ECS7001 NN & NLP

Assignment 2: Neural Machine Translation and

Neural Dialogue Systems

Part A: Neural Machine Translation [50 marks]

1. Task 1: Implementing the encoder [15 marks].

```
# Task 1 encoder
# Start
embedding_source = Embedding(input_dim=self.vocab_source_size,
                             embeddings_initializer='random_uniform',
                             mask_zero=True,
                             output_dim=self.embedding_size,
                             input_length=source_words.shape[1])
source_words_embeddings = embedding_source(source_words)
source_words_embeddings = Dropout(self.embedding_dropout_rate)(source_words_embeddings)
encoder_lstm = LSTM(self.hidden_size,
                    recurrent_dropout=self.hidden_dropout_rate,
                    return_sequences=True,
                    return_state=True)
encoder_outputs,encoder_state_h,encoder_state_c = encoder_lstm(source_words_embeddings)

embedding_target = Embedding(input_dim=self.vocab_target_size,
                             embeddings_initializer='random_uniform',
                             mask_zero=True,
                             output_dim=self.embedding_size,
                             input_length=target_words.shape[1])
target_words_embeddings = embedding_target(target_words)
target_words_embeddings = Dropout(self.embedding_dropout_rate)(target_words_embeddings)
# End Task 1
```

In Task 1 we create a simple encoder model and decoder input layers, encoder model consists of an embedding, a dropout and a LSTM layer. The embedding layer, which converts the simple english input sentences to an embedded list of word arrays. This layer is passed through a dropout layer and finally to an LSTM layer. The significance of the dropout layer is to block partially some of the input during training. The LSTM (long short term memory) layer is an RNN layer used to pass the input sentence (sequence) in a number of consecutive time-steps. And in each time-steps the model learns from the word input to the model for that time-step. This information is stored in a hidden vector. This hidden vector also preserve the information from previous time steps and is passed on to Decoder. In below figure 2, the output of the encoder is the output of lstm_1: LSTM layer and is passed to Decoder layer lstm_2: LSTM.

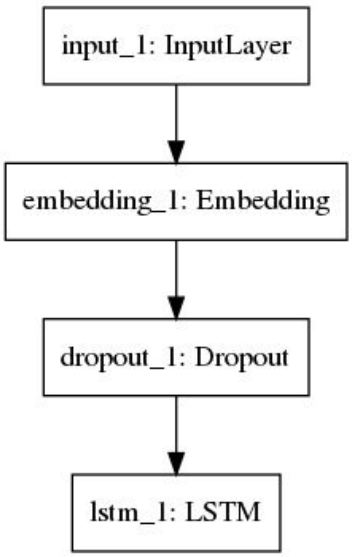

Figure 1: Encoder Model, Task 1

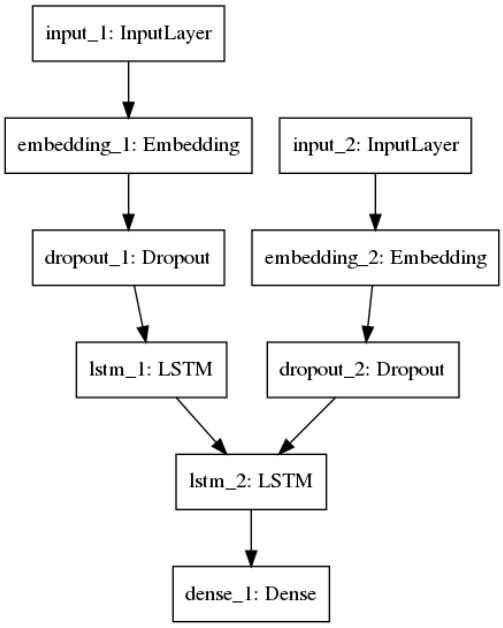

Figure 2: Training Model

2. Task 2: Implementing the decoder [15 marks].

```
decoder_state = [decoder_state_input_h, decoder_state_input_c]
decoder_outputs_test,decoder_state_output_h,decoder_state_output_c =
decoder_lstm(target_words_embeddings,initial_state=decoder_state)
if self.use_attention:
        decoder_attention = AttentionLayer()
        decoder_outputs_test = decoder_attention([encoder_outputs_input,decoder_outputs_test])

decoder_outputs_test = decoder_dense(decoder_outputs_test)
```

In task 2, we implemented the decoder for inference. The decoder for training can be seen in figure 2, input_2 → embedding_2 → dropout_2 → lstm_2 → dense_1, with additional input from lstm_1 of encoder to lstm_2 . The decoder for inference can be seen in figure 3, which has input from target word embeddings, instead of hidden vector inputs of encoder input_2 and input_3 as decoder_state_input_h and decoder_state_input_c is provided. The difference here is the inference model performs one step of decoding unlike decoder for training, for which hidden vector (of encoder) is essential.

