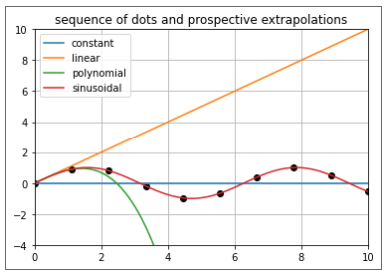
**Recurrent Neural Networks**

Recurrent neural networks are used mainly in Natural Language Processing (NLP). Specific instances would be name recognition (classifying every word in sentence as Name/Not\_Name), sentence autocompletion, sentence translation, sentiment analysis, etc. But for the sake of illustration, consider a simple numerical pattern recognition case. Say we had a bunch of black dots, as depicted below. And we were tasked with predicting how the pattern of dots would continue, given knowledge of only the first couple.



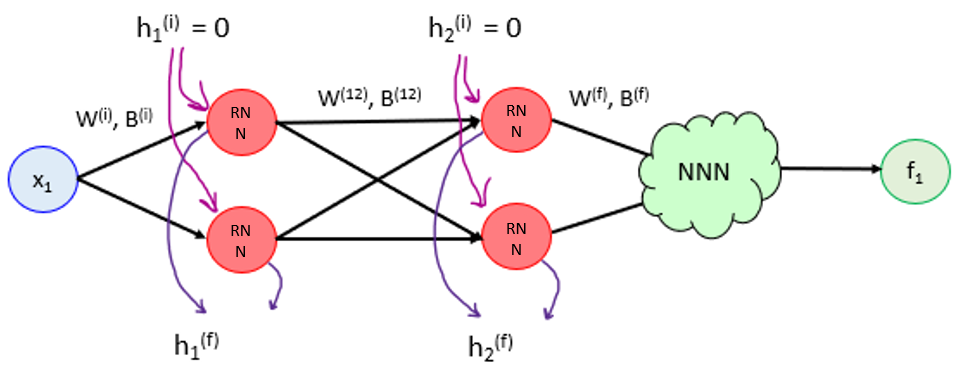
If we were given just the first dot in the sequence (the one at the origin), and asked to make a prediction as to where the sequence was headed, we might just say it’d be a constant. This would be the simplest prediction consonant with our knowledge. If we were given the first two dots in the sequence, then we migh posit the orange line. But if we were then given information about the three dots, we might posit something like the green polynomial. Finally, if we were given the first four or more dots, then we might settle on something like the red sinusoid. So in this fashion, we can see that we can make prediction given just one element in the sequence, but given more information, we can continually refine our prediction.

Say we had a Fibonacci sequence: 1 2 3 5 8 13 21 34 55 89 … and you want to figure out the pattern to predict the next number (or the next next number, it doesn’t have to be the number directly following the input), given some string of numbers. You know the next element, xn+1, of the sequence depends on the previous elements in some fashion, but you don’t know how far the dependence goes, or if the dependence depends on the element (for instance x10 might depend on x9 and x8, but x8 might depend on just x7). So we need a network that can take training data with different numbers of columns. And moreover, incorporate what it learns of the pattern from two-element sequences into what it learns of the pattern from three-element sequences, etc. Now consider the case of sentence completion. For instance, suppose we were to about to type the sentence, ‘There’s no place like home’. A sentence completion software thing might make predictions about the next word in the sequence. After ‘There’s’, it might guess ‘a, no, something’, etc. All of these seem likely possibilities. After ‘There’s no’, it might guess ‘business, place, where, thing’, etc. After ‘There’s no place’, it might guess ‘I’d, like’, etc. And after ‘There’s no place like’, I’m pretty sure it’d guess just ‘home’. So again, our recurrent neural network should be able to take variable sequence lengths and make predictions on the next element. This is essentially what Large Language Models do, I think. Continuing on this theme, we’ll observe that word prediction software doesn’t always venture a prediction. So I imagine there’s a threshold probability it requires before offering one. If the first word in the sequence doesn’t yield a prediction above the threshold, then it will incorporate the next word in the sequence, and see if the new prediction passes the threshold. And so on. And we could slightly generalize our example to require predictions of not just the next dot, number, or word, but the next several dots, numbers, or words.

