Math for Understanding Deep Neural Networks

Machine Learning for IOT

Nikhil Challa

January 5, 2023

1 Backpropagation Derivation

To cover the derivation in stages, I provide the equation assuming gradient descent, but simplify to SGD (stochastic gradient descent) with batch size of 1 for ease of math derivation. Will add more details for gradient descent later, when time permits. Lets start with notations first! Because of the nature of indexing in C language which is from 0 to $N-1$ instead of 1 to N for an array of size N, we will follow the same convention below

S : Number of samples in a batch

M : output dimension size

N : number of layers

J^n : number of nodes for layer n - Change the notations in summation sign to replace text!? eg . eq 30

$\hat{y}$: Target output

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

n : Superscript represents layer index with range (0 , N-1). 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 or be skipped.
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 $dLa, y, \hat{y}, x, a$

$w^n_{i,j}$: 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

$g(\cdot)$: 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

We will use the figure below as reference (some terms are in lower case above). For the math derivation, I have used lower case for terms that are not constants. For programming, I have used upper case for same variables, and used macros for constants

Also, note that the index meaning listed above (i, j, k) is applicable to terms on left hand side (LHS) of the equation. The right hand side (RHS) terms need to follow the value from LHS and so will take alphabet from LHS. As an example if you notice i instead of j for x (which is not possible as per notation rules), it simply means, we need to take index value from w located on LHS.

Categorical Cross Entropy loss function for multi-class classification case

$$L = -\frac{1}{S}\sum_{k=0}^{S-1}\sum_{j=0}^{M-1} \hat{y}_{j,k} \log y_{j,k} \tag{1}$$

Focusing on SGD, we can rewrite loss function as,

$$L = -\sum_{j=0}^{M-1} \hat{y}_j \log y_j \tag{2}$$

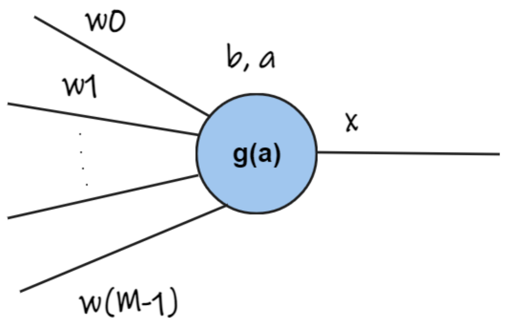

Figure 1: Single Neuron Structure

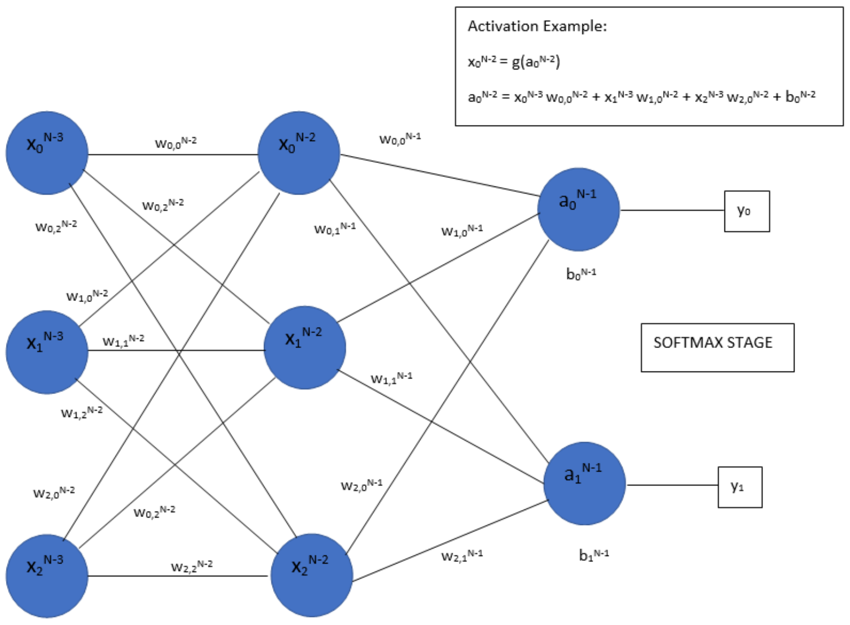

Figure 2: Simplified Neural Network for Derivation

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

$$y_{j,k} = \frac{e^{a_{j,k}}}{\sum\limits_{j=0}^{M-1} e^{a_{j,k}}} \tag{3}$$

Simplified Expression :

$$y_j = \frac{e^{a_j}}{\sum\limits_{j'=0}^{M-1} e^{a_{j'}}} \tag{4}$$

You have to be very careful here when taking derivative of y w.r.t a as it depends on the index of these terms (index represent what is output it corresponds to). Lets cover both cases..

