Lab 8 - Neural Machine Translation

March 5

During the last lab, you've got a chance to play around with the low level Tensorflow API by implementing a coreference system. For this lab, we will continue to use the low level 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.py)
- The remaining two are the parallel corpus, one for English (data.30.en) and one for Vietnamese (data.30.vi).

1. Nmt_model.py, step by step

The script defines in total of two classes: the main class (`NmtModel`) 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 modify. `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 layers and units used in the LSTMs of the model:

```
self.num_layers = 2
```

```
self.hidden_size = 200
```

The size of the word embedding. Here we use randomly initialzed embeddings.

```
self.embedding_size = 100
```

The dropout rate for hidden layer and word embeddings.

```
self.hidden_dropout_rate=0.2
self.embedding_dropout_rate = 0.2
```

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 infront/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 tensorflow graph.

The method first creates the placeholders for the tensorflow graph, which include the source/target sentence batches and the individual lengths of the sentences in the batches, as well as a train/inference indicator.

```
self.source_words = tf.placeholder(tf.int32,[None,None],"source_words")
self.target_words = tf.placeholder(tf.int32,[None,None],"target_words")
self.source_sent_lens = tf.placeholder(tf.int32,[None],"source_sent_lens")
self.target_sent_lens = tf.placeholder(tf.int32,[None],"target_sent_lens")
self.is_training = tf.placeholder(tf.bool,[],"is_training")
```

It then passes all the inputs to the `get_predictions_and_loss` method to get predictions and loss of the training/inference.

```
self.predictions,self.loss = self.get_predictions_and_loss(self.source_words,
self.target_words,self.source_sent_lens,self.target_sent_lens,self.is_training)
```

Finally, the method defines the training mechanism and initializes global variables of our model.

```
trainable_params = tf.trainable_variables()
gradients = tf.gradients(self.loss, trainable_params)
gradients, _ = tf.clip_by_global_norm(gradients, 5.0)
optimizer = tf.train.AdamOptimizer(learning_rate=0.01)
self.train_op = optimizer.apply_gradients(zip(gradients, trainable_params))
self.sess = tf.Session()
self.sess.run(tf.global_variables_initializer())
```

1.3 The get_predictions_and_loss() method processes the inputs and returns the predictions and loss

The method first creates the word embeddings for both source and target languages and some commonly used variables (batch_size, keep_prob etc.).

```
self.embeddings_target = tf.get_variable("embeddings_target", [self.vocab_target_size,
self.embedding_size], dtype=tf.float32)
self.embeddings_source = tf.get_variable("embeddings_source", [self.vocab_source_size,
self.embedding_size], dtype=tf.float32)

batch_size = shape(target_words, 0)
max_target_sent_len = shape(target_words, 1)

embedding_keep_prob = 1 - (tf.to_float(is_training) * self.embedding_dropout_rate)
hidden_keep_prob = 1 - (tf.to_float(is_training) * self.hidden_dropout_rate)
```

It then looks up the word embeddings for the given source/target sentence batches:

```
source_embs = tf.nn.dropout(tf.nn.embedding_lookup(self.embeddings_source,
source_words), embedding_keep_prob)
target_embs = tf.nn.dropout(tf.nn.embedding_lookup(self.embeddings_target,
target_words),embedding_keep_prob)
```

After that, the source sentences are passed to the `encoder` to obtain the LSTM outputs and final states. To implement the `encoder` method is also your first task which will be discussed in the later sections.

```
encoder_outputs, encode_final_states = self.encoder(source_embs, source_sent_lens,
hidden_keep_prob)
```

For the decoder, we will need to use the tensorflow loop (`tf.scan`) method to go through sentences token by token. During the inference the previously predicted tokens will be used as the input to predict the next token, hence they are not known in advance. During the training, we simply feed the gold tokens from the reference translations, to do this we first transpose the original target_embs that has a shape of [batch_size, max_steps, emb] (batch major) into the time_major_target_embs (after transpose the shape become [max_steps, batch_size, emb]).

```
time_major_target_embs = tf.transpose(target_embs,[1,0,2])
```