So why can’t we use a normal neural network (NNN) to work out the pattern? I guess a couple reasons. First, we don’t know how many elements in the sequence are necessary to predict the next. We could just presume we need the last M = 100 or something. But then, for the Fibonacci sequence, we wouldn’t be able to incorporate the data for any of the first 99 elements in the sequence. Also, setting M = 100 might make the training take a loooong time, and also possible come to a false minimum, with so many extraneous weights and biases introduced into the algorithm. Seems better to use a recurrent neural network (RNN) and let it figure out M itself. So in general, it seems like you’d use a RNN when the next element(s) in a sequence depends on the M previous ones, but you don’t know what M is, or even if M changes with n.

**Simple RNN**

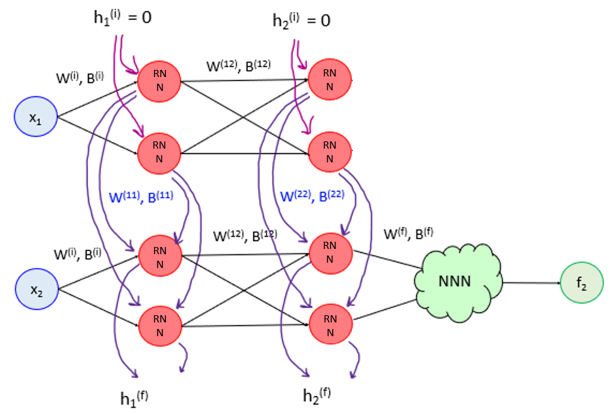
So to start, we would have some kind of neural network that can make a prediction on a single element in the sequence. Let’s posit something simple like this:



The sideways input is called the **hidden state**. It is often initialized to 0. And there will also be a **hidden state output**, which is typically ignored at this level. The RNN itself is typically just a single neuron with an activation function like RNN(x) = tanh(x) or σ(x). This guy might take training data of the sort,

|  |  |
| --- | --- |
| **x1** | **y** |
| 2 | 3 |
| 13 | 21 |

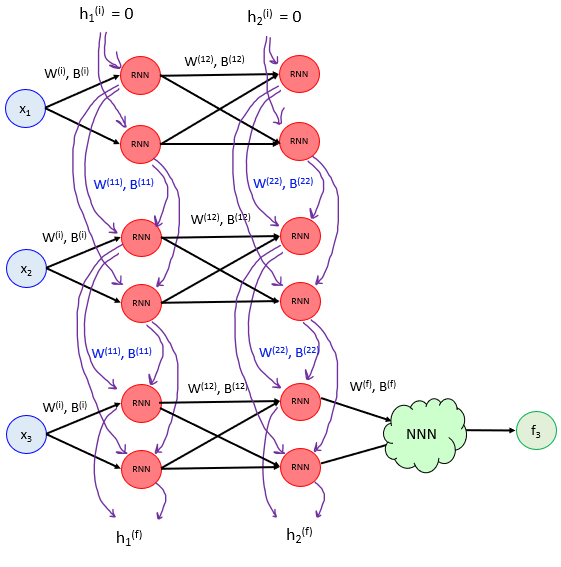
Then we need a way to a way to upgrade our prediction if we know two elements in the sequence. So to start, we feed the next element of our sequence, x2, into an identical neural network, but then connect the two networks by adding additional inputs from the hidden layers of the first network back into the hidden layers of the second network. Want to emphasize that all lines coming out of the RNN’s carry the same values initially. These of course will be multiplied by the weight in the line. It's just easier to draw them coming out from different places on the RNN node, rather than the same place. Also, all lines terminating on an A will of course be added together (and a bias added).



