ECS 7001 - NN & NLP

Assignment 2: Neural Machine Translation and Neural Dialogue Systems

Spyridon Roumpis

181004877

Part A: Neural Machine Translation

Task 1: Implementing the encoder

```
Task 1 encoder

Start
"""
#create two Embedding layers
embedding_source = Embedding(input_dim = self.vocab_source_size,output_dim = self.embedding_size,
                             mask_zero=True)

embedding_target = Embedding(input_dim = self.vocab_target_size,output_dim = self.embedding_size,
                             mask_zero=True)

#passing the inputs through the Embedding layers
source_words_embeddings = embedding_source(source_words)
target_words_embeddings = embedding_target(target_words)

#apply dropout to the embeddings
source_words_embeddings = Dropout(self.embedding_dropout_rate)(source_words_embeddings)
target_words_embeddings = Dropout(self.embedding_dropout_rate)(target_words_embeddings)

#create a LSTM layer to process the source_words_embeddings
encoder_outputs,encoder_state_h,encoder_state_c = LSTM(self.hidden_size,recurrent_dropout=self.hidden_dropout_rate,
                                                       return_sequences=True,return_state=True)(source_words_embeddings)

"""
End Task 1
"""
```

The encoder implementation was developed for this task. At first two Embedding layers, one for the source language and one for the target language were created. The Embedding layers have an input_dim of the vocab_size and an output_dim of the embedding_size for each language (source, target). Then the inputs for each language are passed through the Embedding Layers and a dropout rate is applied to both of them. Finally, a LSTM layer is created to process the source_words_embeddings.

Task 2: Implementing the decoder

```
"""
Task 2 decoder for inference

Start
"""
#create a list with the decoder_states
decoder_states_inputs = [decoder_state_input_h, decoder_state_input_c]

# pass the target_word_embeddings and decoder_states to the decoder_lstm
decoder_outputs,decoder_state_output_h,decoder_state_output_c = decoder_lstm(target_words_embeddings,initial_state=decoder_states_inputs)

#attention model
if self.use_attention:
  decoder_attention = AttentionLayer()
  #decoder_outputs_test = decoder_attention([decoder_outputs_train,decoder_outputs_test])
  decoder_outputs = decoder_attention([encoder_outputs_input,decoder_outputs])

#pass the output into the final layer of the decoder (decoder_dense)
decoder_outputs_test = decoder_dense(decoder_outputs)


"""
End Task 2
"""
```

A decoder for inference was implemented for this task. This decoder(for inference) is quite similar to the decoder for training but it only performs one step of the decoding. The three different layers created in the decoder for training will be used here.
The initial_states for this decoder are the states that are passed to the decoder model. These two states are stored on a list and they are my decoder states. Then the decoder states and the target word embeddings from the previous task will pass to the lstm decoder. An if statement for the attention model is also implemented based on the decoder for training.
Finally, the output of the LSTM or for the attention model will pass into the final layer of the decoder to assign probabilities of the next tokens.

The BLUE Score after training and evaluating the model for 10 epochs is 4.90. Also below it can be seen a sample of the output for the training and the validation.

```
Epoch 1/1
24000/24000 [==============================] - 212s 9ms/step - loss: 1.4659 - accuracy: 0.3803
Time used for epoch 10: 3 m 31 s
Evaluating on dev set after epoch 10/10:
candidates: ['and', 'then', 'finally', ',', 'the', '<unk>', '<unk>', 'is', '<unk>', '.']
references: [['and', 'then', ',', 'as', 'the', 'other', 'axis', '<unk>', 'in', ',', 'those', 'actually', '<unk>', 'into', 'a', '<unk>', '.']]
Model BLEU score: 4.53
Time used for evaluate on dev set: 0 m 11 s
Training finished!
Time used for training: 37 m 5 s
Evaluating on test set:
candidates: ['and', 'yet', ',', 'there', 'are', 'many', 'things', 'that', 'are', '<unk>', '.']
references: [['despite', 'the', 'success', ',', 'physics', 'has', 'its', '<unk>', '.']]
Model BLEU score: 4.90
Time used for evaluate on test set: 0 m 11 s
```