Case1 : y and a have the same index

$$\begin{aligned}
\frac{\partial y_{j_1}}{\partial a_{j_1}} &= \frac{\partial e^{a_{j_1}}}{\partial a_{j_1}} \frac{1}{\sum\limits_{j'=0}^{M-1} e^{a_{j'}}} + e^{a_{j_1}} \frac{\partial \left(\sum\limits_{j'=0}^{M-1} e^{a_{j'}}\right)^{-1}}{\partial a_{j_1}} \\
&= \frac{e^{a_{j_1}}}{\sum\limits_{j'=0}^{M-1} e^{a_{j'}}} - e^{a_{j_1}} \cdot \left(\sum_{j'=0}^{M-1} e^{a_{j'}}\right)^{-2} \cdot \left(\frac{\partial e^{a_{j_1}}}{\partial a_{j_1}} + \sum_{j' \neq j_1} \frac{\partial e^{a_{j'}}}{\partial a_{j_1}}\right) \\
&= y_{j_1} - e^{a_{j_1}} \cdot \frac{1}{\left(\sum\limits_{j'=0}^{M-1} e^{a_{j'}}\right)^2} \cdot (e^{a_{j_1}} + 0's) \\
&= y_{j_1} - y_{j_1}^2 \\
&= y_{j_1}(1 - y_{j_1})
\end{aligned}$$

Case2 : y and a have different index

$$\begin{aligned}
\frac{\partial y_{j_1}}{\partial a_{j_2}} &= \frac{\partial e^{a_{j_1}}}{\partial a_{j_2}} \frac{1}{\sum\limits_{j'=0}^{M-1} e^{a_{j'}}} + e^{a_{j_1}} \frac{\partial \left(\sum\limits_{j'=0}^{M-1} e^{a_{j'}}\right)^{-1}}{\partial a_{j_2}} \\
&= 0 - e^{a_{j_1}} \cdot \left(\sum_{j'=0}^{M-1} e^{a_{j'}}\right)^{-2} \cdot \left(\frac{\partial e^{a_{j_2}}}{\partial a_{j_2}} + \sum_{j' \neq j_2} \frac{\partial e^{a_{j'}}}{\partial a_{j_1}}\right) \\
&= -e^{a_{j_1}} \cdot \frac{1}{\left(\sum\limits_{j'=0}^{M-1} e^{a_{j'}}\right)^2} \cdot (e^{a_{j_2}} + 0's) \\
&= -\frac{e^{a_{j_1}}}{\sum\limits_{j'=0}^{M-1} e^{a_{j'}}} \cdot \frac{e^{a_{j_2}}}{\sum\limits_{j'=0}^{M-1} e^{a_{j'}}} \\
&= -y_{j_1} y_{j_2}
\end{aligned}$$

We can neatly combine the two cases into a single expression below.

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

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

Accumulation function or argument to activation function

$$a^n_{j,k} = \sum_{j'=0}^{M'-1} w^n_{i,j} \cdot x^{n-1}_{j',k} + b^n_j \tag{7}$$

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

Simplified Expression :

$$a^n_j = \sum_{j'=0}^{M'-1} w^n_{j',j} \cdot x^{n-1}_{j'} + b^n_j \tag{8}$$

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 nodes. Remember we do not have a non-linear activation function for last layer.

$$\frac{\partial L}{\partial w_{1,1}^{N-1}} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial a_1^{N-1}}\frac{\partial a_1^{N-1}}{\partial w_{1,1}^{N-1}} \tag{11}$$

...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...

$$\frac{\partial L}{\partial y} = -\sum_{j=0}^{M-1}\frac{\hat{y}_j}{y_j} \tag{12}$$

