LANGuage IDentification

João Augusto Costa Branco Marado Torres

December 17, 2024

1 License

Contents

Table 1: Rainbow model vocabulary

ID	**Word**
0	red
1	orange
2	yellow
3	green
4	cyan
5	blue
6	violet

List of Figures

2 Phrases to Numeric Representation

I was really confused about this, but it really is not that complicated.

Before starting to write any code, I wanted to know how can I transform a phrase like the one you are reading right now into `1`s and `0`s because the computer isn't smart enough to know English. So I went to my friend and asked it some questions. It thought me various ways of achieving what I wanted, and between those options I picked the easiest to understand/implement.

2.1 Bag of Words

The idea it's to build a **vocabulary** which is a list of unique words.

Everyone (including us) has its own vocabulary composed by words of the languages we speak.

Our AI model needs a vocabulary too so he can understand some words. Those words might not exist in our vocabulary.

Let's say our model's vocabulary 1 are the words in English for the 7 colors in the rainbow.

Each word in the vocabulary will have a numerical identifier.

It makes sense for us for that identifiers to start at `0` and increment by `1` for each word.

Now our model can represent our phrases into something it understands.

How?

The question you just asked is represented as a zero column matrix with 7 rows.

The model will try to find words he knows from a phrase and represent it as a column matrix with rows the same as the amount of words in the vocabulary. Each entry represents a word in the vocabulary, here is why it's a good idea to identify the words as I said. The value of each entry can be a simple boolean that represent if a word is present in the phrase or not, or the amount of times the word appears in the phrase. The value really represents how important is the that word in the phrase.

The phrase "How?" does not contain any words know by the model.

The phrase "The colors in the *Guiné-Bissau* flag are red, yellow, green and black in the star." would be represented by the matrix $\begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 0 & 0 \end{bmatrix}^T$ and the phrase "What came first, the orange fruit or the orange color?" could be represented by $\begin{bmatrix} 0 & 2 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T$ if we count the amount of occurrences of the word in the phrase.

The way I implemented the BoW was by reading a file and collect every single word from it separated by white spaces, dashes, punctuation, parenthesis or quotation marks and then spit to `stdout` a formatted vocabulary, so you can pipe it later or write it to a file. Words are case-sensitive, and it's possible to create the vocabulary in alphabetic order.

My implementation can be found in `/src/commands/vocab/bow.zig`.

During the conversation, "tokenization" was mentioned, and it might be cool for you to take a look at it.

2.2 Other ways

- TF-IDF;
- One-Hot Encoding of Words;
- Embedding-like Approach;
- Sub-word tokenization.

3 Neuron

As explained at the start of chapter 3.1 of [Bis06], the base for the simplest supervised linear regression models will look like (1)

$$\begin{aligned} y(\mathbf{x}, \mathbf{w}) &= (1)b + w_0x_0 + w_1x_1 + \ldots + w_Dx_D = \\ &= (1)b + \sum_{j=0}^{D} (w_jx_j) \end{aligned} \tag{1}$$

where $\mathbf{x} = \left[\begin{smallmatrix} x_0 & \cdots & x_D \end{smallmatrix}\right]^T$, a linear function of the parameters $\mathbf{w} = \left[\begin{smallmatrix} w_0 & \cdots & w_D \end{smallmatrix}\right]$ and bias b. We can also make the bias part of the $\mathbf{w}$ and at the same position in $\mathbf{x}$ put the value 1 and have the equation like (2)

$$y(\mathbf{x}, \mathbf{w}) = \mathbf{x} \cdot \mathbf{w} \tag{2}$$

a simple matrix multiplication. Both matrixes have the same amount of elements D.

There is also the notion of applying what is called as **basis function** to the $\mathbf{x}$ parameters before everything. So there will be a matrix of basis functions $\phi = \left[\begin{smallmatrix} \phi_0 & \cdots & \phi_D \end{smallmatrix}\right]^T$ where $b = x_i$ and $\phi_i(b) = 1$.

If that's the case the function will look like this (3).

$$\begin{aligned} y(\mathbf{x}, \mathbf{w}) = \sum_{j=0}^{D} (w_j \phi_j(x_j)) = \\ = \phi(\mathbf{x}) \cdot \mathbf{w} \end{aligned} \tag{3}$$

The functions can be seen as a way to pre-process and extract the important parts of the input This list can have nonlinear functions making $y(\mathbf{x}, \mathbf{w})$ nonlinear too.

