TRANSFORMER MODEL NOTES

Here is the Secret

Author
The Great Loser Shivam Mandloi
maybe from IISc
2025

Contents

1 Introduction: **3**

2 Gradient Descent and Neural Network (NN) **3**

2.1 Vector Calculus 5

3 Blocks used in NN **7**

3.1 Chain Rule 7

3.2 Linear Layer 8

3.3 ReLU (Rectified Linear Unit) 10

3.4 SoftMax 10

3.5 Cross Entropy 12

3.6 MSE (Mean Squared Error) 13

4 Gradient Descent **13**

4.1 Gradient Descent with momentum 14

4.2 AdaGrad (Adaptive Gradient Algorithm) 14

4.3 RMSprop (Root Mean Square propagation) 14

4.4 Adam (Adaptive Moment Estimation) 15

1 Introduction:

In this notes I am going to create transformer model from scratch in C++ and use only raw array. I am glad that you ask why??, here are some reasons, what push me to do that

1. Creating things from scratch always helps you to understand the real meaning, what is going on? and this helps you to understand new architectures, which is very necessary in the time when AI is changing very rapidly.

2. It will also helps you to create strong command on the libraries like Torch, TensorFlow etc. which we generally used to write current nn code.

3. It make your command stronger on the transformer based architecture, which we use generally in sequential data.

3. It is cool to say I create a transformer from scratch, it help me to fool other that I am very smart (for 5 minute after that, they figure it out).

Now that we have convinced ourself that it is too important, we can start work on real thing.
This notes must have lot's of issue, as you know author is kind of, you know right, so that's why if you find any issue can you please also share with this person too The man.

2 Gradient Descent and Neural Network (NN)

The Neural Network is universal function approximation, which can approximate any continuous function with non zero accuracy (Goodfellow Book) by one hidden layer and non linear activation function, that's why all the AI model currently use NN very much. The neural network idea come from the logistic, linear regression, where we use linear function to classify the two classes. Because we are using the linear functions, we have some limit to use this method on different problems, the reason is many problem in real world often have some non linear relation, and that's why linear function is not able to find the relation. To solve this problem we can convert our input to different dim or space, such that in new representation of input can be linearly separable, we can think that we are using similar kind of method used in SVM, where we give the input a high dimension and then try in that dimension to separate it linearly. In NN we can also see all the hidden layer as a kernel function, so instead of directly use linear layer (in output layer), we use some hidden layer to transform our matrix to some other space/dim, by learning all the hidden layer, then use the output layer to predict the output.
But the problem is also with NN is that, it is non linear function approximation (because we use the non linear activation function like Relu, Softmax, tanh etc.), and because we don't have any universal optimization function which can predict all the parameter for NN, to approximate that specific function, this make us to rely on methods which doesn't guaranty us that, to find those optimal parameters. But that also not mean that, we don't have

methods, one of the best method currently to find those optimal parameters is called gradient descent. This algorithm which generally we use to find the parameters to approximate the function.
Now to use the gradient descent we use the gradient (OMG that's the line), to find the direction for the steepest descent, where our minima lie. Because our NN is non linear, it make our cost function to have local minima too, and maximum time when our optimization algorithm find the parameter's, which come to as a local minima (that's why it is very tough for the AI to get the near 100% accuracy). Now in around all the time we are finding the local minima only, but obviously local minima maybe not always good choice, suppose if difference between my local minima and global minima is too high, then in that case I would not able to get good generalize error. To solve this issue we can increase the number of hidden size, increasing the number of hidden size will make less difference between local minima and global minima (There is not any proof for that, it is just practically, that means there is not any math's which tells you how much hidden layer you need). Another thing is that more dense network is generalize fast too, now if we train to much, it memorize the input and then it is not give us good generalization error even when we have less training error.

But the gradient descent is simple and useful algorithm currently, so that's why we are also going to use the same in this project. To know how gradient helps us to find the direction of the minima, suppose u is some unit direction, and $\nabla f(x)$ is gradient at point x, then by directional derivative and dot product definition we can write

$$u.\nabla f(x) = ||u||\ ||\nabla f(x)||\ cos\theta$$

Here θ is angle between vector u and $\nabla f(x)$. To get the direction where our minima lie, we have to only find the direction where our gradient is decreasing the most.

$$\min_u u.\nabla f(x) = \min_u ||u||\ ||\nabla f(x)||\ cos\theta$$

because u is a unit vector, $||u|| = 1$,

$$\min_u u.\nabla f(x) = \min_u ||\nabla f(x)||\ cos\theta$$

$$\min_u u.\nabla f(x) = ||\nabla f(x)||\ \min_u cos\theta$$

$cos\theta$ range is between $[-1, 1]$, that means we can find minimum value at where $cos\theta = -1$ means when $\theta = 180^{\cdot}$, which is the opposite direction of the gradient, that give us, that at any point x in the function $-\nabla f(x)$ give the direction where our minima lie (even if it is local minima).

