NANYANG TECHNOLOGICAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Assignment for SC4002 / CE4045 / CZ4045

AY 2023 - 2024

Group ID: G7

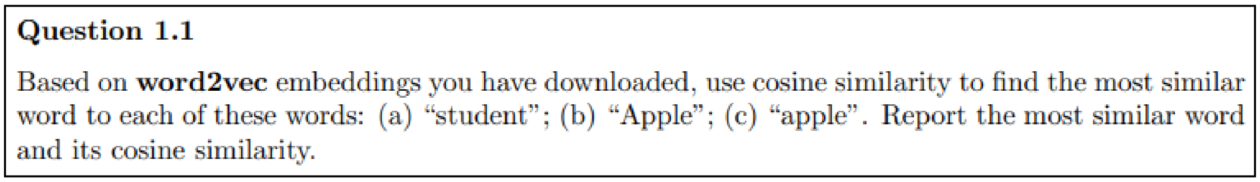
No.	Name	Matriculation Number	Contributions
1	Darren Choo Jian Hao		
2	Aloysius Ang		
3	Wang Shiqiang		
4	Chen Zihang		
5	Gan Hao Yi		

Table of contents

Question 1 - Sequence Tagging: NER.......... 3
(1.1) Word Embedding:.......... 3
(1.2a) Data Preprocessing - part 1:.......... 4
(1.2b) Data Preprocessing - part 2:.......... 5
(1.3a) Dealing With New Words in Train, Development & Test Set:.......... 7
(1.3b) Neural Network Description:.......... 9
(1.3c) Epochs & Run-time:.......... 14
(1.3d) F1-score on Development & Test Set:.......... 15
Question 1 - Conclusion:.......... 16
Question 2 - Sentence-Level Categorization: Question Classification:.......... 17
(a) 5 Final Classes:.......... 17
(b) Aggregation Method:.......... 18
(c) Neural Network Description:.......... 21
(d) Epochs & Run-time:.......... 27
(e) Accuracy on Development & Test Set:.......... 28
Question 2 - Conclusion:.......... 29

Question 1 - Sequence Tagging: NER

(1.1) Word Embedding:



Question 1.1

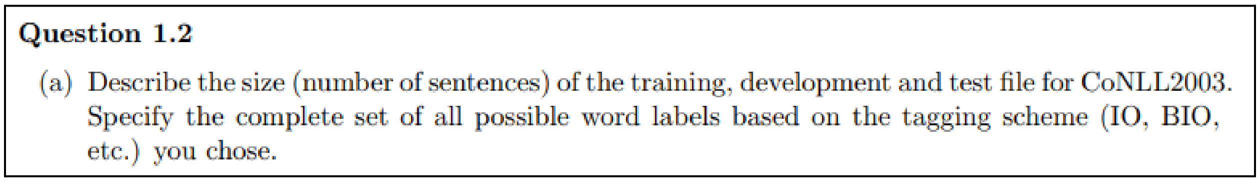
Based on **word2vec** embeddings you have downloaded, use cosine similarity to find the most similar word to each of these words: (a) “student”; (b) “Apple”; (c) “apple”. Report the most similar word and its cosine similarity.

Using Cosine Similarity to find:

Word	**Most similar**	**Cosine Similarity**
“student”	“students”	0.7294867038726807
“Apple”	“Apple_AAPL”	0.7456986308097839
“apple”	“apples”	0.720359742641449

```
The most similar word to "student" is: "students", with a cosine similarity of:
0.7294867038726807
The most similar word to "Apple" is: "Apple_AAPL", with a cosine similarity of:
0.7456986308097839
The most similar word to "apple" is: "apples", with a cosine similarity of:
0.720359742641449
```

(1.2a) Data Preprocessing - part 1:


Question 1.2

(a) Describe the size (number of sentences) of the training, development and test file for CoNLL2003. Specify the complete set of all possible word labels based on the tagging scheme (IO, BIO, etc.) you chose.

Number of sentences:	
Train Dataset	14986
Development Dataset	3465
Test Dataset	3683

```
Number of sentences in the training set: 14986
Number of sentences in the development set: 3465
Number of sentences in the test set: 3683
```

The complete set of all possible word labels based on the tagging scheme IOB1 (taken from SEQEVAL default scheme):

- 'B-MISC',
- 'I-MISC',
- 'I-PER',
- 'I-LOC',
- 'B-ORG',
- 'B-LOC',
- 'O',
- 'I-ORG'

```
Unique labels in train: {'B-MISC', 'I-MISC', 'I-PER', 'I-LOC', 'B-ORG', 'B-LOC', 'O',
'I-ORG'}
Unique labels in val: {'B-MISC', 'I-MISC', 'I-PER', 'I-LOC', 'O', 'I-ORG'}
Unique labels in test: {'B-MISC', 'I-MISC', 'I-PER', 'I-LOC', 'B-ORG', 'B-LOC', 'O',
'I-ORG'}
Unique labels overall: {'B-MISC', 'I-MISC', 'I-PER', 'I-LOC', 'B-ORG', 'B-LOC', 'O',
'I-ORG'}
```

(1.2b) Data Preprocessing - part 2:

> (b) Choose an example sentence from the training set of CoNLL2003 that has at least two named entities with more than one word. Explain how to form complete named entities from the label for each word, and list all the named entities in this sentence.

