

ECS7001P - NN & NLP

Assignment 2: Social Media, Information Extraction and Dialogue

Sparsh Verma 220996233

Part A

1. This code defines a function `get_model()` which returns a compiled Keras model that uses a pretrained DistilBERT model for text classification. The function takes three arguments: `use_tpu` (a boolean indicating whether to use a TPU), `use_gpu` (a boolean indicating whether to use a GPU), and `learning_rate` (the learning rate for the optimizer).

The model architecture consists of an input layer that takes in two inputs: `input_ids` (the tokenized input text) and `input_masks_ids` (a mask to indicate the padding in the input text). These inputs are passed through the `TFDistilBertModel` from the Hugging Face transformers library to get the output of the pretrained DistilBERT model. The output of the DistilBERT model is then passed through a custom layer called `GlobalAveragePooling1DMasked` that computes the average of the output values along the time dimension while taking the mask into account. Finally, the output of this layer is passed through a fully connected `Dense` layer with one output unit and a sigmoid activation function, which is used for binary classification.

The `get_model()` function first checks whether to use TPU or GPU. If `use_tpu` is `True`, it creates a TPU cluster resolver, initializes the TPU system, and creates the model using a `tf.distribute.TPUStrategy`. If `use_gpu` is `True`, it creates the model using a GPU. Otherwise, it creates the model on the CPU. The model is then compiled with an Adam

```
1 model.summary()
```

Model: "model"

Layer (type)	Output Shape	Param #	Connected to
input_ids (InputLayer)	[(None, 128)]	0	[]
input_masks_ids (InputLayer)	[(None, 128)]	0	[]
tf_distil_bert_model_1 (TFDistilBertModel)	TFBaseModelOutput(last_hidden_state=(None, 128, 768), hidden_states=None, attentions=None)	66362880	['input_ids[0][0]', 'input_masks_ids[0][0]']
global_average_pooling1d_masked_1 (GlobalAveragePooling1DMasked)	(None, 768)	0	['tf_distil_bert_model_1[0][0]']
dense (Dense)	(None, 1)	769	['global_average_pooling1d_masked_1[0][0]']

=====
Total params: 66,363,649
Trainable params: 769
Non-trainable params: 66,362,880

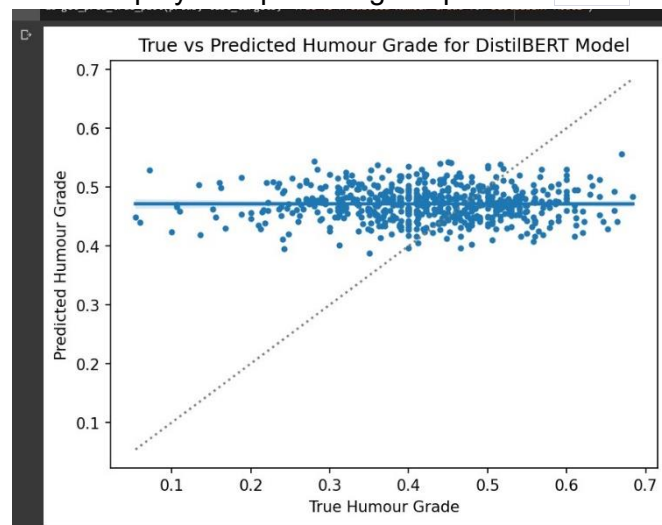
optimizer, mean squared error loss function, and mean squared error metric. Finally, the compiled Keras model is returned.

2. 'Preds' is an array of predicted values, 'labels' is an array of true values, and 'title' is a string used as the title of the plot.

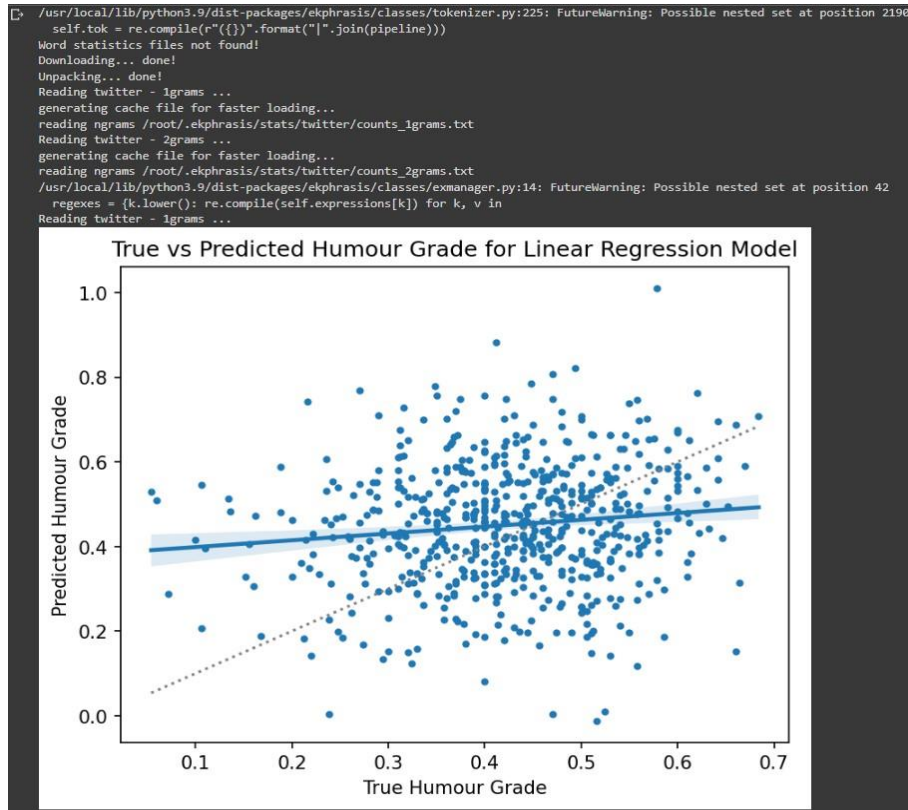
The function creates a scatter plot with the true values on the x-axis and the predicted values on the y-axis. It also includes a gray dashed line representing a perfect match between the true and predicted values.