Task 3: Adding attention

```
Task 3 attention

Start
"""
# transpose the last two dimensions
decoder_out = K.permute_dimensions(decoder_outputs,(0, 2, 1))

#do the multiplications
luong_score = K.batch_dot(encoder_outputs, decoder_out)

# apply a softmax
luong_score = K.softmax(luong_score,axis=1)

# expand dimensions
luong_score = K.expand_dims(luong_score,axis= 3)
encoder_outputs = K.expand_dims(encoder_outputs,axis= 2)

#multiply the two tensors and sum max_source_sent_len dimension to create the encoder_vector
encoder_vector = multiply([encoder_outputs,luong_score])
#encoder_vector = np.dot(encoder_outputs,luong_score)
encoder_vector = K.sum(encoder_vector,axis=1)


"""
End Task 3
```

The attention decoder enables the decoder to access all encoder outputs and focus on different parts of the encoder outputs during different steps.

First, we need to transpose the last two dimensions of the decoder_outputs so a multiplication between the encoder and the decoder outputs can happen. Then a softmax has to be applied to the dimension of the result's multiplication to create an attention score for the outputs of the encoder. Finally, the encoder vector has to be created by doing element-wise multiplication, this time between the encoder outputs and their attention scores, which was calculated before. But the shape of them is not the same so we need to expand the dimensions for both of them. For the attention score an expand to the last dimension to accommodate the hidden_size dimension of the encoder_outputs is needed, the shape becomes [batch_size, max_source_sent_len, max_target_sent_len, 1] . For encoder_outputs the target shape is [batch_size, max_source_sent_len, 1, hidden_size] . Now that the two tensors have the same shape we can multiply them and then sum the max_source_sent_len dimension to create the encoder vector.

The BLUE Score after adding attention to the model, training, and evaluating the model for 10 epochs is almost three times better than before, 15.64. Also below it can be seen a sample of the output for epoch = 10.

```
Epoch 1/1
24000/24000 [==============================] - 280s 12ms/step - loss: 0.9192 - accuracy: 0.5500
Time used for epoch 10: 4 m 40 s
Evaluating on dev set after epoch 10/10:
candidates: ['now', 'let', '&apos;s', 'see', 'the', 'same', 'way', '.']
references: [['so', ',', 'take', 'a', 'look', '.']]
Model BLEU score: 15.27
Time used for evaluate on dev set: 0 m 15 s
Training finished!
Time used for training: 48 m 34 s
Evaluating on test set:
candidates: ['you', 'put', 'it', 'into', 'the', '<unk>', '.']
references: [['you', '<unk>', 'it', 'into', '<unk>', '.']]
Model BLEU score: 15.64
Time used for evaluate on test set: 0 m 15 s
```

Part B: Dialogue

Task 1

```
#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(input_dim=VOCAB_SIZE,output_dim=EMBED_SIZE,input_length=MAX_LENGTH))

# Bidirectional 1
model.add(Bidirectional(LSTM(HIDDEN_SIZE, return_sequences=True)))

# Bidirectional 2
model.add(Bidirectional(LSTM(HIDDEN_SIZE, return_sequences=False)))

# Dense layer
model.add(Dense(HIDDEN_SIZE))

# Activation
model.add(Activation('softmax'))

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

model.summary()
```

The code for our model is given from the above screenshot. To train the model we used also a validation set, 3 epochs, and a batch_size of 1000.

```
model.fit(train_input,train_labels,validation_data=(val_input,val_labels), epochs = 3,batch_size=1000, verbose = 1)
```

To evaluate it the below command was used and gave an overall accuracy of 67.99

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

```
Overall Accuracy: 67.9993562815906
```

