Recurrent Neural Networks

Jean-Martin Albert

August 30, 2017

A Bird's Eye View Of Classical Neural Networks

- A neural network is a fancy way to describe a class of functions $f : \mathbb{R}^n \to \mathbb{R}^m$ parametrized by $\mathbb{R}^\ell$.
- The layers of a NN represent function composition.
- If f is a function $F(\mathbf{x}, \mathbf{w}) : \mathbb{R}^n \times \mathbb{R}^\ell \to \mathbb{R}^n$ such that $\mathcal{C} = \{\mathbf{x} \mapsto F(\mathbf{x}, \mathbf{w}) : \mathbf{w} \in \mathbb{R}^n\}$.
- Each vector $\mathbf{w} \in \mathbb{R}^\ell$ is in some sense a code for an element of $\mathcal{C}$.
- When we are given a function $f : \mathbb{R}^n \to \mathbb{R}^m$, we try to find which element of $\mathcal{C}$ best represents f.

A Bird's Eye View Of Classical Neural Networks

- In tensorflow, the function we are trying to approximate is given in the form of a relation $f(\mathbf{x}) = \mathbf{y}$.
- We provide *placeholders* for the values of $\mathbf{x}$ and $\mathbf{y}$.
- The function $F(\mathbf{x}, \mathbf{w})$ is defined by a *computation graph*.
- For example, here is a linear regression, in which we try to approximate a function $f : \mathbb{R}^{10} \to \mathbb{R}^{15}$ using a function of the form $F(\mathbf{x}, \mathbf{Wb}) = \mathbf{Wx} + \mathbf{b}$. Here $\mathbf{W}$ is an 15×10 matrix, and $\mathbf{b} \in \mathbb{R}^{15}$

A Bird's Eye View Of Classical Neural Networks

```
x = tf.placeholder(shape=[None, 10])
y = tf.placeholder(shape=[None, 15])
W = tf.Variable(shape=[10, 15])
b = tf.Variable(shape=[1, 15])
F = tf.matmul(x, W) + b
D = tf.mean_square_error(y, F)
```

- In tensorflow (and also Theano), vectors are represented as rows instead of columns
- The first element in the *shape* parameter is there for batching, and `None` indicates that the dimension can vary.
- Classical neural networks are great for classification problems involving either fixed (finite) sets $f : A \to B$, where A represents the set to be classified, and B is the set of labels, or functions $f : \mathbb{R}^n \to B$.

Sequences

- In many cases, however, the data to be classified is better represented as a (finite) sequence.
- For example,
  - sentences are sequences of words,
  - words are sequences of letters,
  - movies are sequences of images,
  - sound files are sequences of amplitudes.
- It is possible to deal with sequences with ordinary neural networks, but we run into difficulties when we try to model dependency between different elements of a sequence.

Sequences

Let A be a finite set.

- A finite sequence of elements of A is a tuple $(a_1, ..., a_n)$, where $a_i \in A$ for every i. Here the number n is allowed to change.
- There is a special sequence, the empty sequence, denoted by ϵ.
- The set of all finite sequences of elements of A is denoted A^*.

There are several types of functions on sequences:

1. A function $f : A \to B^*$, which given an element of A outputs a sequence of elements of B. This is called a *one-to-many* function.
2. A function $f : A^* \to B$, which is given a sequence of elments of A and produces a single element of B. This is called a *many-to-one* function.
3. A function $f : A^* \to B^*$ which is given a seuqence of elements of A¡ and outputs a sequence of elements of B. This is called a *many-to-many* function.

Functions on Sequences

- We can of course define functions $A \to B^*$, $A^* \to B$ and $A^* \to B^*$ directly (they are just sets, after all),
- it is more interesting and instructive to make use of the structure of A^* and B^* as sets of sequences of A and B, and see how one can use a function $A \to B$ to define a function $A^* \to B^*$.
- Here is the simplest example: if $A = B$, and $f : A \to A$ is a function, then we can iterate f: $f^0(a) = a$ for every $a \in A$, and $f^{n+1}(a) = f(f^n(a))$. We terminate the iteration at the first value of n for which $f^n(a) = f^{n-1}(a)$ (note that there is no guarantee that this will ever happen).
- This gives a function $f^\omega : A \to A^\omega$.