Choose example sentence from training set that has at least 2 named entities with more than 1 word:

- Explanation:
  - How to form complete named entities from the label for each word
  - List all named entities in this sentence

<u>Sentence 1:</u>
"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."

Word	**Named Entity**	**Explanation**
European Union	I-ORG I-ORG	“European”, “Union” are both labeled as I-ORG, so they form 1 named entity.
Werner Zwingmann	I-PER I-PER	“Werner”, “Zwingmann” are both labeled as I-PER, so they form 1 named entity.

There is a "Germany" and "Britain" in the sentence but since it is only 1 word, it was not flagged by the function `count_entities()` in SC4002_G7_Q1_FNN.ipynb

Here are the named entities in example sentence 1:

- Germany (LOC)
- European Union (ORG)
- Werner Zwingmann (PER)
- Britain (LOC)

Sentence 2:
“Tension has mounted since Israeli Prime Minister Benjamin Netanyahu took office in June vowing to retain the Golan Heights Israel captures from Syria in the 1967 Middle East war.”

Word	**Named Entity**	**Explanation**
Benjamin Netanyahu	I-PER I-PER	The successive "I-PER" tags tell us to concatenate these words into a single named entity of type "PER".
Golan Heights	I-LOC I-LOC	Like "Benjamin Netanyahu", the successive "I-LOC" tags mean we should concatenate these into a single named entity of type "LOC".
Middle East	I-LOC I-LOC	The successive "I-LOC" tags mean these words form a single named entity of type "LOC".

- Israeli is labeled as "I-MISC".
    - Since it's not preceded by another "I-MISC" or "B-MISC", it forms a single-word named entity of type "MISC".

- Israel is labeled as "B-LOC", indicating the start of a new named entity of type "LOC".
    - Since it's not followed by an "I-LOC", it's a single-word named entity.

- Syria is labeled as "I-LOC".
    - It's not preceded by another "I-LOC" or "B-LOC", so it's a single-word named entity of type "LOC".

Here are the named entities in example sentence 2:
- Israeli (MISC)
- Benjamin Netanyahu (PER)
- Golan Heights (LOC)
- Israel (LOC)
- Syria (LOC)
- Middle East (LOC)

(1.3a) Dealing With New Words in Train, Development & Test Set:

(a) Discuss how you deal with new words in the training set which are not found in the pretrained dictionary. Likewise, how do you deal with new words in the test set which are not found in either the pretrained dictionary or the training set? Show the corresponding code snippet.

We used 2 different methods of dealing with new words with respect to different model implementations. Both methods convert the words to a zero vector. We have 2 models, a regular FeedForward Network (FFN) model that performs word level classification, and a bi-directional Long Short Term Memory (LSTM).

FeedForward Network (FFN):

For words that are not found in word2vec, we appended a zero vector with the shape (300,) to represent the word. First, we check if word2vec[word] exists. If it does, we store its vector. If not, we append the zero vector to replace it.

```
for word in range(len(words)):
    if words[word] == ' ':
        # print(f"-{words[word]}-")
        words[word] = ' " '
        # print(f"-{words[word]}-")
        y[word] = "O"
    try:
        X.append(w2v[words[word]])
    except Exception as e:
        # print(f"Error: {e}")
        X.append(np.zeros((300,), dtype=float))
```

Bi-directional Long Short Term Memory (Bi-LSTM):

For words that are not in the pre-trained dictionary, they are represented by the Out-Of-Vocabulary (OOV) token which are converted to zero vectors.

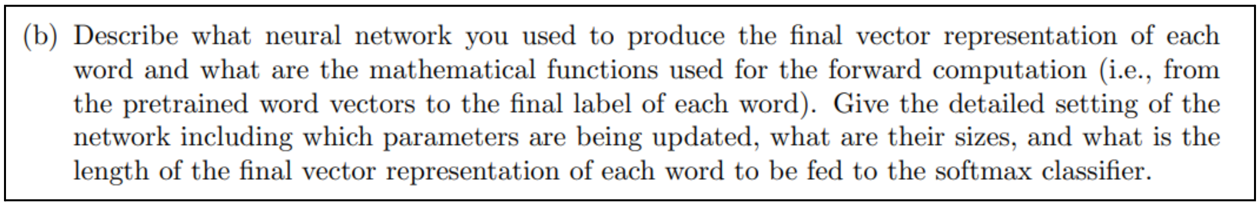
```
word_tokenizer = Tokenizer(filters=[], lower=False, oov_token='OOV'

#create embedding matrix with w2v
vocab_size = len(word_tokenizer.word_index) + 1
embedding_dim = 300
embedding_matrix = np.zeros((vocab_size, embedding_dim))

for word, idx in word_tokenizer.word_index.items():
    try:
        embedding_vector = w2v[word]
        embedding_matrix[idx] = embedding_vector
    except KeyError:
        continue
oov_idx = word_tokenizer.word_index.get('OOV', 0)
if oov_idx != 0:
    embedding_matrix[oov_idx] = np.zeros(embedding_dim)

tag_tokenizer.index_word[0] = 'PAD'
tag_tokenizer.index_word
```

(1.3b) Neural Network Description:



(b) Describe what neural network you used to produce the final vector representation of each word and what are the mathematical functions used for the forward computation (i.e., from the pretrained word vectors to the final label of each word). Give the detailed setting of the network including which parameters are being updated, what are their sizes, and what is the length of the final vector representation of each word to be fed to the softmax classifier.

