ECS7001P - NN & NLP ASSIGNMENT 2: Pre-trained Transformers, Information Extraction and Dialogue

Animesh Devendra Chourey – 210765551

Queen Mary University of London – April 30, 2022

Part A: Using Pre-trained BERT

Task 1: Data preprocessing :

Dataset Setup -

```
from xml.etree.ElementTree import parse

def parse_sentence_term(path, lowercase=False):
    tree = parse(path)
    sentences = tree.getroot()
    data = []
    split_char = '__split__'
    for sentence in sentences:
        text = sentence.find('text')
        if text is None:
            continue
        text = text.text
        if lowercase:
            text = text.lower()
        aspectTerms = sentence.find('aspectTerms')
        if aspectTerms is None:
            continue
        for aspectTerm in aspectTerms:
            term = aspectTerm.get('term')
            if lowercase:
                term = term.lower()
            polarity = aspectTerm.get('polarity')
            start = aspectTerm.get('from')
            end = aspectTerm.get('to')
            piece = [text , term,  polarity , start , end]
            data.append(piece)
    return data
train = parse_sentence_term("train.xml",True)
dev = parse_sentence_term("val.xml",True)
test = parse_sentence_term("test.xml",True)

print("Training entries: {}".format(len(train)))
print("Test entries: {}".format(len(test)))
```

Conversion to index and mask sequences, with separator tokens between text and aspect-

```
x_train_int = []
x_train_masks = []
```

```
for row in train:
  # Combining the review and aspect of training data with <SEP> as special token
  # and then tokenizing them
  ids_train, masks_train, segments_train = tokenize(row[0] + "<SEP>" + row[1], tokenizer)
  x_train_int.append(ids_train)
  x_train_masks.append(masks_train)

x_dev_int = []
x_dev_masks = []
for row in dev:
  # Combining the review and aspect of dev data with <SEP> as special token
  # and then tokenizing them
  ids_dev, masks_dev, segments_dev = tokenize(row[0] + "<SEP>" + row[1], tokenizer)
  x_dev_int.append(ids_dev)
  x_dev_masks.append(masks_dev)

x_test_int = []
x_test_masks = []
for row in test:
  # Combining the review and aspect of test data with <SEP> as special token
  # and then tokenizing them
  ids_test, masks_test, segments_test = tokenize(row[0] + "<SEP>" + row[1], tokenizer)
  x_test_int.append(ids_test)
  x_test_masks.append(masks_test)
```

Task 2: Basic classifiers using BERT: Model 1 and Model 2:

Model 1 - Prebuilt Sequence Classification

```
model.summary()

Model: "model"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_token (InputLayer)       [(None, 128)]        0           []

 masked_token (InputLayer)      [(None, 128)]        0           []

 tf_distil_bert_for_sequence_cl  TFSequenceClassifie  66955779   ['input_token[0][0]',
 assification (TFDistilBertForS  rOutput(loss=None,               'masked_token[0][0]']
 equenceClassification)         logits=(None, 3),
                                 hidden_states=None
                                , attentions=None)

==================================================================================================
Total params: 66,955,779
Trainable params: 66,955,779
Non-trainable params: 0
__________________________________________________________________________________________________
```

Model 2 - Neural bag of words using BERT

```
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_token (InputLayer)       [(None, 128)]        0           []

 masked_token (InputLayer)      [(None, 128)]        0           []

 tf_distil_bert_model (TFDistil  TFBaseModelOutput(l  66362880   ['input_token[0][0]',
 BertModel)                     ast_hidden_state=(N               'masked_token[0][0]']
                                one, 128, 768),
                                 hidden_states=None
                                , attentions=None)

 global_average_pooling1d_maske  (None, 768)         0           ['tf_distil_bert_model[0][0]']
 d (GlobalAveragePooling1DMaske
 d)

 dense (Dense)                  (None, 16)           12304       ['global_average_pooling1d_masked
                                                                 [0][0]']

 dense_1 (Dense)                (None, 3)            51          ['dense[0][0]']

==================================================================================================
Total params: 66,375,235
Trainable params: 66,375,235
Non-trainable params: 0
__________________________________________________________________________________________________
```

The accuracy obtained by the models in lab 4 is as following:

- Neural bag of words without pre-trained word embeddings - **51.6%**
- CNN or LSTM without pre-trained word embeddings - **50.1%**
- Neural bag of words using pre-trained word embeddings - **57.8%**
- CNN or LSTM with pre-trained word embeddings - **64.7%**
- Neural bag of words model with multiple-input - **51.6%**
- CNN or LSTM model with multiple-input - **63.6%**

The accuracy obtained by the models in this lab is as follow:

- Prebuilt Sequence Classification - **79.8%**

```
results = model.evaluate([x_test_int_np,x_test_masks_np], y_test)
print(results)

42/42 [==============================] - 8s 98ms/step - loss: 0.8981 - accuracy: 0.7987
[0.8980957865715027, 0.798652708530426]
```

- Neural bag of words using BERT - **83%**

```
results = model2.evaluate([x_test_int_np,x_test_masks_np], y_test)
print(results)

42/42 [==============================] - 8s 104ms/step - loss: 0.3774 - accuracy: 0.8301
[0.3773697018623352, 0.8300898671150208]
```

As we can see, model 1 and 2 using BERT produces exceptional results compared to the models used in lab-4. This is because, not all tasks can be easily represented by a transformer encoder-decoder architecture, and therefore requires a task-specific model architecture to be added. Rather than glove vectors used in lab 4 we are using BERT embeddings. BERT is bidirectional which combines masks with the sentence predictions i.e. predicting the missing word in the sentence. Also, parameters needed to train the model have also been reduced by the BERT model. Therefore, model 1 and model 2 achieves higher accuracy with low computational costs.

Task 3: Advanced classifier using BERT: Model 3 :

Implementing the LSTM on top of BERT helps in getting the context because of which we see slight increase in the accuracy obtained. The model here achieves accuracy of **83.8%**. The reason for higher accuracy is because the model has LSTM units which factors in better backpropogation.

