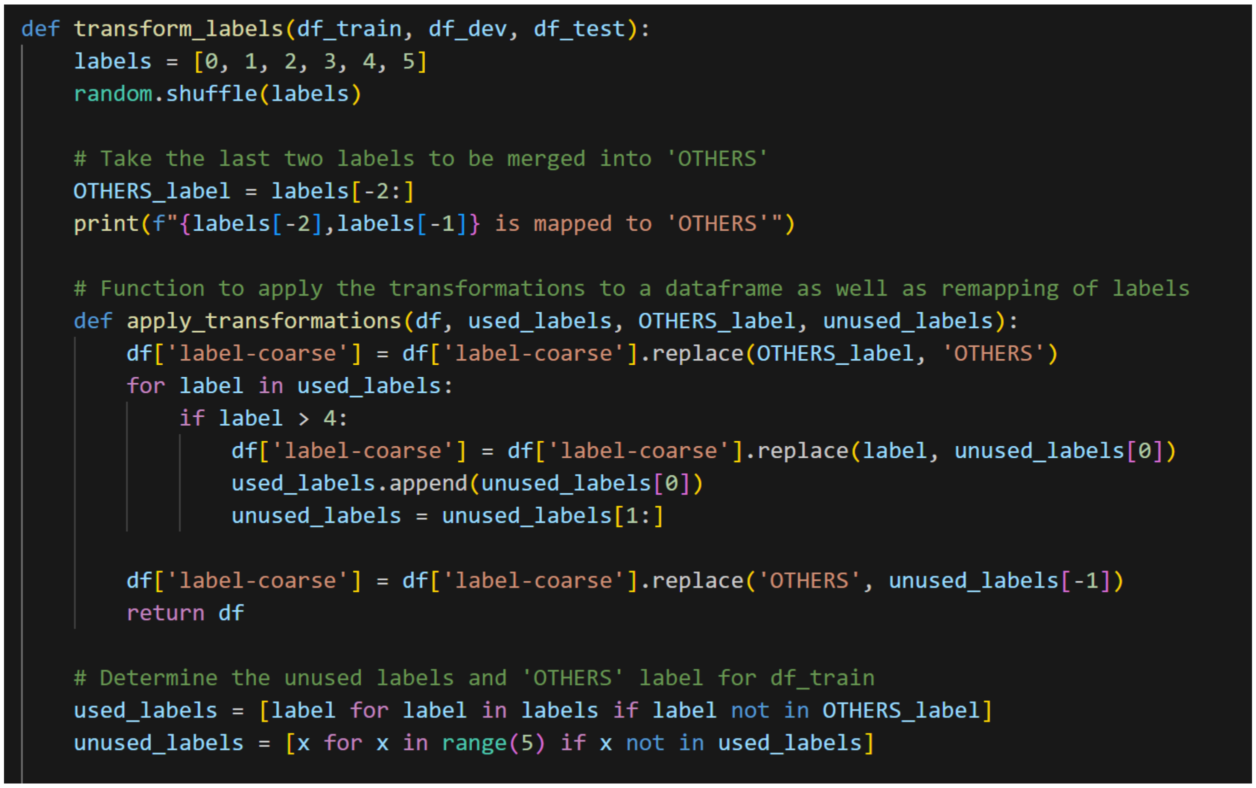
Epoch	Runtime (s)	Training Loss	Validation Loss	F1-Score
1	13.13	0.2480	0.1391	0.7267
2	12.99	0.1304	0.1237	0.7661
3	12.83	0.1087	0.1222	0.7726
4	12.98	0.0903	0.1100	0.7968
5	12.98	0.0784	0.1089	0.7939

F1 Score on Test Set: 0.73



Question 2

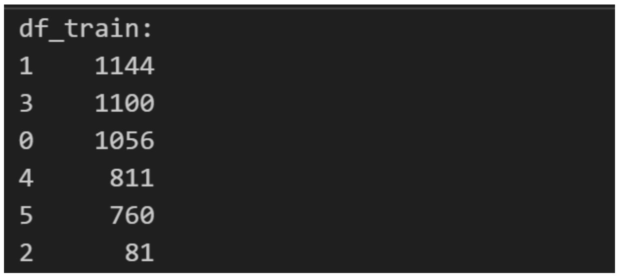
2a) To determine which of the 2 labels should be combined to form the ‘OTHERS’ label, we first put all the possible coarse labels into an array before shuffling them. From there, the last 2 labels will then be chosen to be placed into the ‘OTHERS’ label.



