Inference Decoding

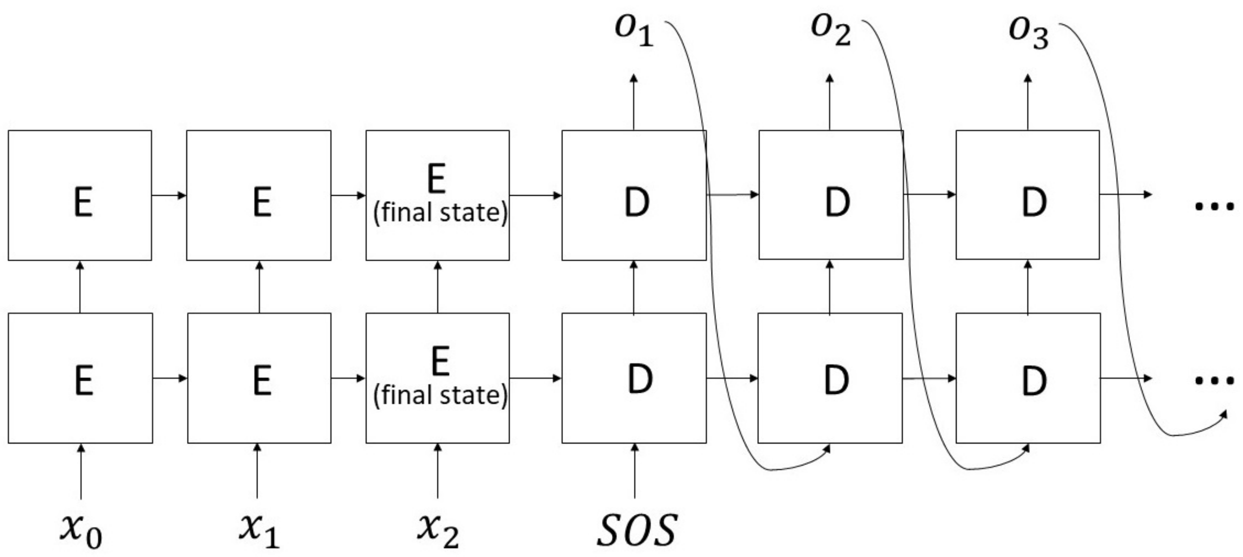
Understand the difference between training and inference decoding.

Chapter Goals:

- Learn how to perform decoding for inference
- Learn about variable scopes for declaring and retrieving variables

A. Decoding without ground truth

During training, we have access to ground truth tokens, which are used as input for the decoder. However, for inference (i.e. generating output in real-time), we only have access to an input sequence. In this case, the input for the decoder at each time is just the decoder's output token from the previous time step.



Using a decoder during inference. The SOS token is always the initial decoder input token. For simplicity, the attention mechanism is not shown in this diagram.

In TensorFlow, we can switch from training to inference mode just by changing the `TrainingHelper` object to an inference helper. The most commonly used inference helper object is `GreedyEmbeddingHelper`.

B. Greedy decoding

The name of `GreedyEmbeddingHelper` comes from the fact that it utilizes a greedy decoding method and incorporates an embedding matrix.

The greedy decoding method means that the output at each decoder time step (also the input into the next time step) is just the highest probability vocabulary word at the time step. The output is converted into a word embedding prior to being used as input for the next time step.

Below we create a `GreedyEmbeddingHelper` with a vocabulary size of 598 and a batch size of 8.

```
import tensorflow as tf

vocab_size = 598
sos = vocab_size
eos = vocab_size + 1
extended_vocab_size = vocab_size + 2

# Placeholder representing the embedding matrix for the vocab
# Embedding dim is 10
embedding_matrix = tf.placeholder(
    tf.float32, shape=(600, 10)
)

batch_size = 8
start_tokens = tf.tile([sos], [batch_size])
helper = tf.contrib.seq2seq.GreedyEmbeddingHelper(
    embedding_matrix, start_tokens, eos)
```

The `GreedyEmbeddingHelper` is initialized with an embedding matrix, a `batch_size` length tensor of SOS tokens, and the EOS token. In the example above, we use the `tf.tile` function to create a length 8 tensor completely filled with the SOS token.

The (embedded) SOS token is always the initial decoder input token. It's used to give the decoder a starting point, and it marks the beginning of the decoded output. The EOS token marks the end of a decoded sequence. When the decoder returns the EOS token at some time step, the decoding process is terminated.