In case we are working with a classification model, we might want to introduce an **activation function** $f(\cdot)$ that after all the computations in $y(\mathbf{x}, \mathbf{w})$ (3) transforms that result into a value corresponding to being the probability of $\mathbf{x}$ being of a certain class.

And this will be what I define as an artificial neuron (4). You can see a visual representation of it on page 5.

$$y(\mathbf{x}, \mathbf{w}) = f(\phi(\mathbf{x}) \cdot \mathbf{w}) \tag{4}$$

Because I couldn't find a simple way of doing anonymous functions in my language of choice, my implementation does not allow specifying basis functions nor activation functions so every basis function can be thought as being the identify function $\phi(x) = x$ and the activation function can be toggled between the identify and sigmoid (5).

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{5}$$

4 Feed-Forward Neural Network

The idea now, as explained in chapter 5.1 of [Bis06], is to train neurons so that their weight values are good.

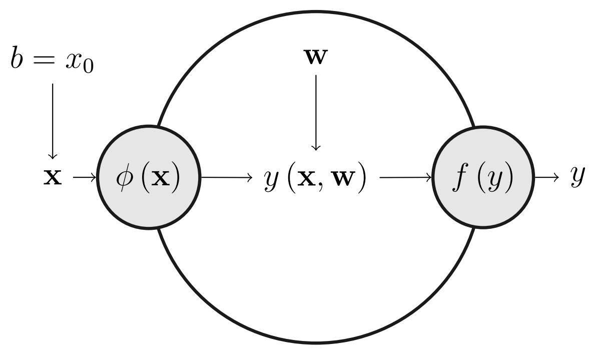

Figure 1: A visual representation of a neuron.

Because a neuron is really just a function, a neuron can be a basis function of another neuron, starting a network.

A network is composed of layers, each with its own number of neurons.

Each of neuron will create a value called **activation** represented by a_j (6) where j is the neuron number (we can think of a layer as a column matrix of neurons), $\mathbf{X}$ is the function input or the activations from the layer before after they passed through an activation function and D is the amount of row in $\mathbf{x}$ -1.

Let's define a way to represent a matrix row:

$$A = \begin{bmatrix} a_0 \\ \vdots \\ a_N \end{bmatrix} \in \mathbb{R}^{N \times M}, a_i \in \mathbb{R}^M, i \in \{x \in \mathbb{N} : x \leq N\}$$

$$A_{i*} = a_i$$

$$\begin{aligned} a_j &= f\left(\sum_{i=1}^{D} (w_{ji} x_i) + (1) b_j\right) \\ &= f\left(\mathbf{x} \cdot W_{j*}\right) \end{aligned} \tag{6}$$

The last layer will be the one giving the outputs $\mathbf{y}$.

If a neural network has n layers, the mathematical representation for the output k will be something like (9). $W^{(n)}$ are the weights of that layer. For this to work, because it's just matrix multiplications, we have to be aware of the dimensions of the matrixes. This is called **forward propagation**.

$$Y_0 = \mathbf{x} \tag{7}$$

$$Y_n = f_n \left(Y_{n-1}^T \cdot w^{(n)} \right), n > 0 \tag{8}$$

$$\begin{aligned} y_k(\mathbf{x}, \mathbf{w}) &= f_n \left(\sum_{i_n=0} w_{ki_n}^{(n)} \dots f_2 \left(\sum_{i_2=0} w_{i_3 i_2}^{(2)} f_1 \left(\sum_{i_1=0} w_{i_2 i_1}^{(1)} x_{i_1} \right) \right) \dots \right) = \\ &= f_n \left(\dots f_2 \left(f_1 \left(\mathbf{x} \cdot w^{(1)} \right)^T \cdot w^{(2)} \right)^T \dots \cdot w^{(n)} \right) = \\ &= Y_n \end{aligned} \tag{9}$$

Layers and neural network representation can be found `/src/utils/neural.zig`.

5 Learning

Now we can feed our network with some data, and it will give us a response (one that we might not want). The network will probably give you some nonsense output and that's probably because the weights in each neuron are also nonsense.

Our goal now it's to try and find the optimal weights for the network.

We will evaluate how bad the network is behaving and after that tell it what it should change to do a better job.

5.1 Error function

First we need a way of measuring how dumb the network is. Error functions measures the difference between the network output and the expected output.

There are a lot of functions that we can choose to calculate that.