<u>FeedForward Network (FFN):</u>
The neural network that we used to produce the final vector representation is a regular FFN.
The **architecture** is as follows:

- **Input layer**: 300 input features.
  - This is because the word vector retrieved from w2v gives us a (300,) vector.

- **4 Hidden layers**: 32 hidden neurons.
  - This value is retrieved after doing a grid search to find the best hyperparameters. Most of the values in the gridsearch have been commented out as it takes too long to run and the removed values all gave a f1_score of 0.
  - 4 hidden layers were used as it provided a slightly better f1 score with the validation dataset than 3 hidden layers.
  - A ReLU activation function was used after each hidden layer to reduce gradient vanishing.

- **Output layer**: Number of classes given by the training dataset which is 8.
  - From the previous layer, we pass the values from 32 hidden neurons to 8 neurons at the output layer.
  - Did not use the overall number of classes as the model should be trained on the training dataset. Should not include any classes (if any) that are from the development / test dataset.
  - We used a softmax function to classify the final word classes.

- **Optimizer**: We used Adam Optimizer.
- **Loss function**: We used the Cross Entropy Loss function.

Mathematical functions:

- **Hidden layers:**
  - ReLU activation function.

- **Output layer:**
  - Cross Entropy Loss function.
  - Softmax activation function.

- Seqeval f1 score calculation.

```
class MLP(nn.Module):
    def __init__(self, no_features, no_hidden, no_labels):
        super().__init__()
        self.mlp_stack = nn.Sequential(
            nn.Linear(no_features, no_hidden),
            nn.ReLU(),

            nn.Linear(no_hidden, no_hidden),
            nn.ReLU(),

            nn.Linear(no_hidden, no_hidden),
            nn.ReLU(),

            nn.Linear(no_hidden, no_hidden),
            nn.ReLU(),

            nn.Linear(no_hidden, no_labels),
            nn.Softmax(dim=1),
        )

    def forward(self, x):
      """
      Added a forward(x) function to return logits.
      """
      logits = self.mlp_stack(x)
      return logits
```

```
def train(no_features, no_hidden, no_classes, lr, loss_fn, train_dataloader, dev_dataloader,
sentence_length):
    model = MLP(no_features, no_hidden, no_classes)
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    early_stopper = EarlyStopper(patience=2, min_delta=0.005)
```

```
loss_fn = nn.CrossEntropyLoss()
```

Network settings (FFN):
Input layer:
- Parameters being updated: None.
- Parameter Size: 300 neurons for each value in the input vector of shape (300,).

Hidden layer 1:
- Parameters being updated: Weight matrix of input 1.
- Parameter Size: weight matrix of size (300 neurons (input layer) + 1 (bias)) x 32 neurons (hidden layer 1) = **9632**.

Hidden layers 2-4:
- Parameters being updated: Weight matrix of hidden layers 2, 3, 4.
- Parameter Size: weight matrix of size (32 neurons (previous hidden layer) + 1 (bias)) x 32 neurons (current hidden layer) * 3 (hidden layers 2, 3, 4) = **3168**.

Output layer (**w** Softmax Classifier):
- Parameters being updated: Weight matrix of output layer.
- Parameter Size: weight matrix of size (32 neurons (last hidden layer) + 1 (bias)) x 8 neurons (8 classes) = **264**.

Additional Information:
- Optimal Batch size (After gridsearch): 512.
- Learning rate: 0.001.
- Number of hidden neurons per layer: 32.

Final vector Representation:
- Final layer vector representation of each word: vector of length 8, each representing the raw scores before being passed into the softmax classifier to normalize the output to a probability distribution of a range from [0, 1].

Bi-directional Long Short Term Memory (Bi-LSTM):

The architecture used for generating the final vector representation of each word is a Sequential model comprised of the following layers:

- **Embedding Layer**:
  - This layer converts token indices to vectors of fixed size, 300 in this case.
  - The weight of this layer is initialized with pre-trained vectors and is kept frozen during training (trainable=False).

- **Bidirectional LSTM Layer**:
  - This layer has 100 LSTM units and returns sequences, effectively doubling the output dimensions due to its bidirectional nature (100 units in each direction).

- **TimeDistributed Dense Layer**:
  - This layer has a number of output units equal to len(tag_tokenizer.word_index) + 1, 9 in this case, each applying a softmax activation function.

Mathematical functions:

- **Embedding Layer**:
  - Vector lookup.

- **Bidirectional LSTM Layer**:
  - LSTM cell operations (sigmoid and tanh activations).

