Attention

Learn about the attention mechanism and why it's important.

Chapter Goals:

- Learn about attention and understand why it's useful
- Incorporate attention into the decoder LSTM

A. Using the encoder

Based on the encoder-decoder model architecture, the only thing that the decoder gets from the encoder is the final state in each layer. The final states basically encapsulate the encoder's extracted information from the input sequence, which is passed into the decoder.

However, trying to encapsulate all the useful information from an input sequence into a final state is a difficult task, especially if the input sequence is large and contains long-term dependencies. This is a problem that has been shown to exist in practice, where decoders perform poorly on input sequences with long-term dependencies.

> **"Sam grew up in Los Angeles. As a child, he dreamed of one day becoming an actor like Brad Pitt or Johnny Depp. Each day he would practice public speaking and impromptu skits near Venice Beach."**

An input text sequence with many long-term dependencies. A regular encoder-decoder model would struggle to decode this input, e.g. for a task like translation.

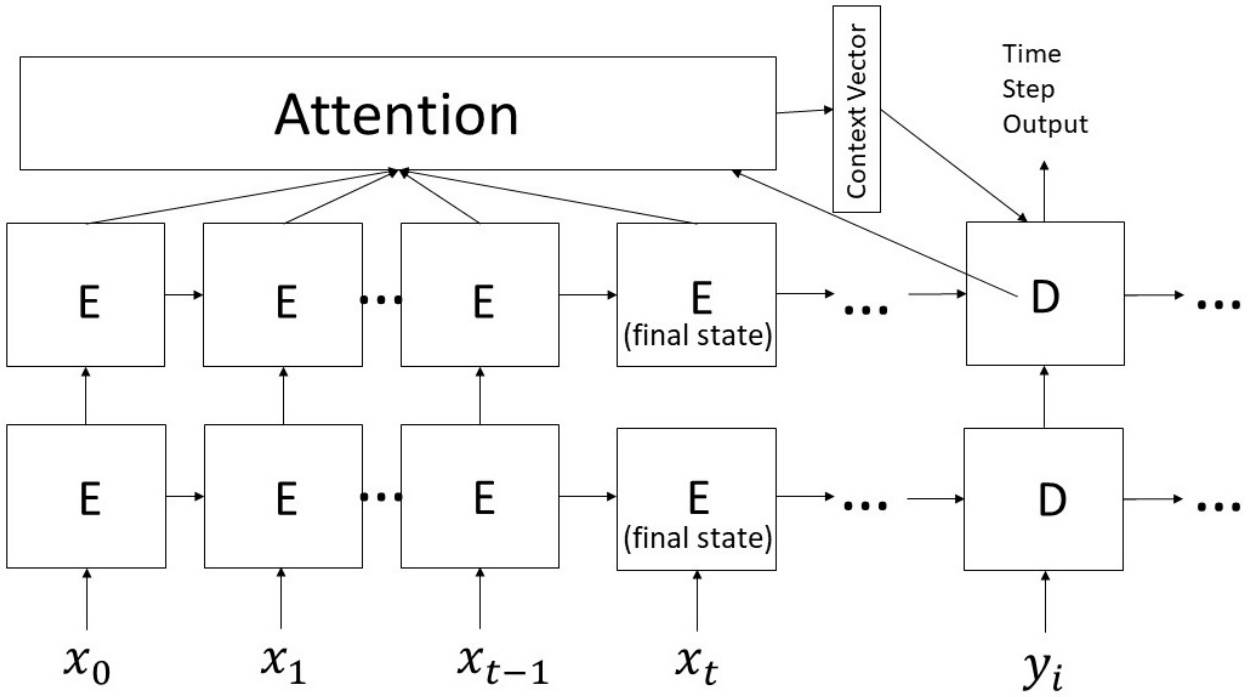
The obvious solution to this issue is to give the decoder access to each of the encoder's intermediate time step outputs. In the previous chapter's diagram, the encoder's outputs were not used. However, if we were to use the encoder's outputs as additional input for the decoder, it would give the decoder a lot more useful information about the input sequence. The way we do this is by using *attention*.

B. How attention works

B. How attention works

Although we want to use the input sequence for the decoder, we don't necessarily want to use each input token equally. With attention, we let the decoder decide which encoder outputs are most useful for the decoder at the current decoding time step.

Using the decoder's hidden state at the current time step, as well as the encoder outputs, attention will calculate something called a *context vector*. The context vector encapsulates the most meaningful information from the input sequence for the current decoder time step, and it's used as additional input for the decoder when calculating the time step's output.



Using attention to calculate a context vector for the current decoder time step (time step i). The context vector is used to obtain the decoder output at time step i.

Attention makes use of trainable weights to calculate a context vector. You can think of attention as a mini neural network, which takes as input the decoder's current state and the encoder outputs, and uses its trainable weights to produce a context vector.