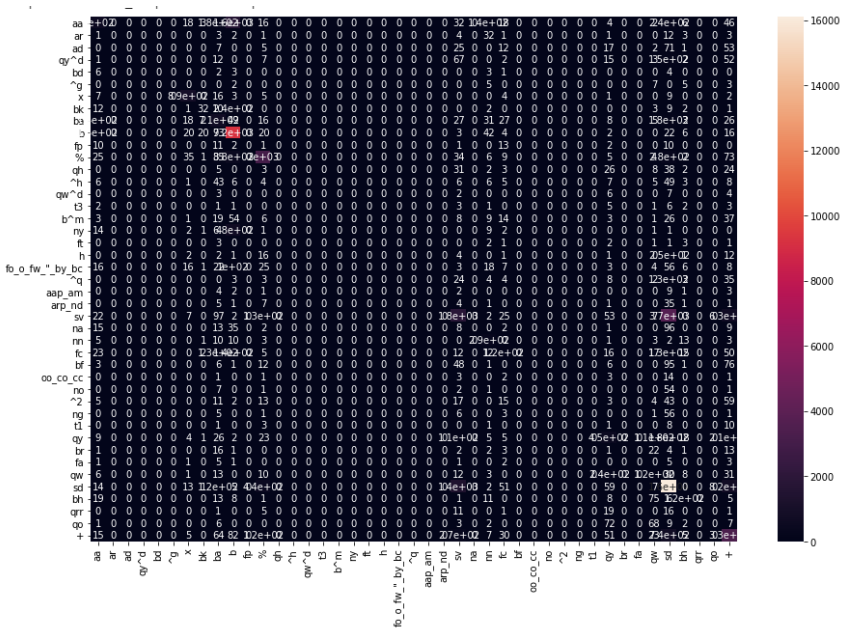
Then we generated predictions for the test data to look at the accuracy of some minority classes. Signal-non-understanding ('br') and summarize/reformulate ('bf') . A confusion matrix and some other metrics were also calculated to help analyze further those classes

```
[ ] # Generate predictions for the test data
    y_pred = model.predict(test_sentences_X,batch_size=100)
```

```
[ ] from sklearn.metrics import confusion_matrix

    confusion_matrix(np.argmax(y_test, axis=1), np.argmax(y_pred, axis=1))
```

```
array([[ 504,    0,    0, ...,    0,    0,   46],
       [   1,    0,    0, ...,    0,    0,    3],
       [   0,    0,    0, ...,    0,    0,   53],
       ...,
       [   0,    0,    0, ...,    0,    0,    1],
       [   1,    0,    0, ...,    0,    0,    7],
       [  15,    0,    0, ...,    0,    0, 3286]])
```



	precision	recall	f1-score	support
aa	0.48	0.18	0.26	2794
ar	0.00	0.00	0.00	63
ad	0.00	0.00	0.00	193
qy^d	0.00	0.00	0.00	311
bd	0.00	0.00	0.00	19
^g	0.00	0.00	0.00	22
x	0.86	0.95	0.90	941
bk	0.49	0.10	0.17	313
ba	0.40	0.57	0.47	1248
b	0.73	0.96	0.83	9634
fp	0.00	0.00	0.00	50
%	0.77	0.76	0.77	3981
qh	0.00	0.00	0.00	142
^h	0.00	0.00	0.00	149
qw^d	0.00	0.00	0.00	22
t3	0.00	0.00	0.00	25
b^m	0.00	0.00	0.00	181
ny	0.00	0.00	0.00	720
ft	0.00	0.00	0.00	14
h	0.00	0.00	0.00	291
fb_o_fw_"_by_bc	0.00	0.00	0.00	287
^q	0.00	0.00	0.00	212
aap_am	0.00	0.00	0.00	22
arp_nd	0.00	0.00	0.00	56
sv	0.46	0.28	0.35	6524
na	0.00	0.00	0.00	181
nn	0.45	0.85	0.59	345
fc	0.30	0.17	0.22	669
bf	0.00	0.00	0.00	249
oo_co_cc	0.00	0.00	0.00	25
no	0.00	0.00	0.00	66
^2	0.00	0.00	0.00	172
ng	0.00	0.00	0.00	73
t1	0.00	0.00	0.00	24
qy	0.41	0.39	0.40	1148
br	0.00	0.00	0.00	66
fa	0.00	0.00	0.00	19
qw	0.24	0.26	0.25	478
sd	0.70	0.85	0.77	18988
bh	0.51	0.45	0.47	265
qrr	0.00	0.00	0.00	54
qo	0.00	0.00	0.00	170
+	0.58	0.70	0.64	4696