- **Time-Distributed Dense Layer**:
  - Linear transformation followed by a softmax activation.

```
model = Sequential()

model.add(Embedding(input_dim=vocab_size, output_dim=300, weights=[embedding_matrix],
trainable=False, mask_zero=True)) #trainable set to false to keep matrix frozen

model.add(Bidirectional(LSTM(units=100, return_sequences=True)))

model.add(TimeDistributed(Dense(len(tag_tokenizer.word_index) + 1, activation='softmax')))

custom_adam = Adam(learning_rate=0.01)

model.compile(optimizer=custom_adam, loss='sparse_categorical_crossentropy',
metrics=['accuracy'])

reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=3, min_lr=0.001)

f1_callback = LambdaCallback(on_epoch_end=on_epoch_end)

history = model.fit(X_train, Y_train.reshape(*Y_train.shape, 1),
                    batch_size=16, epochs=50,
                    validation_data=(X_val, Y_val.reshape(*Y_val.shape, 1)),
                    callbacks=[reduce_lr, f1_callback])
```

Network settings (Bi-directional LSTM):
Embedding layer:

- Parameters being updated: None (**Non-trainable**).
- Parameter Size: vocabulary size (23626) x embedding dimension (300) = **7,087,800**.

Bidirectional LSTM Layer:

- Parameters being updated: Weights and biases in the forward and backward LSTM units.
- Parameter Size: 4 * ((size_of_input + 1) * size_of_output + size_of_output^2), which in this case would be 4 * ((300 + 1) * 100 + 100^2) = 4 * (30100 + 10000) = 4 * 40100 = 160400. Because it's bidirectional, this number is doubled: 160,400 * 2 = **320,800**.

Time-Distributed dense layer (**w** softmax classifier):

- Parameters being updated: Weights and biases in the dense layer.
- Parameter Size: output dimension of the previous layer (200) times the output dimension of this layer (9) plus the bias terms for each output node (9). That is, (200 * 9) + 9 = **1809**.

Additional Information:

- Batch size: 16.
- Optimizer: Adam with a custom learning rate of 0.01.
- Loss function: Sparse categorical cross-entropy.
- Other callbacks: Reduce learning rate by factor of 0.2 on plateau with patience of 3, custom F1 score calculation for early stopping with patience of 5.

Final vector Representation:

- Length of final vector: 9, corresponding to the 8 unique NER tags and the 'PAD' tag.
- Each element of this vector will represent the probability that the input word belongs to one of these 9 classes.

(1.3c) Epochs & Run-time:

> (c) Report how many epochs you used for training, as well as the running time.

<u>FeedForward Network (FFN):</u>

- The number of epochs used was 50. May vary due to early stop being implemented.
- The total time taken for training was 269.5758 seconds.

```
Number of epochs used: 50/50
Time taken for training : 269.57580614089966
```

<u>Bi-directional Long Short Term Memory (LSTM):</u>

- The number of epochs used was 50, but as early-stop was implemented, the training stopped at epoch 11.
    - If the f1_score does not improve after 5 consecutive epochs, the training will stop.
- The total time taken for training was 152.8240 seconds.

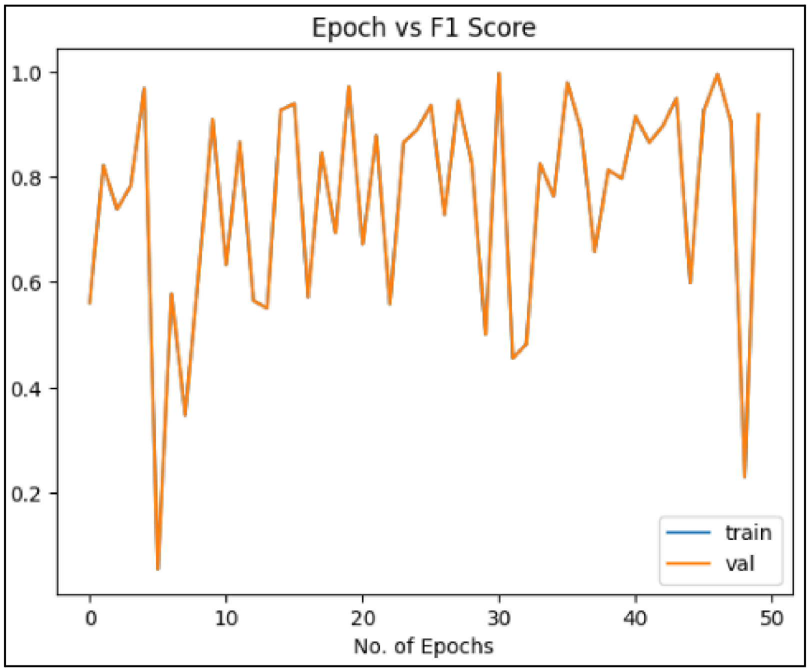
```
Epoch 11, F1 Score: 0.8219904543044017
Early stopping. Best F1 Score: 0.8266196562362274
937/937 [==============================] - 13s 13ms/step - loss: 0.0021
- accuracy: 0.9996 - val_loss: 0.2426 - val_accuracy: 0.9650 - lr: 0.0010
Total runtime: 152.8239767551422 seconds
```

(1.3d) F1-score on Development & Test Set:

(d) Report the **f1_score** on the test set, as well as the **f1_score** on the development set for each epoch during training.

