End Term Assignment

Mathematics for Deep Learning

Question – 1 - Singular Value Decomposition (SVD)

Give a rough idea of how SVD could be used for the compression of information like images. Make your own example and show how it is possible.

Answer – Singular Value Decomposition (SVD) is one of the fundamental linear algebra techniques which is used for the decomposition of the given information into matrices, keeping only the important image information and reducing the size of the other non – important features of the image, which ensures the smooth and faster transfer of the data in any network systems.

The SVD technique converts all the information into three matrices: -

$$A=U\Sigma VT$$

- A: - It is the general core image which we are compressing.
- U: - It is A $m \times m$ → (A square Matrix) orthogonal matrix whose columns are the left singular vectors of A.
  It is representing maximum variance along the rows of A, in terms of image it captures all the vertical patterns and stores important row – wise information.
- Σ: - A diagonal $m \times n$ matrix containing the singular values of A in descending order.
  It represents the importance of each of the component contained in the image.
- V^T: - The transpose of an $n \times n$ orthogonal matrix, where the columns are the right singular vectors of A.

All the three components get combined to get back the original information. The U matrix rotates the information, Σ it scales the information and V^T rotates the information once again.

Example: -

Let us consider one of the matrix A (*The information contained in the image is contained in pixels and all the pixel values could be converted into a matrix*):

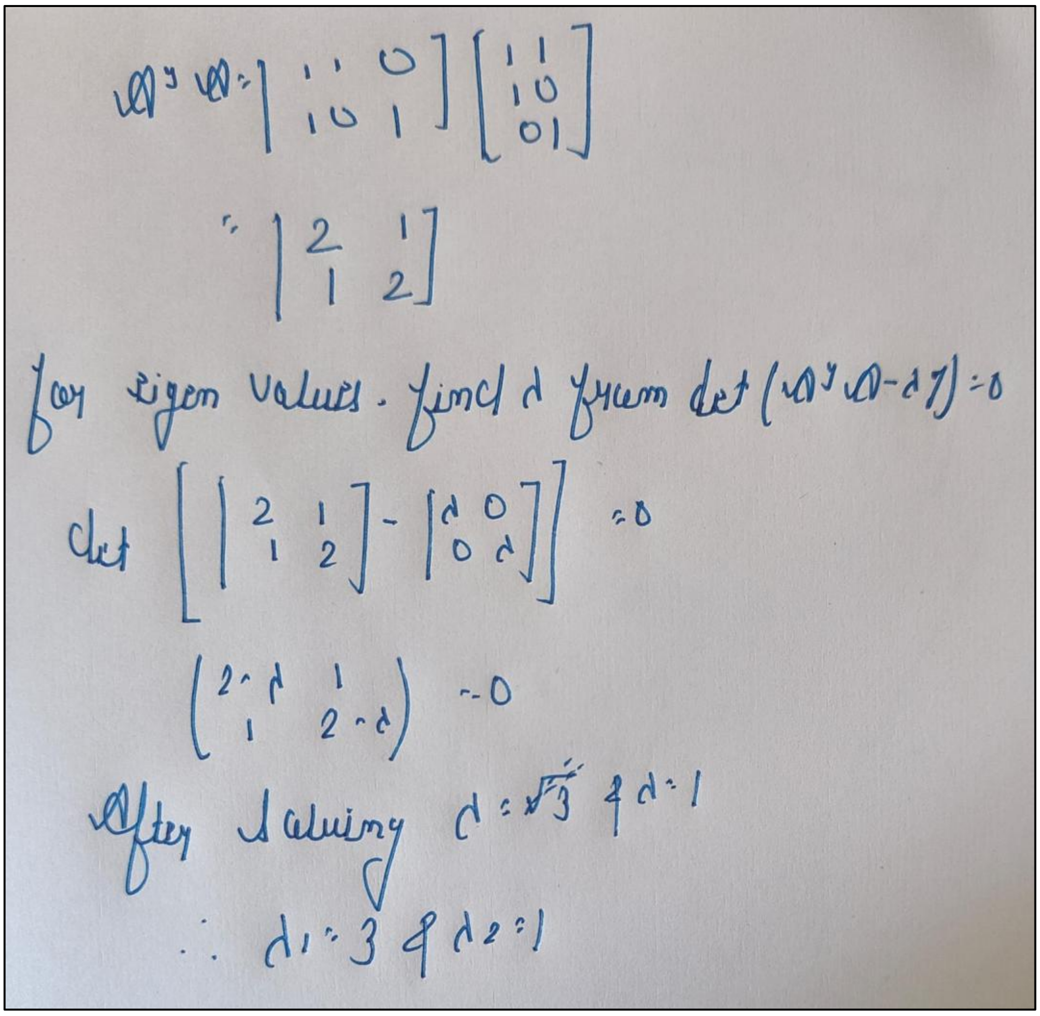
$$A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix};$$