Now by above whatever thing, we got the direction of where our minima lies, we have to only change our parameters, towards the direction of the $-\nabla f(x)$. Above we only talk about the direction and not the magnitude of the change, and the thing is that we also not know, how much we have to change to reach to the minimum value. So for this we are generally multiply very small number α, with direction, to get the vector which has right direction

but have small step, we iteratively update our weight's towards the direction of minima, in each iteration we update the weight one step close to the minima. Here α is called step size, we maximum time take it as a constant value but we can also make it learnable parameter, we saw this thing later.
Now the one of the complex part to code is to find the gradient of the neural network. We use backpropagation algorithm to find the gradient of the NN, this algorithm is not limited to NN, but we can also use it to find the derivative of different algorithm. More generalized version of the backpropagation is the Automatic differential, which is used by torch and tensorflow, to implement the backpropagation. This algorithm used vector calculus to work with function in higher dimension.

2.1 Vector Calculus

Vector calculus, helps us to find the derivative of the higher dim function, suppose we have $f : \mathbb{R}^n \to \mathbb{R}^m$, then to find the derivative of f w.r.t., the input $x \in \mathbb{R}^n$, we need vector calculus. Now here I am only mention the little part of vector calculus, which is only helped (because he also know this much only) to find the gradient used in NN. If you are interested to know full idea, you can check on internet there are plenty of books available (This notes helped me "CMU School of Computer Science matrix calculus notes", you can also check some survey paper on the vector calculus for NN).
Now suppose $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$ are column vector and, there is one function $f : \mathbb{R}^n \to \mathbb{R}^m$, then the derivative of the $f(x)$ w.r.t. the x, $\nabla f \in \mathbb{R}^{m\times n}$ such that.

$$\frac{dy}{dx} = \begin{bmatrix} \frac{dy_1}{dx_1} & \frac{dy_1}{dx_2} \cdots \frac{dy_1}{dx_n} \\ \frac{dy_2}{dx_1} & \frac{dy_2}{dx_2} \cdots \frac{dy_2}{dx_n} \\ & \vdots & \\ \frac{dy_m}{dx_1} & \frac{dy_m}{dx_2} \cdots \frac{dy_m}{dx_n} \end{bmatrix}$$

Here we can simply visualize that $y \in \mathbb{R}^m$ is column vector, where for each element in vector we find the derivative of the element w.r.t. to each element in $x \in \mathbb{R}^n$, such that the resultant Jacobian matrix has dimension $\frac{dy}{dx} \in \mathbb{R}^{m\times n}$, I generally used this formula to find any dimension of Jacobian matrix.

$$\nabla y = \frac{dy}{dx} \nabla x$$

Here ∇y is a small change in output direction and ∇x is a small change in input direction, now the dimension of $\frac{dy}{dx}$ is same which result the output dimension, ∇y, that means let's suppose if our, change in output which has same dimension as $y \in \mathbb{R}^m$ and change in input which has dimension $x \in \mathbb{R}^n$, then the Jacobian matrix will have dimension $\mathbb{R}^{m\times n}$, which satisfy the dimension in above multiplication.
For the case when our $y \in \mathbb{R}$ and y is higher dimension tensor, in that case, the resultant derivative is also higher dimension tensor. let's suppose we have to find the derivative of $y \in \mathbb{R}$, in that case if $x \in R^m$.

$$\frac{dy}{dx} = \begin{bmatrix} \frac{dy}{dx_1} & \frac{dy}{dx_2} \cdots \frac{dy}{dx_n} \end{bmatrix}$$

Same if our $x \in \mathbb{R}^{m\times n}$, then in this case we can assume x as one flatten vector, where if we can stack each column of the matrix in one vector, then the above x become the vector $x \in \mathbb{R}^{mn}$

$$x = \begin{bmatrix} x_{11}\ x_{12} \dots x_{1n} \\ x_{21}\ x_{22} \dots x_{2n} \\ \vdots \\ x_{m1}\ x_{m2} \dots x_{mn} \end{bmatrix}$$

$$vect(x) = \begin{bmatrix} x_{11} \\ x_{12} \\ \vdots \\ x_{1n} \\ x_{21} \\ \vdots \\ x_{m1} \\ \vdots \\ x_{mn} \end{bmatrix}$$