The code used to implement the model is as following:

```
hdepth=16
MAX_SEQUENCE_LENGTH = 128
EMBED_SIZE=100

def create_bag_of_words_BERT_CNN():
  input_ids_in = tf.keras.layers.Input(shape=(128,), name='input_token', dtype='int32')
  input_masks_in = tf.keras.layers.Input(shape=(128,), name='masked_token', dtype='int32')

  bert_embeddings = get_BERT_layer()
  # Embedding Layer
  embedded_sent = bert_embeddings(input_ids_in, attention_mask=input_masks_in)[0]
  # LSTM Layer
  lstm_layer = LSTM(units=100)(embedded_sent)
```

```
  # Output Layer
  label=Dense(3,input_shape=(hdepth,),activation='softmax',
                          kernel_initializer='glorot_uniform')(lstm_layer)
  return Model(inputs=[input_ids_in,input_masks_in], outputs=[label],name='Model2_BERT')

use_tpu = True
if use_tpu:
  # Create distribution strategy
  tpu = tf.distribute.cluster_resolver.TPUClusterResolver()
  tf.config.experimental_connect_to_cluster(tpu)
  tf.tpu.experimental.initialize_tpu_system(tpu)
  strategy = tf.distribute.experimental.TPUStrategy(tpu)

  # Create model
  with strategy.scope():
    model3 = create_bag_of_words_BERT_CNN()
    optimizer3 = tf.keras.optimizers.Adam(lr=5e-5)
    model3.compile(optimizer=optimizer3, loss='binary_crossentropy', metrics=['accuracy'])
else:
  model3 = create_bag_of_words_BERT_CNN()
  model3.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])


model3.summary()
```

```
 Layer (type)                Output Shape         Param #     Connected to
==================================================================================================
 input_token (InputLayer)    [(None, 128)]        0           []

 masked_token (InputLayer)   [(None, 128)]        0           []

 tf_distil_bert_model_6 (TFDist  TFBaseModelOutput(l  66362880  ['input_token[0][0]',
 ilBertModel)                ast_hidden_state=(N               'masked_token[0][0]']
                             one, 128, 768),
                              hidden_states=None
                             , attentions=None)

 lstm_2 (LSTM)               (None, 100)          347600      ['tf_distil_bert_model_6[0][0]']

 dense_11 (Dense)            (None, 3)            303         ['lstm_2[0][0]']

==================================================================================================
Total params: 66,710,783
Trainable params: 66,710,783
Non-trainable params: 0
```

```
results = model3.evaluate([x_test_int_np,x_test_masks_np], y_test)
print(results)
```

```
42/42 [==============================] - 10s 114ms/step - loss: 0.4874 - accuracy: 0.8383
[0.4874451458454132, 0.8383233547210693]
```

Part B - Information Extraction 1: Training a Named Entity Resolver :

Task 1: Create a bidirectional GRU and Multi-layer FFNN :

```
def build(self):
    word_embeddings = Input(shape=(None,self.embedding_size,))
    word_embeddings = Dropout(self.embedding_dropout_rate)(word_embeddings)
    """
    Task 1 Create a two layer Bidirectional GRU and Multi-layer FFNN to compute the ner scores for individual tokens
    The shape of the ner_scores is [batch_size, max_sentence_length, number_of_ner_labels]
    """
    # Bi-directional GRU1 having 50 GRU units i.e. 100 cells with recurrent dropout = 0.2 and return_sequences = True
    word_output = Bidirectional(GRU(50, return_sequences = True, recurrent_dropout = 0.2))(word_embeddings)
    # Bi-directional GRU2 having 50 GRU units i.e. 100 cells with recurrent dropout = 0.2 and return_sequences = True
    word_output = Bidirectional(GRU(50, return_sequences = True, recurrent_dropout = 0.2))(word_output)
    # Dropout layer having rate of 0.2
    dropout_layer_1 = Dropout(self.hidden_dropout_rate)(word_output)
    # Dense layer having 50 neurons and the ReLU Activation Function
    dense_layer_1 = Dense(self.hidden_size, activation='relu')(dropout_layer_1)
    # Dropout layer having rate of 0.2
    dropout_layer_2 = Dropout(self.hidden_dropout_rate)(dense_layer_1)
    # Dense layer having 50 neurons and the ReLU Activation Function
    dense_layer_2 = Dense(self.hidden_size, activation='relu')(dropout_layer_2)
    # Dropout layer having rate of 0.2
    dropout_layer_3 = Dropout(self.hidden_dropout_rate)(dense_layer_2)
    # Output Layer having 5 output neurons and the SoftMax Activation Function
    ner_scores = Dense(5, activation ='softmax')(dropout_layer_3)
    """
    End Task 1
    """
```

The models contain an input layer with shape (None, None,100). Then a dropout is added having 0.5 as embeddings dropout rate to create the word embeddings. These word embeddings are then passed to a Bidirectional GRU layer having 50 units i.e., 100 GRU cells(bidirectional) with a recurrent dropout rate of 0.2. The word_output of GRU is then passed to another GRU layer with similar configurations. After those 3 dropouts and 2 Dense layers are added in the Network with dropout rate as 0.2 and 50 hidden cells in dense layers. At the end, ner_score output layer is that with softmax activation function because of multiple output. Adam Optimizer along with sparse categorical cross-entropy loss function is used to train the model and the metrics is accuracy.

Task 2: Form the predicted named entities :

```
def eval(self, eval_fd_list):
  tp, fn, fp = 0,0,0
  for word_embeddings, _, gold,sent_lens in eval_fd_list:
    predictions = self.model.predict_on_batch([word_embeddings])

    """
    Task 2 create the predictions of NER from the IO label
    e.g.
    0 I         O
    1 met       O
    2 John      PER
    3 this      O
    4 afternoon O
    should give you a person NE John (x,2,2,1)
    where x is the sentence id in the batch, and 2,2 are the start and end indices of the NE,
    1 is the id for 'PER'
    """
    ner_labels_predicted = np.argmax(predictions, axis = 2)
    pred_set = set()
    for i, sent in enumerate(ner_labels_predicted):
      ner_end = 0
      ner_begin = 0
      sent = np.append(sent,0)
      for j, word in enumerate(sent):
        if ner_end != word and ner_end == 0 :
          ner_begin = j
        elif ner_end != 0 and ner_end != word:
          pred_set.add((i,ner_begin, j-1, int(ner_end)))
          ner_end = 0
          if word != 0:
            ner_begin=j
            ner_end = word
          continue
        ner_end = word
    tp += len(gold.intersection(pred_set))
    fp += len(pred_set.difference(gold))
    fn += len(gold.difference(pred_set))
    """
    End Task 2
    """
```