FeedForward Network (FFN):

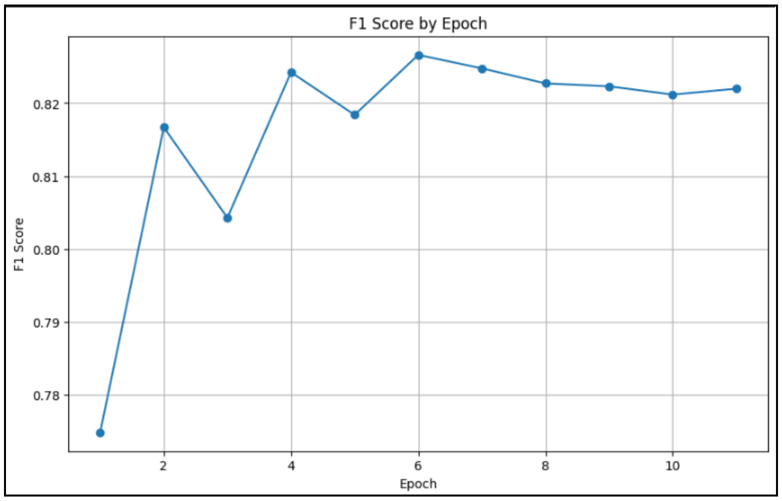
- The f1_score for the development set is in the figure below.
- The score for the development set is across 50 epochs.
- The scores are very unstable as the FFN was built on training on word level, which does not take in context. This would mean that the classification of each word would depend solely on the word itself and not the sentence. This led to a very unstable f1 score during training for the development set.
- The test f1_score = **0.9015**.



```
Test f1 Score: 0.090149818326613911
```

Bi-directional Long Short Term Memory (Bi-LSTM):

- The f1 score for the development set is as shown below.
- It is a lot more stable as compared to the FFN.
- As bi-directional takes in context from the sentence, it is able to learn and classify the words more accurately, which showed in the f1 score as it can be seen to be a lot more stable.
- The f1 score on the test dataset = **0.7384**.



```
Total training time: 152.8239767551422 seconds
F1 score on test set: 0.7383709891779001
```

Question 1 - Conclusion:

In conclusion, the Bi-LSTM model outperformed the FFN in terms of overall stability and reliability. Although the FFN exhibited competitive validation and test F1 scores at certain training episodes, its predictions lacked consistency compared to the Bi-LSTM. The limitation of the FFN lies in its inherent design; it processes input sequentially, word by word, without the capacity to capture the contextual dependencies effectively. This sequential processing precludes the FFN from recognizing long-range dependencies within the data, a crucial factor in understanding sentence-level nuances. In contrast, the Bi-LSTM benefits from its cell state mechanism that encapsulates information from both previous and subsequent contexts, thus enabling a more holistic and informed learning process. Consequently, the Bi-LSTM model demonstrated superior performance in the identification and classification of named entities, which became more pronounced with successive training iterations.

Question 2 - Sentence-Level Categorization: Question Classification:

(a) Specify all the 5 classes you used after converting from the original label set to the new setting.

(a) 5 Final Classes:

We randomized the selection of 2 classes, giving us labels 1 and 2.

```
labels = ["0","1","2","3","4","5"]

#get our random 2 labels that will be changed to OTHERS
np.random.seed(1)
twoLabels = np.random.choice(labels, size=2, replace = False)
twoLabels
```

```
Output: array(['2', '1'], dtype='<U1')
```

Labels 1 and 2 will be combined to form the new class “OTHERS”, thus giving us our 5 final classes:

- OTHERS
- 0
- 3
- 4
- 5

Before moving on, do note that we have employed 2 different models for question 2, a Feed Forward Network (FFN) and a Long Short Term Memory (LSTM). We tried experimenting with different models to see their performance at handling the data.

Ultimately, **LSTM performed the best**.

(b) Aggregation Method:

(b) Describe what aggregation methods you have tried and which is finally adopted (and why). Explain the detailed function of the aggregation method you used. If you have tested different aggregation methods, list their accuracy results to support your claim.

Data embedding and dealing With New Words in Train, Development & Test Set

Before passing the data into our models and conducting any aggregation techniques, we first have to conduct data preparation and make sure the data is in a suitable format for our models.

We replace words not found in the w2v with zero vectors, similar to question 1.3a. The vectors will have a size of (300,).

```
zero_vector = np.zeros(300)
zero_vector.shape

def embed(df):
    #duplicating the df
    copy = df.copy(deep=True)
    copy = copy.reset_index(drop=True)

    #this list will store all the lists of word embeddings of all sentences
    embedded_col = []

    #for each row
    for i in range(len(df)):

        #this list will store all word embeddings in 1 sentence
        embedded_text_list = []

        #for each word in the tokenized_text list
        for j in df["tokenized_text"][i]:
            try:
                #embed the word into its respective vector
                v = w2v[j]

                #append the vector to a list
                embedded_text_list.append(v)

            #we encounter a OOV word
            except:
                embedded_text_list.append(zero_vector)

        embedded_col.append(embedded_text_list)


    copy["embedded_text"] = embedded_col
    return copy
```

Padding Sentences:

Our LSTM model requires the sentences to have the same length. Hence we padded the shorter sentences with zero vectors in order to meet the requirement.
Longest sentence in:

