Recurrent Neural Networks (RNNs) Part 1

Eckel, TJHSST AI2, Spring 2024

Background & Explanation

In the way of research, once neural networks started showing tremendous results in one area (namely, image processing), people began attempting to use them in other areas to see what would happen.

One fairly clear jump is from categorization to *prediction* – can we make a network that takes a sequence of things occurring over time and guesses what happens next? Seems like the traditional back propagation idea should work well here, at least intuitively – define an error function, and propagate it back through some kind of network. But unlike the networks we've seen so far, a network like this would need to progress both through *layers* and through *time*. We would feed it the first item in a sequence, then the next, then the next, etc, one step at a time. The network would calculate a full forward propagation step after each item, based on its prior state and the new input. After feeding in the last piece of input data, we would see what the network outputs as its prediction for what comes next. This means we need a network that actually takes in two sets of information at each step – the *next input in the sequence* and also *the previous state of the network*.

You can think of this as giving neural networks *memory*; they now know what they previously calculated, and can use that information to affect the next step.

Formal definition time. A **recurrent neural network** is one in which each layer has now two weight matrices – one weight matrix applying to the inputs from the previous *layer* (which is new in each step) and one weight matrix applying to the output of the *same* layer in the *previous* step. At each step, you give this network the next item in the sequence, and each layer computes as it did in the back propagation labs we've done, just with two weight matrices applying to these two inputs.

This idea has a certain intuitive appeal, but, of course, there are a lot of neat ideas that don't work. How can we test this idea and compare it to previous solutions (namely, the **dense neural networks**, or DNNs, we used in back propagation – "dense" referring to the fact that every neuron is connected to every neuron in the next layer)? It's possible that in a future year I'll find a naturally occurring data set we can use here, but in the meantime it's quite easy for us to generate a realistic simulated data set that will serve this purpose just fine.

In this assignment, we'll generate such a training set, see how well a naïve estimate and a DNN estimate do, and then build the simplest possible RNN – just a single recurrent node with one input and output – and compare it as well. We'll find that just that simple RNN, with only two weights, isn't that much worse than the full DNN, with fifty. And in future assignments, we'll build from that baseline to outclass DNNs completely.

But first: some training data.

A Natural Data Set for Testing RNNs: Sinusoidal Time Series

Let’s generate a lot of sequences that behave like sequential data measured over time in the real world – each value should be affected by a few of different parameters, plus some noise. Specifically, we’ll take two sine waves with different parameters and add them together, producing a complicated wave, and then take each individual data point and add a little bit of jitter to it. By doing this, we can keep the output values between -1 and 1, since sine waves don’t increase or decrease long-term, which means we won’t have to mess with our data in order for our network to get appropriate values. (We will need to use an activation function that goes from -1 to 1 instead of the sigmoid function which goes from 0 to 1, but this won’t be hard.)

Here's code to generate a single one of these sinusoidal time series:

```
def generate_time_series(n_steps):
    freq1, freq2, offsets1, offsets2 = np.random.rand(4)
    time = np.linspace(0, 1, n_steps)
    series = 0.5 * np.sin((time - offsets1) * (freq1 * 10 + 10))
    series += 0.2 * np.sin((time - offsets2) * (freq2 * 20 + 20))
    series += 0.1 * (np.random.rand(n_steps) - 0.5)
    return series
```

np.linspace simply separates the a range into a given number of same-size steps, so the second line in this function will simply create a sequence of values that runs from 0 to 1 in *n_steps* number of steps. The rest here should be pretty self explanatory – we make two sin functions with different parameters and offsets, one of which has a larger magnitude than the other, and then add in a bit of purely random noise. np.random.rand generates values between 0 and 1; by subtracting 0.5 on the last line it becomes randomly between -0.5 and 0.5, then we multiply by 0.1 to reduce the magnitude. A small amount of chaos is normal in real world observations.

(If you’re having trouble quite visualizing what this looks like, you’ll actually be making graphs in a future assignment; not to worry – it’ll all come together.)

Copy this code into your assignment and use it **without modification** to generate training data.

Note: be aware that the code above outputs a sinusoidal time series as a one-dimensional array; we’ll need to do some work to transform each one of them into the input format each network will need. But DNNs and RNNs need input formatted differently, so we’ll solve that problem later.

Baseline Comparisons

Now, before we write an RNN, let's see how well naïve guessing and simple DNNs do on this data.

Let's make a large training set – say, 7,000 different sinusoidal time series – and a smaller testing set – say, 2,000 different sinusoidal time series. Let's start by making each time series 51 values long, and taking the first 50 values as the input and the last value as the output we want to predict after looking at the first 50.

