Generative AI Part 1

Eckel, TJHSST AI2, Spring 2024

Background & Explanation

Generative AIs like Large Language Models are extraordinarily complex, using a variety of techniques sequentially developed over many years by enormous research teams with resources far beyond anything we can do. But the simple idea at the core of a Generative AI is not so complex, and we can make one that achieves surprisingly good results with only the techniques we've learned so far (and some tweaking of activation functions & data reading strategies).

The core idea is this: take any kind of sequence (in our case, characters that comprise a text written in English) and break it into pieces of a certain size. Use each piece to run a training step on an RNN (eg, use 100 letters to predict the next letter). Do many of these training steps to train your network. Then, use that same network to start with a small input seed and just continually predict what comes next – after generating each letter, put that letter back into the network as the next input, and predict the next. Repeat as long as you want to generate text.

It is, as with many ideas in our study of AI, an elegantly simple idea. Implementing it has quite a few subtleties to contend with, though, that we'll need to work through carefully. Let's begin with talking about activation functions & architecture.

A More Complex Understanding of Activation Functions & Network Architecture

In each of our networks so far, we've used identical network structure and activation functions at each layer. The sizes of each layer have been different, but not what type of node it is or how the math is done. We haven't made networks with different activation functions at different layers, or where some layers are RNNs and some layers are DNNs/perceptrons. We will need to do both of these things to get a text-generating AI to train effectively.

First off – architecture. It turns out that (perhaps intuitively) recurrent layers do better when they send more information to the next step, ie, when the layers are larger. In our previous 1-6-1 network, most of the benefit we were getting was from the 6 layer; the 1 layer doesn't really help much. But it *does* make the network a lot harder to train; each recurrent layer complexifies the math greatly. So, for this assignment, we won't use recurrent nodes at each layer. We'll have an input layer, a couple hidden RNN layers, and then a final DNN layer just to handle output that doesn't pull double duty as an output AND a recurrent layer. This will allow us to train more quickly.

In addition, that last independent DNN layer will use a different activation function & measure of error. So far, we've used mean squared error as our error function, but one can define many others, and when training on data that is categorical (like "which letter is next") as opposed to continuous (like "which value between -1 and 1 should this be"), there's another error function that produces faster convergence & better results. This is the *cross categorical entropy* error function, which operates on the output of a *softmax* activation function.

Let's define these and do some math.

Softmax Activation Function

The softmax activation function works like this. You take all of the $dot^{N,F}$ values and form a new set of numbers by doing e to the power of each value, so that each one is a positive number. Then you sum all of these new values and divide each individual value by the sum, giving a probability distribution (a set of numbers that all add up to 1). You can think of this like the odds that the network thinks each particular possibility is next.

In python code, this looks like this:

```
temp = np.exp(dot_N_F)
a_N_F = temp / np.sum(temp)
```

…where you replace "dot_N_F" and "a_N_F" with however you store the final **dot** matrix and the final **a** matrix. Note that this is *scalar division* – each value in temp is divided by the sum of all values in temp, normalizing those numbers into a probability distribution that sums to 1.

Looking ahead, when we code our letter-generating AI, our final layer's output will be one number for each character, and will represent the probability that our network thinks each particular character should be the next one. We'll use that probability distribution to choose the next letter.

It also lets us use an error function which is much friendlier in categorical networks.

Cross-Categorical Entropy

Once we have an output from our network using the softmax activation function, where all the values sum to 1, we can compare it to an actual probability distribution, the y-vector containing the desired output of the network. In this particular case, we know with certainty the single letter that came next, so any y-vector will be a bunch of 0s at all the other letters and a single 1 at the correct letter. The simplest probability distribution there is!

But in any situation where we want to compare two probability distributions like this and determine the error, our situation included, we can quantify the error using cross-categorical entropy.

The formula, stated mathematically, is this. This is using variables in different ways than our network is using those variables, but it's necessary to do so to state the formula mathematically. So note that here N is not the number of layers, it is the number of values in the final output, and i as a subscript represents a single one of those outputs:

$$CCE = -\sum_{i=1}^{i=N} y\_actual_i * \ln(final\_layer\_output_i)$$

You might notice some deep connections between this and the entropy equations we used in Decision Trees! The difference here is that we're using the actual value as the probability, and the log of the *predicted value* as the information content. (We use base e here, instead of base 2, though this doesn't really change the conceptual idea here, it just means it lines up less well with the exact number of bits necessary to represent that entropy. We make the change just to make the math easier.)

This has the following effects:

- If $y\_actual_i = 0$, then that term of the sum is just 0, and does not contribute to the error at all. This error function therefore doesn't care how likely your code thought all the *wrong* answers were.

