Bidirectional LSTM

Create and run a bidirectional LSTM model.

Chapter Goals:

- Learn about the bidirectional LSTM and why it's used

A. Forwards and backwards

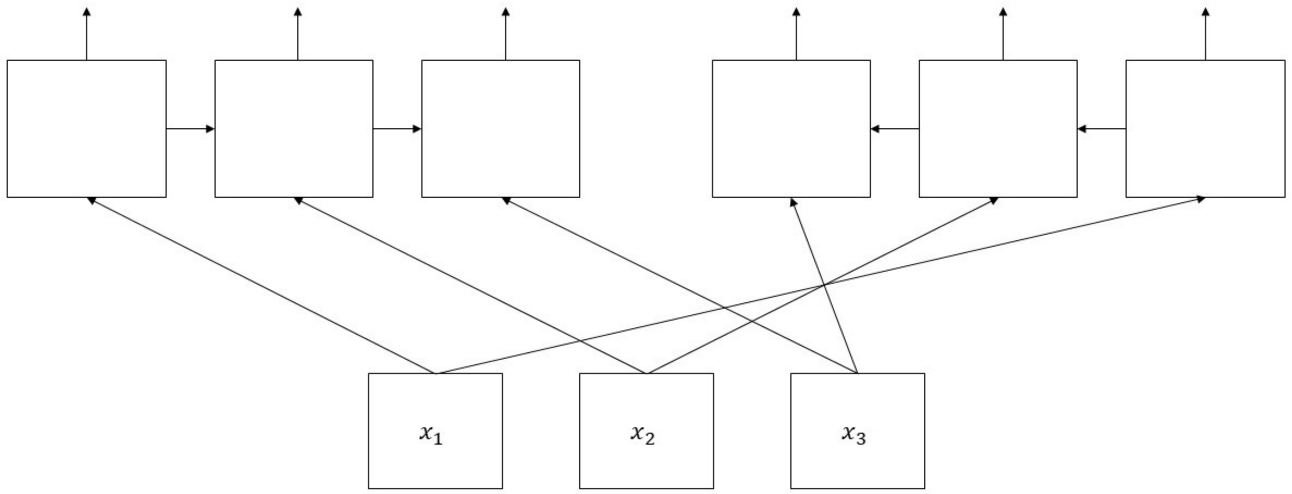
The language model from the **Language Model** section of this course used a regular LSTM which read each input sequence in the *forwards* direction. This meant that the recurrent connections went in the left-right direction, i.e. from time step t to time step $t + 1$.

While regular LSTMs work well for most NLP tasks, they are not always the best option. Specifically, when we have access to a completed text sequence (e.g. text classification), it may be beneficial to look at the sequence in both the forwards and *backwards* directions.

By looking at a sequence in both directions, the model can take into account both the past and future context of words and phrases in the text sequence, which allows it to obtain a better understanding of the text sequence.

B. Bidirectional LSTM structure

The way we look at an input sequence in both directions is with a model called a *bidirectional LSTM*, or BiLSTM for short. The model architecture is incredibly simple: it consists of a regular forwards LSTM and a backwards LSTM, which reads the input sequence in reverse.



The above diagram shows a (unrolled) BiLSTM with 3 time steps. On the left is the forwards LSTM and on the right is the backwards LSTM. The sequence $[x_1, x_2, x_3]$ represents an input (embedded) sequence.

Note: It is also possible to create a bidirectional RNN with general RNN cells rather than LSTM cells. However, since this Lab focuses on LSTM cells, we'll continue using the BiLSTM variant.

C. BiLSTM in TensorFlow

In TensorFlow, we can create and run a BiLSTM using the `tf.nn.bidirectional_dynamic_rnn` function. This function is very similar to the `tf.nn.dynamic_rnn` function for regular LSTMs, with the main difference being that it takes in two LSTM cells rather than one.

The code below shows example usage of `tf.nn.bidirectional_dynamic_rnn`. The `cell_fw` and `cell_bw` variables represent the forwards and backwards LSTM cells, respectively.

```
import tensorflow as tf
cell_fw = tf.nn.rnn_cell.LSTMCell(7)
cell_bw = tf.nn.rnn_cell.LSTMCell(7)

# Embedded input sequences
# Shape: (batch_size, time_steps, embed_dim)
input_embeddings = tf.placeholder(
    tf.float32, shape=(None, 10, 12))
outputs, final_states = tf.nn.bidirectional_dynamic_rnn(
    cell_fw,
    cell_bw,
    input_embeddings
```

```
    input_embeddings,
    dtype=tf.float32)
print(outputs[0])
print(outputs[1])
```

The `tf.nn.bidirectional_dynamic_rnn` function returns a tuple containing the LSTM outputs and the final LSTM states. Since a BiLSTM contains two LSTMs, both `outputs` and `final_states` shown in the example are tuples. We won't worry about `final_states` for now.

However, note that `outputs[0]` represents the outputs of the forwards LSTM while `outputs[1]` represents the outputs of the backwards LSTM. This is important for calculating the model's logits (which we'll do in the next chapter).

Time to Code!

In this chapter you'll be completing the `run_bilstm` function, which runs a bidirectional LSTM on input sequences.

The function has already been filled with code that converts the sequences to embeddings and uses the `make_lstm_cell` function to create the directional LSTM cells. Your task is to use the `tf.nn.bidirectional_dynamic_rnn` function to run the BiLSTM.

We only need to use the first element of the returned tuple from running the BiLSTM. We also use `sequence_lengths` and `tf.float32` for the `sequence_length` and `dtype` keyword arguments, respectively.

Set `lstm_outputs` equal to the first element of the tuple returned by `tf.nn.bidirectional_dynamic_rnn`. Use `cell_fw`, `cell_bw`, and `input_embeddings` as the required arguments and also use the keyword arguments specified above.

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

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

# Text classification model
```

```
# Text classification model
class ClassificationModel(object):
    # Model initialization
    def __init__(self, vocab_size, max_length, num_lstm_units):
        self.vocab_size = vocab_size
        self.max_length = max_length
        self.num_lstm_units = num_lstm_units
        self.tokenizer = tf.keras.preprocessing.text.Tokenizer(num_words=self.vocab_size)

    # Make LSTM cell with dropout
    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)

    # Use feature columns to create input embeddings
    def get_input_embeddings(self, input_sequences):
        inputs_column = tf_fc.sequence_categorical_column_with_identity(
            'inputs',
            self.vocab_size)
        embedding_column = tf.feature_column.embedding_column(
            inputs_column,
            int(self.vocab_size**0.25))
        inputs_dict = {'inputs': input_sequences}
        input_embeddings, sequence_lengths = tf_fc.sequence_input_layer(
            inputs_dict,
            [embedding_column])
        return input_embeddings, sequence_lengths

    # Create and run a BiLSTM on the input sequences
    def run_bilstm(self, input_sequences, is_training):
        input_embeddings, sequence_lengths = self.get_input_embeddings(input_sequences)
        dropout_keep_prob = 0.5 if is_training else 1.0
        cell_fw = self.make_lstm_cell(dropout_keep_prob)
        cell_bw = self.make_lstm_cell(dropout_keep_prob)
        # CODE HERE
```