In addition to the scatter plot, the function uses Seaborn's 'regplot' function to add a regression line to the plot, showing the overall trend between the true and predicted values.

Finally, the function displays the plot using Matplotlib's 'show' function.



The next code performs text pre-processing and feature extraction using Ekphrasis and Scikit-learn libraries, and then training a linear regression model to predict the humor grade of text examples. It also includes an evaluation step that plots the true vs predicted humor grades using a function named "get_pred_true_plot".



- The code is defining a function named `augment_data` that receives a DataFrame `df`, an augmentation object `aug` from the `nlpaug` package, and a string `col_name` representing the name of the column to be augmented. The function applies the augmentation to the specified column of the DataFrame and returns a new DataFrame with the augmented data concatenated with the original.

Then, the code applies this function to `train_df` using a `SynonymAug` object from the `nlpaug` package, and saves the augmented DataFrame as `aug_df`. The `humor_rating` column is normalized to a scale of 0 to 1 by dividing it by 5.

Finally, the code creates two lists of examples and targets for training and testing. The examples are the text data from the augmented DataFrame and the original test

```

[ ] Running on TPU grpc://10.65.180.42:8470
WARNING:tensorflow:TPU system grpc://10.65.180.42:8470 has already been initialized. Reinitializing the
WARNING:tensorflow:5 out of the last 5 calls to <function initialize_tpu_system.<locals>._tpu_init_fn at
Some layers from the model checkpoint at distilbert-base-uncased were not used when initializing TFDist
- This IS expected if you are initializing TFDistilBertModel from the checkpoint of a model trained on
- This IS NOT expected if you are initializing TFDistilBertModel from the checkpoint of a model that ye
All the layers of TFDistilBertModel were initialized from the model checkpoint at distilbert-base-uncas
If your task is similar to the task the model of the checkpoint was trained on, you can already use TFI
Epoch 1/10
32/32 [=====] - 23s 194ms/step - loss: nan - mean_squared_error: nan
Epoch 2/10
32/32 [=====] - 2s 72ms/step - loss: nan - mean_squared_error: nan
Epoch 3/10
32/32 [=====] - 2s 71ms/step - loss: nan - mean_squared_error: nan
Epoch 4/10
32/32 [=====] - 2s 73ms/step - loss: nan - mean_squared_error: nan
Epoch 5/10
32/32 [=====] - 2s 71ms/step - loss: nan - mean_squared_error: nan
Epoch 6/10
32/32 [=====] - 2s 74ms/step - loss: nan - mean_squared_error: nan
Epoch 7/10
32/32 [=====] - 2s 73ms/step - loss: nan - mean_squared_error: nan
Epoch 8/10
32/32 [=====] - 2s 73ms/step - loss: nan - mean_squared_error: nan
Epoch 9/10
32/32 [=====] - 2s 72ms/step - loss: nan - mean_squared_error: nan
Epoch 10/10
32/32 [=====] - 2s 73ms/step - loss: nan - mean_squared_error: nan

```

DataFrame, respectively, and the targets are the humor ratings normalized to a scale of 0 to 1.

4. This is a multi-task regression model using DistilBERT and TensorFlow. The model has two regression outputs. If 'use_tpu' is 'True', the model will be created with TPU strategy. Otherwise, if 'use_gpu' is 'True', the model will be created with GPU strategy. If neither of these is 'True', the model will be created on the CPU.

The 'create_TFBertMultitask' function creates the model architecture using 'TFDistilBertModel' from the Hugging Face transformers library. It takes in two inputs: 'input_ids' and 'input_masks_ids', and outputs two regression values: 'out_reg1' and 'out_reg2'. The 'compile' function specifies the loss function and the optimizer to be used for training the model.

1 model.summary()

Model: "model_7"

Layer (type)	Output Shape	Param #	Connected to
input_ids (InputLayer)	[(None, 100)]	0	[]
input_masks_ids (InputLayer)	[(None, 100)]	0	[]
tf_distil_bert_model_9 (TFDistilBertModel)	TFBaseModelOutput(last_hidden_state=(None, 100, 768), hidden_states=None, attentions=None)	66362880	['input_ids[0][0]', 'input_masks_ids[0][0]']
dropout_195 (Dropout)	(None, 100, 768)	0	['tf_distil_bert_model_9[0][0]']
global_average_pooling1d (GlobalAveragePooling1D)	(None, 768)	0	['dropout_195[0][0]']
out_reg1 (Dense)	(None, 1)	769	['global_average_pooling1d[0][0]']

=====
Total params: 66,363,649
Trainable params: 66,363,649
Non-trainable params: 0
=====

Part B

1. The input to the model appears to be a sequence of word embeddings. The 'Bidirectional' layer with GRU cells is used to capture the sequential information in both forward and backward directions. The 'return_sequences=True' parameter indicates that the output of the layer should be a sequence rather than a single vector.

Two such layers are defined in succession, each followed by a 'Dropout' layer. The 'Dropout' layer randomly sets a fraction of input units to 0 during training, which can prevent overfitting.