Functions on Sequences

We describe a more general situation.

- Let S be a (finite) set of *states* with a distinguished element $\perp \in S$, and
- consider a function $f : A \times S \to B \times S$.
- Let $a_1, ..., a_n$ be a finite sequence of elements of A.
- We define two sequences $b_1, ..., b_n$ and $s_1, ..., s_n$ of elements of B and S respectively as follows:
  1. $b_1, s_1 = f(a_1, \perp)$.
  2. $b_{n+1}, s_{n+1} = f(a_{n+1}, s_n)$.
  3. We define $f^*(a_1...a_n) = b_1...b_n$, with $b_1, ..., b_n$ defined as above.

Functions on Sequences

An interesting special case of this is when $S = B$. In this case, we can have $f : A \times B \to B$. We can define

1. $b_1 = f(a_1, \perp)$,
2. $b_{n+1} = f(a_{n+1}, b_n)$.

This is in fact the main type of function we use in our examples later. Here's a note:

- Defining a function $f : A \to B \times C$ is exactly the same as defining two function, $f_1 : A \to B$ and $f_2 : A \to C$.
- In the case of $f : A \times S \to B \times S$, we are defining two function $f_1 : A \times S \to B$ and $f_2 : A \times S \to S$.

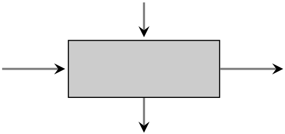
The former takes an input and a state to produce an outputs. The latter, taking the same state and input, produces a new state.

Recurrent Neural Networks

- Recurrent neural networks are just a prescription for a class of functions of the form $f : \mathbb{R}^n \times \mathbb{R}^\ell \times \mathbb{R}^m \to \mathbb{R}^\ell \times \mathbb{R}^m$, where $\mathbb{R}^m$ is the space of parameters.
- The set $\{0, 1\}^*$ of all finite sequences of 0's and 1's is countable,
- and $\mathbb{R}^\ell$, is a real vector space, is uncountable.
- Therefore, we can encode every element $w \in \{0, 1\}^*$ as a vector in S. Informally, $\mathbb{R}^\ell$ is enough to encode any finite state space, and every possible value for the content of the tape of a Turing machine. We get: *Recurrent neural networks are Turing-complete.*
- This explains why recurrent neural networks seem to be able to produce results that other networks can't.
- Training a recurrent network is the same as producing a Turing machine.

Recurrent Neural Networks

A common way to represent a recurrent node is to use a box, the inside of which represent the definition of f. The vertical arrows represent the input and onput of the node, and the horizontal arrows represent the input and output state of the node.



Recurrent Neural Networks

- Just like any regular neural network layer, recurrent nodes can be composed.
- The composition of two recurrent nodes is done by feeding the output of one node into the input of the other, and concatenating their state space.
- Formally, if $f : \mathbb{R}^n \times \mathbb{R}^\ell \times \mathbb{R}^m \to \mathbb{R}^k \times \mathbb{R}^\ell$ and $g : \mathbb{R}^k \times \mathbb{R}^\ell \times \mathbb{R}^m \to \mathbb{R}^q \times \mathbb{R}^\ell$ are two recurrent nodes,

$$(f; g) : \mathbb{R}^m \times \mathbb{R}^\ell \times \mathbb{R}^m \to \mathbb{R}^q \times \mathbb{R}^\ell \times \mathbb{R}^l$$

defined by

$$(f; g)(\mathbf{x}, \mathbf{s}_1, \mathbf{s}_2) = (\mathbf{y}, \mathbf{s}_1', \mathbf{s}_2')$$

where

$$f(\mathbf{x}, \mathbf{s}_1) = \mathbf{x}', \mathbf{s}_1'$$

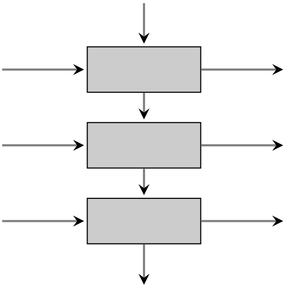
and

$$g(\mathbf{x}', \mathbf{s}_2) = \mathbf{y}, \mathbf{s}_2'$$

.

