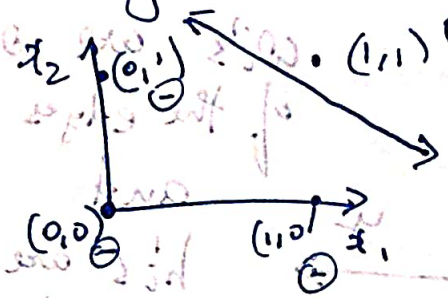
Linearly Separable Functions

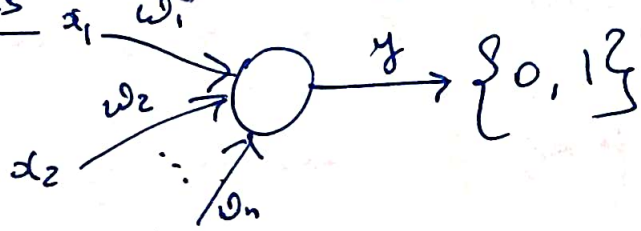
A linearly separable function is a boolean function that can be separated into different classes using a hyperplane in a high-dimensional space.



AND Function is linearly separable.

XOR is not a linearly separable function.

Perceptrons



$y = 1$ if $w_0 + \sum_{i=1}^{n} w_i x_i \geq 0$

$y = 0$ otherwise

a) w_0 is called the bias. if $w_0 \geq 0$, we support +ve class more.

=> We say that the perceptron fires if $w_0 + \sum_{i=1}^{n} x_i w_i \geq 0$

is true

$\{True, False\} \rightarrow \{True, False\}$

=> Any boolean function can be implemented using a network of perceptrons.

Assume that True = +1 and False = -1

For 2 inputs the combinations can be $2^2 = 4$. Hence 2^4 boolean functions can be made for 2 inputs, each combination of input can take output of -1 or 1.

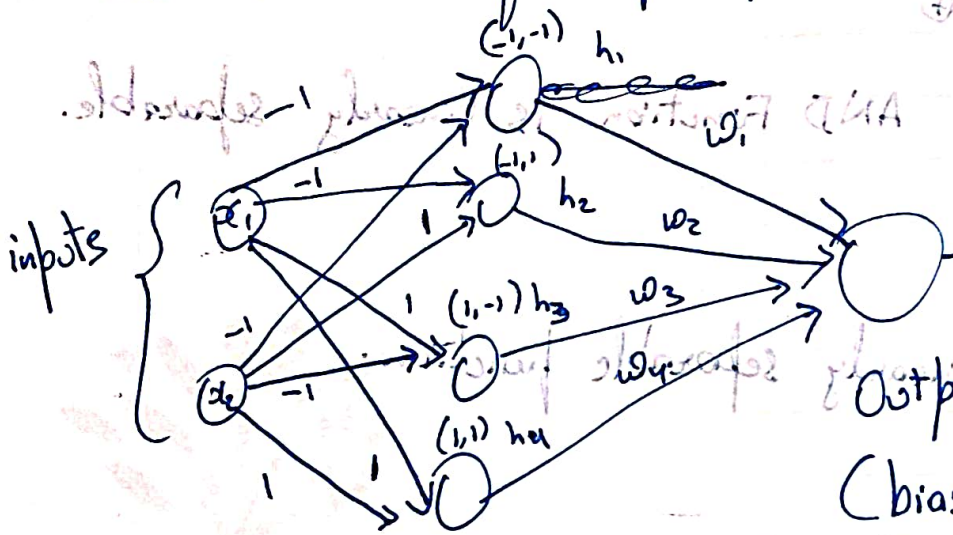
So for n inputs total possible boolean function: 2^{2^n}

=) Example for 2 inputs.

4 possible input combinations:—

$(-1,-1), (-1,1), (1,-1), (1,1)$

Let us make four perceptrons:—



w_i's are weights of the edges and h_i's are the outputs of the hidden layer perceptrons.

Hidden Layer. (bias = -2)

The claim is that this network of perceptrons can represent any boolean function of 2 inputs.

This is because if the input is $(-1,-1)$ only the top neuron will fire, as its output will be $(-1)(-1) + (-1)(-1) - 2$

$= 0 \geq 0$

None other neuron will fire. because we have designed our network in such a way. Hence for each input only the corresponding neuron will fire. So whatever is the desired output for a particular input, that corresponding weight from the corresponding neuron to the output neuron can be adjusted. For example if the desired output from the input $(1,-1)$ is 1, then, only third neuron will

first, hence w_3 can be adjusted w.r.t. w_0 such that $w_3 \geq w_0$. Since no other neuron fires, w_3 can be adjusted independently as it will be the only one contributing to $\sum_{i=1}^{4} w_i I_i \geq w_0$ when input is $(1,-1)$, also w_3 will never contribute whenever input is other than $(1,-1)$ so it will not matter what w_3 is in those cases.

=> When there are 3 inputs, 8 possible input combinations are there. So we use 8 perceptrons in the hidden layer with bias = -3, each firing only to 1 possible input.

Representation power of a multilayer network of perceptrons

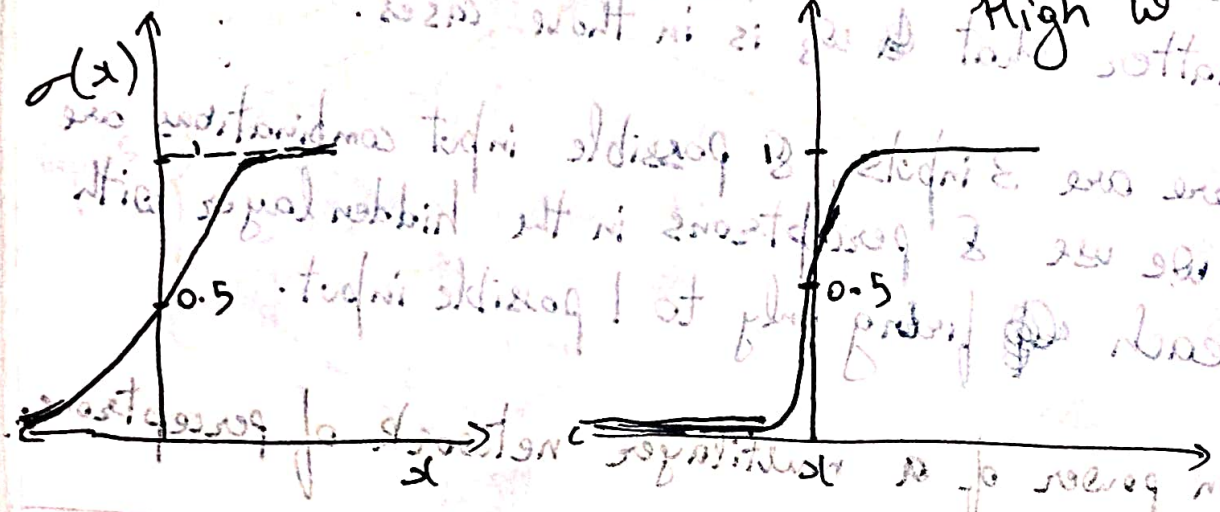
=> A multilayer network of perceptrons with a single hidden layer can be used to represent any boolean function precisely (no errors).

Representation power of a multilayer network of Sigmoid neurons

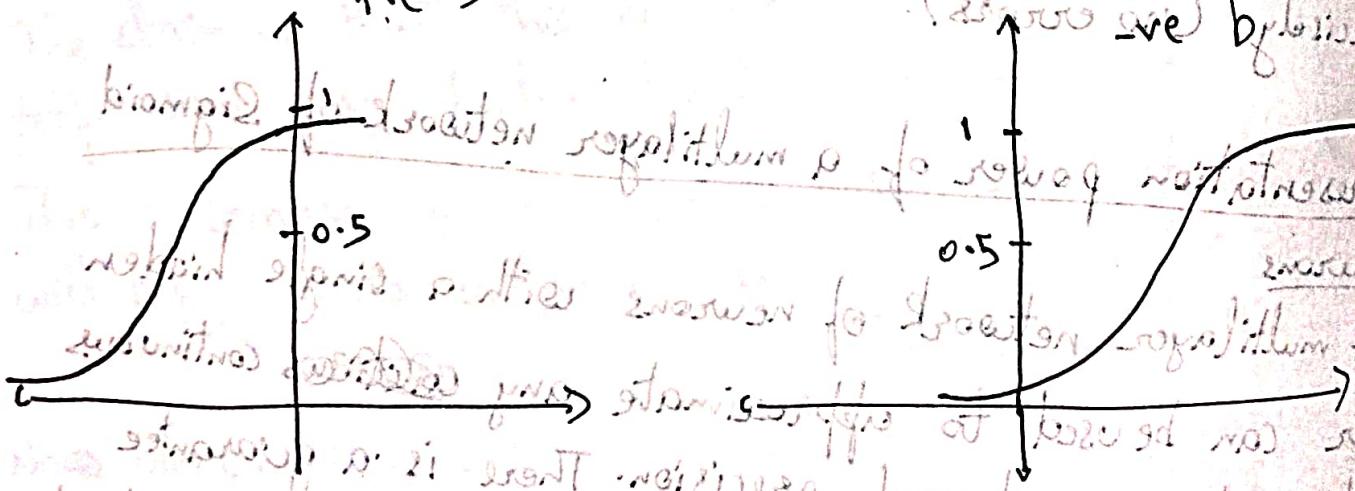
=> A multilayer network of neurons with a single hidden layer can be used to approximate any continuous function to any desired precision. There is a guarantee that for any function $f(x): \mathbb{R}^n \to \mathbb{R}^m$, we can always find a neural network (with 1 hidden layer containing enough neurons) whose output $g(x)$ satisfies $|g(x) - f(x)| < \epsilon$ for any $\epsilon > 0$.

⇒ Some facts about the shape of the sigmoid function (1-D):

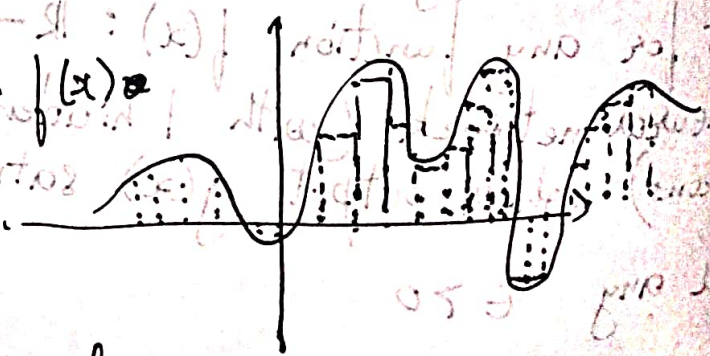
i) If $f(x) = \frac{1}{1+e^{-(wx+b)}}$, when $|w| \to \infty$ the function becomes closer and closer to the step function.



ii) b can varied to linearly translate the position of the curve.



Now Let there be a function $f(x)$

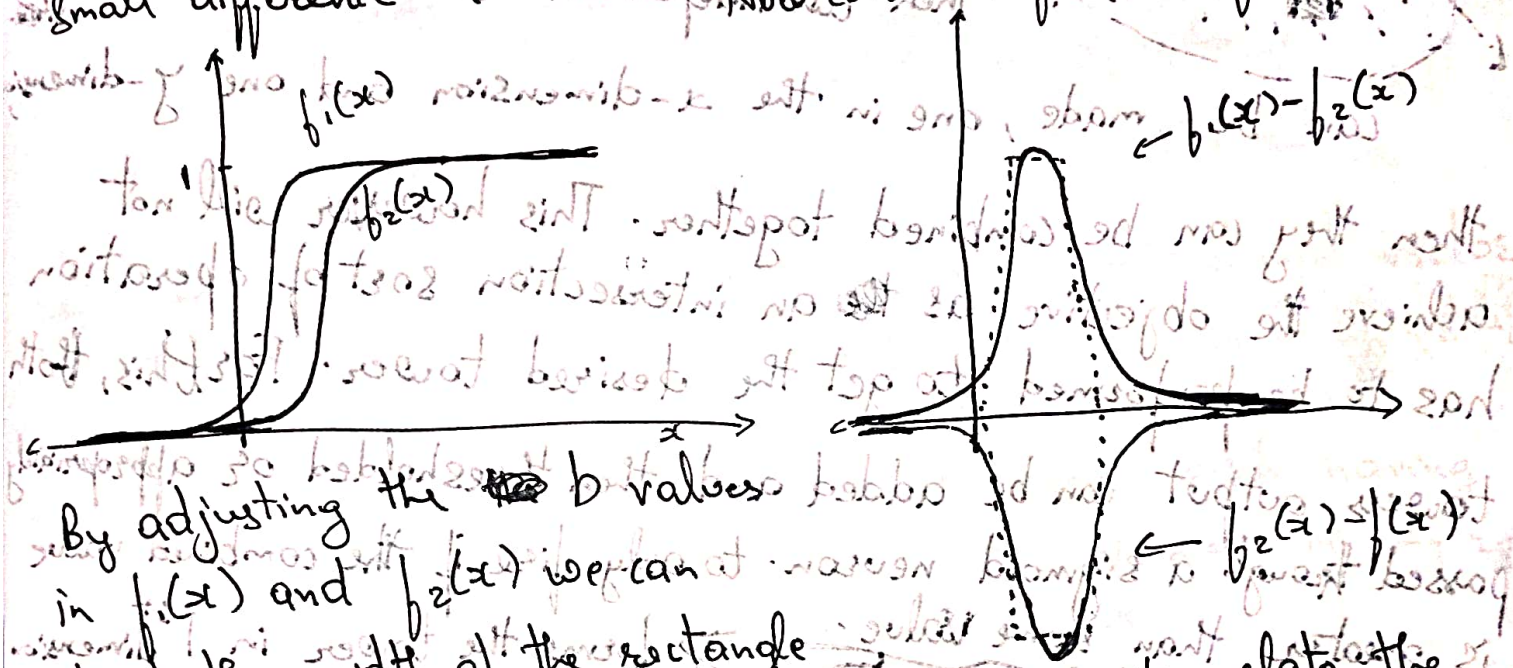


this can be approximated through equal width rectangles.

If we introduce more rectangles by reducing the width of the rectangles, then the approximation is more accurate.

The question is how to create these rectangle like structures using sigmoid neurons.

