

Deep Learning Notes

Harshil Pandey

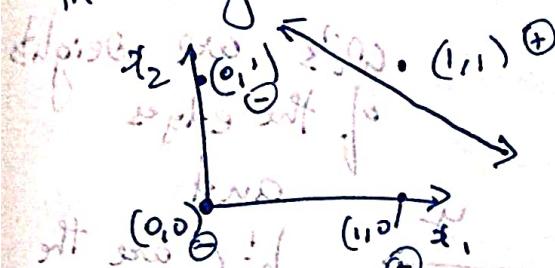
2025

Contents

1. Basics	2
2. Feed Forward Neural Network	8
3. Momentum Based Gradient Descent and Variants	21
4. Learning Rate Schemes	31
5. Bias and Variance	33
6. Ways to improve the Network Performance	36
7. Activation Functions	41
8. Convolutional Neural Network (CNN)	45
9. Ways to Visualize a CNN	51

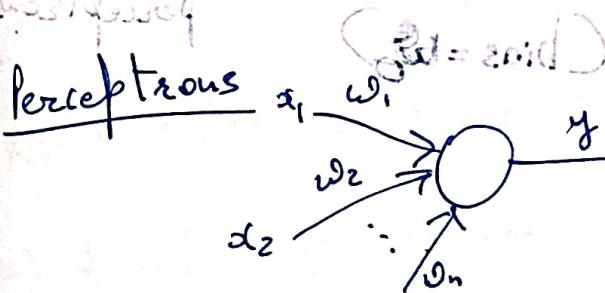
Linearly Separable Functions

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 function is not a linearly separable function.



a) w_0 is called the bias. If $w_0 > 0$, we support the class more.

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

\Rightarrow Any boolean function can be implemented using a network of perceptrons.

Assume that True = +1 and False = -1. There are 4 possible combinations for 2 inputs.

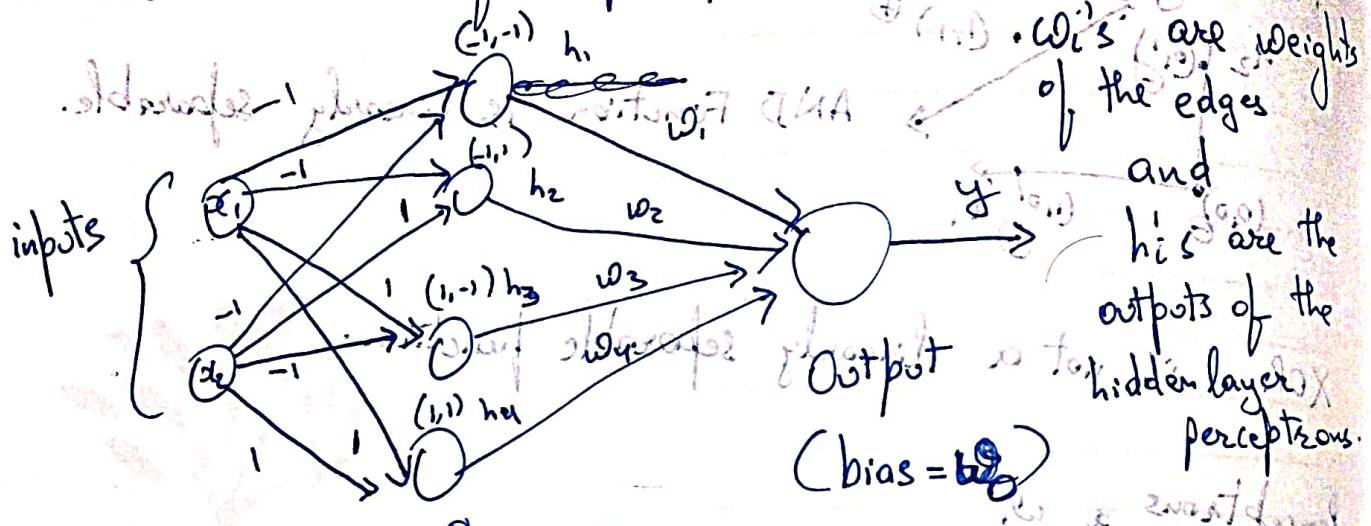
\Rightarrow For 2 inputs, the combinations can be $2^2 = 4$. Hence 2 boolean functions can be made for 2 inputs. Each combination of input can take output of -1 or 1. If input can take result of 2 boolean functions: therefore so for n inputs total possible boolean function: 2^n .

=> Example for 2 inputs.

total possible input combinations:

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

let us make four perceptrons:



at bottom si
to offstage. \therefore $b_i = -2$

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 $(-1) \times (-1) + (-1) \times (-1) - 2 = 0 \geq 0$

No other neuron will fire, because we have designed the 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 the ~~second~~ only third neuron will

first, hence w_3 can be adjusted w.r.t. to b_0 such that $\sum_{i=1}^3 w_i b_i + b_0 \geq 0$. Since no other neuron fires, w_3 can be adjusted independently as it will be the only one contributing to $\sum_{i=1}^3 w_i b_i + b_0$ 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.

\Rightarrow 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.

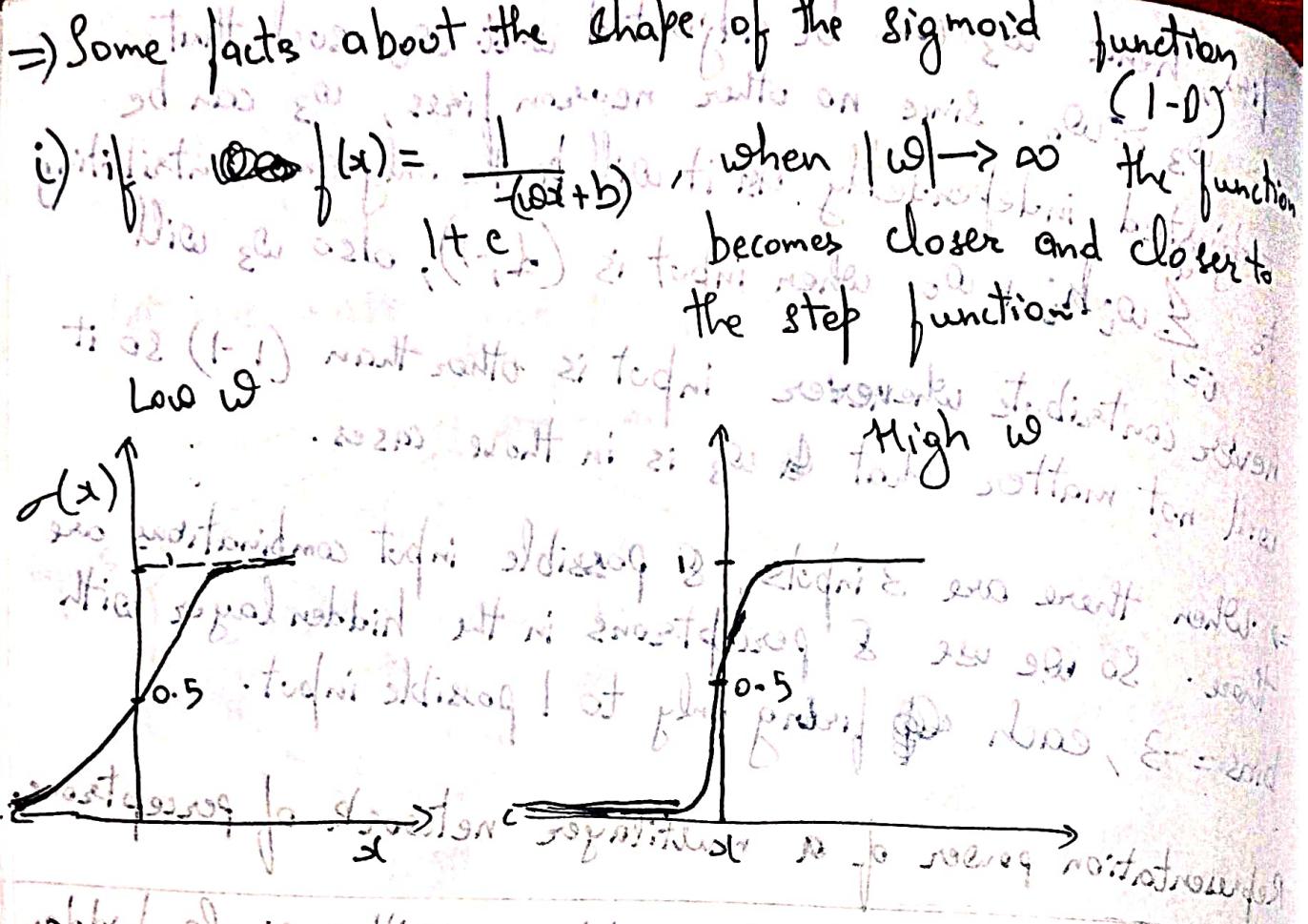
Representation power of a multilayer network of perceptrons:

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

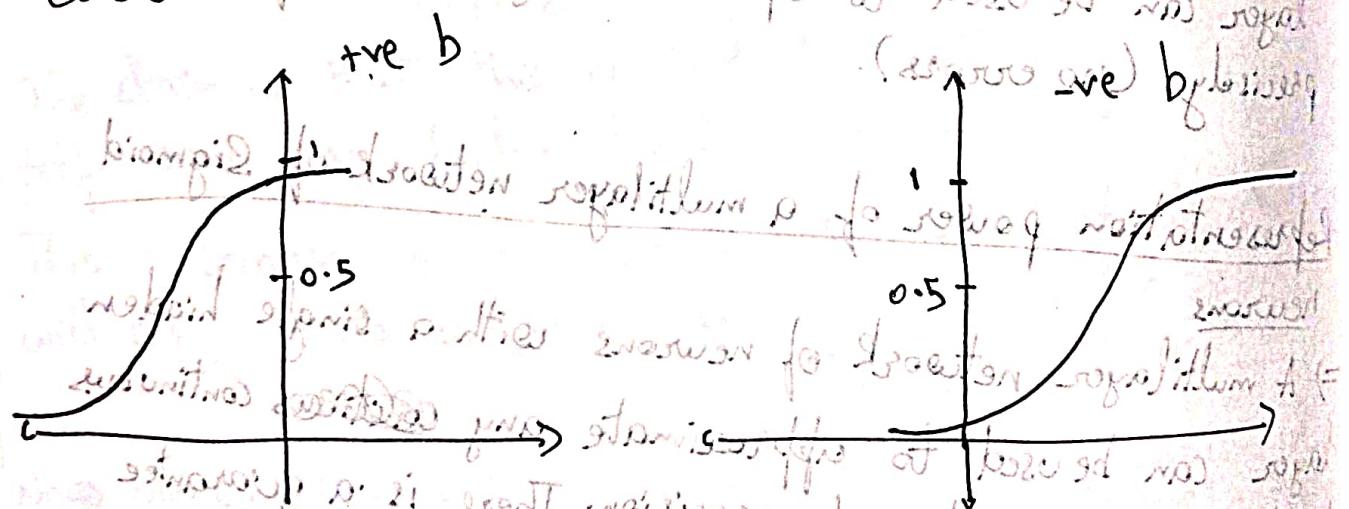
Representation power of a multilayer network of Sigmoid neurons

\Rightarrow 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 $f(x) : \mathbb{R}^n \rightarrow \mathbb{R}^m$, 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 $\epsilon > 0$.