- If $y\_actual_i = 1$, then note ln(1) = 0, so if the predicted value is also 1, there is no error. But if the predicted value is *less* than 1, we get a negative number for the ln, which is then multiplied by the negative sign in front of the summation symbol. In other words, the further away the predicted value is, the greater the positive error.

In other words, using this error function means that our network strengthens *only the connections that lead to the right answer* at each training step, leaving the other answers to be strengthened by the data points that lead to those answers directly. It may not immediately intuitively obvious why this results in faster training, but it sure does.

Derivatives of CCE and Softmax

This is quite a complex piece of math to work out, especially if you aren't comfortable with calculus. I've left the derivation of this as a black assignment – if you'd like to see the math here, by all means do so! That assignment also contains some links to a helpful website that might help you figure it out, though they do use slightly different notation from us.

But for this assignment, I'm just going to give this to you.

When we used MSE, then the first delta matrix was:

$$\Delta^{N,F} = A'(dot^{N,F}) \otimes (y - a^{N,F})$$

All you need to do to implement Softmax / CCE in your back propagation is replace that line of code with this one:

$$\Delta^{N,F} = (I + -1 \cdot a^{N,F}) \cdot y$$

…where **I** represents an identity matrix (a square matrix of all zeroes except for 1s on the diagonal from upper left to lower right) that is of dimensions height(y) x height(y). You can make this in python, with numpy as np, with the command `np.eye(y.shape[0])`.

You may also notice that this addition is a bit funny – the **I** matrix is square, and the **a** matrix we're adding is a column. Same height, but only a width of 1. Numpy handles this by automatically *broadcasting* across the missing dimension; in other words, the top value of $-a^{N,F}$ is added to every square in the top row of the I matrix, and so on down each value/row, before dot multiplication with y. Since numpy does this automatically, the python syntax here for the whole formula is just this:

```
delta_N_F = (np.eye(y.shape[0]) + (-1 * a_N_F)) @ y
```

…where you replace "delta_N_F" and "a_N_F" with however you store the final **delta** matrix and the final **a** matrix.

Brief Tangent: Improve MNIST

Go back to MNIST, and add in the initialization radius formula we discussed in RNNs 2 as well as this new activation function on the last layer, the 10 node output layer, only. Once again when testing just take the largest value output by the last layer as your classification. You should see MNIST converges more quickly and does better! For example, my error on the original version was 4.41% misclassified but the new version got down to 2.18% (!!) misclassified after the same number of epochs & learning rate (6 epochs, lambda = 0.05). I haven't had a chance to test distortions yet; if you did that, see if this improves that performance as well. Let me know your results on Mattermost.

Preparing Letter Data

Ok: time for the main event. We need to read in the data and get it ready for training and train our network.

Download Shakespeare.txt from the course website or choose another text source of your own devising. To help save time, we probably want to make any text we read into all lowercase. After that, you should be left with a string of characters – many lowercase letters, spaces, newlines, some digits, etc. Our classifier will treat all characters the same, and simply use the characters so far to predict the next one.

We want to store our characters, both input and output, as **one-hot vectors** – vectors where all the values are 0 except for a single 1. So, we need to first count how many distinct characters exist in our text, and make all of the characters into one-hot vectors of that length with a single 1. We need to decide which index refers to which character.

In the absence of a straightforward way to do this, since we're not sure what set of characters will be present in our text, we'll just count how many times each character occurs. The most common character (probably space) will be index 0. The next most common (probably e) will be index 1. Etc, etc.

Read in your text, and:

- Pair each character that appears at least once in your text with a specific index that will represent that character in the one-hot vectors you'll build later. Base this index on how often that character appears in your text. Store this as two dictionaries (or the equivalent thereof) for easy lookup back and forth.
- Choose a length of character chain you want to use to train your network. The longer it is, the longer it'll take to train, but the better your results will be. For Shakespeare.txt I chose training data of 100 characters > 1 character, for 101 character substrings total. This took a couple of days to train (and was continuing to improve still when I stopped). If you're willing to leave your computer running overnight a couple of times, then you might want to aim similarly high. If you can only give this a few hours, maybe 20 characters > 1 character, and be ok with getting a bit more chaotic results.
- Generate all substrings of your text of the given length. Make sure you make ALL the substrings, including overlapping substrings! In other words, there should be a substring starting at index 0, at index 1, etc.
- Randomly shuffle and/or select from your total set of substrings so about 15-20% of them are in a testing set that **will not be used for training**. Make sure you do this **randomly** – selecting the first or last 15% will skew the results.
- It is important for this assignment that your train and test sets are consistent, and that multiple different Python scripts can access them. But: **if you turn your train and test sets into one-hot vectors, they get too big!** Shakespeare at 100-character strings makes 8 GB of training data! So: **pickle or otherwise save your train and test sets as .txt strings or as int lists**, not as lists of one-hot numpy arrays. In order to have reasonable memory footprint, we'll convert the training data into one-hot arrays while we train.