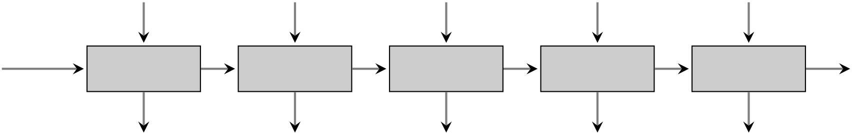
Recurrent Neural Networks

Graphically, we can represent this by stacking the boxes representing f and g on top of one another:



Training RNN's

How do we train recurrent neural networks? We use a variant of back propagation, just like a regular neural network. Heuristically, we unroll the recurrent network "infinitely" many times, until it looks like an ordinary very deep neural network. In practice, we only unroll the network a large but finite number of times, and treat it like an ordinary neural network.



The Basic RNN Cell

We begin with the most basic of recurrent cell.

- Abstractly, a function $f : U \times S \to V \times S$ can be defined using two functions $u : \mathbb{R}^n \times \mathbb{R}^\ell \to \mathbb{R}^n$ and $v : \mathbb{R}^n \times \mathbb{R}^\ell \to \mathbb{R}^\ell$.
- The former function as providing the output
- The latter function as a state updating function.
- The most common definition for u and v for basic recurrent cells is as follows: $u(x, s) = f(A_u x + B_u s)$, and $v(x, s) = \tanh(A_v x + B_v s)$, where
  - A_u, A_v, B_u and B_v are matrices,
  - f is a non-linear function

The Basic RNN Cell (code)

In tensorflow, the code looks like:

```
def basic_rnn_cell(i_tensor, s_tensor, o_dim):
  i_dim = input_tensor.get_shape()[1]
  s_dim = input_tensor.get_shape()[1]
  A_u = tf.Variable(shape=[i_dim, o_dim])
  B_u = tf.Variable(shape=[s_dim, o_dim])
  A_v = tf.Variable(shape=[i_dim, s_dim])
  B_v = tf.Variable(shape=[s_dim, s_dim])
  o_tensor = tf.relu(tf.matmul(i_tensor, A_u) + \
                     tf.matmul(s_tensor, B_u))
  ns_tensor = tf.tanh(tf.matmul(i_tensor, A_v) + \
                      tf.matmul(s_tensor, B_v))
  return o_tensor, ns_tensor
```

The Long-Short-Term-Memory Cell

- It is a cell that can remember elements of a sequence it has already seen
- Its state has two parts: a state and a memory vector;
- A special *forget gate* controls how much of the memory vector gets forgotten
- An *input gate* controls how new information gets stored into the memory vector
- The usual state update and output functions

The Long-Short-Term-Memory Cell

We define:

- $u : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^\ell \to \mathbb{R}^m$ and
- $v : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^\ell \to \mathbb{R}^\ell$

using auxiliary functions F, I, O, and S defined as follows

$$\begin{aligned}
F(x,y) &= \sigma(A_F x + B_F y + b_F) \\
I(x,y) &= \sigma(A_I x + B_I y + b_I) \\
O(x,y) &= \sigma(A_O x + B_O y + b_O) \\
S(x,y) &= \tanh(A_O x + B_O y + b_O)
\end{aligned}$$

The state update is given by

$$v(x,y,s) = F(x,y) \circ s + I(x,y) \circ \sigma_v(A_v x + B_v y + b_v)$$

where $\circ$ denotes pointwise multiplication of vectors, and finally, the output can be defined as

$$u(x,y,s) = O(x,y) \circ \sigma_u(v(x,y,s))$$

The LSTM Cell (code)

```
def lstm_gate(input_tensor, previous_output, op):
  _, N = input_tensor.get_shape()
  _, output_dim = previous_output.get_shape()
  A = tf.Variable(shape=[N, output_dim])
  B = tf.Variable(shape=[output_dim, output_dim])
  b = tf.Variable(shape=[1, output_dim])
  x = tf.matmul(input_tensor, A) + \
        tf.matmul(previous_output, B) + b
  return op(x)

def lstm_cell(input_tensor, output):
  _, output_dim = output.get_shape()
  F = lstm_gate(input_tensor, output, tf.sigmoid)
  I = lstm_gate(input_tensor, output, tf.sigmoid)
  O = lstm_gate(input_tensor, output, tf.sigmoid)
  S = lstm_gate(input_tensor, output, tf.tanh)
  new_state = tf.mul(output, F) + tf.mul(I, S)
  output = tf.mul(O, tf.tanh(new_state))
  return output, new_state
```

The Basic GRU Cell

- GRU's are a simplification of the LSTM cell.
- Compared to LSTM the GRU lacks an output gate.
- Its performance is on par with LSTM cells in most applications.