We get the label for each word in a sentence by applying argmax on the axis 2 of the *predictions* variable. An empty set is then created.In the loop for every extracted word labels and index are iterated. A copy of labels and index is stored inside the tuple (*pred_set*). After that false positives, true positives and false negatives are calculated. For true positives, intersection of pred_set and gold is calculated and then the length is added to the variable tp. For the false positives, the difference of the pred_set and gold is taken and then the length of it is added to the variable fp. Finally, for false negatives, the difference of gold and pred_set is taken and then it's length is added to the variable fn

```
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, None, 100)]       0

bidirectional (Bidirectiona  (None, None, 100)         45600
l)

bidirectional_1 (Bidirectio  (None, None, 100)         45600
nal)

dropout_1 (Dropout)          (None, None, 100)         0

dense (Dense)                (None, None, 50)          5050

dropout_2 (Dropout)          (None, None, 50)          0

dense_1 (Dense)              (None, None, 50)          2550

dropout_3 (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

Starting training epoch 1/5
141/141 [==============================] - 67s 378ms/step - loss: 0.3517 - accuracy: 0.9362
Time used for epoch 1: 1 m 7 s
Evaluating on dev set after epoch 1/5:
F1 : 28.05%
Precision: 39.02%
Recall: 21.89%
Time used for evaluate on dev set: 0 m 8 s

Starting training epoch 2/5
141/141 [==============================] - 43s 306ms/step - loss: 0.1084 - accuracy: 0.9669
Time used for epoch 2: 0 m 43 s
Evaluating on dev set after epoch 2/5:
F1 : 63.16%
Precision: 65.57%
Recall: 60.92%
Time used for evaluate on dev set: 0 m 2 s

Starting training epoch 3/5
141/141 [==============================] - 61s 432ms/step - loss: 0.0750 - accuracy: 0.9777
Time used for epoch 3: 1 m 0 s
Evaluating on dev set after epoch 3/5:
F1 : 73.11%
Precision: 73.31%
Recall: 72.90%
Time used for evaluate on dev set: 0 m 2 s

Starting training epoch 4/5
141/141 [==============================] - 63s 446ms/step - loss: 0.0576 - accuracy: 0.9831
Time used for epoch 4: 1 m 2 s
Evaluating on dev set after epoch 4/5:
F1 : 79.14%
Precision: 80.46%
Recall: 77.87%
Time used for evaluate on dev set: 0 m 2 s

Starting training epoch 5/5
141/141 [==============================] - 61s 433ms/step - loss: 0.0489 - accuracy: 0.9854
Time used for epoch 5: 1 m 1 s
Evaluating on dev set after epoch 5/5:
F1 : 80.87%
Precision: 83.75%
Recall: 78.17%
Time used for evaluate on dev set: 0 m 2 s
```

```
Training finished!
Time used for training: 3 m 8 s

Evaluating on test set:
F1 : 75.46%
Precision: 75.03%
Recall: 75.90%
Time used for evaluate on test set: 0 m 1 s
```

Part C - Information Extraction 2: A Coreference Resolver for Arabic

Task 1: Preprocessing:

```
def get_data(json_file, is_training, preprocess_text):
    processed_docs = []

    for line in open(json_file):

      # read the document in
      doc = json.loads(line)

      # check that there are coreferent mentions in this document
      clusters = doc['clusters']

      sentences = doc['sentences']

      if(preprocess_text==True):
          preprocessed_sents = [[preprocess_arabic_text(t) for t in sent] for sent in sentences]
          doc['sentences'] = preprocessed_sents

      if len(clusters) == 0:
          continue

      #  get the mentions and their cluster information.
      gold_mentions, gold_mention_map, cluster_ids, num_mentions =  get_mentions(clusters)# TASK 1.1 YOUR CODE HERE

      # splits the mentions into two arrays, one representing the start indices,
      # and the other for the end indices
      raw_starts, raw_ends = zip(*gold_mentions)

      # pad sentences, create glove sentence embeddings, create mention starts and ends for padded document
      word_emb, starts, ends = tensorize_doc_sentences(doc['sentences'], gold_mentions) # TASK 1.2 YOUR CODE HERE

      # generate (anaphor, antecedent) pairs and their labels
      mention_pairs, mention_pair_labels, raw_mention_pairs =  generate_pairs(num_mentions,cluster_ids, starts,
                                                                              ends, raw_starts, raw_ends,
                                                                              is_training) # TASK 1.3 YOUR CODE HERE
      mention_pairs, mention_pair_labels = np.array(mention_pairs),np.array(mention_pair_labels)

      # add the processed document to the list
      processed_docs.append((word_emb, mention_pairs, mention_pair_labels, clusters, raw_mention_pairs))

    return processed_docs
```

Variable *cluster* is passed to the function *get_mentions* to get the mentions and their cluster information. *tensorize_doc_sentences* function is used to generate the padded documents embeddings, copy of the mention starts and end indices adjusted for padding. FInally, *generate_pairs* function returns the pairs of (anaphora indexes, antecedents indexes) and their labels.