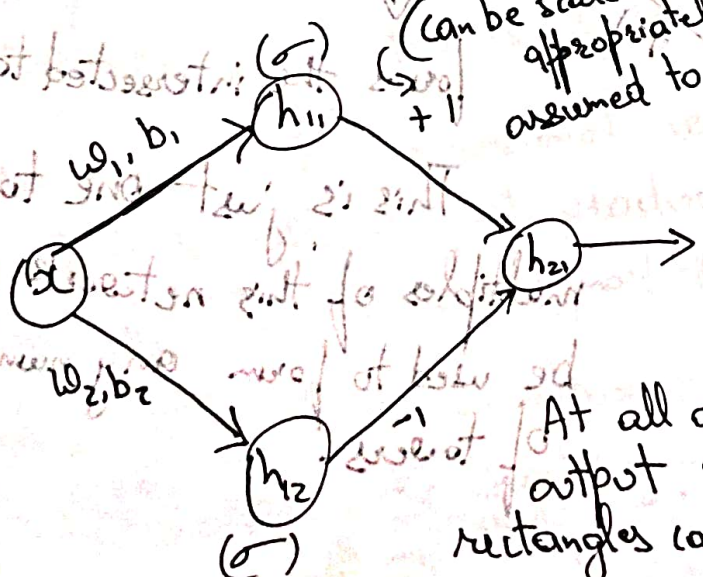
Consider two sigmoid functions with very high w and small difference in b. Now consider $f_1(x) - f_2(x)$



By adjusting the b values in $f_1(x)$ and $f_2(x)$ we can adjust the width of the rectangle like structure, we can also manipulate it to translate the structure linearly.

The height of the rectangle can be manipulated by scaling the output of the sigmoid appropriately.

Combining this into a neural network:-

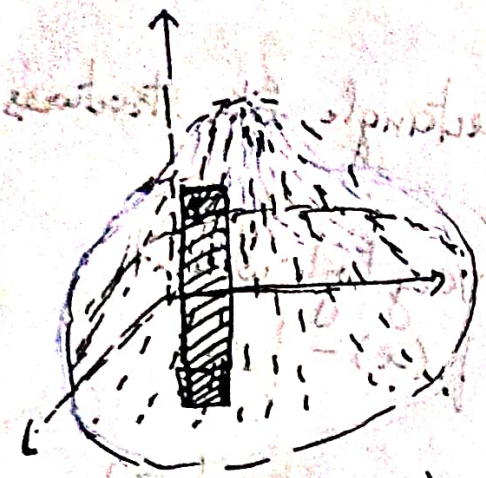


(can be scaled appropriately to achieve the correct height, assumed to be 1 for simplicity)

Notice that by adjusting w's and b's we can make a very specific rectangle at desired position and height. At all other x input, the network will output 0 and won't fire. Any number of rectangles can be made to approximate the function by using the multiples of this network.

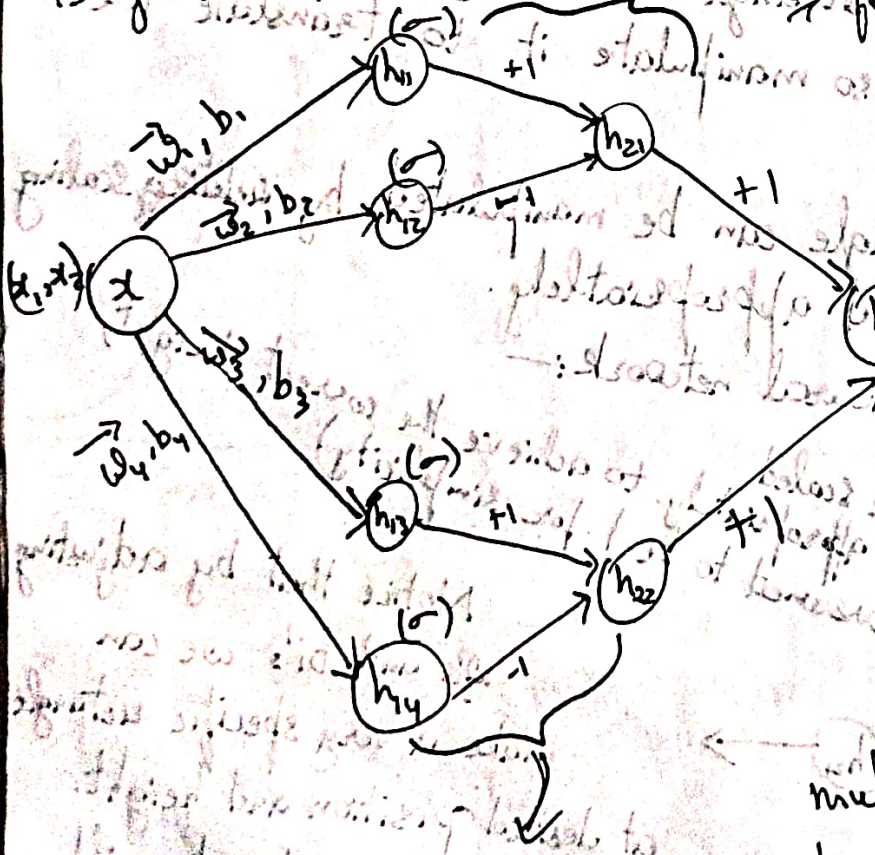
For 2-D case :- each dimension can have its own tower.

Suppose the tower has to be made between the area :- $\begin{bmatrix} (x_{min} - x_{max}) \\ (y_{min} - y_{max}) \end{bmatrix}$



then two independent towers can be made, one in the x-dimension and one y-dimension.

then they can be combined together. This however will not achieve the objective as an intersection sort of operation has to be performed to get the desired tower. For this, both towers output can be added and then thresholded or appropriately passed through a sigmoid neuron to only fire if the combined value is greater than some value.



forms the tower in 1st dimension say.

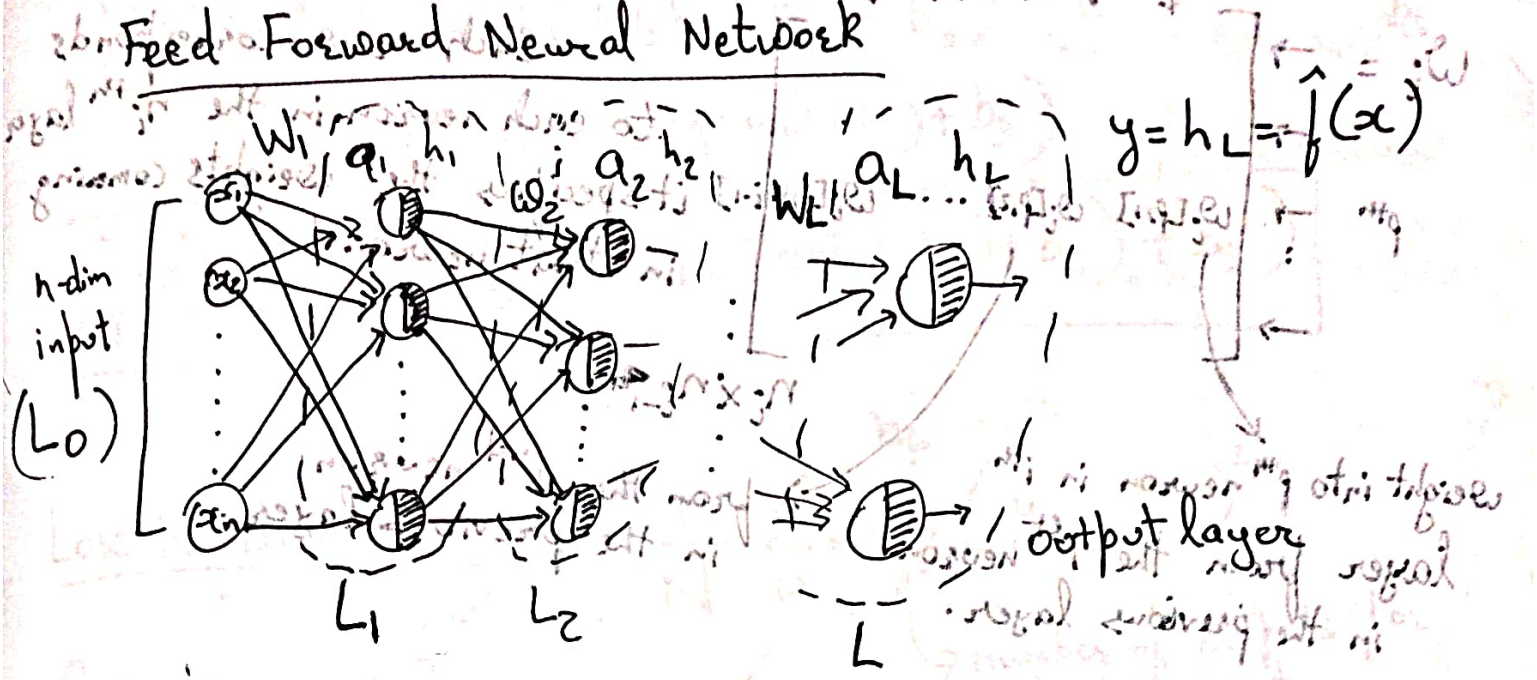
The hidden layer weights can be appropriately scaled.

forms the intersected tower.

This is just one tower, multiples of this network can be used to form any number of towers.

forms the tower in 2nd dimension say.

Feed Forward Neural Network



⇒ The input to the network is an n-dimensional vector. $x \in \mathbb{R}^n$. The input layer is called the 0^{th} Layer or L_0.

⇒ The network contains L-1 hidden layers each having n_i neurons. $n_0 = n$ (input). i represents the layer index.

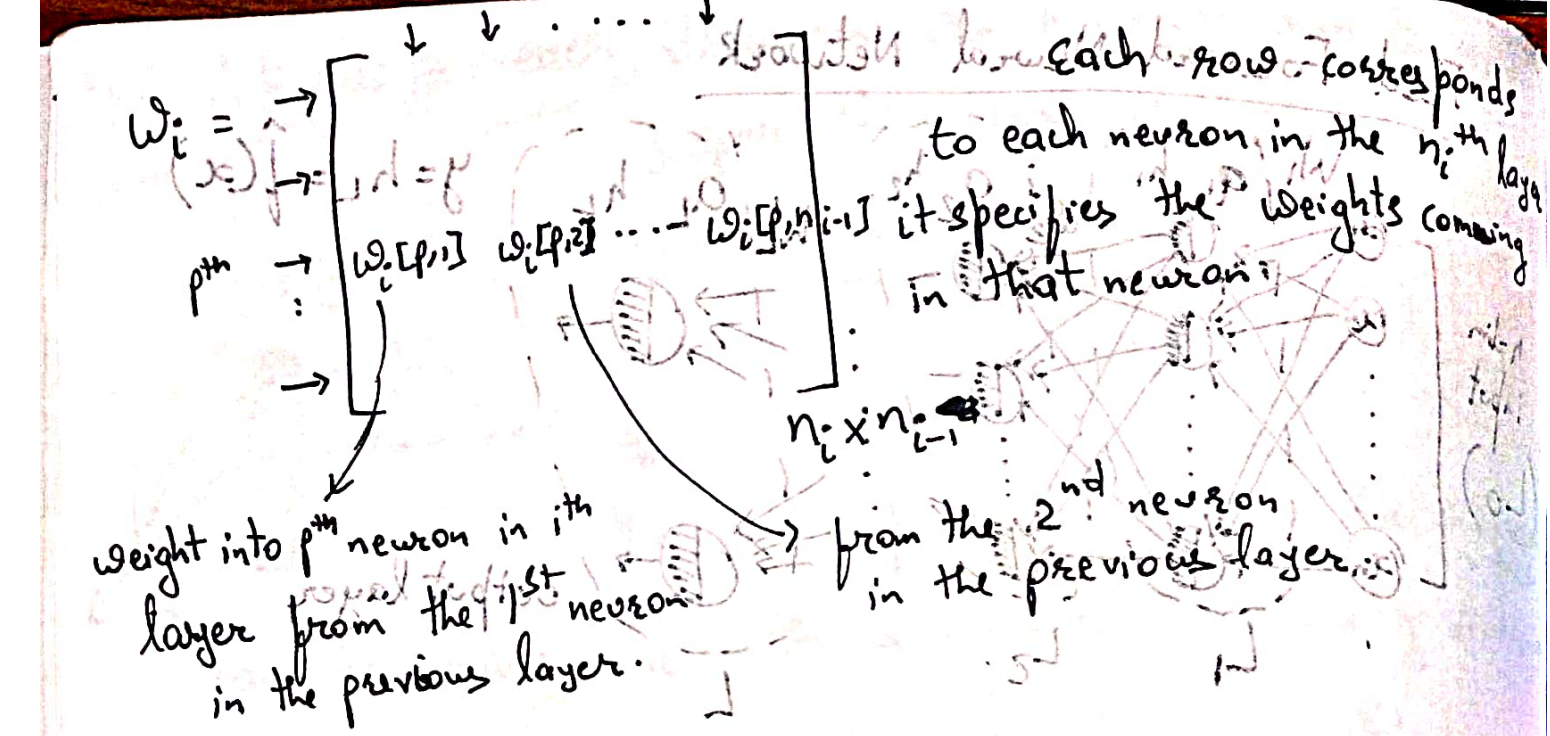
⇒ Each neuron in the hidden layer and output layer can be split into two parts: pre-activation (a_i) and activation (h_i).

⇒ a_i is an $\mathbb{R}^{n_i}$ dimensional vector such that the j^{th} component of a_i is the input given to the j^{th} neuron of the i^{th} layer L_i. It is equivalent to $w^T x + b$ for the j^{th} neuron.

⇒ h_i is an $\mathbb{R}^{n_i}$ dimensional vector such that the j^{th} component of h_i is the output produced by the j^{th} neuron of the i^{th} layer L_i. If neuron is sigmoid then it is equivalent to $\sigma(w^T x + b)$.

⇒ $W_i \in \mathbb{R}^{n_i \times n_{i-1}}$ is a matrix corresponding to layer i.

$W_i[p, q]$ is the weight given to the input from the q^{th} neuron in $L_{n_{i-1}}$ th layer to p^{th} neuron in L_{n_i} th layer.



Each row corresponds to each neuron in the n_i^{th} layer it specifies the weights coming in that neuron.

⇒ The activation a_i of the i^{th} layer will be:

$$a_i = W_i \times h_{i-1} + b_i$$

W_i: $n_i \times n_{i-1}$, h_{i-1}: $n_{i-1} \times 1$, b_i: $n_i \times 1$

⇒ The activation of the i^{th} layer (h_i) is the element wise function of a_i, for example an element wise sigmoid:-

$$h_i = \vec{\sigma}(a_i) \quad \text{where } \vec{\sigma}: \mathbb{R}^{n_i} \to \mathbb{R}^{n_i}$$

such that $\vec{\sigma}(a_i)_j = \sigma(a_{ij})$

⇒ The activation at the output Layer is given by:

$$f(x) = h_L = O(a_L)$$ (also an element wise function)

Writing it as a function of x — (assume only 2 hidden Layers are there)

$$f(x) = O(a_3)$$
$$= O(W_3 h_2 + b_3)$$
$$= O(W_3 g(a_2) + b_3)$$

$$\hat{f}(x) = O(W_3 g(W_2 h_1 + b_2) + b_3)$$
$$= O(W_3 g(W_2 g(a_1) + b_2) + b_3)$$

$$\boxed{\hat{f}(x) = O(W_3 g(W_2 g(W_1 x + b_1) + b_2) + b_3)}$$

Loss Function :- $L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{n_L} (\hat{y}_{ij} - y_{ij})^2$

$\{W_1, W_2, \cdots, W_L\}$, $\{b_1, b_2, \cdots, b_L\}$ → parameters

N → number of input data points

n_L → number of outputs for each data point.