To define the functions u and v, we use auxiliary functions U(pdate) and R(eset) defined as follows

$$\begin{aligned} U(x,y) &= \sigma(A_U x + B_U y + b_U) \\ R(x,y) &= \sigma(A_R x + B_R y + b_R) \end{aligned}$$

The state update (and output) is given by

$$v(x,y,s) = U(x,y) \circ s + (1-s) \circ \sigma_h(A_v x + B_v(R(x,y) \circ y) + b_v)$$

The GRU Cell (code)

```
def gru_gate(input_tensor, previous_output, port_op):
  _, N = input_tensor.get_shape()
  _, output_dim = previous_output.get_shape()
  A = tf.Variable(shape=[N, output_dim])
  B = tf.Variable(shape=[output_dim, output_dim])
  b = tf.Variable(shape=[output_dim, output_dim])
  x = tf.matmul(input_tensor, A) + \
        tf.matmul(previous_output, B) + b
  return post_op(x)

def gru_cell(input_tensor, output, state):
  U = gru_gate(input_tensor, output, tf.sigmoid)
  R = gru_gate(input_tensor, output, tf.sigmoid)
  O = gru_gate(input, tf.mul(R, output))
  return [tf.mul(R, output) + tf.mul((1-R), O)]*2
```

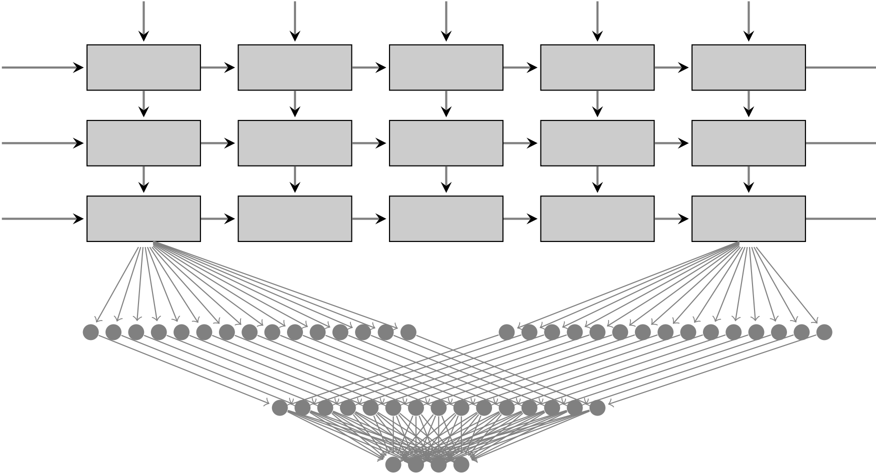
A many-to-one example

- As an example of a many-to-one RNN, consider the Buzzometer sentiment analysis tool.
- On input of a sequence of **characters**, we output one of 4 classes: negative, neutral, positive and irrelevant.
- For training, the network was unrolled to 256 characters, which is twise the average length of a message in out database.
- Longer messaages were truncated, and shorter messages were padded with 0.

The architecture is:

- A 3 layer *bi-directional RNN*
- We keep the last output of the forward and the backward networks, and combine them linearly: $\mathbf{w} = W_1\mathbf{v}_f + W_2\mathbf{v}_b$
- The vector $\mathbf{w}$ is then projected to $\mathbb{R}^4$.