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

$$\begin{aligned}
\frac{\partial L}{\partial y}\frac{\partial y}{\partial a_1^{N-1}} &= -\sum_{j=1}^{M}\frac{\hat{y}_j}{y_j}\cdot\Big(y_1\cdot(\delta_{1,j}-y_j)\Big) && (13)\\
&= -\frac{\hat{y}_1}{\cancel{y_1}}\cdot\Big(\cancel{y_1}\cdot(1-y_1)\Big) - \sum_{j\neq 1}\frac{\hat{y}_j}{\cancel{y_j}}\cdot\Big(y_1\cdot(-\cancel{y_j})\Big) && (14)\\
&= -\hat{y}_1 + \hat{y}_1 y_1 + \sum_{j\neq 1}\hat{y}_j y_1 && (15)\\
&= y_1\cdot\left(\sum_{j=0}^{M-1}\hat{y}_j\right) - \hat{y}_1 && (16)\\
&= y_1 - \hat{y}_1 && (17)
\end{aligned}$$

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

$$\frac{\partial L}{\partial a_j^{N-1}} = y_j - \hat{y}_j = dLa_j^{N-1} \tag{18}$$

Gradient for weight :

$$\frac{\partial L}{\partial w_{i,j}^{N-1}} = \frac{\partial L}{\partial a_j^{N-1}}\cdot\frac{\partial a_j^{N-1}}{\partial w_{i,j}^{N-1}} = dLa_j^{N-1}\cdot x_i^{N-2} \tag{19}$$

Similarly gradient for bias :

$$\frac{\partial L}{\partial b_j^{N-1}} = \frac{\partial L}{\partial a_j^{N-1}}\cdot\frac{\partial a_j^{N-1}}{\partial b_j^{N-1}} = dLa_j^{N-1} \tag{20}$$

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

$$x_j^n = \max(0, a_j^n) \tag{21}$$

$$\frac{\partial x_j^n}{\partial a_j^n} = \frac{\partial}{\partial a_j^n}\max(0, a_j^n) = \max(0, \frac{\partial}{\partial a_j^n} a_j^n) = \begin{cases} 1 & \text{if } a_j > 0 \\ 0 & \text{if } a_j \leq 0 \end{cases} \tag{22}$$

To pack the above expression in complete equation, we will represent it as $(0 \text{ or } 1)_j$

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

$$\frac{\partial L}{\partial w_{1,1}^{N-2}} = \underbrace{\frac{\partial L}{\partial x_1^{N-2}}}_{\text{clubbed}} \frac{\partial x_1^{N-2}}{\partial a_1^{N-2}} \frac{\partial a_1^{N-2}}{\partial w_{1,1}^{N-2}}$$

Digging deeper into clubbed term and tracing it back to previous layer...

$$\frac{\partial L}{\partial x_1^{N-2}} = \frac{\partial L}{\partial a_1^{N-1}}\frac{\partial a_1^{N-1}}{\partial x_1^{N-2}} + \frac{\partial L}{\partial a_2^{N-1}}\frac{\partial a_2^{N-1}}{\partial x_1^{N-2}} = dLa_1^{N-1} \cdot w_{1,1}^{N-1} + dLa_2^{N-1} \cdot w_{1,2}^{N-1} \tag{23}$$

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

Gradient for weight :

$$\frac{\partial L}{\partial w_{1,1}^{N-2}} = \underline{(dLa_1^{N-1} \cdot w_{1,1}^{N-1} + dLa_2^{N-1} \cdot w_{1,2}^{N-1}) \cdot (0 \text{ or } 1)_1} \cdot x_1^{N-2} = dLa_1^{N-2} \cdot x_1^{N-3} \tag{24}$$

Similarly gradient for bias :

$$\frac{\partial L}{\partial b_1^{N-2}} = \frac{\partial L}{\partial a_1^{N-2}}\frac{\partial a_1^{N-2}}{\partial b_1^{N-2}} = dLa_1^{N-2} \tag{25}$$

Generalizing the algorithm for all layers except the last layer

Gradient for weight :

$$\frac{\partial L}{\partial w_{i,j}^n} = dLa_j^n \cdot x_i^{n-1} = \underbrace{\left(\sum_{i'=0}^{\text{num nodes in layer n+1 minus 1}} dLa_{i'}^{n+1} \cdot w_{j,i'}^{n+1}\right) \cdot (0 \text{ or } 1)_j}_{dLa_j^n} \cdot x_i^{n-1} \tag{26}$$