The NNN would often be the same structure. This might take training data of the sort,

|  |  |  |
| --- | --- | --- |
| **x1** | **x2** | **y** |
| 1 | 2 | 3 |
| 8 | 13 | 21 |
| 5 | 8 | 13 |

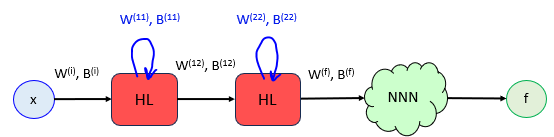
The matrix W(11) is, in this picture, a 2×2 matrix describing the weights of the inputs going side ways from the first layer to the first layer. Specifically W(11)ij is the weight going from element j in the first layer to element i in the second layer. Similarly, W(22) is a 2×2 matrix describing the weight going from the second layer to the second layer. And also, it is clear that this network is not like a normal neural network. In those networks, there are no sideways or backwards connections. So we’d have to evaluate this in a particular order. We’d run x1 through its part of the network, and get all of the outputs of those nodes. And then we’d run x2 and all of those aforementioned node outputs from the first part of the network, through the second part of the network. And if we have three elements in our sequence, then these same weights will be fed into a third identical neural network that takes x3 as an additional input. Then we’ll have:



and a prediction would be made, etc. Want to emphasize *again* that all lines coming out of the RNN’s carry the same values initially. These of course will be multiplied by the weight in the line. It's just easier to draw them coming out from different places on the RNN node, rather than the same place. Also, all lines terminating on an RNN node will of course be added together (and a bias added). Again, we’d typically ignore the first two outputs, as well as the final hidden state output. This might take training data of the sort:

|  |  |  |  |
| --- | --- | --- | --- |
| **x1** | **x2** | **x3** | **y** |
| 13 | 21 | 34 | 55 |
| 21 | 34 | 55 | 89 |

It’s important to recognize that for an RNN, all weights and biases labeled the same in the diagram above *are* the same. This saves computational time, and is a luxury afforded I guess by the somewhat optimized network structure of the RNN. For short, the RNN is written as,



So the network we use changes depending on whether we have 1, 2, 3, …, n elements in the sequence. If we’re only ever given two elements in the sequence, from which to predict the next, then we’d just use the two input model. Likewise we’d just use the three input model if we only ever have three elements in the sequence from which to predict the next. Again, might ask whether we would be better off just using a general NNN with two inputs if we *knew* that our sequences were only ever two elements long. But I think it is advisable to use RNN even in this case because it is designed to take account of a causal structure between the elements in the sequence and the output. Because its structure is different than a NNN, since the RNN has sideways and backwards connections? Don’t know for sure. But I guess if we have sequences with variable numbers of elements, and we want to predict the next, then using this architecture provides the best way to incorporate what is learned about the pattern from two-element sequences into the three-element sequence algorithm, and three-element sequences into four-element sequences, etc. And note that regardless of how many elements in the sequence we have, the model has the *same number of independent fitting parameters*.

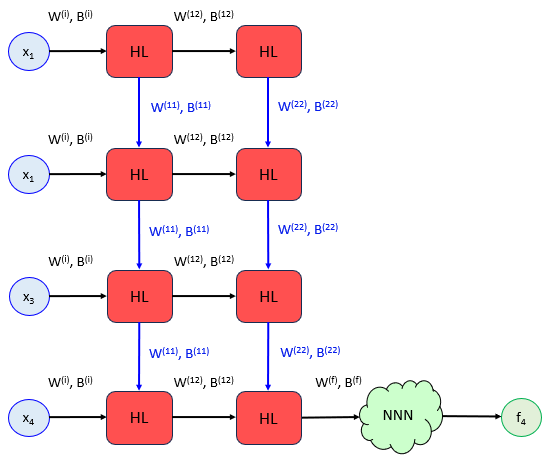
Should mention that this isn’t the most general model to look for patterns to handle sequences. We could in principle change (or allow to change) the input weights and biases going from the x’s to the hidden layers, the weights and biases going between hidden states, and the weights and biases going from the hidden states into the NNN. But to keep things simple, RNN’s keep these weights the same from time step to time step. And this might also already be the optimal architecture for teasing out patterns/causal structure between most types of sequences?

Seems trivial to extend this model to include vector inputs, i.e., x1, x2, etc., are an array of values. And to vector outputs, i.e., f is an array of values. This would be appropriate if the sequence elements were arrays. For instance, a sequence of words could be thought of as a sequence of an array of letters.