A many-to-one example (model architecture)



A many-to-one example (code)

```
model_input = tf.placeholder(shape=[SEQ_LENGTH])
_ = tf.one_hot(model_input, depth=E_DIM, axis=-1)
_ = tf.reshape(_, [-1, SEQ_LENGTH, E_DIM])
fw = multi_layer_rnn(N_LAYERS, STATE_DIM)
bw = multi_layer_rnn(N_LAYERS, STATE_DIM)
OP = tf.nn.bidirectional_dynamic_rnn
output, _ = OP(fw, bw, _, dtype=tf.float32)
fw_output = tf.reshape(output[0][:, -1:],
                       [-1, STATE_DIM])
bw_output = tf.reshape(output[1][:, :1],
                       [-1, STATE_DIM])
f = project(fw_output, E_DIM)
b = project(bw_output, E_DIM)
e = tf.add(f, b)
model_output = project(e, NUM_CLASSES)
prediction = tf.argmax(model_output, 1)
```

A many-to-one example (output)

	Negative	Neutral	Positive	Irrelevant
Negative	291	113	50	127
Neutral	108	113	50	85
Positive	0	0	0	0
Irrelevant	0	0	0	0

	Negative	Neutral	Positive	Irrelevant
Negative	292	52	36	55
Neutral	61	176	21	62
Positive	5	15	65	15
Irrelevant	33	28	14	70

	Negative	Neutral	Positive	Irrelevant
Negative	319	38	11	43
Neutral	20	188	2	20
Positive	10	18	91	10
Irrelevant	27	27	7	169

A many-to-one example (output)

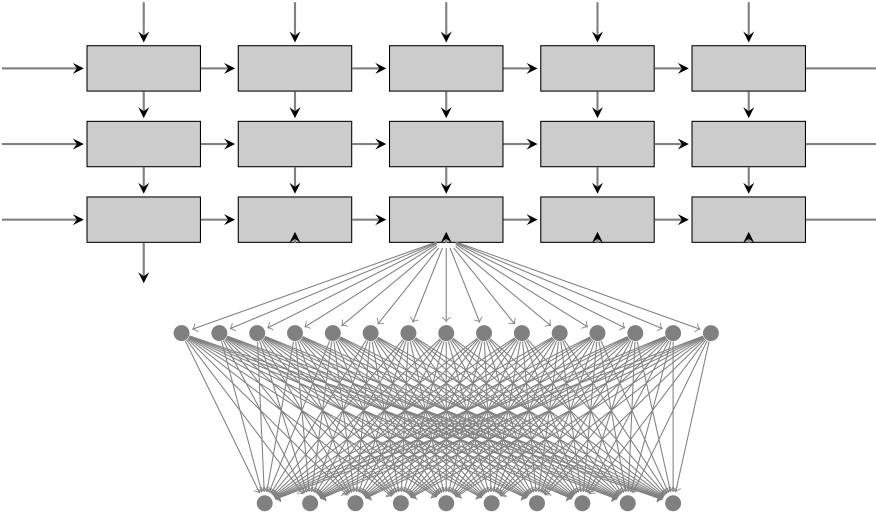
	Negative	Neutral	Positive	Irrelevant
Negative	334	4	4	9
Neutral	3	283	2	7
Positive	1	0	125	2
Irrelevant	1	0	4	221

	Negative	Neutral	Positive	Irrelevant
Negative	321	16	3	20
Neutral	8	286	4	6
Positive	4	2	125	16
Irrelevant	11	9	1	168

A many-to-many example

- we define an architecture that generates text in the style of a particular author, or body of text.
- The architecture is very simple, and consists of three stacked GRU cells.
- This model will also expose some of the challenges in training recurrent networks.

A many-to-many example (model architecture)



A many-to-one example (output)

- For training we network was unrolled to 30 characters, and the training was done in batches of 32 strings.
- The main challenge for training is that the strings should continue from batch to batch.
- For example, consider training a similar network unrolled to 2 characters with a batch size of 3.
- Consider the sentence `The quick brown fox jumps over the lazy dog`.
- The first step is to cut the string into substrings of length 2: `|Th|e␣|qu|ic|k␣|br|ow|n␣|fo|x␣|ju|mp|s␣|ov|er|␣t|he|␣l|az|y␣|do|g.|`.
- In a normal batching situation, we would then cut this liet of strings into chunks of length 3, like so: `|(Th|e␣|qu)|(ic|k␣|br)|(ow|n␣|fo)|(x␣|ju|mp)|(s␣|ov|er)|(␣t|he|␣l)|(az|y␣|do)|` but this causes a problem.