The accuracy for both ‘br’ and ‘bf’ is zero.

```
[ ] # Argmax value of "br" as index
    br_index = np.argmax(one_hot_encoding_dic["br"])

    # Accuracy, using the index
    br_acc = confusion_mat[br_index][br_index] / sum(confusion_mat[br_index])
    print('"br" accuracy: ' + str(br_acc*100))

    # Argmax value of "bf"
    bf_index = np.argmax(one_hot_encoding_dic["bf"])

    # Accuracy, using the index
    bf_acc = confusion_mat[bf_index][bf_index] / sum(confusion_mat[bf_index])
    print('"bf" accuracy: ' + str(bf_acc*100))
```

```
"br" accuracy: 0.0
"bf" accuracy: 0.0
```

The main reason for this is because our dataset is highly imbalanced which means that the classes are not represented equally. Looking at the precision and the recall we see that our model is doing badly. We need to have high scores for both precision and recall so the model can return accurate results (precision), as well as returning a majority of all positive results (recall). An ideal system with high precision and high recall will return many results, with all results labeled correctly.

Due to the reduced lack of training data for the minority classes, these minority classifiers will not be very confident in classification, as they have not been fully optimized. The frequent classifiers will be more optimized and will generate more confident scores for all examples, effectively crowding out the less confident minority classifiers.

Balanced Model

One thing we can do to try to improve performance is, therefore, to balance the data more sensibly. As the dataset is highly imbalanced, we can simply weight up the minority classes proportionally to their underrepresentation while training.

```
# Re-built the model for the balanced training

model_balanced = Sequential()
model_balanced.add(Embedding(input_dim=VOCAB_SIZE,output_dim=EMBED_SIZE,input_length=MAX_LENGTH))
model_balanced.add(Bidirectional(LSTM(HIDDEN_SIZE, return_sequences=True)))
model_balanced.add(Bidirectional(LSTM(HIDDEN_SIZE, return_sequences=False)))
model_balanced.add(Dense(HIDDEN_SIZE))
model_balanced.add(Activation('softmax'))

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

model_balanced.summary()
```

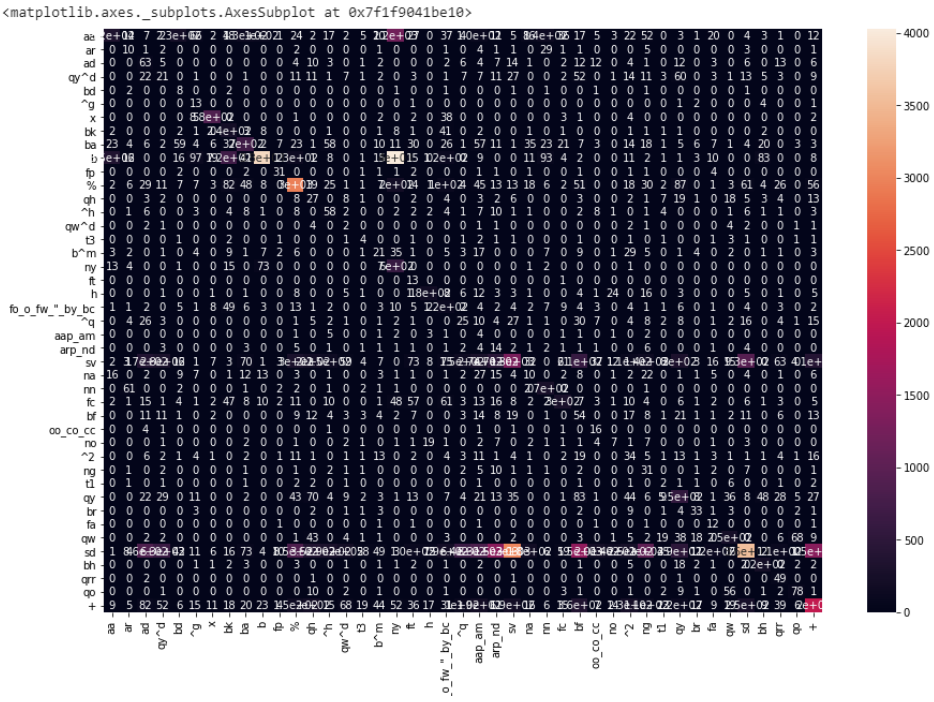
```
# Train the balanced network -  takes  time to achieve good accuracy
for i in range(10):
  model_balanced.fit(train_input,train_labels,validation_data=(val_input,val_labels), epochs = 5,batch_size=1000, verbose = 1,class_weight=d_class_weights)
```