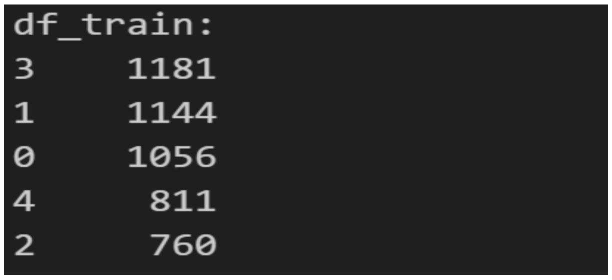
```python
def transform_labels(df_train, df_dev, df_test):
    labels = [0, 1, 2, 3, 4, 5]
    random.shuffle(labels)

    # Take the last two labels to be merged into 'OTHERS'
    OTHERS_label = labels[-2:]
    print(f"{labels[-2],labels[-1]} is mapped to 'OTHERS'")

    # Function to apply the transformations to a dataframe as well as remapping of labels
    def apply_transformations(df, used_labels, OTHERS_label, unused_labels):
        df['label-coarse'] = df['label-coarse'].replace(OTHERS_label, 'OTHERS')
        for label in used_labels:
            if label > 4:
                df['label-coarse'] = df['label-coarse'].replace(label, unused_labels[0])
                used_labels.append(unused_labels[0])
                unused_labels = unused_labels[1:]

        df['label-coarse'] = df['label-coarse'].replace('OTHERS', unused_labels[-1])
        return df

    # Determine the unused labels and 'OTHERS' label for df_train
    used_labels = [label for label in labels if label not in OTHERS_label]
    unused_labels = [x for x in range(5) if x not in used_labels]
```

After which, we will then remap the remaining 5 labels into integers from [0:4] to form the 5 classes.

Below is an example of the data output for the training set after undergoing transformation.



```
df_train:
1    1144
3    1100
0    1056
4     811
5     760
2      81
```



```
df_train:
3    1181
1    1144
0    1056
4     811
2     760
```

In this instance, the following mapping is observed in the following sequence:

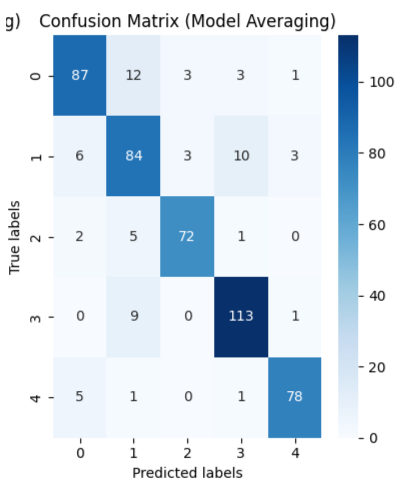
- Labels 2,3 were mapped to ‘OTHERS’
- Label 5 was mapped to 2
- Label ‘OTHERS’ was mapped to 3
- All other labels remained the same

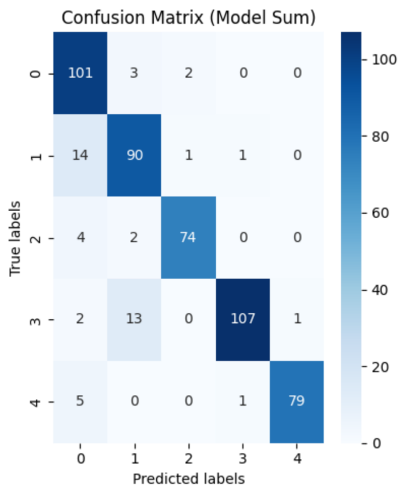
Initial Label	Final Label
1	1
2	3
3	3
4	4
5	2

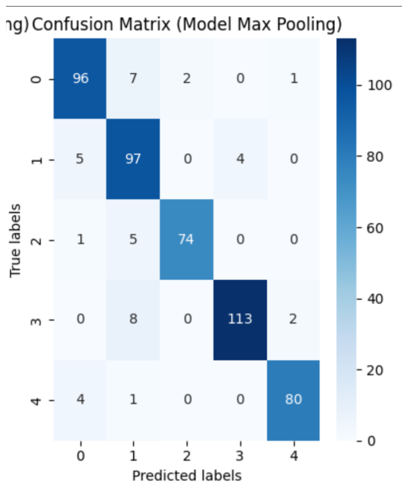
2b) In designing the neural network, several aggregation methods were used, namely averaging, sum and max pooling.

1. Summing as the name implies simply takes the sum of all word vectors within the sentence itself to obtain the final vector
2. Averaging takes an additional step compared to summing by accounting for the length of a sentence when calculating the final vector.
3. Max pooling is taking the maximum attribute from each word vector within the sentence in order to form the final vector.