Continuous function approximation
by multiple hidden layers present



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



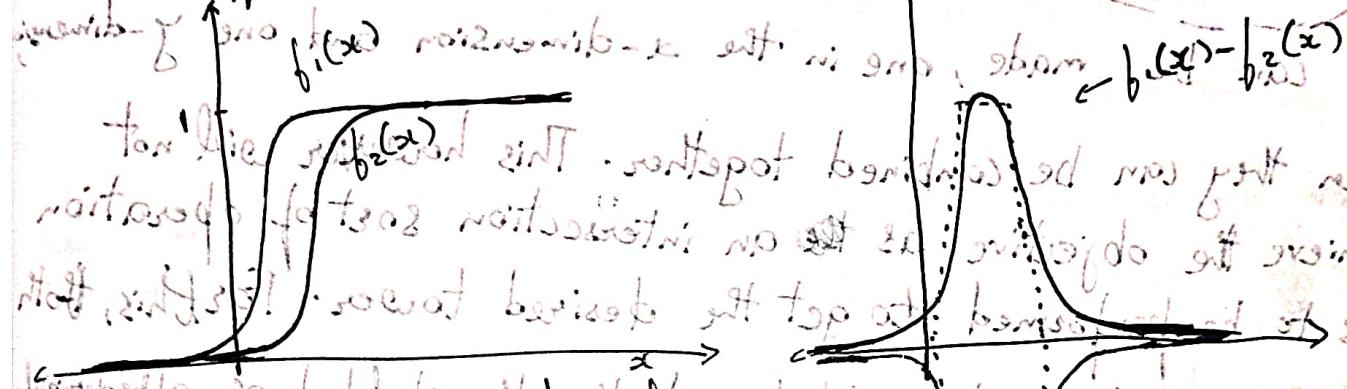
Now let there be a function $f(x)$

$$g = f(x) - f_0(x)$$

This can be approximated through equal width rectangles.

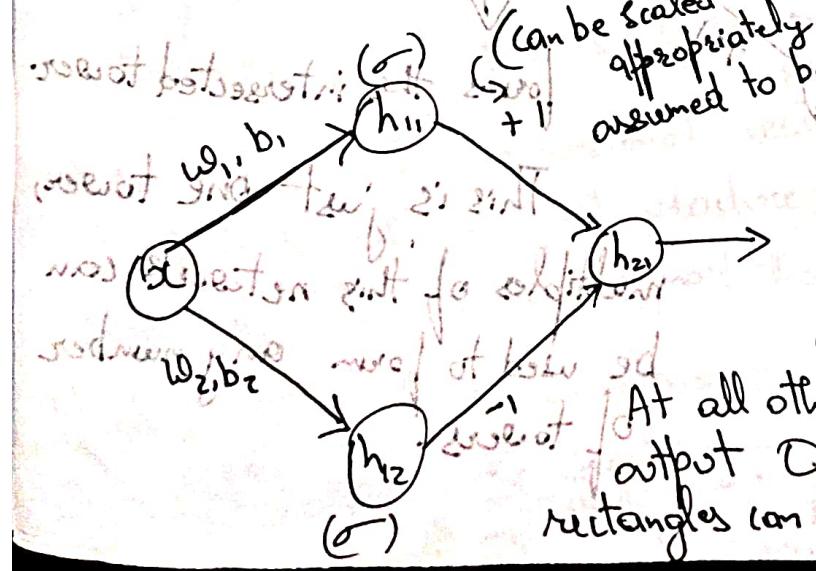
If we introduce more rectangles by reducing the width of the rectangles, then the approximation is more accurate. The question is how to create these rectangle like structures using sigmoid neurons.

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



By adjusting the bias values b_1 and b_2 in $f_1(x)$ and $f_2(x)$, we can manipulate the position of the rectangle. By adjusting the width of the rectangle, we can also manipulate it to translate the like structure linearly.

The height of the rectangle can be manipulated by scaling the output of the sigmoid appropriately. Combining this into a neural network:

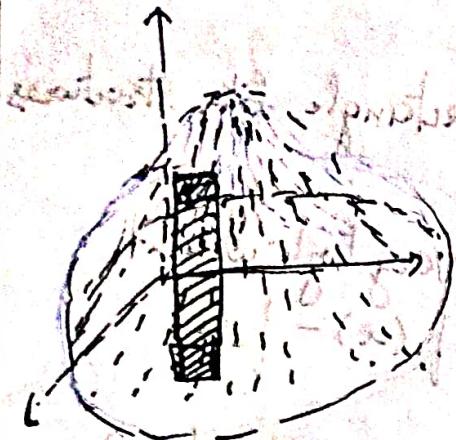


(can be scaled appropriately to achieve the correct height, assumed to be 1 for simplicity)

Notice that by adjusting w 's and b 's we can make a very specific rectangle at desired position and height.

At all other inputs, 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 :-



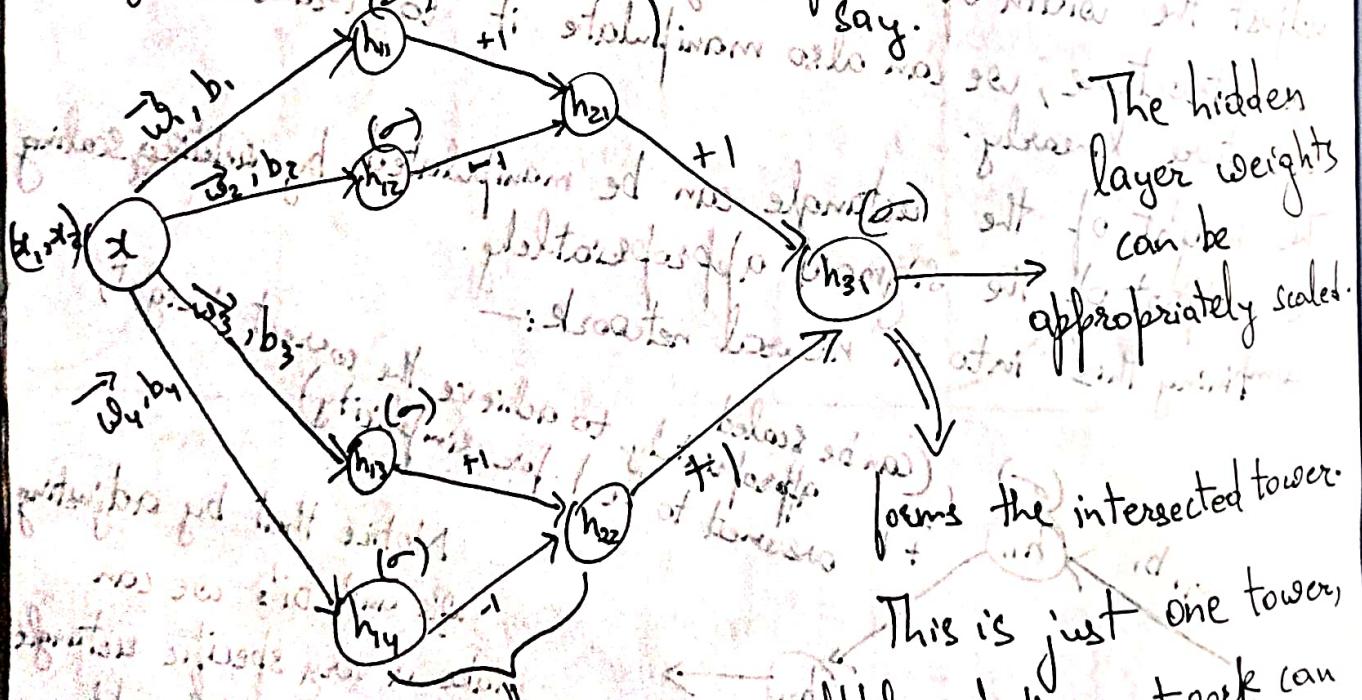
$$\left[(x_{\min} - x_{\max}) \right] \quad \left[(y_{\min} - y_{\max}) \right]$$

then two independent towers in the

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.



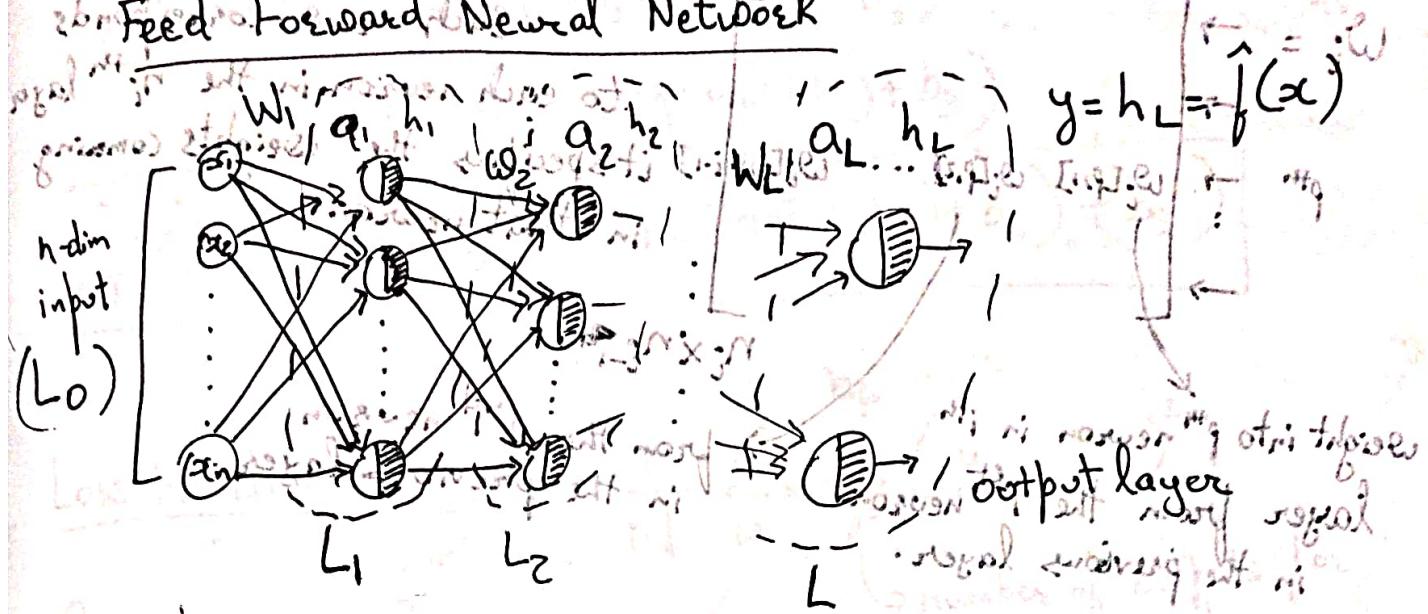
The hidden layer weights can be appropriately scaled.

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 the 2nd dimension say.

