Decoder Object

Learn about the decoder object for the encoder-decoder model.

Chapter Goals:

- Convert the encoder's final state into the proper format for decoding with attention
- Create a `BasicDecoder` object to use for decoding

A. Creating the initial state

The final state from the encoder is a tuple containing an `LSTMStateTuple` object for each layer of the BiLSTM. However, if we want to use this as the initial state for an attention-wrapped decoder, we need to convert it into an `AttentionWrapperState`.

The conversion is a two step process. We first use the `zero_state` function of the attention-wrapped decoder cell to create a blank `AttentionWrapperState`. We then use the `clone` function to copy the encoder's final state into the blank `AttentionWrapperState`. The result is the initial state for the attention-wrapped decoder.

Below we demonstrate how to create the decoder's initial state from the encoder's final state.

```
import tensorflow as tf

batch_size = 10
# dec_cell is an attention-wrapped decoder LSTM cell
zero_cell = dec_cell.zero_state(batch_size, tf.float32)

# final_state is the final encoder state of a 2-layer BiLSTM
initial_state = zero_cell.clone(cell_state=final_state)
```

The `zero_state` function takes the batch size and the state's type (normally

`tf.float32`) as required arguments. We set the `clone` function's `cell_state`

keyword argument with the encoder's final state to create the decoder's initial state.

B. The BasicDecoder object

The decoder object that we use for decoding is the `BasicDecoder`. To create an instance of the `BasicDecoder` object, we need to pass in the decoder cell, helper object, and initial decoder state as required arguments.

```
import tensorflow as tf

decoder = tf.contrib.seq2seq.BasicDecoder(
    dec_cell, helper, initial_state)
```

Creating a BasicDecoder object.

The `BasicDecoder` constructor has a keyword argument called `output_layer`, which can be used to apply a fully-connected layer to the model's outputs. This is a nice shortcut when calculating the model's logits.

```
import tensorflow as tf

num_units = 500 # extended vocab size
projection_layer = tf.layers.Dense(num_units)
decoder = tf.contrib.seq2seq.BasicDecoder(
    dec_cell, helper, initial_state,
    output_layer=projection_layer)
```

Creating a BasicDecoder object with direct logit calculation.

Instead of using the `tf.layers.dense` function, we create a fully-connected layer object using the `tf.layers.Dense` constructor. The required argument for initialization is the number of hidden units. In the example above, we set it equal to the extended vocabulary size, which gives us the proper shape for the logits.

Time to Code!

In this chapter you'll be completing the `create_basic_decoder` function, which is used in the model's `decoder` function to create a `BasicDecoder` object from a decoder LSTM cell and Helper object.

When creating the decoder object, we'll apply a projection layer to the end, which will calculate the model's logits directly.

Set `projection_layer` equal to `tf.layers.Dense` initialized with `extended_vocab_size`.

Before we can create the decoder object, we need to make sure that the initial state for the decoder is in the correct format.

We'll use the `zero_state` function to create a blank `AttentionWrapperState`, then use the `clone` function to copy the `final_state` contents into the blank `AttentionWrapperState`.

Set `zero_cell` equal to `dec_cell.zero_state` applied with `batch_size` and `tf.float32` as the required arguments.

Set `initial_state` equal to `zero_cell.clone` applied with `batch_size` as the `cell_state` keyword argument.

We can now create the decoder object using the `BasicDecoder` constructor. We'll use `dec_cell`, `helper`, and `initial_state` as the required arguments for initialization.

Set `decoder` equal to `tf_s2s.BasicDecoder` initialized with the specified required arguments as well as `projection_layer` for the `output_layer` keyword argument. Then return `decoder`.

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

def create_basic_decoder(extended_vocab_size, batch_size, final_state, dec_cell, helper):
    # CODE HERE
    pass

# 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)

def make_lstm_cell(self, dropout_keep_prob, num_units):
    cell = tf.nn.rnn_cell.LSTMCell(num_units)
    return tf.nn.rnn_cell.DropoutWrapper(cell, output_keep_prob=dropout_keep_prob)

def stacked_lstm_cells(self, is_training, num_units):
    dropout_keep_prob = 0.5 if is_training else 1.0
    cell_list = [self.make_lstm_cell(dropout_keep_prob, num_units) for i in range(self.nu
    cell = tf.nn.rnn_cell.MultiRNNCell(cell_list)
    return cell

# Helper funtion to combine BiLSTM encoder outputs
def combine_enc_outputs(self, enc_outputs):
    enc_outputs_fw, enc_outputs_bw = enc_outputs
    return tf.concat([enc_outputs_fw, enc_outputs_bw], -1)

# Create the stacked LSTM cells for the decoder
def create_decoder_cell(self, enc_outputs, input_seq_lens, is_training):
    num_decode_units = self.num_lstm_units * 2
    dec_cell = self.stacked_lstm_cells(is_training, num_decode_units)
    combined_enc_outputs = self.combine_enc_outputs(enc_outputs)
    attention_mechanism = tf_s2s.LuongAttention(
        num_decode_units, combined_enc_outputs,
        memory_sequence_length=input_seq_lens)
    dec_cell = tf_s2s.AttentionWrapper(
        dec_cell, attention_mechanism,
        attention_layer_size=num_decode_units)
    return dec_cell

# Create the helper for decoding
def create_decoder_helper(self, decoder_inputs, is_training, batch_size):
    if is_training:
        dec_embeddings, dec_seq_lens = self.get_embeddings(decoder_inputs, 'decoder_emb')
        helper = tf_s2s.TrainingHelper(
            dec_embeddings, dec_seq_lens)
    else:
        pass
    return helper, dec_seq_lens

# Create the decoder for the model
def decoder(self, enc_outputs, input_seq_lens, final_state, batch_size,
    decoder_inputs=None, maximum_iterations=None):
    is_training = decoder_inputs is not None
    dec_cell = self.create_decoder_cell(enc_outputs, input_seq_lens, is_training)
    helper, dec_seq_lens = self.create_decoder_helper(decoder_inputs, is_training, batch_
    decoder = create_basic_decoder(
        self.extended_vocab_size, batch_size, final_state, dec_cell, helper)
```