- Train Dataset = 37 words
- Development Dataset = 32 words
- Test Dataset = 17 words

```
Total sentences in the training set: 4952
Max words in sentences: 37

Total sentences in the development set: 500
Max words in sentences: 32

Total sentences in the test set: 500
Max words in sentences: 17
```

```python
def padEmbedding(df, maxWords):
    #duplicating the df
    copy = df.copy(deep=True)
    copy = copy.reset_index(drop=True)

    padded_embedding = []

    for i in range(len(df)):
        embedding = copy["embedded_text"][i]
        numWords = len(embedding)
        numToPad = maxWords - numWords

        for i in range(numToPad):
            embedding.append(zero_vector)

        padded_embedding.append(embedding)

    copy["padded_embedding"] = padded_embedding

    return copy
```

Aggregation method for FeedForward Network (FFN):

For the Feed Forward Network model, the aggregation was done on the last hidden layer, before passing it into the output layer and subsequently the softmax classifier. The aggregation method we used was getting the mean of the vectors using torch.mean(). We found that this was the best aggregation method for our model.

```
# Last hidden layer (layer 4)
out = self.fc4(out)
out = self.relu4(out)

# Perform mean aggregation along the sequence dimension (dim=1)
out = torch.mean(out, dim=1, keepdim=False)

# Output layer
out = self.fc5(out)
out = self.softmax(out)
return out
```

Aggregation method for Long Short Term Memory (LSTM):
For our LSTM model, the aggregation that we did was to reshape the output from the last time step of our model, before passing it into the softmax classifier. The output is in the form of [batch_size, sequence_length, hidden_size].Hidden size is the number of LSTM units.

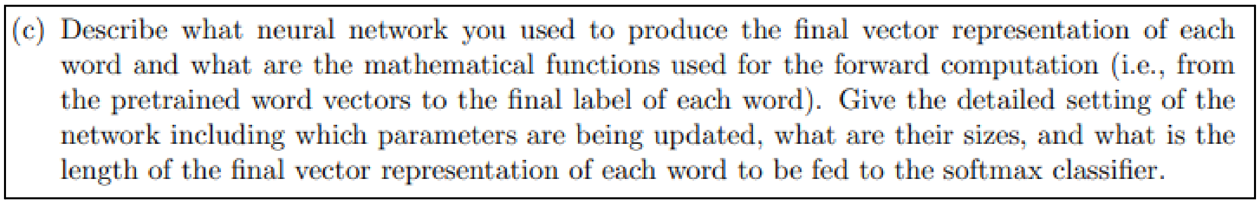
Subsequently, we slice the output using “out[:, -1, :]”, giving us the output from the last time step for each sequence in the batch, which is a tensor of shape [batch_size,hidden_size].

```
# Forward pass through LSTM
out, _ = self.lstm(X, (h0, c0))

# Only take the output from the last time step
out = self.fc(out[:, -1, :]) #this is our aggregation

# Apply softmax to the output for classification
out = self.softmax(out)
```

(c) Neural Network Description:



(c) Describe what neural network you used to produce the final vector representation of each word and what are the mathematical functions used for the forward computation (i.e., from the pretrained word vectors to the final label of each word). Give the detailed setting of the network including which parameters are being updated, what are their sizes, and what is the length of the final vector representation of each word to be fed to the softmax classifier.

FeedForward Network (FFN):

The neural network that we used to produce the final vector representation is a regular FFN.
The **architecture** is as follows:

- **Input layer**: The size of the input vector is calculated as follows:
  $input = ((batch\_size \ * \ number\ of\ words\ in\ each\ sentence\ in\ training\ data)\ x\ (word\ vector\ size))$
  $= ((128\ x\ 37)\ x\ 300) = (4736\ x\ 300)$
    - The batch_size and the number of words in each sentence were initialized during the data preparation phase, where we converted the data to the appropriate torch tensor format. The batch_size is 128 and the number of words is 37 (number of words in each sentence in the training data after padding).
    - Word vector size refers to the size of the word embeddings (in this case 300). This parameter is defined as ‘input_size’ as seen later and must be determined by the user.

- **4 Hidden layers**: 256 hidden neurons
    - 4 hidden layers were used as it provided a much higher accuracy than other layer sizes (we tested with a 2 and 3 layer model but they did not perform as well)
    - A ReLU activation function was used after each hidden layer to reduce gradient vanishing.

- **Output layer**:
    - Number of classes is 5. (6 originally but we combined 2 into 1)
    - At the last hidden layer, we pass the values through the mean aggregation function before passing them to 5 neurons at the output layer.
    - A softmax function is then used to classify the final word classes.

- **Optimizer**: We used Adam Optimizer.
- **Loss function**: We used the Cross Entropy Loss.

Mathematical functions:

- **Hidden layers:**
    - ReLU activation function.