We will use this $E(\mathbf{w})$ error function (11) that is a sum of the squares of those differences in outputs.

$$E_n(\mathbf{w}) = |\mathbf{y}(\mathbf{x}_n, \mathbf{w}) - \mathbf{t}_n|^2 \tag{10}$$

$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} E_n(\mathbf{w}) \tag{11}$$

We have the input training matrix $\{\mathbf{x}_n\}$, N is the input size. Also, we have $\{\mathbf{t}_n\}$ with the expected outputs for each input.

Our goal is to minimize the error function (11) because the closer to 0 that the error function return gets, means the differences between our neural network output and the expected output are less big.

5.2 Parameter Optimization

The way to minimize a function is by calculating its derivative on a point, so we can get the instantaneous rate of change at that point, that is, how much is the function growing/decreasing on that point. But that only works for unary function. Our neural network is a function with an unlimited (limited by the hardware which is running on) number of parameters (the weights). Because of that we instead use **partial derivatives**, and we calculate it for each one of the neural network weights (and biases if they are separated from the weights). Actually, we are calculation what's called the **gradient**, which is a vector of which one of those partial derivatives. That vector's direction is towards where that function grows (greatest rate of increase), we want to go the other way around.

Our goal is to reach a minimum with $\nabla E(\mathbf{w}) = \vec{0}$ and while we don't reach it, we keep changing the weights based on $-\nabla E(\mathbf{w})$. One of the problems that might arise is that $\nabla E(\mathbf{w}) = \vec{0}$ does not mean that we found the perfect weight values, because a function can have multiple minimums and with the amount of parameters that a neural network has, it surely does, but also, it does not mean for sure we meet a minimum and instead meet a maximum of saddle point. Also targeting non error can be impossible, so we might instead target a maximum error that we allow the neural network to have.

It also probably hard as hell to find the perfect weights only using math; I wouldn't get involved in that. At least it would take some time. So normally we just give each weight a random value, and we have the initial weight vector $\mathbf{w}^{(0)}$. From that, we iteratively update the weights based on a weight vector update $\Delta\mathbf{w}^{(\tau)}$ as shown in (12).

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \Delta\mathbf{w}^{(\tau)} \tag{12}$$

In our case we are going to do **gradient descent optimization** so from (12) we get (13).

$$\begin{aligned} \mathbf{w}^{(\tau+1)} &= \mathbf{w}^{(\tau)} + \Delta\mathbf{w}^{(\tau)} = \\ &= \mathbf{w}^{(\tau)} + \left(\left(\mathbf{w}^{(\tau)} + \eta\nabla E\left(\mathbf{w}^{(\tau)}\right)\right) - \mathbf{w}^{(\tau)}\right) = \\ &= \mathbf{w}^{(\tau)} + \eta\nabla E\left(\mathbf{w}^{(\tau)}\right) \end{aligned} \tag{13}$$

The $\eta > 0$ and it's called **learning rate**, so we can define how fast we want the optimization to be based on the gradient.

While implementing this, I had some problems.

Sometimes the $E(\mathbf{w})$ was minimizing and from out of nowhere it would explode into big numbers. But it seems there are some techniques that seem to maybe solve that problem such as **regularization** which is mentioned in page 30 [Bis06]. It can also be because of the way I calculated the derivatives: by using the limit definition. You can define an ϵ on your neural network, and it will calculate the derivatives using it like this: $\lim_{\epsilon\to\infty} \frac{f(x+\epsilon)-f(x)}{\epsilon}$. It would also depend on the input, and the initial weights.

During implementation, I also thought that this system can cause a lot of problems in multilayer neural networks, because changing the weights of one neuron will affect the behavior of every neuron forward because they depend on its activation, and we are changing every weight at the same time.

But I guess that's why **error backpropagation** is preferred over this method.

5.3 Error Backpropagation

Maybe because of what I mentioned above, error backpropagation exist, where the gradient is calculated from the last layer of the neural network (the output layer) to the first.

This should work for any feed-forward neural network with any activation functions and error function.

We have a general error function which is the sum of the errors of each input data.

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w}) \tag{14}$$

We want to compute $\nabla E_n(\mathbf{w})$.

So we start by feeding forward where we calculate all the activations for every input $\mathbf{x}_i$. In all the equations forward (τ) represents the layer.

Remember from the equation (1). Here we have

$$a_k^{(\tau)} = \sum_i \left(w_{ki}^{(\tau)} z_i^{(\tau-1)} \right) \tag{15}$$

$$z_i^{(\tau)} = f^{(\tau)}\left(a_i^{(\tau)}\right) \tag{16}$$

Now as we did before, we want to calculate the gradient, so we need to calculate the partial derivative $\frac{\partial E_n}{\partial w_{ji}^{(\tau)}}$ for every weight.

