Federated Learning Project

Aaron Belikoff

October 2024

1 Abstract

Training Recurrent Neural Network (RNN) models is a difficult process due to the interconnected nature of each model step. In writing this, I aim to show you my process of training a RNN to generate text in the style of Shakespeare, and how I worked around problems that came up along the way. By using distributed training, I sidestep many of the problems that plague RNN training, and am able to train a RNN from scratch across a network of computers.

2 Key Terms

- **Hidden State:** A hidden state is a vector that is output from the RNN at one step, and taken as input at the next step. It is called a state because it contains the current state of the input, as the hidden state is dependent on the inputs of all previous steps. It is called a *hidden* state because it is not directly interactable by anything outside of the function, i.e. it is not a text-related input or output.
- **Parameters:** Parameters are also inputs to the function, but they are not usually thought of as inputs. Consider the equation $f(x) = mx + b$ where m and b are parameters, and x is an input, and we set a value y to be a target output. We are treating this as an optimization problem where x and y are fixed, so we are trying to answer "What values of m and b will minimize the difference between $mx + b$ and y". Because m and b are not fixed, we call them parameters to indicate that we are trying to find values for them that minimize the difference between the function output given the input x and the target y.
- **Embedding Vector:** An embedding/representation vector is a vector that represents *something.* In our case, that *something* is a letter, so we can say "The embedding vector of 'a' is ..." which means we are defining that vector to be the representation of the letter "a".

3 What is a Recurrent Neural Network (RNN)?

When I say RNN I am referring to a function

$$\vec{H_n}, \vec{y_n} = f(\vec{H_{n-1}}, \vec{x_n})$$

Specifically, for one "step" of the model,

$$\vec{H_n} = tanh(\vec{H_{n-1}}\mathbf{W_{hh}} + \vec{x_n}\mathbf{W_{ih}})$$

$$\vec{y_n} = \vec{H_n}\mathbf{W_{ho}}$$

Where

- $\vec{}$ denotes a vector.
- $\mathbf{W_{hh}}$, $\mathbf{W_{ih}}$ and $\mathbf{W_{ho}}$ are matrices.
- A subscript n means the item is from the current step.
- A subscript $n-1$ means the item is from the previous step.

- $\mathbf{W_{hh}}$ has a subscript hh which stands for hidden-to-hidden.
- $\mathbf{W_{ih}}$ has a subscript ih which stands for input-to-hidden.
- $\mathbf{W_{ho}}$ has a subscript ho which stands for hidden-to-output.

A RNN is recurrent because it takes the hidden state from the previous step, $\vec{H_{n-1}}$, as an input to the current step, $f(\vec{H_{n-1}}, \vec{x})$, to produce the current hidden state $\vec{H_n}$. So its output gets fed back in at input for the next step.

We add the tanh function for two reasons. One, we need to add some non-linear "activation" function in-between successive vector-matrix multiplications in order to make sure these different vector-matrix multiplications are having an effect. If we have two in a row without an activation function, we can reduce those two linear transformations into one linear transformation.

$$\vec{x_1} = \vec{x_0}\mathbf{W_0} + \vec{b_0}$$

$$\vec{x_2} = \vec{x_1}\mathbf{W_1} + \vec{b_1}$$

$$\vec{x_2} = (\vec{x_0}\mathbf{W_0} + \vec{b_0})\mathbf{W_1} + \vec{b_1}$$

$$\vec{x_2} = \vec{x_0}\mathbf{W_0}\mathbf{W_1} + \vec{b_0}\mathbf{W_1} + \vec{b_1}$$

$$\mathbf{W} = \mathbf{W_0}\mathbf{W_1}$$

$$\vec{b} = \vec{b_0}\mathbf{W_1} + \vec{b_1}$$

$$\vec{x_2} = \vec{x}\mathbf{W} + \vec{b}$$

So we need a non-linear function, but why `tanh`? Because we are using an RNN for our model, the hidden states in our model will be multiplied N times, 1 time in each step of the sequence. This means that if a value c is > 1 it will diverge towards infinity because

$$\lim_{x\to\infty} c^x = \infty$$

This is a big problem because of the aforementioned numerical instability of large numbers. Choosing `tanh` as our activation function gets rid of this problem because `tanh` maps all real values $\mathbb{R}$ to $\{x \in \mathbb{R} \mid -1 < x < 1\}$.