Task 2: Building the model:

```python
def build_model():
    # 1 (a.) Initialize the model inputs
    word_embeddings = Input(shape = (None, None, EMBEDDING_SIZE,)) # YOUR CODE HERE
    mention_pairs = Input(shape = (None, 4,), dtype = 'int32') # TASK 2.1a YOUR CODE HERE

    # squeeze the (batch_size X num_sents X num_words X embedding_size) into a
    # (num_sents X num_words X embedding_size) tensor
    word_embeddings_no_batch = Lambda(lambda x: K.squeeze(x,0))(word_embeddings)

    # 1 (b.). Apply embedding dropout to the squeezed embeddings.
    word_embeddings_dropped = Dropout(EMBEDDING_DROPOUT_RATE)(word_embeddings_no_batch)# TASK 2.1b YOUR CODE HERE

    # TASK 2.2. YOU CREATE A TWO LAYER BIDIRECTIONAL LSTM
    word_output = Bidirectional(LSTM(HIDDEN_SIZE, return_sequences = True), name = 'BiLSTM_1')(word_embeddings_dropped)
    word_output = Bidirectional(LSTM(HIDDEN_SIZE, return_sequences = True), name = 'BiLSTM_2')(word_output)

    # flattening the lstms output and apply dropout.
    flatten_word_output = Lambda(lambda x:K.reshape(x, [-1, 2 * HIDDEN_SIZE]))(word_output)
    flatten_word_output = Dropout(HIDDEN_DROPOUT_RATE)(flatten_word_output)

    # we gather the embeddings represented by [anaphor_start, anaphor_end, antecedent_start, antecedent_end] for each pair.
    mention_pair_emb = Lambda(lambda x: K.gather(x[0], x[1]))([flatten_word_output, mention_pairs])

    # we flatten them such that each mention_pair is represented by a 400D tensor.
    ffnn_input = Reshape((-1,8*HIDDEN_SIZE))(mention_pair_emb)

    # TASK 2.3. CREATE THE MULTILAYER PERCEPTRONS THEN SQUEEZE OUT THE LAST DIMENSION USING LAMBDA
    DenseLayer1 = Dense(HIDDEN_SIZE, activation = 'relu')(ffnn_input)
    DenseLayer1 = Dropout(HIDDEN_DROPOUT_RATE)(DenseLayer1)

    DenseLayer2 = Dense(HIDDEN_SIZE, activation = 'relu')(DenseLayer1)
    DenseLayer2 = Dropout(HIDDEN_DROPOUT_RATE)(DenseLayer2)

    DenseLayer3 = Dense(1, activation = 'sigmoid')(DenseLayer2)

    mention_pair_scores = Lambda(lambda x:K.squeeze(x,-1))(DenseLayer3)

    model = Model(inputs=[word_embeddings,mention_pairs], outputs=mention_pair_scores)
    model.compile(optimizer='adam',loss='binary_crossentropy')
    print(model.summary())
    return model
```

```
model = build_model()
```

```
Model: "model"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_1 (InputLayer)           [(None, None, None,  0           []
                                 300)]

 lambda (Lambda)                (None, None, 300)    0           ['input_1[0][0]']

 dropout (Dropout)              (None, None, 300)    0           ['lambda[0][0]']

 BiLSTM_1 (Bidirectional)       (None, None, 100)    140400      ['dropout[0][0]']

 BiLSTM_2 (Bidirectional)       (None, None, 100)    60400       ['BiLSTM_1[0][0]']

 lambda_1 (Lambda)              (None, 100)          0           ['BiLSTM_2[0][0]']

 dropout_1 (Dropout)            (None, 100)          0           ['lambda_1[0][0]']

 input_2 (InputLayer)           [(None, None, 4)]    0           []

 lambda_2 (Lambda)              (None, None, 4, 100  0           ['dropout_1[0][0]',
                                )                                 'input_2[0][0]']

 reshape (Reshape)              (None, None, 400)    0           ['lambda_2[0][0]']

 dense (Dense)                  (None, None, 50)     20050       ['reshape[0][0]']

 dropout_2 (Dropout)            (None, None, 50)     0           ['dense[0][0]']

 dense_1 (Dense)                (None, None, 50)     2550        ['dropout_2[0][0]']

 dropout_3 (Dropout)            (None, None, 50)     0           ['dense_1[0][0]']

 dense_2 (Dense)                (None, None, 1)      51          ['dropout_3[0][0]']

 lambda_3 (Lambda)              (None, None)         0           ['dense_2[0][0]']

==================================================================================================
Total params: 223,451
Trainable params: 223,451
Non-trainable params: 0
__________________________________________________________________________________________________
```

Task 3: Coreference evaluation :

```
def evaluate_coref(predicted_mention_pairs, gold_clusters, evaluator):

    # turn each cluster in the list of gold cluster into a tuple (rather than a list)
    gold_clusters = [[tuple(m) for m in cl] for cl in gold_clusters] # TASK 3.1 CODE HERE

    # mention to gold is a {mention: cluster of mentions it belongs, including the present mention} map
    mention_to_gold = {}
    # TASK 3.2 WRITE CODE HERE TO GENERATE mention_to_gold from gold_clusters
    for cluster in gold_clusters:
      for mention in cluster:
        mention_to_gold[mention] = tuple(cluster)

    # get the predicted_clusters and mention_to_predict using get_predicted_clusters()
    predicted_clusters, mention_to_predicted = get_predicted_clusters(predicted_mention_pairs) # TASK 3.3 CODE HERE

    # run the evaluator using the parameters you've gotten
    evaluator.update(predicted_clusters, gold_clusters, mention_to_predicted, mention_to_gold)
```

```
Training finished!
Time used for training: 12 m 2 s

Evaluating on test set:
Average F1 (py): 39.01%
Average precision (py): 43.87%
Average recall (py): 59.21%
Time used for evaluate on test set: 0 m 2 s
```

Task 4: Some questions :

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

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

```
Training finished!
Time used for training: 13 m 19 s

Evaluating on test set:
Average F1 (py): 36.24%
Average precision (py): 41.78%
Average recall (py): 49.40%
Time used for evaluate on test set: 0 m 2 s
```

  As we can see from the results below is that the F1 score has decreased slightly from 39% to 36%. By using pre-processing we are removing the punctuations, stop words and turn them into lower case. By doing so, we are decreasing the feature space dimension, which ultimately results in avoiding sparsity and also eliminating the redundant tokens.

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

  1. *MAX_ANT = 100 and NEG_RATIO = 2*

```
# the maximum number of candidate antecedents we will give to each of the candidate mentions.
MAX_ANT = 100

# the ratio of negative to postive examples
NEG_RATIO = 2
```

```
Training finished!
Time used for training: 12 m 58 s

Evaluating on test set:
Average F1 (py): 40.73%
Average precision (py): 45.17%
Average recall (py): 58.57%
Time used for evaluate on test set: 0 m 2 s
```

2. *MAX_ANT = 200 and NEG_RATIO = 4*

```
# the maximum number of candidate antecedents we will give to each of the candidate mentions.
MAX_ANT = 200

# the ratio of negative to postive examples
NEG_RATIO = 4
```

```
Training finished!
Time used for training: 13 m 40 s

Evaluating on test set:
Average F1 (py): 46.19%
Average precision (py): 50.14%
Average recall (py): 55.47%
Time used for evaluate on test set: 0 m 2 s
```

3. *MAX_ANT = 300 and NEG_RATIO = 2*

```
# the maximum number of candidate antecedents we will give to each of the candidate mentions.
MAX_ANT = 300

# the ratio of negative to postive examples
NEG_RATIO = 2
```

```
Training finished!
Time used for training: 12 m 59 s

Evaluating on test set:
Average F1 (py): 40.07%
Average precision (py): 44.80%
Average recall (py): 58.35%
Time used for evaluate on test set: 0 m 2 s
```

- **How would you improve the accuracy?**

  We can conclude form the above experiments that the F1 score is not good for the highly imbalanced data. The one inference is that the keeping the NEG_RATIO high especially lower MAX_ANT will give us lower F1 score. One thing which can be concluded is that as accuracy is highly sensitive to imbalanced data, it would be good to take smaller NEG_RATIO and smaller MAX_ANT to improve the accuracy.