Feed Forward Neural Network



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

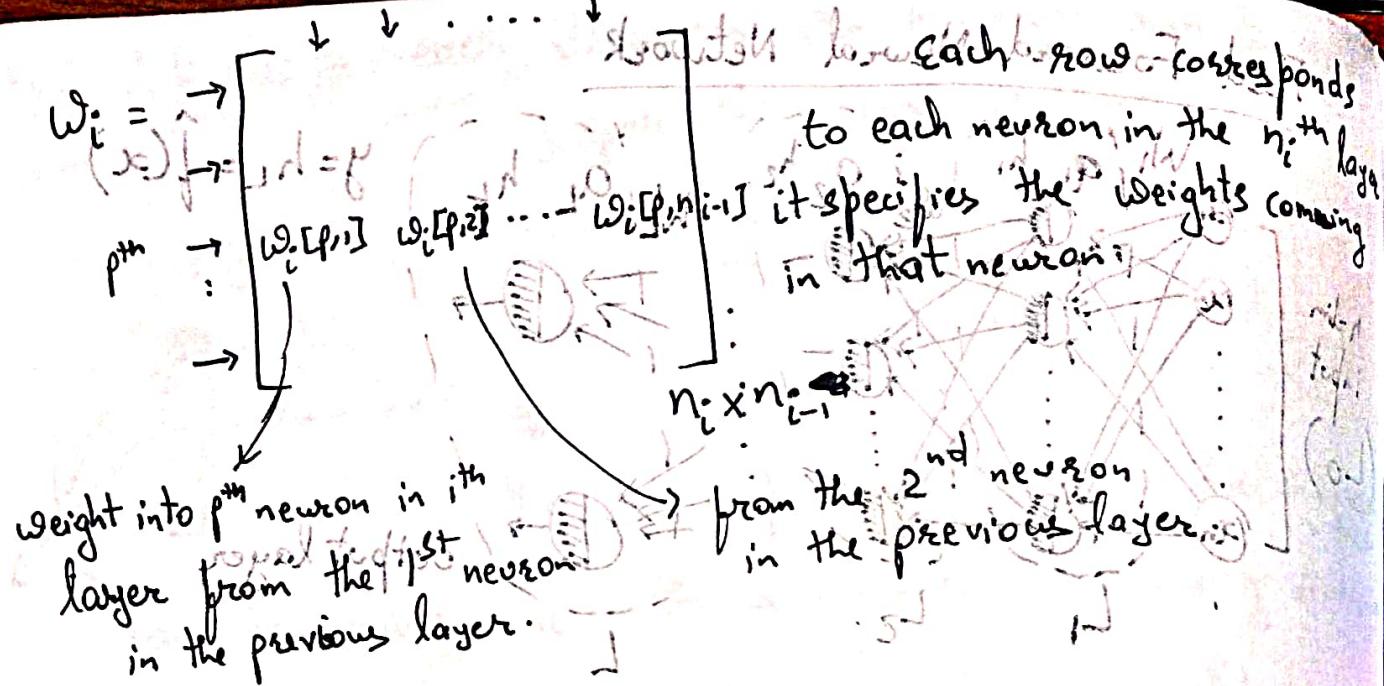
\Rightarrow The network contains $L-1$ hidden layers, each having n_i neurons. $n_0 = n$ (input). i represents the layers index.

\Rightarrow Each neuron in the hidden layers and output layer can be split into two parts: pre-activation and activation.

\Rightarrow a_i is an n -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 .

\Rightarrow o_i is an n -dimensional vector such that the j^{th} component of o_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 $f_j(\omega x + b)$.

\Rightarrow $w_i \in \mathbb{R}^{n \times n_{i-1}}$ 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_{i-1} layer to p^{th} neuron in L_i layer.



Set pre-activation of the i^{th} layer will be :

⇒ The activation a_i of the i^{th} layer is given by

$a_i = W_i \times h_{i-1} + b_i$ (forward pass, $n_i \times 1$)

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

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

z) The activation at the Output Layer is given by:

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

Writing it as a function of x (assume only 2 hidden layers are there),

$$h_2(x) = O(a_2) = O(W_2 h_1 + b_2)$$

$$h_3(x) = O(a_3) = O(W_3 g(a_2) + b_3)$$

$$\hat{f}(x) := \Theta(\omega_3 g(\omega_2 h(x) + b_2) + b_3)$$

$$= \Theta(\omega_3 g(\omega_2 g(a) + b_2) + b_3)$$

$$\boxed{\hat{f}(x) = \Theta(\omega_3 g(\omega_2 g(\omega_1 x + b_1) + b_2) + b_3)} \quad (9)$$

(total cost) \rightarrow works

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

To test thing with N data points. y_{ij} : adding all outputs for each data point.

$\{w_1, w_2, \dots, w_L\}$: number of outputs for each data point.

$\{b_1, b_2, \dots, b_L\}$: offset for input data points.

[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.

$h_L = \Theta(a_L) = w_L a_L + b_L$, (w_L and b_L are not the same as w_1 and b_1)

(e) For Multiclass 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 most appropriate output activation is the softmax function:

$$\hat{y}_j = \Theta(a_L)_j = \frac{e^{a_{Lj}}}{\sum_{k=1}^n e^{a_{Lk}}} = \left\{ \begin{array}{l} \text{the } j^{\text{th}} \text{ component of an} \\ \text{output vector for 1 data} \\ \text{point. } \hat{y}_j \in (0, 1) \end{array} \right.$$

In this case, the more appropriate loss function is the cross-entropy loss $L(\theta) = -\sum_{j=1}^{n_r} y_j \log(\hat{y}_j)$ (Loss for 1 data point).

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

\hat{y}_j represents the probability of the 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 $e_t = [0 \dots 0, 1, 0]$

The loss corresponding to this point is:

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

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

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

Let us start with $\nabla_{\hat{y}} L(\theta)$ - the t^{th} element of $\nabla_{\hat{y}} L(\theta)$.

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

$$\left(\frac{\partial L(\theta)}{\partial \hat{y}_j} \right) = \begin{cases} \frac{1}{\hat{y}_t} & \text{if } j=t \\ 0 & \text{if } j \neq t \end{cases}$$

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

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

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

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

For that we know that:- $\hat{y}_j = \frac{e^{a_{Lj}}}{\sum_{k=1}^n e^{a_{Lk}}} \quad \{ \text{so } t \text{ max.} \}$

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

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

\Downarrow

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

$$= -\frac{1}{\hat{y}_t} \cdot \frac{\partial}{\partial \hat{y}_t} \left(\frac{e^{a_{L,t}}}{\sum_{k=1}^n e^{a_{Lk}}} \right)$$

$$= -\frac{1}{\hat{y}_t} \cdot \left(\frac{\sum_{k=1}^n e^{a_{Lk}} \cdot \frac{\partial (e^{a_{L,t}})}{\partial a_{Lj}} - e^{a_{L,t}} \cdot \frac{\partial (\sum_{k=1}^n e^{a_{Lk}})}{\partial a_{Lj}}}{(\sum_{k=1}^n e^{a_{Lk}})^2} \right)$$

$$= -\frac{1}{y_t} \times \left(\frac{\sum_{k=1}^{n_L} e^{\alpha_{Lk}} \alpha_{Lt} \mathbb{1}(t=j) \frac{(0.16)}{0.06} e^{\alpha_{Lj}}}{\left(\sum_{k=1}^{n_L} e^{\alpha_{Lk}} \right)^2} \right)$$

$$= -\frac{1}{y_t} \times \left(\frac{\mathbb{1}(t=j) e^{\alpha_{Lj}}}{\sum_{k=1}^{n_L} e^{\alpha_{Lk}}} - \frac{e^{\alpha_{Lj}}}{\sum_{k=1}^{n_L} e^{\alpha_{Lk}} \times \left(\sum_{k=1}^{n_L} e^{\alpha_{Lk}} \right)} \right)$$

Belongs to the following

$$= -\frac{1}{y_t} \left(\frac{\mathbb{1}(t=j) \hat{y}_j}{\sum_{k=1}^{n_L} e^{\alpha_{Lk}}} \right)$$

This step shows that \hat{y}_j is the expected value of y_j given the observed information.

\hat{y}_j is the first term of \hat{y} .

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

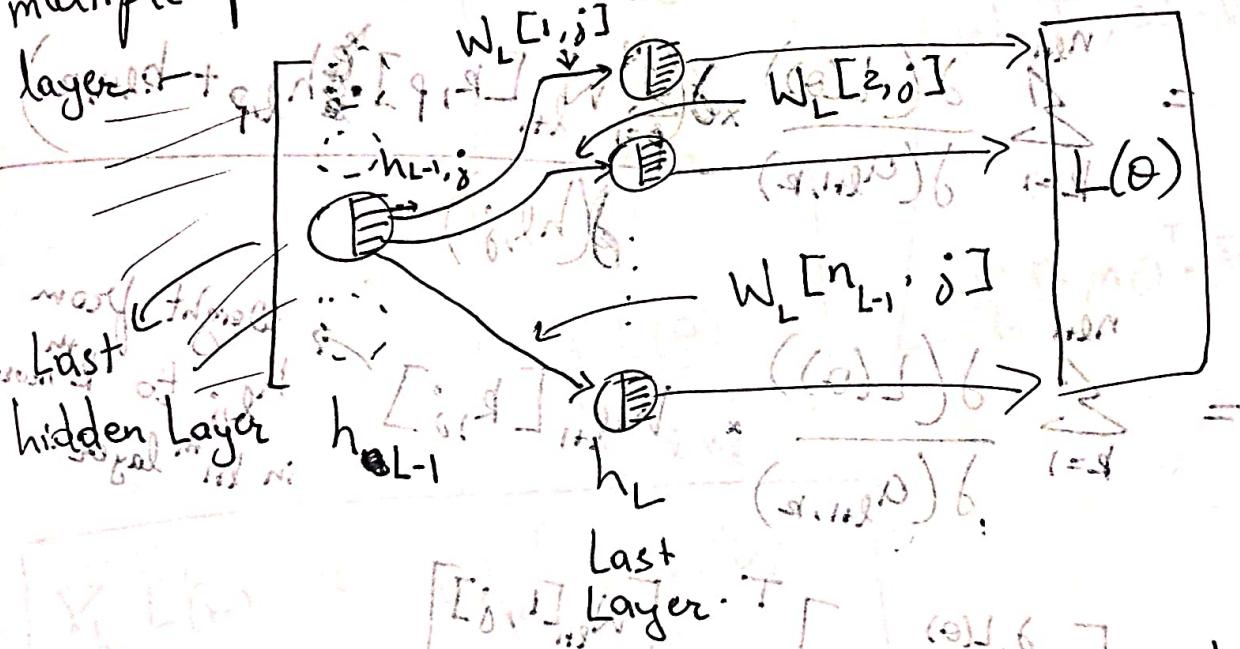
$$\Rightarrow \frac{\partial L(\theta)}{\partial \alpha_{Lj}} = -\left(\mathbb{1}(t=j) - \hat{y}_j \right) \frac{(0.16)}{0.06} = \frac{(0.16)}{0.06}$$

Therefore: $\nabla_{\alpha_L} L(\theta) = - \begin{bmatrix} \mathbb{1}(t=1) - \hat{y}_1 \\ \mathbb{1}(t=2) - \hat{y}_2 \\ \vdots \\ \mathbb{1}(t=n_L) - \hat{y}_{n_L} \end{bmatrix}$

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

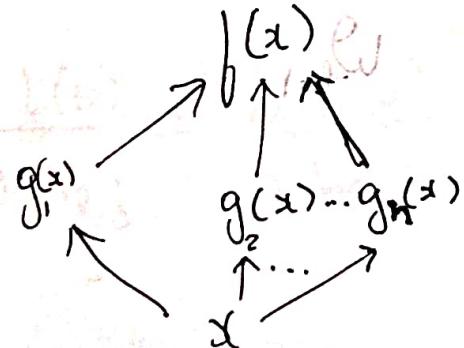
We now have the formulas for gradients of the ~~next~~ output layer. This is straightforward as the output layer directly goes into the loss function.