4 What does it mean to train a RNN?

If we want the RNN to generate text in the style of Shakespeare, we need two things.

1. A way to pass text through the function, so the input and output is text.
2. Values for the parameter matrices of the RNN, $\mathbf{W_{hh}}$, $\mathbf{W_{ih}}$ and $\mathbf{W_{ho}}$, that make the function generate *Shakespearean* text and not just random letters.

4.1 How do we have an RNN's input and output be text?

The RNN's input and output are vectors. This means we need to somehow represent text as vectors. There are many approaches to this, but one of the simplest ways to do it is have each symbol in our vocabulary have it's own unique number, and we will call these numbers "tokens". Then we can make each token be its own vector where every element except its number is 0, and its element is 1 [1].

Lets see what this would look like if our vocabulary was just the lowercase alphabet.

$a = 0, b = 1, c = 2, d = 3, e = 4, ..., z = 25$

Then because the token for a is 0, we define the embedding vector for a to be

$$\vec{a} = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$$

because we set the 0th element of the vector to 1, and all the other elements to 0.

We define the embedding vector for b to be

$$\vec{b} = [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$$

We define the embedding vector for c to be

$$\vec{c} = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$$

etc.

There are 26 elements in each embedding vector because our vocabulary (the alphabet in this case) has 26 symbols in it. If we decided to use the uppercase AND lowercase alphabet, each vector would have a size of 52 elements (26 uppercase letters + 26 lowercase letters), but still only one element in each embedding vector would have a value of 1.

So now when we want to pass a symbol/letter as input to the RNN, we just need to get its corresponding embedding vector, and then pass that vector as input.

And for the output, it is a little bit different. The output vector will not be a perfect embedding vector of all zeros except for one 1. So we instead scale the output vectors so all the elements are between 0 and 1, and the sum of the elements is equal to 1. This scaling allows us to treat these numbers as probabilities, and randomly sample from the probabilities to get our output symbol. So if after scaling the output vector the value for "a" is 0.3 (remember that the value for a is the 0th element of the vector because the token for "a" is 0), that means that "a" has a 30

To scale the output vector we use a function called `softmax` where the ith element of the scaled vector is

$$y_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

Where

- x is the unscaled vector.
- y is the scaled vector.
- n is the number of elements in the vector.

There are a few reasons we choose this specific function for scaling our output vector.

1. It is a non-linear scaling function which gives the model more expressive power.
2. It can handle negative numbers.
3. The softmax output is invariant to scaling which means we can shift x by a constant c, and as long as all elements are shifted by the same c, the output will be the same. This is useful for actually calculating the scaling on computers because it helps us avoid instability issues with very large numbers.

4.2 How do we find the right parameters?

In order to find the right parameters we need two things:

1. A loss function which tells us how "good" any specific parameters are.
2. To optimize that loss function to find the best parameters. Our goal is to have a RNN that writes Shakespearean text, so if we have a dataset of all of Shakespeare's works, our goal is to minimize the difference between the output text of our model, and the actual text.

4.2.1 Loss Function

The loss of a single scaled output vector, given a correct output token, is

$$L(x, token) = -\ln \frac{x_{token}}{\sum_{j=1}^{n} x_j}$$

Where

- x is the scaled output vector.
- x_{token} is the *token*th element of the scaled output vector. *token* is the token of the correct output symbol/letter.

Intuitively, this function means that when the probability of the correct token is 1, then the loss is $-\ln 1 = 0$, and as the probability of the correct token decreases towards 0, the loss approaches infinity.