After the

```

Loading word embeddings from glove.6B.100d.txt.ner.filtered...
Finished loading word embeddings
Model: "model"

=====
Layer (type)                Output Shape                Param #
=====
input_2 (InputLayer)        [(None, None, 100)]        0
bidirectional (Bidirectiona  (None, None, 100)          45600
l)
dropout_1 (Dropout)         (None, None, 100)          0
bidirectional_1 (Bidirectio  (None, None, 100)          45600
nal)
dropout_2 (Dropout)         (None, None, 100)          0
dense (Dense)               (None, None, 50)           5050
dropout_3 (Dropout)         (None, None, 50)           0
dense_1 (Dense)             (None, None, 50)           2550
dropout_4 (Dropout)         (None, None, 50)           0
dense_2 (Dense)             (None, None, 5)            255
=====
Total params: 99,055
Trainable params: 99,055
Non-trainable params: 0
=====
Load 141 training batches from train.conll03.json
Load 33 dev batches from dev.conll03.json
Load 35 test batches from test.conll03.json

```

bidirectional GRU layers, the code defines a loop that applies a set of fully connected layers ('Dense') with ReLU activation and dropout. The number of fully connected layers is determined by the 'ffnn_layer' parameter.

Finally, the output layer is defined as a 'Dense' layer with softmax activation, which outputs a probability distribution over the set of NER labels defined in 'self.ner_labels'.

2. The code initializes temporary variables 'tp_temp', 'fn_temp', and 'fp_temp', each containing 5 elements initialized to 0. These arrays are used to count the true positives, false negatives, and false positives for each NER label. Then, the code iterates over each example in 'eval_fd_list' and applies the model to obtain a prediction. The predicted labels are compared to the gold labels to compute the counts of true positives, false positives, and false negatives for each NER label. The counts are accumulated in the temporary arrays. After processing all examples, the final counts are computed by multiplying the temporary arrays by the number of examples with each NER label, which is stored in the 'count' array.

The final values of 'tp', 'fn', and 'fp' are obtained by summing the values of the corresponding arrays. These values can be used to compute the precision, recall, and F1 score of the model.

```
Starting training epoch 1/5
141/141 [=====] - 38s 205ms/step - loss: 0.3591 - accuracy: 0.9386
Time used for epoch 1: 0 m 48 s
Evaluating on dev set after epoch 1/5:
F1 : 40.94%
Precision: 64.31%
Recall: 30.02%
Time used for evaluate on dev set: 1 m 14 s

Starting training epoch 2/5
141/141 [=====] - 26s 188ms/step - loss: 0.1130 - accuracy: 0.9657
Time used for epoch 2: 0 m 41 s
Evaluating on dev set after epoch 2/5:
F1 : 77.72%
Precision: 86.31%
Recall: 70.68%
Time used for evaluate on dev set: 1 m 19 s

Starting training epoch 3/5
141/141 [=====] - 26s 184ms/step - loss: 0.0780 - accuracy: 0.9766
Time used for epoch 3: 0 m 40 s
Evaluating on dev set after epoch 3/5:
F1 : 80.87%
Precision: 89.44%
Recall: 73.80%
Time used for evaluate on dev set: 1 m 10 s

Starting training epoch 4/5
141/141 [=====] - 26s 183ms/step - loss: 0.0641 - accuracy: 0.9804
Time used for epoch 4: 0 m 25 s
Evaluating on dev set after epoch 4/5:
F1 : 86.70%
Precision: 90.47%
Recall: 83.23%
Time used for evaluate on dev set: 1 m 6 s

Starting training epoch 5/5
141/141 [=====] - 25s 181ms/step - loss: 0.0566 - accuracy: 0.9829
Time used for epoch 5: 0 m 25 s
Evaluating on dev set after epoch 5/5:
F1 : 86.51%
Precision: 90.68%
Recall: 82.71%
Time used for evaluate on dev set: 1 m 7 s

Training finished!
Time used for training: 9 m 1 s

Evaluating on test set:
F1 : 83.98%
Precision: 88.75%
Recall: 79.70%
Time used for evaluate on test set: 1 m 11 s
```

Part C

1. The processing code defined 2 functions 'preprocess_arabic_text' and 'get_data'. The 'preprocess_arabic_text' function takes a string of Arabic text as input and removes the diacritics from the text using regular expressions. The function then returns the processed text. The 'get_data' function takes three arguments: 'json_file', 'is_training', and 'preprocess_text'. The 'json_file' argument is the path to a file containing JSON objects representing documents with coreferent mentions. The 'is_training' argument is a boolean indicating whether the data is being used for training or testing.

The `'preprocess_text'` argument is a boolean indicating whether to preprocess the Arabic text using the `'preprocess_arabic_text'` function.

The function reads each line of the JSON file and loads it as a dictionary using the `'json'` module. It checks that the document has coreferent mentions, and if it does not, it skips to the next document.

It then extracts the gold mentions, their cluster information, and the number of mentions using the `'get_mentions'` function (not shown in the code).

The function then splits the mentions into start and end indices and uses the `'tensorize_doc_sentences'` function (not shown in the code) to pad the document's sentences, create sentence embeddings, and create mention start and end indices for the padded document.

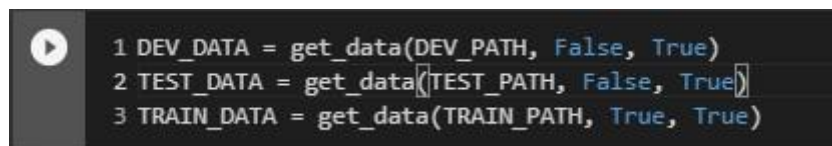
The function then uses the `'generate_pairs'` function (not shown in the code) to generate (anaphor, antecedent) pairs and their labels.