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

```
Overall Accuracy: 51.48818760223611
```

Evaluation

<matplotlib.axes._subplots.AxesSubplot at 0x7f1f9041be10>



	precision	recall	f1-score	support
aa	0.43	0.12	0.18	2794
ar	0.07	0.16	0.10	63
ad	0.04	0.33	0.07	193
qy^d	0.02	0.07	0.03	311
bd	0.02	0.42	0.04	19
^g	0.05	0.59	0.09	22
x	0.93	0.94	0.93	941
bk	0.18	0.75	0.29	313
ba	0.61	0.56	0.59	1248
b	0.94	0.40	0.56	9634
fp	0.46	0.62	0.53	50
%	0.65	0.74	0.69	3981
qh	0.03	0.19	0.05	142
^h	0.09	0.39	0.14	149
qw^d	0.01	0.09	0.01	22
t3	0.04	0.16	0.06	25
b^m	0.09	0.12	0.10	181
ny	0.10	0.84	0.17	720
ft	0.03	0.93	0.06	14
h	0.59	0.62	0.60	291
fo_o_fw_"_by_bc	0.13	0.40	0.20	287
^q	0.03	0.12	0.04	212
aap_am	0.00	0.18	0.00	22
arp_nd	0.01	0.25	0.01	56
sv	0.28	0.20	0.23	6524
na	0.02	0.06	0.03	181
nn	0.45	0.79	0.57	345
fc	0.61	0.44	0.51	669
bf	0.02	0.22	0.03	249
oo_co_cc	0.05	0.64	0.09	25
no	0.03	0.11	0.05	66
^2	0.04	0.20	0.07	172
ng	0.02	0.42	0.04	73
t1	0.02	0.08	0.03	24
qy	0.31	0.48	0.38	1148
br	0.28	0.50	0.36	66
fa	0.05	0.63	0.10	19
qw	0.57	0.52	0.55	478
sd	0.72	0.19	0.30	18988
bh	0.53	0.82	0.65	265
qrr	0.13	0.91	0.23	54
qo	0.46	0.46	0.46	170
+	0.49	0.43	0.46	4696

```
"br" accuracy: 50.0
"bf" accuracy: 21.686746987951807
```

For the balanced model, it can be seen from the above pictures that the overall accuracy is less than before which is something we are expecting. “ in the framework of imbalanced data-sets, accuracy is no longer a proper measure, since it does not distinguish between the numbers of correctly classified examples of different classes. Hence, it may lead to erroneous conclusions. “

Looking at the minority classes it can be seen that their accuracy has improved dramatically. After weighting the minority classes (it took some time to get good results) both the precision and the recall of them have improved and are more accurate.

A way to handle imbalanced classes is to balance them by sampling, either by oversampling or undersampling instances. In theory, this should not lead to classifiers biased toward one class or the other. However, in practice, sampling has flaws. Oversampling the minority can lead to model overfitting, since it will introduce duplicate instances. Similarly, undersampling the majority can end up leaving out important instances.

Also, a different method for improving the accuracy of the classes is by applying thresholding. It relies on Bayes’ Theorem and the fact that neural networks estimate posterior distribution. In practice, it means that given a datapoint xx, the output for a neuron representing class ii corresponds to

$$yi(x)=p(i|x)=p(i)\cdot p(x|i)p(x)yi(x)=p(i|x)=p(i)\cdot p(x|i)p(x)$$

where p(i)p(i) is a prior probability for class ii.

In addition to them, One-class classification can be applied too(novelty detection). This is a concept learning technique that recognizes positive instances rather than discriminating between two classes.

Can we improve things by using context information? Yes, adding context information improves the overall accuracy but it is not guaranteed that it will improve the accuracy of all the classes, though. Having a look in the example below it can be seen the improvement of the network by using context information.