This is suitable for regression type problems.

Output Function for regression problems :- We would like to allow all real values as outputs so just a linear transformation can work nicely.

$h_L = O(a_L) = W_0 a_L + b_0$ (W_0 and b_0 are not the same as W_L and b_L)

For Multi Class Classification :- The true labels are $\{0, 1\}$ where only a single 1 is there in the output vector for 1 datapoint all other are zero. Can be thought of as a PMF with all its mass on the correct class. For modelling such pmf's the more appropriate output activation is the **softmax function**.

$$\hat{y}_j = O(a_L)_j = \frac{e^{a_{Lj}}}{\sum_{k=1}^{n_L} e^{a_{Lk}}}$$ → the j^{th} component of an output vector for 1 data point. $\hat{y}_j$ will $\in (0,1)$

In this case, the more appropriate loss function is the cross-entropy loss:

$$L(\theta) = -\sum_{j=1}^{n_L} y_j \log(\hat{y}_j) \quad \text{(Loss for 1 data point)}$$

$$\Rightarrow L(\theta) = -\log(\hat{y}_t) \quad \text{where } y_t = 1 \text{ (true label)}$$

and all other $y_j = 0$

$\hat{y}_j$ represents the probability of that data point that it belongs to the j^{th} class.

Gradient Computations (Cross entropy loss and softmax output neurons)

Let for some data point, the correct class is t^{th} class. Then the output vector for that point will be $e_t = [0\ 0\ \dots\ \underset{t^{th}\text{ position}}{1}\ \dots\ 0]$

The loss corresponding to this point is:

$$L(\theta) = -\log(\hat{y}_t)$$

Let's calculate $\nabla_{h_L} L(\theta)$, gradient with respect to the output layer

$$\nabla_{h_L} L(\theta) = \nabla_{\hat{y}} L(\theta)$$

Let us start with $\frac{\partial L(\theta)}{\partial \hat{y}_j}$, the j^{th} element of $\nabla_{\hat{y}} L(\theta)$.

$$\Rightarrow \frac{\partial L(\theta)}{\partial \hat{y}_j} = \frac{\partial(-\log(\hat{y}_t))}{\partial \hat{y}_j} = \mathbb{1}(t=j) \times \frac{-1}{\hat{y}_t}$$

$$= \begin{cases} -\frac{1}{\hat{y}_t} & \text{if } j = t \\ 0 & \text{if } j \neq t \end{cases}$$

So $\nabla_{\hat{y}} L(\theta) = \begin{bmatrix} \frac{\partial L(\theta)}{\partial \hat{y}_1} \\ \frac{\partial L(\theta)}{\partial \hat{y}_2} \\ \vdots \\ \frac{\partial L(\theta)}{\partial \hat{y}_{n_L}} \end{bmatrix} = \frac{-1}{\hat{y}_t} \begin{bmatrix} \mathbb{1}(1=t) \\ \mathbb{1}(2=t) \\ \vdots \\ \mathbb{1}(n_L=t) \end{bmatrix} = \frac{-1}{\hat{y}_t} \times e_t$

$\therefore \boxed{\nabla_{\hat{y}} L(\theta) = \frac{-1}{\hat{y}_t} \times e_t}$

⇒ Now Let's calculate the gradient based on the the pre-activation layer of the output. $\nabla_{a_L} L(\theta)$.

For that we know that :- $\hat{y}_j = \frac{e^{a_{Lj}}}{\sum_{k=1}^{n_L} e^{a_{Lk}}}$ } softmax.

$L(\theta) = -\log(\hat{y}_t)$

$\frac{\partial L(\theta)}{\partial a_{Lj}} = \frac{\partial(-\log(\hat{y}_t))}{\partial a_{Lj}} = \frac{\partial(-\log(\hat{y}_t))}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial a_{Lj}}$

$\Downarrow$

$\frac{\partial L(\theta)}{\partial \hat{y}_t} \times \frac{\partial \hat{y}_t}{\partial a_{Lj}}$

$= -\frac{1}{\hat{y}_t} \times \frac{\partial \left(\frac{e^{a_{Lt}}}{\sum_{k=1}^{n_L} e^{a_{Lk}}} \right)}{\partial a_{Lj}}$

$= -\frac{1}{\hat{y}_t} \times \left(\frac{\sum_{k=1}^{n_L} e^{a_{Lk}} \times \frac{\partial(e^{a_{Lt}})}{\partial a_{Lj}} - e^{a_{Lt}} \frac{\partial(\sum_{k=1}^{n_L} e^{a_{Lk}})}{\partial a_{Lj}}}{\left(\sum_{k=1}^{n_L} e^{a_{Lk}} \right)^2} \right)$

$$= -\frac{1}{\hat{y}_t} \times \left(\frac{e^{a_{Lt}} \left(\sum_{k=1}^{n_L} e^{a_{Lk}} \right) \mathbb{1}(t=j) - e^{a_{Lt}} e^{a_{Lj}}}{\left(\sum_{k=1}^{n_L} e^{a_{Lk}} \right)^2} \right)$$

$$= -\frac{1}{\hat{y}_t} \left(\frac{\mathbb{1}(t=j) e^{a_{Lt}}}{\sum_{k=1}^{n_L} e^{a_{Lk}}} - \frac{e^{a_{Lt}} e^{a_{Lj}}}{\sum_{k=1}^{n_L} e^{a_{Lk}} \times \sum_{k=1}^{n_L} e^{a_{Lk}}} \right)$$

$$= -\frac{1}{\hat{y}_t} \left(\mathbb{1}(t=j) \hat{y}_t - \hat{y}_t \hat{y}_j \right)$$

$$= -\left(\mathbb{1}(t=j) - \hat{y}_j \right)$$

$$\Rightarrow \frac{\partial L(\theta)}{\partial a_{Lj}} = -\left(\mathbb{1}(t=j) - \hat{y}_j \right)$$

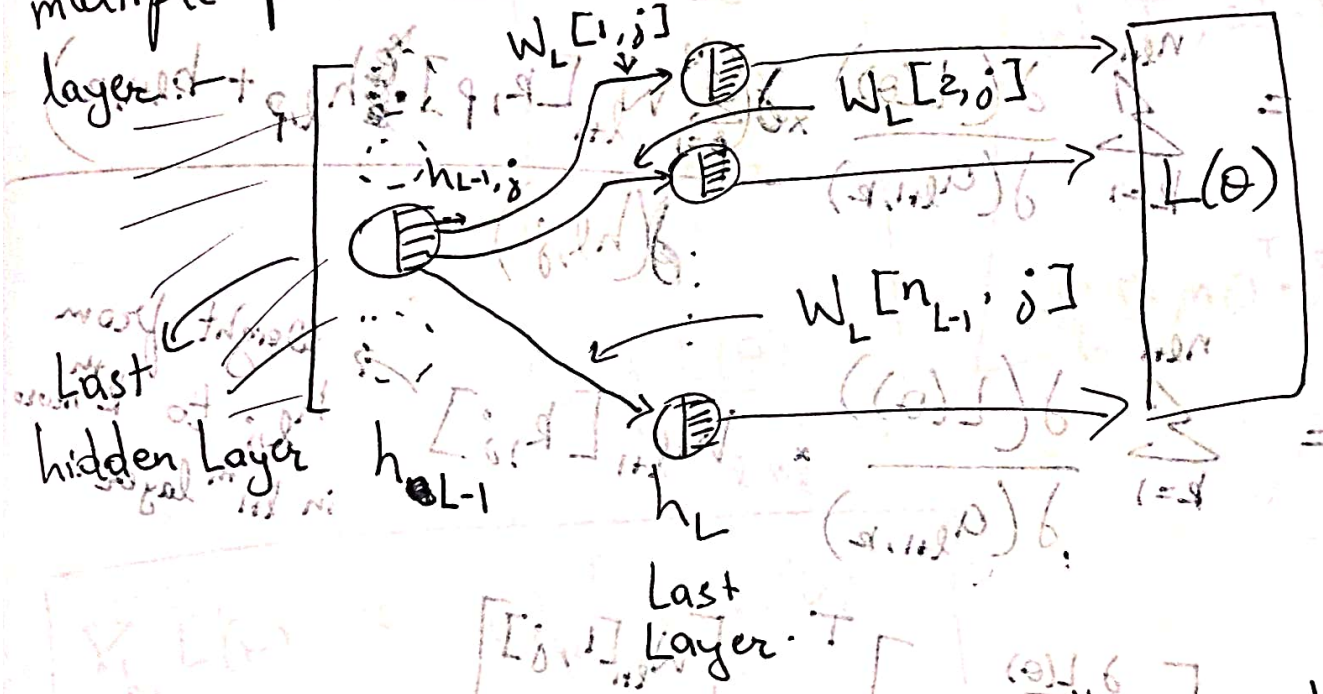
Therefore:- $\nabla_{a_L} L(\theta) = -$

$$\begin{bmatrix} \mathbb{1}(t=1) - \hat{y}_1 \\ \mathbb{1}(t=2) - \hat{y}_2 \\ \vdots \\ \mathbb{1}(t=n_L) - \hat{y}_{n_L} \end{bmatrix}$$

$$\boxed{\nabla_{a_L} L(\theta) = -(e_t - \hat{y})}$$

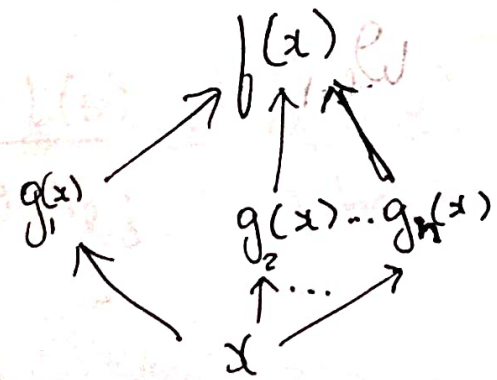
We now have the formulas for gradients of the output layer. This is straightforward as the output layer directly goes into the Loss function.

This is not true however, for any other layer as the output of hidden layer affects the loss function through multiple paths. Let us take a neuron in the last hidden layer:



We can see that output of $h_{L-1,j}$ j^{th} neuron of the last hidden layer affects the path through L paths.

⇒ Hence we use chain rule for multiple path to calculate $\frac{\partial L(\theta)}{\partial h_{lj}}$ l is for the layer number and j is for the neuron number.



Let $f(x)$ be a function of x such that it can be separated as $g_1(x), g_2(x)$ and $\dots g_n(x)$. Then

$$\frac{\partial f(x)}{\partial x} = \sum_{i=1}^{n} \frac{\partial f(x)}{\partial g_i(x)} \times \frac{\partial g_i(x)}{\partial x}$$

In our case $f(x) = L(\theta)$, $g_i(x) = a_{l+1,k}$

$x = h_{l,j}$

k^{th} pre-activation of the next layer!

$$\frac{\partial L(\theta)}{\partial h_{l,j}} = \sum_{k=1}^{n_{l+1}} \frac{\partial (L(\theta))}{\partial (a_{l+1,k})} \frac{\partial (a_{l+1,k})}{\partial (h_{l,j})}$$

$$= \sum_{k=1}^{n_{l+1}} \frac{\partial (L(\theta))}{\partial (a_{l+1,k})} \times \frac{\partial \left(\sum_{p=1}^{n_l} W_{l+1}[k,p] \cdot h_{l,p} + b_{l+1,k} \right)}{\partial (h_{l,j})}$$

$$= \sum_{k=1}^{n_{l+1}} \frac{\partial (L(\theta))}{\partial (a_{l+1,k})} \times W_{l+1}[k,j]$$

$W_{l+1}[k,j]$ → weight from $h_{l,j}$ to k^{th} neuron in $l+1^{th}$ layer

$$= \begin{bmatrix} \frac{\partial L(\theta)}{\partial a_{l+1,1}} \\ \frac{\partial L(\theta)}{\partial a_{l+1,2}} \\ \vdots \\ \frac{\partial L(\theta)}{\partial a_{l+1,n_{l+1}}} \end{bmatrix}^T \cdot \begin{bmatrix} W_{l+1}[1,j] \\ W_{l+1}[2,j] \\ \vdots \\ W_{l+1}[n_{l+1},j] \end{bmatrix}$$

j^{th} column of W_{l+1}

$\nabla_{a_{l+1}} L(\theta)$

$$\Rightarrow \frac{\partial L(\theta)}{\partial h_{l,j}} = W_{l+1}[\bullet, j]^T \cdot \nabla_{a_{l+1}} L(\theta)$$

j^{th} column

$$\nabla_{h_l} L(\theta) = \begin{bmatrix} \frac{\partial L(\theta)}{\partial h_{l,1}} \\ \frac{\partial L(\theta)}{\partial h_{l,2}} \\ \vdots \\ \frac{\partial L(\theta)}{\partial h_{l,n_l}} \end{bmatrix} = \begin{bmatrix} W_{l+1}[\bullet, 1]^T \nabla_{a_{l+1}} L(\theta) \\ W_{l+1}[\bullet, 2]^T \cdot \nabla_{a_{l+1}} L(\theta) \\ \vdots \\ W_{l+1}[\bullet, n_l]^T \cdot \nabla_{a_{l+1}} L(\theta) \end{bmatrix}$$

$$\boxed{\nabla_{h_l} L(\theta) = W_{l+1}^T \cdot \nabla_{a_{l+1}} L(\theta)}$$

$\Rightarrow$ Now let us look at the $\nabla_{a_l} L(\theta)$. We know that

$h_l = g(a_l)$

$h_{l,j} = g(a_{l,j})$ (Element wise function)

'g' is the activation function

$$\frac{\partial L(\theta)}{\partial a_{l,j}} = \frac{\partial L(\theta)}{\partial h_{l,j}} \frac{\partial (h_{l,j})}{\partial a_{l,j}} = \frac{\partial (L(\theta))}{\partial (h_{l,j})} g'(a_{l,j})$$

$$\nabla_{a_l} L(\theta) = \begin{bmatrix} \frac{\partial (L(\theta))}{\partial (a_{l,1})} \\ \frac{\partial (L(\theta))}{\partial (a_{l,2})} \\ \vdots \\ \frac{\partial (L(\theta))}{\partial (a_{l,n_l})} \end{bmatrix} = \begin{bmatrix} \frac{\partial L(\theta)}{\partial h_{l,1}} \times g'(a_{l,1}) \\ \frac{\partial L(\theta)}{\partial h_{l,2}} \times g'(a_{l,2}) \\ \vdots \\ \frac{\partial L(\theta)}{\partial h_{l,n_l}} \times g'(a_{l,n_l}) \end{bmatrix}$$

$$\nabla_{a_l} L(\theta) = \nabla_{h_l} L(\theta) \odot [\dots, g'(a_{l,j}) \dots]$$