This is not true however, for any layer ~~except~~ 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 $h_{L-1, j}$ th neuron of the last hidden layer affects the path through L paths. Hence we use chain rule for multiple path to calculate

$\frac{\partial L(\theta)}{\partial h_{L-1, j}}$ th \Rightarrow 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

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

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

Let us break it into two parts:

- First part is $\alpha_{l+1, k}$ (pre-activation of the next layer).
- Second part is $L(\theta)$.

Now, we can write:

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

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

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

$$= \sum_{k=1}^{n_{l+1}} \frac{\delta(L(\theta))}{\delta(a_{l+1,k})} \times W_{l+1}[k, :] \quad \begin{matrix} \text{Weight from} \\ h_{l+1} \text{ to } k^{\text{th}} \\ \text{node} \\ \text{in } l+1^{\text{th}} \text{ layer} \end{matrix}$$

$\text{grad} = \left[\frac{\partial L(\theta)}{\partial \theta_1}, \dots, \frac{\partial L(\theta)}{\partial \theta_n} \right]$
 - $\frac{\partial L(\theta)}{\partial \theta_j}$ \rightarrow $W_{l+1}^{[j], \theta_j}$ \rightarrow $W_{l+1}^{[2], j}$ \rightarrow $W_{l+1}^{[1], j}$ \rightarrow $W_{l+1}^{[n], j}$
 - $\frac{\partial L(\theta)}{\partial \theta_j}$ \rightarrow $W_{l+1}^{[j], \theta_j}$ \rightarrow $W_{l+1}^{[2], j}$ \rightarrow $W_{l+1}^{[1], j}$ \rightarrow $W_{l+1}^{[n], j}$
 - $\frac{\partial L(\theta)}{\partial \theta_j}$ \rightarrow $W_{l+1}^{[j], \theta_j}$ \rightarrow $W_{l+1}^{[2], j}$ \rightarrow $W_{l+1}^{[1], j}$ \rightarrow $W_{l+1}^{[n], j}$
 - $\frac{\partial L(\theta)}{\partial \theta_j}$ \rightarrow $W_{l+1}^{[j], \theta_j}$ \rightarrow $W_{l+1}^{[2], j}$ \rightarrow $W_{l+1}^{[1], j}$ \rightarrow $W_{l+1}^{[n], j}$

$\frac{\partial L(\theta)}{\partial \theta_j}$ is called the j^{th} column of the gradient matrix with respect to the j^{th} feature.

$$\nabla_{\theta} L(\theta) = \frac{1}{N} \sum_{i=1}^N \nabla_{\theta} l_i(\theta)$$

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

(s.t.) $\theta = [(\theta)_1 \ \dots \ (\theta)_n]^T$

Here

$$\nabla_{h_l} L(\theta) = \begin{bmatrix} \frac{\partial L(\theta)}{\partial h_{l,1}} \\ \vdots \\ \frac{\partial L(\theta)}{\partial h_{l,n}} \end{bmatrix}$$

$$= \begin{bmatrix} W_{l+1}^T [\cdot, 1]^T \cdot \nabla_{a_{l+1}} L(\theta) \\ \vdots \\ W_{l+1}^T [\cdot, n]^T \cdot \nabla_{a_{l+1}} L(\theta) \end{bmatrix}$$

What happens if there are multiple layers? $L(\theta) = \dots \circ (\theta)_1 \circ \dots \circ (\theta)_L$

So $\nabla_{h_l} L(\theta)$ has $(\theta)_l$ elements.

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

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

~~$h_l = g(a_l)$~~

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

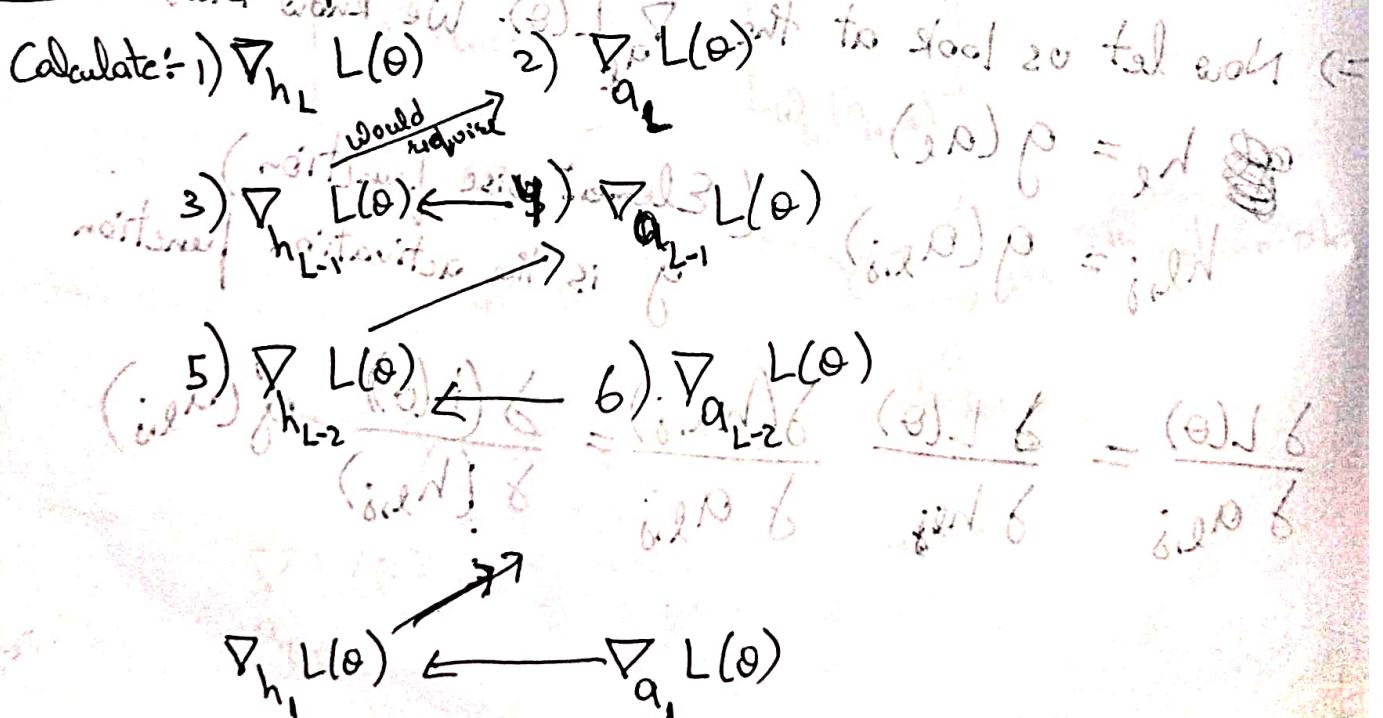
(g is the activation function)

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

$$\nabla_{a_e} L(\theta) = \left[\begin{array}{c} \frac{\partial L(\theta)}{\partial a_{e,1}} \\ \frac{\partial L(\theta)}{\partial a_{e,2}} \\ \vdots \\ \frac{\partial L(\theta)}{\partial a_{e,n_e}} \end{array} \right] = \left[\begin{array}{c} \frac{\partial L(\theta)}{\partial h_{e,1}} \cdot g'(a_{e,1}) \\ \frac{\partial L(\theta)}{\partial h_{e,2}} \cdot g'(a_{e,2}) \\ \vdots \\ \frac{\partial L(\theta)}{\partial h_{e,n_e}} \cdot g'(a_{e,n_e}) \end{array} \right]$$

$\nabla_{a_e} L(\theta) = \nabla_{h_e} L(\theta) \odot [\dots, g'(a_{e,i}) \dots]$ Element wise multiplication of $\nabla_{h_e} L(\theta)$ and $g'(a_e)$ also called as Hadamard Product

Chronology of Gradient Computation



\Rightarrow Computing gradients for the parameters $\nabla_{W_k} L(\theta)$ $n_k \times n_{k-1}$
 We know that :-

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

$$\text{and } \nabla_b L(\theta)$$

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

$$\cancel{W_l[i, k]}$$

Note that $W_l[j, k]$ only comes in the computation of $a_{l,j}$. Hence

$W_l[j, k]$ affects $L(\theta)$ only

through 1 path.

Hence:-

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

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

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

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

$$\Rightarrow \nabla_{W_l} L(\theta) =$$

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

$$\nabla_{W_l} L(\theta) = \begin{bmatrix} \frac{\partial L(\theta)}{\partial a_{l,1}} \cdot h_{l-1,1} & \frac{\partial L(\theta)}{\partial a_{l,2}} \cdot h_{l-1,2} & \dots & \frac{\partial L(\theta)}{\partial a_{l,n_{l-1}}} \cdot h_{l-1,n_{l-1}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial L(\theta)}{\partial a_{l,1}} \cdot h_{l-1,1} & \frac{\partial L(\theta)}{\partial a_{l,2}} \cdot h_{l-1,2} & \dots & \frac{\partial L(\theta)}{\partial a_{l,n_{l-1}}} \cdot h_{l-1,n_{l-1}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial L(\theta)}{\partial a_{l,1}} \cdot h_{l-1,1} & \frac{\partial L(\theta)}{\partial a_{l,2}} \cdot h_{l-1,2} & \dots & \frac{\partial L(\theta)}{\partial a_{l,n_{l-1}}} \cdot h_{l-1,n_{l-1}} \end{bmatrix}$$

$$\Rightarrow \nabla_{W_L} L(\theta) = \begin{bmatrix} \frac{\partial L(\theta)}{\partial a_{L,1}} \\ \frac{\partial L(\theta)}{\partial a_{L,2}} \\ \vdots \\ \frac{\partial L(\theta)}{\partial a_{L,N_L}} \end{bmatrix} = \begin{bmatrix} h_{L-1,1} & h_{L-1,2} & \dots & h_{L-1,N_{L-1}} \end{bmatrix}^T$$

(a) J 6

$$\boxed{\nabla_{W_L} L(\theta) = \nabla_{a_L^T} L(\theta) \cdot h_{L-1}^T}$$

(a) J 6

Simpler argument
as $\frac{\partial L(\theta)}{\partial a_{L,i}}$
is just $\frac{\partial L(\theta)}{\partial b_L}$

For $\nabla_{a_L^T} L(\theta)$, notice that $\frac{\partial L(\theta)}{\partial a_{L,i}} = \frac{\partial L(\theta)}{\partial a_{L,0}} \times \frac{\partial a_{L,0}}{\partial a_{L,i}}$

$$\text{So } \boxed{\nabla_{a_L^T} L(\theta) = \nabla_{a_L^T} L(\theta) \cdot \frac{\partial a_{L,0}}{\partial a_{L,i}}}$$

(a) J 6

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

(a) J 6

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

(a) J 6

$$\frac{e^x - e^{-x}}{e^x + e^{-x}} \cdot \frac{e^x + e^{-x}}{e^x + e^{-x}}$$

(a) J 6

$$= \frac{e^{2x} - 1}{e^{2x} + 1}$$

(a) J 6

$$= \frac{e^{2x} + 1 - 2}{e^{2x} + 1}$$

(a) J 6

Contours

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

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

\Rightarrow Wherever the contour rings are well located, the function has a steep curvature.
 \Rightarrow 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 slopes, contour rings are further apart.

\Rightarrow 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.

Rapid progress is made in areas where contour rings are closer.