Then we define the `_decoder_scan` method, which will be called by the `tf.scan` method to conduct a single loop. The method used by `tf.scan` are required to take two input tuples, the first one stores the outputs of the previous step, the second one stores one slice of the input variables. In our case, the outputs from the previous step have three variables: logits (the probability distribution on target language vocabulary), pred (the prediction), and states (the LSTM states). The input variables only contain one variable (time_marjor_target_embs), in each loop 1 slice of the variable will be feed into the `_docoder_scan` method, which has a shape of [batch_size, emb].

```
def _decoder_scan(pre,inputs):
 pre_logits, pre_pred, pre_states = pre
 step_embeddings = inputs
```

It then finds the embeddings for the predicted tokens of the previous step, depends on whether it is training or inference the embeddings for gold/predicted tokens will be used. The `tf.cond` method is equivalent to an `if … else` statement.

```
pred_embeddings = tf.nn.embedding_lookup(self.embeddings_target,pre_pred)
step_embeddings = tf.cond(is_training,lambda : step_embeddings,
lambda : pred_embeddings)
```

After that, we pass the step_embeddings, encoder_outputs (for attention model) and pre_states (the LSTM states of the previous step) together with the hidden_keep_prob to the `step_decoder` method that processes the decoding for one step. Depends on the configuration (use_attention) it will process either attention decoder or basic decoder. Task 2 and task 3 are to implement the basic and attention decoders respectively, both should be implemented in the `step_decoder` method. We will come back to this later.

```
curr_logits, curr_states = self.step_decoder(step_embeddings, encoder_outputs, pre_states,
hidden_keep_prob)
curr_pred = tf.argmax(curr_logits,1,output_type=tf.int32)
```

```
return curr_logits, curr_pred, curr_states
```

To use the `tf.scan` method we need to also provide the initial values of the variables of "pre" (pre_logits, pre_pred, and pre_states). The init_logits are all initialized to zeros as we don't use them for the first step of the loop. However, we need to use init_pred which is initialized to the special token SOS (the start of the sentence). The final states of the encoder (encoder_final_states) are used as the initializer for pre_states. The `tf.scan` method returns a list of tensors for each output, we use `tf.stack` method to stack them into a single tensor. And transpose the time major variables back to batch major.

```
init_logits = tf.zeros([batch_size,self.vocab_target_size])
init_pred = tf.ones([batch_size],tf.int32) * self.SOS

time_major_logits, time_major_preds, _ = tf.scan(_decoder_scan, time_major_target_embs,
initializer=(init_logits, init_pred, encode_final_states))
time_major_logits,  time_major_preds = tf.stack(time_major_logits),
tf.stack(time_major_preds)

logits = tf.transpose(time_major_logits,[1,0,2])
predictions = tf.transpose(time_major_preds,[1,0])
```

Then we create a mask to get rid of the padding before computing the loss:

```
logits_mask = tf.sequence_mask(target_sent_lens-1,max_target_sent_len)
flatten_logits_mask = tf.reshape(logits_mask,[batch_size*max_target_sent_len])
flatten_logits = tf.boolean_mask(tf.reshape(logits, [batch_size * max_target_sent_len,
self.vocab_target_size]), flatten_logits_mask)

gold_labels_mask = tf.concat([tf.zeros([batch_size,1],dtype=tf.bool), tf.sequence_mask(
target_sent_lens-1, max_target_sent_len-1)],1)
flatten_gold_labels_mask=tf.reshape(gold_labels_mask,[batch_size*max_target_sent_len])
flatten_gold_labels = tf.boolean_mask(tf.reshape(target_words,[batch_size *
max_target_sent_len]), flatten_gold_labels_mask)
```

Finally, we compute the softmax cross-entropy loss:

```
loss = tf.reduce_mean( tf.nn.sparse_softmax_cross_entropy_with_logits(
labels=flatten_gold_labels, logits=flatten_logits))
```