Now, if we find the derivative of $y \in \mathbb{R}$, in that case we can directly use $vect(x)$,

$$\frac{dy}{d\ vect(x)} = \begin{bmatrix} \frac{dy}{dx_{11}} & \frac{dy}{dx_{12}} \cdots \frac{dy}{dx_{1n}} & \frac{dy}{dx_{21}} \cdots \frac{dy}{dx_{m1}} \cdots \frac{dy}{dx_{mn}} \end{bmatrix}$$

$$\frac{dy}{d\ vect(x)} = \begin{bmatrix} \frac{dy}{dx_{11}} & \frac{dy}{dx_{12}} \cdots \frac{dy}{dx_{1n}} \\ \frac{dy}{dx_{21}} & \frac{dy}{dx_{22}} \cdots \frac{dy}{dx_{2n}} \\ & \vdots \\ \frac{dy}{dx_{m1}} & \frac{dy}{dx_{m2}} \cdots \frac{dy}{dx_{mn}} \end{bmatrix}$$

We can also use same trick to find the derivative of other higher dimension tensor, for example suppose we have $y \in \mathbb{R}^k$ and $x \in \mathbb{R}^{m\times n}$, then to find the $\frac{dy}{dx}$, we can use same trick where, we first use the above rule to find derivative of each vector w.r.t. to other vector and then we convert to it's original shape.

$$\frac{dy}{d\ vect(x)} = \begin{bmatrix} \frac{dy_1}{dx_{11}} & \frac{dy_1}{dx_{12}} \cdots \frac{dy_1}{dx_{1n}} & \frac{dy_1}{dx_{21}} \cdots \frac{dy_1}{dx_{m1}} \cdots \frac{dy_1}{dx_{mn}} \\ \frac{dy_2}{dx_{11}} & \frac{dy_2}{dx_{12}} \cdots \frac{dy_2}{dx_{1n}} & \frac{dy_2}{dx_{21}} \cdots \frac{dy_2}{dx_{m1}} \cdots \frac{dy_2}{dx_{mn}} \\ & \vdots & \\ \frac{dy_k}{dx_{11}} & \frac{dy_k}{dx_{12}} \cdots \frac{dy_k}{dx_{1n}} & \frac{dy_k}{dx_{21}} \cdots \frac{dy_k}{dx_{m1}} \cdots \frac{dy_k}{dx_{mn}} \end{bmatrix}$$

The above term will become the 3^{rd} order tensor, now to find the exactly tensor format, we can use the above trick, where we find the derivative of scalar w.r.t. the matrix, we use the scalar and find the derivative by each element in matrix. Here despite one scalar we have the vector, above when we fold it down then each new column come as a new dimension in tensor. For example in above case when we fold the the derivative vector, we add new row and create a matrix. As the same reason here we have matrix and if we want to fold this matrix, a new elements goes to new dimension which is new matrix and we got tensor with dimension $\frac{dy}{dx} = \mathbb{R}^{m\times k\times n}$. I know this is stupid idea, and yes I am also sure there is much good reason also existed rather than just finding the similarity, but the thing is that it worked.

3 Blocks used in NN

There are different blocks we use when create any neural network architecture, some of them are linear layer, activation function (softmax, sigmoid, tanh, RELU etc.) and loss function. To create transformers block, we also need these blocks. Now to use the gradient descent optimization, we need to find the derivative of our neural network, for that we use the backpropagation. We also need to have derivative of each those block with respect to the input and learnable parameter. So in this section we are going to find derivative for each important block which helped us in transformer, we also talk about additional block which is not directly used but can helped us in different situation.
Before start to explore the different nn blocks, let's finalize some notation which we going to use, any vector, like $x \in \mathbb{R}^n$ or even derivative $\frac{dy}{dx} \in \mathbb{R}^n$ will be a column vector, and if we have to explicitly make the row vector we can use this notation x^T or $(\frac{dy}{dx})^T$ to say that vector is row vector. One can also use the row vector instead of column vector but then, all the blocks math's should satisfied dimension of different equation and derivative, but all the block math's will be same just dimension of result will be changed.

3.1 Chain Rule

We are using chain rule to find the derivative of loss w.r.t. to different parameter present in different layer. NN is non polynomial approximation, we can see NN as sequence of function, suppose we have function $g(x)$ and $f(y)$ such that.

$$L(x) = g(f(x))$$

Then to find the derivative of $L(x)$ w.r.t. to x we can use chain rule.

$$\frac{dL(x)}{dx} = \frac{dg(f(x))}{df(x)} \frac{df(x)}{dx}$$

This same chain rule we can use in to find the derivative in NN. suppose we have this function

$$y = W^{(2)} g(W^{(1)} x + b^{(1)}) + b^{(2)}$$

Here $W^{(1)}, W^{(2)}, b^{(1)}$ and $b^{(2)}$ are parameter's and g is activation function. Then we can also see this above as a composition function such that.

$$y = L^{(2)}(g(L^{(1)}(x)))$$

Now in case of higher dimension function, which is often case in neural network, we can fix that each derivative in chain is column vector.