- **Output layer:**
    - Cross Entropy Loss function.
    - Softmax activation function.
- Accuracy score calculation

```python
class FeedForwardNet(nn.Module):
    def __init__(self, input_size, hidden_size1, hidden_size2, hidden_size3, hidden_size4,
num_classes):
        super(FeedForwardNet, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size1)
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size1, hidden_size2)
        self.relu2 = nn.ReLU()
        self.fc3 = nn.Linear(hidden_size2, hidden_size3)
        self.relu3 = nn.ReLU()
        self.fc4 = nn.Linear(hidden_size3, hidden_size4)
        self.relu4 = nn.ReLU()
        self.fc5 = nn.Linear(hidden_size4, num_classes)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu1(out)
        out = self.fc2(out)
        out = self.relu2(out)
        out = self.fc3(out)
        out = self.relu3(out)

        # Last hidden layer (layer 4)
        out = self.fc4(out)
        out = self.relu4(out)

        # Perform mean aggregation along the sequence dimension (dim=1)
        out = torch.mean(out, dim=1, keepdim=False)

        # Output layer
        out = self.fc5(out)
        out = self.softmax(out)
        return out

input_size = 300  # Size of the word embeddings
hidden_size = 256  # Users are able to adjust the number of hidden neurons
num_classes = 5

model = FeedForwardNet(input_size, hidden_size, hidden_size, hidden_size, hidden_size,
num_classes)

# Loss function
criterion = nn.CrossEntropyLoss()

# Optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

Network settings (FFN):
Input layer:
- Parameters being updated: None
- Size: Vector of shape (4736 x 300)

Hidden layer 1:
- Parameters being updated: Weight matrix of input 1.
- Size: weight matrix of size (4736, 300) neurons (input layer) x (300, 256) neurons (hidden layer 1)

Hidden layers 2:
- Parameters being updated: Weight matrix of hidden layers 2.
- Size: weight matrix of size (300, 256) neurons (previous hidden layer) x (256, 256) neurons (current hidden layer)

Hidden layers 3-4:
- Parameters being updated: Weight matrix of hidden layers 3, 4.
- Size: weight matrix of size (256, 256)neurons (previous hidden layer) x (256, 256) neurons (current hidden layer)

Output layer (**w** Softmax Classifier):
- Parameters being updated: Weight matrix of output layer.
- Size: weight matrix of size 256 neurons (last hidden layer after aggregation) x 5 neurons (5 classes) = **1,280**

Additional Information:
- Optimal Batch size: 128
- Learning rate: 0.001
- Number of hidden neurons per layer: 256

Final vector Representation:
Final layer vector representation of each word: vector of length 5, each representing the raw scores before being passed into the softmax classifier to normalize the output to a probability distribution of a range from [0, 1].

Long Short Term Memory (LSTM):
The architecture used for generating the final vector representation of each word is a LSTM.
The **architecture** for the best performing set of parameters is as follows:

- **LSTM Layer**:
  - This input layer has 300 input features (input_size), representing the dimensions of each word embedding in a sequence. 64 LSTM units (hidden_size) in 1 LSTM layer. It is set to work with batch-first data, meaning the input data takes on the shape (batch_size: 64, sequence_length: 37, input_size: 300). Length of each question (sequence_length) was padded with zero vectors to the maximum length of 37 for all vectors.

- **Output layer**:
  - Number of classes is 5. (6 originally but we combined 2 into 1)
  - We take the output from the last time step for each sequence in the batch
  - A softmax function is then used to classify the final word classes.

- **Optimizer**: We used Adam Optimizer.

- **Loss function**: We used the Cross Entropy Loss function.

Mathematical functions:

- **LSTM Layer:**
  - Initialisation of hidden state and cell state as zeros

- **Output layer:**
  - Softmax activation function
  - Cross Entropy Loss function

- Accuracy score calculation

```
class LSTMModel(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size):
        super(LSTMModel, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, X):
        # Initialize hidden and cell state with zeros
        h0 = torch.zeros(self.num_layers, X.size(0), self.hidden_size)
        c0 = torch.zeros(self.num_layers, X.size(0), self.hidden_size)

        # Forward pass through LSTM
        out, _ = self.lstm(X, (h0, c0))

        # Only take the output from the last time step
        out = self.fc(out[:, -1, :]) #this is our aggregation

        # Apply softmax to the output for classification
        out = self.softmax(out)

        return out
```

```
model = LSTMModel(input_size, hidden_size, num_layers, output_size)
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=lr)
```

Network settings (LSTM):
LSTM layer:

- Parameters being updated: Weights for input features, previous hidden state and biases
- Input_size: 300 and hidden_size: 64
- Weights for input features with shape (input_size, 4 * hidden_size).
- Weights for the previous hidden state with shape (hidden_size, 4 * hidden_size).
- Biases with shape (4 * hidden_size).
- Total number of parameters = (300 * 4 * 64) + (64 * 4 * 64) + (4 * 64) = **93,504**

Output layer (**w** Softmax Classifier):

- Parameters being updated: Weight matrix of output layer.
- Size: weight matrix of size 64 (LSTM Units) x 5 neurons (5 classes) + 5 (bias term for each output node) = **325**

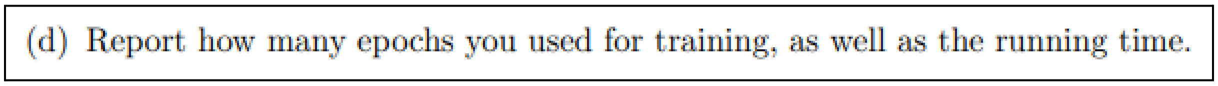
Additional Information:

- Optimal Batch size: 64
- Learning rate: 0.001
- Number of LSTM layers: 1
- Number of LSTM units per LSTM layer: 64

Final vector Representation:

- Length of final vector representation: 300
- This is fed the to the softmax classifier

(d) Epochs & Run-time:


(d) Report how many epochs you used for training, as well as the running time.

<u>FeedForward Network (FFN):</u>
Due to the complexity of a 4 hidden layer network, we limited the number of epochs to 100, including early stopping with patience 10 or if the loss goes below the max_loss parameter set at 0.955.

- The number of epochs used was 100. May vary due to early stop being implemented.
- The total time taken for training was 226.1418 seconds.