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

It is always a good practice to record the time usage of individual process, so you always known 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)
```

1.5 The train() method oversees the training process.

It trains the model by going through all the training documents a number of times. It also outputs the average loss and time usage of the training. 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()
 losses = []
 source_train,target_train = train_data
 for i, (source,target) in enumerate(zip(source_train,target_train)):
   source_words,source_sent_lens = source
   target_words,target_sent_lens = target
   fd = {self.source_words:source_words,self.target_words:target_words,
         self.source_sent_lens:source_sent_lens,self.target_sent_lens:target_sent_lens,
         self.is_training:True}

   _, loss= self.sess.run([self.train_op, self.loss], feed_dict=fd)

   losses.append(loss)
   if (i+1) % 100 == 0:
     print("[{}]: loss:{:.2f}".format(i+1, sum(losses[i + 1 - 100:]) / 100.0))
 print("Average epoch loss:{}".format(sum(losses) / len(losses)))
 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)))
```

1.6 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.

```
def get_target_sentences(self, sents,vocab,reference=False,isnumpy=False):
 str_sents = []
 for sent in sents:
   str_sent = []
   for t in sent:
     if isnumpy:
       t = t.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
```

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

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

```
def eval(self, dataset):
 source_batches, target_batches = dataset
 references = []
 candidates = []
 vocab = self.target_dict.vocab
 PAD = self.target_dict.PAD

 for i, (source, target) in enumerate(zip(source_batches, target_batches)):
   source_words, source_sent_lens = source
   target_words, target_sent_lens = target
   infer_target_words = [[PAD for i in range(self.max_target_step)] for b in target_words]

   fd = {self.source_words: source_words, self.target_words: infer_target_words,
        self.source_sent_lens: source_sent_lens,
```

```
        self.is_training: False}
    predictions = self.sess.run(self.predictions,feed_dict=fd)

    references.extend(self.get_target_sentences(target_words,vocab,reference=True))
    candidates.extend(self.get_target_sentences(predictions,vocab,isnumpy=True))

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

1.8 The shape() method helps us get the n-th dimension of a given tensor.

In tensorflow there are two ways of getting a tensor's shape. The `Tensor.get_shape()[n]` method can get a predefined dimension of the tensor, such as the last dimension of the word_embeddings (100) or the dimension of the hidden layer (200). The second method (`tf.shape()[n]`) returns the dynamic size of the tensor, such as the max_target_sent_len. Those sizes are not fixed across different batches thus are dynamic.

1.9 The LanguageDict() class stores the language resources

This class only have a 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 = [t for t,c in word_counter.items() if c > 10]
    self.vocab.append('<pad>')
    self.vocab.append('<unk>')
    self.word2ids = {w:id for id, w in enumerate(self.vocab)}
    self.UNK = self.word2ids['<unk>']
    self.PAD = self.word2ids['<pad>']
```

1.10 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:

```
  lines = [line for line in open(path,'r')]
  if max_num_examples > 0:
    max_num_examples = min(len(lines), max_num_examples)
    lines = lines[:max_num_examples]

  sents = [[tok.lower() for tok in sent.strip().split(' ')] for sent in lines]
```

```
if add_start_end:
  for sent in sents:
    sent.append('<end>')
    sent.insert(0,'<start>')

lang_dict = LanguageDict(sents)

sents = [[lang_dict.word2ids.get(tok,lang_dict.UNK) for tok in sent] for sent in sents]

batches = []
for i in range(len(sents) // batch_size):
  batch = sents[i * batch_size:(i + 1) * batch_size]
  batch_len = [len(sent) for sent in batch]
  max_batch_len = max(batch_len)
  for sent in batch:
    if len(sent) < max_batch_len:
      sent.extend([lang_dict.PAD for _ in range(max_batch_len - len(sent))])
  batches.append((batch, batch_len))


unit = len(batches)//10
train_batches = batches[:8*unit]
dev_batches = batches[8*unit:9*unit]
test_batches = batches[9*unit:]

return train_batches,dev_batches,test_batches,lang_dict
```

1.11 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__':
 batch_size = 100
 max_example = 30000
 use_attention = True
 source_train, source_dev, source_test, source_dict = load_dataset("data.30.vi",
 max_num_examples=max_example, batch_size=batch_size)
 target_train, target_dev, target_test, target_dict = load_dataset("data.30.en",
 max_num_examples=max_example,batch_size=batch_size, add_start_end=True)
 print("read %d/%d/%d train/dev/test batches" % (len(source_train),len(source_dev),
 len(source_test)))

 train_data = (source_train,target_train)
 dev_data = (source_dev,target_dev)
```

```
test_data = (source_test,target_test)

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

2 Task 1: Implement the Encoder

In this task, you will work on the `encoder` method.