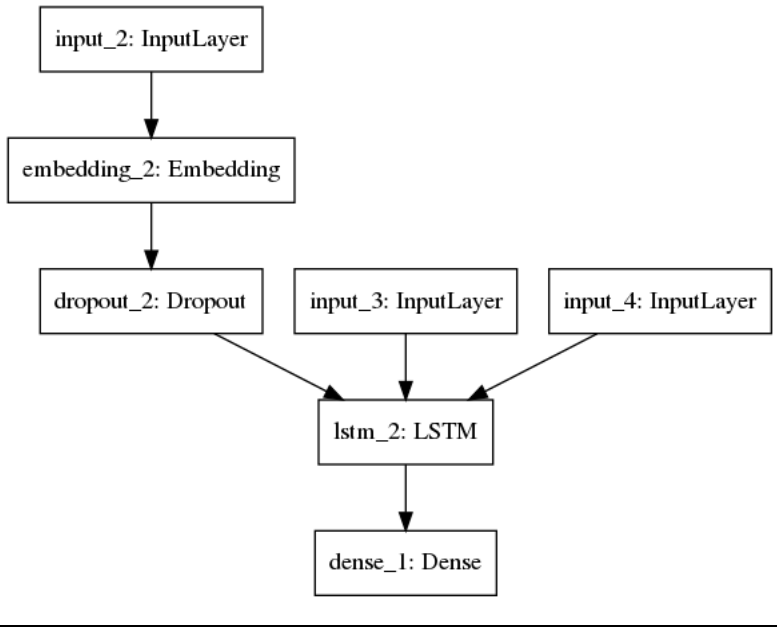

Figure 3: Decoder Inference Model

The BELU (Bilingual Evaluation Understudy) score is given by

$$\text{BLEU} = \min\left(1, \frac{\textit{output-length}}{\textit{reference-length}}\right) \left(\prod_{i=1}^{4} \textit{precision}_i\right)^{\frac{1}{4}}$$

It is a score for comparing a candidate translation of text to one or more reference translations. It takes into consideration n-gram factors of the referencing translation with the candidate translation. The precision here is the n-gram precision which is the count of matching n-grams in the occurrence of words in reference text with respect to the candidate sentence.

We see in the results that upon reaching the 10th epoch the BELU score has managed to achieve 6.39 score, with accuracy of 39.73%.

```
Starting training epoch 10/10
Epoch 1/1
24000/24000 [==============================] - 103s 4ms/step - loss: 1.4049 - accuracy: 0.3973
Time used for epoch 10: 1 m 43 s
Evaluating on dev set after epoch 10/10:
Model BLEU score: 5.98
Time used for evaluate on dev set: 0 m 5 s
Training finished!
Time used for training: 18 m 33 s
Evaluating on test set:
Model BLEU score: 6.39
Time used for evaluate on test set: 0 m 5 s
```

Evaluating on dev set after epoch **1/10**:

candidates: and i 'm going to <unk> , and i 'm going to <unk> , and i 'm going to <unk> the <unk> .

references: there are four <unk> <unk> that , each time this ring <unk> it , as it <unk> the <unk> of the display , it <unk> up a position signal .

```
Model BLEU score: 1.41
===
Evaluating on dev set after epoch 10/10:
candidates:  there &apos;s a <unk> <unk> , and it &apos;s <unk> , and it &apos;s a <unk> <unk>
.
references:  there are four <unk> <unk> that , each time this ring <unk> it , as it <unk> the
<unk> of the display , it <unk> up a position signal .
Model BLEU score: 5.98
==
Evaluating on test set:
candidates:  and then , it &apos;s <unk> , and it &apos;s <unk> .
references:  then <unk> can be made and <unk> .
Model BLEU score: 6.39
==
```

It can be seen that in 1/10 epoch on dev set the BELU score was very low around 1.41, and in epoch was 5.98 at epoch 10/10, the sentences, candidates at epoch 1 and 10 have relatively improved after the learning over 10 epoch. Also there is an increase of 15% in accuracy. This suggests that encoder and decoder models are playing their individual roles but the BELU scores can be increased if there is some architectural changes to the model such as adding an attention layer.

3. Adding attention [20 marks].

Again, you must include the code, an explanation, the BLEU score, and a sample of the output

```
luong_score = tf.matmul(decoder_outputs, encoder_outputs, transpose_b=True)
alignment = tf.nn.softmax(luong_score, axis=2)
context = tf.matmul(K.expand_dims(alignment,axis=2), K.expand_dims(encoder_outputs,axis=1))
encoder_vector = K.squeeze(context,axis=2)
```

In this task we add an attention layer to the Encoder-decoder model as seen in Figure 4, more specifically we write the loung attention layer. Attention is an interface between the encoder and decoder that provides the decoder with information from every encoder hidden state. As seen in figure, Loung et al. attention uses hidden states at the top LSTM layers in both the encoder and decoder. The Luong attention mechanism uses the current decoder's hidden and encoder state to compute the alignment vector which is the softmax of both the products. The context is generated from the product of alignment and the encoder. Out of the three, additive/concat, dot product, location-based, and 'general', here are using additives as discussed in GNMT paper. This operation is performed in the last line above.