```
Epoch 50/100, Train Loss: 1.0643, Val Loss: 0.9530, Val Accuracy: 95.40%, F1 score: 0.86
Early stopping at epoch 50 with val loss 0.9530

Time taken: 226.1418 seconds
```

<u>Long Short Term Memory (LSTM):</u>
We limited the number of epochs to 1000, triggering early stop when development set cross entropy loss fell below 0.0152 and did not decrease any further for 3 consecutive epochs. This ensured training only stopped when sufficiently accurate training was complete.

- The number of epochs used was 416.
- The total time taken for training was 1016.509 seconds.

For the run with the set of best parameters:

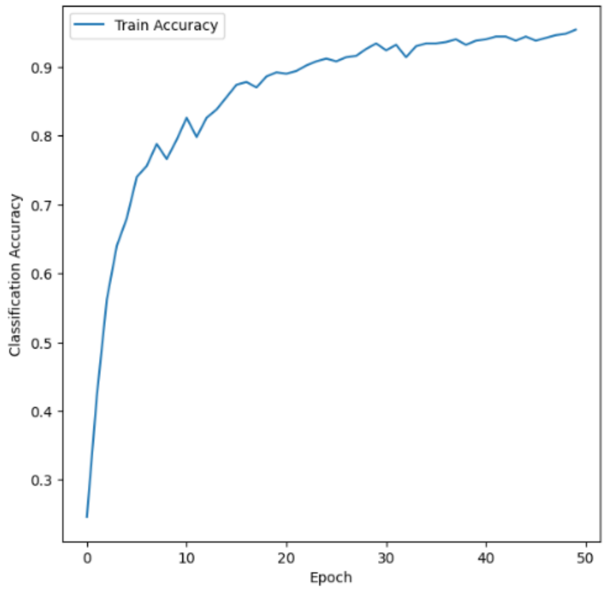
```
Epoch 416/1000, Val Accuracy: 95.60%
Time taken: 1016.509 seconds
```

(e) Accuracy on Development & Test Set:

> (e) Report the accuracy on the test set, as well as the accuracy on the development set for each epoch during training.

<u>FeedForward Network (FFN):</u>

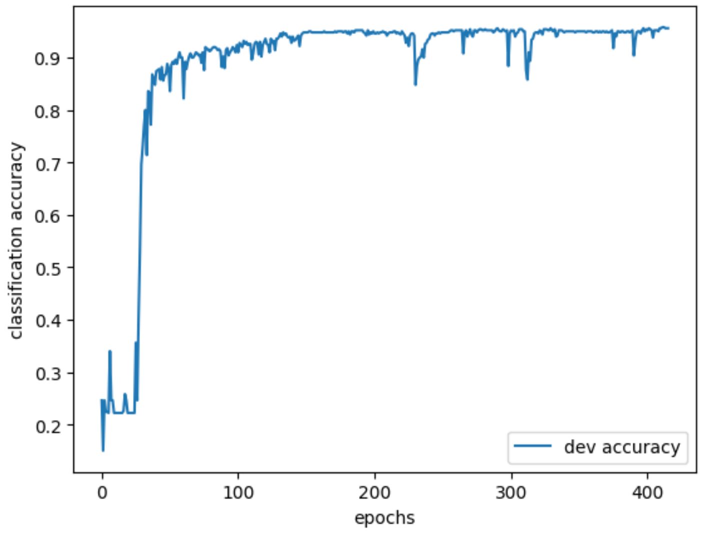
- The accuracy score for the development set is in the figure below.
- The score for the development set is across 50 epochs.
- Unlike in question 1, the score seems to be quite stable. This could be because the model takes in the entire sentence instead of just one word at a time, which takes into consideration the context of the sentences. Furthermore, the classification of sentences would depend on the sentence as a whole and not just the individual words.
- The test accuracy = **73.80%**



```
Test Accuracy: 73.80%
```

Long Short Term Memory (LSTM):

- The accuracy score for the development set using the best set of network parameters is in the figure below.
- This is across 416 epochs (early stopped).
- The test accuracy = **90.80%.**



```
Test Accuracy: 90.80%
```

Question 2 - Conclusion:

Ultimately, the LSTM model performed better than the regular FFN model.The LSTM model was able to triumph over the FFN model in terms of both validation scores and test scores. Similar to Question 1, this is likely due to the fact that LSTM are able to capture long range dependency of the sentence, whereas FFN is unable to.