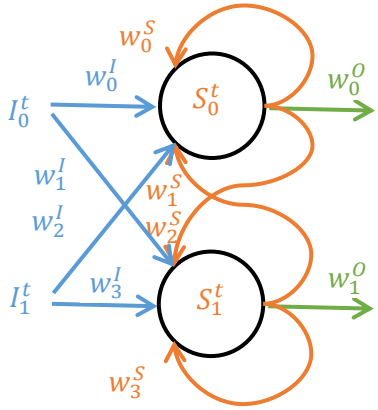
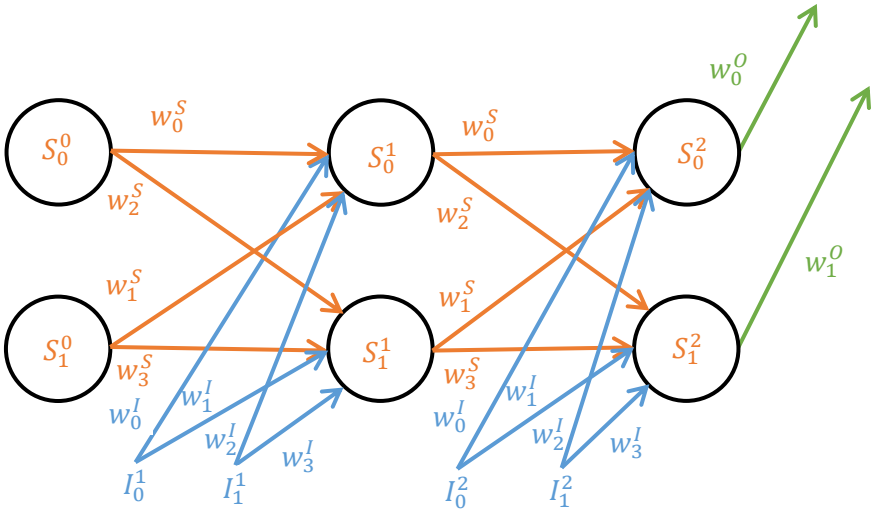
Topic 7: Recurrent neural networks

Apart from convolutional neural networks, another way to process a sequence is by using recurrent neural networks (RNNs). These are basically neural nets with a layer that feeds back into itself, like this:



The two neurons shown are each receiving an input 'I' at time 't' from the blue arrows called the input weights. They also receive an input 'S' at time 't' from themselves through the orange arrows called state weights. This means that their activations are going to be a function of both their current activations and their (blue) inputs. This creates a form of memory where the neurons are keeping a state that is evolving with every new input. After every input, the current state will be a vector that represents a summary of all the inputs the neurons have seen up to now. At the end, the current state is passed to the rest of the neural network through the green arrows called the output weights.

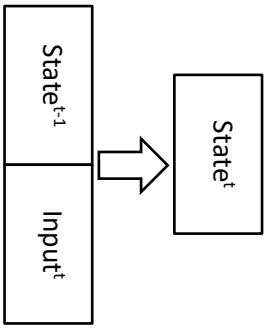
Cyclic graphs are not nice to deal with, especially for training the neural network. To get around this we usually unroll the RNN into a linear graph by replicating the parameters across time like this:



Note how corresponding parameters across time are identical. We can see that this chain of replicated neurons has an initial state at time step 0 which does not depend on any input and has a final state at the end which represents the whole sequence of inputs. The initial state can be the all zeros vector or can be a learnable vector that is optimised together the rest of the network or it can even be a separate input

that gets injected into the RNN's initial state in order to change how it remembers the sequence of inputs. How long should the chain of replicated neurons be? You might think that unrolling a cycle results in an infinitely long chain but for the purposes of training it only needs to be as long as the longest input in the dataset, which is finite.

From this replicated chain, we can see that the parameters of a single link in the chain are enough to define the RNN since all the links are identical. This is the basic neural network of the RNN:



Given a current state and an input, the recurrent layer concatenates them together and transforms them into a next state. Alternatively, instead of concatenating them you can also multiply them both by a weight matrix that turns them into equally sized vectors and then add the two vectors together (these are mathematically identical operations: $x_1 W_1 + x_2 W_2 \equiv x_{12} W_{12}$ where x_{12} is the horizontal concatenation of the vectors and W_{12} is the vertical concatenation of the matrices).

Once this base neural network is learned then it can be replicated into a chain for as long as the number of inputs needed. Note that contrary to a convolutional network, the input sequence of the RNN cannot be processed in parallel since each input depends on the result of its previous input, which makes RNNs significantly slow on long input sequences.

1. Scan function

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/07_-_Recurrent_neural_networks/01_-_Scan_function.py

The simplest way to create a recurrent function in Tensorflow is to use the scan function. The scan function takes in a sequence of inputs, an initial state, and a function that transforms an input and state into a new state. It takes each input and uses the function to produce a sequence of states. You can then take the final state or use the list of intermediate states to pass to another scan function for example. Note that the Python function that says how to transform an input and state into a new state is only called once whilst building the Tensorflow graph. After it has been built it is then the graph that gets replicated and not the function that gets called multiple times.