3.2 Linear Layer

This is one of the important block which you can around in any NN model, it is kind of basic building block. Logistic and linear learning are those methods which used linear layer only, we can use this layer to find the hyperparameter to separate two classes, we can also use it to generate the parameter for any distribution. Here is equation of the linear layer.

$$y = Wx + b$$

Here $y \in \mathbb{R}^m$, $x \in \mathbb{R}^n$, $W \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} w_{11}\ w_{12} \dots w_{1n} \\ w_{21}\ w_{22} \dots w_{2n} \\ \vdots \\ w_{m1}\ w_{m2} \dots w_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \tag{3.1}$$

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} x_1 w_{11} + x_2 w_{12} + \dots + x_n w_{1n} + b_1 \\ x_1 w_{21} + x_2 w_{22} + \dots + x_n w_{2n} + b_2 \\ \vdots \\ x_1 w_{m1} + x_2 w_{m2} + \dots + x_n w_{mn} + b_m \end{bmatrix}$$

Now we need to find three derivatives from the above equation, $\frac{dy}{dW}$, $\frac{dy}{db}$ and $\frac{dy}{dx}$. To find the derivative w.r.t. W which is matrix we have to flatten the w matrix.

$$\frac{dy}{dW} = \begin{bmatrix} \frac{d\ x_1 w_{11} + x_2 w_{12} + \dots + x_n w_{1n} + b_1}{dw_{11}} & \dots & \frac{d\ x_1 w_{11} + x_2 w_{12} + \dots + x_n w_{1n} + b_1}{dw_{mn}} \\ \frac{d\ x_1 w_{21} + x_2 w_{22} + \dots + x_n w_{2n} + b_2}{dw_{11}} & \dots & \frac{d\ x_1 w_{21} + x_2 w_{22} + \dots + x_n w_{2n} + b_2}{dw_{mn}} \\ & \vdots & \\ \frac{d\ x_1 w_{m1} + x_2 w_{m2} + \dots + x_n w_{mn} + b_m}{dw_{11}} & \dots & \frac{d\ x_1 w_{m1} + x_2 w_{m2} + \dots + x_n w_{mn} + b_m}{dw_{mn}} \end{bmatrix}_{m \times mn}$$

Now because I don't have space so I write some portion from the equation (laziness and lie both are key of unsuccess), but when we find the derivative, then above equation become

$$\frac{dy}{dW} = \begin{bmatrix} x_1\ x_2\ \dots\ x_n\ (0)^{21}\ (0)^{22} \dots (0)^{2n} \dots\dots (0)^{m1} (0)^{m2} \dots (0)^{mn} \\ (0)^{11}\ (0)^{12} \dots (0)^{1n}\ x_1\ x_2\ \dots\ x_n \dots\dots (0)^{m1} (0)^{m2} \dots (0)^{mn} \\ \vdots \\ (0)^{11}\ (0)^{12} \dots (0)^{1n}\ (0)^{21} (0)^{22} \dots (0)^{2n} \dots\dots\ x_1\ x_2\ \dots\ x_n \end{bmatrix}_{m \times mn}$$

Here $(0)^{ij}$ is 0 derivative w.r.t. w_{ij}. Now here is tensor format of above equation

$$\frac{dy}{dW} = \begin{bmatrix} \begin{bmatrix} x_1\ x_2\ \dots\ x_n \\ 0\ 0 \dots 0 \\ \vdots \\ 0\ 0 \dots 0 \end{bmatrix}_{m \times n} & \begin{bmatrix} 0\ 0 \dots 0 \\ x_1\ x_2\ \dots\ x_n \\ \vdots \\ 0\ 0 \dots 0 \end{bmatrix}_{m \times n} & \dots\dots & \begin{bmatrix} 0\ 0 \dots 0 \\ 0\ 0 \dots 0 \\ \vdots \\ x_1\ x_2\ \dots\ x_n \end{bmatrix}_{m \times n} \end{bmatrix}_{m \times m \times n} \tag{3.2}$$