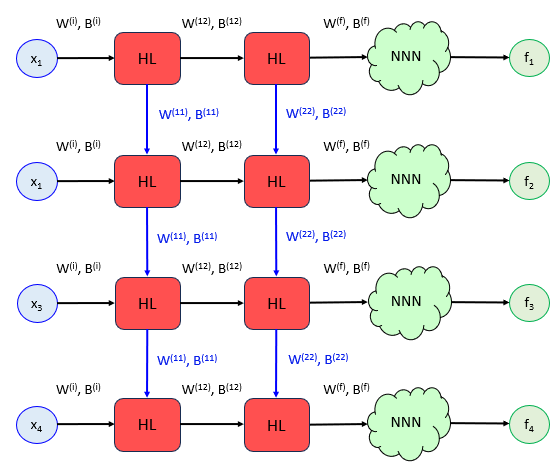
The loss function should be the usual, SSE for regression, or LL for classification.

**Other RNN Architectures**

So here’s a quick survey of the different RNN architectures. Note I’m leaving off the initial hidden state input, which we said was typically zero, and I’m also leaving off the final hidden output, which we don’t typically care about. The left network is what we examined above. It can be used for predicting the next element of a sequence, word completions, text classifications, etc.



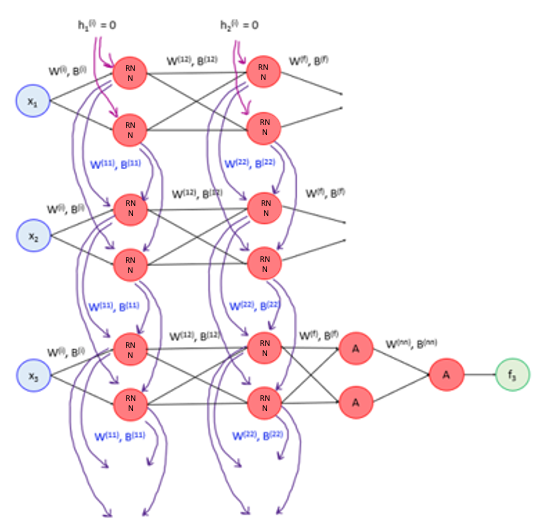
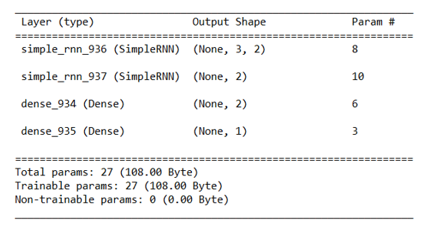
On the bottom we have something that could be useful for say live word translations, where every input basically needs an output. Note the weights and biases in the normal neural network (NNN) would be the same for all four time steps.



We’ll talk about more advanced architectures when we get to encoder-decoder networks.

**Counting parameters in Keras**

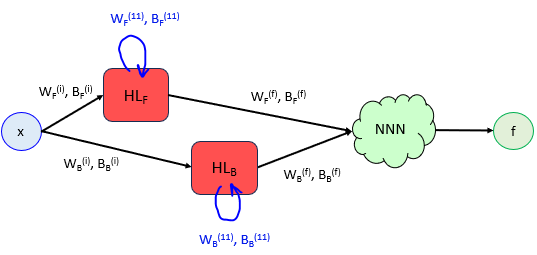
Here’s a model and its weights. Snapshot taken from Special Classes power point.

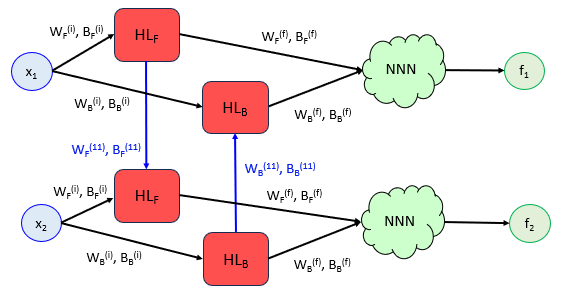
Above is the model summary. 27 parameters because 1 horizontal weight + 2 vertical weights + 1 bias going into each of the two hidden nodes in the first layer. So that’s 8 parameters total so far. Then 2 horizontal weights + 2 vertical weights + 1 bias going into each of the two hidden nodes in the second layer. So that’s 10 total. Then 2 weights + 1 bias going into each hidden node in the first nn processing layer. So that 6 total. And then 2 weights + 1 bias going into the last hidden node in the nn processing layer. So that’s 3 total. So the grand total is 27.