In order to get the loss of a specific set of weights, we need to:

1. Get the output vectors for every symbol in our text dataset.
2. Get the loss of all those output vectors using their corresponding correct output symbol.
3. Take the mean of all those losses to get our final loss.

4.2.2 Parameter Optimization

Now that we have the loss function, we need to find the values of the parameter matrices $\mathbf{W_{hh}}$, $\mathbf{W_{ih}}$ and $\mathbf{W_{ho}}$ that make the output of the loss function as low as possible.

One approach for optimization is to find the critical points of the function, and then evaluate the critical points to find the one with the lowest value.

Another method to optimize a loss function that does not require finding the critical points is called Gradient Descent. Gradient descent is an iterative method where you find the gradient of the loss function at a certain point in the parameter space, then take a small step in the opposite direction of the gradient, which will (hopefully) decrease the loss function.

Algorithm 1 Gradient Descent

```
Input: Learning rate $\alpha$, noise standard deviation $\sigma$, initial policy parameters $\theta_0$
for t=0, 1, 2... do
    $L_t \leftarrow L(\theta_t)$
    Get $\frac{\partial L_t}{\partial \theta_t}$
    $\theta_{t+1} \leftarrow \theta_t - \alpha \cdot \frac{\partial L_t}{\partial \theta_t}$
end for
```

Where

- θ_t is a vector containing all of the RNN parameters at timestep t.
- L_t is a scalar loss value.

This method is widely used to optimize functions/models that are too large or complex to efficiently solve with exact methods. In practice the datasets are usually much too large to compute the loss over the whole dataset, so we usually use a random sample of *batch* items from the dataset. Before starting training we define *batch* to be some positive integer of our choosing.

In order to calculate the gradient of the loss w.r.t. the RNN parameters, we can use the chain rule. The derivatives for the last step of the forward pass are

$$\frac{\partial L}{\partial \mathbf{W_{ho}}} = \frac{\partial L}{\partial \vec{y_n}} \cdot \frac{\partial \vec{y_n}}{\partial \mathbf{W_{ho}}}$$

$$\frac{\partial L}{\partial \vec{H_n}} = \frac{\partial L}{\partial \vec{y_n}} \cdot \frac{\partial \vec{y_n}}{\partial \vec{H_n}}$$

$$\frac{\partial L}{\partial (\vec{H_{n-1}}\mathbf{W_{hh}} + \vec{x_n}\mathbf{W_{ih}})} = \frac{\partial L}{\partial \vec{H_n}} \cdot \frac{\partial \vec{H_n}}{\partial (\vec{H_{n-1}}\mathbf{W_{hh}} + \vec{x_n}\mathbf{W_{ih}})}$$

$$\frac{\partial L}{\partial \mathbf{W_{hh}}} = \frac{\partial L}{\partial (\vec{H_{n-1}}\mathbf{W_{hh}} + \vec{x_n}\mathbf{W_{ih}})} \cdot \frac{\partial (\vec{H_{n-1}}\mathbf{W_{hh}} + \vec{x_n}\mathbf{W_{ih}})}{\partial \mathbf{W_{hh}}}$$

$$\frac{\partial L}{\partial \mathbf{W_{ih}}} = \frac{\partial L}{\partial (\vec{H_{n-1}}\mathbf{W_{hh}} + \vec{x_n}\mathbf{W_{ih}})} \cdot \frac{\partial (\vec{H_{n-1}}\mathbf{W_{hh}} + \vec{x_n}\mathbf{W_{ih}})}{\partial \mathbf{W_{ih}}}$$

$$\frac{\partial L}{\partial \vec{H_{n-1}}} = \frac{\partial L}{\partial(\vec{H_{n-1}}\mathbf{W_{hh}} + \vec{x_n}\mathbf{W_{ih}})} \cdot \frac{\partial(\vec{H_{n-1}}\mathbf{W_{hh}} + \vec{x_n}\mathbf{W_{ih}})}{\partial \vec{H_{n-1}}}$$