Finally, the function appends a tuple containing the processed document's word embeddings, mention pairs, mention pair labels, clusters, and raw mention pairs to a list and returns the list.

For `'DEV_DATA'` and `'TEST_DATA'`, the `'is_training'` argument is set to `'False'`, indicating that the data is not being used for training. The `'preprocess_text'` argument is set to `True`, indicating that the Arabic text should be preprocessed using the `'preprocess_arabic_text'` function.

For `'TRAIN_DATA'`, the `'is_training'` argument is set to `True`, indicating that the data is being used for training. The `'preprocess_text'` argument is also set to `True`, indicating that the Arabic text should be preprocessed using the `'preprocess_arabic_text'` function.

The resulting variables `'DEV_DATA'`, `'TEST_DATA'`, and `'TRAIN_DATA'` will contain lists of tuples, with each tuple representing a processed document. Each tuple contains the word embeddings for the document, the mention pairs and their labels, the clusters, and the raw mention pairs.



```
1 DEV_DATA = get_data(DEV_PATH, False, True)
2 TEST_DATA = get_data(TEST_PATH, False, True)
3 TRAIN_DATA = get_data(TRAIN_PATH, True, True)
```

2. The following are the steps for building this model:

- a. Initializes the input layers:
 - i. `word_embeddings`: a 4D tensor of shape `(batch_size, num_sents, num_words, embedding_size)` that represents the word embeddings of each word in the input document.
 - ii. `mention_pairs`: a 4D tensor of shape `(batch_size, num_pairs, 4)` that represents the start and end indices of the anaphor and antecedent mentions in each pair.
- b. Applies a dropout layer to the word embeddings, then passes them through two bidirectional LSTM layers. The output is a 3D tensor of shape `(batch_size, num_sents, 2 * hidden_size)`.
- c. Extracts the word embeddings of each mention in each pair and flattens them into a 2D tensor of shape `(batch_size * num_pairs, 8 * hidden_size)`.

- d. Applies two feedforward neural network layers to the flattened mention embeddings, with dropout after each layer. The final output is a 2D tensor of shape `(batch_size * num_pairs, 1)`.
- e. Squeezes out the last dimension of the output tensor using a lambda layer.
- f. Defines the model with the input layers and output layer, compiles it with the Adam optimizer and binary cross-entropy loss function, and returns it.

Overall, this model is designed to take in word embeddings and identify pairs of anaphoric mentions and their antecedents, outputting a binary score indicating whether each pair is coreferent or not.

3. The `evaluate_coref()` function evaluates the predicted clusters against the gold clusters using the provided evaluator. It takes in the predicted mention pairs, the gold clusters, and the evaluator object as inputs.
 The first step is to convert each cluster in the gold cluster list into a tuple of tuples (one tuple per mention in the cluster), as required by the evaluator.
 Next, the function generates a mapping of mentions to their corresponding gold clusters using the `mention_to_gold` dictionary.
 The `get_predicted_clusters()` function is then called to obtain the predicted clusters and a mapping of mentions to their corresponding predicted clusters.
 Finally, the evaluator is updated with the predicted and gold clusters, as well as the two mention-to-cluster mappings.

```
[85] 1 def evaluate_coref(predicted_mention_pairs, gold_clusters, evaluator):
      2
      3     # turn each cluster in the list of gold cluster into a tuple (rather than a list)
      4     gold_clusters = [tuple(tuple(mention) for mention in gc) for gc in gold_clusters]
      5
      6     # mention to gold is a {mention: cluster of mentions it belongs, including the present mention} map
      7     mention_to_gold = {}
      8     for gold_cluster in gold_clusters:
      9         for mention in gold_cluster:
     10             mention_to_gold[mention] = gold_cluster
     11
     12     # get the predicted_clusters and mention_to_predict using get_predicted_clusters()
     13     predicted_clusters, mention_to_predicted = get_predicted_clusters(predicted_mention_pairs)
     14
     15     # run the evaluator using the parameters you've gotten
     16     evaluator.update(predicted_clusters, gold_clusters, mention_to_predicted, mention_to_gold)
```

This `eval()` function takes in a trained model and a list of evaluation documents, which contain word embeddings, mention pairs, gold clusters, and raw mention pairs. For each evaluation document, the function generates mention pair scores using the trained model and extracts predicted antecedents for each anaphor. Then, it uses the `evaluate_coref()` function to evaluate the predicted mention pairs against the gold clusters and update the evaluation metrics. Finally, after evaluating each document, it prints the average precision, recall, and F1 score based on the updated metrics.

Overall, this function evaluates the performance of the model on the evaluation documents by computing precision, recall, and F1 score for coreference resolution.


```

1 def eval(model, eval_docs):
2     coref_evaluator = metrics.CoreEvaluator()
3
4     for word_embeddings, mention_pairs, _, gold_clusters, raw_mention_pairs in eval_docs:
5
6         # get the mention pair scores from the model
7         mention_pair_scores = model.predict_on_batch([word_embeddings, mention_pairs])
8
9         predicted_antecedents = {}
10        best_antecedent_scores = {}
11        # for a given anaphor
12        for (ana, ant), score in zip(raw_mention_pairs, mention_pair_scores[0]):
13            # only candidate antecedents with (ana, ante) above 0.5 are considered as valid system proposed candidates
14            if score >= 0.5 and score > best_antecedent_scores.get(ana, 0):
15                # we chose the best among these to be the predicted antecedent for that anaphor
16                predicted_antecedents[ana] = ant
17                best_antecedent_scores[ana] = score
18
19        # getting the [anaphor, antecedent] pairs.
20        predicted_mention_pairs = [[k,v] for k,v in predicted_antecedents.items()]
21
22        # evaluate the predicted mention pairs
23        evaluate_coref(predicted_mention_pairs, gold_clusters, coref_evaluator)
24
25        # after evaluating each document, get the conll prf
26        p, r, f = coref_evaluator.get_prf()
27        print("Average F1 (py): {:.2f}%".format(f * 100))
28        print("Average precision (py): {:.2f}%".format(p * 100))
29        print("Average recall (py): {:.2f}%".format(r * 100))