**Bidirectional RNNs**

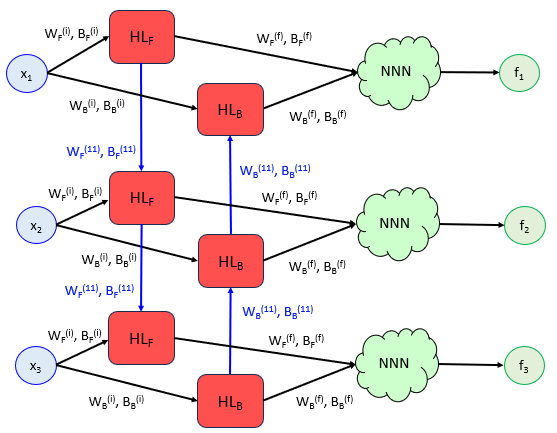
Real-time word translation problems would have such an architecture. Consider word recognition problems, where words later in the sentence can have an impact on the word names earlier in the sentence. For instance, ‘I love apple orchards’, and ‘I love apple iphones’ have two different uses of ‘apple’. One is a fruit, and the other is a company. But you can’t tell without looking at the word that follows ‘apple’. So in that case we build a bidirectional RNN. The simplest possible looks schematically something like this:



where HL is short for some hidden layer, as has been illustrated above. And I’m leaving off the hidden state input and outputs from the diagram. What isn’t apparent from the diagram is that as we unroll the network, the connections in HLb go backwards, while the connections in HLa go forwards, as usual. So when unrolled once, we’d have:



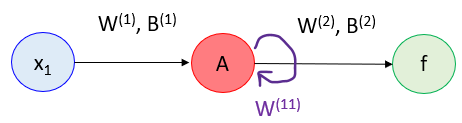
And unrolled twice, it would look like this:



etc. And as usual, all weights labelled the same, are the same.

**Example. Fibonacci sequence**

Would like to use an RNN to work out the next number in the Fibonacci sequence. Let’s try a simple RNN model like this. We’ll represent the RNN by a simple activation function, A (I actually used a linear activation function in the RNN I wrote, woops, not typical).



So with two elements in the sequence it’d be:

A picture containing text, circle, screenshot, diagram

Description automatically generated

and with three elements,

A picture containing text, screenshot, circle, diagram

Description automatically generated

etc. So there are five parameters in the network. To make this easier, I think I’ll just stick with one or two-element sequences. The output of the one element network would be:



and of two element network:



and of the three element network:



Results are pretty good when use a linear activation function A(x) = x. Of course, the Fibonacci sequence *is* linear, so that’s probably why.

A screenshot of a computer program

Description automatically generated with medium confidence

Results are *okay* when use softmax A(x). Note that our output *should* be f = x1 + x2. And with a linear activation function A(x) = x, our output reduces to:



Thus, the exact result would require,



The first two equations imply W(11) = 1. So then altogether we can say,



And despite the pretty good fit, our parameters are *way* off what they should be. I don’t know if this is because we just found a local minimum or what. For instance, if I just say B(1) = B(2) = 0, W(1) = W(2) = W(11) = 1, then I get:

A screenshot of a computer program

Description automatically generated with medium confidence

(ignore the error)

**Scaling Data**

Always scale your data! I ran an RNN on a sequence with values running between 0 and 1000, and got a bad fit (good slope actually, but with a constant offset that wouldn’t go away even if I manually initialized the bias to different values, or adjusted learning rates, etc.). But when I rescaled the data by dividing it by 1000 so that it lied between 0 and 1, I got a good fit. I wouldn’t have thought it would matter that much.