A many-to-many example (output)

```
|Th|e␣|qu|ic|k␣|br|ow|n␣|fo|x␣|ju|mp|s␣|ov|er|␣t|he|␣l|az|y
|do|g.|.
|(Th|e␣|qu)|(ic|k␣|br)|(ow|n␣|fo)|(x
|ju|mp)|(s␣|ov|er)|(␣t|he|␣l)|(az|y␣|do)|
```

- The first sequence of the first batch is `Th`, which will leave the network in a certain state *s*.
- For this state to be updated properly, the first element of the second batch should be `|e␣|`, and *not* `|ic|`.
- We must therefore use a different batching strategy. The string will give us 7 batches in total, so we number the subsequences with a batch index from 1 to 7 in order, starting over at 1 when we run out of indices.

A many-to-one example (output)

1	2	3	4	5	6	7
Th	e␣	qu	ic	k␣	br	ow
n␣	fo	x␣	ju	mp	s␣	ov
er	␣t	he	␣l	az	y␣	do
g.						

A many-to-many example (code)

```
model_input = tf.placeholder(shape=[None, SEQ_LENGTH]
initial_state = tf.placeholder(shape=[N_LAYERS, S_DIM
_ = tf.one_hot(model_input, depth=E_DIM, axis=-1)
encode = multi_layer_rnn(N_LAYERS, STATE_DIM)
state_tuple = tuple(tf.unstack(initial_state, axis=0)
OP = tf.nn.dynamic_rnn
output, state = OP(encode, _,
                   dtype=tf.float32,
                   initial_state=state_tuple)
output = tf.reshape(output, [-1, STATE_DIM])
output = project(output, E_DIM)
out = tf.reshape(out, [-1, SEQ_LENGTH])
model_output = tf.nn.softmax(output)
output = tf.argmax(output, 1)
```

A many-to-many example (text generating)

```
def generate_text(length, session=None):
    generated_text = ''
    character = [[ord('␣')]]
    istate = np.zeros([N_LAYERS, 1, STATE_DIM])
    while len(generated_text) < length:
        feed_dict = {model_input: character,
                     initial_state: istate}
        next_char, state = session.run(
          [out, state], feed_dict=feed_dict)
        op = np.random.multinomial
        next_char_id = op(1, next_char.squeeze(), 1)
        next_char_id = next_char_id.argmax()
        next_char_id = next_char_id \
               if chr(next_char_id) in \
                  string.printable else ord("␣")
        generated_text += chr(next_char_id)
        character = [[next_char_id]]
        istate = state
    return generated_text
```

A many-to-many example (output)

A many-to-many example (output)

A many-to-many example (output)

A many-to-many example (output)

A many-to-many example (output)

A many-to-many example (output)

A many-to-many example (output)

A many-to-many example (output)

A many-to-many example (output)

A many-to-many example (output)

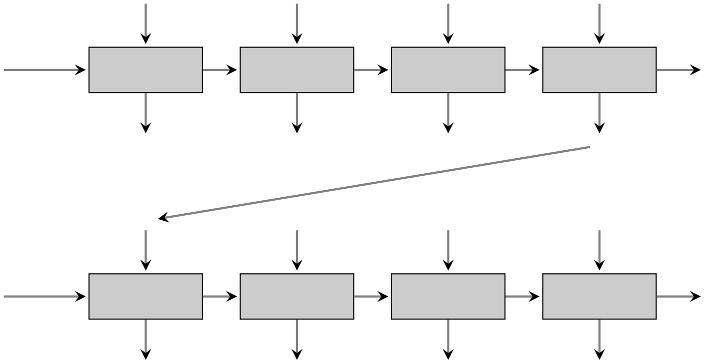
A many-to-many example (output)

A many-to-many example (output)

Many-to-one-to-many example

- A recurrent neural network that can sort sequences
- Two parts: an encoder, and a decoder
- The encoder encodes sequences into fixed length vectors
- The decoder transforms this vector into a sorted list of numbers.
- For simplicity, the model was restricted to sequences of length 32
- the elements of the sequence were all betweeen 1 and 128

Many-to-one-to-many example



Many-to-one-to-many example (code for encoder)

```
model_input = tf.placeholder('uint8',
                             shape=[None, SEQ_LENGTH])
_ = tf.one_hot(model_input, depth=E_DIM, axis=-1)
_ = tf.reshape(_, [-1, SEQ_LENGTH, E_DIM])
encode = multi_layer_rnn(N_LAYERS, STATE_DIM)
OP = tf.nn.dynamic_rnn
encoded_input, state = OP(encode, _, dtype=tf.float32)
encoder_output = state
```

