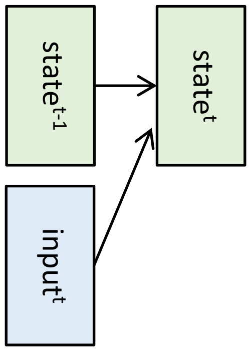
Topic 7: Recurrent neural networks

Apart from convolutional neural networks, another way to process a sequence is by using recurrent neural networks (RNNs). RNNs are neural networks with a memory. A memory is just a vector that stores information about what was seen in the past. In our case we want to keep a memory of the words in a sentence. In RNNs, the memory vector is called a hidden state vector, or just state.

The basic building block of an RNN is a special kind of neural layer that receives as input two vectors and gives as output a single vector. The inputs are an existing state and a new input (such as a word) whilst the output is a new state that represents the old state being modified by the new input.



The two input vectors are basically just concatenated together and the concatenated vector processed by a normal fully connected layer. The equation of this layer would look like this:

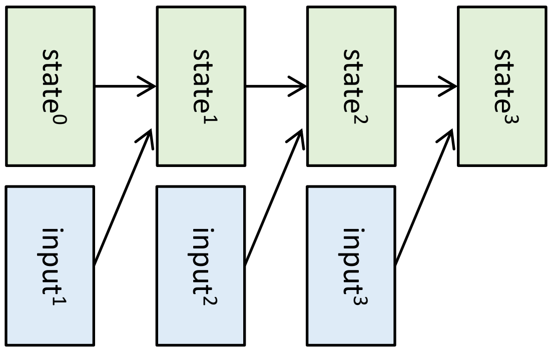
$$s^t = f(\text{conc}(s^{t-1}, x^t)W + b)$$

It is also mathematically equivalent to instead write the equation this way:

$$s^t = f(s^{t-1}W_s + x^tW_x + b)$$

where W_s and W_x are two different weight matrices.

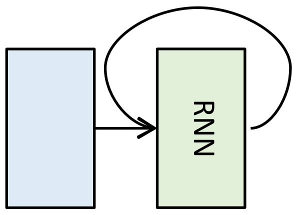
The fact that the layer takes in a state and returns another state which are of the same vector size means that we can replicate the layer in a chain with one output state being connected to the next input state.



Note how the above chain is reading in three inputs. The final state is a vector that is composed of these three inputs and can be used as a single vector representation of these three inputs. In the same way, the third state represents the first two inputs and the second state represents the first input alone. The first state is called the initial state and it is basically an empty memory which can be just a vector of zeros. Alternatively, instead of forcing the initial state to be zeros you can also let gradient descent optimise the content of the vector. The final state can then be connected to a softmax layer which classifies something about the sequence of inputs.

In order for the above neural network to be an RNN, the weights and biases of all the layers in the chain must be the same. Basically, during gradient descent, all the weights and biases in the chain would be the same variables (reused) which would then require the use of things like the product rule when performing differentiation.

Sometimes you'll see RNNs being drawn in a diagram as a cycle:



But this hides a lot of the way the RNN actually works in practice. It should be clear though that this is equivalent to an infinite chain of recurrent layers and that the previous chain is an unrolled version of the above cyclic network.

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 provided script, 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, the function adding the input to the current sum, and the initial state being zero.

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 vectors are both the same thing: the state. After processing the whole input sequence, the dynamic_rnn function returns a tuple of 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

The dynamic_rnn function returns a sequence of output vectors and a final state vector but each can be several tensors rather than just one. You can use each tensor for different purposes such as for returning intermediate values in the calculation of the next state.

4. Simple RNN

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

In order to create an RNN using the dynamic_rnn function, all you have to do is define the Cell RNN's 'call' function to concatenate the state and input vectors, pass it through a single neural layer, and finally return the output. In the provided script, 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

In practice, is it not reasonable to expect the number of words in a sentence to always be the same. If we want to train an RNN that is able to handle different input lengths then we need to train it with a training set of different input lengths. A neural network can only be provided with a rectangular matrix of sentences as input so the lengths of the rows cannot be different. One solution is to use minibatching but only providing sentences of the same length in a minibatch. This would also mean that for many sentence lengths, the minibatch will consist of just one sentence.

To fix this problem, the dynamic_rnn function also accepts the lengths of the sequences as input. This means that you can provide a rectangular matrix of padded sentences but also say what their actual length is such that any words that are found after the given length are treated as pad words and are ignored. The state after the last valid input will just be zeros and the final state will be the state vector obtained by the last non-pad word.

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, 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 which has already been discussed in an earlier topic. One way to reduce this problem is to use ReLU as an activation function. 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

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

with

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

where s^t is the state vector at time step t, x^t is the input vector at time step t, and 'conc' means vector concatenation. The LSTM and GRU take the above template and add gating factors to it, which is when each value in a neural layer are multiplied a 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:

$$g^t = \mathrm{sig}(\mathrm{conc}(s^{t-1}, x^t)W + b)$$

The LSTM is defined as follows:

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

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

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 output vectors returned by the dynamic_rnn function are the hidden state vectors.

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

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

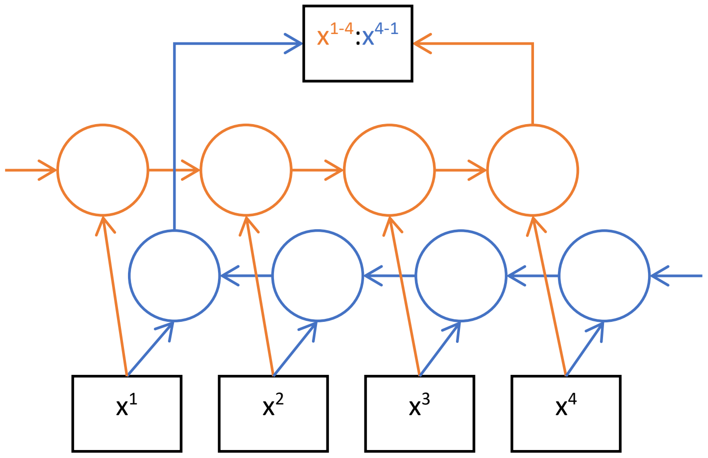
where g_z^t is the update gate and g_r^t is the reset gate.

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

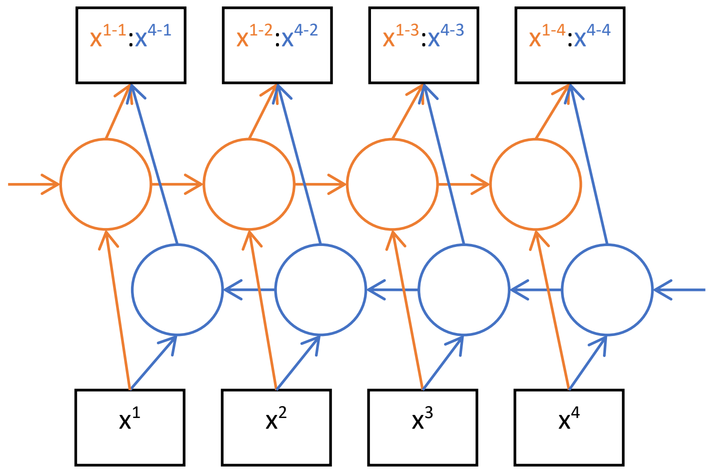
8. Bidirectional RNNs

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 being used to encode each element in a sequence:


x1-1:x4-1
x1-2:x4-2
x1-3:x4-3
x1-4:x4-4
x1
x2
x3
x4