Now, we have to decide the order of each of the three matrices:

A is of (3x2), since U and V^T are square matrices, their order is (3x3) and (2x2) respectively. And for the middle matrix Σ the order is (3x2).

Let us first find the matrix $\Sigma = \begin{bmatrix} \sigma 1 & 0 \\ 0 & \sigma 2 \\ 0 & 0 \end{bmatrix}$.

For $\sigma 1$ and $\sigma 2$ we have to find the Eigen values of the matrix A^TA.



$A^T A = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$

$= \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$

for eigen values. find λ from $\det(A^T A - \lambda I) = 0$

$\det\left[\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} - \begin{bmatrix} \lambda & 0 \\ 0 & \lambda \end{bmatrix}\right] = 0$

$\begin{pmatrix} 2-\lambda & 1 \\ 1 & 2-\lambda \end{pmatrix} = 0$

After solving $\lambda = \sqrt{3}$ & $\lambda = 1$

$\therefore \lambda_1 = 3$ & $\lambda_2 = 1$

The larger eigen value is chosen as the first value for σ1.

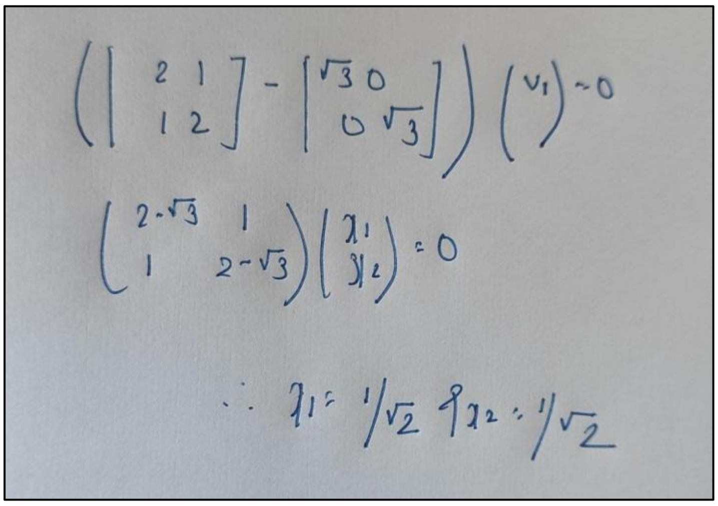
$\sigma 1 = \sqrt{3}$ and $\sigma 2 = 1$

Therefore, $\Sigma = \begin{bmatrix} \sqrt{3} & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}$.

Now, V = [V1 V2].

V1 and V2 could be find by using the equation (A^TA – λi I)Vi = 0.

By equating the values and solving



$$\left(\begin{bmatrix}2 & 1\\ 1 & 2\end{bmatrix} - \begin{bmatrix}\sqrt{3} & 0\\ 0 & \sqrt{3}\end{bmatrix}\right)\begin{pmatrix}v_1\end{pmatrix} = 0$$

$$\begin{pmatrix}2-\sqrt{3} & 1\\ 1 & 2-\sqrt{3}\end{pmatrix}\begin{pmatrix}x_1\\ x_2\end{pmatrix} = 0$$