```

4. Would the performance decrease if we do not preprocess the text? If yes (or no), then why?

based on "Anaphora Resolution with Real Preprocessing" paper by Klener et al. . It is evident that the quality of preprocessing determines the quality of the rest, namely, the decision made by linguistic filters and the classification carried out by the machine learning classifier. Different preprocessing procedures effect the precision and recall which mean it would eventually effect f1 score. The reason for that is preprocessed versions lose some of the morphological features within the words and between them. How would you improve the accuracy?

If we could add some more features before we feed the input to the neural network model it would be better. things like name entities. Also I think GRUs will work slightly better than LSTM here. if the context here is important, we could try and make embeddings trainable so that it biases toward the context better.

Experiment with different values for max antecedent (MAX_ANT) and negative ratio (NEG_RATIO), what do you observe?

We can see that how the different values effect the average f1 score.

MAX_ANT	NEG_Ratio	AVG F1(py)
---------	-----------	------------

250	2	46.32%
-----	---	--------

100	2	40.25%
-----	---	--------

400	2	40.38%
-----	---	--------

MAX_ANT	NEG_Ratio	AVG F1(py)
---------	-----------	------------

250	1	35.05%
-----	---	--------

250	3	45.49%
-----	---	--------

250	5	46.57%
-----	---	--------

The best result was for MaX_ENT of 250 and NEG_RATIO of 5. By my experiments there seems to be 250 is a reasonable value for MAX_ENT and increasing the NEG_RATIO would slightly increase the score.

EMBEDDING_DROPOUT_RATE	AVG F! (py)
------------------------	-------------

0.2	42.22%
-----	--------

0.1	42.71%
-----	--------

0.0	43.36%
-----	--------

0.0 & NEG_RATIO = 4	45.86%
---------------------	--------

0.0 & NEG_RATIO = 5	46.57%
---------------------	--------

Part D

1. Embedding layer: This layer converts integer-encoded vocabulary into dense vectors of fixed size. The input shape is determined by the VOCAB_SIZE and EMBED_SIZE parameters.

Bidirectional LSTM layer 1: This layer has 43 hidden units and returns sequences to the next layer. The Bidirectional wrapper allows the network to capture information from both forward and backward directions.

Bidirectional LSTM layer 2: This layer has 43 hidden units and returns a single vector to the next layer. Again, the Bidirectional wrapper is used.

Dense layer: This layer has `HIDDEN_SIZE` units and applies a linear transformation to the input.

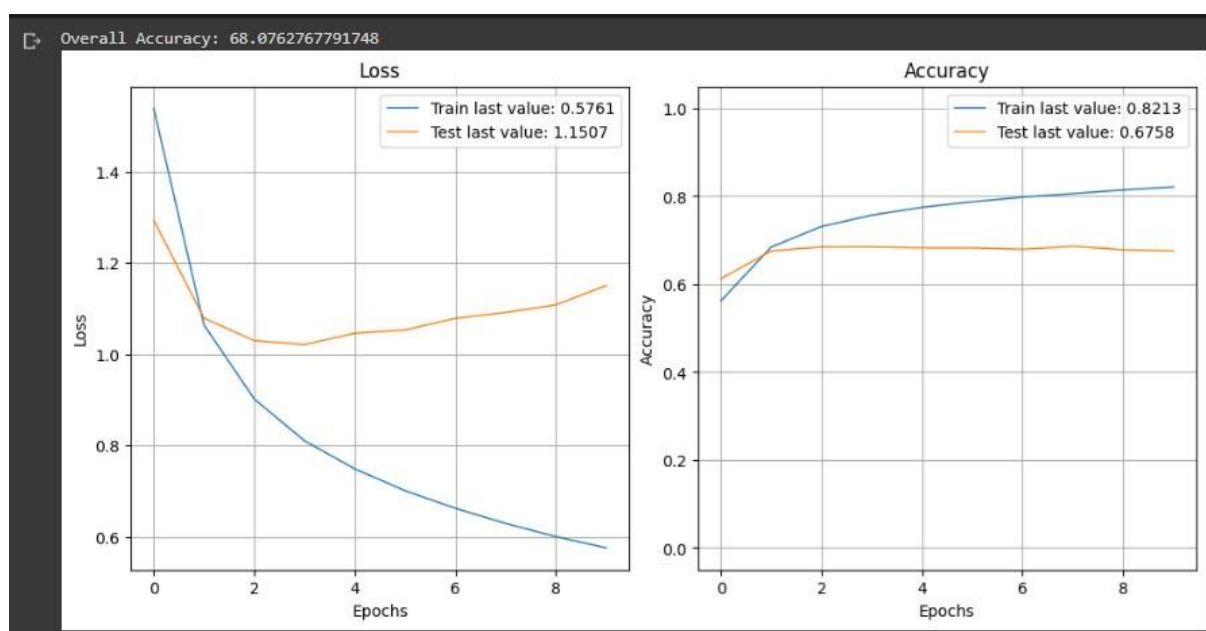
Activation layer: This layer applies a softmax activation function to produce the output probabilities for each class.