This derivative we are going to use in backpropagation. Now to find the derivative w.r.t. to

x and b, we can use (1) again

$$\frac{dy}{db} = \begin{bmatrix} \frac{x_1w_{11}+x_2w_{12}+\cdots+x_nw_{1n}+b_1}{db_1} \dots \frac{x_1w_{11}+x_2w_{12}+\cdots+x_nw_{1n}+b_1}{db_m} \\ \frac{x_1w_{21}+x_2w_{22}+\cdots+x_nw_{2n}+b_2}{db_1} \dots \frac{x_1w_{21}+x_2w_{22}+\cdots+x_nw_{2n}+b_2}{db_m} \\ \vdots \\ \frac{x_1w_{m1}+x_2w_{m2}+\cdots+x_nw_{mn}+b_m}{db_1} \dots \frac{x_1w_{m1}+x_2w_{m2}+\cdots+x_nw_{mn}+b_m}{db_m} \end{bmatrix}_{m\times m}$$

$$\frac{dy}{db} = \begin{bmatrix} 1\ 0 \dots 0 \\ 0\ 1 \dots 0 \\ \vdots \\ 0\ 0 \dots 1 \end{bmatrix}_{m\times m} \tag{3.3}$$

$$\frac{dy}{dx} = \begin{bmatrix} \frac{x_1w_{11}+x_2w_{12}+\cdots+x_nw_{1n}+b_1}{dx_1} \dots \frac{x_1w_{11}+x_2w_{12}+\cdots+x_nw_{1n}+b_1}{dx_n} \\ \frac{x_1w_{21}+x_2w_{22}+\cdots+x_nw_{2n}+b_2}{dx_1} \dots \frac{x_1w_{21}+x_2w_{22}+\cdots+x_nw_{2n}+b_2}{dx_n} \\ \vdots \\ \frac{x_1w_{m1}+x_2w_{m2}+\cdots+x_nw_{mn}+b_m}{dx_1} \dots \frac{x_1w_{m1}+x_2w_{m2}+\cdots+x_nw_{mn}+b_m}{dx_n} \end{bmatrix}_{m\times n}$$

$$\frac{dy}{dx} = \begin{bmatrix} w_{11}\ w_{12} \dots w_{1n} \\ w_{21}\ w_{22} \dots w_{2n} \\ \vdots \\ w_{m1}\ w_{m2} \dots w_{mn} \end{bmatrix}_{m\times n} \tag{3.4}$$

Equation (3.2), (3.3) and (3.4), are the equation we are going to use in backpropagation. Now suppose we have this function.

$$l = L(y)$$

Here y is linear function define above, to find the derivative of $\frac{dl}{dw}$, $\frac{dl}{db}$ and $\frac{dl}{dx}$ (here last derivative then used by previous layer in NN), we can use chain rule and here is final version.

$$\frac{dl}{dw} = \frac{dl}{dy}\frac{dy}{dw}$$

$$\frac{dl}{db} = \frac{dl}{dy}\frac{dy}{db}$$

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

First two derivative we can use in optimization algorithm, last one is pass to previous layer, Now here we can write derivative this way.

$$\frac{dl}{dw} = \frac{dl}{dy}x^T$$

$$\frac{dl}{db} = \frac{dl}{dy}$$

$$\frac{dl}{dx} = ((\frac{dl}{dy})^T W)^T = W^T\frac{dl}{dy}$$

Here I used above derivative which we find, and assume that the derivative of $\frac{dl}{dy}$ is column vector and same vector format I also provide to previous layer, by transpose the result to find the $\frac{dl}{dx}$.

3.3 ReLU (Rectified Linear Unit)

ReLU is activation function which we generally used in hidden layer.

$$y = g(x)$$

Here g is ReLU activation function, suppose $x \in \mathbb{R}^n$, then the result is also $y \in \mathbb{R}^n$ such that,

$$y = \begin{cases} x, & \text{if } x \geq 0, \\ 0, & \text{else} \end{cases}$$

Now to find the derivative, where input has dimension $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^n$, the derivative will be matrix with dimension $\mathbb{R}^{n \times n}$ by the vector calculus. Because ReLU function is not doing much just making input element to 0 which has negative value, so it's derivative is also sparse matrix, a diagonal matrix which has 1, if input was the non negative value and 0 if it is negative value.

$$\frac{dy}{dx} = \begin{cases} 1, & \text{if } x > 0, \\ \text{undefined} & x = 0 \\ 0, & \text{else} \end{cases}$$

Here at $x = 0$, the derivative of ReLU function is undefined, because the left derivative and right derivative does not match. Because there is not any learning parameter we have to find only one derivative.
Now suppose we have this chain of derivative

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

Here suppose x is input to ReLU function and y is output, then $\frac{dy}{dx}$ is matrix describe above, where it makes element of $\frac{dl}{dx}$, 0 if it's corresponding input was negative.