$$\therefore x_1 = 1/\sqrt{2} \;\&\; x_2 = 1/\sqrt{2}$$

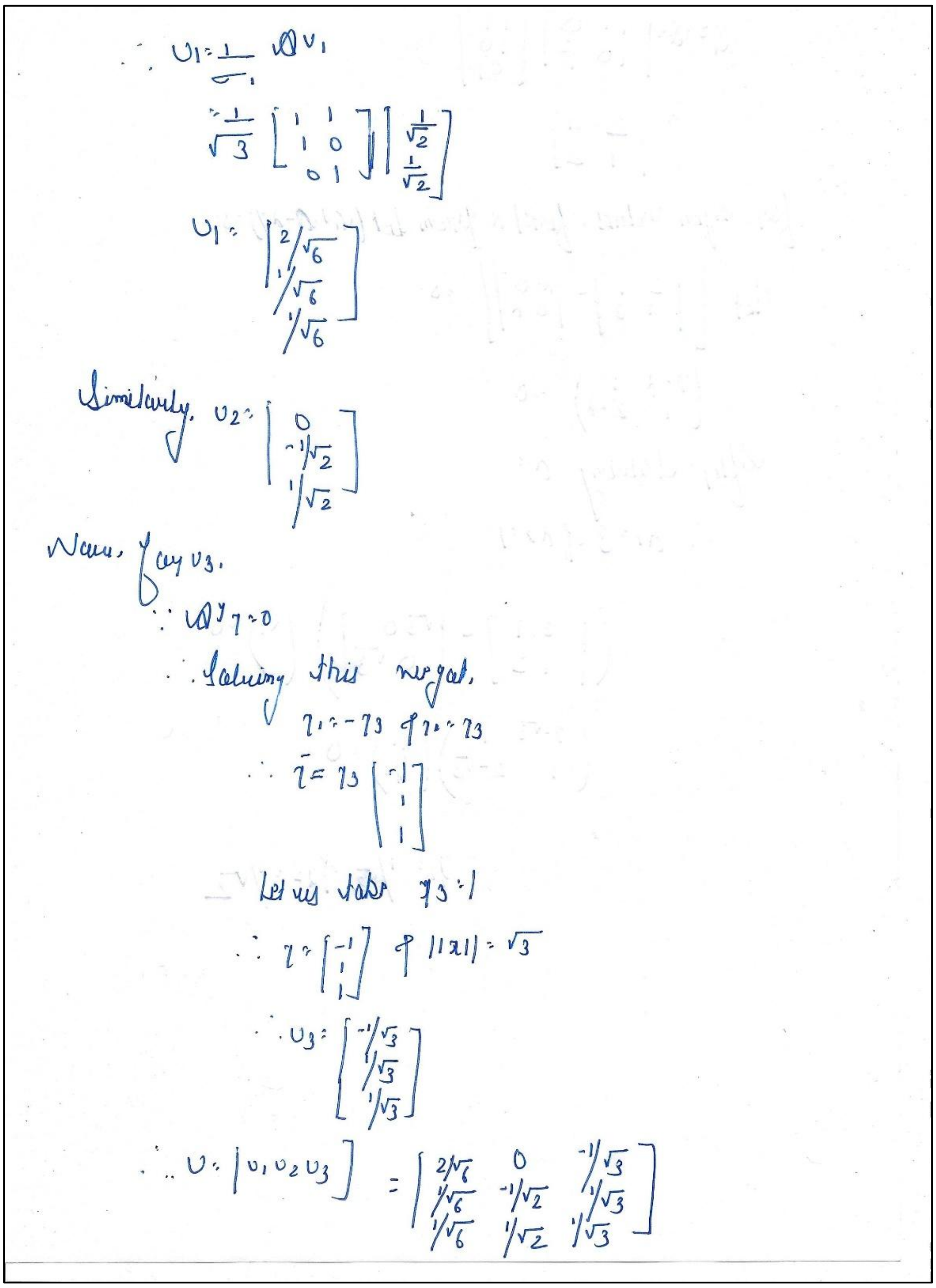
Therefore $V_1 = \begin{bmatrix}1/\sqrt{2}\\ 1/\sqrt{2}\end{bmatrix}$. Similarly, $V_2 = \begin{bmatrix}-1/\sqrt{2}\\ 1/\sqrt{2}\end{bmatrix}$.