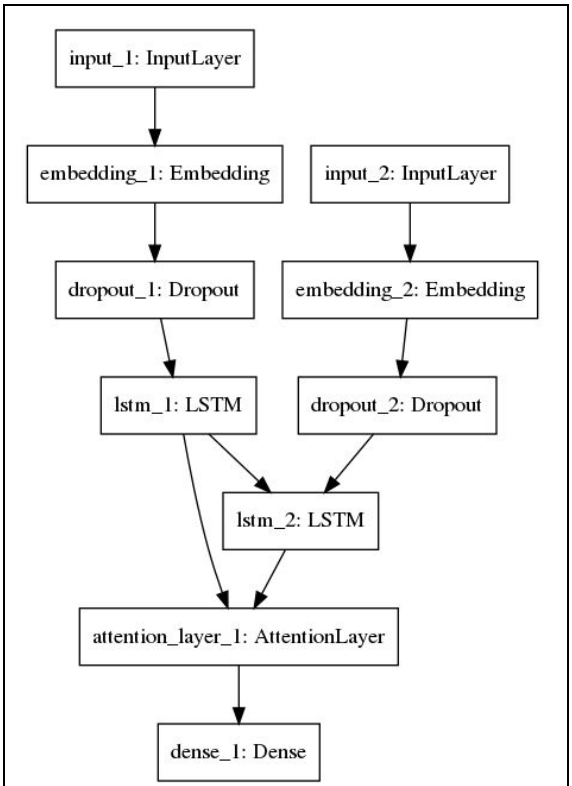

Figure 4: Adding attention

Below we can see, the BLEU score has improved considerably from 6.39 to 15.57 on the test set, and so has the accuracy from 39.73% to 55.21%. Thus, we can understand that after adding attention, we are able to better understand the context, generate alignment to improve our performance score.

```
Starting training epoch 10/10
Epoch 1/1
24000/24000 [==============================] - 20s 836us/step - loss: 0.9116 -
accuracy: 0.5521
Time used for epoch 10: 0 m 20 s
Evaluating on dev set after epoch 10/10:
Model BLEU score: 15.09
Time used for evaluate on dev set: 0 m 3 s
Training finished!
Time used for training: 4 m 9 s
Evaluating on test set:
Model BLEU score: 15.57
Time used for evaluate on test set: 0 m 4 s

Evaluating on dev set after epoch 1/10:
candidates:  and it &apos;s a <unk> , and it &apos;s <unk> , and it &apos;s <unk> .
references:  and here you can see it &apos;s <unk> about the <unk> axis only , creating <unk>
.
Model BLEU score: 3.57

======
Evaluating on dev set after epoch 10/10:
candidates:  and this can be <unk> that it &apos;s just <unk> around the <unk> .
references:  and here you can see it &apos;s <unk> about the <unk> axis only , creating <unk>
```

```
.
Model BLEU score: 14.97
======
Evaluating on test set:
candidates:  then , <unk> can be done with <unk> .
references:  then <unk> can be made and <unk> .
Model BLEU score: 15.60
```

From above we can see that the candidate and reference n-grams phrases have more similarity than the model which did not have attention. THe BLEU score have risen considerably with as good approximations are seen from attention model to generate better candidates.

Part B: Dialogue

1. Task 1: Implementing an utterance-based tagger, using standard text classification methods from lectures [20 marks].

```
model = Sequential()
model.add(Embedding(VOCAB_SIZE,EMBED_SIZE,input_length=MAX_LENGTH))
model.add(Bidirectional(LSTM(HIDDEN_SIZE,return_sequences=True,recurrent_dropout=0.7)))
model.add(Bidirectional(LSTM(HIDDEN_SIZE,recurrent_dropout=0.7)))
model.add(Dense(HIDDEN_SIZE))
model.add(Activation('sigmoid'))
```

	Model without added weights	Model with added weights
Accuracy	`print("Overall Accuracy:", score[1]*100)` `Overall Accuracy: 67.43944883346558`	`print("Overall Accuracy:", score[1]*100)` `Overall Accuracy: 53.31651568412781`
Majority and Minority Class Accuracies	`br 0.0` `bf 0.0`	`br 0.4430379746835443` `bf 0.12807881773399016`