**Exploding/Vanishing Gradient**

One problem with RNN’s is the exploding/vanishing gradient problem. Note the loopy W’s are used once for a single-element input, twice recursively for a two-element input, and n-times recursively for an n-element input. Apropos gradient descent, when taking the derivative w/r to the loopy W’s, we will get terms of order (loopy W)n-1 for an n-input network. And so if (loopy W) < 1, this will go to zero (vanishing gradient problem), while if (loopy W) > 1, it will go to infinity (exploding gradient problem). And so the respective gradient will do the same. This makes large input models very difficult to train. Either the step size will be to small or too large. To illustrate, let’s take our three input RNN above.

A picture containing text, screenshot, circle, diagram

Description automatically generated

The output is:



And I’ll assume SSE is our loss function. And taking the derivative w/r to W(11),



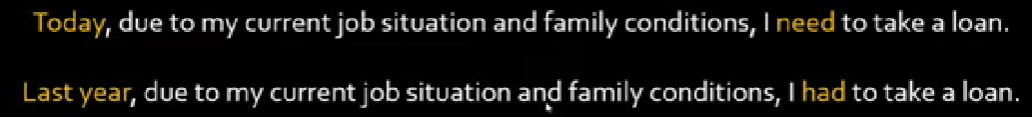
So can see we keep pulling down factors of W(11). So if W(11) < 1, then this gradient will be really small. And if W(11) > 1, the gradient will be really big, resulting in gradient descent step sizes either way too small or way too big. Either way, this will prevent proper tuning of the model. And if we take a derivative w/r to W(1), we get:



We run into a similar problem as before. But moreover, we can see that earlier information in the input string is supressed copared to later information. For instance, the term containing x3 as a factor has zero W(11)’s, while the x2 term as one W(11), and the x1 term has two W(11)’s. So the influence of x1 is supressed a lot compared to x3. And this problem would obviously be exacerbated for longer input strings. And taking derivative with respect to W(2),



We have the same situation as with W(11). Anyway, the vanishing/exploding gradient problem will obviously be an issue. Can use a sentence completion task as an example, taken from CodeBasics guy.



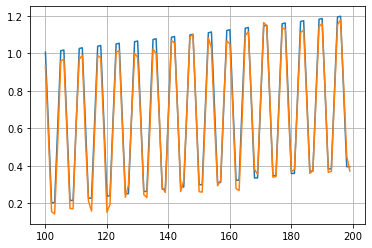
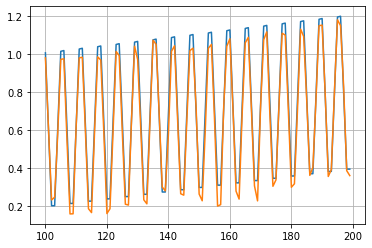
When we type in the first clause of the sentence as input, the computer will try to predict the second clause. But whether it uses the word *need* or *had* in the second clause, depends on the first word of the input clause, i.e., *Today* or *Last year*. And as we can extrapolate from the above example, in gradient descent, the influence of that first word, being the 10th word back from the end of the clause, will be modulated by a factor of something like [W(11)]10, which will result in hardly any influence at all if W(11) is small. So basically, the algorithm isn’t practically capable of dealing with inputs that aren’t fairly short. So it’s said to have a *short term memory*.

**Evaluating Fit**

It might be a good idea to compare whatever predictions or loss you get from the RNN with a simple rolling average. The RNN should be doing at least as well as that.

**Exploring the Model and Hyperparameters**

I’ve noticed so far that performance is enhanced pretty dramatically by having a lot of hidden nodes in the RNN layer. Even one layer with 10 nodes worked well for one series. So seems having a wide layer is more important than having a deep layer. That said, the more of either, the merrier. Having a complex output processing normal neural network doesn’t seem to matter as much compared to having a large breadth and depth of the RNN layers. Below I show the fit for two different RNN’s training on 15 terms in a series, and trying to predict the next 5. The predictions are graphed in orange (blue is the target). Fit on the left is for one layer with 10 RNN nodes. Fit on the right is two layers with 5 RNN nodes each, and it's not quite as good and also a bit slower, FWIW.

Very important to scale the data between -1 and 1, since the default RNN activation function is tanh(x). So data outside this range will tend make tanh(x) dead, kind of.