The `model.compile()` method compiles the model and specifies the optimizer, loss function, and metrics to be used during training.

```
Model: "sequential"
```

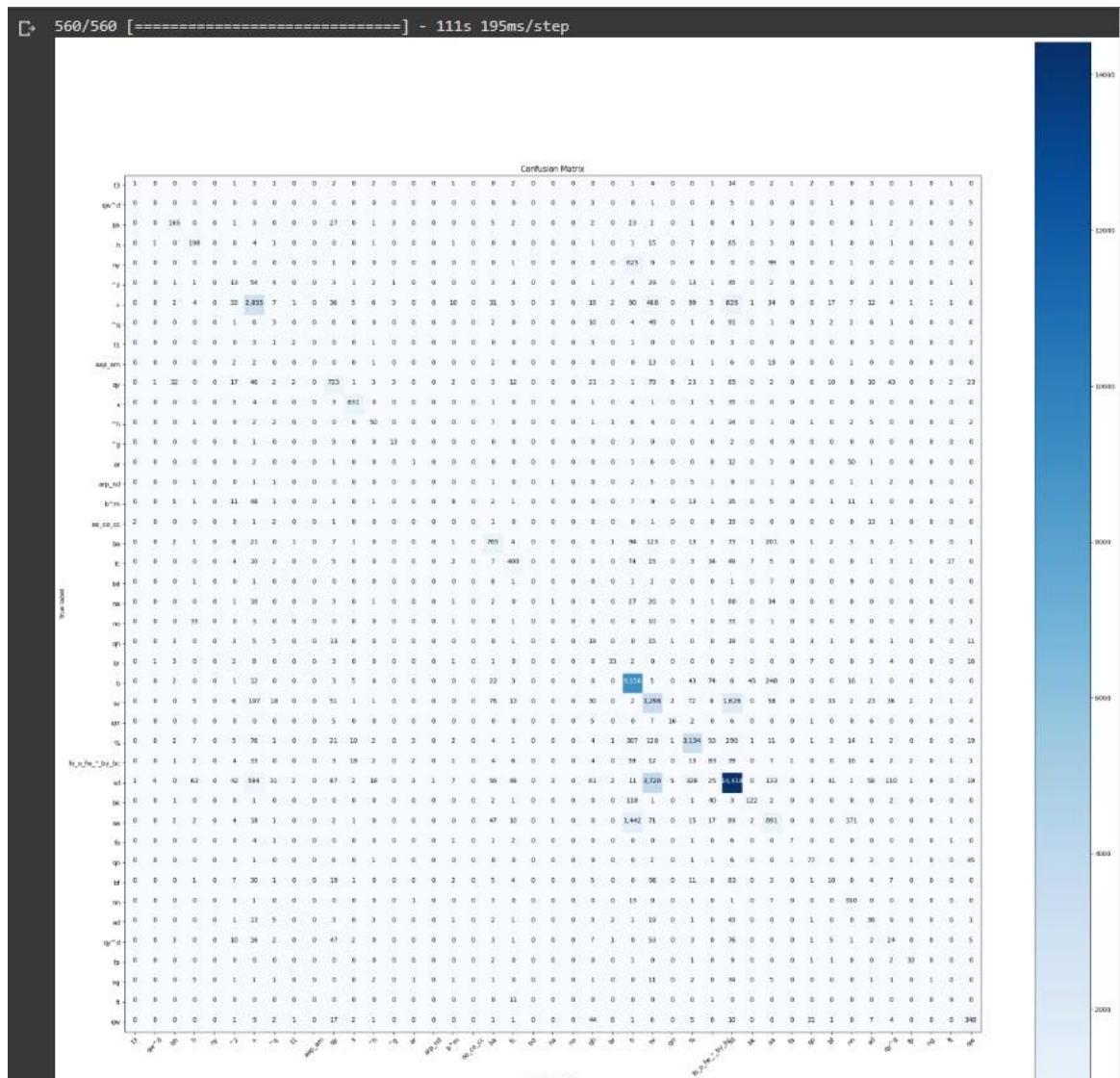
Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 150, 100)	4373300
bidirectional_2 (Bidirectional)	(None, 150, 86)	49536
bidirectional_3 (Bidirectional)	(None, 86)	44720
dense_3 (Dense)	(None, 43)	3741
activation (Activation)	(None, 43)	0

```
=====
Total params: 4,471,297
Trainable params: 4,471,297
Non-trainable params: 0
=====
```



2. `plot_confusion_matrix`: This function plots a confusion matrix using Matplotlib. The function takes the following arguments:
 - `cm`: The confusion matrix as a NumPy array.
 - `target_names`: A list of target class names.
 - `title`: The title of the plot.
 - `cmap`: The color map to use for the plot.
 - `normalize`: A Boolean indicating whether to normalize the confusion matrix.

`decoding_dic`: This function creates a dictionary to decode one-hot encoded data.
 The `plot_confusion_matrix` function is useful for visualizing the performance of a classification model. The confusion matrix shows how many samples were classified correctly and incorrectly for each class. The diagonal elements of the matrix represent the number of correct predictions for each class, while the off-diagonal elements represent the number of misclassifications. The `normalize` argument can be used to normalize the values in the matrix to the range [0, 1], making it easier to compare performance across classes with different numbers of samples.
 The `decoding_dic` function is used to decode the one-hot encoded data by creating a dictionary that maps the index of the maximum value in each one-hot encoded vector to the corresponding class name.



Position: 24
 Instance : 86
 Position: 35
 Instance : 290
 accuracy of br is : 0.38372093023255816
 accuracy of bf is : 0.034482758620689655

- [illegible]

The resulting tensor is then reshaped to have shape (MAX_LENGTH, EMBED_SIZE, 1).