Therefore V = [V1 V2] = $\begin{bmatrix}1/\sqrt{2} & -1/\sqrt{2}\\ 1/\sqrt{2} & 1/\sqrt{2}\end{bmatrix}$ and $V^T = \begin{bmatrix}1/\sqrt{2} & 1/\sqrt{2}\\ -1/\sqrt{2} & 1/\sqrt{2}\end{bmatrix}$

And now, for the calculation of U = [U1 U2 U3].

$U_i = (1/\ \sigma_i) \times (A.V_i)$

By using the formula and equating the values, we got



∴ $u_1 = \frac{1}{\sigma_1} A v_1$

$= \frac{1}{\sqrt{3}} \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$

$u_1 = \begin{bmatrix} 2/\sqrt{6} \\ 1/\sqrt{6} \\ 1/\sqrt{6} \end{bmatrix}$

Similarly, $u_2 = \begin{bmatrix} 0 \\ -1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}$

Now, for u_3.

∵ $A^T x = 0$

∴ Solving this we get,

$x_1 = -x_3$ & $x_2 = x_3$

∴ $x = x_3 \begin{bmatrix} -1 \\ 1 \\ 1 \end{bmatrix}$

Let us take $x_3 = 1$

∴ $x = \begin{bmatrix} -1 \\ 1 \\ 1 \end{bmatrix}$ & $\|x\| = \sqrt{3}$

∴ $u_3 = \begin{bmatrix} -1/\sqrt{3} \\ 1/\sqrt{3} \\ 1/\sqrt{3} \end{bmatrix}$

∴ $U = \begin{bmatrix} u_1 & u_2 & u_3 \end{bmatrix} = \begin{bmatrix} 2/\sqrt{6} & 0 & -1/\sqrt{3} \\ 1/\sqrt{6} & -1/\sqrt{2} & 1/\sqrt{3} \\ 1/\sqrt{6} & 1/\sqrt{2} & 1/\sqrt{3} \end{bmatrix}$

$$\therefore A = U\Sigma V^T$$

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} = \underbrace{\begin{bmatrix} 2/\sqrt{6} & 0 & -1/\sqrt{3} \\ 1/\sqrt{6} & -1/\sqrt{2} & 1/\sqrt{3} \\ 1/\sqrt{6} & 1/\sqrt{2} & 1/\sqrt{3} \end{bmatrix} \begin{bmatrix} \sqrt{3} & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ -1/\sqrt{2} & 1/\sqrt{2} \end{bmatrix}}$$

Thus, through SVD we have compressed the information into three Matrices, which are containing the core information.

Question – 2 - Dimensionality Reduction

Reduce dimension from 2 to 1. You are free to use the internet for this problem but justify what you have done based on what was taught in class.

Answer – In this question we have to use the Application of Principal Component Analysis (PCA). Principal Component Analysis is a statistical method for the reduction of the higher dimension, it converts high dimensional information into smaller ones (*which is contained of most of the original information),* for their easier transmission.

Feature	Example - 1	Example - 2	Example - 3	Example - 4
X_1	4	8	13	7
X_2	11	4	5	14

Mean of X_1 = $\overline{X1}$.

Mean of X_1 = $\overline{X2}$.

$$X_1 = \frac{1}{4}(4+8+13+7) = 8$$

$$X_2 = \frac{1}{4}(11+4+5+14) = 8.5$$

Now, we need to calculate the Covariance Matrix,

$$C = \begin{bmatrix} \mathrm{cov}(X_1, X_1) & \mathrm{cov}(X_1, X_2) \\ \mathrm{cov}(X_2, X_1) & \mathrm{cov}(X_2, X_2) \end{bmatrix}$$

$$\therefore \operatorname{Cov}(X_1, X_1) = \frac{1}{N-1} \sum_{k=1}^{N} \underbrace{(X_{1k} - \bar{X}_1)(X_{1k} - \bar{X}_1)}$$