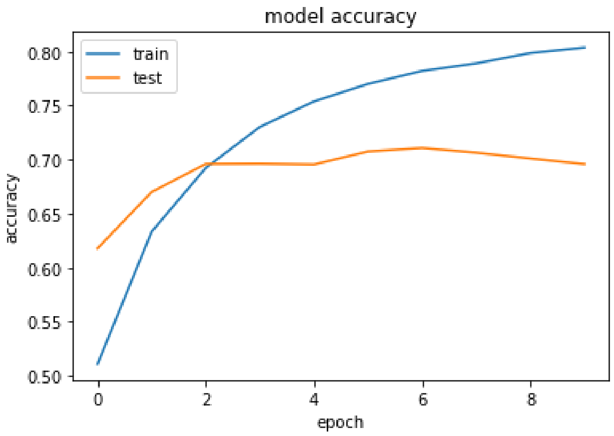
Part D - Dialogue 1: Dialogue Act Tagging

Task 1: Implementing an utterance-based tagger, using standard text classification methods from lectures :

```
#Building the network

# Include 2 BLSTM layers, in order to capture both the forward and backward hidden states
model = Sequential()
# Embedding layer
model.add(Embedding(VOCAB_SIZE, 100, input_length = MAX_LENGTH))
# Bidirectional 1
model.add(Bidirectional(LSTM(HIDDEN_SIZE, return_sequences= True)))
# Bidirectional 2
model.add(Bidirectional(LSTM(HIDDEN_SIZE)))
# Dense layer
```

```
model.add(Dense(HIDDEN_SIZE, activation=’relu’))
# Activation
model.add(Activation(’softmax’))

model.compile(loss=’categorical_crossentropy’,optimizer=’adam’,metrics=[’accuracy’])

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 150, 100)          4373200

 bidirectional (Bidirectiona  (None, 150, 86)          49536
 l)

 bidirectional_1 (Bidirectio  (None, 86)               44720
 nal)

 dense (Dense)               (None, 43)                3741

 activation (Activation)     (None, 43)                0

=================================================================
Total params: 4,471,197
Trainable params: 4,471,197
Non-trainable params: 0
_________________________________________________________________
```




```
# Train the model - use validation
history = model.fit(train_input, train_labels, batch_size=512, epochs=10, validation_data=(val_input, val_labels))

Epoch 1/10
274/274 [==============================] - 57s 166ms/step - loss: 1.9848 - accuracy: 0.5087 - val_loss: 1.7809 - val_accuracy: 0.5477
Epoch 2/10
274/274 [==============================] - 43s 158ms/step - loss: 1.4433 - accuracy: 0.6313 - val_loss: 1.4223 - val_accuracy: 0.6287
Epoch 3/10
274/274 [==============================] - 43s 158ms/step - loss: 1.2320 - accuracy: 0.6858 - val_loss: 1.3840 - val_accuracy: 0.6299
Epoch 4/10
274/274 [==============================] - 43s 158ms/step - loss: 1.1426 - accuracy: 0.7095 - val_loss: 1.3711 - val_accuracy: 0.6350
Epoch 5/10
274/274 [==============================] - 44s 159ms/step - loss: 1.0888 - accuracy: 0.7238 - val_loss: 1.3762 - val_accuracy: 0.6360
Epoch 6/10
274/274 [==============================] - 44s 160ms/step - loss: 1.0533 - accuracy: 0.7335 - val_loss: 1.3566 - val_accuracy: 0.6408
Epoch 7/10
274/274 [==============================] - 44s 159ms/step - loss: 1.0239 - accuracy: 0.7408 - val_loss: 1.3881 - val_accuracy: 0.6381
Epoch 8/10
274/274 [==============================] - 43s 159ms/step - loss: 0.9988 - accuracy: 0.7471 - val_loss: 1.4132 - val_accuracy: 0.6360
Epoch 9/10
274/274 [==============================] - 43s 158ms/step - loss: 0.9767 - accuracy: 0.7531 - val_loss: 1.4197 - val_accuracy: 0.6366
Epoch 10/10
274/274 [==============================] - 43s 158ms/step - loss: 0.9579 - accuracy: 0.7571 - val_loss: 1.4557 - val_accuracy: 0.6361

score = model.evaluate(test_sentences_X, y_test, batch_size=100)

560/560 [==============================] - 31s 56ms/step - loss: 1.3569 - accuracy: 0.6603

print("Overall Accuracy:", score[1]*100)

Overall Accuracy: 66.02804660797119
```

Task 2: Minority DA tag class analysis and utterance-based tagger with re-balanced weighted cost function :

```
# Calculate Accuracies for "br" and "bf"
acc_class = confusion_matrix.diagonal()/confusion_matrix.sum(axis=1)

index_br = list(one_hot_encoding_dic["br"][one_hot_encoding_dic["br"]==1].index)[0]
br_accuracy = acc_class[index_br]*100
print("br accuracy: {}".format(br_accuracy))

index_bf = list(one_hot_encoding_dic["bf"][one_hot_encoding_dic["bf"]==1].index)[0]
bf_accuracy = acc_class[index_bf]*100
print("bf accuracy: {}".format(bf_accuracy))
```

```
br accuracy: 51.78571428571429
bf accuracy: 2.366863905325444
```