We can use the derivatives chain rule to write the partial derivative because the error function needs the activation after the weight to be dependent on that weight.

$$\frac{\partial E_n}{\partial w_{ji}^{(\tau)}} = \frac{\partial E_n}{\partial a_j^{(\tau)}} \frac{\partial a_j^{(\tau)}}{\partial w_{ji}^{(\tau)}} \tag{17}$$

And calculate those two separately.

$$\begin{aligned}
\frac{\partial E_n}{\partial a_j^{(\tau)}} &= \frac{\partial}{\partial a_j^{(\tau)}} \left(a_j^{(\tau)} - t_j\right)^2 = \\
&= 2\left(a_j^{(\tau)} - t_j\right) \frac{\partial}{\partial a_j^{(\tau)}} \left(a_j^{(\tau)} - t_j\right) = \\
&= 2\left(a_j^{(\tau)} - t_j\right)(1) = \\
&= 2\left(a_j^{(\tau)} - t_j\right) = \\
&= \delta_j
\end{aligned} \tag{18}$$

$$\begin{aligned}
\frac{\partial a_j^{(\tau)}}{\partial w_{ji}^{(\tau)}} &= \frac{\partial}{\partial w_{ji}^{(\tau)}} \sum_i \left(w_{ji}^{(\tau)} z_i^{(\tau-1)}\right) = \\
&= \frac{\partial}{\partial w_{ji}^{(\tau)}} \left(w_{ji}^{(\tau)} z_i^{(\tau-1)} + \sum_{k \neq i} \left(w_{jk}^{(\tau)} z_k^{(\tau-1)}\right)\right) = \\
&= (1)\, z_i^{(\tau-1)} + \sum_{k \neq i} 0 = \\
&= z_i^{(\tau-1)}
\end{aligned} \tag{19}$$

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j^{(\tau)} z_i^{(\tau-1)} \tag{20}$$

We call $\delta_j^{(\tau)}$ the errors. Because we already calculated every $z_i^{(\tau-1)}$ while forward propagating, we really just need to calculate the errors.

The $\delta_j^{(\tau)}$ we calculated is for the output layer. For the layer before that, we can also use the chain rule.

$$\begin{aligned}
\delta_j^{(\tau-1)} &= \frac{\partial E_n}{\partial a_j^{(\tau-1)}} \\
&= \sum_k \left(\frac{\partial E_n}{\partial a_k^{(\tau)}} \frac{\partial a_k^{(\tau)}}{\partial a_j^{(\tau-1)}} \right) = \\
&= \sum_k \left(\delta_k^{(\tau)} \frac{\partial}{\partial a_j^{(\tau-1)}} \left(w_{kj}^{(\tau)} f^{(\tau-1)} \left(a_j^{(\tau-1)} \right) + \sum_{i \neq j} \left(w_{ki}^{(\tau)} z_{ki}^{(\tau-1)} \right) \right) \right) = \\
&= \sum_k \left(\delta_k^{(\tau)} w_{ki}^{(\tau)} f'^{(\tau-1)} \left(a_i^{(\tau-1)} \right) \right) = \\
&= f'^{(\tau-1)} \left(a_j^{(\tau-1)} \right) \sum_k \left(w_{kj}^{(\tau)} \delta_k^{(\tau)} \right)
\end{aligned} \tag{21}$$

$$\begin{aligned}
\frac{\partial E_n}{\partial w_{ji}^{(\tau)}} &= \frac{\partial E_n}{\partial a_j^{(\tau)}} \frac{\partial a_j^{(\tau)}}{\partial w_{ji}^{(\tau)}} = \\
&= \frac{\partial E_n}{\partial a_{j^{(\tau+l)}}^{(\tau+l)}} \prod_{L=1}^{l} \left(\frac{\partial a_{j^{(\tau+l-(L-1))}}^{(\tau+l-(L-1))}}{\partial a_{j^{(\tau+l-L)}}^{(\tau+l-L)}} \right) \frac{\partial a_{j^{(\tau)}}^{(\tau)}}{\partial w_{ji}^{(\tau-l)}}
\end{aligned} \tag{22}$$

After calculating all errors from the output layer, you are able to calculate from the other layers, one at a time, backwards, and you'll end up with the derivatives for the gradient.

References

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning.* Springer, 2006. ISBN: 9780387310732.