3.4 SoftMax

This activation function generally used in output layer to convert any vector of elements to the distribution. The formula of the activation function give such that, suppose $x \in \mathbb{R}^n$ is input vector to the softmax function then

$$y = softmax(x) \in \mathbb{R}^n$$

here,

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

Now it is not always necessary to use this function in output layer, we can use it in hidden layer too, it all depends on the use case and the maximum time we need distribution at the

output layer (for example, the LLM model generates distribution over all the vocabulary for the next word prediction). In transformer block we used this activation function to calculate the attention for different elements in sequence. Another thing is we clip the input array in softmax to some max and min value, so that our exponential does not become too large. Clip value of input array between $[-50, 50]$, work well. Another way is to subtract each element in input vector by the max element in vector.
Now to use backpropagation we need to find the derivative of softmax function w.r.t. input vector x.

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} \frac{e^{x_1}}{\sum_{j=1}^{n} e^{x_j}} \\ \frac{e^{x_2}}{\sum_{j=1}^{n} e^{x_j}} \\ \vdots \\ \frac{e^{x_n}}{\sum_{j=1}^{n} e^{x_j}} \end{bmatrix}$$

$$\begin{bmatrix} \frac{dy_1}{dx_1} & \frac{dy_1}{dx_2} & \cdots & \frac{dy_1}{dx_n} \\ \frac{dy_2}{dx_1} & \frac{dy_2}{dx_2} & \cdots & \frac{dy_2}{dx_n} \\ & \vdots & & \\ \frac{dy_n}{dx_1} & \frac{dy_n}{dx_2} & \cdots & \frac{dy_n}{dx_n} \end{bmatrix} = \begin{bmatrix} \frac{d}{dx_1}\frac{e^{x_1}}{\sum_{j=1}^{n} e^{x_j}} & \frac{d}{dx_2}\frac{e^{x_1}}{\sum_{j=1}^{n} e^{x_j}} & \cdots & \frac{d}{dx_n}\frac{e^{x_1}}{\sum_{j=1}^{n} e^{x_j}} \\ \frac{d}{dx_1}\frac{e^{x_2}}{\sum_{j=1}^{n} e^{x_j}} & \frac{d}{dx_2}\frac{e^{x_2}}{\sum_{j=1}^{n} e^{x_j}} & \cdots & \frac{d}{dx_n}\frac{e^{x_2}}{\sum_{j=1}^{n} e^{x_j}} \\ & \vdots & & \\ \frac{d}{dx_1}\frac{e^{x_n}}{\sum_{j=1}^{n} e^{x_j}} & \frac{d}{dx_2}\frac{e^{x_n}}{\sum_{j=1}^{n} e^{x_j}} & \cdots & \frac{d}{dx_n}\frac{e^{x_n}}{\sum_{j=1}^{n} e^{x_j}} \end{bmatrix} \tag{3.4.1}$$

This took me too much time to write (so errors can also exist, find them, means error can be everywhere but this is the capital).
We can now see that it will not be a sparse derivative like the ReLU function. Here every element in matrix has been calculating by using all the elements of input vector. But if we see carfully then there are only two cases, first is $\frac{dy_i}{dx_i}$ and second is $\frac{dy_i}{dx_j}$, that means if we find this two derivative in generalized version we can able to find the full derivative matrix.
First let find the first derivative

$$\frac{dy_i}{dx_i} = \frac{d}{dx_i}\frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

$$\frac{dy_i}{dx_i} = \frac{(\sum_{j=1}^{n} e^{x_j})\ \frac{de^{x_i}}{dx_i} - e^{x_i}\ (\frac{d\sum_{j=1}^{n} e^{x_j}}{dx_i})}{(\sum_{j=1}^{n} e^{x_j})^2}$$

$$\frac{dy_i}{dx_i} = \frac{(\sum_{j=1}^{n} e^{x_j})\ e^{x_i} - e^{x_i}\ e^{x_i}}{(\sum_{j=1}^{n} e^{x_j})^2}$$

$$\frac{dy_i}{dx_i} = \frac{(\sum_{j=1}^{n} e^{x_j})\ e^{x_i}}{(\sum_{j=1}^{n} e^{x_j})^2} - \frac{e^{x_i}\ e^{x_i}}{(\sum_{j=1}^{n} e^{x_j})^2}$$

$$\frac{dy_i}{dx_i} = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} - \frac{e^{x_i}\ e^{x_i}}{(\sum_{j=1}^{n} e^{x_j})(\sum_{j=1}^{n} e^{x_j})}$$

$$\frac{dy_i}{dx_i} = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}(1 - \frac{e^{x_i}}{(\sum_{j=1}^{n} e^{x_j})})$$

$$\frac{dy_i}{dx_i} = y_i(1 - y_i) \tag{3.4.2}$$

Now let's find the second derivative.

