Lab 8 - Neural Machine Translation

March 10th

During the last lab, you've got a chance to play around with the Keras functional API by implementing a coreference system. For this lab, we will continue to use the Keras functional API to create a neural machine translation system based on the sequence-to-sequence (seq2seq) models proposed by Sutskever et al., 2014 and Cho et al., 2014. The seq2seq model is widely used in machine translation systems such as Google's neural machine translation system (GNMT) (Wu et al., 2016).

In today's lab and the one next week, we will explore the seq2seq model, as well as using attention in machine translation. The model you will implement during these two labs is similar to the GNMT and has the potential to achieve competitive performance with the GNMT by using larger and deeper networks.

For training and evaluating our mode we will use the English-Vietnamese parallel corpus of TED talks provided by the IWSLT Evaluation Campaign. For our tasks, we will translate from Vietnamese into English.

Again we will provide part of the code, and you are asked to fill the code blocks. In total, you will be given three files:

- One of them is the unfinished source code (nmt_model_keras.py)
- The remaining two are the parallel corpus, one for English (data.30.en) and one for Vietnamese (data.30.vi).

1. Nmt_model_keras.py, step by step

The script defines a total of three classes: the main class (`NmtModel`) , the attention layer class (`AttentionLayer`) and a helper class (`LanguageDict`). The `NmtModel` class contains most of the code of the NMT system, and is the one you are asked to finish for task 1 and 2. The `AttentionLayer` class is a custom layer to implement the attention mechanism, Task 3 is to finish this class. `LanguageDict` is a class that stores resources related to languages, such as vocab, word2ids etc.

1.1 The __init__() method: initialize the network parameters.

The method takes three arguments. The first two are instances of `LanguageDict`, one for the source language (Vietnamese) and one for the target language (English); the third argument is a boolean variable (`use_attention`) that indicates which model (attention/basic) should be used.

The method first defines a number of network parameters:

The number of hidden units used in the LSTMs of the model:
self.hidden_size = 200

The size of the word embedding. Here we use randomly initialized embeddings.
self.embedding_size = 100

The dropout rate for hidden layers and word embeddings.
self.hidden_dropout_rate=0.2
self.embedding_dropout_rate = 0.2

The batch size.
self.batch_size = 100

Then some resources for source/target languages:

The maximum length of the target sentences, this will be used during inference to avoid the model run forever.
self.max_target_step = 30

The size of vocabularies for both languages:
self.vocab_target_size = len(target_dict.vocab)
self.vocab_source_size = len(source_dict.vocab)

The instances of `LanguageDicts`:
self.target_dict = target_dict
self.source_dict = source_dict

The indices of the special tokens in the target language, (<start> the start of a sentence, <end> the end of a sentence). The are added in front/behind the target sentences to let the decoder known the status of translation:
self.SOS = target_dict.word2ids['<start>']
self.EOS = target_dict.word2ids['<end>']

The indicator of attention mechanism:
self.use_attention = use_attention

1.2 The build() method builds the keras models.

The method first creates the inputs for both training and inference models, which include the source/target sentence batches.

source_words = Input(shape=(None,),dtype='int32')
target_words = Input(shape=(None,), dtype='int32')

The inputs specifically used for the inference models are defined later.

It then comes to your first task, where you are required to create embeddings for both source/target languages as well as the encoder. We will discuss this in the later section.

After that we define the decoder for the training. The NMT has separate decoders for training and inference. During the training we feed the decoder the gold tokens, hence we process all tokens in the sentences in a single step. During the inference, the system processes one token at a time, the predicted token of the current step will be used as the input for the next step. More specifically, the size of `target_words` will be `[batch, max_sent_len]` during training and will be `[batch, 1]` during inference. Although the training and inference models behave slightly differently, they share all the layers (`decoder_lstm`, `decoder_attention` and `decoder_dense`)

```
decoder_lstm=LSTM(self.hidden_size,recurrent_dropout=self.hidden_dropout_rate,return_sequences =True,return_state=True)
decoder_outputs_train,_,_=decoder_lstm(target_words_embeddings,initial_state=encoder_states)


if self.use_attention:
 decoder_attention = AttentionLayer()
 decoder_outputs_train = decoder_attention([encoder_outputs,decoder_outputs_train])

decoder_dense = Dense(self.vocab_target_size,activation='softmax')
decoder_outputs_train = decoder_dense(decoder_outputs_train)
```

Finally, we define the training model:

```
adam = Adam(lr=0.01,clipnorm=5.0)
self.train_model = Model([source_words,target_words], decoder_outputs_train)
self.train_model.compile(optimizer=adam,loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
self.train_model.summary()
```