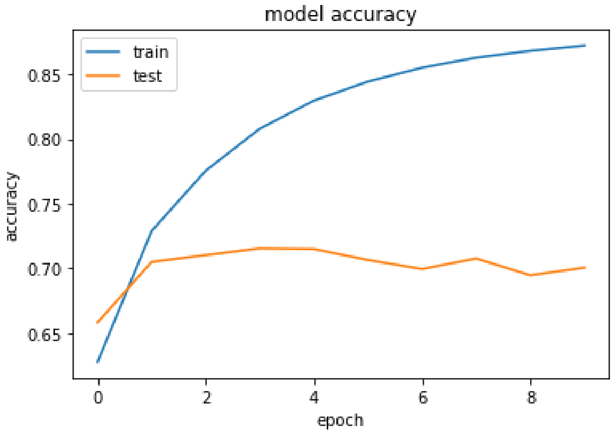
Task 3: Implementing a hierarchical utterance+DA-context-based tagger :

```
# Re-built the model for the balanced training
model_balanced = Sequential()
# Embedding layer
model_balanced.add(Embedding(VOCAB_SIZE, 100, input_length = MAX_LENGTH))
# Bidirectional 1
model_balanced.add(Bidirectional(LSTM(HIDDEN_SIZE, return_sequences= True)))
# Bidirectional 2
model_balanced.add(Bidirectional(LSTM(HIDDEN_SIZE)))
# Dense layer
model_balanced.add(Dense(HIDDEN_SIZE, activation='relu'))
# Activation
model_balanced.add(Activation('softmax'))

model_balanced.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['accuracy'])

model_balanced.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_1 (Embedding)     (None, 150, 100)          4373200

 bidirectional_2 (Bidirectio  (None, 150, 86)          49536
 nal)

 bidirectional_3 (Bidirectio  (None, 86)               44720
 nal)

 dense_1 (Dense)             (None, 43)                3741

 activation_1 (Activation)   (None, 43)                0

=================================================================
Total params: 4,471,197
Trainable params: 4,471,197
Non-trainable params: 0
_________________________________________________________________
```

```
# Train the balanced network - Seems to take long time to achieve good accuracy?
history1 = model_balanced.fit(train_input, train_labels, batch_size=512, epochs=10, validation_data=(val_input, val_labels))
```

```
Epoch 1/10
274/274 [==============================] - 52s 165ms/step - loss: 2.3073 - accuracy: 0.4111 - val_loss: 1.9707 - val_accuracy: 0.4810
Epoch 2/10
274/274 [==============================] - 43s 158ms/step - loss: 1.8481 - accuracy: 0.5159 - val_loss: 1.8754 - val_accuracy: 0.5070
Epoch 3/10
274/274 [==============================] - 43s 158ms/step - loss: 1.7330 - accuracy: 0.5531 - val_loss: 1.8531 - val_accuracy: 0.5135
Epoch 4/10
274/274 [==============================] - 43s 158ms/step - loss: 1.6540 - accuracy: 0.5790 - val_loss: 1.8493 - val_accuracy: 0.5179
Epoch 5/10
274/274 [==============================] - 43s 158ms/step - loss: 1.5999 - accuracy: 0.5944 - val_loss: 1.8522 - val_accuracy: 0.5156
Epoch 6/10
274/274 [==============================] - 43s 157ms/step - loss: 1.5591 - accuracy: 0.6046 - val_loss: 1.8708 - val_accuracy: 0.5170
Epoch 7/10
274/274 [==============================] - 44s 159ms/step - loss: 1.5297 - accuracy: 0.6126 - val_loss: 1.9042 - val_accuracy: 0.5114
Epoch 8/10
274/274 [==============================] - 43s 158ms/step - loss: 1.5083 - accuracy: 0.6180 - val_loss: 1.9213 - val_accuracy: 0.5124
Epoch 9/10
274/274 [==============================] - 44s 159ms/step - loss: 1.4856 - accuracy: 0.6241 - val_loss: 1.9217 - val_accuracy: 0.5198
Epoch 10/10
274/274 [==============================] - 43s 157ms/step - loss: 1.4603 - accuracy: 0.6309 - val_loss: 1.9521 - val_accuracy: 0.5136
```




```
# Overall Accuracy
score = model_balanced.evaluate(test_sentences_X, y_test, batch_size=100)

560/560 [==============================] - 33s 59ms/step - loss: 1.8449 - accuracy: 0.5463

print("Overall Accuracy:", score[1]*100)

Overall Accuracy: 54.629528522491455
```

Model 3 : CNN and BLSTM

```
# concatenate tensors
concatenated_tensors = Concatenate()([maxpool_0, maxpool_1, maxpool_2])
# flatten concatenated tensors
flatten_concatenated_tensors = TimeDistributed(Flatten())(concatenated_tensors)
# dense layer (dense_1)
dense_1 = Dense(100, activation='relu')(flatten_concatenated_tensors)
# dropout_1
dropout_1 = Dropout(drop)(dense_1)
```

```
# BLSTM model

# Bidirectional 1
Bidirectional1 = Bidirectional(LSTM(100, return_sequences='true'))(dropout_1)
# Bidirectional 2
Bidirectional2 = Bidirectional(LSTM(100))(Bidirectional1)
# Dense layer (dense_2)
dense_2 = Dense(100, activation='relu')(Bidirectional2)
# dropout_2
dropout_2 = Dropout(drop)(dense_2)
```

```
# concatenate 2 final layers
flattened_dropout_1 = Flatten()(dropout_1)
concatenate_2_final_layer = Concatenate()([flattened_dropout_1,dropout_2])
# output
output_layer = Dense(HIDDEN_SIZE, activation='relu')(concatenate_2_final_layer)

model2 = Model(inputs=[inputs], outputs=[output_layer])

# Compile the model
model2.compile(loss='binary_crossentropy', optimizer='adam', metrics = ['accuracy'])
model2.summary()
```

```
Model: "model"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_1 (InputLayer)           [(None, 150)]        0           []

 embedding_2 (Embedding)        (None, 150, 100)     4373200     ['input_1[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]']
```

```
 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_2 (Dense)                (None, 1, 100)       19300       ['time_distributed[0][0]']

 dropout (Dropout)              (None, 1, 100)       0           ['dense_2[0][0]']

 bidirectional_4 (Bidirectional  (None, 1, 200)      160800      ['dropout[0][0]']
 )

 bidirectional_5 (Bidirectional  (None, 200)         240800      ['bidirectional_4[0][0]']
 )

 dense_3 (Dense)                (None, 100)          20100       ['bidirectional_5[0][0]']

 flatten_1 (Flatten)            (None, 100)          0           ['dropout[0][0]']

 dropout_1 (Dropout)            (None, 100)          0           ['dense_3[0][0]']

 concatenate_1 (Concatenate)    (None, 200)          0           ['flatten_1[0][0]',
                                                                  'dropout_1[0][0]']

 dense_4 (Dense)                (None, 43)           8643        ['concatenate_1[0][0]']

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

We can see that with **60%** accuracy that combining the CNN and BLSTM model together outperforms the other models used.