Element wise multiplication of $\nabla_{h_l} L(\theta)$ and $g'(a_l)$ also called as Hadamard Product

Chronology of Gradient Computation

Calculate:- 1) $\nabla_{h_L} L(\theta)$ 2) $\nabla_{a_L} L(\theta)$

Would require

3) $\nabla_{h_{L-1}} L(\theta)$ ← 4) $\nabla_{a_{L-1}} L(\theta)$

5) $\nabla_{h_{L-2}} L(\theta)$ ← 6) $\nabla_{a_{L-2}} L(\theta)$

$\nabla_{h_1} L(\theta)$ ← $\nabla_{a_1} L(\theta)$

⇒ Computing gradients for the parameters. $\nabla_{W_k} L(\theta)_{n_k \times n_{k-1}}$ and $\nabla_{b_k} L(\theta)_{n_k \times 1}$

We know that :-

$$a_l = W_l h_{l-1} + b_l$$

$$\frac{\partial L(\theta)}{\partial W_l[j,k]}$$

$W_l[j,k]$

$$\nabla_{W_l} L(\theta) = \begin{bmatrix} \frac{\partial L(\theta)}{\partial W_l[1,1]} & \frac{\partial L(\theta)}{\partial W_l[1,2]} & \cdots & \frac{\partial L(\theta)}{\partial W_l[1,n_{l-1}]} \\ \frac{\partial L(\theta)}{\partial W_l[2,1]} & \frac{\partial L(\theta)}{\partial W_l[2,2]} & \cdots & \frac{\partial L(\theta)}{\partial W_l[2,n_{l-1}]} \\ \vdots & & & \\ \frac{\partial L(\theta)}{\partial W_l[n_l,1]} & \frac{\partial L(\theta)}{\partial W_l[n_l,2]} & \cdots & \frac{\partial L(\theta)}{\partial W_l[n_l,n_{l-1}]} \end{bmatrix}$$

Note that, only comes in the computation of $a_{l,j}$. Hence $W_l[j,k]$ affects $L(\theta)$ only through 1 path.

Hence:- $$\frac{\partial L(\theta)}{\partial W_l[j,k]} = \frac{\partial L(\theta)}{\partial a_{l,j}} \times \frac{\partial a_{l,j}}{\partial W_l[j,k]}$$

$$= \frac{\partial L(\theta)}{\partial a_{l,j}} \times \frac{\partial \left(\sum_{p=1}^{n_{l-1}} W_l[j,p] \cdot h_{l-1,p} + b_{l,j} \right)}{\partial W_l[j,k]}$$

Weight to j^{th} neuron from k^{th} neuron

$$\frac{\partial L(\theta)}{\partial W_l[j,k]} = \frac{\partial L(\theta)}{\partial a_{l,j}} \times h_{l-1,k}$$

→ derivative w.r.t. j^{th} pre-act

→ output from k^{th} neuron.

$$\begin{bmatrix} \frac{\partial L(\theta)}{\partial a_{l,1}} h_{l-1,1} & \frac{\partial L(\theta)}{\partial a_{l,1}} h_{l-1,2} & \cdots & \frac{\partial L(\theta)}{\partial a_{l,1}} h_{l-1,n_{l-1}} \\ \frac{\partial L(\theta)}{\partial a_{l,2}} h_{l-1,1} & \frac{\partial L(\theta)}{\partial a_{l,2}} h_{l-1,2} & \cdots & \frac{\partial L(\theta)}{\partial a_{l,2}} h_{l-1,n_{l-1}} \\ \vdots & & & \\ \frac{\partial L(\theta)}{\partial a_{l,n_l}} h_{l-1,1} & \frac{\partial L(\theta)}{\partial a_{l,n_l}} h_{l-1,2} & \cdots & \frac{\partial L(\theta)}{\partial a_{l,n_l}} h_{l-1,n_{l-1}} \end{bmatrix}$$

⇒ $\nabla_{W_l} L(\theta)$ =

*

*

$$\Rightarrow \nabla_{W_\ell} L(\theta) = \begin{bmatrix} \frac{\partial L(\theta)}{\partial a_{\ell,1}} \\ \frac{\partial L(\theta)}{\partial a_{\ell,2}} \\ \vdots \\ \frac{\partial L(\theta)}{\partial a_{\ell,n_\ell}} \end{bmatrix} \cdot \begin{bmatrix} h_{\ell-1,1} & h_{\ell-1,2} & \cdots & h_{\ell-1,n_{\ell-1}} \end{bmatrix}$$

$$\boxed{\nabla_{W_\ell} L(\theta) = \nabla_{a_\ell} L(\theta) \cdot h_{\ell-1}^T}$$

Similar argument as $\frac{\partial L(\theta)}{\partial W_\ell[j,k]}$

For $\nabla_{b_\ell} L(\theta)$: notice that $\frac{\partial L(\theta)}{\partial b_{\ell,i}} = \frac{\partial L(\theta)}{\partial a_{\ell,i}} \times \frac{\partial a_{\ell,i}}{\partial b_{\ell,i}}$

$$= \frac{\partial L(\theta)}{\partial a_{\ell,i}} \times 1$$

$$\text{so } \boxed{\nabla_{b_\ell} L(\theta) = \nabla_{a_\ell} L(\theta)}$$

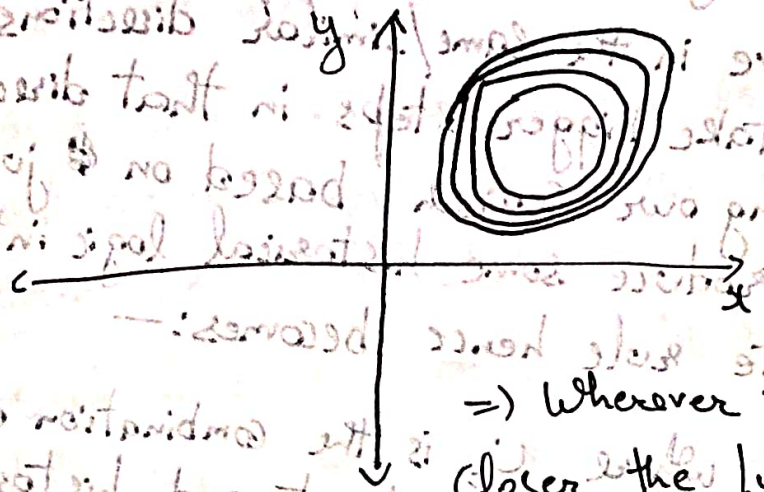
$$\Rightarrow \sigma(x) = \frac{1}{1+e^{-x}} \qquad \sigma'(x) = \sigma(x)(1-\sigma(x))$$

$$\Rightarrow \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \qquad \tanh'(x) = 1 - (\tanh(x))^2$$

$$= \nabla_{W_\ell} L(\theta)$$

Contours

Suppose there is a function from $\mathbb{R}^2 \to \mathbb{R}$, $z = f(x,y)$, then the contour map may look like:-



Each contour rings states the (x,y) values where the $f(x,y)$ is the same.

=> Wherever the contour rings are closer the function has a steep curvature

=> Wherever the contour rings are further apart, the function has a gentle slope/curvature.

The logic is that in areas of steep curvature, a small change in x and y values may bring a larger change in $f(x,y)$. Similarly in areas of gentle slopes a large change is required to bring the same amount of change in $f(x,y)$.

Δz change is achieved in small changes in x and y, steep curvature, contour rings are closer.

Δz change is achieved in larger changes in x and y, gentle slope, contour rings are further apart.

=> When gradient descent is applied, it makes tiny progress in areas where contour rings are far apart as the slope is small. It makes larger steps in areas where contour rings are closer as the slope is larger.

Momentum Based Gradient Descent

The small progress made in the flat areas causes the gradient descent to converge slowly. But if we are repeatedly asked to move in the same/similar directions, then it makes sense to take bigger steps in that direction. So instead of just guiding our search based on just the gradient, we introduce some historical logic into the search. So the update rule hence becomes:—

$$w_{t+1} = w_t - \eta v_t$$

where v_t is the combination of the current gradient and historical directions.

where,

$$v_t = \nabla w_t + \beta v_{t-1}$$

we start with w_0 and $v_0 = \nabla w_0$, $0 \le \beta < 1$.

Suppose we want w_3, then:— $w_3 = w_2 - \eta v_2$

$$v_2 = \nabla w_2 + \beta v_1$$

$$v_1 = \nabla w_1 + \beta v_0 = \nabla w_1 + \beta \nabla w_0$$

$$\therefore w_3 = w_2 - \eta(\nabla w_2 + \beta(\nabla w_1 + \beta \nabla w_0))$$

$$= w_2 - \eta(\nabla w_2 + \beta \nabla w_1 + \beta^2 \nabla w_0)$$

$$= w_2 - \eta \sum_{i=0}^{2} \beta^{2-i} \nabla w_i$$

$$\therefore w_{t+1} = w_t - \eta \underbrace{\sum_{i=0}^{t} \beta^{t-i} \nabla w_i}_{v_t}$$

you give the most importance to ∇w_2 which is the current gradient, then ∇w_1, and then the least importance to ∇w_0.

So if we are making small steps in some direction, after some steps the "momentum" (v_t) will grow and we will start making bigger steps.

The problem with this strategy is that sometimes the momentum is too large and we overshoot unneccessarily in some directions. But it certainly converges faster than normal G.D. in most situations.

To deal with the overshooting problem of Momentum based gradient descent we look at the Nesterov Accelerated Gradient Descent (NAG). Instead of directly combining the historical direction v_{t-1} with the current gradient to calculate v_t we check what would happen if ∇w_t was not there. Then v_t will be βv_{t-1} :- $v_t = \beta v_{t-1}$

So $w'_{t+1} = w_t - \eta v_t = w_t - \eta \beta v_{t-1}$

if we ignore the step size η, then w'_{t+1} will be $w_t - \beta v_{t-1}$

Let us calculate the gradient w.r.t this w'_{t+1}

$\nabla w'_{t+1}$, now let's look at the 2 cases :-

1) Not overshooting :- if we don't overshoot, then $\nabla w'_{t+1}$ will be in line with the previous directions, so if v_t is calculated like this :-

$v_t = \nabla(w'_{t+1}) + \beta v_{t-1}$ (it would add on the momentum since both $\nabla w'_{t+1}$ and βv_{t-1} are similar directions).

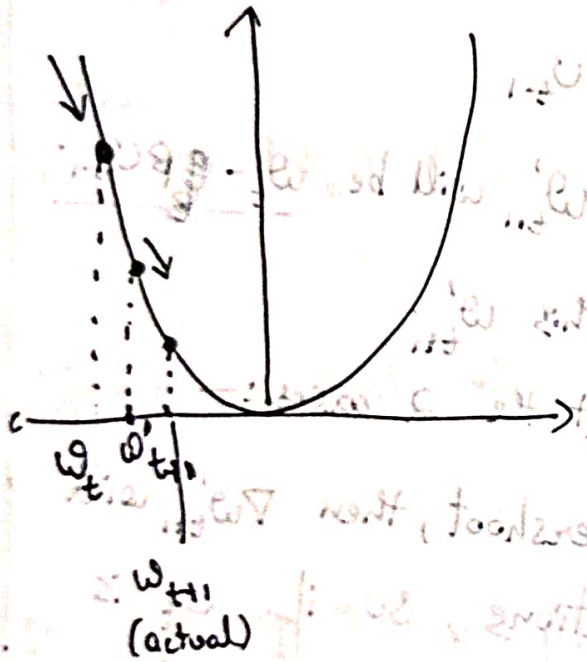
actual $w_{t+1} = w_t - \eta v_t$

2) Overshoot : if we overshoot, then $\nabla w'_{t+1}$ will be in opposite like direction with the previous directions, so if v_t is calculated like this :-

$v_t = \nabla w'_{t+1} + \beta v_{t-1}$, it would reduce the momentum affect as $\nabla w'_{t+1}$ and βv_{t-1} are in opposite like directions. Hence this will prevent/reduce overshooting as it would reduce the momentum affect.

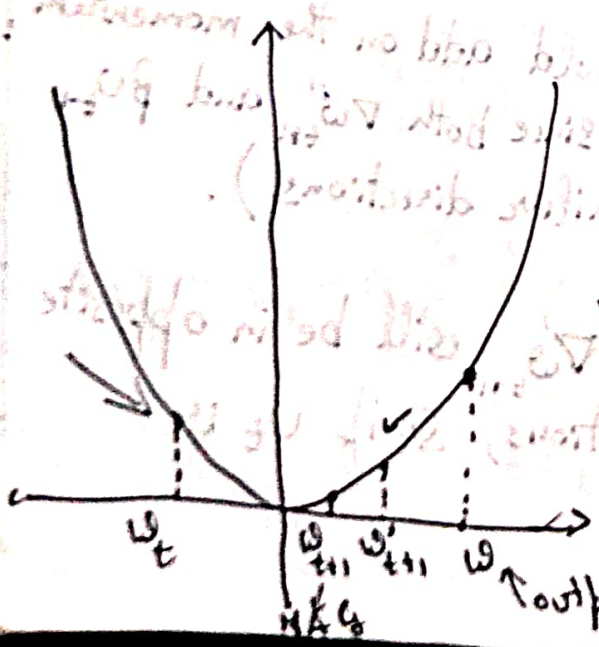
In both cases:-

$$v_t = \nabla(w_t - \beta v_{t-1}) + \beta v_{t-1}$$
$$w_{t+1} = w_t - \eta v_t$$



$w'_{t+1} = w_t - \beta v_{t-1}$ looking ahead to w'_{t+1}

In this case, suggests that we should go along with the momentum and leads us to w_{t+1}



In this case looking ahead to w'_{t+1} suggests that we will overshoot and it cancels some momentum to reduce overshoot and leads us to w_{t+1}

w would have been the output for normal Momentum Based GD. (Overshooting is high)

so NAG oscillates lesser than MGD.

One more way to speed up convergence is to set the correct learning rate η. Tips for Annealing the learning rate.

1) Halve the learning rate after every 5 epochs.

2) Halve the learning rate after an epoch if the validation error is more than what it was at the end of the previous epoch. So if the validation error is more at the end of current epoch, then reinitialize the weights to what was learned after previous epoch and run the current epoch again with $1/2$ the learning rate.

3) Exponential Decay :- $\eta = \eta_0 e^{-kt}$ where η_0 and k are hyperparameters and t is the step number. η_0 or $k \uparrow$ then the decay is more rapid.

