The Maths behind Neural Networks

Alex Punnen

Contents

Chapter 8

Back Propagation in Full - With Softmax & CrossEntropy Loss

The previous chapters should have given an intuition of Back Propagation. Let's now dive much deeper into Back Propogation. We will need all the information covered in the previous chapters, plus a bit more involved mathematical concepts.

It is good to remember here Geoffrey Hinton's talk available in Youtube - All this was invented not out of some mathematical model; but based on trial and error and checking what works. So do not treat this as distilled science. This is ever evolving.

Let's think of a l layered neural network whose input is $x = a^0$ and output is a^l.In this network we will be using the **sigmoid (σ)** function as the activation function for all layers except the last layer l. For the last layer we use the **Softmax activation function**. We will use the **Cross Entropy Loss** as the loss function.

Below are some of the concepts that we had already covered in brief in previous chapters; and some which we have not touched previously; but without which it will not be possible to build a practical deep neural network solution.

- Understand what **Scalar's, Vectors, Tensors** are and that Vectors and Tensors are written as matrices and Vector is one dimension matrix whereas Tensor's are many dimensional usually. (Technically a Vector is also a Tensor). After this, you can forget about Tensors and think only of Vectors Matrices and Scalar. Mostly just matrices.
- That **linear algebra for matrices** that will be used is just properties of matrix multiplication and addition that you already know. A linear equation of the form

  $$y = m * x + c$$

  in matrix form used in a neural network is $z_l = w_l * a_{l-1} + b_l$.
- The **Index notation for dealing with Vectors and Matrices** - A Primer on Index Notation John Crimaldi

- **Matrix multiplication** plays a major part and there are some parts that may be confusing
  - Example- **Dot Product** is defined only between Vectors, though many articles and tutorials will be using the dot product. Since each row of a multidimensional matrix acts like a Vector, the Numpy dot function(numpy.dot) works for matrix multiplication for non-vectors as well. Technically **numpy matmul** is the right one to use for matrix multiplication. $np.dot(A, B)$ is same as $np.matmul(A, B)$.numpy.**Numpy einsum** is also used for dimensions more than two. If A and B are two dimensional matrices $np.dot(A, B) = np.einsum('ij, jk- > ik', A, B)$. And einsum is much easier than numpy.tensordot to work with. For Hadamard product **numpy.multiply**

    There is no accepted definition of matrix multiplication of dimensions higher than two!
  - **Hadamard product**. It is a special case of the element-wise multiplication of matrices of the same dimension. It is used in the magic of converting index notation to Matrix notation. You can survive without it, but you cannot convert to Matrix notation without understanding how. It is referred to in Michel Neilsen's famous book Neural Networks and Deep Learning Michel Neilsen in writing out the Error of a layer with respect to previous layers.
- Calculus, the concept of Derivatives, **Partial Derivatives, Gradient, Matrix Calculus, Jacobian Matrix**
  - That derivative of a function -the *derivative function* $f'(x)$, gives the slope or gradient of the function at any 'point'. As it is the rate of change of one variable with respect to to another. Visually, say for a function in 2D space , say a function representing a line segment, that means change in Y for a change in X - rise over run,slope.
  - For multi variable function, example a Vector function, we need the rate of change of many variables with respect to to another, we do so via `Partial derivatives` concept - notation ∂ ; and the gradient becomes a Vector of partial derivatives. To visualize this, picture a hill, or a function of x,y,z variables that can be plotted in a 3D space, a ball dropped on this hill or graph goes down this `gradient vector` .To get the *derivative function* $f'(x, y, z)$ to calculate this gradient you need `multivariable calculus`, again something that you can ignore most of the time,except the slightly different rules while calculating the derivative function.
  - Take this a notch further and we reach the Jacobian matrix. For a Vector of/containing multivariable functions, the partial derivatives with respect to to say a Matrix or Vector of another function, gives

a *Matrix of Partial Derivatives* called the `Jacobian Matrix`. And this is also a gradient matrix. It shows the 'slope' of the *derivative function* at a matrix of points. In our case the derivative of the Loss function (which is a scalar function) with respect to Weights (matrix), can be calculated only via intermediate terms, that include the derivative of the Softmax output (Vector) with respect to inputs (matrix) which is the Jacobian matrices. And that is matrix calculus. Again something that you can now ignore henceforth.

- Knowing what a Jacobian is, and how it is calculated, you can blindly ignore it henceforth. The reason is that, most of the terms of the Jacobian evaluate to Zero for Deep learning application, and usually only the diagonal elements hold up, something which can be represented by index notation. *"So it's entirely possible to compute the derivative of the softmax layer without actual Jacobian matrix multiplication . . . the Jacobian of the fully-connected layer is sparse.- The Softmax function and its derivative-Eli Bendersky"*
  - Note -When you convert from Index notation to actual matrix notation, for example for implementation then you will need to understand how the index multiplication transforms to Matrix multiplication - transpose. Example from The Matrix Calculus You Need For Deep Learning (Derivative with respect to Bias) Terence,Jermy

$$\frac{\partial z^2}{\partial w^2} = (1^{\rightarrow})^T * diag(a^1) = (a^1)^T$$

- Calculus - **Chain Rule - Single variable, Multi variable Chain rule, Vector Chain Rule**
  - Chain rule is used heavily to break down the partial derivate of Loss function with respect to weight into a chain of easily differentiable intermediate terms
  - The Chain rule that is used is actually Vector Chain Rule , but due to nature of Jacobian matrices generated- sparse matrices, this reduces to resemble Chain rule of single variable or Multi-variable Chain Rule. Again the definite article to follow is The Matrix Calculus You Need For Deep Learning (Derivative with respect to Bias) Terence,Jermy, as some authors refer as Multi variable Chain rule in their articles

    Single Variable Chain Rule

$$y = f(g(x)) = f(u) \text{ where } u = g(x)$$

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}$$

Vector Chain Rule

In the notation below, **y** is a Vector output and x is a scalar. Vectors are represented in bold letters though I have skipped it here.

$$y = f(g(x))$$

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial g} * \frac{\partial g}{\partial x}$$

Here $\frac{\partial y}{\partial g}$ and $\frac{\partial g}{\partial x}$ are two Jacobian matrices containing the set of partial derivatives. But since only the diagonals remain in deep learning application we can skip calculating the Jacobian and write in index notation as

$$\frac{\partial y}{\partial x} = \frac{\partial y_i}{\partial g_i} \frac{\partial g_i}{\partial x_i}$$

The Neural Network Model

I am writing this out, without index notation, and with the super script representing just the layers of the network.

$$\boxed{[}\textbf{10px}, \textbf{border} : \textbf{2pxsolidred}]^{a^0} \rightarrow \boxed{[}5px, border : 2pxsolidblack]\underbrace{\text{hidden layers}}_{a^{l-2}} \rightarrow \boxed{[}5px, border : 2pxsolidbl$$

Y is the target vector or the Truth vector. This is a one hot encoded vector, example $Y = [0, 1, 0]$, here the second element is the desired class.The training is done so that the CrossEntropyLoss is minimized using Gradient Loss algorithm.

P is the Softmax output and is the activation of the last layer a^l. This is a vector. All elements of the Softmax output add to 1; hence this is a probability distribution unlike a Sigmoid output.The Cross Entropy Loss L is a Scalar.

Note the Index notation is the representation an element of a Vector or a Tensor, and is easier to deal with while deriving out the equations.

Softmax (in Index notation)

Below I am skipping the superscript part, which I used to represent the layers of the network.

$$p_j = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

This represent one element of the softmax vector, example $\vec{P} = [p_1, p_2, p_3]$

Cross Entropy Loss (in Index notation)

Here y_i is the indexed notation of an element in the target vector Y.

$$L = -\sum_j y_j \log p_j$$

Hence $\frac{\partial a_{ij}}{\partial X}$ can be written as $\text{diag}(f'(X))$; $(A = f(X))$

Note that Multiplication of a vector by a diagonal matrix is element-wise multiplication or the Hadamard product; *And matrices in Deep Learning implementation can be seen as stacked vectors for simplification.*

More details about this here Jacobian Matrix for Element wise Opeation on a Matrix (not Vector)

References

Easier to follow (without explicit Matrix Calculus) though not really correct - Supervised Deep Learning Marc'Aurelio Ranzato DeepMind
Easy to follow but lacking in some aspects - Notes on Backpropagation-Peter Sadowski Slightly hard to follow using the Jacobian - The Softmax function and its derivative-Eli Bendersky More difficult to follow with proper index notations (I could not) and probably correct - Backpropagation In Convolutional Neural Networks Jefkine

---

End