$$= \frac{1}{4-1} \left((4)^2 + (0)^2 + (5)^2 + 1^2 \right)$$

$$= \frac{1}{3} (16 + 25 + 1) = 14$$

$$\operatorname{Cov}(X_1, X_2) = \frac{1}{N-1} \sum_{k=1}^{N} (X_{1k} - \bar{X}_1)(X_{2k} - \bar{X}_2)$$

$$= \frac{1}{4-1} \left((-4) \times (2.5) + 0 \times (-3.5) + 5(-3.5) + (-1)(5.5) \right)$$

$$= -33/3 = -11$$

Similarly,

$$\operatorname{Cov}(X_2, X_1) = -11$$

$$\operatorname{Cov}(X_2, X_2) = \frac{1}{N-1} \sum_{k=1}^{N} (X_{2k} - \bar{X}_2)(X_{2k} - \bar{X}_2)$$

$$= 1/3 \left((2.5)^2 + 4.5^2 + 3.5^2 + 5.5^2 \right)$$

$$= 23$$

$$\therefore C = \begin{bmatrix} 14 & -11 \\ -11 & 23 \end{bmatrix}$$

Now, we need to Calculate the Eigen Value of the Covariance Matrix C,

$$\therefore \det(C-\lambda I)=0$$

$$\begin{vmatrix} 14-\lambda & -11 \\ -11 & 23-\lambda \end{vmatrix} = 0$$

$$\therefore \lambda^2 - 37\lambda + 201 = 0$$

$$\therefore \lambda_{1,2} = \frac{1}{2}\left(37 \pm \sqrt{565}\right)$$

$$\therefore \lambda_1 = 30.3849 \ \& \ \lambda_2 = 6.6151$$

Now, we need to calculate the Eigen Vector,

$$\therefore V = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

$$\& \ (C-\lambda I)V = 0$$

$$\begin{bmatrix} 14-\lambda & -11 \\ -11 & 23-\lambda \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \text{ —(*)}$$

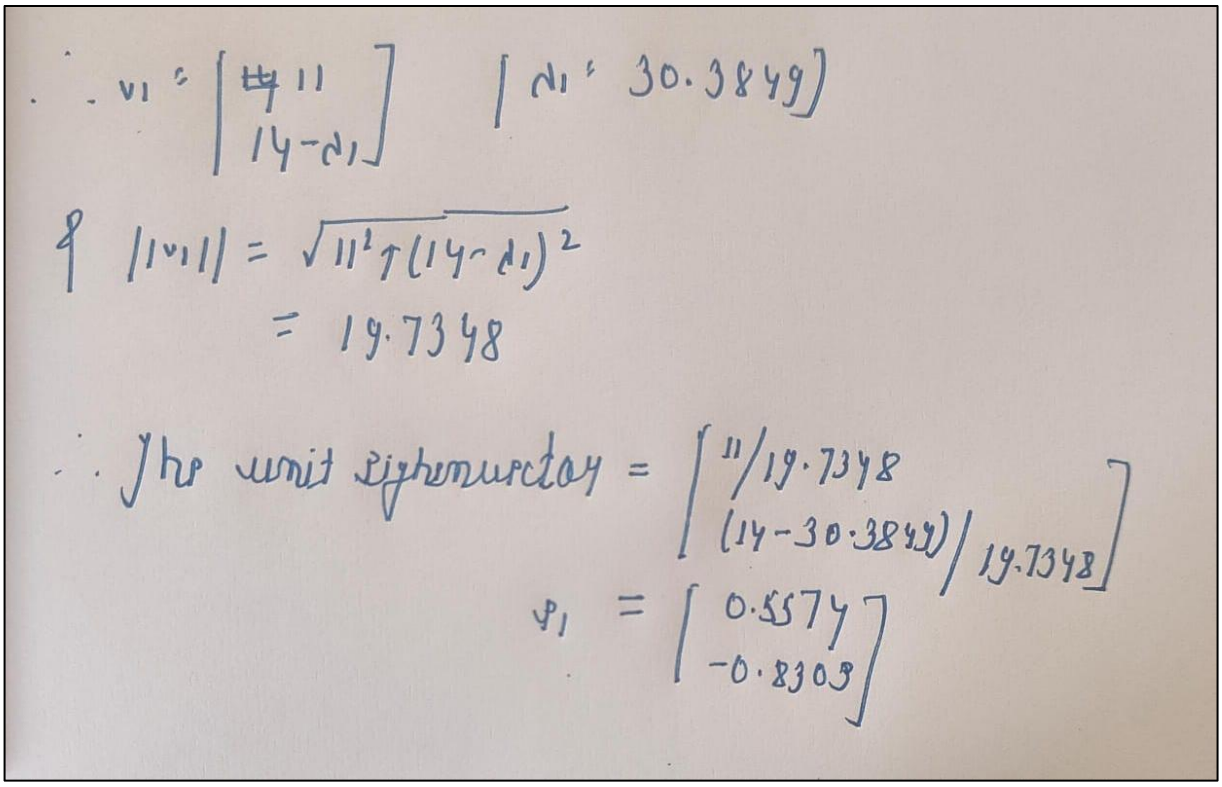
After Solving (*) we got our Eigen Vector,