```
# Train the model - use validation
model2.fit(train_input, train_labels, batch_size=512, epochs=10, validation_data=(val_input,val_labels))
```

```
Epoch 1/10
274/274 [==============================] - 54s 146ms/step - loss: 0.1301 - accuracy: 0.4762 - val_loss: 0.0925 - val_accuracy: 0.5113
Epoch 2/10
274/274 [==============================] - 38s 138ms/step - loss: 0.0846 - accuracy: 0.5709 - val_loss: 0.0914 - val_accuracy: 0.4952
Epoch 3/10
274/274 [==============================] - 38s 138ms/step - loss: 0.0778 - accuracy: 0.5926 - val_loss: 0.0774 - val_accuracy: 0.5480
Epoch 4/10
274/274 [==============================] - 38s 138ms/step - loss: 0.0756 - accuracy: 0.6137 - val_loss: 0.0907 - val_accuracy: 0.5608
Epoch 5/10
274/274 [==============================] - 38s 137ms/step - loss: 0.0687 - accuracy: 0.6523 - val_loss: 0.0716 - val_accuracy: 0.6241
Epoch 6/10
274/274 [==============================] - 38s 138ms/step - loss: 0.0945 - accuracy: 0.5944 - val_loss: 0.0884 - val_accuracy: 0.5771
Epoch 7/10
274/274 [==============================] - 38s 138ms/step - loss: 0.0758 - accuracy: 0.6223 - val_loss: 0.0768 - val_accuracy: 0.5882
Epoch 8/10
274/274 [==============================] - 38s 137ms/step - loss: 0.0697 - accuracy: 0.6596 - val_loss: 0.0735 - val_accuracy: 0.6121
Epoch 9/10
274/274 [==============================] - 38s 137ms/step - loss: 0.0652 - accuracy: 0.6847 - val_loss: 0.0729 - val_accuracy: 0.6174
Epoch 10/10
274/274 [==============================] - 38s 137ms/step - loss: 0.0736 - accuracy: 0.6332 - val_loss: 0.0738 - val_accuracy: 0.5832
<keras.callbacks.History at 0x7fc40350d410>
```

```
score = model2.evaluate(test_sentences_X, y_test, batch_size=100)
```

```
560/560 [==============================] - 7s 12ms/step - loss: 0.0719 - accuracy: 0.6079
```

```
print("Overall Accuracy:", score[1]*100)
```

```
Overall Accuracy: 60.793888568878174
```

Part E - Dialogue 2: A Conversational Dialogue System

Task 1: Implementing the encoder

```
class Encoder(tf.keras.Model):
    def __init__(self, vocab_size, embedding_dim, enc_units):
        super(Encoder, self).__init__()
        self.batch_sz = batch_size
        self.enc_units = enc_units

        #  pass the embedding into a bidirectional version of the GRU - as you can see in the call() method below, you can use just 1 G
        self.embeddings = embeddings # Embedding layer
        self.dropout = Dropout(0.2) # Dropout Layer with 0.2 as dropout rate
        self.Inp = Input(shape=(max_len_q,)) # size of questions
        # Bidirectional GRU layer1 with 50 GRU units i.e 100 cells and return_sequences = True
        self.Bidirectional1 = Bidirectional(GRU(self.enc_units, return_state= False, return_sequences= True))
        # Bidirectional GRU layer2 with 50 GRU units i.e 100 cells and return_sequences = True and return_states = True
        self.Bidirectional2 = Bidirectional(GRU(self.enc_units, return_state= True, return_sequences= True))

    def bidirectional(self, bidir, layer, inp, hidden):
        return bidir(layer(inp, initial_state = hidden))

    def call(self, x, hidden):
        x = self.embeddings(x)
        x = self.dropout(x)
        x = self.Bidirectional1(x)
        x = self.dropout(x)
        output, state_f,state_b = self.Bidirectional2(x)

        return output, state_f, state_b

    def initialize_hidden_state(self):
        return tf.zeros((self.batch_sz, self.enc_units))
```

Two GRU layers are defined sonsecutively as mentioned. Second layer GRU's last hidden and cell state will be used to initialize the first GRU layer of decoder therefore return_state of second GRU layer is True.

Task 2: Implementing the decoder with attention

The first GRU layer of the Decoder is initialized by encoder's last hidden and cell state. New decoder's state for the next word generation at next time step is the concatenated last hidden and cell states. Attention layer is used to compute the attention weights which is a scaled dot product of all encoder hidden states and decoder's hidden state at current time step, to get the relevance of each input token for generating the word at current time step.

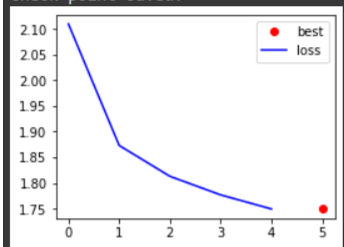
```python
class Decoder(tf.keras.Model):
    def __init__(self, vocab_size, embedding_dim, dec_units):
        super(Decoder, self).__init__()
        self.batch_sz = batch_size
        self.embeddings = embeddings
        self.units = 2 * dec_units # because we use bidirectional encoder
        self.fc = Dense(vocab_len, activation='softmax', name='dense_layer')
        # Create the decoder with attention - as you'll see in the call() method below, it will need two GRU layers
        self.dropout = Dropout(0.2) # Dropout Layer with 0.2 as dropout rate
        self.attention = BahdanauAttention(self.units) # BahdanauAttention Layer with 100 units

        # GRU Layer 1 with 100 GRU units and return_sequences = True
        self.decoder_gru_l1 = GRU(self.units, return_sequences=True, return_state=False)
        # GRU layer 2 with 100 GRU units and return_sequences = True and return_state = True
        self.decoder_gru_l2 = GRU(self.units, return_sequences=False, return_state=True)



    def call(self, x, hidden, enc_output):

        # enc_output shape == (batch_size, max_length, hidden_size)
        context_vector, attention_weights = self.attention(hidden, enc_output)

        # x shape after passing through embedding == (batch_size, 1, embedding_dim)
        x = self.embeddings(x)

        # x shape after concatenation == (batch_size, 1, embedding_dim + hidden_size)
        x = tf.concat([tf.expand_dims(context_vector, 1), x], axis=-1) # concat input and context vector together

        # passing the concatenated vector to the GRU
        x = self.decoder_gru_l1(x)
        x = self.dropout(x)
        output, state = self.decoder_gru_l2(x)
        x = self.fc(output)
        return x, state, attention_weights
```

Task 3: Investigating the behaviour and the properties of the encoder

- Look at the attention weights and compare them after 5, 50 and 140 epochs:-

  5th Epoch-

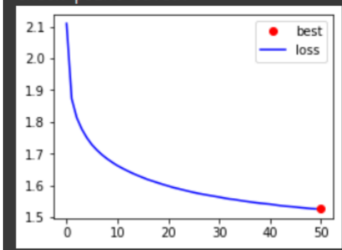
```
Epoch 5 Batch 598 Loss: 1.4660
Epoch 5 Batch 1196 Loss: 2.1242
Epoch 5 Batch 1794 Loss: 1.9338
Epoch 5 Batch 2392 Loss: 2.0549
Epoch 5 Batch 2990 Loss: 1.8285
Epoch 5 Batch 3588 Loss: 2.1910