Ultimately, we chose to use max pooling as it displayed the highest accuracy as shown in the table and confusion matrix below:







Aggregation Method	Accuracy (%)	F1 Score
Averaging	86.80	0.8680
Sum	90.60	0.9020
Max pooling	92.20	0.9200

2c) In our neural network, we mainly adopted a LSTM model. Similarly to part (I), The LSTM layer makes it more feasible for the Recurrent Neural Network to ensure that information is preserved over a longer time frame. While it does not guarantee that no information is lost, it simplifies the process in which the model can pick up on long-distance dependencies.

Parameters involved:

- Input: a 2D matrix with size (batch size) by **33** (maximum length of sentence)
- Output: a vector of length **5** (number of classes available) to be used in the softmax classifier to predict the final label for each question
- batch_size: size of the batch used for the mini-batch training **(64)**
- hidden_units: dimension of the cells in LSTM model **(64)**

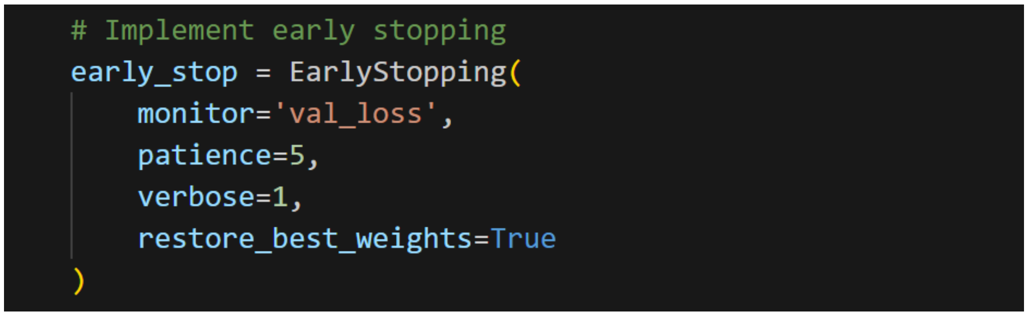
Within the LSTM model, there are 3 gates involved namely:

- Forget gate: controls what is forgotten
$$f^{(t)} = \sigma(W_f h^{(t-1)} + U_f x^{(t)} + b_f$$
- Input gate: controls what is written to the cell
$$i^{(t)} = \sigma(W_i h^{(t-1)} + U_i x^{(t)} + b_i$$
- Output gate: controls what is output to the hidden state
$$o^{(t)} = \sigma(W_o h^{(t-1)} + U_o x^{(t)} + b_o$$

These gates are then used to compute how much to forget, update and output:

- New cell content: the new information to be written to the cell
$$\bar{c}^{(t)} = tanh(W_c h^{(t-1)} + U_c x^{(t)} + b_c)$$
- Cell state: forget some content from the previous cell state and update the new cell state with new cell content
$$c^{(t)} = f^{(t)} \circ c^{(t-1)} + i^{(t)} \circ \bar{c}^{(t)}$$
- Hidden state: output content from the cell
$$h^{(t)} = o^{(t)} \circ tanh(c^{(t)})$$

2d/e) In training our model, we used up to a maximum of 50 epoch whilst implementing an early stop condition whereby the training will be forcefully stopped should there be no improvements in 5 epoch, after which it will restore the weights used 5 epochs before the current one.



```
# Implement early stopping
early_stop = EarlyStopping(
    monitor='val_loss',
    patience=5,
    verbose=1,
    restore_best_weights=True
)
```

The results of the training for each epoch are as follows:

Epoch	Runtime (s)	Loss	Accuracy	val_Loss	val_Accuracy
1	14.118	1.2259	0.4968	0.8117	0.6960
2	11.139	0.7402	0.7314	0.4987	0.8300
3	11.144	0.5621	0.8059	0.4319	0.8360
4	11.144	0.4862	0.8449	0.3962	0.8660
5	11.145	0.4201	0.8663	0.3380	0.8840
6	11.145	0.3714	0.8788	0.4348	0.8620
7	11.145	0.3529	0.8901	0.3522	0.8920
8	11.145	0.3096	0.9000	0.3491	0.8880
9	11.145	0.2853	0.8974	0.3521	0.8920
10	11.145	0.2643	0.9160	0.3353	0.9060
11	11.144	0.2376	0.9243	0.3155	0.9060
12	11.142	0.2497	0.9229	0.3328	0.9040
13	11.144	0.2281	0.9289	0.3240	0.9120
14	11.143	0.2309	0.9277	0.3034	0.9220
15	11.145	0.2009	0.9362	0.2985	0.9200