Similarly gradient for bias :

$$\frac{\partial L}{\partial b_j^n} = dLa_j^n \tag{27}$$

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

Finally we need to define the weight (and bias) change.

$$\Delta w = -\eta \frac{\partial L}{\partial w}, \quad \Delta b = -\eta \frac{\partial L}{\partial b} \tag{28}$$

Modifications for Batch size > 1

Lets assume we take all samples in dataset and call it Gradient Descent method, or GD. The deviation from SGD shown above is listed below

1. We perform single update to the weight and bias using all the samples.
2. The values for dLa and x used above will be for a single sample in batch. The w terms will be fixed until the gradient contribution from all samples are calculated.

This will change the implementation slightly. Because we are not supposed to update the weights and bias until all sample contributions are factored, we need to store a copy of network in memory; perform the gradient for first sample; and restore network for next sample. Once we have the contributions from all samples

Lets look at the equations again, specifically equation 18 and equation 24

$$\frac{\partial L}{\partial w_{i,j}^N} = \frac{1}{S} \underbrace{\sum_{k=0}^{S-1} (y_{j,k} - \hat{y}_{j,k})}_{dLa_{j,k}^N} \cdot x_{i,k}^{N-1} \tag{29}$$

$$\frac{\partial L}{\partial w_{i,j}^n} = dLa_j^n \cdot x_i^n = \frac{1}{S} \sum_{k=0}^{S-1} \underbrace{\left(\sum_{i'=0}^{\text{num nodes in layer n+1 minus 1}} dLa_{i',k}^{n+1} \cdot w_{j,i'}^{n+1} \right) \cdot (0 \text{ or } 1)_{j,k}}_{dLa_{j,k}^n} \cdot x_{i,k}^{n-1} \tag{30}$$

Notice that we need to store each $dLa_{j,k}$ in memory to be used by subsequent layers, as we do back propagation. Now, we also need to store $x_{j,k}$ or we keep running forward propagation to utilize the correct $x_{j,k}$ but perform one layer backward to keep updating $dLa_{j,k}$ correctly! We can now see a trade-off between memory and time for case of $x_{j,k}$, but we need more memory for case of $dLa_{j,k}$

I have listed the two methods below, pay careful attention to indexes used to signify the differences in memory allocation approach.

Case1 : Time saver but more memory

1. Take sample1, perform forward propagation and store $x_{j,k}$ for that sample.
2. Repeat step1 for all samples and store all values of $x_{j,k}$
3. Now we perform back propagation and store each value of $dLa_{j,k}$

Case2 : Memory save but more time

1. Take sample1, perform forward propagation and store x_j for that sample.
2. Then we perform back propagation but only 1 layer and store the value of $dLa^{N}_{j,k}$ for that sample.
3. Now we take sample2, perform forward propagation and store x_j for that sample.
4. Then we perform back propagation but only 1 layer and store the value of $dLa_{j,k}$ for that sample.
5. We basically repeat 1 and 2 for all samples but we have made progress only 1 layer in backward propagation.
6. Now we again go back to sample1, perform forward propagation and store x_j for that sample.
7. We then perform back propagation, but we can go two layers deep and store value of $dLa^{N-1}_{j,k}$ for that sample.
8. Similar to step1 and 2, we repeat but we can go two layers deep.
9. In this fashion, we keep going deeper and deeper until we hit first layer.

We can have third case were we dont need to store dLa for each sample but you can only imagine the complexity in algorithm. On comparison, Case1 is lot easier to understand and implement but at the cost of increase memory and is proportional to number of samples in a batch. We will adopt Case1 in the code (future work)

What do we need to store in every node in Neural Network Layers?