We calculate $\frac{\partial L}{\partial \vec{H_{n-1}}}$ because we want to calculate the derivatives all the way through the n steps. In order to do this we need to keep track of the gradient so far for a valid chain rule calculation. So the general algorithm for backpropagation on one sequence of steps would be

Algorithm 2 RNN Backpropagation

```
$\nabla\mathbf{W_{ho}} \leftarrow 0$
$\nabla\mathbf{W_{hh}} \leftarrow 0$
$\nabla\mathbf{W_{ih}} \leftarrow 0$
$\nabla\vec{H} \leftarrow 0$
for t=n, n-1, n-2, ..., 0 do
    $\nabla\mathbf{W_{ho}} \leftarrow \nabla\mathbf{W_{ho}} + \frac{\partial L}{\partial \vec{y_n}} \cdot \frac{\partial \vec{y_n}}{\partial \mathbf{W_{ho}}}$
    $\nabla\vec{H} \leftarrow \frac{\partial L}{\partial \vec{y_n}} \cdot \frac{\partial \vec{y_n}}{\partial \vec{H_n}} + \frac{\partial L}{\partial \vec{H_{n+1}}} \cdot \frac{\partial \vec{H_{n+1}}}{\partial \vec{H_n}}$
    $\nabla\mathbf{W_{hh}} \leftarrow \nabla\mathbf{W_{hh}} + \nabla\vec{H} \cdot \frac{\partial \vec{H_n}}{\partial(\vec{H_{n-1}}\mathbf{W_{hh}}+\vec{x_n}\mathbf{W_{ih}})} \cdot \frac{\partial(\vec{H_{n-1}}\mathbf{W_{hh}}+\vec{x_n}\mathbf{W_{ih}})}{\partial \mathbf{W_{hh}}}$
    $\nabla\mathbf{W_{ih}} \leftarrow \nabla\mathbf{W_{ih}} + \nabla\vec{H} \cdot \frac{\partial \vec{H_n}}{\partial(\vec{H_{n-1}}\mathbf{W_{hh}}+\vec{x_n}\mathbf{W_{ih}})} \cdot \frac{\partial(\vec{H_{n-1}}\mathbf{W_{hh}}+\vec{x_n}\mathbf{W_{ih}})}{\partial \mathbf{W_{ih}}}$
end for
```

Now we have the gradient we need to do gradient descent. But these calculations depend on the hidden states, which means we need to store all n hidden states as we do the steps on the forward pass. This is not much of a problem at first, but as the sequences get very long, it is not feasible to use this method as it takes up too much space.

There is another way to calculate the gradient that uses a constant amount of storage no matter how long the sequence is. But method, while theoretically better when the sequences are extremely long, is very slow in practice due to the large amount of computation required.

These shortcomings lead us to a third option for training the model: distributed training.

5 How does distributed training work?

The basic idea of distributed training is using multiple workers together to train a model faster than you would be able to on a single worker. There are many different ways to approach this, but I am using evolution strategies [2] to train the RNN.

This means that instead of calculating the exact gradient, I am sampling points in the parameter space around the current parameter point and then using those samples to estimate the gradient. The exact algorithm can be found in the evolution strategies paper I cited (algorithm 2).

I want to focus more on the actual parameter update, and explain it a bit more in depth. The update rule is

$$\theta_{t+1} = \theta_t + \alpha \frac{1}{n\sigma} \sum_{i=1}^{n} F_i \epsilon_i$$

The reason this works is because we are getting a weighted sum of directions in the parameter space. The higher the reward of a sampled direction, the more weight it has in moving the parameters for the next step. And because F is normalized to have a mean of 0 and a standard deviation of 1, any scores below the mean will push the update gradient away from their position.

We then normalize the gradient by the number of points we sampled and the standard deviation that we used to sample the points before finally doing the parameter update.

This approach is beneficial because it eliminates the need to calculate the gradient exactly, allowing us to avoid massive space usage or massive amounts of compute and slow training. It also allows us to use as many workers as we want (because each sampled point can be tested independently), which gives us another large speedup.