Task 2

```
[ ] from keras.layers import Input,Reshape,Conv2D,MaxPool2D,BatchNormalization,Flatten

    filter_sizes = [3,4,5]
    num_filters = 64
    drop = 0.2
    VOCAB_SIZE = len(wordvectors) # 43,731
    MAX_LENGTH = len(max(sentences, key=len))
    EMBED_SIZE = 100 # arbitary
    HIDDEN_SIZE = len(unique_tags)

    # CNN model
    inputs = Input(shape=(MAX_LENGTH, ), dtype='int32')
    embedding = Embedding(input_dim=VOCAB_SIZE, output_dim=EMBED_SIZE, input_length=MAX_LENGTH)(inputs)
    reshape = Reshape((MAX_LENGTH, EMBED_SIZE, 1))(embedding)

    # 3 convolutions
    conv_0 = Conv2D(num_filters, kernel_size=(filter_sizes[0], EMBED_SIZE), strides=1, padding='valid', kernel_initializer='normal', activation='relu')(reshape)
    bn_0 = BatchNormalization()(conv_0)
    conv_1 = Conv2D(num_filters, kernel_size=(filter_sizes[1], EMBED_SIZE), strides=1, padding='valid', kernel_initializer='normal', activation='relu')(reshape)
    bn_1 = BatchNormalization()(conv_1)
    conv_2 = Conv2D(num_filters, kernel_size=(filter_sizes[2], EMBED_SIZE), strides=1, padding='valid', kernel_initializer='normal', activation='relu')(reshape)
    bn_2 = BatchNormalization()(conv_2)

    # maxpool for 3 layers
    maxpool_0 = MaxPool2D(pool_size=(MAX_LENGTH - filter_sizes[0] + 1, 1), padding='valid')(bn_0)
    maxpool_1 = MaxPool2D(pool_size=(MAX_LENGTH - filter_sizes[1] + 1, 1), padding='valid')(bn_1)
    maxpool_2 = MaxPool2D(pool_size=(MAX_LENGTH - filter_sizes[2] + 1, 1), padding='valid')(bn_2)

    # concatenate tensors
    merge = keras.layers.concatenate([maxpool_0,maxpool_1,maxpool_2])
    # flatten concatenated tensors
    flat = Flatten()(merge)
    # dense layer (dense_1)
    dense_1 = Dense(HIDDEN_SIZE)(flat)
    # dropout_1
    dropout_1 = Dropout(drop)(dense_1)
```

```
[ ] # BLSTM model

    # Bidirectional 1
    BLSTM1 = Bidirectional(LSTM(HIDDEN_SIZE, return_sequences=True))(embedding)

    # Bidirectional 2
    BLSTM2 = Bidirectional(LSTM(HIDDEN_SIZE, return_sequences=False))(BLSTM1)

    # Dense layer (dense_2)
    dense_2 = Dropout(drop)(BLSTM2)

    # dropout_2
    dropout_2 = Dropout(drop)(dense_2)
```

Concatenate 2 last layers and create the output layer

```
[ ]  # concatenate 2 final layers

     final = keras.layers.concatenate([dropout_1, dropout_2])

     # output
     output = Dense(units=HIDDEN_SIZE, activation='softmax')(final)
```

```
[ ]  from keras import Model

     # Train the model - using validation
     model = keras.Model(inputs=inputs,outputs=output)
     model.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['accuracy'])
     model.fit(train_input,train_labels,validation_data=(val_input,val_labels), epochs = 5,batch_size=1000, verbose = 1)
```

It can be seen that the overall accuracy is improved compared to both previous models, it is not a huge improvement but it is still an improvement. For the minority classes, br's accuracy was improved while bf's wasn't.

```
Overall Accuracy: 70.33022046089172

"br" accuracy: 59.70149253731343
"bf" accuracy: 1.3333333333333335
```

What are frequent errors? Show one positive example where adding context changed the prediction.

To compare the classes we created again a heatmap of the confusion matrix, as it can be seen below.

One positive example is the ' fp' class. In the balanced model only 31 instances out of 69 where predict indeed as 'fp' (true positive) , while in this model (combined with context information) 42 instances were predicted correctly.

<matplotlib.axes._subplots.AxesSubplot at 0x7f413ffec5c0>