Many-to-one-to-many example (code for decoder)

```
with tf.variable_scope('decoder'):
    training_decoder_input = \
           tf.zeros_like(Globals.model_input)
    _ = tf.one_hot(training_decoder_input,
                   depth=E_DIM, axis=-1)
    _ = tf.reshape(_, [-1, SEQ_LENGTH, E_DIM])
    decode = multi_layer_rnn(N_LAYERS, STATE_DIM)
    OP = tf.nn.dynamic_rnn
    decoded_output, state = OP(decode, _,
                            dtype=tf.float32,
                            initial_state=state)
    decoded_output = tf.reshape(decoded_output,
                            [-1, STATE_DIM])
    output = project(decoded_output, E_DIM)
    out = tf.argmax(output, 1)
```

A many-to-one-to-many example (output)

- 39, 9, 113, 57, 39, 124, 125, 86, 65, 30, 39, 124, 98, 65, 27, 62, 79, 102, 96, 45, 96, 42, 54, 92, 33, 61, 14, 106, 89, 4, 61, 95;
- 57, 6, 6, 6, 6, 6, 96, 96, 96, 96, 96, 96, 96, 96, 96, 96, 96, 118, 118, 118, 118, 118, 118, 118, 118, 118, 118, 118, 118, 118, 118, 118

- 22, 31, 58, 43, 96, 83, 95, 65, 17, 48, 65, 122, 94, 109, 113, 49, 74, 61, 39, 127, 3, 14, 107, 91, 55, 2, 108, 7, 119, 60, 32, 21;
- 14, 14, 14, 17, 17, 6, 6, 6, 96, 96, 96, 96, 96, 96, 96, 118, 118, 118, 118, 118, 118, 118, 118, 118, 118, 118, 118, 118, 118, 118, 118, 118

- 117, 15, 77, 4, 57, 121, 108, 76, 72, 119, 41, 4, 94, 44, 103, 124, 120, 20, 79, 121, 68, 12, 31, 97, 93, 19, 45, 27, 119, 92, 119, 92;
- 65, 65, 95, 95, 95, 96, 96, 96, 96, 96, 96, 96, 96, 96, 96, 96, 96, 118, 118, 118, 118, 118, 118, 118, 118, 118, 118, 118, 118, 118, 118, 118

A many-to-one-to-many example (output)

- 75, 6, 57, 57, 108, 34, 78, 71, 112, 115, 108, 48, 67, 1, 14, 9, 14, 115, 83, 62, 86, 91, 61, 40, 105, 92, 86, 84, 30, 84, 19, 107;
- 1, 6, 9, 14, 14, 19, 0, 34, 40, 48, 57, 57, 61, 62, 67, 71, 75, 78, 83, 84, 84, 85, 86, 91, 92, 105, 107, 107, 109, 111, 114, 115

- 59, 108, 56, 66, 76, 38, 100, 61, 47, 79, 102, 15, 24, 92, 8, 26, 126, 43, 5, 90, 41, 2, 60, 85, 2, 104, 86, 40, 35, 47, 61, 91;
- 2, 2, 5, 8, 15, 24, 26, 35, 38, 40, 41, 42, 47, 47, 56, 59, 60, 61, 61, 66, 76, 79, 84, 86, 89, 91, 92, 100, 102, 104, 108, 126

- 25, 50, 64, 4, 40, 47, 6, 14, 97, 32, 87, 103, 44, 25, 84, 40, 95, 13, 113, 66, 38, 79, 106, 40, 26, 16, 74, 50, 119, 32, 80, 16;
- 4, 6, 13, 14, 16, 16, 25, 25, 26, 32, 32, 38, 39, 40, 40, 44, 47, 50, 50, 64, 66, 74, 79, 80, 84, 86, 95, 97, 103, 106, 113, 119

A many-to-one-to-many example (output)

