LSTM Output

Run an LSTM model on input sequences and retrieve the output.

Chapter Goals:

- Compute the output of your LSTM model

A. TensorFlow implementation

In TensorFlow, the way we create and run an RNN is with the function `tf.nn.dynamic_rnn`. The function takes in two required arguments. The first is the cell object that is used to create the RNN (e.g. an `LSTMCell`, `MultiRNNCell`, etc.). The second is the batch of input sequences, which are usually first converted to word embedding sequences.

Of the keyword arguments for the function, it is required that either `initial_state` or `dtype` is set. The `initial_state` argument specifies the starting state for the input cell object. We'll use this argument in later parts of this section.

The `dtype` argument specifies the type of both the initial cell state and RNN output. Most of the time, we can just set this argument to `tf.float32`, since the RNN outputs are normally floating point numbers.

Below is an example demonstrating how to use `tf.nn.dynamic_rnn`. Note that the input sequences have maximum length 10 and embedding size 12.

```
import tensorflow as tf
cell = tf.nn.rnn_cell.LSTMCell(7)
# Input sequences for the LSTM
# Shape: (batch_size, time_steps, embed_dim)
input_sequences = tf.placeholder(
    tf.float32,
    shape=(None, 10, 20)
)
output, final_state = tf.nn.dynamic_rnn(
    cell,
    input_sequences,
    dtype=tf.float32
)
```

The `tf.nn.dynamic_rnn` function returns a tuple containing the RNN outputs as well as the final state of the RNN. For now, we only need to focus on the RNN output.

You'll notice from the example that the output first and second dimensions are equal to the input batch. This is because the RNN calculates the output for each time step of each sequence in the input batch. The third dimension, however, is equal to the number of hidden units in the cell object. For RNNs with multiple cells (i.e. `MultiRNNCell` cell object), the third dimension is equal to the number of hidden units in the final cell.

B. Sequence lengths

Because each of the input sequences can have varying lengths, it is likely that many of them will contain padding. Since padding is essentially a sequence filler, and therefore adds nothing of value to the RNN, we don't really want the RNN to waste computation on the padded parts of a sequence.

Instead, we can use the `sequence_length` argument in `tf.nn.dynamic_rnn`. This argument takes in a 1-D integer tensor, specifying the non-padded lengths of each sequence in the input batch.

Below is an example that uses `sequence_length` in `tf.nn.dynmaic_rnn`. The `cell` and `input_sequences` variables are the same as ones from the previous example. In this case, the input batch size is 5.

```
import tensorflow as tf
lens = [4, 9, 10, 5, 10]
cell = tf.nn.rnn_cell.LSTMCell(7)
input_sequences = tf.placeholder(
    tf.float32,
    shape=(None, 10, 20)
)
output, final_state = tf.nn.dynamic_rnn(
    cell,
    input_sequences,
    sequence_length=lens,
    dtype=tf.float32
)
```

By using the `sequence_length` argument, the function can skip unnecessary computation for the padded parts of each sequence, which can greatly reduce training time.

Time to Code!

In this chapter, you'll be completing the `run_lstm` function, which runs the LSTM model on input sequences.

As you can see, the input sequences have already been converted into embeddings, and the sequence lengths have been calculated. What's left for you to do is call the `tf.nn.dynamic_rnn` function to run our LSTM. We'll use `sequence_lengths` for the function's `sequence_length` argument to speed up the computation. We should also set the `dtype` argument to `tf.float32`.

Set `lstm_outputs` equal to the first element of the tuple returned by `tf.nn.dynamic_rnn`, with `cell` and `input_embeddings` as the required arguments to the function. Also use the keyword arguments specified above.

Return a tuple containing `lstm_outputs` as the first element and `binary_sequences` as the second element.

```
import tensorflow as tf

# LSTM Language Model
class LanguageModel(object):
    # Model Initialization
    def __init__(self, vocab_size, max_length, num_lstm_units, num_lstm_layers):
        self.vocab_size = vocab_size
        self.max_length = max_length
        self.num_lstm_units = num_lstm_units
        self.num_lstm_layers = num_lstm_layers
        self.tokenizer = tf.keras.preprocessing.text.Tokenizer(num_words=vocab_size)

    # Create a cell for the LSTM
    def make_lstm_cell(self, dropout_keep_prob):
        cell = tf.nn.rnn_cell.LSTMCell(self.num_lstm_units)
        return tf.nn.rnn_cell.DropoutWrapper(
            cell, output_keep_prob=dropout_keep_prob)

    # Stack multiple layers for the LSTM
    def stacked_lstm_cells(self, is_training):
        dropout_keep_prob = 0.5 if is_training else 1.0
        cell_list = [self.make_lstm_cell(dropout_keep_prob) for i in range(self.num_lstm_laye
        cell = tf.nn.rnn_cell.MultiRNNCell(cell_list)
        return cell

    # Convert input sequences to embeddings

    def get_input_embeddings(self, input_sequences):
        embedding_dim = int(self.vocab_size**0.25)
        initial_bounds = 0.5 / embedding_dim
        initializer = tf.random_uniform(
            [self.vocab_size, embedding_dim],
            minval=-initial_bounds,
            maxval=initial_bounds)
        self.input_embedding_matrix = tf.get_variable('input_embedding_matrix',
            initializer=initializer)
        input_embeddings = tf.nn.embedding_lookup(self.input_embedding_matrix, input_sequence
        return input_embeddings

    # Run the LSTM on the input sequences
    def run_lstm(self, input_sequences, is_training):
        cell = self.stacked_lstm_cells(is_training)
        input_embeddings = self.get_input_embeddings(input_sequences)
        binary_sequences = tf.sign(input_sequences)
        sequence_lengths = tf.reduce_sum(binary_sequences, axis=1)
        # CODE HERE
```