4) $1/t$ decay, $\eta = \frac{\eta_0}{1 + kt}$

5) For Momentum algorithms

$$\beta_t = \min\left(1 - 2^{-1-\log_2\left(\left\lfloor \frac{t}{250} \right\rfloor + 1\right)}, \beta_{max}\right), \text{ where}$$

β_{max} is chosen from $\{0.999, 0.995, 0.99, 0.9, 0\}$

6) Line Search try many values of η for each step, choose the one which causes the least loss.

All the methods discussed above change the learning rate for the whole gradient. This means irrespective of the features, the learning rate remains the same for all. But suppose we take the gradient

for the whole dataset and some of the features were important but very sparse. In this case the gradient update will be $w_{t+1} = w_t - \eta \nabla w_t$.

in the gradient

Naturally for sparse features, the corresponding element will be very small. Hence the updates made for these sparse features will be small as compared to other dense features. At the end of the algorithm (exhausting the iteration limit) the w^* for the sparse features may not be the optimal value as the updates were small in the iteration. Can we adapt the learning rate for each feature based on their frequency? (sparse features having a relatively higher learning rate).

AdaGrad :- Decay the learning rate for each parameter in proportion to their update history (more updates means more decay).

(larger magnitude)

Update Rule for AdaGrad :- $V_t = V_{t-1} + (\nabla w_t)^2$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{V_t + \epsilon}} * \nabla w_t$$

↑ scaler, as it refers to gradient w.r.t each parameter

$V_t = \sum_{k=0}^{t} (\nabla w_k)^2$

1) For dense features, V_t grows much faster as ∇w is large, so the decay in the learning rate is much faster.

2) For sparse features, V_t grows slowly as ∇w is small, so the decay in the learning rate is slow.

⇒ Adagrad slows down near the minimum due to decaying learning rate. This is because after some iterations, V_t is large and decay is too much, especially for dense features. Moreover, near the minima, the gradients are also small, which when combined with large accumulated V_t slows down the algorithm. V_t keeps on increasing.

⇒ To avoid this problem, we can use an algorithm that decays the denominator V_t in such a way that in the steep areas, V_t is large but in gentle areas V_t decreases.

(effective learning is small) (effective learning is large)

This is Done in RMS Prop algorithm

$$V_t = \beta V_{t-1} + (1-\beta)(\nabla w_t)^2 \qquad 0 \le \beta \le 1, \text{ typically } 0.9, 0.99.$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{V_t + \epsilon}} \nabla w_t$$

Let us expand V_t ($\beta = 0.9$)

$$V_3 = 0.9 V_2 + 0.1 \nabla w_3^2$$

$$V_3 = 0.9(0.9 V_1 + 0.1 \nabla w_2^2) + 0.1 \nabla w_3^2$$

$$V_1 = 0.9 V_0 + 0.1 \nabla w_1^2$$

$$V_0 = 0.1 \nabla w_0^2$$

$$V_1 = 0.1 \times 0.9 \nabla w_0^2 + 0.1 \nabla w_1^2$$

$$\Rightarrow V_3 = 0.9\left(0.9\left(0.1 \times 0.9 \nabla w_0^2 + 0.1 \nabla w_1^2\right) + 0.1 \nabla w_2^2\right) + 0.1 \nabla w_3^2$$

$$V_3 = 0.9\left(0.1 \times 0.9^2 \nabla w_0^2 + 0.1 \times 0.9 \nabla w_1^2 + 0.1 \nabla w_2^2\right) + 0.1 \nabla w_3^2$$

$$= 0.1 \times 0.9^3 \nabla w_0^2 + 0.1 \times 0.9^2 \nabla w_1^2 + 0.1 \times 0.9 \nabla w_2^2 + 0.1 \nabla w_3^2$$

$$= 0.1 \sum_{i=0}^{3} (0.9)^{3-i} \nabla w_i^2$$

$\therefore v_t = (1-\beta)\sum_{i=0}^{t} \beta^{(t-i)} (\nabla w_i)^2$, we see that as t increases, the contribution from the early gradients decays to 0.

In steep areas ∇w's are large hence v_t also becomes large and the learning rate reduces.

In gentle areas, ∇w's are small and v_t decays hence the learning rate increases.

So RMSprop retains the advantages of AdaGrad for sparse and dense features but also has better convergence behaviour because the learning rate can increase or decrease.

One problem RMSprop faces is that it can oscillate around the solution. Near the minima ∇w_i are small hence making v_t decay to 0. (0)

This makes the effective learning rate constant $\frac{\eta}{\sqrt{\epsilon}}$, which is susceptible to oscillation. So RMSprop depends on the initialization of these parameters.

⇒ To reduce this effect we can adapt the numerator as well this allows us to avoid the dependency on the initializations. This algorithm is called AdaDelta. The algorithm is as follows:-

$$v_t = \beta v_{t-1} + (1-\beta)(\nabla w_t)^2$$

$$\Delta w_t = -\sqrt{\frac{u_{t-1}+\epsilon}{v_t+\epsilon}}\,\nabla w_t$$

$$w_{t+1} = w_t + \Delta w_t$$

$$u_t = \beta u_{t-1} + (1-\beta)(\Delta w_t)^2$$

Notice that Δw_t depends on current v_t and previous u_{t-1}.

1) In steep areas v_t is more updated and increases. but u_{t-1} is not yet updated, so in Δw_t numerator is smaller than the denominator, hence the learning rate is smaller.

$u_t < v_t$

2) In flat regions, v_t is more update and reduces, but v_{t-1} is not yet updated, so in Δw_t, the denominator is small, the learning rate becomes larger. ($v_t > v_t$)

⇒ AdaDelta adapts the learning rate more smoothly than RMSProp algorithm.

Adam (Adaptive Moments): Very similar to the RMSProp algorithm, but introduces momentum in the calculation. So instead of directly using ∇w_t in the gradient update, we update it using the momentum term itself.

$$m_t = \beta_1 m_{t-1} + (1-\beta_2)\nabla w_t$$

(β_1^t - β_1 raised power t)

Scaling/removing bias from m_t (explained later): $\hat{m}_t = \frac{m_t}{(1-\beta_1^t)}$

$$v_t = \beta_2 v_{t-1} + (1-\beta_1)(\nabla w_t)^2, \text{ then correcting the bias: } \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

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

In this momentum, we give more weightage to overall previous gradients in the form of $\beta_1 m_{t-1}$, the recent gradient is given slightly less weightage since it gets multiplied by $(1-\beta_1)$. It is like a moving exponential average of ∇w_t's. We can't calculate $E[\nabla w]$, as it will be expensive, but what about

$$E[m_t] = E\left[(1-\beta_1)\sum_{i=0}^{t} \beta_1^{(t-i)} \nabla w_i\right]$$

$$= (1-\beta_1)\sum_{i=0}^{t} \beta_1^{(t-i)} E[\nabla w_i]$$

Since $E[\nabla w_i] = E[\nabla w]$

$$\Rightarrow E[m_t] = (1-\beta_1) \sum_{i=0}^{t} \beta^{t-i} \times E[\nabla w]$$

$$= (1-\beta_1) E[\nabla w] \sum_{i=0}^{t} \beta^{(t-1)}$$

$$= (1-\beta_1) E[\nabla w] \times \frac{(1-\beta_1^t)}{(1-\beta_1)}$$

$E\left[\frac{m_t}{1-\beta_1^t}\right] = E[\nabla w]$. therefore $\frac{m_t}{1-\beta_1^t}$ gives an unbiased estimate of ∇w.

⇒ A lack of initialization bias correction would lead to initial steps that are much larger. Bias correction also ensures that initial learning rate is not very huge.

⇒ One more advantage of bias correction is that it reduces, the affect of noisy points taken in Mini batch GD for gradient computation.

Max Prop :- If we closely revist the formula for RMS Prop:-

$$V_t = \beta V_{t-1} + (1-\beta)(\nabla w_t)^2 \quad ; \quad w_{t+1} = w_t - \frac{\eta}{\sqrt{V_t + \epsilon}} \nabla w_t$$

Although it achieves the requirement of adjusting the learning rate based on the gradient. But is still aggressive and that makes it susceptible to noise. Take for example a sparse feature in a Mini Batch G.D and the gradient is considerable. Now a batch comes which was noisy and no data point had this feature, so gradient will be zero, in the current rule, the zero gradient which is noisy will affect V_t and reduce it and hence increase the learning rate. This is undesirable.

⇒ We change the way V_t is calculated to make it less susceptible to noise. That is done in the max prop, instead of using the

L^2 norm (squares) we can use the max norm. So the update formula becomes:-

$$v_t = \max\{\beta v_{t-1}, |\nabla w_t|\}$$

$$w_{t+1} = w_t - \frac{\eta}{v_t + \epsilon} \nabla w_t$$

↳ square root is removed as max norm is used now.

Thinking about the same situation, zero gradient will never be picked. But still the v_t decays as β is multiplied, so if the behaviour is consistent, v_t will decay and the original behaviour is still retained.

~~But in the case of~~

Introducing Max norm in Adam :- AdaMax

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)(\nabla w_t) \quad , \quad \hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$v_t = \max\{\beta_2 v_{t-1}, |\nabla w_t|\} \quad , \quad w_{t+1} = w_t - \frac{\eta}{v_t + \epsilon} \hat{m}_t$$

notice that $\hat{v}_t$ is not needed as max norm does not drastically increase the learning rate.

Introducing Nesterov Accerelation in Adam :- NADAM

Slightly different notation (did not understand)

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

$$\hat{m}_{t+1} = \frac{m_{t+1}}{1-\beta_1^{t+1}}$$

$$v_{t+1} = \beta_2 v_t + (1-\beta_2)(\nabla w_t)^2 \quad , \quad \hat{v}_{t+1} = \frac{v_{t+1}}{1-\beta_2^{t+1}}$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_{t+1}} + \epsilon}\left(\beta_1 \hat{m}_{t+1} + \frac{(1-\beta_1)\nabla w_t}{1-\beta_1^{t+1}}\right)$$

Learning rate Schemes

Based on epochs
1) Step Decay
2) Exponential Decay
3) Cyclical
4) Cosine Annealing

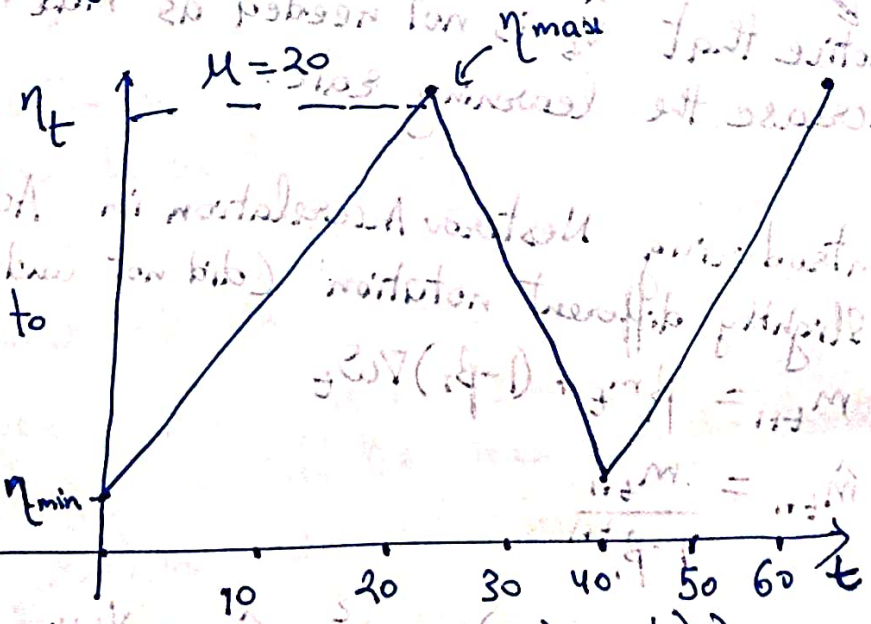
Base Validation
1) Line Search
2) Log Search

Based on Gradients
1) AdaGrad
2) RMSProp
3) AdaDelta
4) Adam
5) AdaMax
6) Nadam
7) AMSGrad
8) AdamW.

Cyclical Learning Rate :- At saddle points, the gradients become small, and it becomes difficult to move out from those areas. If we allow the learning rate to increase after some iterations irrespective of the gradient, then there is a chance to escape the saddle points. Adaptive learning rate helps in escaping saddle points but they are computationally expensive.
We can vary learning rate cyclically as a simple strategy.

i) Triangular :-

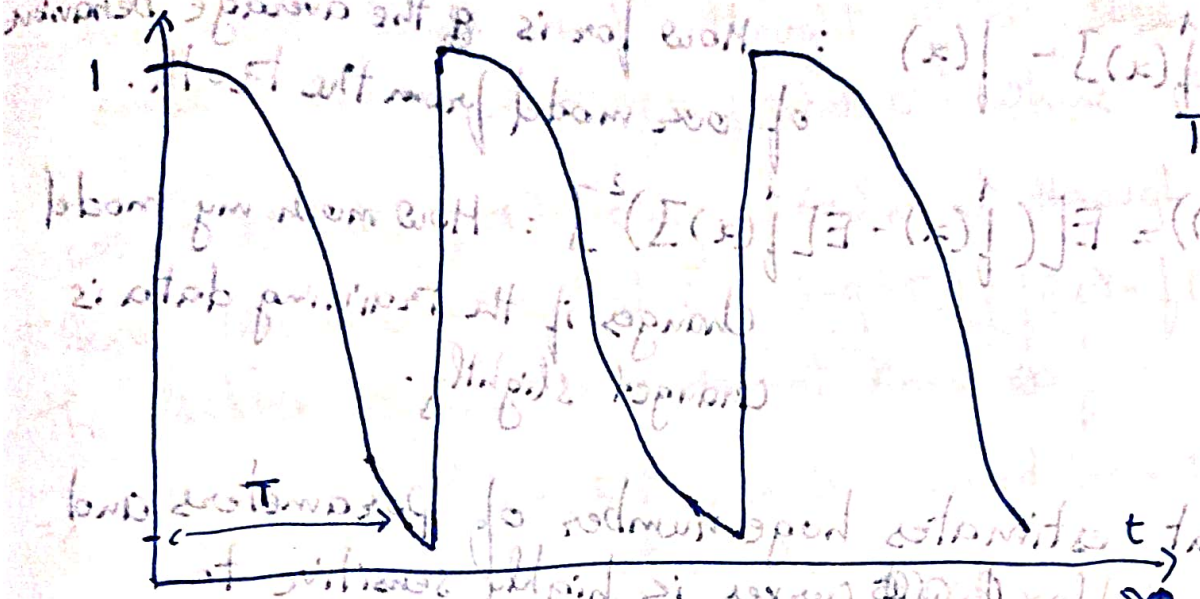
After every 20 steps, the learning rate reaches from minimum to maximum.



