Decoding Output

Decode the model's outputs for training and inference.

Chapter Goals:

- Retrieve the decoder outputs and return the model's logits during training

After creating the decoder object for our model, we can perform the decoding using the `dynamic_decode` function.

```
import tensorflow as tf

extended_vocab_size = 500
batch_size = 10
# decoder is a BasicDecoder object
outputs = tf.contrib.seq2seq.dynamic_decode(decoder)

decoder_output = outputs[0]
logits = decoder_output.rnn_output
decoder_final_state = outputs[1]
decoded_sequence_lengths = outputs[2]
```

Using the dynamic_decode function for decoding and producing logits.

The `dynamic_decode` function takes in one required argument, which is the decoder object. It returns a tuple containing three elements:

1. The decoder's output. For a `BasicDecoder` input, the decoder's output takes the form of a `BasicDecoderOutput` object.
2. The decoder's final state. This isn't used in our encoder-decoder model.
3. The lengths of each of the decoder's output sequences. This also isn't used in our encoder-decoder model.

If the `BasicDecoder` input was initialized with the `output_layer` keyword argument, the `rnn_output` of the `BasicDecoderOutput` object will be the model's logits

B. Limiting the decoded length

A problem that sometimes occurs when decoding, especially in tasks like text summarization, is the decoder returning output sequences that are too long. We can manually limit the decoder output length with the `dynamic_decode` function’s `maximum_iterations` keyword argument.

By setting this keyword argument with an integer, k, we guarantee that the decoder will not output sequences with length longer than k.

Time to Code!

In this chapter you'll be completing the `run_decoder` function, which is used in the model's `decoder` function to run the decoder object.

We can run the decoder object using the `dynamic_decode` function. We only care about the first element in the returned tuple, which contains the output of our decoder.

Set `dec_outputs` equal to the first element of the tuple returned by `tf_s2s.dynamic_decode`. Call the function with `decoder` as the required argument and `maximum_iterations` for the `maximum_iterations` keyword argument.

During training, the model's logits are located in the `rnn_output` property of `dec_outputs`. The `decoder` function will return both the model's logits and the ground truth sequence lengths.

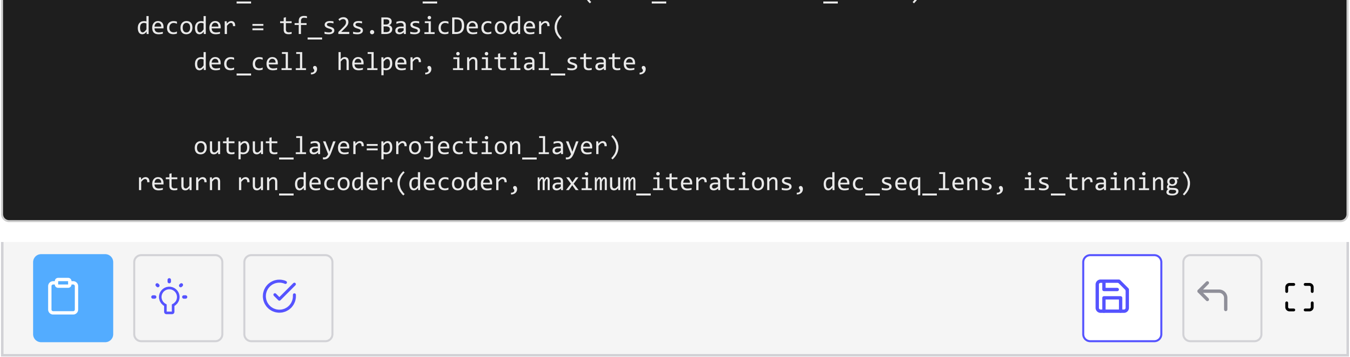
Create an `if` block that checks if `is_training` is `True`. Inside the `if` block, return a tuple containing the model's logits and `dec_seq_lens`, in that order.

When we're not training, we will return the model's predictions, which is in the `sample_id` attribute of `dec_outputs` (more on this later).

Outside the `if` block, return `dec_outputs.sample_id`.

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

```
def run_decoder(decoder, maximum_iterations, dec_seq_lens, is_training):
    # 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_
        projection_layer = tf.layers.Dense(self.extended_vocab_size)
        zero_cell = dec_cell.zero_state(batch_size, tf.float32)
        initial_state = zero_cell.clone(cell_state=final_state)
```



```
        decoder = tf_s2s.BasicDecoder(
            dec_cell, helper, initial_state,

            output_layer=projection_layer)
        return run_decoder(decoder, maximum_iterations, dec_seq_lens, is_training)
```