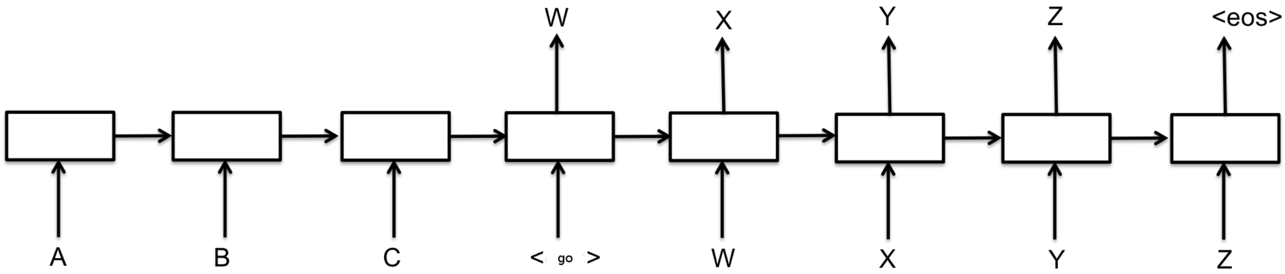
ADL2016 Hw3 Report

Brief introduction

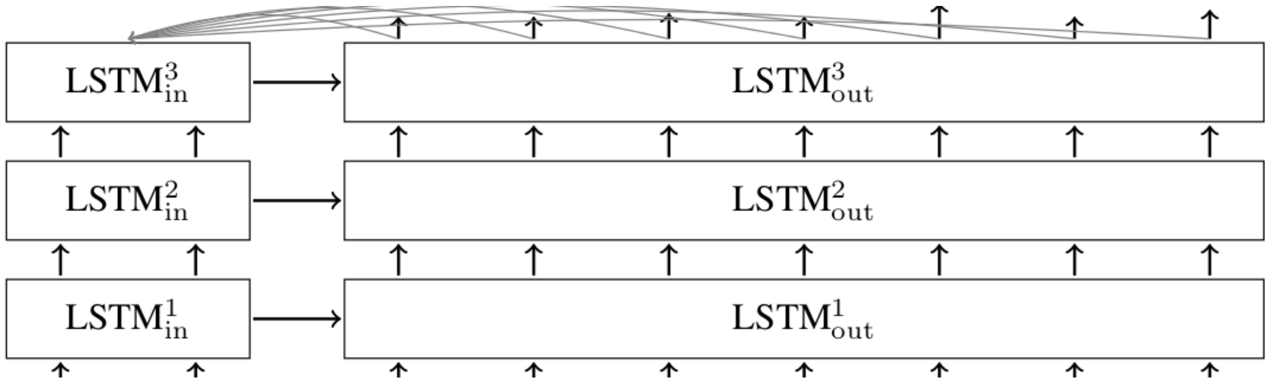
A basic sequence-to-sequence model, as introduced in Cho et al., 2014 (pdf), consists of two recurrent neural networks (RNNs): an encoder that processes the input and a decoder that generates the output. This basic architecture is depicted below.



<fig1>

Each box in the picture above represents a cell of the RNN, most commonly a GRU cell or an LSTM cell. Encoder and decoder can share weights or, as is more common, use a different set of parameters.

In the basic model depicted above, every input has to be encoded into a fixed-size state vector, as that is the only thing passed to the decoder. To allow the decoder more direct access to the input, an attention mechanism was introduced in Bahdanau et al., 2014 (pdf). A multi-layer sequence-to-sequence network with LSTM cells and attention mechanism in the decoder looks like this.


$\mathrm{LSTM}^3_{\mathrm{in}}$
$\mathrm{LSTM}^3_{\mathrm{out}}$
$\mathrm{LSTM}^2_{\mathrm{in}}$
$\mathrm{LSTM}^2_{\mathrm{out}}$
$\mathrm{LSTM}^1_{\mathrm{in}}$
$\mathrm{LSTM}^1_{\mathrm{out}}$

<fig2>

TensorFlow seq2seq Library

```
outputs, states = embedding_rnn_seq2seq(
    encoder_inputs, decoder_inputs, cell,
    num_encoder_symbols, num_decoder_symbols,
    output_projection=None, feed_previous=False)
```

In the embedding_rnn_seq2seq model, all inputs (both encoder_inputs and decoder_inputs) are integer-tensors that represent discrete values. They will be embedded into a dense representation, but to construct these embeddings we need to specify the maximum number of discrete symbols that will appear: num_encoder_symbols on the encoder side, and num_decoder_symbols on the decoder side.

In the above invocation, we set feed_previous to False. This means that the decoder will use decoder_inputs tensors as provided. If we set feed_previous to True, the decoder would only use the first element of decoder_inputs. All other tensors from this list would be ignored, and instead the previous output of the decoder would be used. This is used for decoding translations in our translation model, but it can also be used during training, to make the model more robust to its own mistakes, similar to Bengio et al., 2015 (pdf).

One more important argument used above is output_projection. If not specified, the outputs of the embedding model will be tensors of shape batch-size by num_decoder_symbols as they represent the logits for each generated symbol. When training models with large output vocabularies, i.e., when num_decoder_symbols is large, it is not practical to store these large tensors. Instead, it is better to return smaller output tensors, which will later be projected onto a large output tensor using output_projection. This allows to use our seq2seq models with a sampled softmax loss, as described in Jean et. al., 2014 (pdf).