Formula is given by:-

$$\eta_t = \eta_{min} + (\eta_{max} - \eta_{min}) \cdot \max\left(0, \left(1 - \left|\frac{t}{U} - \left(2L1 + \frac{t}{20}\right) + 1\right|\right)\right)$$

ii) Cosine Annealing (Warm-restart)



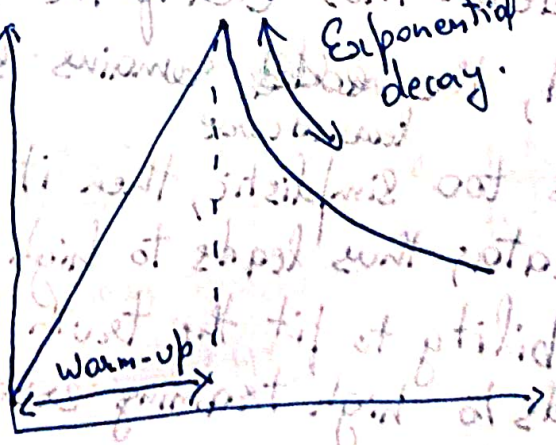
T is the restart interval.

$$\eta_t = \eta_{min} + \frac{\eta_{max} - \eta_{min}}{2}\left(1 + \cos\left(\pi \frac{t\%(T+1)}{T}\right)\right)$$

But due to abrupt increases in the learning rate, it may escape (overshoot) near the minima. In that case we could use techniques such as early stopping to roll back to the minimum.

Warm Start (found to be helpful in transformer architectures)

Typically, we set the initial learning rate to a high value and then decay it. On the contrary, using a low initial learning rate helps the model to warm and converge better. This is called warm start. This is because, initially the weights are random and we allow the model to make small steps in early gradient directions. Then after certain Number of iterations, the usual exponential decay strategy is applied. Number of warm-up rounds can be a choice.



Bias and Variance of a Model :-

Bias :- $E[\hat{f}(x)] - f(x)$: How far is the average behaviour of our model from the truth.

Variance $(\hat{f}(x)) = E[(\hat{f}(x) - E[\hat{f}(x)])^2]$: How much my model changes if the training data is changed slightly.

A model that estimates huge number of parameters and capable to fit complex curves, is highly sensitive to training data. If this model is trained on a slighly different sample of training data, it will significantly differ in its learned parameters. Thus it has high variance. Since this model can fit the training data, it leads to low error and hence low bias as it has the capability to fit the truth closely. But sometimes, if the data has noise, then this model gets severely affected because even though the training error will be small as always, but, on unseen data, it performs poorly. This is called overfitting. It still has low bias (usually) as the model is capable to fit the truth if we ignore the noise. So overfitting is a consequence of high variance and noise.

⇒ Similarly a model that estimates very few number of parameters. Thus even if the training data sample is slighly changed, the learned curve remains similar. Thus low variance. If the model is too simplistic, then it may fail to capture the structure in the data, thus leads to high bias as the model does not have the capability to fit the truth irrespective of training data. This leads to high training error and is called underfitting.

Estimating error From test data

the true labels can be thought of as combination of some function and some noise. So: $y = f(x) + \epsilon$ where $\epsilon \in N(0, \sigma^2)$

we estimate $f(x)$ using $\hat{f}(x)$ or $\hat{y}$ through our model. Hence we are interested in using $E[(\hat{f}(x) - f(x))^2]$. But the problem is that we do not know f.

$\Rightarrow E[(\hat{f}(x) - f(x))^2]$ can be derived from the following formula:

$$E[(\hat{y} - y)^2] = E[((\hat{f}(x) - f(x)) - \epsilon)^2]$$

$$= E[(\hat{f}(x) - f(x))^2] + E[\epsilon^2] - 2E[\epsilon(\hat{f}(x) - f(x))]$$

$$\Rightarrow E[(\hat{f}(x) - f(x))^2] = E[(\hat{y} - y)^2] + 2E[\epsilon(\hat{f}(x) - f(x))] - E[\epsilon^2]$$

$$= \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 - \sigma^2 + 2E[\epsilon(\hat{f}(x) - f(x))]$$

For 2 independent variables X and Y, and one having 0 expectation, $E[XY] = E[X] \times E[Y] = 0$ (i.e either $E[X]$ or $E[Y]$ is 0)

Since in test data, none of the points participated in estimating $\hat{f}(x)$, $(y - f(x))$ is independent from $(\hat{f}(x) - f(x))$. Hence. ϵ is independent from $(\hat{f}(x) - f(x))$.

So $E[(\hat{f}(x) - f(x))^2] = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 - \sigma^2$ ← depends on the noise.

If the data is sufficiently clean, then this is a good estimate of the true error (LHS).

If instead we used training data for estimation then ϵ is not independent from $\hat{f}(x) - f(x)$. Hence $E[\epsilon(\hat{f}(x) - f(x))] \neq 0$

the true error LHS is larger than the empirical error.

$$\frac{1}{n}\sum(\hat{y}_i - y_i)^2.$$

So using the empirical training error does not give a true representation of the actual error. But training is based on minimizing this error.

Stein's Lemma:- $\frac{1}{n}\sum \epsilon_i(\hat{f}(x_i) - f(x_i)) = \frac{\sigma^2}{n}\sum_{i=1}^{n} \frac{\partial \hat{f}(x_i)}{\partial y_i}$

If $\frac{\partial \hat{f}(x_i)}{\partial y_i}$ is large when there is high variance in the model, changing y_i brings large change in the learned model.

LHS of Steins Lemma is similar to $E[\epsilon(\hat{f}(x) - f(x))]$

So we can say that true error is a function of model complexity.

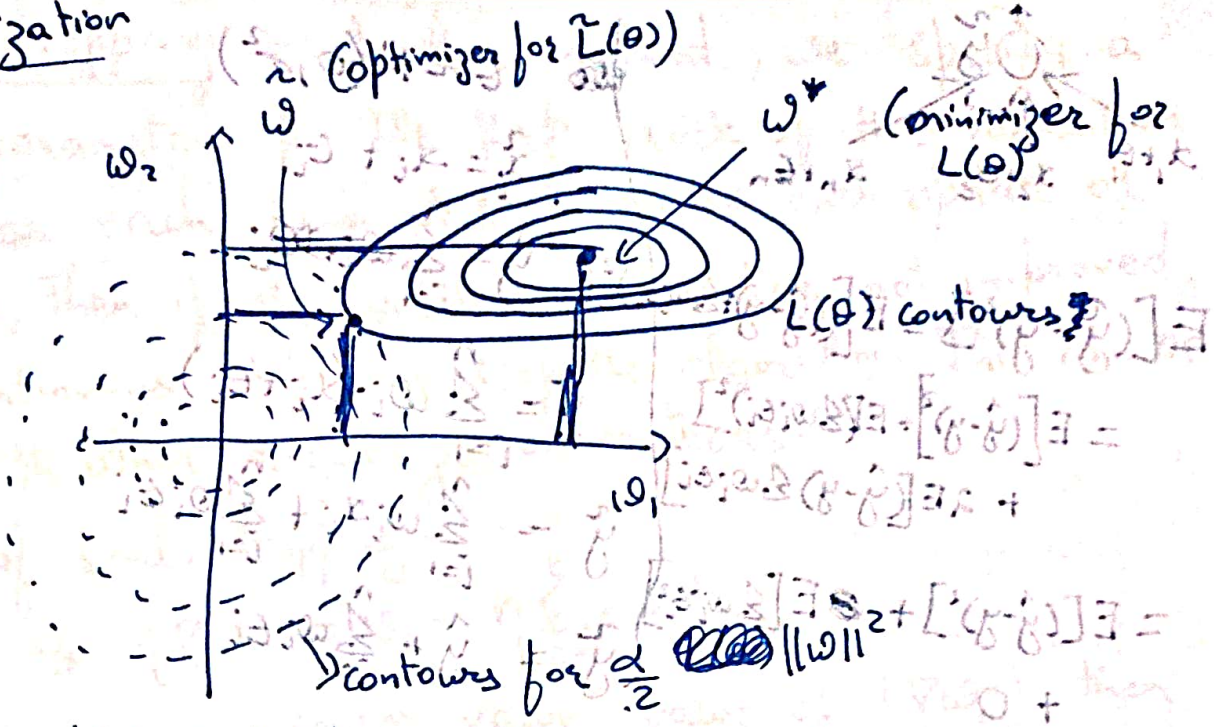
$$E[(\hat{f}(x) - f(x))] = \text{empirical train error} - \text{small constant} + \Omega(\text{model complexity})$$

The more complex the model, the true error may increase.

So it is better to minimize a modified objective function that also incorporates model complexity into it.

$\min \hat{L}(\theta)$ becomes $\left(L_{train}(\theta) + \Omega(\theta)\right)$. But instead of calculating $\Omega(\theta)$ as given by steins lemma we can calculate it using an approximation that captures model complexity. This strategy is called regularization.

i) L2 Regularization

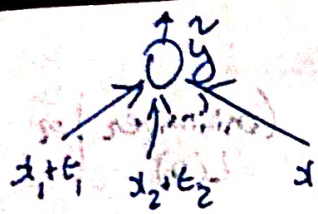


of Hessian of $L(\theta)$

the eigen value corresponding to w_1 will be smaller. Hence the ellipse is elongated in w_1 axis. Moreover, there is more shrinkage in w_1 as compared to w_2. This can also be inferred through the contours of $L(\theta)$, because in the axis of w_1, the contours are further apart, so derivative is small. So loss is less sensitive to w_1 than w_2. So w_1 is shriked more.

ii) Data Augmentation :- we can introduce new training data points by slightly perturbing the already present data points and then keeping their labels same. Example, shifting or rotating an image to create a new image.

iii) Injecting noise at inputs :- at each epoch, currupt some part of the input, thus it becomes difficult for the model to overfit on the training data as at each epoch the training data slightly changes. Can be thought of data augmentation at each epoch. For an input output network, without any hidden layers, the noise injection at input is equivalent to L2 regularization



$\epsilon_i \sim N(0, \sigma^2)$

$\hat{x}_i = x_i + \epsilon_i$

$\hat{y} = \sum_{i=1}^{n} \omega_i \hat{x}_i$

$\hat{y} = \sum_{i=1}^{n} \omega_i (x_i + \epsilon_i)$

$\hat{y} = \sum_{i=1}^{n} \omega_i x_i + \sum_{i=1}^{n} \omega_i \epsilon_i$

$\hat{y} = \hat{y} + \sum_{i=1}^{n} \omega_i \epsilon_i$

$E[(\hat{y}-y)^2] = E[((\hat{y}-y)+\sum \omega_i \epsilon_i)^2]$

$= E[(\hat{y}-y)^2] + E[(\sum \omega_i \epsilon_i)^2] + 2E[(\hat{y}-y)\sum \omega_i \epsilon_i]$

$= E[(\hat{y}-y)^2] + E[\sum \omega_i^2 \epsilon_i^2] + 0$

This is because $E[(\sum \omega_i \epsilon_i)^2] = E[\sum \omega_i^2 \epsilon_i^2]$

Since ϵ_i is independent from ϵ_j and $E[\epsilon_i \epsilon_j] = E[\epsilon_i]E[\epsilon_j] = 0$

So, all interacting terms in LHS will be 0.

Hence only square terms will survive.

Also ϵ_i is independent from $(\hat{y}-y)$ so $E[(\hat{y}-y)\epsilon_i \omega_i]$

$= E[\hat{y}-y]\, \omega_i E[\epsilon_i]$ (↳ 0)

$= 0$

So $E[(\hat{y}-y)^2] = L(\theta) + E[\sum \omega_i^2 \epsilon_i^2]$

$$\boxed{E[(\hat{y}-y)^2] = L(\theta) + \sigma^2 \sum \omega_i^2}$$ as $E[\epsilon_i^2] = \sigma^2$

iv) Injecting noise at the outputs :- Instead of fitting on the true labels, we perturb the true labels by adding some noise. This has a similar effect to L2 regularization.

v) Early Stopping :- In this method, we specify a patience parameter p. We keep track of the validation error after each step. A step can be each update or each epoch. Then if the validation error has not improved after p continuous steps we stop the algorithm. This can be combined with other regularization methods.

Analysis of Early Stopping :-

$$w_{t+1} = w_t - \eta \nabla w_t = w_0 - \eta \sum_{t=0}^{t} \nabla w_i$$

Let T be the maximum (absolute) value of $|\nabla w_i|$, then

$$|w_{t+1} - w_0| \leq \eta T (t+1)$$

, thus t controls the upper bound of $|w_{t+1} - w_0|$. If we limit t, in some sense we are limiting $|w_{t+1} - w_0|$ and if $w_0 = 0$, then this acts as a regularizer on the weights

vi) Ensemble Methods :- bagging does not help if the ensembles are correlated as they will make similar mistakes. Bagging is most effective when ensemble models are decorrelated as mistake of one rectified by the other model

2 options available in bagging are :-

i) Train different neural networks having different architecture

ii) Train multiple instances of the same network using different training samples

Both these options are computationally expensive.

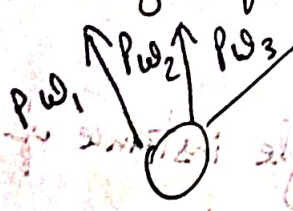
vii) Dropout :- In this method we train a single instance of neural network, but at each step (update or epoch) we drop some neurons by temporarily removing all their incoming and outgoing edges. Neurons to drop are decided randomly

sampled, where some probability is associated with each neuron. For example each neuron has $p = 0.8$ probability of being retained. The total number of possible neural networks that can be made are exponential in the number of total nodes. The full algorithm is given below:-

1) Initialize the network by choosing which neurons are active and participate in the computation.
2) Do forward propogation using the weights from previous iterations : updated
3) Do backward propogation and update the weights that are active
4) Repeat the procedure. We can decide if we want to keep one network for 1 update or 1 epoch. After each step, a new or in rare cases a previous configuration is made active and weights are updated.

Each network is trained twice rarely, but each weight may be updated many times in different configurations.
subset

At test time, the output is calculated from the whole network, but the output of each neuron will be multiplied by the probability it was retained with in each sample which is p or by the fraction of times it was active during training

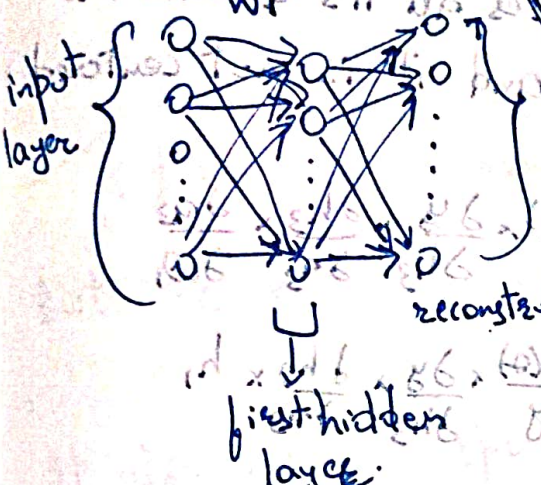


Dropout essentially applies or making noise to the hidden units. It prevents units from co adapting. A hidden unit cannot rely too much on other units as they may get dropped out any time. It allows the model to discover new features and makes it more robust.