C. Embedding variable scope

When using feature columns to create embeddings (e.g. the `get_embeddings` function on line 35 in the code), we don't have direct access to the embedding matrix variable. This means we need to create an embedding matrix variable with the exact same name as the one used during training. To do this, we need to create the variable in the appropriate *variable scope*.

A variable scope is similar to directory paths in an operating system. We may know the name of the file (in this case the variable), but to retrieve it we have to be in the proper directory (the variable scope). We can also create a variable under a certain variable scope similar to how we would create files in certain directories.

We specify variable scopes in TensorFlow using the `tf.variable_scope` function. You'll notice that `tf.variable_scope` is used in the `get_embeddings` function.

```
import tensorflow as tf

with tf.variable_scope('scope1'):
    v1 = tf.get_variable('var', shape=(2, 2))

with tf.variable_scope('scope2'):
    v2 = tf.get_variable('var', shape=(2, 2))
```

The above code block creates variables named `'var'` inside scopes `'scope1'` and `scope2'`. Despite the shared name, these variables are different, since the scopes differ.

We can also use nested scopes to create a variable. A nested scope is similar to putting a file in subdirectories of outer directories.

```
import tensorflow as tf

with tf.variable_scope('s'):
    with tf.variable_scope('sub1'):
        v1 = tf.get_variable('v', shape=(2, 2))
    with tf.variable_scope('sub2'):
        v2 = tf.get_variable('v', shape=(2, 2))

print(v1)
print(v1)
print(v2)
```

Mimicking a nested scope using a single scope is also possible. We just need to use the appropriate "path" when specifying the scope name.

```
import tensorflow as tf

with tf.variable_scope('s/sub1'):
    v1 = tf.get_variable('v', shape=(2, 2))

print(v1)
```

The `tf.variable_scope` function also has a useful `reuse` keyword argument. While it's not used in the code for this course, it sets the argument with value `tf.AUTO_REUSE`, in order to re-use previously created variables.

We don't use it in the encoder-decoder model because the training and inference portions of the code are separate (i.e. we don't create the same embedding matrix twice in one model run).

D. Inference Decoding Translation Example

The code widget below uses inference decoding to translate english sentences into French. Type a sentence into the text box and it will print out the French translation.

```
translate()
```

Enter text here

```
Press Run To Execute
```

Time to Code!

In this chapter you'll be completing the `create_decoder_helper` function from the **Training Helper** chapter. This time, you'll be creating the Helper object for inference.

We've already filled in part of the code to obtain the decoder's embedding matrix and initialize the start and end tokens. Your job now is to create the `GreedyEmbeddingHelper` object for inference decoding.

Inside the `else` block of the `create_decoder_helper` function, set `helper` equal to `tf_s2s.GreedyEmbeddingHelper` applied with `embedding_weights`, `start_tokens`, and `end_token` as the three input arguments.

The `create_decoder_helper` function returns a tuple containing the helper object and ground truth sequence lengths. However, in inference mode there are no ground truth sequences. In this case, we just set `dec_seq_lens` equal to `None`.

Inside the `else` block, set `dec_seq_lens` equal to `None`.

```python
import tensorflow as tf
tf_fc = tf.contrib.feature_column
tf_s2s = tf.contrib.seq2seq

# Seq2seq model
class Seq2SeqModel(object):
    def __init__(self, vocab_size, num_lstm_layers, num_lstm_units):
        self.vocab_size = vocab_size
        # Extended vocabulary includes start, stop token
        self.extended_vocab_size = vocab_size + 2
        self.num_lstm_layers = num_lstm_layers
        self.num_lstm_units = num_lstm_units
        self.tokenizer = tf.keras.preprocessing.text.Tokenizer(
            num_words=vocab_size)

    # Create the helper for decoding
    def create_decoder_helper(self, decoder_inputs, is_training, batch_size):
        if is_training:
            # ALREADY COMPLETED IN TRAINING HELPER CHAPTER
            pass
        else:
            DEC_EMB_SCOPE = 'decoder_emb/sequence_input_layer/sequences_embedding'
            with tf.variable_scope(DEC_EMB_SCOPE):
                embedding_weights = tf.get_variable(
                    'embedding_weights',
                    shape=(self.extended_vocab_size, int(self.extended_vocab_size**0.25)))
            start_tokens = tf.tile([self.vocab_size], [batch_size])
            end_token = self.vocab_size + 1
            # CODE HERE
        return helper, dec_seq_lens
```