An extremely rough version of what was supposed to be a concomitant narrative of the code. But I have read up on how blogs are never perfect, and the goal is to push something, and iterate after you have committed. So this rough description of the code will get better with time.

Credit: WildML, fastai,

Reasons for post - learn by reverse-engineering something (Geoffrey Hinton) and demonstrate via an implementation how Recurrent Neural Networks (RNN) works. There are many blog posts (I presume) that explain RNNs but I hope to do so in a way that intuitively makes sense to me. Hopefully that adds clarity to those searching. I will first go through a

* Post will work through the details of back propagation and mini-batch gradient descent
* My workings, and my understanding are far [\*3 for emphasis] from perfect, and I am embarrassed about this. But I still choose to publish my code and my blog (as it is) so that I can be told how best to improve it (rather than assuming and trying things). For that reason, I welcome good critique.

Taking the fast.ai course (Practical Deep Learning for Coders), I recently learned about language modelling and Recurrent Neural Networks (RNNs). It is an old-ish neural network architecture, aimed at improving memory of elements in a sequence, unlike vanilla (link to meaning of vanilla) neural networks. Between now and then, there have been several derivatives of RNNs used for different language tasks (link to top NLP RNN applications), the latest breakthrough being GRUs.

This blog post will chiefly be concerned with understanding the inner workings of Vanilla RNNs, and not the basics about how RNNs work. If RNNs are new to you, I would suggest going over these links (in order of progressive difficulty): link, link, link, ...

The problems I tend to think of on a regular basis end up having NLP as part of solutions that I reflect upon, which furthers my interest in related deep architectures. The

Problem being solved: generating Nietzsche-like content using a 3-character RNN model with 3 time steps (fast.ai course)

Simple architecture used in my first iteration so that I can effectively learn important parts such as backpropagation and the minibatch training loop.

Rather challenging concepts:

* Consolidating dimensions of all layers
* Calculating gradients wrt errors.

Model simplifications:

* Architecture of the RNN ie sequence of functions used
* Batch size of 1 - to reduce one dimension so that matrix dimensions don’t go out of control for me. A larger batch size would only require adding an additional dimension
* Indexing of categorical data (ie the characters) and not representing each category with an embedding vector. This is deprecated because it assumes that each category is equidistant in n-dimensional space, which is likely not the case since different characters have different relationships with other characters.
* I could have chosen to use torch in order to make gradient calculations easier. Although that simplifies the process significantly, I chose to use numpy so that I could demonstrate reverse mode differentiation and the calculation of gradients in code. Like Geoffrey Hinton said about learning by reengineering.
* Start with small dataset so it is quicker to debug, then use full dataset. Good practice in general (link Karpathy’s top neural network training hacks)

RNN code implementation to-do:

* Add javadoc like comments while you try to understand so it will be easier to come back to it next time -- find out what python equivalent of javadoc is
* Modularize gradient calculations for each type of function -- will be easier to see opportunity for shortening code.
* Do an iteration of bidirectional RNN
* Start with small dataset -- overfit if poss.
* Calculate local gradient for every computational node while doing forward prop, that way, it is easier to keep track of the inputs and outputs in code.
* If can’t figure out the derivative of matrix wrt another matrix -- symbolically figure out by brute force what the element wise derivatives are, and then generalize.
* Figure out how the data goes into the RNN - see p 34 of handwritten notes
  + Show schematic shown on p34 of notes.
  + How to prepare data (steps):
    - Decide on batch size
    - (Total number of chars in text) / (mini-batch size) = intermediate
    - (intermediate) / (time-steps = ‘bptt’) = number of mini-batches
    - Each ‘training example’ in a mini-batch would then be a sequence of characters of length = time-steps in arch = ‘bptt’
    - Each training example in a mini-batch will be unrelated to another training example in the same mini batch because they will be far spread out (choose a relatively small mini batch size e.g. 32,64,128)
    - For simplicity, choose batch size of 1 and then increase it later
    - Need to pass training data and test data at each iteration. Training data will be mini-batch of size bptt, ie from characters i to i+bptt. Test data for each batch would be data staggered one character down. Therefore batches for train and test data can be done separately. Next mini-batch’s train data will start at the last character of the previous mini-batch’s test example.
    - Once those respective containers are created, must create lists of first, second and third chars across mini-batches. Therefore, during training and testing, just iterate over that one list for the specific character.
  + Form of data required to go into RNN during training per iteration:
    - A list of inputs for each time step. Size = # of baches \* bs \* time\_steps and the same for real outputs, but staggered one down
    - The corressponding y values are of size: # of batches \* bs \* time\_steps \* # of categories.
      * Additional dimension for the one hot encoding vector for each output value
      * One hot encoding determined by using scikit-learn’s OneHotEncoder in sklearn.preprocessing.

Acknowledge how to make better:

* Gradient checking
* Larger batch size
* LSTM/GRU instead of vanilla RNN