Unsupervised Pre-Training :-

Instead of initializing the parameters randomly, is there a way to initialize them?

Consider the first hidden layer, having some n_1 neurons.



If we ignore the rest of the network and set up a new network such that task of this network is to reconstruct the input layer with the least error possible, with only the 1 hidden layer consisting of the neurons from 1st hidden layer of the original network.

So the objective is such that :- $\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{n_0}(\hat{x}_{ij}-x_{ij})^2$

each input is $\mathbb{R}^{n_0}$, and there are n such inputs. The reconstructed output is also $\mathbb{R}^{n_0}$. This is like PCA, the objective is same.

Whatever weights are learned as W_1 (between input to 1st hidden layer) can be used initialization in the original network.

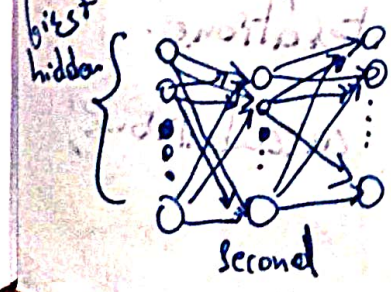
For the weights of the second layer (W_2) we calculate the output of each neuron in the first layer for each data point using the weights in the first step. So output of the 1st hidden layer will be $\mathbb{R}^{n_1}$, and there are n such inputs.

The objective will now be :- $\min \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{n_1}(\hat{h}_{ij}^{(1)}-h_{ij}^{(1)})^2$

$h_{ij}^{(1)}$ is the output from the j^{th} neuron of the 1st layer for the i^{th} point.

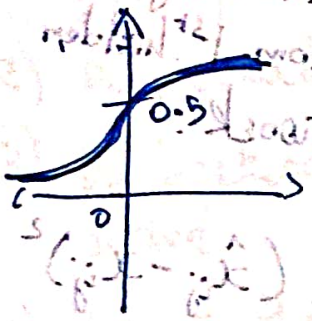
From this we obtain W_2.

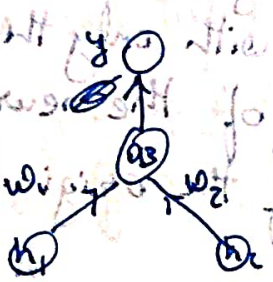
We do this to obtain W_1 and then initialize these as the weights for the original network.



Activation Functions :- 1) Linear :- If we have a deep linear neural network, then it faces difficulty in making approximate all the functions. Therefore some non-linearity is needed.

2) Sigmoid :- Sigmoid function compresses all its domain into $[0,1]$. The output of sigmoid is positive and it is not centered around 0.



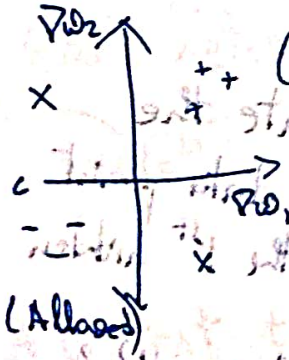


$$\nabla w_1 = \frac{\partial L(\theta)}{\partial y} \times \frac{\partial y}{\partial h_3} \times \frac{\partial h_3}{\partial a_3} \times \frac{\partial a_3}{\partial w_1}$$

$$= \frac{\partial L(\theta)}{\partial y} \times \frac{\partial y}{\partial h_3} \times \frac{\partial h_3}{\partial a_3} \times h_1$$

$$\nabla w_2 = \frac{\partial L(\theta)}{\partial y} \times \frac{\partial y}{\partial h_3} \times \frac{\partial h_3}{\partial a_3} \times h_2$$

Since neurons are sigmoid, h_1 and h_2 are positive. Besides h_1 and h_2, ∇w_1 and ∇w_2 are same. Hence both ∇w_1 and ∇w_2 will have the same sign. Second and fourth quadrant gradients cannot be easily calculated as sign of the components are opposite.



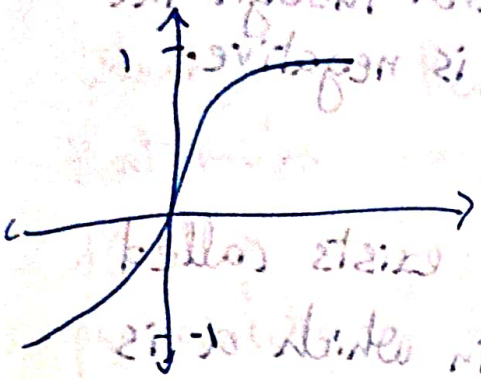
Centering removes this problem.

One more problem with sigmoid neuron is that near large absolute values of input, the neuron outputs 0 or 1, but the gradient vanishes as the rate of change is very small. So incorrect initializations of weights makes the neuron saturate $(0,1)$ and after saturation, since the gradient is small, the updates happen very small which keeps it saturated in the next iterations.

Third problem is that sigmoid involves sigmoid calculations, which is also expensive.

3) Tanh function : $\tanh(x)$

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x)$$



- compresses all its input to the range $[-1, 1]$. It is zero centered so it removes the opposite sign gradient.

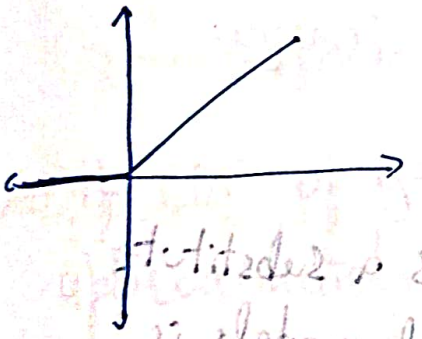
But the problem of vanishing gradient still exists. and saturation

It also involves eponential computations, so it is also relatively expensive.

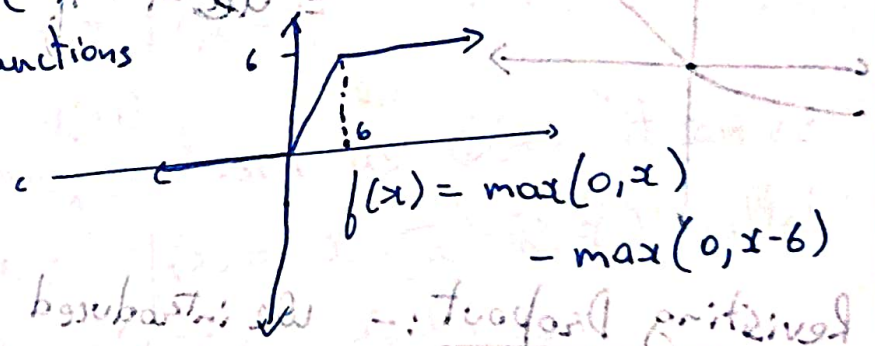
4) Relu (Rectified Linear Unit)

$$f(x) = \max(0, x)$$

Linear functions



We can recover piecewise



Advantages of ReLU:-

Does not saturate in the positive region, computationally efficient, in practice it converges much faster than sigmoid / tanh

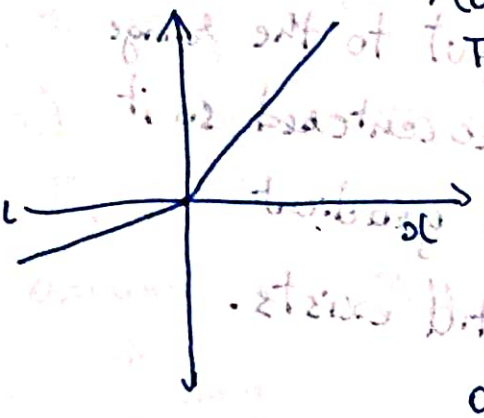
$$\frac{d}{dx}\text{ReLU}(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$$

=) If bias of some ReLU is set to a large negative value, $b << 0$, then the output as well as gradient for this neuron will be 0. Hence the vanishing gradient problem still exists for one half of input.

Leaky ReLU : No Saturation.

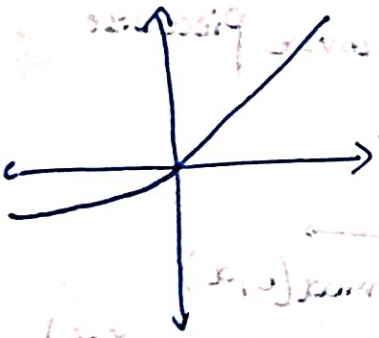
$f(x) = \max(0.1x, x)$ → can be α, any $\alpha > 0$. Typically small.

Some gradient value still exists even though the input is negative.



Another version exists called Parametric ReLU in which α is also treated as a parameter, which gets updated during backpropagation.

Exponential Linear Unit



$$f(x) = x \text{ if } x > 0$$
$$= \alpha e^x - 1 \text{ if } x \leq 0$$

Revisiting Dropout :- We introduced Dropout as a substitute to bagging. But in bagging, each individual models is overfit on the its corresponding bagged training data. But in dropout, each subnetwork configuration hardly fits on the data. To achieve the bagging effect with dropout, we introduce MaxOut neuron $\Rightarrow \max\{w_1 x + b_1, \cdots, w_n x + b_n\}$. Only the neuron that contributes the most gets the update, as for others, gradient is 0.

Dropouts can also be written as $U(x) = \begin{cases} 1 \cdot x, & p \\ 0 \cdot x, & (1-p) \end{cases}$

output of a neuron is as it is (x) if it got selected, the probability of that is p. And output is 0 otherwise.

It makes sense to combine and represent more compactly:- $f(x) = m \cdot x$

where $m \sim \text{Bernoulli}(\phi(x))$ where $\phi(x)$ is a function that maps $-\infty, -\infty$ between $[0, 1]$. For dropout $\phi(x) = p$ works, as each neuron will be selected with probability of p.

Alternatively it can be a cummulative distribution function of a Gaussian variable or even logistic function.

$$E[f(x)] = E[m \cdot x]$$
$$= x\, E[m]$$
$$= x \left[1 \times \phi(x) + 0 \times (1 - \phi(x))\right]$$
$$E[f(x)] = x\,\phi(x) \rightsquigarrow P(X \le x)$$

if CDF of gaussian function, $N(0,1)$ is chosen, as $\phi(x)$, it can be approximately written as $f(x) = x\,\sigma(1.72x)$ called GeLU.

GeLU. $\text{GeLU}(x) = x \cdot \sigma(1.72x)$

SELU (Scaled Exponential Linear Unit): Linear unit with output scaled such that mean is 0 and unit variance.

$$f(x) = \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha e^x - \alpha & \text{if } x \le 0 \end{cases}$$

$\lambda \approx 1.0507009$
$\alpha \approx 1.6773263$

Studies found that functions of the form $f(x) = x\,\sigma(\beta x)$ are good. Function of such forms, where β is a learnable parameter is called SWISH. if $\beta = 1.702$, it is GeLU.

when $f(x) = x\sigma(x)$ it is called SILU, Sigmoid weighted linear unit.

Convolution operation:-

$$S_{ij} = (I * K)_{ij} = \sum_{a=\lfloor -m/2 \rfloor}^{\lfloor m/2 \rfloor} \sum_{b=\lfloor -n/2 \rfloor}^{\lfloor n/2 \rfloor} I_{i-a, j-b} \cdot K_{\frac{m}{2}+a, \frac{n}{2}+b}$$

3D-filter is referred to as a volume. In case of 3-D filters, the volume is slid over the 3D image same as 2D case, and the convolution result is 2D.

Defining Quantities :-

⇒ Width (W_1), Height (H_1) and Depth (D_1) of the original input

⇒ The Stride S (increment in the slide of the kernel)

⇒ The number of filters is K.

⇒ The spatial extent (F) of each filter (the depth of each filter is same as the depth of each input)

⇒ The output is $W_2 \times D_2 \times D_2$.

Usually the extent of the filter is an odd integer as center of the matrix is properly defined.

If no padding is applied, the convolution reduces the size of the image as for the pixels in the border of the input, convolution is not defined.

The formula for output dimensions is given as:-

$$W_2 = W_1 - (F_1 - 1) = W_2 - F_1 + 1$$

$$H_2 = H_1 - F_1 + 1$$

This is because if the $F = 2a+1$ for $a \geq 0$, a columns are lost in the left border and a colums are lost in the right border of the input.

So total colums lost $= 2a = F-1$

$\therefore$ remaining $= W_1 - (F-1)$

Similarly for height, remaining height $= H_1 - (F-1)$

If we add padding of P pixels around the image, then the formula is upated as follows:-

$$W_2 = W_1 - F + 1 + 2P$$
$$H_2 = H_1 - F + 1 + 2P$$

Generally we do a stride $S = 1$ which is look at all pixels. $S = 2$ means that place kernel at every alternate pixel. (in both x and y dimension)

$$W_2 = \frac{W_1 - F + 2P}{S} + 1$$

$$H_2 = \frac{H_1 - F + 2P}{S} + 1$$

$\Rightarrow$ K filters will give us $K = 2D$ outputs. Hence $D_2 = K$

Convolutional Neural Networks :-

In traditional Machine learning, we would convert the image into a flat vector and the apply the classification algorithm. We can also apply edge detection algorithms and kernels before applying the classification algorithms. But these kernels are hard coded before the algorithm, and is treated completely as preprocessing.

We can also learn these kernels and treat them as params of the model. Not only this, multiple kernels can be learnt. Furthermore the neural network's architecture allows to learn multiple layers of kernels to produce the final output.

Difference between a CNN and a normal feedforward Network

In a feed-forward network, each neuron is connected to each neuron in the previous network. It is dense network and all weights are independently treated.

But in a CNN:-

Suppose there is an image of 4x4 given as input to the network, and a 2x2 kernel is trying to be learnt for the first hidden layer.

i) The input image is flattened :-

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

$\Rightarrow$ | 1 | 2 | .... | 15 | 16 |

Hence input layer has 16 neurons.

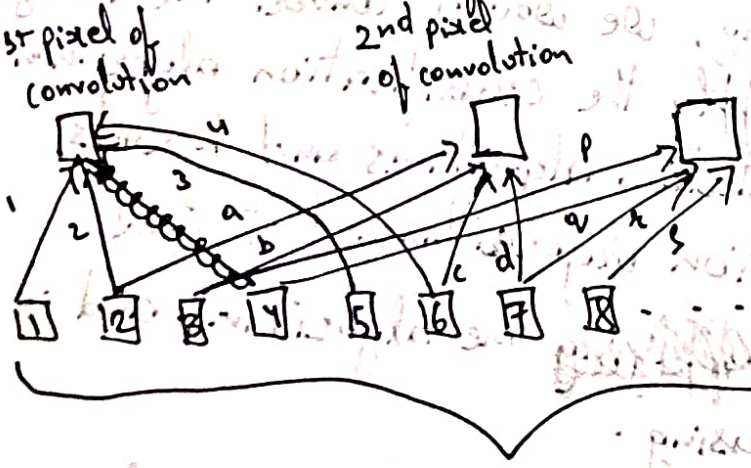
ii) The kernel is flattened :-

1	2
3	4

$\Rightarrow$ | 1 | 2 | 3 | 4 |

iii) The output image is calculated and flattened as well. So each pixel in the output of the convolution is a neuron in the next layer.