For inference, we will need to create two models, one for encoder and one for decoder. The encoder model is simple, we just specify the input and the output. The input is the source_words and it returns all three outputs of the encoder LSTM which will be finished by you in Task 1.

```
self.encoder_model=Model(source_words,[encoder_outputs,encoder_state_h,encoder_state_c])
self.encoder_model.summary()
```

For the decoder, it takes three additional inputs the initial states for the `decoder_lstm` and the `encoder_outputs`. The `encoder_outputs_input` is the `encoder_outputs` from the `encoder_model`, we simply pass it back for every step, it will be used for the attention model.

```
decoder_state_input_h = Input(shape=(self.hidden_size,))
decoder_state_input_c = Input(shape=(self.hidden_size,))
encoder_outputs_input = Input(shape=(None,self.hidden_size,))
```

Then is your Task 2, you are required to implement the decoder for inference. We will discuss this later.

Finally, we define the `decoder_model` for inference:

```
self.decoder_model=Model([target_words,decoder_state_input_h,decoder_state_input_c,encoder_outputs_input],[decoder_outputs_test,decoder_state_output_h,decoder_state_output_c])
self.decoder_model.summary()
```

1.3 The time_used() method outputs the time differences between the current time and the input time.

It is always good practice to record the time usage of an individual process, so you always know which part is most expensive to run.

```
curr_time = time.time()
used_time = curr_time-start_time
m = used_time // 60
s = used_time - 60 * m
return "%d m %d s" % (m, s)
```

1.4 The train() method oversees the training process.

It trains the model by going through all the training documents a number of times. It also evaluates the model on the development set after each epoch, and finally evaluates the model on the test set after the training finishes.

```
start_time = time.time()
for epoch in range(epochs):
 print("Starting training epoch {}/{}".format(epoch + 1, epochs))
 epoch_time = time.time()
 source_words_train, target_words_train, target_words_train_labels = train_data
self.train_model.fit([source_words_train,target_words_train],target_words_train_labels,
batch_size=self.batch_size)

 print("Time used for epoch {}: {}".format(epoch + 1, self.time_used(epoch_time)))
 dev_time = time.time()
 print("Evaluating on dev set after epoch {}/{}:".format(epoch + 1, epochs))
 self.eval(dev_data)
 print("Time used for evaluate on dev set: {}".format(self.time_used(dev_time)))
```

```
print("Training finished!")
print("Time used for training: {}".format(self.time_used(start_time)))

print("Evaluating on test set:")
test_time = time.time()
self.eval(test_data)
print("Time used for evaluate on test set: {}".format(self.time_used(test_time)))
```

1.5 The get_target_sentences() method takes sentence indices and return the string tokens

The method is a helper for the `eval` method, which is used to create reference and candidate sentences for evaluation.

```
str_sents = []
num_sent, max_len = sents.shape
for i in range(num_sent):
 str_sent = []
 for j in range(max_len):
   t = sents[i,j].item()
   if t == self.SOS:
     continue
   if t == self.EOS:
     break

   str_sent.append(vocab[t])
 if reference:
   str_sents.append([str_sent])
 else:
   str_sents.append(str_sent)
return str_sents
```

1.6 The eval() method runs a test on the given dataset.

The method first translates the source sentences into the target language, and then compares them with the reference sentences, as a result, it outputs the standard BLEU scores.

```
source_words, target_words_labels = dataset
vocab = self.target_dict.vocab
```

```
encoder_outputs,state_h,state_c=self.encoder_model.predict(source_words,batch_size=self
.batch_size)
predictions = []
step_target_words = np.ones([source_words.shape[0],1]) * self.SOS
for _ in range(self.max_target_step):
step_decoder_outputs,state_h,state_c=self.decoder_model.predict([step_target_words,
state_h,state_c,encoder_outputs],batch_size=self.batch_size)
 step_target_words = np.argmax(step_decoder_outputs,axis=2)
 predictions.append(step_target_words)

candidates = self.get_target_sentences(np.concatenate(predictions,axis=1),vocab)
references = self.get_target_sentences(target_words_labels,vocab,reference=True)

score = corpus_bleu(references,candidates)
print("Model BLEU score: %.2f" % (score*100.0))
```

1.7 The AttentionLayer() class creates a custom layer for attention

This class contains three methods, the first one is used for passing the mask to the next layer. The mask is originally created by the `Embedding` layer with the `mask_zeros` attribute set to `True`, so it will remove the paddings for things like `Loss` and `metric` or `LSTM` layers. The `AttentionLayer()` takes two inputs: the `encoder_outputs` and the `decoder_outputs` and it returns a `new_decoder_outputs` that leverages the `decoder_outputs` with the `encoder_outputs`. So here we return the mask for the `decoder_outputs`. The second method computes the output shape of our layer. The output shape of the layer is the same to the decoder_outputs in the first two dimensions and for the last dimension it doubles the representation.

```
def compute_mask(self, inputs, mask=None):
 if mask == None:
   return None
 return mask[1]

def compute_output_shape(self, input_shape):
 return (input_shape[1][0],input_shape[1][1],input_shape[1][2]*2)
```

The third method is the main method for the layer, and also the one you will need to implement for your task 3 and we will come back later for that.