C. Attention mechanisms

The exact process for computing the context vector depends on the *attention mechanism* that's used. There are quite a few attention mechanism variations, but the two most popular ones in TensorFlow are `BahdanauAttention` and

`LuongAttention`. These two attention mechanisms are named after their main inventors, Dzmitry Bahdanau and Minh-Thang Luong, respectively.

The main difference between the two mechanisms is how they combine the encoder outputs and current time step hidden state when computing the context vector. The Bahdanau mechanism uses an *additive* (concatenation-based) method, while the Luong mechanism uses a multiplicative method.

In our code we use the Luong mechanism because it has been shown to work better in certain seq2seq tasks, such as machine translation. However, both TensorFlow mechanisms are initialized in nearly identical fashion.

Below we show how to initialize the two attention mechanisms as TensorFlow objects.

```
import tensorflow as tf

# Placeholder representing the
# individual lengths of each input sequence in the batch
input_seq_lens = tf.placeholder(tf.int32, shape=(None,))

num_units = 8
bahdanau = tf.contrib.seq2seq.BahdanauAttention(
    num_units,
    # combined encoder outputs (from previous chapter)
    combined_enc_outputs,
    memory_sequence_length=input_seq_lens)
luong = tf.contrib.seq2seq.LuongAttention(
    num_units,
    # combined encoder outputs (from previous chapter)
    combined_enc_outputs,
    memory_sequence_length=input_seq_lens)
```

The first required argument for the attention mechanisms is just the number of hidden units in the encoder outputs, i.e. the final dimension size. The second required argument is the encoder outputs. The `memory_sequence_length` keyword argument is a tensor representing the length of each input sequence, and it's used to ensure that attention is only applied at non-padding time steps.

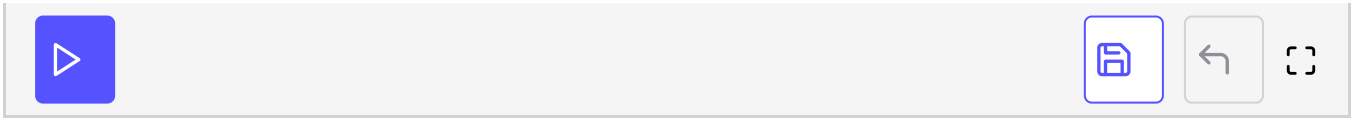
D. TensorFlow AttentionWrapper

While the purpose and high-level usage of attention is easy enough to understand, the implementation requires quite a bit of linear algebra and

advanced mathematics. Luckily, TensorFlow gives us an easy-to-use API for adding attention to an LSTM decoder cell via the `AttentionWrapper` function. Below is an example of how to use the `AttentionWrapper` function to add attention to the decoder cell.

```
import tensorflow as tf

# Decoder LSTM cell
dec_cell = tf.nn.rnn_cell.LSTMCell(8)
dec_cell = tf.contrib.seq2seq.AttentionWrapper(
    dec_cell,
    luong, # LuongAttention object
    attention_layer_size=8)
```



The required arguments for the `AttentionWrapper` function are the decoder cell and the attention mechanism to use. Of the keyword arguments, the one we use in the code is `attention_layer_size`.

When using the decoder in TensorFlow, we pass along the *attention value* at each decoder time step into the cell state at the next time step. The default behavior is to just pass along the context vector as the attention value. However, if we set the `attention_layer_size` or `attention_layer` keyword arguments, a fully-connected layer will combine the context vector with the time step's output. The output of the fully-connected layer is used as the attention value.

Using a fully-connected layer to create the attention value can benefit the model's performance, by utilizing the decoder's outputs as additional information. The value used for `attention_layer_size` specifies the number of hidden units in the fully-connected layer. In the example above, we set it equal to the number of decoder hidden units.

Time to Code!

In this chapter you'll be completing the `create_decoder_cell` function to use attention.

The attention mechanism we'll use for the decoder is `LuongAttention`. Note that we provide a shorthand variable for the `tf.contrib.seq2seq` module, called `tf.c2s`.

called `tf_s2s`.

Set `attention_mechanism` equal to `tf_s2s.LuongAttention` applied with `num_decode_units` and `combined_enc_outputs` as the required arguments, along with `input_seq_lens` as the `memory_sequence_length` keyword argument.

Using the attention mechanism, we can apply an `AttentionWrapper` around the `dec_cell` variable initialized in the previous chapter.

Set `dec_cell` equal to `tf_s2s.AttentionWrapper` applied with `dec_cell` and `attention_mechanism` as the required arguments, along with `num_decode_units` as the `attention_layer_size` keyword argument.

After wrapping the decoder cell with the attention mechanism, we return the cell.

Return `dec_cell`.

```
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)

    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)

    # Create multi-layer LSTM cells
    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):
```

```
    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)
        # CODE HERE
```