Neural Translation Model - Sampled softmax and output projection

For one, as already mentioned above, we want to use sampled softmax to handle large output vocabulary. To decode from it, we need to keep track of the output projection. Both the sampled softmax loss and the output projections are constructed by the following code in seq2seq_model.py.

```
 if num_samples > 0 and num_samples < self.target_vocab_size:
  w = tf.get_variable("proj_w", [size, self.target_vocab_size])
  w_t = tf.transpose(w)
  b = tf.get_variable("proj_b", [self.target_vocab_size])
  output_projection = (w, b)

  def sampled_loss(inputs, labels):
   labels = tf.reshape(labels, [-1, 1])
   return tf.nn.sampled_softmax_loss(w_t, b, inputs, labels,
num_samples,
                    self.target_vocab_size)
```

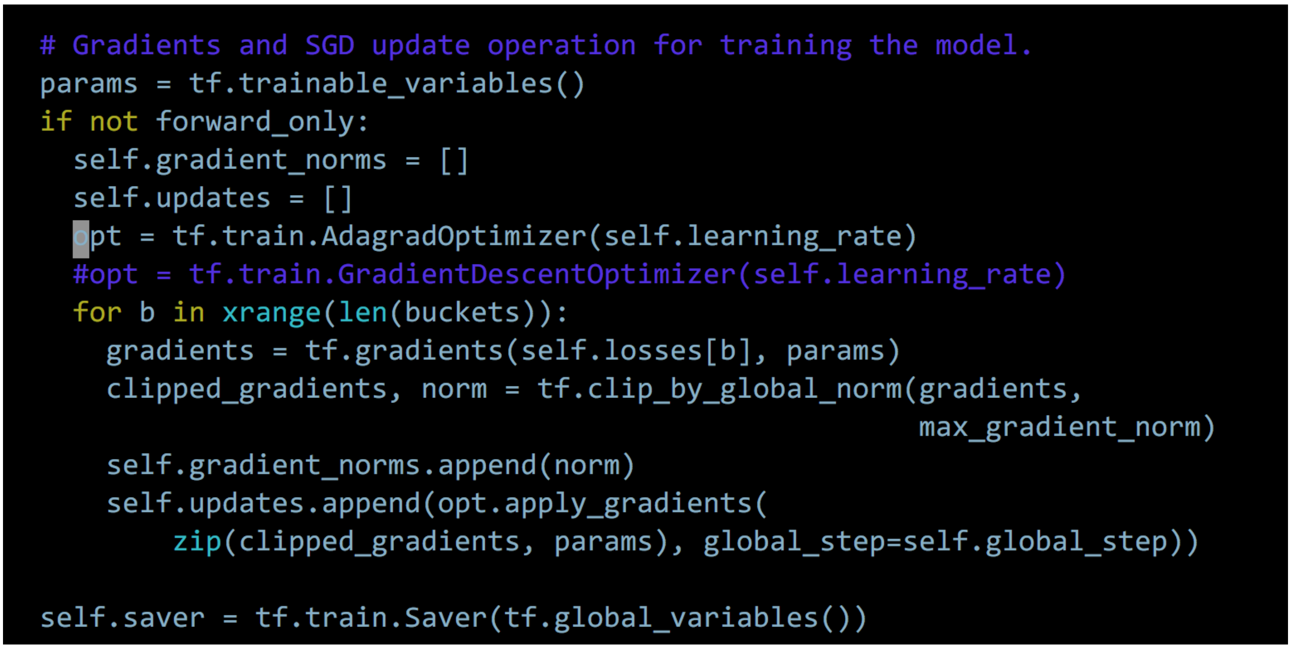
First, note that we only construct a sampled softmax if the number of samples (512 by default) is smaller than the target vocabulary size. For vocabularies smaller than 512, it might be a better idea to just use a standard softmax loss.

Then, as you can see, we construct an output projection. It is a pair, consisting of a weight matrix and a bias vector. If used, the rnn cell will return vectors of shape batch-size by size, rather than batch-size by target_vocab_size. To recover logits, we need to multiply by the weight matrix and add the biases, as is done in lines 124-126 in seq2seq_model.py.

```
if output_projection is not None:
  self.outputs[b] = [tf.matmul(output, output_projection[0]) +
                     output_projection[1] for …]
```

My Code

I add a python json parser to handle the input and modified several part of the code to get the correct input. I change the optimizer to AdagradOptimizer, and initial the learning rate to 0.5. After 20000 steps, the outcome can beat the strong baseline.



```
# Gradients and SGD update operation for training the model.
params = tf.trainable_variables()
if not forward_only:
  self.gradient_norms = []
  self.updates = []
  opt = tf.train.AdagradOptimizer(self.learning_rate)
  #opt = tf.train.GradientDescentOptimizer(self.learning_rate)
  for b in xrange(len(buckets)):
    gradients = tf.gradients(self.losses[b], params)
    clipped_gradients, norm = tf.clip_by_global_norm(gradients,
                                                     max_gradient_norm)
    self.gradient_norms.append(norm)
    self.updates.append(opt.apply_gradients(
        zip(clipped_gradients, params), global_step=self.global_step))

self.saver = tf.train.Saver(tf.global_variables())
```

Conclusion

In this assignment, I learned how to model sequences and do machine translation or NLG tasks. Very exciting to see the result.