Vanilla Reccurent Neural Network Architecture in NumPy

Aaron Avram

June 18 2025

Introduction

In this writeup I will describe my process in building a full Vanilla RNN from scratch in NumPy for name generation. This project is a kind of extension to my NameNet project which used a MLP to generate names. Now, I will use an RNN to learn a representation of an input context then feed this into a MLP to predict the next token in the sequence.

Class Design

I borrow a lot of design from my ImageNet project, but here I hope to make my work more general. I want to develop a Network class that supports different kinds of layers, just as my other project does. However, adding recurrent layers is tricky, as they work quite differently from feed forward layers. Thus, I will introduce a more abstract Recurrent class that extends layer, which has a cell. This cell keeps track of the update step and the parameters of the layer. This choice should allow me to plug in different types of recurrent layers as I build my project.

The Recurrence

Fundamentally, a recurrence is an inductive process that takes an initial state and sequentially applies an update rule. One could think of this as taking steps in time between each update. Now there are some extra details. So consider a sequence of input vectors $x^{(1)}, \dots, x^{(\tau)}$, then given an initial state $h^{(0)}$ and an uptate rule explicated by a function f parametrized by θ and x (an arbitrary input value) we can define:

$$h(t) = f(h^{(t-1)}; \theta, x^{(t)}), \text{ This is our hidden state at each step } t$$

In terms of output, some RNN's produce outputs at each step by applying something like a softmax to the hidden state at that value. This way you can produce an output of the same length as your input. Additionally, the predicted can be added to the recurrence. However, for my use case, it makes sense to only have one output, which will be the result of applying the softmax function to the final hidden state $h^{(\tau)}$ to get a probability distribution over all characters.

Note, that our function f will be a linear function applied to the inputs, then passed through an activation function, say tanh:

$$\begin{aligned} z^{(t)} &= Wh^{(t-1)} + Ux^{(t)} + b \\ h^{(t)} &= \tanh(z^{(t)}) \end{aligned}$$

Where W and U are square matrices and b is a vector.

This formulation allows us to scale up the input processing to receive batches of input vectors, with each example stored as a row. Where $h^{(t)}$ becomes a matrix with its ith row specificying hidden state of the ith example in the batch at step t. Thus the update step scales up to:

$$\begin{aligned} z^{(t)} &= h^{(t-1)}W^T + x^{(t)}U^T + b \\ h^{(t)} &= \tanh(z^{(t)}) \end{aligned}$$

1 Gradient Calculations

The gradient step is actually quite simple. In this scenario we are working with a final activation using softmax and a loss function computed via the negative log likelihood. So for a training batch of size n, with d classes of outputs:

$$\begin{aligned} L &= -\frac{1}{n}\sum_{i=1}^{n} \log(\Pr(y_i | x_i^{(1)}, \ldots, x_i^{(\tau)})) \\ &= -\frac{1}{n}\sum_{i=1}^{n} \log\Big(\frac{\exp(h_{iy_i}^{(\tau)})}{1 + \sum_{j=1}^{d} \exp(h_{ij}^{(\tau)})}\Big) \\ &= -\frac{1}{n}\sum_{i=1}^{n} \Big(h_{iy_i}^{(\tau)} - \log(1 + \sum_{j=1}^{d} \exp(h_{ij}^{(\tau)}))\Big) \end{aligned}$$

Now taking the partial derivative with respect to h_{lk}:

$$\begin{aligned} &= -\frac{1}{n}\Big(\delta_{k(y_l)} - \frac{\exp(h_{lk}^{(\tau)})}{1 + \sum_{j=1}^{d} \exp(h_{lk}^{(\tau)})}\Big) \\ &= \frac{1}{n}(\Pr(h_{lk} | x_l^{(\tau)}, \ldots, x_l^{(\tau)}) - \delta_{k(y_l)}) \end{aligned}$$

Then it is clear that $J(L)(h^{(\tau)})$ is:

$$J(L)(h^{(\tau)}) = \text{softmax}(h^{(\tau)}) - y_{\text{onehot}}$$

Now we can recursively calculate the necessary gradients. For W and U we will accumulate their gradients in a sum over each step. Now suppose for some $h^{(t)}$ we have $J(L)(h^{(t)})$ then, we can find $J(L)(z^{(t)})$ by elementwise multiplication with the tanh elementwise derivative:

$$J(L)(z^{(t)}) = J(L)(h^{(t)}) \odot \tanh'(z^{(t)})$$

Next we find the Jacobian of the weight matrix at this step, and to avoid confusion we denote the weight matrix with a t subscript to represent the fact that this is only the jacobian for the t step. First, remember: $z^{(t)} = h^{(t-1)}W^T + x^{(t)}U^T + b$. So:

$$z_{ij}^{(t)} = \sum_{k=1}^{d} h_{ik}^{(t-1)} W_{jk} + \sum_{k=1}^{d} x_{ik}^{(t)} U_{jk} + b_j$$

Thus, we have:

$$\frac{\partial z_{ij}^{(t)}}{W_{kl}} = \begin{cases} 0 & j \neq k \\ h_{il}^{(t-1)} & j = k \end{cases}$$

Then we can construct:

$$\begin{aligned} J(L)(W)_{ij} &= \frac{\partial L}{\partial W_{ij}} \\ &= \sum_{k=1}^{d} \frac{\partial L}{\partial z_{ki}^{(t)}} \frac{\partial z_{ki}^{(t)}}{\partial W_{ij}} \\ &= \sum_{k=1}^{d} \frac{\partial L}{\partial z_{ki}^{(t)}} h_{kj} \\ J(L)(W_t) &= [(h^{(t-1)})^T][J(L)(z^{(t)})] \\ J(L)(W) &= \sum_{t=2}^{\tau} [(h^{(t-1)})^T][J(L)(z^{(t)})] \end{aligned}$$

Note that this is only at each time step, we then sum up these sort of partial jacobians to get the entire derivative matrix of the weight matrix. We can repeat this process for U to find that $J(L)(U) = \sum_{t=2}^{\tau} [(x^{(t)})^T][J(L)(z^{(t)})]$. Now we look to h:

$$\frac{\partial z_{ij}^{(t)}}{h_{kl}^{(t-1)}} = \begin{cases} 0 & i \neq k \\ W_{jl} & i = k \end{cases}$$

Next:

$$\begin{aligned} J(L)(h^{(t-1)})_{ij} &= \frac{\partial L}{\partial h_{ij}^{(t-1)}} \\ &= \sum_{k=1}^{d} \frac{\partial L}{\partial z_{ik}^{(t)}} \frac{\partial z_{ik}^{(t)}}{\partial h_{ij}^{(t-1)}} \\ &= \sum_{k=1}^{d} \frac{\partial L}{\partial z_{ik}^{(t)}} W_{kj} \\ J(L)(h^{(t-1)}) &= [J(L)(z^{(t)})][W] \end{aligned}$$

Similarly for each $x^{(t)}$: $J(L)(x^{(t)}) = [J(L)(z^{(t)})][U]$. This will be important, as the model is fed embeddings of each character, and the embeddings matrix will be trained alongside the other model parameters. Thus consider a index form of some character w, then what we feed in is $x := Cw$ if C is the embeddings matrix. Thus, $J(L)(C)(w) = J(L)(x)$, so our jacobians for each $x^{(t)}$ contains the derivative information for a row of the embeddings matrix. Finally, we must find the jacobian of the bias, but it is clear that $J(L)(b) = \sum_{t=2}^{\tau} J(L)(z^{(t)})$