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

dLa : partial differential w.r.t aggregation (or argument to activation function) for back propagation. Variable name : dLa

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

d : To drop or not to drop, valid only if dropouts is used for networks. Variable name : D

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 Understanding CNN backpropagation

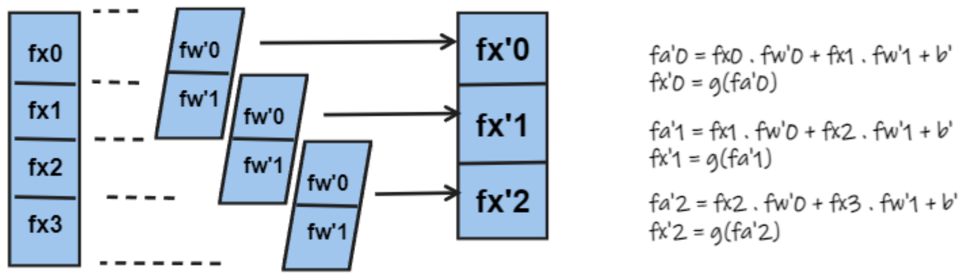
fx0
fx1
fx2
fx3
fw'0
fw'1
fx'0
fx'1
fx'2
fa'0 = fx0 . fw'0 + fx1 . fw'1 + b'
fx'0 = g(fa'0)
fa'1 = fx1 . fw'0 + fx2 . fw'1 + b'
fx'1 = g(fa'1)
fa'2 = fx2 . fw'0 + fx3 . fw'1 + b'
fx'2 = g(fa'2)

Figure 3: Simplified 1D Convolution Neural Network for derivation

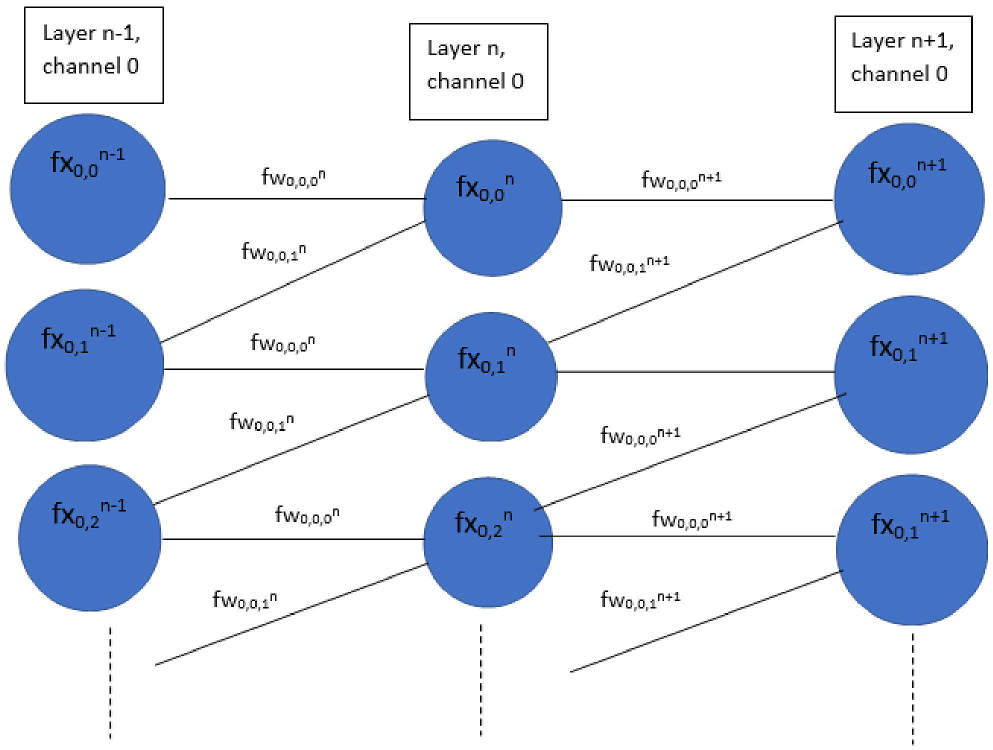

Figure 4: 1D Convolution Neural Network for derivation

The Figure 3 above describles the basic convolution process for 1D case along with non-linear activation function. Notice that we have to do a slight notation change for CNN vs DNN, because the index representation is different. We simply add prefix 'f' to all the terms to make it easier that these are variables to represent filter coefficient or filter output of CNN layers. For bias, I have not change the notation as the indexing is similar to DNN, we have 1 bias per output channel for CNN, we have 1 bias per output node for DNN. If we need to understand backpropagation for CNN, we need to consider two cases.