Let the Eigen Vector be V,

Therefore, V = $\begin{bmatrix} V1 \\ V2 \end{bmatrix}$

V = $\begin{bmatrix} 11 \\ 14 - \lambda \end{bmatrix}$.

So, as we have learned that in the conversion from higher dimension to lower dimension, we have to choose such an axis which is consist of maximum of the information i.e., why larger Eigen Value is chosen for the further calculation and Analysis.



$\therefore v_1 = \begin{bmatrix} \cancel{4}\ 11 \\ 14-\lambda_1 \end{bmatrix}$ $[\lambda_1 = 30.3849]$

$\& \ ||v_1|| = \sqrt{11^2 + (14-\lambda_1)^2}$

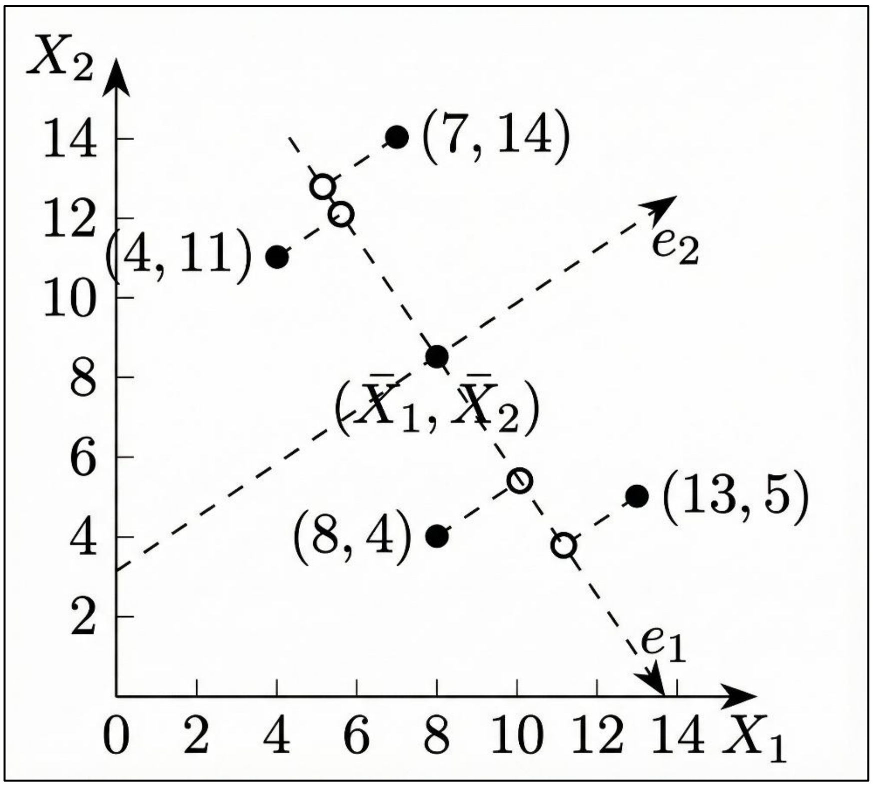
$= 19.7348$

$\therefore$ The unit eigenvector $= \begin{bmatrix} 11/19.7348 \\ (14-30.3849)/19.7348 \end{bmatrix}$

$e_1 = \begin{bmatrix} 0.5574 \\ -0.8303 \end{bmatrix}$

The e1 unit vector is in the direction which is contained of maximum information of the original data.