1.8 The LanguageDict() class stores the language resources

This class only has an initialization method, the method takes a corpus as the input and builds the vocab and word2ids for the language.

```
class LanguageDict():
 def __init__(self, sents):
   word_counter = collections.Counter(tok.lower() for sent in sents for tok in sent)
```

```
  self.vocab = []
  self.vocab.append('<pad>') #zero paddings
  self.vocab.append('<unk>')
  self.vocab.extend([t for t,c in word_counter.items() if c > 10])

  self.word2ids = {w:id for id, w in enumerate(self.vocab)}
  self.UNK = self.word2ids['<unk>']
  self.PAD = self.word2ids['<pad>']
```

1.9 The load_dataset() method creates train/dev/test batches

The method reads the given file and load the first max_num_examples sentences and split them into train/dev/test dataset:

```
source_lines = open(source_path).readlines()
target_lines = open(target_path).readlines()
assert len(source_lines) == len(target_lines)
if max_num_examples > 0:
 max_num_examples = min(len(source_lines), max_num_examples)
 source_lines = source_lines[:max_num_examples]
 target_lines = target_lines[:max_num_examples]

source_sents = [[tok.lower() for tok in sent.strip().split(' ')] for sent in source_lines]
target_sents = [[tok.lower() for tok in sent.strip().split(' ')] for sent in target_lines]
for sent in target_sents:
 sent.append('<end>')
 sent.insert(0,'<start>')

source_lang_dict = LanguageDict(source_sents)
target_lang_dict = LanguageDict(target_sents)

unit = len(source_sents)//10

source_words = [[source_lang_dict.word2ids.get(tok,source_lang_dict.UNK) for tok in sent]
for sent in source_sents]
source_words_train = pad_sequences(source_words[:8*unit],padding='post')
source_words_dev = pad_sequences(source_words[8*unit:9*unit],padding='post')
source_words_test = pad_sequences(source_words[9*unit:],padding='post')

eos = target_lang_dict.word2ids['<end>']

target_words = [[target_lang_dict.word2ids.get(tok,target_lang_dict.UNK) for tok in sent[:-1]]
for sent in target_sents]
```

```
target_words_train = pad_sequences(target_words[:8*unit],padding='post')

target_words_train_labels = [sent[1:]+[eos] for sent in target_words[:8*unit]]
target_words_train_labels = pad_sequences(target_words_train_labels,padding='post')
target_words_train_labels = np.expand_dims(target_words_train_labels,axis=2)

target_words_dev_labels = pad_sequences([sent[1:] + [eos] for sent in target_words[8 *
unit:9 * unit]], padding='post')
target_words_test_labels = pad_sequences([sent[1:] + [eos] for sent in target_words[9 *
unit:]], padding='post')

train_data = [source_words_train,target_words_train,target_words_train_labels]
dev_data = [source_words_dev,target_words_dev_labels]
test_data = [source_words_test,target_words_test_labels]

return train_data,dev_data,test_data,source_lang_dict,target_lang_dict
```

1.10 The __main__ method starts the training.

Please note you will need to change the use_attention variable within the method to evaluate with different versions of the model.

```
if __name__ == '__main__':
 max_example = 30000
 use_attention = True
 train_data,dev_data,test_data,source_dict,target_dict=load_dataset("data.30.vi","data.30.e
n",max_num_examples=max_example)
 print("read %d/%d/%d train/dev/test batches" % (len(train_data[0]),len(dev_data[0]), len(
test_data[0])))

 model = NmtModel(source_dict,target_dict,use_attention)
 model.build()
 model.train(train_data,dev_data,test_data,10)
```

2 Task 1: Implement the Embedding Layers and the Encoder

In this task, you will work at the beginning of the `build` method.

This task is fairly simple to implement. You will need to first create two `Embedding` layers (one for source language and one for target language). Then pass the embedding for the source language into a `LSTM` layer.

Let's first look at the inputs. You have in total of two inputs:

- source_words: the word indices of the sentences in the source language. This will have the shape `[batch_size, max_source_sent_len]` during both training and inference.
- target_words: the word indices of the sentences in the target language. During training it will have a shape of `[batch_size, max_target_sent_len]`, but will have a shape of `[batch_size, 1]` during the inference.

You will need to first create two `Embedding` layers `embedding_source` and `embedding_target`. The `Embedding` layers will randomly initialize the embeddings for individual words in the vocabulary and the embeddings will be trained during the training. The `Embedding` layers have an `input_dim` of the `vocab_size` and a `output_dim` of the `embedding_size`. Please note the `vocab_size` for source and target language are usually different. Also you will need to set the `mask_zero` to `True` in order to get rid of the paddings.