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, 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:-

where, $\nabla w_t = \nabla w_t + \beta U_{t-1}$ where U_t is the combination of the current gradient and historical directions.

where,
 $U_t = \nabla w_t + \beta U_{t-1}$ we start with w_0 and $0 \leq \beta < 1$.
 (β) is called β is called momentum factor.
 $w_0 = \nabla w_0$, then $w_1 = w_0 - \eta \nabla w_0$.
 $w_1 = \nabla w_1 + \beta U_0 = \nabla w_1 + \beta \nabla w_0$.

$$\begin{aligned} \text{So } w_2 &= w_1 - \eta (\nabla w_1 + \beta (\nabla w_0 + \beta \nabla w_0)) \\ &= w_1 - \eta (\nabla w_1 + \beta \nabla w_1 + \beta^2 \nabla w_0) \\ &= w_1 - \eta \sum_{i=0}^{2-1} \beta^i \nabla w_i \quad \text{which is the current gradient} \\ &\quad \text{if we then } \nabla w_1, \text{ and then the least} \\ &\quad \text{importance to } \nabla w_0. \quad \text{so if we make small steps} \\ \therefore w_{t+1} &= w_t - \eta \sum_{i=0}^t \beta \nabla w_i \quad \text{in some direction, after some steps,} \\ &\quad \text{the "momentum" } (U_t) \text{ will grow} \\ &\quad \text{and we will start making bigger} \\ &\quad \text{steps.} \end{aligned}$$

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

G.D. in most situations is better than momentum based To deal with the overshooting problem of Momentum based gradient descent we look at the Nesterov Accelerated Gradient Descent. Instead of directly combining the historical direction U_{t-1} with the current gradient to calculate U_t we check what would happen if ∇w_t was not there. Then U_t will be βU_{t-1} :- $U_t = \beta U_{t-1}$

$$\text{So } W'_{t+1} = W_t - \eta U_t = W_t - \eta \beta U_{t-1}$$

If we ignore the step size η , then W'_{t+1} will be $W_t - \beta U_{t-1}$. Let us calculate the gradient w.r.t this 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 U_t is calculated like this:-

$$U_t = \nabla(W'_{t+1}) + \beta U_{t-1} \quad (\text{it would add on the momentum})$$

Since both $\nabla W'_{t+1}$ and βU_{t-1} are similar directions.

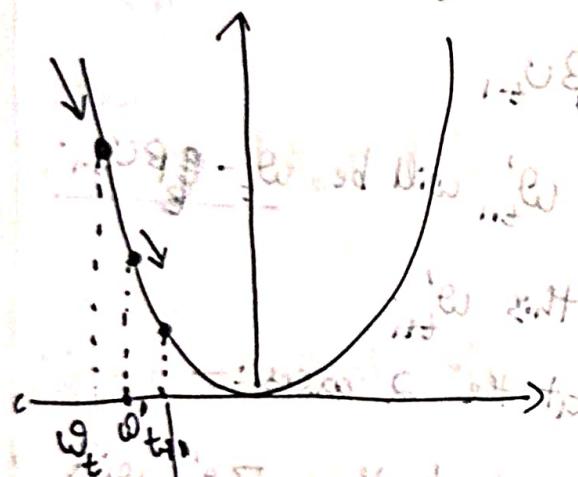
2) Overshoot:- if we overshoot, then $\nabla W'_{t+1}$ will be in opposite like direction with the previous directions. So if U_t is calculated like this:-

$U_t = \nabla W_{t+1} + \beta U_{t-1}$, it would reduce the momentum effect as ∇W_{t+1} and βU_{t-1} affect in opposite like directions. Hence this will prevent/reduce overshooting as it would reduce the momentum effect.

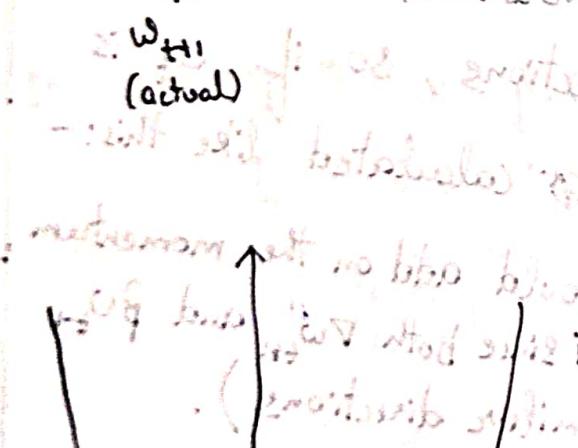
In both cases:-

$$U_t = \nabla(W_t - \beta U_{t-1}) + \beta U_{t-1}$$

$$W_{t+1} = W_t - \eta U_t$$

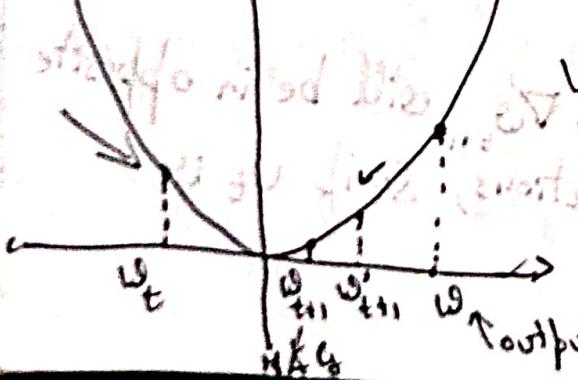


$W_{t+1} = W_t - \beta U_{t-1}$ looking ahead to W_{t+1} . In this case, looking ahead to W_{t+1} suggests that we should go along with the momentum and leads us to the next equilibrium.



$W_{t+1} = W_t - \beta U_{t-1}$ looking ahead to W_{t+1} suggests that we will overshoot and it cancels some momentum to avoid overshoot and leads us to W_{t+1} .

W would have been the output for Momentum-Based GD. (Overshooting is high)



Output for Momentum Based GD:

so NAG oscillates, lesser than SGD. To stabilize it we can do so:
1) halve learning rate after every 5 epochs.

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 initial hyperparameters and t is the step number. If η_0 or $k \uparrow$ then the decay is more rapid.

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

5) For Momentum algorithms

$$\beta_t = \min\left(1 - 2^{-1 - \log_2(\lfloor \frac{t}{350} \rfloor + 1)}, \beta_{\max}\right), \text{ where}$$

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 $\Delta w_t = w_t - \eta \nabla w_t$. The benefit of this is that gradients of most of the features will be zero 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 iterations. 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 :- $v_t = v_{t-1} + (\nabla w_t)^2$

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

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

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.

\Rightarrow 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 learning algorithm. v_t keeps on increasing.

\Rightarrow 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)

This is done in RMS Prop-algorithm.

$$v_t = \beta v_{t-1} + (1-\beta)(\nabla w_t) \quad 0 \leq \beta \leq 1, \text{ typically } 0.9, 0.99$$

let us expand v_t

$$\begin{aligned} v_3 &= 0.9v_2 + 0.1\nabla w_3^2 \\ v_3 &= 0.9(0.9v_1 + 0.1\nabla w_2^2) + 0.1\nabla w_3^2 \\ v_1 &= 0.9v_0 + 0.1\nabla w_1^2 \\ v_0 &= 0.9(0.9v_0 + 0.1\nabla w_0^2) + 0.1\nabla w_1^2 \\ \Rightarrow v_3 &= 0.9(0.9(0.9v_0 + 0.1\nabla w_0^2) + 0.1\nabla w_1^2) + 0.1\nabla w_3^2 \end{aligned}$$

$$\begin{aligned} v_3 &= 0.9(0.9(0.9v_0 + 0.1\nabla w_0^2) + 0.1\nabla w_1^2) + 0.1\nabla w_3^2 \\ &= 0.9^3 v_0 + 0.9^2 \cdot 0.1 \nabla w_0^2 + 0.9 \cdot 0.1 \nabla w_1^2 + 0.1 \nabla w_3^2 \\ &\approx 0.1 \sum_{i=0}^3 (0.9)^i \nabla w_i^2 \end{aligned}$$

$\therefore v_t = (1-\beta) \sum_{i=0}^{t-1} \beta^i (\nabla w_i)^2$, we see that as t increases, v_t increases and it is not until $i=0$ that the contribution from the early gradients decays to 0.

In steep areas ∇w_i 's are large hence v_t also becomes large and the learning rate reduces. On the other hand, In gentle areas, ∇w_i '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.

This makes the effective learning rate constant $\frac{\eta}{\sqrt{v_t}}$, 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:-

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

$$\Delta w_t = -\sqrt{\frac{v_{t-1}}{v_t}} \nabla w_t$$

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

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

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

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

2) In ~~flat~~ 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. ($v_t \neq v_t$)

\Rightarrow 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 to stabilize the update. and repeat from $(\beta_1)^t$ for m_t

$m_t = \beta_1 m_{t-1} + (1-\beta_1) \nabla w_t$ where β_1 is a constant for scaling/bias removal. for $\hat{m}_t = \frac{m_t}{(1-\beta_1)^t}$ and $\hat{v}_t = \frac{v_t}{(1-\beta_2)^t}$ (explained later)

then correcting the bias: $v_t = \beta_2 v_{t-1} + (1-\beta_2) (\nabla w_t)^2$

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

$$\hat{v}_t = \frac{v_t}{(1-\beta_2)^t} = \frac{v_t - \hat{v}_{t-1}}{(1-\beta_2)^t} = \hat{v}_{t-1} + \frac{\beta_2}{(1-\beta_2)^t} v_t = \hat{v}_{t-1} + \frac{\beta_2}{(1-\beta_2)^{t-1}} v_t = \hat{v}_{t-1} + \beta_2 v_t$$

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

In this momentum, we give more weightage to overall previous gradients in the form of $\beta_1 m_{t-1}$, the gradient is given 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

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

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

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

$E[m_t] \approx E[\nabla w]$, therefore $\frac{m_t}{1-\beta_1}$ gives an unbiased estimate of ∇w .

\Rightarrow 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.

Max Prop :- If we closely revisit the formula for RMSPROP:-

$$v_t = \beta v_{t-1} + (1-\beta)(\nabla w_t)^2, \quad w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t + \epsilon}} \nabla w_t$$

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 GD 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.

\Rightarrow 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:-

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

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

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 Adam :- start from scratch~~

Introducing Max norm in Adam :- Adamax

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

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

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

NADAM

Introducing Nesterov Acceleration in Adam :-

Slightly different notation (did not understand)

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

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

$$v_{t+1} = (\beta_2 v_t + (1-\beta_2) (\nabla w_t)^2)^{1/2} = \frac{v_{t+1}}{1-\beta_2^{t+1}}$$

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

Learning rate Schemes

<u>Based on epochs</u>	<u>Base Validation</u>	<u>Based on gradients</u>
1) Step Decay	1) Line Search	1) AdaGrad
2) Exponential Decay	2) Log Search	2) RMSProp
3) Cyclical	3) Adam	3) Adadelta
4) Cosine Annealing	4) Adam	4) AdamW
		5) Adamax
		6) Nadam
		7) AMSGrad
		8) Adam

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 gradients, then there is a chance to escape the saddle points. Adaptive learning rate helps in escaping saddle points but they are computationally expensive.