Do this carefully, verifying each step along the way.

Make sure your training and testing sets are saved/pickled (along with your correspondence of characters to one-hot indices), so your code won't have to recreate them every time and you can make sure your results remain consistent.

Choose a Network Architecture and Prepare Training Algorithm

Let's let C = the number of different characters in your text. You'll want a network with C inputs and an output layer of C standard perceptrons, receiving only input from the previous layer (not the previous step), fed through the softmax activation function. In between, you'll want two hidden layers (again, feel free to try more, but training time will expand a lot). Each of these layers should be recurrent, accepting input from the previous step & previous layer.

For Shakespeare.txt, I chose a C > 128 > 128 > C network, with the two 128-node layers being recurrent. This, again, took a couple of days to train. For smaller text chains, try C > 40 > 40 > C. The activation function for the recurrent layers can be either tanh or sigmoid. I got slightly better results with tanh.

Prepare your training algorithm; this will require more complex code than before. For instance, you won't be back propagating the last layer through time – the last layer is just a dense layer, and only propagates down. (So, you don't even need to calculate the last layer for any step other than the last one in each training input!) Make sure that back propagation calculates the last delta using softmax + CCE error (as shown earlier in this assignment), propagates that back to the previous *layer* only, and then from *there*, do the complex back propagation necessary to back propagate through both layers and steps for the remaining layers.

You'll also need your training function to convert your training data into one-hot vectors on the fly, so as to not run into the memory footprint problems mentioned earlier. One entry in your training set is a string. As you train, step by step, you'll want your code to take each letter and convert it to a 2-d column one-hot vector, then feed that to the network.

Set your code so that every so often, it saves the network specification (weights and biases). I wouldn't wait until the end of an epoch to do so if you have a large number of training data points. (Shakespeare.txt has more than a million.) Maybe save your progress every 10,000 data points. **Don't overwrite your files. Save a new file each time. It will be interesting to compare later, and this is part of the spec for your next assignment.** For example, I saved each one as a file with a name like Shakespeare_234000.pkl where 234000 was the number of data points I'd trained, and kept making new files as that went up.

Start it training. I found I needed a very small learning rate; I used 0.001. I genuinely don't know how different architectures might need different values.

IMPORTANT: Verify Your Setup

There are a lot of subtleties here. Take a moment to double check. Have you done all these things?

- Train and test sets are **distinct** (that is, you didn't ever put the same text string in both) and **saved as separate files**, as text strings or int lists.
- You have also saved **your dictionaries matching characters to one-hot indices**.
- You used **every possible overlapping text string** when building your train and test sets.
- Your network has a **final DNN layer** that is **not recurrent** that uses the **softmax** activation function.
- Your network has **two RNN layers** that **are recurrent** that use **tanh or sigmoid** activation functions.
- You are saving your network specification every so often in **different files every time** so you can recreate your progress later.

Once you're sure you've got all this right, let it train for a while and then build some code to check that your training seems like it's working. Specifically…

Check Your Progress

Write a separate file to load your pickled testing set & the dictionaries matching characters to one-hot indices, then load a pickled network, and run some tests. You can do this in several ways:

- Very quickly, within the first 10,000 training data points, your code should start to realize that the most common characters occur more often. If you look with your eyeballs at the actual column vector output of the network for a few particular testing data points, you should see very early on that the probabilities start to get weighted towards spaces and common letters (e, t, a, o, i, n, s). This is a good sign. If your code consistently thinks $ is the most likely next character, you may have a problem.
- You can run through the test set and calculate cross categorical entropy error, numerically, and then find average CCE across the data set. This should notably decrease after 10,000 data points, then 20,000, etc.
- You can count the number of misclassified letters on the testing set. This is likely to be quite high at first (all or nearly all characters misclassified), don't worry! But it should slowly diminish over time.

One thing I found was that, when my learning rate was too high, my network would overwhelmingly decide that a certain character was the best one, and would return that character for *every* input string (or nearly so). So, if you find after 30,000 or 40,000 data points that *every output thinks "n" is the most likely*, or something like that, your learning rate is probably too high.

There are probably other common mistakes here too. I look forward to discovering them with you!

Specification

Send me a message about how you've tested to make sure your training is actually making progress across the first 100,000 data points or so. (Depending on the size of your data set, this might be a fraction of an epoch or multiple epochs.) Then, settle in and be patient.

No code submission for this one. You have credit if you send me your revised MNIST results ("Brief Tangent: Improve MNIST") and some evidence your network is improving (see previous page). I'm interested to see what results we get, and reserve the right to get more specific here as I learn over time what makes the most sense.

Next Up: Generating Text!

Once you verify that this is working, **let it keep training** while you work on the next part of the assignment. Make sure it's saving a new .pkl file each time it saves, keeping track of its progress.