Naïve Guessing

Before we look at any networks, let's make a very dumb guess. Let's just use the 50th value as our guess for what the 51st will be. Let's see how wrong this is by finding the **mean squared error**. Loop over all 2,000 time series in our testing set, and use the 50th value as your guess. (I shouldn't need to remind you, this means the value at index 49). Compare it to the final value, and find the squared error – $(\text{real} - \text{predicted})^2$. Find the average of this squared error across all 2,000 series. You should get a value close to 0.02. This number is pretty meaningless on its own, but it gives us a good baseline; if our network can't do better than THAT, we can safely say it's a bad network for this problem!

DNN

Next, let's use a perceptron. For this, we won't use any hidden layers; we'll just use a single 50-input perceptron, a 50 > 1 network. Note that this will have 51 parameters – the weights for each input value, and the bias. We do need to tweak a few things for this to work:

- You'll need to reshape the 51-value time series into a 50x1 input array and a 1x1 output array, **both of which are 2-dimensional**. [[y]] is what each output should look like, not [y].
- We'll need to change our activation function. We're trying to predict a value between -1 and 1, we can't use an activation function that only outputs between 0 and 1. We could just use 2 * sigmoid - 1, but since we need to change anyway, let's use this opportunity to look at a different activation function: y = tanh(x), or hyperbolic tangent. This looks a bit like sigmoid, except it ranges from -1 to 1 instead of 0 to 1, exactly as we want.
    - In numpy, access this with `np.tanh(your_input)` – this is already vectorized, and does not need np.vectorize to be used.
    - The derivative of **$y = \tanh(x)$** is **$y' = 1 / (\cosh(x))^2$**. Use `np.cosh(your_input)` as you'd expect.

Despite the large size of our training data – 7,000 series – this should run through each epoch very fast. 51 parameters isn't a lot. I got my best results with a learning rate of 0.1 and it still took many epochs, but I was able to get a mean squared error of about a quarter of the naive guess; about 0.005. Interestingly, adding a hidden layer of 10 perceptrons didn't really improve this much! In any case: it's definitely better than naïve guessing. This is good.

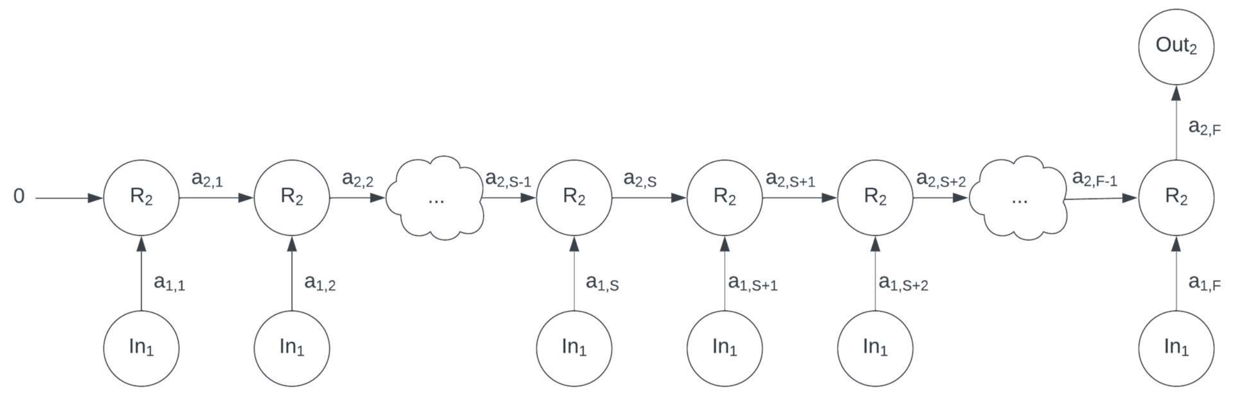
Before moving on, message me on Mattermost: what was your mean squared error for your naïve estimate? For your DNN estimate?

Now, let's find out – how does this compare to RNNs?

The Simplest RNN

Let's make the simplest possible RNN – a single input node with 1 input and a single output node with 1 output. Much like our previous unit, this "input node" is not a perceptron – it just feeds a value into the network. The output node is very similar to a perceptron – it has a single weight for its input and a bias. There is just one change. To be a recurrent neural network, we need to also make our output node a **recurrent node**, which means it needs to take its own output in the previous step, and give that a weight as well.

This is what the whole process will look like:



In this diagram, there are only two nodes – In_1 and R_2. As the diagram moves to the right, it moves through time; each input value is put in one at a time, and at each step the recurrent node receives the input at that step and the output of the previous step.

You'll notice some new notation here. To keep track of not only the node but also the step, each output a value is given a subscript with two values – node and step. So $a_{1,1}$ refers to the output of node 1 during step 1. We use S to refer to any particular step (see the middle part of the diagram) and F to refer to the final step.

Note also that, for the first step, we don't have any previous output to use. Just send in a zero. Later, this will become an appropriately sized vector of zeroes, but for now we just need a single zero.