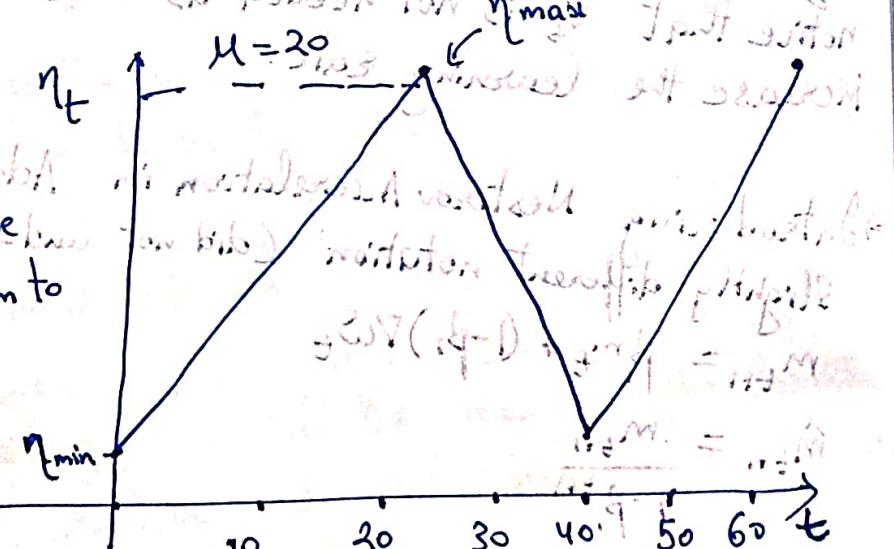
We can vary learning rate cyclically as a simple strategy.

i) Triangular :-

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

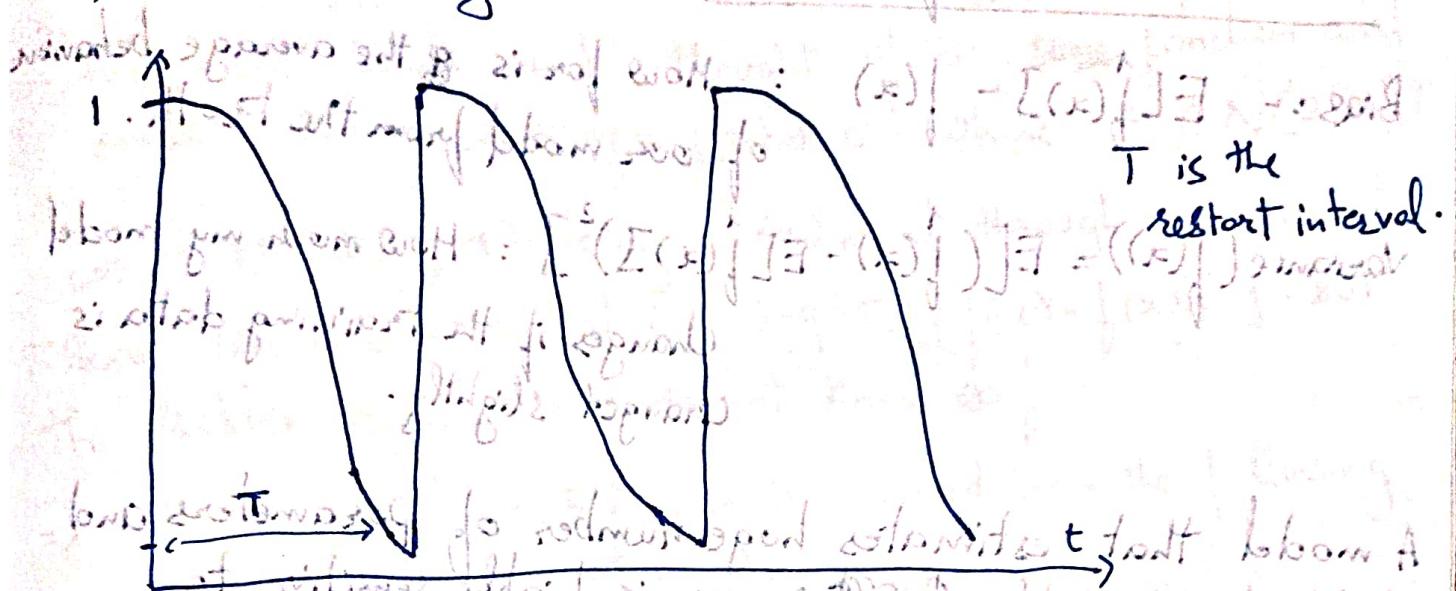
Formula is given by:-

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



$$\left(\frac{2L+1-t}{20} \right) + 1 = \frac{2L+1-t+20}{20} = \frac{21-t}{20}$$

ii) Cosine Annealing (Warm-restart) ↳ empirical basis



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

Warm Start (Yann LeCun) found to be helpful in transformed 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 it to learn and converge better. This is called Warm Start.

the model to warm and converge better. 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 behavior 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 slightly different sample of training data, it will learn significantly different 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. If 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.

\Rightarrow Similarly, a model that estimates very few number of parameters. Thus even if the training data sample is slightly 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 $f(x)$ function and some noise. So, $y = f(x) + \epsilon$ where $\epsilon \sim N(0, \sigma^2)$

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

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

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

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

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

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

ϵ is independent from $(\hat{f}(x) - f(x))$. So $E[(\hat{f}(x) - f(x))^2] = \frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2 - \bar{\epsilon}^2$ 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 $E[\epsilon_i | \mathcal{D}]$ is not independent from $f(x) - \hat{f}(x)$. Hence $E[\epsilon_i^2 | \mathcal{D}] \neq 0$. The true error LHS is larger than the error.

So here we want $\sum_{i=1}^n E[(y_i - \hat{y}_i)^2]$.

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: $E\left[\sum_{i=1}^n \epsilon_i (\hat{f}(x_i) - f(x_i))\right] = \frac{1}{n} \sum_{i=1}^n \frac{\partial f(x_i)}{\partial y_i}$

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

So LHS of Stein's 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.

The complex the model, the modified objective function

so it is better to minimize a modified objective function

that also incorporates model complexity into it.

$\min_{\theta} L(\theta) = L(\text{train error}) + \Omega(\theta)$. But instead of calculating

$L(\theta)$ given by Stein's lemma, $\Omega(\theta)$ is given by

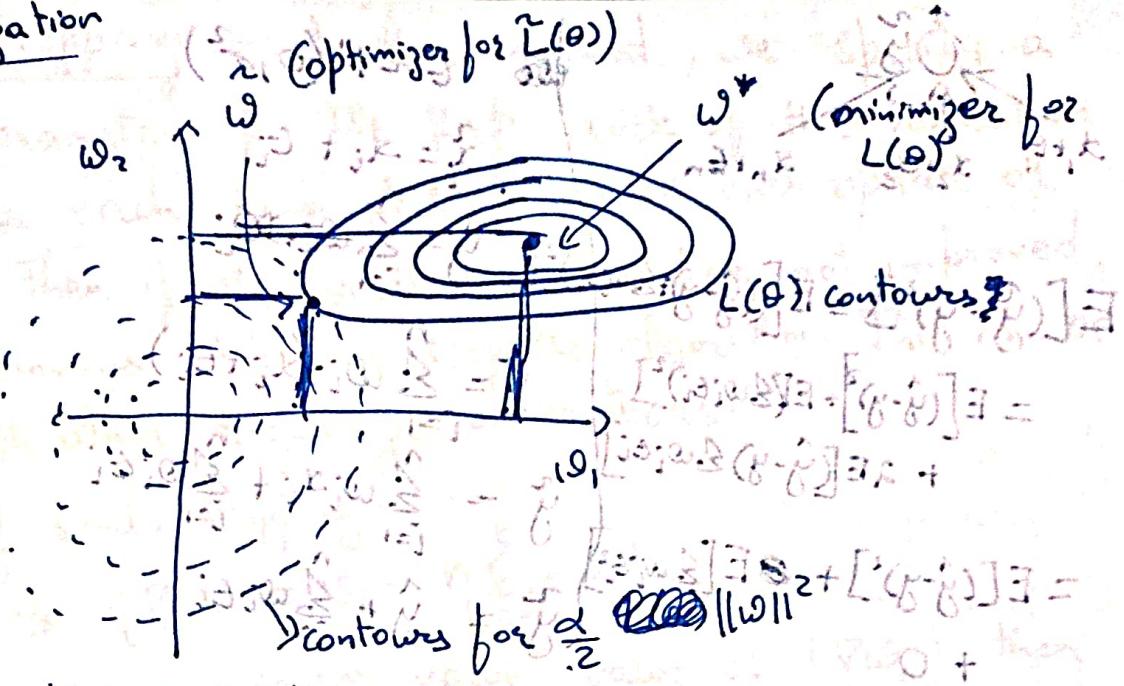
We can calculate it using an approximation that captures model complexity. This strategy is called regularization.

we denote $L(\theta) = L((\theta)(x))$, where

we have $L(\theta) = L((\theta)(x))$. This suffices if $f(x)$ is linear.

we have $L(\theta) = L((\theta)(x))$. This suffices if $f(x)$ is non-linear.

i) L₂ regularization



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 shrunk 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, corrupt 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.

$$\begin{aligned}
 & \text{Diagram showing } \hat{y} = \sum_{i=1}^n w_i x_i + b \\
 & \text{Inputs: } x_1, x_2, \dots, x_n \\
 & \text{Weights: } w_1, w_2, \dots, w_n \\
 & \text{Bias: } b \\
 & \text{Error: } E[(\hat{y} - y)^2] = E[((\hat{y} - y) - \sum_{i=1}^n w_i e_i)^2] \\
 & = E[(\hat{y} - y)^2] + E[\sum_{i=1}^n w_i e_i]^2 \\
 & + 2E[(\hat{y} - y) \sum_{i=1}^n w_i e_i] \\
 & = E[(\hat{y} - y)^2] + E[\sum_{i=1}^n w_i^2 e_i^2] \\
 & + 0
 \end{aligned}$$

This is because $E[(\sum_{i=1}^n w_i e_i)^2] = E[\sum_{i=1}^n w_i^2 e_i^2]$
 since e_i is independent from e_j and $E[e_i e_j] = E[e_i] E[e_j] = 0$
 So, all interacting terms in LHS will be 0.
 Hence only square term will survive.
 Also e_i is independent from $(\hat{y} - y)$ so $E[(\hat{y} - y) e_i] = 0$
 $= E[(\hat{y} - y)^2] + \sum_{i=1}^n w_i^2 E[e_i^2]$

$$\begin{aligned}
 & \text{So } E[(\hat{y} - y)^2] = L(\theta) + E[\sum_{i=1}^n w_i^2 e_i^2] \\
 & E[(\hat{y} - y)^2] = L(\theta) + \sum_{i=1}^n w_i^2
 \end{aligned}$$

Instead of fitting on the true labels, we inject 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.

It adds a regularization term to the loss function, which encourages the model to have smaller weights.

v) Early Stopping: In this method, we keep specifying 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:-
 $\omega_{t+1} = \omega_t - \eta \nabla \omega_t$ is a good approximation based on (s) $\omega_{t+1} = \omega_t - \eta \sum_{i=0}^t \nabla \omega_i$ is a good approximation based on (s) ω_{t+1} is stable based on value of $|\nabla \omega_i|$, then let T be the maximum value (absolute)

of t such that $|\nabla \omega_i| \leq \eta T$ this step controls the $|\omega_{t+1} - \omega_0| \leq \eta T$ this step controls the $|\omega_{t+1} - \omega_0|$. If we limit t , in some upper bound of $|\omega_{t+1} - \omega_0|$. If we limit t , in some sense we are limiting $|\omega_{t+1} - \omega_0|$ and if $\omega_0 = 0$, 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 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 propagation using the weights from previous iterations.
- 3) Do backward propagation and update the weights that are active.
- 4) Repeat the procedure. We can decide if we want to keep one network or all. 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.