*** Epoch 5 Loss 1.7494 ***

####################
Greedy| Q: Hello   A: hi
%
Greedy| Q: How are you ?  A: i am not a good time
%
Greedy| Q: What are you doing ?  A: i am not a good time
%
Greedy| Q: What is your favorite restaurant ?  A: i am sorry
%
Greedy| Q: Do you want to go out ?  A: i am not
####################
check point saved!
```




```
Best epoch so far:  5
Time  157.532 sec

========================================
```

50th Epoch-

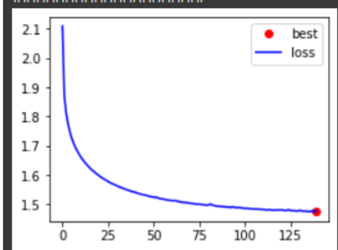
```
Epoch 50 Batch 598 Loss: 1.4675
Epoch 50 Batch 1196 Loss: 1.8088
Epoch 50 Batch 1794 Loss: 1.8868
Epoch 50 Batch 2392 Loss: 1.9947
Epoch 50 Batch 2990 Loss: 1.4675
Epoch 50 Batch 3588 Loss: 1.8246

*** Epoch 50 Loss 1.5257 ***

####################
Greedy| Q: Hello   A: hi
%
Greedy| Q: How are you ?  A: i am not like a lot of things
%
Greedy| Q: What are you doing ?  A: i am not
%
Greedy| Q: What is your favorite restaurant ?  A: i am not
%
Greedy| Q: Do you want to go out ?  A: i am not
####################
check point saved!
```




```
Best epoch so far:  50
Time  158.085 sec

========================================
```

140th Epoch-

```
Epoch 140 Batch 598 Loss: 1.4759
Epoch 140 Batch 1196 Loss: 1.7359
Epoch 140 Batch 1794 Loss: 1.6911
Epoch 140 Batch 2392 Loss: 1.9015
Epoch 140 Batch 2990 Loss: 1.3107
Epoch 140 Batch 3588 Loss: 1.6248

*** Epoch 140 Loss 1.4769 ***

####################
Greedy| Q: Hello   A: hello
%
Greedy| Q: How are you ?  A: i am not like a delightful item
%
Greedy| Q: What are you doing ?  A: i am not
%
Greedy| Q: What is your favorite restaurant ?  A: i am fine
%
Greedy| Q: Do you want to go out ?  A: i am not
####################
```




```
Best epoch so far:  139
Time  160.581 sec

========================================
```

- Did the models learn to track local relations between words?

  Ans. With every epoch, the model is learning to track the local relations between the words. From the results we can observe that the loss is receding and the answers to the questions considerably improves with each epoch. For instance, for the question *What is your favourite restaurant?* the answer in the epoch 66 is *i am not*. However, by the 147th epoch the model is performing much better compared to previous epoch answers by answering *i do not know* and thus the responses are much improved.

- 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?

Ans. In the starting we can see that with epoch same answers are being generated with respect to different questions. For instance, the answer *i am not* is being answered frequently indicating the case of overfitting. However, as epochs increases the model learns to respond with different answers by being able to differentiate between different questions and answer accordingly.

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

  Ans. Model is learning to track some major syntactic relations in the utterances with each epochs. From the results we can see that for the question *How are you* the model answered *i am not* in the 7th epoch and later in the 26th epoch *i am not like a lot of things*.From this we can see that, the model is slowly learning some major syntactic relations in utterances as the epochs increases.

```
*** Epoch 7 Loss 1.7104 ***

####################
Greedy| Q: Hello   A: hello
%
Greedy| Q: How are you ?  A: i am not
%
Greedy| Q: What are you doing ?  A: i am not
%
Greedy| Q: What is your favorite restaurant ?  A: i am not
%
Greedy| Q: Do you want to go out ?  A: i am not
####################
check point saved!
Best epoch so far:  7
Time  157.601 sec

========================================
```

```
*** Epoch 26 Loss 1.5776 ***

####################
Greedy| Q: Hello   A: hello
%
Greedy| Q: How are you ?  A: i am not like a lot of things
%
Greedy| Q: What are you doing ?  A: i am not going to be a lot of things
%
Greedy| Q: What is your favorite restaurant ?  A: i am not
%
Greedy| Q: Do you want to go out ?  A: i am not
####################
check point saved!
Best epoch so far:  26
Time  156.449 sec

========================================
```

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

  Ans. Yes, with the increase in epochs, the model gets better in understanding the POS tags too therefore generating meaningful sentence. For ex, in the 14th epoch the model responded with *i am not going to be a lot of things* to the question *How are you ?* and responds with *i am not going to be a peach farmer* in the 76th epoch to the same question. Even though the both the responses are syntatically and grammatically incorrect, the model is nonetheless leaning to encode the linguistic features.

```
*** Epoch 14 Loss 1.6377 ***

####################
Greedy| Q: Hello   A: hello
%
Greedy| Q: How are you ?  A: i am not going to be a lot of things
%
Greedy| Q: What are you doing ?  A: i am not going to be a lot of other peoples
%
Greedy| Q: What is your favorite restaurant ?  A: i am not going to be a lot of the way
%
Greedy| Q: Do you want to go out ?  A: i am not
####################
check point saved!
Best epoch so far:  14
Time  157.445 sec

========================================
```

```
*** Epoch 76 Loss 1.5001 ***

####################
Greedy| Q: Hello   A: hi
%
Greedy| Q: How are you ?  A: i am not going to be a peach farmer
%
Greedy| Q: What are you doing ?  A: i am not
%
Greedy| Q: What is your favorite restaurant ?  A: i am not
%
Greedy| Q: Do you want to go out ?  A: i am sorry
####################
check point saved!
Best epoch so far:  76
Time  157.397 sec

========================================
```

- What is the effect of more training on the length of response?

  Ans. By training more, it is observed that the model learns better. More training on the length of the response will increase the prediction accuracy and the loss will be minimised. The length increase in the responses is not observed all the time. With more training time it cannot be said that the response length will be long or short, but we can say that the responses will be more specific and relevant towards the questions asked. Greedy Decoding will also gets improved, which ultimately gives better results.

- 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?

  Ans. In order to overcome the problem of forgetting the most relevant early query tokens, we can use attention mechanism for the said issue. The issue arises mostly because of vanishing gradient problem. Another way we can approach this issue is by replacing the GRU with LSTM architecture. GRU has two gates reset and update whereas LSTM has three i.e. input, output and forget. LSTM is more accurate and efficient at retaining the information in longer sentences. We can also implement early stopping to save computational time.