We can make the following observations :-



i) The output layer is sparsely connected with the previous layer. This is because the first pixel of convolution output is calculated using only 1, 2, 5 and 6 pixel of input. Similarly for all pixels. So the connectivity is sparse

ii) Since a single kernel is applied here, $1 = a = p$, $2 = b = q$, $3 = c = r$, $4 = d = s$

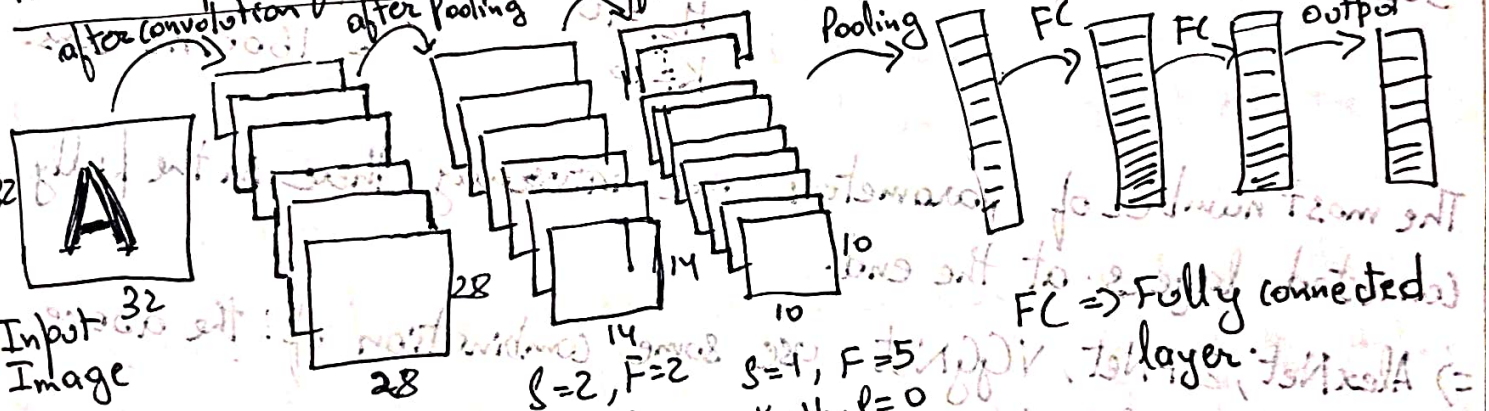
There is weight sharing in this neural network. More kernels can be applied but that also increases the number of output neurons (next convolution output, addition in depth) so weight sharing is still there.

Due to this weight sharing :- i) we look at the neighborhood of each pixel.

ii) we don't really loose information as in the next layers, there will be interaction between pixels which dependent on the input pixels. So indirect iteraction is there.

=> After performing convolution the result can be passed through an activation function.

An Example of a convolutional Neural Network:-



The first layer consists of 6 different 5x5 filters. Each Filter is 5x5 and there are 6 such such filters, so there are 6x25 = 150 parameters. Note that this means that there will be 150 "unique"/independent connections from the input layer to the first hidden layer. Moreover there are 28x28x6 neurons in the first hidden layers. This is decided by the formula discussed before.

The Layer after first hidden layer is the max pooling layer. In this Each component matrix in the depth of the first hidden layer is passed through a max filter independently to obtain the 6 corresponding matrices. Notice that a stride of 2 is used in the example, dimension rules apply in the same manner.

In max filter, just the max is selected within the neighborhood defined by the kernel.
Max pooling is done to reduce the size of the layers and hence reduce the number of parameters, preventing overfitting.
Also note that the max pooling layer learns no parameters, it is a fixed process involving no learnt parameters.

=) The next hidden layer (second) is again a convolutional layer with filter size of $5 \times 5 \times 6$. There are 16 such filters. Hence the number of parameters are 2,400. The number of neurons in the second hidden layer will be :— $W_2 = 14 - 5 + 1 = 10$
$H_2 = 10$ $= 10 \times 10 \times 16$
$K = 16$ $= 1600$ neurons.

The most number of parameters are naturally there in the fully connected layers at the end.

=) AlexNet, ZFNet, VGGNet use some combination of the above discussed architecture.

Notice that in each layer, if we are using multiple kernels, all those kernels are the same, both in regard of type of kernel and size of kernel. We can combine these options and the concatenate them to produce the output.
But this introduces a problem, if a filter is $F \times F \times D$, then each entry in the output requires $F \times F \times D$ computations. This makes it difficult to apply multiple kernels. We can use 1×1 convolutions to reduce the depth of the input.

⇒ The $1 \times 1 \times D$ convolution operation applies across the depth of the input.

=) We can apply D_1 such 1×1 Kernels to the input

image to produce a $W_2 \times H_2 \times D_2$ output, which is a reduction in the depth. Based on this convolved output we can use different sized kernels or pooling. Moreover we can also calculate another $W_2 \times H_2 \times D_2$ by 1×1 kernels and apply another set of kernels on this output. The possibilities are huge. Yes the computations are increased but the number of parameters in the later layers are greatly reduced.

To make the output consistent in dimensions, appropriate padding is added.

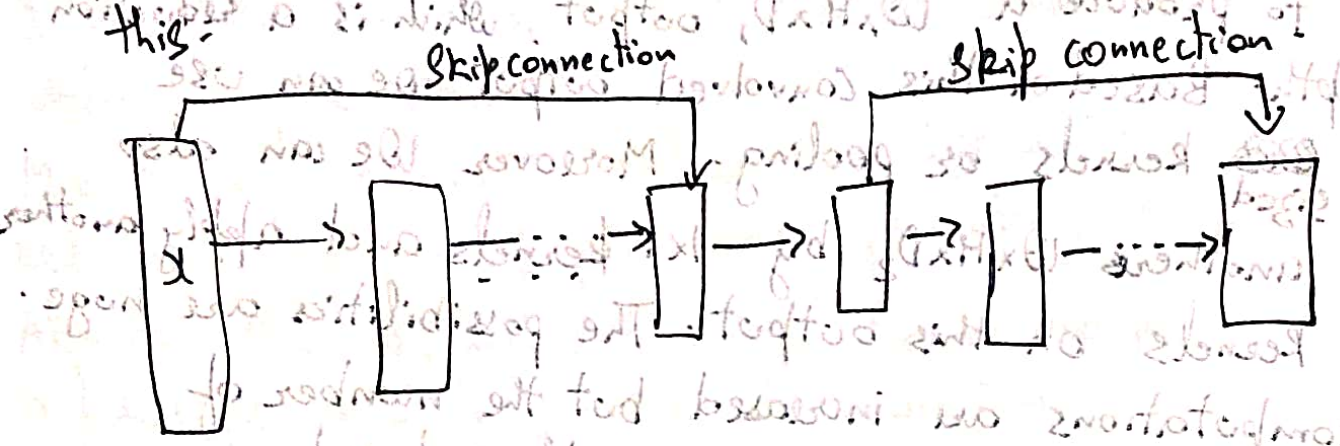
All these computations are done in parallel. This whole unit is called an inception module. The unit consists of 1×1 convolutions and pooling and other size convolutions. The output is concatenated and treated as the output of the inception module as a single layer.

=> One more change can be to do average pooling in the last layer before the fully connected layer. So if it has $W \times F \times D$ output, then average pooling reduces it to $1 \times 1 \times D$ outputs, applied by taking average of each component matrix. This reduces the number of parameters.

All these improvements were used in the GoogleNet.

ResNet :- Suppose we have a shallow Network that performs well... But if we make it deeper, we expect it to perform at least as better as the shallow network. But it does not happen in practice. To improve the performance of the deep network, we can use skip connections. Skip connections supply a copy of input to further layers. We can introduce skip connections after every few layers. Infact, ResNet did

this:



Ways to Visualize and Interpret a CNN

=> For each neuron or set of neurons, we can feed in images to this CNN and identify the images which cause these neurons to fire.

=> We can then trace back to the path in the image which causes these neurons to fire.

This will allow us to understand which neuron(s) are excited at what input.

=> One way to do this is to plot the kernels / filters as images and interpret them as patterns. The filters detect the patterns observed in their image plot. The problem with this approach is that the results are interpretable only for the first layer. Because after that the filters are applied on the input of previous layers.

=> second approach is through occlusion experiments. In this method, we occlude (gray out) different patches in the image and see the effect on the predicted probability of the correct class. The areas that affect the probability the most are the most important in the image.

=) Third approach is to calculate the gradient of a neuron with respect to each input pixel. This will us an image of gradients. Each image can be seen and wherever the value of gradient is high, more whiter pixels will appear in the output of this step with respect to that neuron. To calculate these gradients we can use back propogation. One problem with this method is that gradients can be highly positive or negative in areas of importance, but when visualised through a digital image, these gradients cancel the effect of each other in these areas and that leads to blunt results and noisy output.

To deal with this problem, we use guided backpropogation.

=) In this method, we choose a neuron of interest and set all other neurons in that layer to 0. Then we input an image to the network and perform a feed forward step. Then we perform the backpropogation step, to calculate the gradient matrix of this neuron w.r.t. to the input. But instead of allowing negative gradient to flow back, we clip negative gradients to 0. This is done to all neurons in all the layers. This prevents the effect discussed earlier and the results become more vivid in the resultant image. Basically we focus only on the positive influences (gradients)

=) One more method to interpret the results of a CNN is to optimize over images. The goal here is to create an image which gets classified as a certain class. So, if this image is passed through a trained ConvNet it should maximize the probability of the class of interest. Doing this would allow us to visualize what images are classified in which manner.

To do this we can set this as an optimization problem with respect to the pixels of the input image.

$$\text{argmax}\left(\text{Score}_c(I) - \lambda \mathcal{L}(I)\right) = \text{argmax}\left(-L(\theta)\right)$$

where $\text{Score}_c(I)$ = score for class before softmax

$\mathcal{L}(I)$ = some regularizer to ensure that I looks like an image.

In the whole process we keep the parameters of the convolutional neural network fixed. We adjust the image pixels so that the score of the class is maximized.

⇒ We start with a zero image / randomly initialized image

⇒ Set the score vector to be $[0\ 0 \ldots 1 \ldots 0 \ldots 0]$

⇒ Do a forward pass.

⇒ Compute the gradients $\frac{\partial L(\theta)}{\partial I_{ij}}$ for all i, j

⇒ Update the pixel $I[i,j] = I[i,j] - \eta \nabla_{I[i,j]} L(\theta)$

⇒ Repeat the process from forward pass.

Instead of limiting this process to only the output, we can focus on any one neuron in any layer to see what images makes that neuron to fire.

To do this

⇒ Feed an image through the network. (The propagation can stop at the neuron of interest)

⇒ Set activation in layer of interest to all zero, except for the neuron of interest.

⇒ Backprop to image

⇒ $I[i,j] = I[i,j] - \eta \nabla_{I[i,j]} L(\theta)$, where $L(\theta)$ can be negative of the pre-activation of the neuron of interest

Another similar idea is to create images using embeddings.

So there is a pre-trained CNN, now the objective is to feed this network with an image, not the embeddings of each layer, and then create/reconstruct an image such that the created image's embeddings are as close to the original image's embeddings.

Embeddings are the output of each layer.

So here the loss function is the squared error between the embeddings of the orignal image and the created image.

$$L(I) = \| \phi(I) - \phi(I_0) \|^2 + \lambda \| \phi(x) \|_6^6$$

$\phi(I)$ = vector of embeddings of the created image

$\phi(I_0)$ = vector of embeddings of the original image.

$$I[i,j] = I[i,j] - \eta \frac{L(I)}{I[i,j]}$$

=) This can be done for embeddings obtained from any layer.

=) We observe that the created image gets more abstract as we take embeddings from deeper layers.

Deep Dream

Suppose instead of starting with a blank image, we start with an actual image. Now we select some neuron of the network and our objective is to change the image such that this neurons fire even more.

our objective becomes $\max L(I)$

where $L(I) = h_{ij}^2$

Where h_{ij} is the output of the i^{th} neuron of the j^{th} layer.

We can repeat the same procedure of forward propogation and back propogation and update the image pixels $I[i,j] = I[i,j] + \eta \frac{\nabla L(I)}{I[i,j]}$

Doing this iteratively would make the image more and more like the patterns that cause the neuron to fire.

The Network has been trained to detect certain patterns which appear frequently in the training dataset. So it starts to see those patterns even when they hardly exist.

=) For example, if a cloud looks a little bit like a bird, the network will make it look more like a bird. This in turn will make the network recognize the bird even more strongly on the next pass, until a detailed bird appears in the image.

Deep Art

The objective of this is to ~~input~~ provide two images; a Content image, and a Style image, and then create an image such that the content of the created image is same as the content image but it is in the style of the Style image.

This is done through matching the embeddings of both the content image and Style image with the created image.

For Capturing the content of the image:-

$$L_{content}(\vec{p}, \vec{x}) = \sum_{ijk} (p_{ijk} - x_{ijk})^2$$ where $\vec{p}$ is the embeddings of the content image. $\vec{x}$ is the embeddings of the created image. This is done for some layer and the formula is written in the tensor notation where the output embeddings is in the form of tensor.

For capturing the style of the image:-

It turns out that if $V \in \mathbb{R}^{64 \times (256 \times 256)}$ is the activation/output at a layer, then, $V^T V \in \mathbb{R}^{64 \times 64}$ captures the style of the image.

The deeper layers capture more of this style information.

The objective can be the the following:-

$L_{style} = E_l = \sum_{ij} (G^l_{ij} - A^l_{ij})^2$, where G^l is the Gram matrix of the style image computed at layer L, and A^l is similarly for the created image.

⇒ The total loss is given as:- $L_{total}(I) = \alpha L_{content}(I) + \beta L_{style}(I)$

Fooling Deep CNNs

By optimizing over images, we can also fool deep CNN's to wrongly predict classes for obvious images. This is done through maximizing the log likelihood of the incorrect class and making changes in the images. The resultant images fools the network into predicting the wrong classes.

Now if instead, we gave blank image as input, then the resultant image is very noisy and un-interpretable, but, even then the network convincingly predicts it as belonging to a class.

This happens because images are very high dimensional entities. Out of these high dimensional images, only a few are interpretable by the human brain. But for a CNN, each image in this space is just a vector, and it just passes that vector through a learnt function which has its decision boundary. Even though the image is noisy to us, the CNN predicts it according to the output of the learnt function.