Case1: We can start with the flattened layer and move backward. For simplicity, lets assume only 1 channel case and the flattened output is same as above, just a vector of X_i'. We will also assume the default value for filter, which is stride of 1, and padding set to "valid" (tensorflow notation), as you can see in figure above. We will deal with max padding later.

Now if we need to find the gradient w.r.t a filter coefficient, we need to take all the contributors. That will be summation across all the output nodes as shown below...

$$\frac{\partial L}{\partial f w_1^n} = \sum_{i=0}^{2} \frac{\partial L}{\partial f x_i^n} \cdot \frac{\partial f x_i^n}{\partial f a_i} \cdot \frac{\partial f a_i}{\partial f w_1} \tag{31}$$

$$= dFLa_0^n \cdot fx_1^{n-1} + dFLa_1^n \cdot fx_2^{n-1} + dFLa_2^n \cdot fx_3^{n-1} \tag{32}$$

Notice the similarity with Equation 24, except now the x_i^{n-1} has to be selected based on filter design and because we have the same filter coefficient (eg fw_1^n in above expression) present in multiple links, we have more than one term instead of just one term in DNN Case. Also we need to use new term for derivative of L w.r.t accumulation function in CNN layer ($dFLa$) vs that of DNN layers (dLa)

Note that the derivative of Loss w.r.t accumulated output can be expressed either in flatten format or in CNN format but since we need to propagate from next layer which is fully connected, it makes sense to use flattened format. The relationship assuming multiple channel in CNN layer is provided after the Figure 5, via Equation 33.

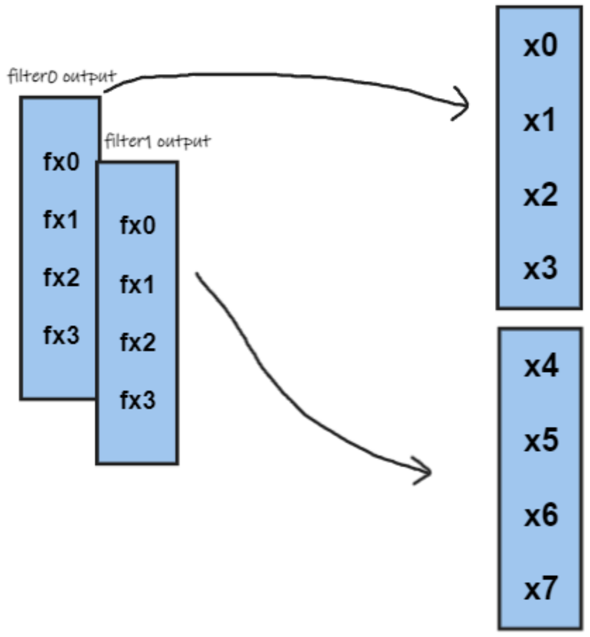

Figure 5: Understanding notation differences between CNN vs DNN

$$dFLa_{j,q}^n = dLa_{Q^n \cdot j+q}^{n+1} \tag{33}$$

... where Q^n is the filter output before flattening (filter0 output and filter1 output length above). It will be the same for all channels. We are treating flattening layer as a separate layer and hence increment in layer index when we go from CNN output (n) to flatten output ($n+1$).

Lets write a more generic equation for gradient, where the filter size for each channel is FS^n,

$$\frac{\partial L}{\partial fw^n_{i,j,c}} = \sum_{q=0}^{Q^n-1} dLa^{n+1}_{Q^n \cdot j+q} \cdot (0 \text{ or } 1)_q \cdot fx^{n-1}_{i,q+c} \tag{34}$$

where the indexes are described as
First index (i) : Represents the filter index from previous layer. Applicable only for fw
Second index (j) : Represents the filter index in the current layer. Applicable for $fw, dFLa, fx, fa$
Third index (c) : Represents the filter coefficient index in the current layer. Applicable for fw

We will need another index to represent the output node index for given filter, lets use (q) for this, and we have already reserved Q for the total length.

Note that Equation 30 is applicable only if stride length is 1 and no padding (to match input size to output size), but is applicable to any number of input or output layers.