In the code, we'll see how to use the scan function to find the sum of a sequence of numbers, with the state being the current sum up to that point and the function adding the input to the current sum.

2. RNN cell

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/07_-_Recurrent_neural_networks/02_-_RNN_cell.py

The scan function is nice but what is usually used is a more powerful version called a 'dynamic_rnn' function. This takes an instance of a class called an RNN cell which contains a 'call' method that says how to transform an input and state into a new state. The call function returns a tuple of two things: an output vector and a state vector. For most purposes, you can assume that these two things are both the same thing: the state. However, after processing the whole input sequence, the dynamic_rnn function returns two things: a sequence of all intermediate output vectors as a matrix (a 3tensor actually since you pass in a batch of inputs) and a single final state vector (again, a matrix actually since you pass in a batch of inputs).

3. RNN cell extras

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/07_-_Recurrent_neural_networks/03_-_RNN_cell_extras.py

As already said, the dynamic_rnn function returns a sequence of outputs vectors and a final state vector. You can use several vectors to represent both the outputs and state rather than just one vector. You can use each vector for different purposes and each one will be returned in its own tensor. This is also useful for returning information about the internal workings of the RNN at each time step (such as any intermediate calculations made).

4. Simple RNN

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/07_-_Recurrent_neural_networks/04_-_Simple_RNN.py

The dynamic_rnn function can be used to create a simple recurrent neural network, the network described at the beginning of this topic. All you have to do is define the Cell RNN's 'call' function to concatenate the state and input vectors and pass it through a single layer neural net.

In the code, an RNN is used to encode a sentence in order to determine its sentiment.

5. RNN cell seqlen

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/07_-_Recurrent_neural_networks/05_-_RNN_cell_seqlen.py

RNNs are very sensitive to the length of the input sequence and changing the number of pad words will change the final state. Unfortunately, we need to provide a rectangular matrix as a batch of sentences and so we cannot do away with pad words. To fix this problem, the dynamic_rnn function also accepts the lengths of the sequences. This will make the recurrent layer ignore all inputs after the length of the sentence. The state after the last valid input will just be zeros and the final state will be the last valid state rather than the last state in the matrix of states over time. The lengths of the sequences cannot be determined from the sequences matrix (not easily at least) and so we usually just use a separate placeholder that accepts a vector of sequence lengths (one length for every sequence).

Another advantage of sequence lengths is that you don't need to specify an explicit pad word in your sentences. You can instead use an existing word in your vocabulary and it won't matter as it will be ignored. This will make your vocabulary one word smaller which will make your embedding matrix one row smaller.

6. Simple RNN seqlen

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/07_-_Recurrent_neural_networks/06_-_Simple_RNN_seqlen.py

We can now retry the sentiment analysis example with sequence lengths.

7. Contrib cells

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/07_-_Recurrent_neural_networks/07_-_Contrib_cells.py

We don't need to define an RNN Cell class every time we need to use one as Tensorflow provides readymade ones for us. Unfortunately, simple RNNs that use an activation function which is a squashing function like tanh or sigmoid tend to ignore inputs that happen early in a long input sequence during training. This is called the vanishing gradient problem and it is caused by the fact that the first input in a sequence does not significantly change RNN's final state and this problem get exponentially worse the longer input sequence is. One way to solve this is to use ReLU as an activation function, although this can instead result in the exploding gradient problem instead as there is nothing to control the rate of increase of the state's values. Most people prefer to instead use a more advanced form of RNN such as the long short-term memory (LSTM) or the gated recurrent unit (GRU) which are also readymade in Tensorflow.

In short, these advanced RNNs replace the simple RNN's equation

$$s^t = \tanh\big((s^{t-1} + + x^t)W + b\big)$$

with

$$s^t = \tanh\big((s^{t-1} + + x^t)W + b\big) + s^{t-1}$$

where s^t is the state vector at time step t, x^t is the input vector at time step t, and ++ means vector concatenation.

