**Sequence to sequence in Tensorflow**

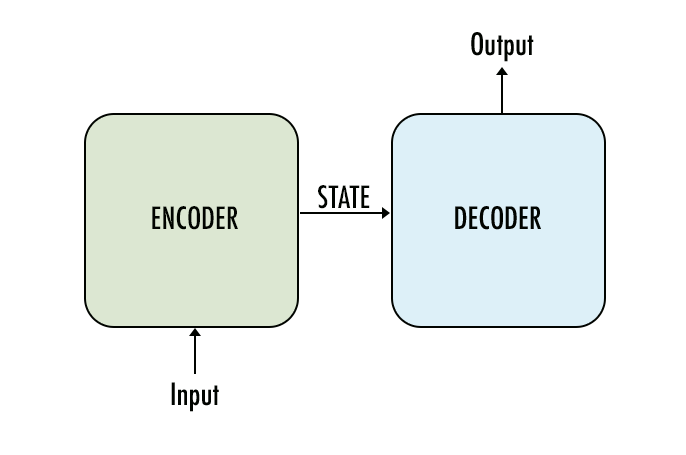
Tensorflow has a bunch of APIs that help you build a sequence to sequence model (also called a seq2seq model for short). It’s important to note that these APIs have changed at the end of 2016 (and in a way are still evolving). A lot of the tutorials you’ll find on the web for seq2seq in tensorflow use the now-deprecated tf.contrib.legacy\_seq2seq (which was previously called “tf.nn.seq2seq”).

The modules of note for seq2seq are:

* [**tf.nn**](https://www.tensorflow.org/api_guides/python/nn#Recurrent_Neural_Networks), which allows us to construct different kinds of RNNs
* [**tf.contrib.rnn**](https://www.tensorflow.org/api_guides/python/contrib.rnn), which defines a number of RNN cells (an RNN cell is a required parameter for the RNNs defined in tf.nn).
* [**tf.contrib.seq2seq**](https://www.tensorflow.org/api_docs/python/tf/contrib/seq2seq), which contains seq2seq decoders and loss operations.

**The Main Components**

If we take a high-level view, a seq2seq model has these main components:



* Encoder: this is a **[tf.nn.dynamic\_rnn](https://www.tensorflow.org/api_docs/python/tf/nn/dynamic_rnn" \t "_blank)**.
* Decoder: this is a [**tf.contrib.seq2seq.dynamic\_rnn\_decoder**](https://www.tensorflow.org/versions/r1.0/api_docs/python/tf/contrib/seq2seq/dynamic_rnn_decoder)

**tf.nn.dynamic\_rnn**

Read the documentation of **[tf.nn.dynamic\_rnn](https://www.tensorflow.org/api_docs/python/tf/nn/dynamic_rnn" \t "_blank)**. What inputs will we pass into this function?

**Your reflection**

**an instance of the RNN cell, the inputs**

**Things to think about**

Reading the documentation for tf.nn.dynamic\_rnn, you'll see tf.nn.dynamic\_rnn(cell, inputs, sequence\_length=None, initial\_state=None, dtype=None, parallel\_iterations=None, swap\_memory=False, time\_major=False, scope=None)

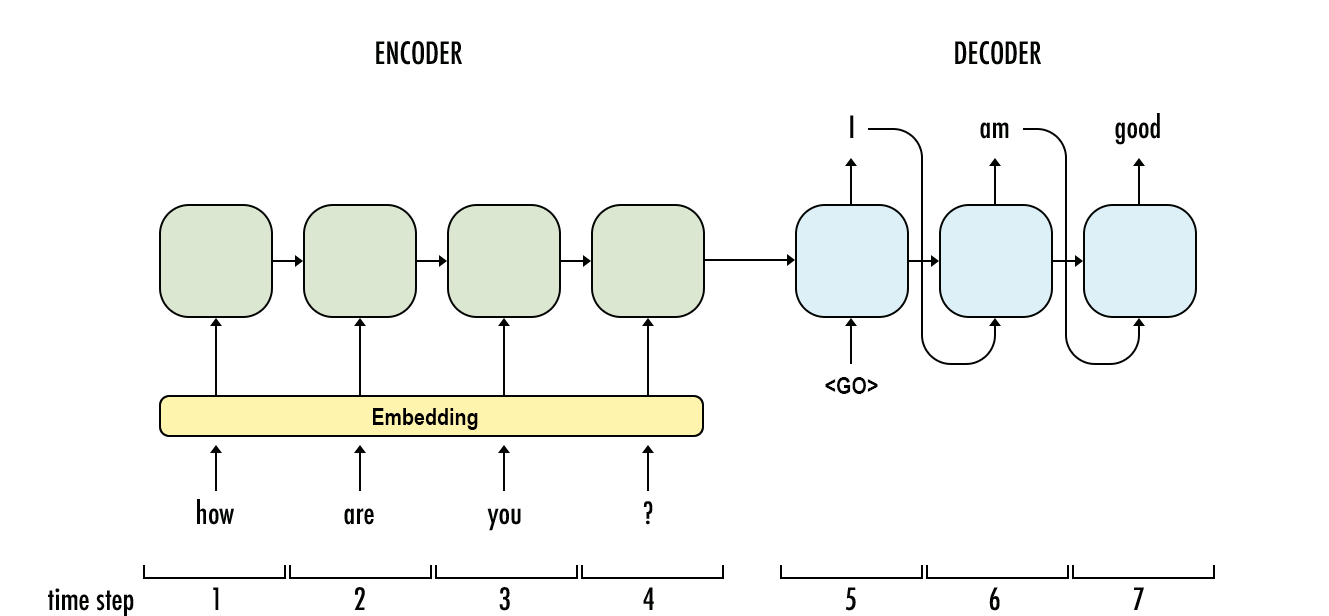
So you need to at least pass in the RNN cell you built (for example tf.contrib.rnn.BasicLSTMCell). You'll also need to give it the inputs tensor, which in this case is the input text data, typically coming from the embedding layer. I also typically pass in an initial\_state which you've seen in the previous RNN lessons.

**seq2seq module**

Read [**https://www.tensorflow.org/api\_docs/python/tf/contrib/seq2seq**](https://www.tensorflow.org/api_docs/python/tf/contrib/seq2seq) to understand its major component. You can ignore everything with “attention” for the time being.

**Inputs**

When we want our model to do inference, we need to feed it an input sequence. Let's assume we're building a chatbot, and that our sequences will be words. We need to first convert the words to a proper numeric representation that the network can use for its computations. This conversion is done using **[tf.nn.embedding\_lookup](https://www.tensorflow.org/api_docs/python/tf/nn/embedding_lookup" \t "_blank)** which we can use (after some processing of our data) to turn the words into vectors.



These models work a lot better if we feed the decoder our target sequence regardless of what its timesteps actually output in the training run. So unlike in the inference graph, we will not feed the output of the decoder to itself in the next timestep. Before the model trains on samples, the data needs to be preprocessed.

**Example 1:**

Let's assume we have only two examples in our dataset (example dialog from The Matrix):

| **source** | **target** |
| --- | --- |
| Can you fly that thing? | Not yet |
| Is Morpheus alive? | Is Morpheus still alive, Tank? |

In the preprocessing stage, say we decide it's not relevant for our bot to know names. So after tokenization, making things lower-case, and replacement of names with <UNK>, our dataset now looks like this:

| **Source** | **Target** |
| --- | --- |
| can you fly that thing ? | not yet |
| is <UNK> alive? | is <UNK> still alive , <UNK> ? |

Our input batch is shaping up

| **can** | **you** | **fly** | **that** | **thing** | **?** |
| --- | --- | --- | --- | --- | --- |
| is | <UNK> | alive | ? | <PAD> | <PAD> |

Because we are using embedding, we have to first compiled a "vocabulary" list containing all the words we want our model to be able to use or read. The model inputs will have to be tensors containing the IDs of the words in the sequence.

**Example 2:**

Say we want to train our model on this tiny dataset:

| **source** | **target** |
| --- | --- |
| How are you? | I am good |

Before we can even train the model, we have to first tokenize the dataset, do away with capitalization, then build a vocabulary of all the unique tokens. In our example, this vocabulary would look like this:

| **id** | **word** |
| --- | --- |
| 0 | how |
| 1 | are |
| 2 | you |
| 3 | ? |
| 4 | i |
| 5 | am |
| 6 | good |

There are four symbols, however, that we need our vocabulary to contain. Seq2seq vocabularies usually reserve the first four spots for these elements:

* **<PAD>**: During training, we'll need to feed our examples to the network in batches. The inputs in these batches all need to be the same width for the network to do its calculation. Our examples, however, are not of the same length. That's why we'll need to pad shorter inputs to bring them to the same width of the batch
* **<EOS>**: This is another necessity of batching as well, but more on the decoder side. It allows us to tell the decoder where a sentence ends, and it allows the decoder to indicate the same thing in its outputs as well.
* **<UNK>**: If you're training your model on real data, you'll find you can vastly improve the resource efficiency of your model by ignoring words that don't show up often enough in your vocabulary to warrant consideration. We replace those with <UNK>.
* **<GO>**: This is the input to the first time step of the decoder to let the decoder know when to start generating output.

Note: Other tags can be used to represent these functions. For example I've seen <s> and </s> used in place of <GO> and <EOS>. So make sure whatever you use is consistent through preprocessing, and model training/inference.

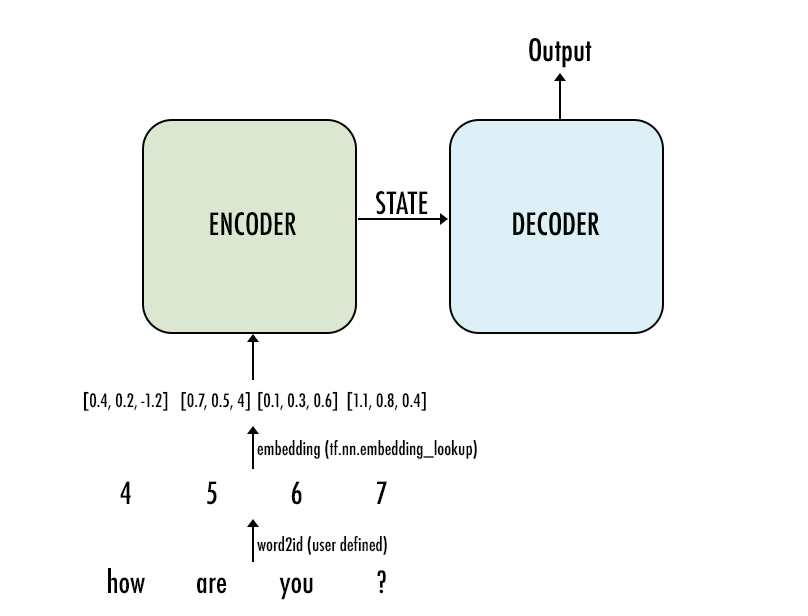
Let's go ahead and add them to the top of our vocabulary:

| **id** | **word** |
| --- | --- |
| 0 | <PAD> |
| 1 | <EOS> |
| 2 | <UNK> |
| 3 | <GO> |
| 4 | how |
| 5 | are |
| 6 | you |
| 7 | ? |
| 8 | i |
| 9 | am |
| 10 | good |

Now that we have established our vocabulary, we just replace the words with their ids, and that would be the input tensor into the encoder. So "how are you ?" becomes:

|  |  |  |  |
| --- | --- | --- | --- |
| 4 | 5 | 6 | 7 |

This is a way to look at the input for inference (where we set 3 as the embedding size, so each word would be represented by a vector of size 3):



**Training inputs**

Preparing the inputs for the training graph is a little more involved for two reasons:

* These models work a lot better if we feed the decoder our target sequence regardless of what its timesteps actually output in the training run. So unlike in the inference graph, we will not feed the output of the decoder to itself in the next timestep.
* Batching

**Example:**

Like before, let's assume we have only two examples in our dataset (example dialog from The Matrix):

| **source** | **target** |
| --- | --- |
| Can you fly that thing? | Not yet |
| Is Morpheus alive? | Is Morpheus still alive, Tank? |

In the preprocessing stage, say we decide it's not relevant for our bot to know names. So after tokenization, making things lower-case, and replacement of names with <UNK>, our dataset now looks like this:

| **Source** | **Target** |
| --- | --- |
| can you fly that thing ? | not yet |
| is <UNK> alive? | is <UNK> still alive , <UNK> ? |

Our input batch is shaping up

| **can** | **you** | **fly** | **that** | **thing** | **?** |
| --- | --- | --- | --- | --- | --- |
| is | <UNK> | alive | ? | <PAD> | <PAD> |

I created a manual vocabulary. But using it, the proper input tensor is now ready:

| **4** | **5** | **6** | **7** | **8** | **9** |
| --- | --- | --- | --- | --- | --- |
| 10 | 2 | 11 | 9 | 0 | 0 |

One of the original sequence to sequence papers, **[Sutskever et al. 2014](https://arxiv.org/abs/1409.3215" \t "_blank)**, reported better model performance if the inputs are reversed. So you may also choose to reverse the order of words in the input sequence.

Now let's look at our target input tensor

| **not** | **yet** |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| is | <UNK> | still | alive | , | <UNK> | ? |

Now we need to:

1. Add <GO> to the beginning
2. Add <EOS> to the end
3. Add padding

When we do that, it looks like this:

| **<GO>** | **not** | **yet** | **<EOS>** | **<PAD>** | **<PAD>** | **<PAD>** | **<PAD>** | **<PAD>** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| <GO> | is | <UNK> | still | alive | , | <UNK> | ? | <EOS> |

And so, our target input tensor emerges:

| **3** | **14** | **15** | **1** | **0** | **0** | **0** | **0** | **0** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | 10 | 2 | 12 | 11 | 13 | 2 | 9 | 1 |

Note: I'm showing this processing steps here to explain how the shape and values of the tensor. In practice, we stop using words much earlier in the process. During the preprocessing we do the following:

* we build our vocabulary of unique words (and count the occurrences while we're at it)
* we replace words with low frequency with <UNK>
* create a copy of conversations with the words replaced by their IDs
* we can choose to add the <GO> and <EOS> word ids to the target dataset now, or do it at training time