Now, let us plot all these data on the graph sheet,



In this graph, we have chosen the mean values of X1 and X2 as our new origin, and taken e_1, as one of the directions.

We are projecting all the data points in the graph on the e_1 axis, this gives us our First Principal Component. And we got all the two-dimension data converted into 1 dimension.

(*For the calculation of the second principal component, we have to project all the data points on the e_2 axis, which is made by choosing $\lambda 2$ as the Eigen Value in the (*) equation, that gives us e_2 eigen vector*).

Therefore, the required First Principal Components are;

Feature	Example - 1	Example - 2	Example - 3	Example - 4
X_1	4	8	13	7
X_2	11	4	5	14
FPC	-4.3052	3.7361	5.6928	-5.1238

Question – 3 - Stochastic vs. Batch Gradient Descent

Mention the difference between Stochastic Gradient Descent (SGD) and Batch Gradient Descent. Which is better if a dataset is much more complex? Provide a mathematical proof as well.

Answer – Gradient Descent (*Optimisation Algorithms*) is one of the most important steps in the Machine Learning, it is used to minimise the cost function by updating the parameters step by step. The two most widely Gradient Descent are Stochastic Gradient Descent and Batch Gradient Descent.

The difference between the two is as mentioned below: -

Batch Gradient Descent: -

Batch Gradient Descent uses the complete training dataset for the computation of the gradient of cost function ($\boldsymbol{J(\theta)}$), this descent ensures high precision but simultaneously is computationally expensive, as it uses the complete dataset for each iteration making the computation slow when dealing with large datasets.

Advantages of using BGD are: -

i. **Accurate Gradient Estimates**: Since it using the entire dataset, the gradient estimate is precise, finding accurate minima for the cost function ($\boldsymbol{J(\theta)}$)
ii. **Good for Smooth Error Surfaces**: It works well for convex or relatively smooth error manifolds.
iii. **Good to use with Small Datasets**: As it takes the entire training dataset for the computation, it ensures very good precision.

Disadvantages of using BGD: -

i. **Slow Convergence**: When dealing with the large datasets, the convergence becomes slow as the gradient is computed over the entire dataset, it can take a long time for the convergence.

ii. **High Memory Usage**: As in this case the datasets are processed in each iteration requiring a high demand of the memory, which makes it computationally intensive.
iii. **Can get Stuck in the Local Minima points**: As the direction of BGD is smooth and exact, it moves through all the local minima points, and if the grad($\boldsymbol{J(\theta)}$) = 0, it can't move ahead as the upgradation of the cost function couldn't be done.

$$\boldsymbol{\theta} := \boldsymbol{\theta} - \boldsymbol{\eta} \boldsymbol{\nabla} \boldsymbol{J}(\boldsymbol{\theta})$$

Stochastic Gradient Descent: -
Stocastic Gradient Descent uses small training sets (Small Subsets of the original Dataset) for computing the gradient of the cost function. It sends only single training example (small subset) through each iteration. This makes the computation very fast and doesn't demand much higher memory allocation.

Advantages of using SGD: -

i. **Faster Convergence**: In this case the gradient is updated after each iteration or data points (*According to the formula below*). The algorithm reaches faster to the convergence point.

$$\boldsymbol{\theta} := \boldsymbol{\theta} - \boldsymbol{\eta} \boldsymbol{\nabla} \boldsymbol{J} \boldsymbol{i}(\boldsymbol{\theta})$$

The upgradation rule.

ii. **Lower Memory Requirements**: As in this the computation is done with only single Training Example through each iteration, it requires less memory allocation as well as less time. This makes SGD suitable for large datasets.

iii. **Escape Local Minima**: SGD can escape local minima and find the global minimum, as it uses any randomly chosen training example and grad($\boldsymbol{J(\theta)}$)≠0 even at the local minimum (*Since one example doesn't contain the entire dataset*), especially for non-convex functions.