The classifier has high overall accuracy of 67.4% and for majority class it scores around 89% when trained without any additional weights, but gives a poor classification score for minority classes such as “br” and “bf”. The reason being there is less training data for minority classes, therefore the classifier fails to identify and optimise those classes. When we apply class weighting, the overall and majority class accuracies decreases to around 53% but we see a little increase in minority class accuracy. This can be attributed as we are giving more weights to samples of minority class than majority class.

	row	text	true_y	unbalanced_pred_y	balanced_pred_y
0	51507	it's just part of a day's work. /	fo_o_fw_"_by_bc	+	+
1	5759	Yes, /	fo_o_fw_"_by_bc	ny	sd
2	22602	Okay, /	fo_o_fw_"_by_bc	bk	bk
3	6920	# {F Oh, } okay. # /	bk	b	bk
4	27370	{F Oh, } I see. /	bk	b	bk
5	35288	Okay, /	bk	fo_o_fw_"_by_bc	fo_o_fw_"_by_bc

From the above table we can see some of the errors. In case of unbalanced class prediction, classes such as “fo_o_fw_"_by_bc” and “bk”, have nearly zero correct predictions. When we add class_weights to the same model, we see there is a slight improvement and this is evident in the above table. Although row 0 and row 2 have been correctly been identified but others are still mis-classified.

*The above table is generated by randomly picking a few instances of test data.

2. Task 2: Implementing a hierarchical utterance+DA-context-based tagger [20 marks].

CNN part code snippet

```
# concatenate tensors
concatenate_tensors = concatenate([maxpool_0, maxpool_1,maxpool_2],axis=-1)
# flatten concatenated tensors
flatten_concat = Flatten()(concatenate_tensors) # dim 2
# dense layer (dense_1)
dense_1 = Dense(HIDDEN_SIZE)(flatten_concat) # dim 2
# dropout_1
dropout_1 = Dropout(drop)(dense_1) # dim 2
```

BLSTM part code

```
# BLSTM model
# time_xx = TimeDistributed(Flatten())(concatenate_tensors)
time_xx = TimeDistributed(Dense(HIDDEN_SIZE))(concatenate_tensors)

_CNN_to_LSTM =
Reshape((time_xx.shape[1]*time_xx.shape[2],int(time_xx.shape[3])),input_shape=time_xx.shape[1:
4])(time_xx)
# https://github.com/keras-team/keras/issues/11425
# _CNN_to_LSTM = Dense(units=HIDDEN_SIZE,input_shape=(1,),activation='relu')(time_xx)
# _CNN_to_LSTM = Dropout(drop)(_CNN_to_LSTM)

# Bidirectional 1
b1 = Bidirectional(LSTM(HIDDEN_SIZE, return_sequences=True))(_CNN_to_LSTM)
# Bidirectional 2
b2 = Bidirectional(LSTM(HIDDEN_SIZE))(b1)
# Dense layer (dense_2)
dense_2 = Dense(HIDDEN_SIZE)(b2)
# dropout_2
dropout_2 = Dropout(drop)(dense_2)
```

Output layers

```
# concatenate 2 final layers
y = concatenate([dropout_1, dropout_2],axis = -1)
# output
out = Dense(HIDDEN_SIZE)(y)
out = Activation('sigmoid')(out)
m = Model(inputs,out)
m.compile(loss='categorical_crossentropy',optimizer='sgd',metrics=['accuracy'], )
m.summary()
```

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

```
Overall Accuracy: 69.4966197013855
```

```
bf 0.3968253968253968
br 0.0755813953488372
```

	row	text	true_y	unbalanced_pred_y	balanced_pred_y	CNN+LSTM_pred_y
0	51507	it's just part of a day's work. /	fo_o_fw_"_by_bc	+	+	fo_o_fw_"_by_bc
1	5759	Yes, /	fo_o_fw_"_by_bc	ny	sd	sd
2	22602	Okay, /	fo_o_fw_"_by_bc	bk	fo_o_fw_"_by_bc	fo_o_fw_"_by_bc
3	6920	# {F Oh, } okay. # /	bk	b	bk	bk
4	27370	{F Oh, } I see. /	bk	b	bk	bk
5	35288	Okay, /	bk	fo_o_fw_"_by_bc	fo_o_fw_"_by_bc	bk

The overall accuracy is around 70% with minority class bf 0.5% and br 16%.

From above table we can see that columns true_y and CNN+LSTM_pred_y are almost same except 1 mis-classification error. Most mis classification are seen with classes such as 'ar' and other classes which are not at all identified. But to an extent the above model is slighly better than balanced and unbalanced models as it provides certain degree of reasonable results.

From above table we can also see that row 5 "okay" has tre class "bk" which was identified as "fo_o_fw_"_by_bc " by both weighted and unweighted models but after adding context the class was correctly identified as "bk" here in this model.

3. Task 3 Bert-Based Model for Dialogue Act Tagging [10 marks]

Please see the code file for model and explanation.