The model then applies three convolutional layers of sizes [3, 4, 5] with num_filters filters each. The convolutional layers slide a window over the input tensor and produce a feature map for each window position. The size of the window is (filter_size, EMBED_SIZE) where filter_size is the size of the convolutional window in terms of the number of words.

Batch normalization is applied to each convolutional layer to improve convergence and generalization.

The maximum value over the feature maps for each convolutional window is then pooled to create a feature vector for each convolutional layer.

```
10
11 # CNN model
12 inputs = Input(shape=(MAX_LENGTH, ), dtype='int32')
13 embedding = Embedding(input_dim=VOCAB_SIZE + 1, output_dim=EMBED_SIZE, input_length=MAX_LENGTH)(inputs)
14 reshape = Reshape((MAX_LENGTH, EMBED_SIZE, 1))(embedding)
15
16 # 3 convolutions
17 conv_0 = Conv2D(num_filters, kernel_size=(filter_sizes[0], EMBED_SIZE), strides=1, padding='valid', kernel_initializer=
18 bn_0 = BatchNormalization()(conv_0)
19 conv_1 = Conv2D(num_filters, kernel_size=(filter_sizes[1], EMBED_SIZE), strides=1, padding='valid', kernel_initializer=
20 bn_1 = BatchNormalization()(conv_1)
21 conv_2 = Conv2D(num_filters, kernel_size=(filter_sizes[2], EMBED_SIZE), strides=1, padding='valid', kernel_initializer=
22 bn_2 = BatchNormalization()(conv_2)
23
24 # maxpool for 3 layers
25 maxpool_0 = MaxPool2D(pool_size=(MAX_LENGTH - filter_sizes[0] + 1, 1), padding='valid')(bn_0)
26 maxpool_1 = MaxPool2D(pool_size=(MAX_LENGTH - filter_sizes[1] + 1, 1), padding='valid')(bn_1)
27 maxpool_2 = MaxPool2D(pool_size=(MAX_LENGTH - filter_sizes[2] + 1, 1), padding='valid')(bn_2)
28
29 # concatenate tensors
30 concatenated_tensors = concatenate([maxpool_0, maxpool_1, maxpool_2])
31 # flatten concatenated tensors
32 flattened_tensors = TimeDistributed(Flatten())(concatenated_tensors)
33 # dense layer (dense_1)
34 dense_1 = Dense(EMBED_SIZE, activation='relu')(flattened_tensors)
35 # dropout_1
36 dropout_1 = Dropout(drop)(dense_1)
```

The last layer of the model is a dense layer with softmax activation function, which means that it outputs a probability distribution over the tags for each input word. The dropout_2 layer applies dropout regularization to the outputs of the dense layer to prevent overfitting. Finally, the output of the dropout_2 layer is flattened into a 1D tensor, which is then passed to the output layer.

```
1 # BLSTM model
2
3 # Bidirectional 1
4 Bidirectional_1=Bidirectional(LSTM(EMBED_SIZE,return_sequences=True))(dropout_1)
5 # Bidirectional 2
6 Bidirectional_2=Bidirectional(LSTM(EMBED_SIZE,return_sequences=False))(Bidirectional_1)
7 # Dense layer (dense_2)
8 dense_2=Dense(EMBED_SIZE, activation='softmax')(Bidirectional_2)
9 flattened_layer = Flatten()(dropout_1)
10 # dropout_2
11 dropout_2 = Dropout(drop)(dense_2)
```

Concatenate 2 final layer

```

embedding_2 (Embedding)      (None, 150, 100)    4373200    ['input_3[0][0]']
reshape (Reshape)            (None, 150, 100, 1) 0      ['embedding_2[0][0]']
conv2d (Conv2D)              (None, 148, 1, 64)  19264      ['reshape[0][0]']
conv2d_1 (Conv2D)            (None, 147, 1, 64)  25664      ['reshape[0][0]']
conv2d_2 (Conv2D)            (None, 146, 1, 64)  32064      ['reshape[0][0]']
batch_normalization (BatchNorm (None, 148, 1, 64)  256      ['conv2d[0][0]']
alization)
batch_normalization_1 (BatchNo (None, 147, 1, 64)  256      ['conv2d_1[0][0]']
rmalization)
batch_normalization_2 (BatchNo (None, 146, 1, 64)  256      ['conv2d_2[0][0]']
rmalization)
max_pooling2d (MaxPooling2D)   (None, 1, 1, 64)    0      ['batch_normalization[0][0]']
max_pooling2d_1 (MaxPooling2D) (None, 1, 1, 64)    0      ['batch_normalization_1[0][0]']
max_pooling2d_2 (MaxPooling2D) (None, 1, 1, 64)    0      ['batch_normalization_2[0][0]']
concatenate (Concatenate)      (None, 1, 1, 192)   0      ['max_pooling2d[0][0]',
'max_pooling2d_1[0][0]',
'max_pooling2d_2[0][0]']
time_distributed (TimeDistribu (None, 1, 192)      0      ['concatenate[0][0]']
ted)
dense_5 (Dense)               (None, 1, 100)      19300     ['time_distributed[0][0]']
dropout_5 (Dropout)           (None, 1, 100)      0      ['dense_5[0][0]']
bidirectional_6 (Bidirectional (None, 1, 200)      160800    ['dropout_5[0][0]']
)
bidirectional_7 (Bidirectional (None, 200)         240800    ['bidirectional_6[0][0]']
)
dense_6 (Dense)               (None, 100)         20100     ['bidirectional_7[0][0]']
flatten_1 (Flatten)           (None, 100)         0      ['dropout_5[0][0]']
dropout_6 (Dropout)           (None, 100)         0      ['dense_6[0][0]']
concatenate_1 (Concatenate)    (None, 200)         0      ['flatten_1[0][0]',
'dropout_6[0][0]']
dense_7 (Dense)               (None, 43)          8643      ['concatenate_1[0][0]']

=====
Total params: 4,900,603
Trainable params: 4,900,219
Non-trainable params: 384

```

Part E

1. The `Encoder` takes in the vocabulary size, embedding dimension, and the number of encoder units as input parameters. I then define an embedding layer and two Bidirectional GRU layers.