These two equations are recurrence relations as the next state is defined by the previous state. Let's try expanding them so that they consist of just input terms and the initial state (which is not defined in terms of earlier states). This is how the first equation expands:

$$s^1 = \tanh\big((s^0 + + x^1)W + b\big)$$

$$s^2 = \tanh\Big(\big(\tanh\big((s^0 + + x^1)W + b\big) + + x^2\big)W + b\Big)$$

$$s^3 = \tanh\Big(\Big(\tanh\Big(\big(\tanh\big((s^0 + + x^1)W + b\big) + + x^2\big)W + b\Big) + + x^3\Big)W + b\Big)$$

As the input sequence gets longer, the first equation expands such that early inputs get buried inside a bunch of nested tanh functions, which reduces their maximum value significantly. On the other hand, the second equation expands as follows:

$$s^1 = \tanh\big((s^0 + +x^1)W + b\big) + s^0$$

$$s^2 = \tanh\Big(\big(\big(\tanh\big((s^0 + +x^1)W + b\big) + s^0\big) + +x^2\big)W + b\Big) + \tanh\big((s^0 + +x^1)W + b\big) + s^0$$

$$s^3 = \tanh\Bigg(\Bigg(\Big(\tanh\Big(\big(\big(\tanh\big((s^0 + +x^1)W + b\big) + s^0\big) + +x^2\big)W + b\Big) + \tanh\big((s^0 + +x^1)W + b\big) + s^0\Big) + +x^3\Bigg)W + b\Bigg) + \tanh\Big(\big(\big(\tanh\big((s^0 + +x^1)W + b\big) + s^0\big) + +x^2\big)W + b\Big) + \tanh\big((s^0 + +x^1)W + b\big) + s^0$$

Note how this expansion now results in creating a separate term for every input rather than nesting everything together in the same term. Each term has a different input being inside just a single level of tanh function. This means that all inputs are squashed by tanh by the same amount, which solves the vanishing gradient problem. It also solves the exploding gradient problem since the value of a state can only increase by at most 1 and decrease by at most -1 at every time step because the new state is the current state plus an amount that is given by tanh, which is between 1 and -1.

The LSTM and GRU also introduce the concept of gating into the neural network, which is when each value in a neural layer are multiplied by some fraction to control whether to leave it as-is or to replace it with a zero (depending on whether it is multiplied by a 1 or a 0). The fraction is produced by a sigmoid layer which is multiplied by the gated layer. Let the gate layer be defined as follows:

$$g^t = \text{sig}\big((s^{t-1} + +x^t)W + b\big)$$

The LSTM is defined as follows:

$$c^t = g_i^t \times \tanh\big((s^{t-1} + +x^t)W + b\big) + g_f^t \times c^{t-1}$$

$$s^t = g_o^t \times \tanh(c^t)$$

where c^t is called a cell state, g_i^t is the input gate, g_f^t is the forget gate, and g_o^t is the output gate. The LSTM has two separate states, one called the cell state (c) and one called the hidden state (s). This means that it needs two separate initial states, which makes it rather complicated. The hidden state is what is usually used as a final state. The outputs vectors returned by the dynamic_rnn function are the hidden state vectors.

The GRU is simpler than the LSTM and is defined as follows:

$$s^t = g_z^t \times \tanh\Big(\big((g_r^t \times s^{t-1}) + +x^t\big)W + b\Big) + (1 - g_z^t) \times s^{t-1}$$

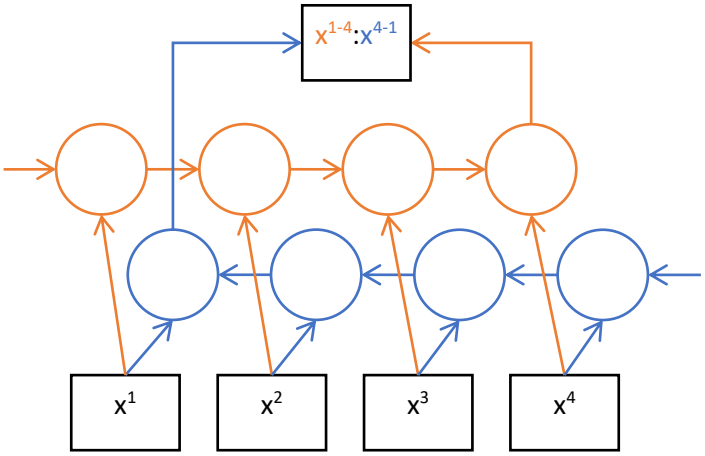
where g_z^t is the update gate and g_r^t is the reset gate.

A nice explanation of simple RNNs, LSTMs, and GRUs can be found in this blog.

8. Bidirectional RNNs

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/07_-_Recurrent_neural_networks/08_-_Bidirectional_RNNs.py

You can get an even better representation of a sequence by using two RNNs at once: one for encoding the sequence forwards and one for encoding the same sequence but backwards (in reverse). This results in having two final state vectors that can be concatenated together to create a single vector. This technique is called a bidirectional RNN and you can do this in Tensorflow by simply using the function 'bidirectional_dynamic_rnn'. Here is an illustration of how the bidirectional RNN is used to encode a sequence:



The bidirectional RNN is also useful for getting a left and a right context for every element in a sequence. To do this you use the bidirectional_dynamic_rnn's outputs instead of the state, thereby getting an intermediate state for each item. By concatenating corresponding intermediate states, you get a vector representing the left and right context of every element, which is useful for tasks such as spell checking or part of speech tagging. Here is an illustration of a bidirectional RNN encodes is used to encode each element in a sequence: