Math for Understanding Deep Neural Networks

Machine Learning for IOT

Nikhil Challa

December 15, 2022

1 Backpropagation Derivation

To cover the derivation in stages, I provide the equation assuming batch update, but simplify to SGD (stochastic gradient descent) with batch size of 1 for ease of math derivation

Categorical Cross Entropy loss function for multi-class classification case

$$L = -\frac{1}{S}\sum_{i=1}^{S}\frac{1}{M}\sum_{j=1}^{M}\hat{y}_{j,k}\log y_{j,k} \tag{1}$$

Ignoring the constants, and focusing on SGD, we can rewrite loss function as,

$$L = -\sum_{j=1}^{M}\hat{y}_{j}\log y_{j} \tag{2}$$

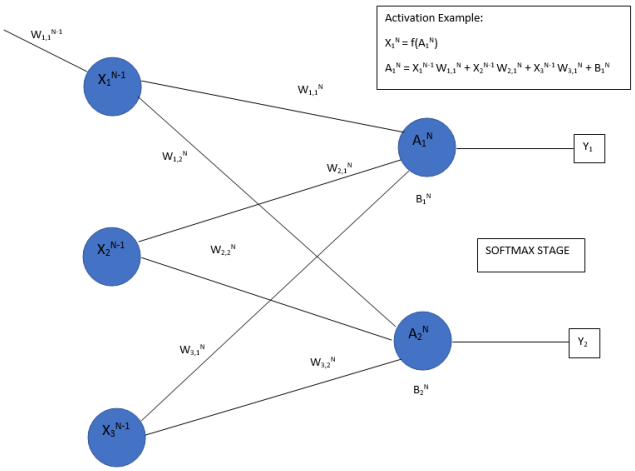

Figure 1: Simplified Neural Network for Derivation

We will use the above figure as reference but first we will define the convention used (some terms are in lower case below).

S : Number of samples

M : output dimension

$\hat{y}$: Target output

y : output from Network (sometimes represented as p (probability))

n : Superscript represents layer index with range (1 , N). Applicable for w, b, a, x

i, j, k : We will use triple subscript representation for easy of understanding the math. If the index has not significance for the term, it will be represented by a dot.
First index (i) : Represents the node index from previous layer. Applicable only for w
Second index (j) : Represents the node index in the current layer. Applicable for $w, b, a, x, y, \hat{y}$
Third index (k) : Represents the value for each sample in data set. Applicable for $dA, y, \hat{y}, x, a$

$w^n_{i,j}$: Weight that is connected between node i at layer $n-1$ to node j at layer n.

b^n_j : Bias for node j at layer n.

$a^n_{j,k}$: Accumulation of weights and bias or the argument to activation function

$f(\cdot)$: Activation function (RELU in our case for below derivation)

$x^n_{j,k}$: output of activation function or sample input if $n = 1$ (layer 1)

η : Learning rate

Derivation for gradient (Cross Entropy Loss function + Softmax layer + ReLU activation function)

Note : Last layer doesnt have activation function
Lets first understand the softmax equation and its derivative

$$y_{j,k} = \frac{e^{a_{j,k}}}{\sum_{j=1}^{M} e^{a_{j,k}}} \tag{3}$$

Simplified Expression :

$$y_j = \frac{e^{a_j}}{\sum_{j=1}^{M} e^{a_j}} \tag{4}$$

$$\frac{\partial y_{j_1}}{\partial a_{j_2}} = y_{j_2} \cdot (\delta_{j_2,j_1} - y_{j_1}) \tag{5}$$

$$= \begin{cases} y_{j_2} \cdot (1 - y_{j_1}) & j_1 = j_2 \quad \text{or } \delta_{j_2,j_1} = 1 \\ y_{j_2} \cdot (-y_{j_1}) & j_1 \neq j_2 \quad \text{or } \delta_{j_2,j_1} = 0 \end{cases} \tag{6}$$

Accumulation function or argument to activation function

$$a^n_{j,k} = \sum_{j'=1}^{M'} w^n_{i,j} \cdot x^{n-1}_{j',k} + b^n_j \tag{7}$$

...where M' num nodes in layer n-1

Simplified Expression :

$$a^n_j = \sum_{j'=1}^{M'} w^n_{j',j} \cdot x^{n-1}_{j'} + b^n_j \tag{8}$$

Derivative w.r.t weights and bias :

$$\frac{\partial a^n_j}{\partial w^n_{j',j}} = x^{n-1}_{j'} \tag{9}$$

$$\frac{\partial a^n_j}{\partial b^n_j} = 1 \tag{10}$$

Lets now identify the gradient for weights starting with outer most weights.

$\frac{\partial L}{\partial w^N_{1,1}} = \frac{\partial L}{\partial y} \frac{\partial y}{\partial a^N_1} \frac{\partial a^N_1}{\partial w^N_{1,1}}$ ...applying chain rule and tracing from output to the desired weight, taking $w_1, 1$ as an example

The first two subterms are a bit tricky, remember that the loss function is summation across all output values, but the derivative w.r.t x_i depends on corresponding j subscript value for $y_{i,j}$. This is why I skipped

the subscript for y in the chain rule derivation above. Re-introducing it below...

$$\frac{\partial L}{\partial y} = -\sum_{j=1}^{M} \frac{\hat{y}_j}{y_j} \tag{11}$$

Combining the two above partial derivative to get full expression interms of its subscripts

$$\frac{\partial L}{\partial y}\frac{\partial y}{\partial a_1^N} = -\sum_{j=1}^{M} \frac{\hat{y}_j}{y_j} \cdot \Big(y_1 \cdot (\delta_{1,j} - y_j)\Big) \tag{12}$$

$$= -\frac{\hat{y}_1}{\cancel{y_1}} \cdot \Big(\cancel{y_1} \cdot (1 - y_1)\Big) - \sum_{j \neq 1} \frac{\hat{y}_j}{\cancel{y_j}} \cdot \Big(y_1 \cdot (-\cancel{y_j})\Big) \tag{13}$$

$$= -\hat{y}_1 + \hat{y}_1 y_1 + \sum_{j \neq 1} \hat{y}_j y_1 \tag{14}$$

$$= y_1 \sum_{j=1}^{M} \hat{y}_j - \hat{y}_1 \tag{15}$$

$$= y_1 - \hat{y}_1 \tag{16}$$

Notice how the equation got simplified. We will store this so it can be used later for back propagation and is common term for weights and bias as shown below. Generalizing the expression for any weights and bais for last layer

$$\frac{\partial L}{\partial a_j^N} = y_j - \hat{y}_j = dA_j^N \tag{17}$$

Gradient for weight :

$$\frac{\partial L}{\partial w_{i,j}^N} = \frac{\partial L}{\partial a_j^N}\frac{\partial a_j^N}{\partial w_{i,j}^N} = dA_j^N \cdot x_i^{N-1} \tag{18}$$

Similarly gradient for bias :

$$\frac{\partial L}{\partial b_j^N} = \frac{\partial L}{\partial a_j^N}\frac{\partial a_j^N}{\partial b_j^N} = dA_j^N \tag{19}$$

For all layers except the last layer, we are using RELU activation function

$$x_j^n = \max(0, a_j^n) \tag{20}$$

$$\frac{\partial x_j^n}{\partial a_j^n} = \begin{cases} 1 & \text{if } a_j > 0 \\ 0 & \text{if } a_j \leq 0 \end{cases} \tag{21}$$

Now we shall try for weights located deeper. Note that its not straight forward replica from above as now we need to account for contributions from other paths like via $w_{1,2}^N$ also. All of that can be clubbed within first subterm below

$$\frac{\partial L}{\partial w_{1,1}^{N-1}} = \underbrace{\frac{\partial L}{\partial x_1^{N-1}}}_{\text{clubbed}} \frac{\partial x_1^{N-1}}{\partial a_1^{N-1}} \frac{\partial a_1^{N-1}}{\partial w_{1,1}^{N-1}}$$

Digging deeper into clubbed term...

$$\frac{\partial L}{\partial x_1^{N-1}} = \frac{\partial L}{\partial a_1^N} \frac{\partial a_1^N}{\partial x_1^{N-1}} + \frac{\partial L}{\partial a_2^N} \frac{\partial a_2^N}{\partial x_1^{N-1}} = dA_1^N \cdot w_{1,1}^N + dA_2^N \cdot w_{1,2}^N$$

Notice that we dont need to recompute the previous steps as we good deeper, provided we store the terms!

Gradient for weight :

$$\frac{\partial L}{\partial w_{1,1}^{N-1}} = (dA_1^N \cdot w_{1,1}^N + dA_2^N \cdot w_{1,2}^N) \cdot (0 \text{ or } 1) \cdot x_1^{N-1} = dA_1^{N-1} \cdot x_1^{N-1} \tag{22}$$

Similarly gradient for bias :

$$\frac{\partial L}{\partial b_1^N} = \frac{\partial L}{\partial a_1^{N-1}} \frac{\partial a_1^{N-1}}{\partial b_1^{N-1}} = dA_1^{N-1} \tag{23}$$

Generalizing the algorithm for all layers except the last layer

Gradient for weight :

$$\frac{\partial L}{\partial w_{i,j}^n} = dA_j^n \cdot x_i^n = \underbrace{\left(\sum_{i'=1}^{\text{num nodes in layer n+1}} dA_{i'}^{n+1} \cdot w_{j,i'}^{n+1} \right) \cdot (0 \text{ or } 1)}_{dA_j^n} \cdot x_i^{n-1} \tag{24}$$

Similarly gradient for bias :

$$\frac{\partial L}{\partial b_j^n} = dA_j^n \tag{25}$$

Neat trick : We can completly skip computing the summation term or dA_j^n if the value of $a_j^n \leq 0$

Finally we need to define the weight (and bias) change.
$\Delta w = -\eta \frac{\partial L}{\partial w}, \quad \Delta b = -\eta \frac{\partial L}{\partial b}$

What do we need to store in every node?

b : bias for forward propagation. Variable name : B

w : incomming weights for forward propagation and backward propagation. Variable name : W

x : output of activation function for both forward and backward propagation. Variable name : X

dA : partial differentail w.r.t aggregation (or argument to activation function) for back propagation

Δw : we cannot apply the weight change until the gradiant is already calculated for all weights and bias in the network. Variable name : dW

Δb : Same reason as above. Variable name : dB

Note that to calculate the gradient, we need to use original weights and bias. Once gradient calculation is complete, we then apply the difference for all the weights and bias before performing forward propagation

2 Forward Propagation Derivation and other details

The forward propagation is just applying accumulation function, followed by non-linear activation function to update node value. This is applicable for all layers except the last layer, we we skip non-linear activation function.

Fix for overflow issue:
Because of the nature of the problem, where we use un-normalized node values from Model as vector input for implementing Distributed Deep learning, it may happen that the input to softmax may exceed float or double type limits while performing exponential operation. The input limit is about 709 for double type and 88 for float type, so we perform scaling to ensure the max absolute value doesnt excced the limits.

Source code location:
For source code please refer to NN_functions.h