At test time, the output is calculated from the whole network but the output of each neuron can 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 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 neurons. If we ignore the rest of the network and set up a new network such that the task of this network is to reconstruct the input layer with the least error possible with only the 1st hidden layer consisting of the neurons from 1st hidden layer of the original network.

So the objective is such that:-

$$\frac{1}{n} \sum_{i=1}^n \sum_{j=1}^m (\hat{x}_{ij} - x_{ij})^2$$

Each input is R^n , and there are n such inputs. The reconstructed output is also R^n . This is VPCA, the objective is same. Whatever weights are learned as W_1 (between input to 1st hidden layer) can be used as 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 R^m , and there are n such inputs.

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

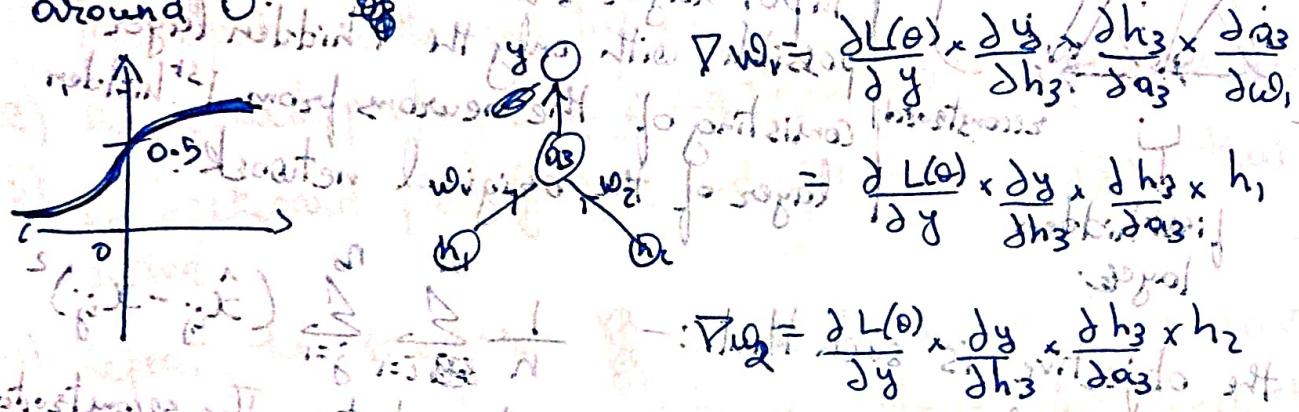
h_{ij} is the output from the j^{th} neuron of the i^{th} point of x . \hat{h}_{ij} is the output from the j^{th} neuron of the i^{th} point of the 1st hidden layer. From this we obtain W_2 .

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 creating 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.



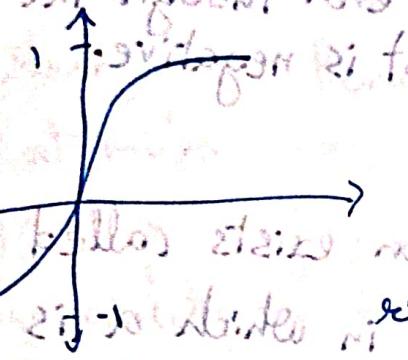
Since neurons are sigmoid, h_1 and h_2 are positive. Besides h_1 and h_2 , $\frac{\partial w_1}$ and $\frac{\partial w_2}$ are same. Hence both $\frac{\partial w_1}$ and $\frac{\partial w_2}$ will have the same sign. Second and fourth quadrant gradients cannot be easily calculated as signs of the components

$\frac{\partial w_2}{\partial x}$ (Allowed) are opposite. It is mitigated by centering w_2 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.

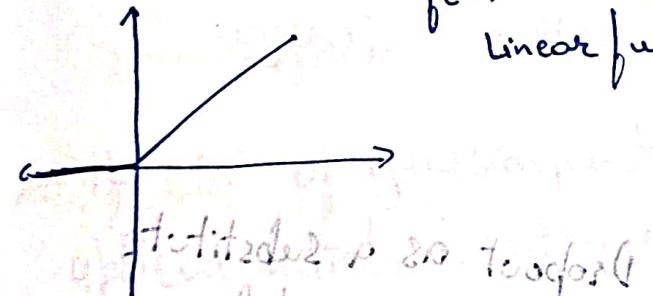
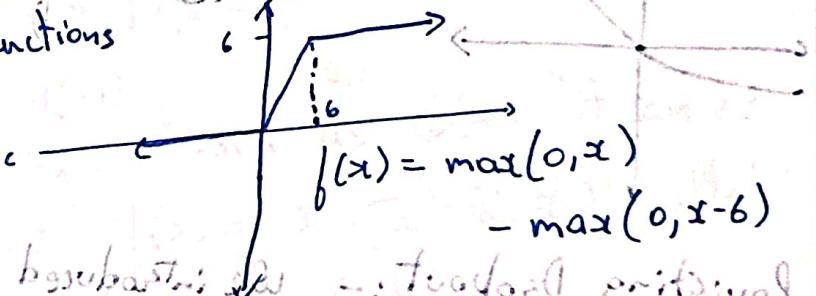
3) Tanh function: $\tanh(x)$. without b it's called hyperbolic tangent function.
 $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$



compresses all its input to the range $[-1, 1]$. It is zero centred so it removes the opposite sign gradient. But the problem of vanishing gradient still exists. And it also involves exponential computations, so it is also relatively expensive.

4) ReLU (Rectified Linear Unit)

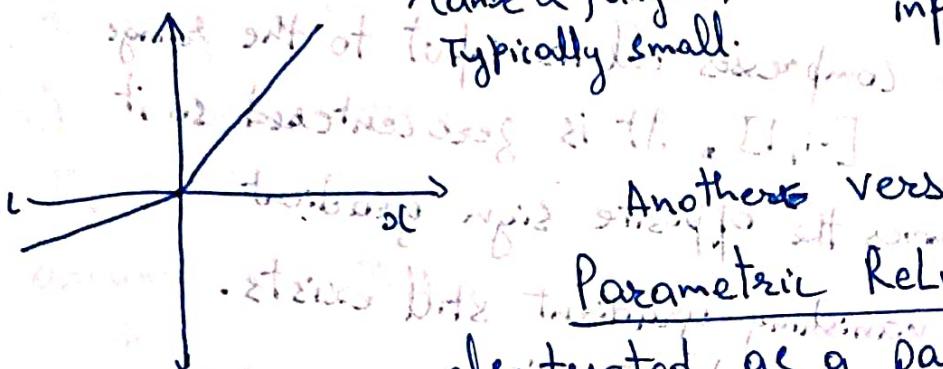
$f(x) = \max(0, x)$ We can recover piecewise linear functions

Advantages of ReLU:
 Does not saturate in the positive region, computationally efficient, since it converges much faster than sigmoid/tanh.
 $\frac{\partial \text{ReLU}(x)}{\partial x} = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$

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

Leaky Relu : No Saturation. (Without : without shift) Some gradient value still exists even though the input is negative.

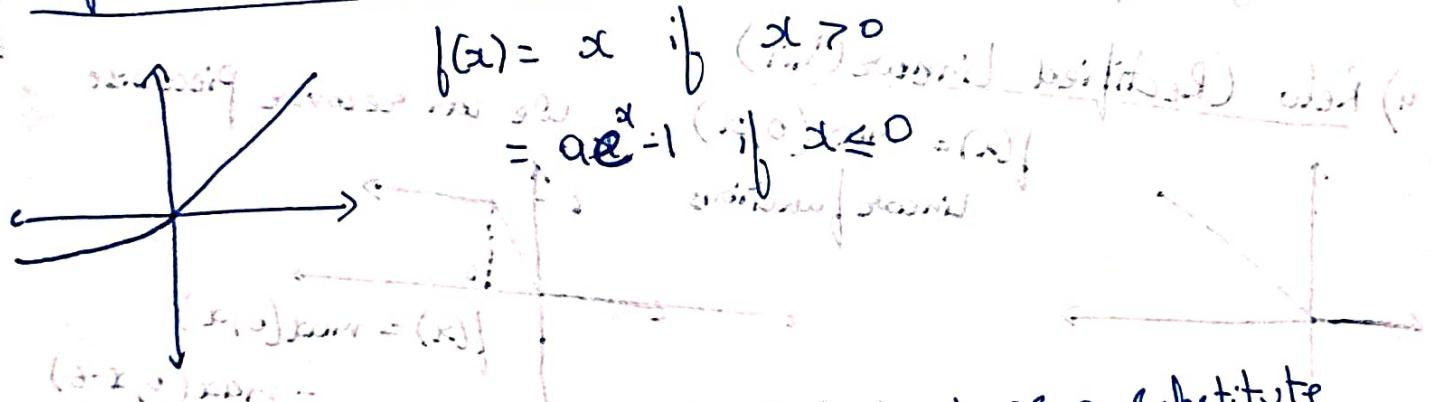


Another version exists called Parametric Relu in which α is

also treated as a parameter, which gets

updated during backpropagation.

Exponential Linear Unit



Revisiting Dropout :- We introduced Dropout as a substitute to bagging. But in bagging, each individual model is overfit on 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. $\Rightarrow \max \{w_1 x + b_1, \dots, w_n x + b_n\}$. Only the neuron that contributes the most gets the update, as for others, gradient is 0.

Dropouts can also be written as $O(x) = \begin{cases} 1 \cdot x & \text{if } \text{selected} \\ 0 \cdot x & \text{if not selected} \end{cases}$. 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 $[0, 1]$: Fair dropout most learned

$\phi(x) = p$ works, as each neuron will be selected with probability of p .

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

$$E[f(x)] = E[m \cdot x] = x \cdot E[m]$$

$$\text{Therefore } E[f(x)] = x \left[1 \times \phi(x) + 0 \times \frac{1 - \phi(x)}{1 - \phi(x)} \right]$$

$$(E[f(x)]) = x \phi(x) \rightarrow P(X \leq x)$$

if CDF of gaussian function is chosen as $\phi(x)$, it can be approximately written as $f(x) = x \cdot \sigma(1.702x)$ called GelU.

SELU (Scaled Exponential Linear Unit): Linear Unit

with output scaled such that mean is 0 and unit variance.

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

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

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

$$x \cdot \phi(x) = (x)_\phi \rightarrow \text{Handwritten}$$

where $\phi(x)$ makes $(x)_\phi$ differentiable or make

Convolution operation: If I is inserted in ϕ - form to K

$$\begin{bmatrix} m_1 \\ m_2 \end{bmatrix} \quad \begin{bmatrix} n_1 \\ n_2 \end{bmatrix}$$

$$S_{ij} = (I * K)_{ij} = \sum_{a=-\frac{m}{2}}^{\frac{m}{2}} \sum_{b=-\frac{n}{2}}^{\frac{n}{2}} I_{i-a, j-b} \cdot K_{\frac{m}{2}+a, \frac{n}{2}+b} \cdot \phi \text{ for } g_t \text{ is derivative}$$

position, position of I with K is g_t as per ϕ . In case of 3-D, 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. $E_{out, 3D} = [C_{ij}]_3$

Defining Quantities :-

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

\Rightarrow The stride S (increment in the slide of the kernel)

\Rightarrow The number of filters K is, K is misspelled as F .