Generic equation

$$\frac{\partial L}{\partial fw^n_{i,j,c}} = \sum_{q=0}^{Q^n-1} \underbrace{\left(\sum_{i'=0}^{\text{num nodes in layer n+2 minus 1}} dLa^{n+2}_{i'} \cdot w^{n+2}_{j=Q^n \cdot j+q,i'} \right)}_{dLa^{n+1}_{Q^n \cdot j+q}} \cdot (0 \text{ or } 1)_q \cdot fx^{n-1}_{i,q+c} \tag{35}$$

Similarly gradient for bias :

$$\frac{\partial L}{\partial b^n_j} = \sum_{q=0}^{Q^n-1} dLa^{n+1}_{Q^n \cdot j+q} \tag{36}$$

Case2: Unfortunately to go further into CNN layers, we need to understand the conversion from flatten to non-flatten notation for $dFLa$. Lets first list downt the basic differences between computing for DNN layers vs computing for CNN layers

1. For CNN, the filter coefficients (weights) are the same irrespective of what input pixels or vectors they are applied to. For DNN, we have a fully connected network and there is different weight for each link. The bias on the other hand is quite similar; 1 bias for each neuron in layer for DNN; 1 bias for each filter or channel for CNN.

2. For CNN, not all nodes from previous layer will contributed to next layer, it depends on the filter size! For DNN, we need to consider all nodes from previous layers as its fully connected.

Lets use Figure 4 and identify the gradient for filter weight from layer n

$$\frac{\partial L}{\partial fw^n_{0,0,1}} = \sum_{q=0}^{Q^n-1} \frac{\partial L}{\partial fx^n_{0,q}} \cdot \frac{\partial fx^n_{0,q}}{\partial fa^n_{0,q}} \cdot \frac{\partial fa^n_{0,q}}{\partial fw^n_{0,0,1}} \tag{37}$$

$$= \sum_{q=0}^{Q^n-1} \frac{\partial L}{\partial fx^n_{0,q}} \cdot (0 \text{ or } 1)_{0,q} \cdot fx^{n-1}_{0,q+1} \tag{38}$$

Similar to derivation shown in Equation 22, we need to go backwards,

$$\frac{\partial L}{\partial fx^n_{0,q}} = \sum_{j=0}^{J-1} \sum_{q'=\max(0,q-FS^{n+1}+1)}^{\min(FS^{n+1}-1,q)} \frac{\partial L}{\partial fa^{n+1}_{j,q'}} \cdot \frac{\partial fa^{n+1}_{j,q'}}{\partial fx^n_{0,q}} \tag{39}$$

$$= \sum_{j'=0}^{J'-1} \sum_{q'=\max(0,q-FS^{n+1}+1)}^{\min(FS^{n+1}-1,q)} dFLa^{n+1}_{j',q'} \cdot fw^{n+1}_{0,j',q'} \tag{40}$$

where J' is number of channels in layer $n+1$

Generalizing the expression

$$\frac{\partial L}{\partial fw^n_{i,j,c}} = \sum_{q=0}^{Q^n-1} \frac{\partial L}{\partial fx^n_{j,q}} \cdot (0 \text{ or } 1)_{j,q} \cdot fx^{n-1}_{i,q+c} \tag{41}$$

$$= \sum_{q=0}^{Q^n-1} \underbrace{\left(\sum_{j'=0}^{J'-1} \sum_{q'=\max(0,q-FS^{n+1}+1)}^{\min(FS^{n+1}-1,q)} dFLa^{n+1}_{j',q'} \cdot fw^{n+1}_{j,j',q'} \right) \cdot (0 \text{ or } 1)^n_{j,q}}_{dFLa^n_{j,q}} \cdot fx^{n-1}_{i,q+c} \tag{42}$$

Almost identical to DNN case, we can write for Bias!

$$\frac{\partial L}{\partial b^n_j} = \sum_{q=0}^{Q^n-1} dFLa^n_{j,q} \tag{43}$$

3 Forward Propagation Derivation and other details

The forward propagation is just applying accumulation function, followed by non-linear activation function to update node value and we move from input to output as opposed to output to input for backpropagation. This is applicable for all layers except the last layer where we we skip non-linear activation function and apply softmax.

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 range is within ± 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