- 96, 59, 77, 29, 39, 112, 23, 79, 60, 110, 97, 69, 107, 13, 96, 124, 2, 12, 55, 16, 106, 110, 30, 118, 119, 52, 22, 37, 113, 93, 73, 58;
- 2, 12, 13, 16, 22, 23, 29, 0, 37, 39, 52, 55, 58, 59, 60, 69, 73, 77, 79, 93, 96, 96, 97, 106, 107, 109, 111, 111, 113, 118, 119, 124

- 44, 117, 104, 116, 111, 115, 58, 79, 8, 9, 1, 39, 112, 43, 64, 31, 126, 12, 36, 10, 93, 87, 40, 5, 108, 92, 11, 75, 113, 104, 64, 109;
- 1, 5, 8, 9, 10, 11, 12, 0, 36, 39, 40, 0, 44, 58, 64, 64, 75, 79, 0, 92, 93, 104, 104, 108, 109, 111, 113, 113, 115, 116, 117, 126

- 58, 19, 10, 52, 22, 61, 67, 96, 3, 7, 116, 96, 54, 24, 19, 45, 127, 124, 11, 114, 53, 75, 126, 84, 122, 41, 75, 1, 119, 18, 92, 51;
- 1, 3, 7, 10, 11, 18, 19, 19, 22, 24, 41, 45, 51, 52, 53, 54, 58, 61, 67, 75, 75, 84, 92, 96, 96, 114, 116, 119, 122, 124, 126, 127

A many-to-one-to-many example (output)

- 34, 73, 95, 91, 93, 108, 43, 75, 38, 70, 66, 40, 108, 127, 25, 94, 34, 26, 89, 23, 95, 43, 2, 54, 11, 19, 105, 52, 108, 77, 93, 86;
- 2, 11, 19, 23, 25, 26, 34, 34, 38, 40, 0, 0, 52, 54, 66, 70, 73, 75, 77, 86, 89, 91, 93, 93, 94, 95, 95, 105, 107, 108, 108, 127

- 95, 106, 119, 69, 40, 41, 28, 114, 12, 1, 106, 87, 117, 78, 54, 37, 110, 24, 9, 114, 107, 87, 33, 76, 5, 90, 29, 14, 96, 109, 1, 3;
- 1, 1, 3, 5, 9, 12, 14, 24, 28, 29, 33, 37, 40, 41, 54, 69, 76, 78, 86, 0, 90, 95, 96, 106, 106, 107, 109, 110, 114, 114, 117, 119

- 122, 97, 90, 61, 72, 66, 60, 60, 25, 125, 84, 73, 114, 46, 112, 76, 110, 62, 58, 34, 126, 124, 102, 35, 11, 100, 47, 113, 85, 64, 22, 89;
- 11, 22, 25, 34, 35, 46, 47, 58, 60, 60, 61, 62, 64, 66, 72, 73, 76, 84, 85, 89, 90, 97, 100, 102, 110, 113, 113, 114, 122, 124, 125, 126

A many-to-one-to-many example (output)

- 69, 41, 40, 61, 9, 120, 37, 85, 103, 47, 81, 77, 126, 72, 86, 72, 6, 72, 92, 7, 22, 100, 127, 19, 89, 72, 62, 6, 64, 26, 45, 15;
- 6, 6, 7, 9, 15, 19, 22, 26, 37, 40, 41, 45, 47, 61, 62, 64, 69, 72, 72, 72, 72, 77, 81, 85, 86, 89, 92, 100, 103, 120, 126, 127

- 95, 40, 116, 120, 68, 75, 24, 114, 39, 81, 52, 37, 76, 10, 15, 105, 79, 15, 44, 89, 115, 5, 26, 11, 85, 53, 85, 114, 39, 29, 34, 16;
- 5, 10, 11, 15, 15, 16, 24, 26, 29, 34, 37, 39, 39, 40, 44, 52, 53, 68, 75, 76, 79, 81, 85, 85, 89, 95, 105, 114, 114, 115, 116, 120

- 20, 61, 78, 15, 61, 8, 108, 36, 85, 96, 40, 80, 106, 24, 66, 82, 96, 43, 126, 33, 80, 116, 35, 86, 98, 81, 76, 89, 6, 103, 117, 16;
- 6, 8, 15, 16, 20, 24, 33, 35, 36, 40, 0, 61, 61, 66, 76, 78, 80, 80, 81, 82, 85, 86, 89, 96, 96, 98, 103, 106, 108, 116, 117, 126