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

\Rightarrow The output is $W_2 \times H_2 \times D_2$.

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, the pixels in the border of the input (convolution is not defined).

The formula for output dimensions is given as:-

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

$$H_2 = H_1 - F + 1$$

This is because if $F = 2a+1$ for $a \geq 0$,
 a columns are lost in the left border and a columns are
lost in the right border of the input. \therefore total lost columns = $2a = F-1$.
So 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 updated as follows:

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

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

Generally we do $s=1$ which is to 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 then 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 applying the algorithms 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 feed-forward Network

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

But in a CNN:-

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

i) The input image is flattened :-

Hence input layer has 16 neurons.

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

$$\Rightarrow [1|2| \dots |15|16]$$

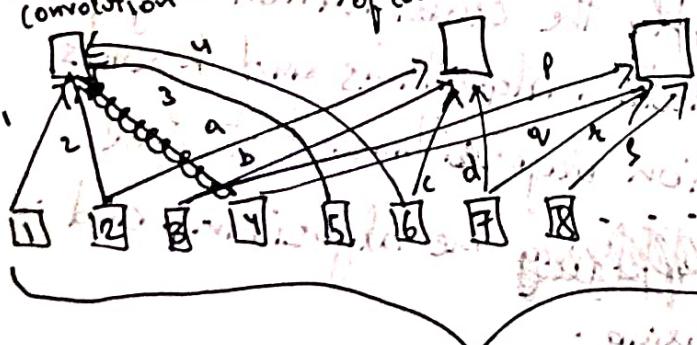
ii) The kernel is flattened :-

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 pixels 1, 2, 5 and 6 of input.

Similarly for all pixels. So the connectivity is sparse.

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

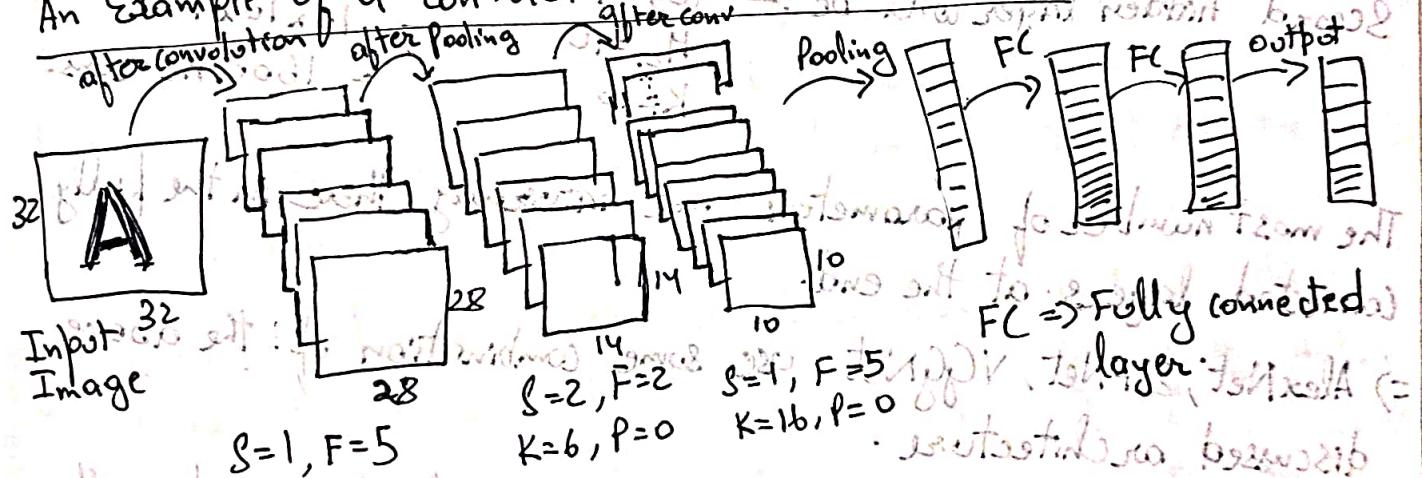
There is weight sharing in this neural network. More kernels can be applied but that also increases the number of output neurons (next layer convolution output, addition in depth) so weight sharing is still there.

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

ii) We don't really lose information as in the next layers, there will be interaction between pixels which depend on the input pixels. So indirect interaction 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 such filters, so there are $6 \times 25 = 150$ 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 $28 \times 28 \times 6$ neurons in the first hidden layer. This is decided by the formula discussed before.

The Layer after first hidden layer is the max pooling layer. In this layer is passed through a max filter independently to obtain the 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 neighbourhood 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 2400. The number of neurons in the second hidden layer will be :-



$$W_2 = 14 - 5 + 1 = 10 \\ H_2 = 10 \\ K = 16 \\ = 10 \times 10 \times 16 = 1600 \text{ neurons.}$$

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 fixed 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.

$H \rightarrow$ The $(1 \times 1) \times D$ convolution operation applies across the depth of the input.



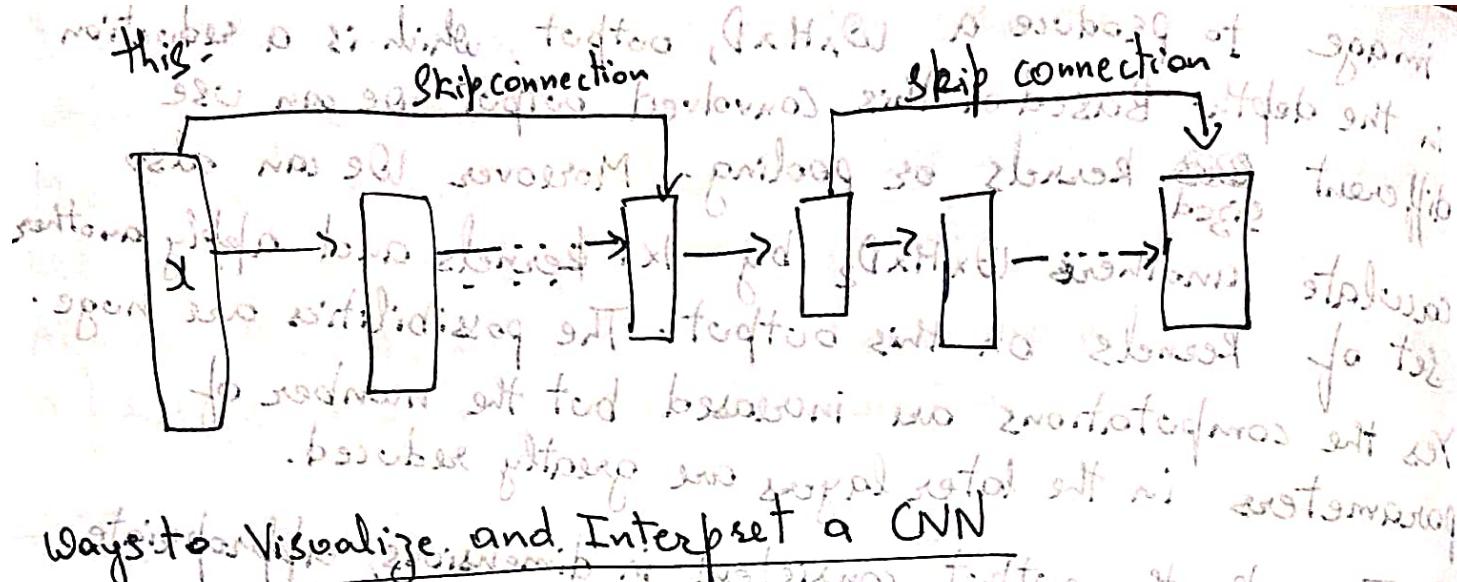
=> We can apply D such 1×1 kernels to the input

image to produce a $W \times H \times D$, 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 $W \times H \times D_2$ 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 $E \times F \times D$ output, then average pooling reduces it to 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. In fact, ResNet did



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 patch in the image which causes these neurons to fire.
This will allow us to understand which neuron(s) are fired 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 gradients 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 white pixels will appear in the output of this step with respect to that neuron. To calculate these gradients we can use backpropagation. 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 ~~area~~ 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 backpropagation. 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 backpropagation step to calculate the gradient matrix 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 prevent 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. The reason is we have a need of the argmax ($-L(\theta)$)

$$\text{argmax} (\text{Score}(I) - \lambda \cdot \mathcal{R}(I))$$

Now doing with the score, first is the loss for pixel with respect to the softmax. where $\text{Score}_c(I)$ = score for class before softmax and $\mathcal{R}(I)$ = some regularizer to ensure that I looks

real-like and image is shriveling. Now it is better with this. 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. Now at

=> Set the score vector to be $[0, 0, \dots, 100, 0, 0, 0]$ better with this.

=> ~~Do a forward pass~~ Do a forward pass.

=> Compute the gradients $\frac{\partial L(\theta)}{\partial I_{i,j}}$ for all neurons has been activated

=> Update the pixel $I_{i,j} = I_{i,j} - \eta \frac{\nabla_i L(\theta)}{I_{i,j}}$.

=> Repeat the process from forward pass if it is not converged.

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 σ in layer of interest to all zero, except

. before the neuron of interest. σ is sigmoid function of

=> Backprop to image

=> $I_{i,j} = I_{i,j} - \eta \frac{\nabla L(\theta)}{I_{i,j}}$, where $L(\theta)$ can be negative of

the pre-activation of the neuron of interest. η is step size of gradient descent.

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 the loss function is the squared error between the embeddings of the original image and the created image.

$$L(I) = \|\phi(I) - \phi(I_0)\|^2 + \lambda \|\phi(x)\|_2^6$$

- $\phi(I)$ = vector of embeddings of the created image.
 - $\phi(I_0)$ = vector of embeddings of the original image.
 - $I[i, j] = I[i, j] - \eta L(I)$
- \Rightarrow This can be done for embeddings obtained from any layer.
- \Rightarrow 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 neuron fires even more. Our objective becomes $\max L(I)$.

Where h_{ij} is the output of the j^{th} neuron of the i^{th} layer. Where $L(I) = h_{ij}^2$. We can repeat the same procedure of forward propagation and back propagation and update the image pixels $I[i, j] = I[i, j] + \eta \nabla_{I[i, j]} L(I)$.

• 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 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:-

$$L_{\text{content}}(\vec{p}, \vec{x}) = \sum_{ijk} (p_{ijk} - x_{ijk})^2 \quad \text{where } \vec{p} \text{ is the embeddings of the content image. } \vec{x} \text{ is the embeddings of the created image.}$$

This is done for some layers of the created image. This is done for some layers and the formula is written in the tensor notation where the final 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 $VT \in \mathbb{R}^{164 \times 64}$ captures the style of the image.

The deeper layers capture more of this style information.

The objective can be the following:-

$$L_{\text{style}} = E_e = \sum_{ij} (G_{ij} - A_{ij})^2, \text{ where } G \text{ is the Gram matrix}$$

of the style image computed at layer L; and A is similarly for the created image.

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

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. It samples little random noise & expect to get this noise from background as very high dimensional entities. 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 is this space just a vector, and it just passes that vector through a function which has its decision boundary. Even though the image is noisy to us, the CNN predicts it according to the output of the function.

This is because the function is not continuous. Therefore it is not differentiable at every point.