Secondly, you need to look up the embeddings for the current inputs (`source_words` and `target_words`) by passing them through the `Embedding` layers you created. The embeddings for source and target words need to be called `source_words_embeddings` and `target_words_embeddings` respectively. After that you will need to apply dropout to the embeddings by using the `Dropout` layers. The dropout rate for the word embeddings is called `embedding_dropout_rate`.

Thirdly, you can create a `LSTM` layer to process the `source_words_embeddings`, you will need to set the `return_sequences` to `True` to get the outputs for all tokens (call it `encoder_outputs`) and set the `return_state` to `True` to get the hidden state (`encoder_state_h`) and cell state (`encoder_state_c`) from the encoder LSTM.

3 Task 2: Implement the Decoder for inference

In this task, you will work on the second half of the `build` method.

The decoder for inference is similar to the decoder for training but it only performs one step of the decoding. Remember the decoders share all the layers, you will need to use the layers created in the decoder for training. In total three layers were used in both decoders, namely the `decoder_lstm`,(the decoder LSTM layer) `decoder_dense` (the decoder final layer) and the `decoder_attention` (the attention layer for attention based model).

Let's take a look how should we create the decoder for inference:

First, unlike the decoder for training that uses the `encoder_states` as the `initial_state` for `decoder_lstm`, we need to use the states passed to the decoder_model instead (`decoder_state_input_h, decoder_state_input_c`). You need to put them together in a list to create the decoder_states. If you take a look at the `eval` method you will find out that for the first step, the `decoder_states` passed into the

model is actually the `encoder_states` (same as in decoder for training), while in the subsequent steps the `decoder_states` become the one the `decoder_model` returns in the previous step.

Secondly, you will need to pass the `target_word_embeddings` and `decoder_states` to the `decoder_lstm`.

Thirdly you will write a if statement for the attention model just like we did in the decoder for training.

Finally, pass the output of the LSTM (for basic model) or the attention layer (for attention model) into the final layer of the decoder (`decoder_dense`) to assign probabilities of the next tokens.

After you finish the decoder for inference you've already made a functional NMT system, why not test it out see how well it works. Please note you need to set the `use_attention` to `False` since you haven't implemented the attention layer yet. The system will take about 10 minutes to finish 10 epochs of training and you will get a BLEU score around 4-5.

4 Task 3: Implement the Attention layer

In this task, you will work on the `call` method of the `AttentionLayer` class.

The attention decoder is the secret recipe for the success of the NMT. It enables the decoder to access all encoder outputs and focus on different parts of the encoder outputs during different steps. By contrast, the basic model only has access to the final states of the encoder. There are a few different styles of attention mechanism; here we use the one proposed in Luong et al., (2015), which computes the score between `decoder_outpus` and `encoder_outputs` by matrix multiplication.

Please note since we are creating a custom layer we can use any backend methods directly and no need to wrap the methods with a `Lambda` layer.

First, let's take a look at the shape of our inputs (`encoder_outputs`, `decoder_outputs`), the `encoder_outputs` has a shape of `[batch_size, max_source_sent_len, hidden_size]` and the decoder_outputs has a shape of `[batch_size, max_target_sent_len, hidden_size]`. In order to multiply them, we need first transpose the last two dimensions of the `decoder_outputs` to make it shape becomes `[batch_size, hidden_size, max_target_sent_len]`, you will need to use the backend `permute_dimensions` method to do this.

Once the `decoder_output` is transposed we use the `batch_dot` method to do the multiplications. Let's call the output `luong_score`, it has a shape of `[batch_size, max_source_sent_len, max_target_sent_len]` then you need apply a softmax to

the dimension that have a size of `max_sourse_sent_len` to create an attention score for the `encoder_outputs`.

Finally we are going to create the `encoder_vector` by doing element-wise multiplication between the `encoder_outputs` and their attention scores (`luong_score`). But as you may have noticed the shape of the `luong_score` is actually not the same as the `encoder_outputs`, so we need to use the `expand_dims` method to expand dimensions for both of them. For `luong_score` you need to expand the last dimension to accommodate the `hidden_size` dimension of the `encoder_outputs`, so after the expansion, 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]`. When multiply between the two tensors, the expanded dimensions will be broadcasted so that they have the same shape. The last step is to sum the `max_source_sent_len` dimension to create the `encoder_vector`.

Before returning the `new_decoder_outputs` we concatenate the `decoder_outputs` and the `encoder_vector` using the `concatenate` method (the code is already provided).

Congratulations again, you've created an attention NMT system, let's run your code (remember to set `use_attention` to `True`), it will take about 10 minutes on GPU to train and you will get a much better BLEU score, usually around 15 (more than three times of the score for the basic version).