Recurrent Neural Networks (RNNs) Part 2

Eckel, TJHSST AI2, Spring 2024

Background & Explanation

In class, we worked out this math for a general description of an RNN with N layers and F steps:

```
for each epoch:
    for each x_sequence, y in training set:
        for each layer L in network:
            $a^{L,0}$ = an appropriately sized column vector of zeroes
        for each step S from 1 up to F = len(x_sequence) inclusive:
            $a^{0,S}$ = x_sequence[S]  # or possibly [S-1] – do you see why?
            for each layer L in network:
                $dot^{L,S} = wl^{L} \cdot a^{L-1,S} + ws^{L} \cdot a^{L,S-1} + b^{L}$
                $a^{L,S} = A(dot^{L,S})$
        $\Delta^{N,F} = A'(dot^{N,F}) \otimes (y - a^{N,F})$
        for each step S (counting down from F-1):
            $\Delta^{N,S} = A'(dot^{N,S}) \otimes (ws^{N})^{T} \cdot \Delta^{N,S+1}$
        for each layer L in network (counting down from N-1):
            $\Delta^{L,F} = A'(dot^{L,F}) \otimes (wl^{L+1})^{T} \cdot \Delta^{L+1,F}$
        for each step S (counting down from F-1):
            for each layer L in network (counting down from N-1):
                $\Delta^{L,S} = A'(dot^{L,S}) \otimes (ws^{L})^{T} \cdot \Delta^{L,S+1} + A'(dot^{L,S}) \otimes (wl^{L+1})^{T} \cdot \Delta^{L+1,S}$
        for each layer L in network:
            $b^{L} = b^{L} + \lambda \cdot sum(\Delta^{L,S}$ for all S)
            $wl^{L} = wl^{L} + \lambda \cdot sum(\Delta^{L,S} \cdot (a^{L-1,S})^{T}$ for all S)
            $ws^{L} = ws^{L} + \lambda \cdot sum(\Delta^{L,S} \cdot (a^{L,S-1})^{T}$ for all S except the first)
```

Assignment

Simply this: let's make a recurrent network that beats the linear network handily. We noticed that the one-node network from last assignment had way fewer parameters (3) than the [50, 1] perceptron (50 weights and 1 bias for a total of 51); we can start here by making an RNN that actually is of similar parameter size. A [1, 6, 1] network comes close:

- wl1 is 6 x 1 (6 parameters)
- ws1 is 6 x 6 (36 parameters)
- b1 is 6 x 1 (6 parameters)
- wl2 is 1 x 6 (6 parameters)
- ws2 is 1 x 1 (1 parameter)
- b2 is 1 x 1 (1 parameter)

Code a [1, 6, 1] recurrent network with the algorithm above to predict the 51st value in each of the sequences in your training data. Run!

…except you will probably find that it turns out it isn't quite that easy. If you code this up directly and just run it, you're likely to get an overflow error. A value is getting too big. What's wrong?

*(A quick note: if you **don't** get an overflow error now, **keep reading anyway**. The solution to this error will make your code converge much faster and with lower error regardless, and you definitely will get this error in a later assignment if you don't see it yet, so let's make sure to take care of it either way.)*

Exploding / Vanishing Gradients

One problem with any back propagating network, but especially with RNNs, is that since the same process is being repeated over and over, that process is likely to either overflow or diminish exponentially. Think about back propagating along the steps – you're using the same weight matrix each time, and the same activation function derivative, so if the values go up a few times… they'll keep going up. Probably overflowing. Or if they go down a few times… they'll likely keep going down. So close to zero that they don't really make any difference. How can we get a handle on this problem?

There are a few strategies here.

Clipping

I'll mention this first, but you probably don't need it in this problem; it's sort of a strategy of last resort. But if all else fails: when calculating the delta matrices, just pick a number and don't let any delta values exceed that number. Hopefully it's intuitive why this would help solve the exploding gradients problem!

Of course, this doesn't help you at all with *vanishing* gradients, but while those are a problem, they're more a problem of inefficiency; those don't crash your code.

A function that already works on matrix inputs (ie, needs no vectorizing) that will clip every value in a delta matrix to somewhere between 5 and -5 is this:

```
def clip(inp):
    l = 5
    return np.piecewise(inp, [inp < -1*l, inp > l, abs(inp) <= l], [lambda x: -1*l, lambda x: l, lambda x: x])
```

I include this here because this is a new unit and I'm not sure if your progression will be the same as mine. But for what it's worth, I didn't need this in this particular problem. If you DO turn out to need this, let me know – I'm curious! And either way, keep it in your back pocket for future assignments.

Better Initialization Range

This strategy works for both exploding and vanishing gradients. It's sort of shocking how well it works!

The basic idea is this: we ought to be able to control how big or small gradients get by controlling the size of the weight values we choose when we initialize our network, since each back propagation step is weighted by one of the weight matrices transposed. As the weight matrices get larger and larger, on larger and larger networks, maybe the individual weights themselves should get smaller – that way, summing multiple weights times previous deltas, on the matrix multiplication step, won't overflow.

As a teacher, I don't like to give Magic Formulas without being able to tell you where they come from, but I reluctantly do so now. Someone did the math and figured this out, but I must admit, I was unable to find a derivation for it. Hopefully in a future year I can give you a sense of where this comes from.

So for now: a Magic Formula. You'll be amazed what a difference this makes.

To better initialize the values in your weight and bias matrices, for any layer (either a DNN layer or an RNN layer), do this:

$$temp = \frac{(\#\ of\ inputs\ into\ this\ layer) + (\#\ of\ outputs\ out\ of\ this\ layer)}{2}$$

$$r = \sqrt{\frac{3}{temp}}$$

(Why a square root? Why a 3? No idea! I'd love to find out someday.)

Calculate r, then initialize all weight and bias values to somewhere between -r and r, instead of -1 to 1. That's it!

Note that for a DNN, the sum on the top of the first fraction works out to (size of previous layer + size of current layer), but for an RNN, the number of inputs into the current layer is actually the size of the previous layer + the size of the current layer, taking into account both input vectors. So when you're coding this for an RNN, the sum works out to (size of previous layer + 2 * size of current layer) in total.

Implement Better Initialization Range

Let's use this formula to fix our exploding gradients problem. (Or, if you didn't get exploding gradients, let's use the formula anyway and see how much better the network performs!) Re-initialize the weights and biases in your [1, 6, 1] network according to the formula given above, and set it training! Make sure you are, as before, training on the train set and testing on the test set to get your mean squared error.

The results from the [50, 1] perceptron were about an mse of 0.005, sometimes as little as 0.0045. If you give this RNN a couple dozen epochs at a learning rate of 0.01, you should see an mse of 0.0036 or less! It's better! We did it!

Bonus Experiments (if you feel like playing around some more)

1) If this RNN, with the same approximate number of parameters as the [50, 1] perceptron, does not produce results you find sufficiently convincing, try a bigger one. A [1, 20, 20, 1] network takes a little longer to train but should get an mse of .0026 or less. (Took me about 100 epochs at lambda = 0.01.) Because our sequences have a certain amount of random noise in them, we'll never get to zero; this is pretty great!
2) You might be asking "does that weight/bias matrix initialization trick work well on DNNs too"? Why yes it does! If you're curious, feel free to apply the initialization radius formula to your MNIST code. You should get a network that converges notably faster. If you do try this, I'd love to read any results you get on Mattermost!

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 and document match the specifications above.
- Your code just runs (no command line input necessary). It should generate the train and test sets (same as RNNs 1), then run an RNN with dimensions [1, 6, 1] 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!