$$\frac{dy_i}{dx_j} = \frac{d}{dx_j} \frac{e^{x_i}}{\sum_{k=1}^{n} e^{x_k}}$$

$$\frac{dy_i}{dx_j} = \frac{(\sum_{k=1}^{n} e^{x_k})\ \frac{de^{x_i}}{dx_j} - e^{x_i}\ (\frac{d \sum_{k=1}^{n} e^{x_k}}{dx_j})}{(\sum_{k=1}^{n} e^{x_k})^2}$$

$$\frac{dy_i}{dx_j} = \frac{(\sum_{k=1}^{n} e^{x_k})\ 0 - e^{x_i}\ e^{x_j}}{(\sum_{k=1}^{n} e^{x_k})^2}$$

$$\frac{dy_i}{dx_j} = \frac{-e^{x_i}\ e^{x_j}}{(\sum_{k=1}^{n} e^{x_k})^2}$$

$$\frac{dy_i}{dx_j} = -y_i\ y_j \tag{3.4.3}$$

Using Equations (3.4.2) and (3.4.3), we can find the derivative matrix (3.4.1)

$$\frac{dy}{dx} = \begin{bmatrix} \frac{dy_1}{dx_1} & \frac{dy_1}{dx_2} & \cdots & \frac{dy_1}{dx_n} \\ \frac{dy_2}{dx_1} & \frac{dy_2}{dx_2} & \cdots & \frac{dy_2}{dx_n} \\ & \vdots & & \\ \frac{dy_n}{dx_1} & \frac{dy_n}{dx_2} & \cdots & \frac{dy_n}{dx_n} \end{bmatrix} = \begin{bmatrix} y_1(1-y_1) & -y_1y_2 \cdots - y_1y_n \\ -y_1y_2 & y_2(1-y_2) \cdots - y_2y_n \\ & \vdots \\ -y_ny_1 & -y_ny_2 \ldots y_n(1-y_n) \end{bmatrix}$$

Now we can use this matrix in backpropagation. In ReLU function we used same chain rule to find the derivative of loss w.r.t. input vector x.

$$\frac{dl}{dx} = (\frac{dl}{dy})^T \frac{dy}{dx}$$

Here, $\frac{dl}{dy} \in \mathbb{R}^n$ is a column vector, and $\frac{dy}{dx} \in \mathbb{R}^{n \times n}$

3.5 Cross Entropy

By using neural network we often time want to predict the $p(y|x)$ as an output. Now one of the best cost function in this situation is the Cross Entropy, where we try to maximize the $p(y|x)$, if y is taken from the distribution of true data generating process then this probability should be high. we wanted to make it high as well, that's why we can use this as a cost function.

suppose $A \in \mathbb{R}^n$ is our actual distribution and $P \in \mathbb{R}^n$ is predicted distribution, then in that case we want to maximize this.

$$L = CrossEntropy(A, p)$$

$$L = -\sum_{i=1}^{n} A_i \log P_i$$

Here maximum time our vector A is sparse vector, where only one element of the vector is one and rest are zero, that means we can also use it efficiently.
Now Cross Entropy based on the MLE (maximum likelihood estimation), Where likelihood is how probable our output is, in math terms it is $p(y|x)$, if we compare this thing with

our above equation then we also try to maximize the the term $logp_i$, which is non zero if our A_i is non zero. That means we are trying to maximize the probability of element in distribution which is true value. Because in NN we try to minimize the cost that's why we use negative sign to make maximize problem to minimize problem.
Now let's find the derivative, here P is predicted distribution, and it passed as a input to our loss function, so that's why we are going to find derivative w.r.t. to p

$$\frac{dL}{dp} = \frac{d}{dP}(-\sum_{i=1}^{n} A_i \log P_i)$$

$$\frac{dL}{dp} = \begin{bmatrix} \frac{d}{dP_1}(-\sum_{i=1}^{n} A_i \log P_i) & \frac{d}{dP_2}(-\sum_{i=1}^{n} A_i \log P_i) \dots \frac{d}{dP_n}(-\sum_{i=1}^{n} A_i \log P_i) \end{bmatrix}$$

$$\frac{dL}{dp} = \begin{bmatrix} -\frac{A_1}{P_1} & -\frac{A_2}{P_2} \dots - \frac{A_n}{P_n} \end{bmatrix}$$

But in nn we have only one value which is true in this case rest of above element will be zero except then that element.

3.6 MSE (Mean Squared Error)

Suppose A is actual vector and P is predicted vector by neural network then the MSE L is

$$L = \frac{1}{n}\sum_{i=1}^{n}(P_i - A_i)^2$$

In above term you can also use $A_i - P_i$, and if you consistent with this change in all calculation, you also get same results.
Now to find the derivative of above loss function w.r.t to p will be

$$\frac{dL}{dP} = \frac{d}{dP}\frac{1}{n}\sum_{i=1}^{n}(P_i - A_i)^2)$$

$$\frac{dL}{dP} = \begin{bmatrix} \frac{d}{dP_1}(\frac{1}{n}\sum_{i=1}^{n}(P_i - A_i)^2) & \frac{1}{n}\frac{d}{dP_2}(\sum_{i=1}^{n}(P_i - A_i)^2) \dots \frac{1}{n}\frac{d}{dP_n}(\sum_{i=1}^{n}(P_i - A_i)^2) \end{bmatrix}$$

$$\frac{dL}{dP} = \begin{bmatrix} \frac{2}{n}(P_1 - A_1) & \frac{2}{n}(P_2 - A_2) \dots \frac{2}{n}(P_n - A_n) \end{bmatrix}$$

$$\frac{dL}{dP} = \frac{2}{n}(P - A) \in \mathbb{R}^n$$

4 Gradient Descent

Now as we completed learning about different block which we are going to use, now it's time to know about optimization, generally we are using the gradient descent algorithm, where we try to learn different variant of this algorithms.
Now in gradient descent we are changing our weights w and b, to the direction of the gradient, we talk about this in section (2). Here we only talk about the variants of gradient descent. Every case which we are going to look, will have same steps, like we are updating our parameter in gradient direction, we are just using different way to find the learning

parameter. For notation we are using g_t denote the gradient w.r.t. to loss, for iteration t.

$$g_t = \frac{dL}{dw} \text{ or } \frac{dL}{db}$$

4.1 Gradient Descent with momentum

$$w_{t+1} = w_t - v_{t+1}$$

$$v_{t+1} = \rho v_t + \eta g_t$$

In the above formula ρ is a momentum coefficient, and η is a learning rate. Here instead of using one step gradient, we try to learn the direction from the previous gradient direction too. The intuition is that If the gradients consistently point in the same direction, the velocity v_t will increase exponentially (up to a limit), leading to faster convergence down the slope. By different

4.2 AdaGrad (Adaptive Gradient Algorithm)

(Gemini Explanation)
Unlike standard Gradient Descent (or even Momentum, which uses one learning rate for all parameters), AdaGrad maintains an accumulator, typically denoted as G_t, to track the history of squared gradients for each parameter.

$$\mathbf{G}_t = \mathbf{G}_{t-1} + \mathbf{g}_t \odot \mathbf{g}_t$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

ϵ (epsilon) is a small smoothing term (e.g., 10^{-8}) to prevent division by zero, and $\odot$ denotes the element-wise (Hadamard) product/division.
The magic of AdaGrad lies in the denominator of the parameter update rule: $\sqrt{\mathbf{G}_t + \epsilon}$. This term provides a scale factor that is unique for every parameter

$$\text{Adaptive Learning Rate for } w_i \propto \frac{\eta}{\sqrt{\sum_{\tau=1}^{t} (g_{\tau,i})^2 + \epsilon}}$$

Here τ is sequence in training iteration, so g_τ is all the previous gradient which we calculate for given parameter w.

4.3 RMSprop (Root Mean Square propagation)

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

Here γ is decay rate, we set it mainly 0.9 or 0.99. In AdaGrad we always increases $G_t = \sum_{\tau=1}^{t} g_\tau^2$, causing the learning rate to relentlessly shrink and eventually stall training. Here we in RMSprop we are decays old squared gradients, preventing the accumulator from growing indefinitely and allowing the learning rate to remain stable and non-zero.
In above equation we are calculating $\sqrt{E[g^2]_t + \epsilon}$ element wise.

4.4 Adam (Adaptive Moment Estimation)

First Moment Estimate (mean of gradients)

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

Second Moment Estimate (Uncentered Variance of Gradients)

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(g_t \odot g_t)$$

Here β_1 and β_2 set to 0.9 or 0.999, Since $\mathbf{m}_t$ and $\mathbf{v}_t$ are initialized to zero, they are biased toward zero during the initial steps. Adam corrects this bias

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

The final parameter update uses the corrected moments

$$w_t = w_{t-1} - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$$

References