In the `call()` method of the `Encoder` class, I first apply the embeddings to the input sequence `x`, then apply dropout to the embeddings. I then pass the embeddings through the first Bidirectional GRU layer and apply dropout again before passing the output of the first Bidirectional GRU layer to the second Bidirectional GRU layer. Finally, I return the output and the states from the second Bidirectional GRU layer.

I also define an `initialize_hidden_state()` method that returns a tensor of zeros with a shape of `(batch_size, encoder_units)`, which is used as the initial hidden state of the Bidirectional GRU layers.

```
1 class Encoder(tf.keras.Model):
2     def __init__(self, vocab_size, embedding_dim, enc_units):
3         super(Encoder, self).__init__()
4         self.batch_sz = batch_size
5         self.enc_units = enc_units
6         self.embeddings = Embedding(vocab_size, embedding_dim)
7
8         # pass the embedding into a bidirectional version of the GRU
9         #
10        self.dropout = Dropout(0.2)
11        self.Inp = Input(shape=(max_len_q,)) # size of questions
12
13        self.Bidirectional1 = Bidirectional(GRU(self.enc_units, return_state=False, return_sequences=True, recurrent_initializer='glorot_uniform'))
14        self.Bidirectional2 = Bidirectional(GRU(self.enc_units, return_state=True, return_sequences=True, recurrent_initializer='glorot_uniform'))
15
16    def bidirectional(self, bidir, layer, inp, hidden):
17        return bidir(layer(inp, initial_state=hidden))
18
19    def call(self, x, hidden):
20
21        x = self.embeddings(x)
22        x = self.dropout(x) # dropout 1
23        x = self.Bidirectional1(x)
24        x = self.dropout(x) # dropout 2
25        x = self.Bidirectional2(x)
26        output, state_f, state_b = x
27
28        return output, state_f, state_b
29
30    def initialize_hidden_state(self):
31        return tf.zeros((self.batch_sz, self.enc_units))
```

2. Now attention layer named BahdanauAttention. It takes two inputs: query and values, both of which are hidden states with shape `(batch_size, hidden_size)`. The layer then calculates a score between query and each element in values, using a neural network with two dense layers and the hyperbolic tangent activation function. The score is then passed through a dense layer with one output to get a single scalar score for each element in the values. The scores are then normalized using the softmax function along the time axis to get the attention weights. The context vector is then obtained by taking the element-wise multiplication of attention weights and values and then summing over the time axis. The output of this layer is the context vector and attention weights.

Then a well-defined implementation of a Decoder class for sequence-to-sequence (Seq2Seq) models with attention. The decoder consists of a bidirectional GRU layer followed by a single GRU layer, with a Bahdanau attention mechanism applied at each decoding step.

The `call` method takes in the input sequence (`x`), the decoder hidden state (`hidden`) and the output of the encoder (`enc_output`). It first computes the attention vector and weights for the current hidden state and encoder output. The input sequence is then passed through an embedding layer, and the context vector is concatenated with the embedded input to create the final decoder input. This concatenated vector is then

passed through the decoder GRU layers, followed by a dense layer with a softmax activation function to generate the output probabilities for the next word in the sequence. The method returns the output probabilities, the decoder state and the attention weights.

Overall, this implementation is consistent with the architecture of standard Seq2Seq models with attention, and should work well for a variety of sequence-to-sequence tasks.

3. Did the models learn to track local relations between words?

Yes, the models do learn to track local relation between words. If we consider the question 'What do you want to eat', the matrix shows increased correlation between words of the question and the answer as well as in the question and answer itself.

Did the models attend to the least frequent tokens in an utterance? Can you see signs of overfitting in models that hang on to the least frequent words?

Yes the model did attend to the least frequent tokens in the utterance. If you consider the question 'what is your favourite restaurant', there was considerable difference in the output to this question. However, we can see that no answer is relevant to the question and the model is giving random answers to the question. This shows signs of overfitting.

Did the models learn to track some major syntactic relations in the utterances (subject-verb, verb-object)?

Yes it was able to track the syntactic relations correctly. Even if it did struggle even in the 140th epoch where it says 'i am not sure you are not' which is grammatically incorrect, it was able to get the Noun - verb - object concatenated structures correctly as well.

Do they learn to encode some other linguistic features? Do they capture part-of-speech tags (POS tags)?

They were indeed able to capture various linguistic features like Tense. It is able to detect the present tense and answer in the appropriate sense. They are also able to detect the POS Tags like for eg in the response 'i am seventyfour' it correctly identified the structure of Subject - Verb - Object.

What is the effect of more training on the length of responses?

The training did not really affect the length of responses. Sometimes the length got less and then got more again and vice versa. So the length of the response is not related to the epochs.

In some instances, by the time the decoder has to generate the beginning of a response, it may already forget the most relevant early query tokens. Can you suggest ways to change the training pipeline to make it easier for the model to remember the beginning of the query when it starts to generate the response?

We can experiment with the usage of other algorithms like CNN or LSTM to make the pipeline better. Sutskever et al., 2014; Bahdanau et al., 2014; Luong et al., 2015 suggest the usage of Neural generative models which are able to map dialogue history directly into the response in current turn, while requires a minimum amount of hand-crafting.