Let's be precise about this. Since we're only dealing with one node, with one input, with one output, we actually don't need to use matrices for this. You CAN if you want to – you'll need to in the next assignment – but we could track this network only labelling individual values.

- Our node has *two* weight values. Let's call them wl_2 (for the **w**eight from the previous **l**ayer in recurrent node **2**) and ws_2 (for the **w**eight from the previous **s**tep in recurrent node 2). We also, of course, have b_2. These three values fully specify the recurrent node and, so, the network.
- We'll continue to use tanh as our activation function.
- During forward propagation, the input node sends in each value in our input sequence one at a time on each step. Our recurrent node, at each step S, calculates:
  - $dot_{2,S} = wl_2 * a_{1,S} + ws_2 * a_{2,S-1} + b_2$
  - $a_{2,S} = A(dot_{2,S})$
  - (Note we need to keep a list of all of our dots and as, as each step will produce a new value and we'll need them all for back propagation!)

- How does back propagation work here? Well, now, instead of propagating back through *layers* we propagate back through *steps*, and there's only one value that matters there – ws_2. So let's make a delta for the final *step,* and then back propagate from there to get a delta for each prior step:
    - $\Delta_{2,F} = (y - a_{2,F}) * A'(dot_{2,F})$ This is exactly what you expect – the first steps of the chain we did for back propagation in the last unit – just applied to a single value.
    - $\Delta_{2,S} = \Delta_{2,S+1} * ws_2 * A'(dot_{2,S})$ This just continues the chain rule back from F to each previous step S, all the way to the beginning. Shouldn't be hard to verify. Note that ws_2 is the same every time.
- Now we need to update each weight and bias. We have F different steps; we'll make an update for each weight and bias for each step, sum them, and add them all (multiplied by the learning rate). Each of these update steps mirrors what we did in back propagation; they should be again easy to verify.
    - $b_2 = b_2 + \lambda \cdot \text{sum}(\Delta_{2,S} \text{ for all } S)$
    - $ws_2 = ws_2 + \lambda \cdot \text{sum}(\Delta_{2,S} * a_{2,S-1} \text{ for all } S \text{ except the first step})$
    - $wl_2 = wl_2 + \lambda \cdot \text{sum}(\Delta_{2,S} * a_{1,S} \text{ for all } S)$
- Since we're modifying each value F times on a single training step, a smaller learning rate is recommended. Try about a tenth or possibly even a hundredth of what you used for DNNs.

Implementation Advice

We will learn how to implement RNNs with layers of arbitrary size using matrix operations, just like we did in Back Propagation, in the next assignment. If you'd like to work ahead and try and figure out the general matrix formulas for all this, feel free! You can do this assignment with a more general piece of code quite easily! But for most students, I recommend writing this separately and just using individual values, not worrying about numpy arrays/matrices, and getting the hang of how information flows here conceptually. We can focus on coding the general case next time.

If you **DO** code with matrices, your input for the time series problem we're working on is now not a single 50x1 array, but rather 50 separate 1x1 arrays. Make sure they're all 2d arrays, even though they're only 1x1. Your output is another 1x1 2d array (this part is the same as the DNN.)

If you **DON'T** code with matrices, and just keep track of individual values, then your input for the time series problem we're working on is a list of 50 individual values. Your output is another single, individual value.

Testing

After you run a training epoch (all 7,000 training series), then test your network on the testing set. Loop over all 2,000 testing series and feed them to your network one value at a time. After the 50th value has been inputted, the outputted value is your network's prediction. As with both of the above examples, find the mean squared error of these 2,000 outputs compared to the real values.

You should get, after a few epochs, that this doesn't do as well as the perceptron but it does much better than our naïve guess – something like 0.011 mean squared error. This is pretty great, considering the perceptron has 51 parameters and this RNN just has 3; we've made something a bit less effective that is MUCH smaller.

Send me a mattermost message with your one-node RNN's mean squared error after a few epochs.

What results might we get if we use an RNN with roughly the same number of parameters as this perceptron? More?

That's where we're going in the next assignment.

Get this one ready to turn in, and then get ready for some math!

Specification

Submit **a single python script** to the link on the course website.

This assignment is **complete** if:

- You follow the instructions on the submission form to format your submission properly.
- Your code matches the specifications above.
- Your code does three different things depending on command line input:
  - If it receives the argument “N”, it should generate the train and test sets, then print the naïve estimate mean squared error on the test set.
  - If it receives the argument “D”, it should generate the train and test sets, then run a DNN (a single perceptron) to train. After each epoch, calculate and print the mean squared error on the test set. **Please make sure you TRAIN on the TRAIN set and TEST on the TEST set.**
  - If it receives the argument “R”, it should generate the train and test sets, then run an RNN (a single node) to train. After each epoch, calculate and print the mean squared error on the test set. **Please make sure you TRAIN on the TRAIN set and TEST on the TEST set.**