6 Results

I did a final training run which unfortunately did not end with beautiful Shakespearean prose. When the model is initialized with random parameters, it's loss is around 4.33. This makes sense because the vocab size I am using is 76, and so the probability of randomly choosing the right token is $\frac{1}{76}$ and the loss associated with that is $-\ln\frac{1}{76}$ which evaluates to around 4.3307. A loss of 0 would be ideal but unrealistic. Just from personal experience, when the loss gets to the low 2s, you start to recognize words, and when you get to the low 1s, you can sometimes get recognizable sentences! But as you can see from the graph below, we did not get anywhere near that, with the loss hovering around 3.35 as I am writing this (I am still continuing the training run). There could be many reasons for this, but my top few guesses would be I didn't let it train for long enough, I didn't choose the right hyperparameters (learning rate for the gradient update, sigma for the federated learning, and many others), or I need to add more parameters to the RNN to make it able to model more complex relationships.

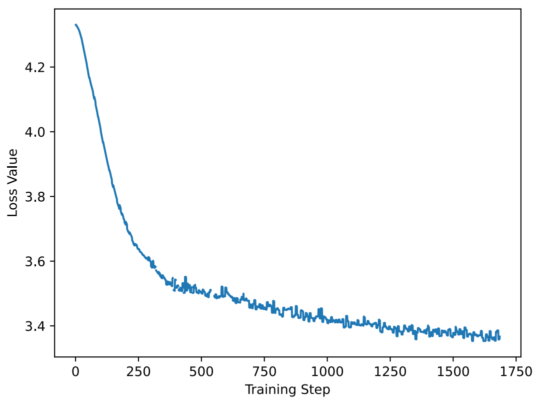

Figure 1: The loss curve for the final training run

The actual training data I used was paragraphs from shakespeares works. I downloaded all his works from the Folger Shakespeare Library `https://www.folger.edu/explore/shakespeares-works/`, and then split them into paragraphs (and I just made each of his sonnets 1 training sequence).

A couple (non-contiguous) randomly sampled paragraphs from the dataset:

1.
He sees her coming and begins to glow,
Even as a dying coal revives with wind,
And with his bonnet hides his angry brow,
Looks on the dull earth with disturbed mind,
Taking no notice that she is so nigh,
For all askance he holds her in his eye.
2.
MESSENGER
My honorable lords, health to you all.
Sad tidings bring I to you out of France,
Of loss, of slaughter, and discomfiture:
Guyen, Champaigne, Rheims, Roan, Orleance,
Paris, Gisors, Poitiers, are all quite lost.

And then here is my RNN doing its very best :)

nDEKBh Dhtin nelt nhr),w6i scela nyoy ?nicoa
nehnsn lghasso!eTatTe idagBhh et n oak eo rm
anes eai,us h..larsT,tyaOw o8ctgoYdnI -ifa pIicClm
ng tqip -h[nB gmRsee Dtr
nlfhi e]d p 8nBmIn Idhs n24eoaeusno
f o

f annnhiI,eaDsd iGs,ofctoR6h,K2pNo o Ite n ri hdtta t
neas2!zbset,unaesZbf w e”
f2efo ea ?NZ[telosroQ csn Oeoe itfte me

It is better than random, it is starting to learn how long words should be, and it is adding line breaks that create lines of roughly the right length but alas, it is not very Shakespearean.

All code for the project can be found at `https://github.com/EndeavoringOrb/federatedLearning`. The project is not well documented, but I plan to document it soon.

References

[1] Andrej Karpathy. *The Unreasonable Effectiveness of Recurrent Neural Networks.* URL: `https://karpathy.github.io/2015/05/21/rnn-effectiveness/`. (accessed: 10.08.2024).

[2] Tim Salimans - Jonathan Ho - Xi Chen - Szymon Sidor - Ilya Sutskever. “Evolution Strategies as a Scalable Alternative to Reinforcement Learning”. In: *arxiv.org* (2017). DOI: `https://doi.org/10.48550/arXiv.1703.03864`.