The `encoder` is fairly simple to implement. It is very similar to the bi-directional LSTMs you implemented in the last lab. Here you are required to implement a two-layer single directional LSTMs.

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

- embeddings: the word embedding of the source sentence batch, which has a shape of [batch_size, max_steps, emb].
- sent_lens: the individual sentence lengths of the sentences in the batch.
- hidden_keep_prob: the keep probability of the hidden layers.

You will need first create a list of two lstm_cells using `tf.nn.rnn_cell_LSTMCell` and `tf.nn.rnn_cell.DropoutWrapper`. The hidden layer size of the LSTM is defined in global variable self.hidden_size.

Secondly, the `tf.nn.rnn_cell.MultiRNNCell` can be used as a wrapper for the two layers of LSTMs and make them one multi-layer LSTM.

Thirdly, you can use `tf.nn.dynamic_rnn` to run the LSTM cells to get you the outputs and the final states, return both of them.

3 Task 2: Implement the Basic Step Decoder

In this task, you will work on the `step_decoder` method.

The basic step decoder performs one step of the decoding, which takes a step_embeddings (with a shape of [batch_size, emb]) and pre_states (previous LSTM states) and performs one step of multi-layer LSTM. It then makes the predictions by assign weights to individual words in the vocabulary of the target language.

Similar to task 1, we first create a two-layer LSTM cell using `tf.nn.rnn_cell_LSTMCell`, `tf.nn.rnn_cell.DropoutWrapper` and `tf.nn.rnn_cell.MultiRNNCell`.

Secondly, since we don't process the whole sentences, instead we only process one step of the LSTM. We can directly call the LSTM cell by feeding the cell the step_embeddings and the pre_states, it will return the LSTM output of this step (step_decoder_output) and new states.

Finally, we need to create an FFNN to compute the probability distribution on target language vocabulary (logits). Since the vocabulary can be large, a deep FFNN can be expensive to run, here we use an FFNN without any hidden layer. Which means we only need to create a single weights and bias. Remember the weights have a shape of [last dimension of the input, the output dimension] and the bias has a size of [the output dimension]. The input is the step_decoder_output and the output dimension is the size of the vocabulary of the target language (self.vocab_target_size). You might want to use `tf.nn.xw_plus_b` method for computing the above logits.

After you finished the basic step decoder you've already made a functional NMT system, why not test it out see how well it works. The system will take about 20 minutes to finish 10 epochs of training and you will get a BLEU score around 4-5.

4 Task 3: Implement the Attention Step Decoder

In this task, you will still work on the `step_decoder` method.

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 decoder 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 step_decoder_output and encoder_outputs by matrix multiplication.

We can reuse the code from task 2 upto the second step and put the code created for the last step of task 2 in an `if` statement (i.e. `if not self.use_attention:`).

In the `else` statement we start to create the attention decoder:

First, let's compute the raw attention scores for individual encoder outputs. You need to use the `tf.matmul` method to perform the matrix multiplication between step_decoder_output and encoder_outputs. Remember the step_decoder_output is a matrix with a shape of [batch_size, emb] but the encoder_outputs is a tensor with a shape of [batch_size, max_step, emb]. In order to make the multiplication we need to use `tf.expand_dims` to change the shape of the step_decoder_output in to [batch_size,emb,1]. The expected raw score has a shape of [batch_size, max_step, 1] so put the encoder_outputs first then the expanded step_decoder_output.

After we got the raw score, we need to apply softmax (using `tf.nn.softmax`) to the score, which will give you a probability distribution that sum to one. You need to make sure you apply the softmax on the right axis.

Then we need can multiply the softmax_score with the encoder_outputs to create a weighted sum of the encoder_outputs let's call it encoder_vector. Here you will need to use the `tf.reduce_sum` function and think carefully which axis to sum.

The final step is to concatenate the step_decoder_output and the encoder_vector using the `tf.concat` method. And then create an FFNN to compute the logits.

Congratulations again, you've created an attention NMT system, let's run your code (remember to set use_attention=True), it will take about 20 minutes to train and you will get a much better BLEU score, usually above 10 (more than twice of the score for the basic version).