Disadvantages of using SGD: -

i. **Noisy Gradient Estimates**: Since the gradient is based on a single and random data point, the estimates can be approximate and noisy, leading to less accurate results.

ii. **Location of Convergence Point Issues**: SGD can coverage to the global minimum point at a faster rate but at the same time it can oscillate around the global minimum, and not getting settled. This can be mitigated by gradually decreasing the learning rate.

iii. **Requires Shuffling**: To ensure randomness, the dataset should be shuffled before each iteration (*Epoch*).

For the case of the more complex Dataset Stocastic Gradient Descent is considered as a better optimiser Algorithm as it uses small training examples for each iteration making the computation of the gradient of the Cost function faster, as well as it requires less memory allocation.

<u>Mathematical Proof: -</u>

Let us consider a Training Dataset as,

$$D = \{(x_i, y_i)\}$$

$$i\ varies\ from\ 1\ to\ N$$

Let us take the Activation Function as $f(x)$,

Hence, the predicted value for the original y is, $\hat{y} = f(x_i; w)$.

Here, w is the respective weights.

Let us take the Lost Function L as,

$$\boxed{L_i(w) = \frac{1}{2} \times (f(x_i; w) - yi)^2}$$

Hence, the average of the loss computed is,

$$\boxed{\nabla J(w) = \frac{1}{N} \sum_{i=1}^{N} (f(x_i; w) - y_i)\, f'(z_i)\, x_i}$$

The gradient of the Mean Square Error (MSE) is,

$$\boxed{\nabla J(w) = \frac{1}{N} \sum_{i=1}^{N} (f(x_i; w) - y_i)\, \nabla_w f(x_i; w)}$$

Batch Gradient Descent – Parameter (Weight) Upgradation Rule: -

$$w_{t+1} = w_t - \eta \nabla J(w_t)$$

Equating the value of the gradient of the Cost Function,

$$\boxed{w_{t+1} = w_t - \eta \frac{1}{N} \sum_{i=1}^{N} (f(x_i; w_t) - y_i)\, \nabla_w f(x_i; w_t)}$$

Here,

- The Computation is very expensive.
- The Gradient is very deterministic.
- The time Complexity for this Computation is $O(n)$, as for each computation the entire data is read, making the computation expensive.

Stocastic Gradient Descent - Parameter (Weight) Upgradation Rule: -

$$\boxed{w_{t+1} = w_t - \eta \left(f(x_{i_t}; w_t) - y_{i_t}\right) \nabla_w f(x_{i_t}; w_t)}$$

The SGD Gradient,

$$\boxed{\begin{array}{c} g_t = \nabla J(w_t) + \xi_t \\ \text{Here, } \xi_t = \nabla L_{i_t}(w_t) - \nabla J(w_t) \\ \nabla L_{i_t}(w_t) \text{ is the cost function} \end{array}}$$

$\nabla J(w_t)$ is the average of the loss, Empirical Mean Square Error Loss

Here, ξ_t is the noise term which is only present in the SGD and not in BGD.

The BGD Gradient,

$$g_t = \nabla J(w_t)$$

At the saddle points: -

For the Batch Gradient Descent,

The value of the $\nabla J(w_t) = 0$, hence the upgradation in the weight doesn't takes place and the Neural Network doesn't move forward making BGD to stall.

For the Stochastic Gradient Descent,

The value of the $\nabla J(w_t) = 0$, still the gradient doesn't become zero because of the presence of the noise in the equation, randomly pushing the parameter away from the saddle point. Hence, the Gradient Descent doesn't stops upgrading.

(***The noise present in the stochastic Gradient Descent is acting as an implicit regularization, which favours fatter minima, which leads to better generalization under Mean Square Error***)

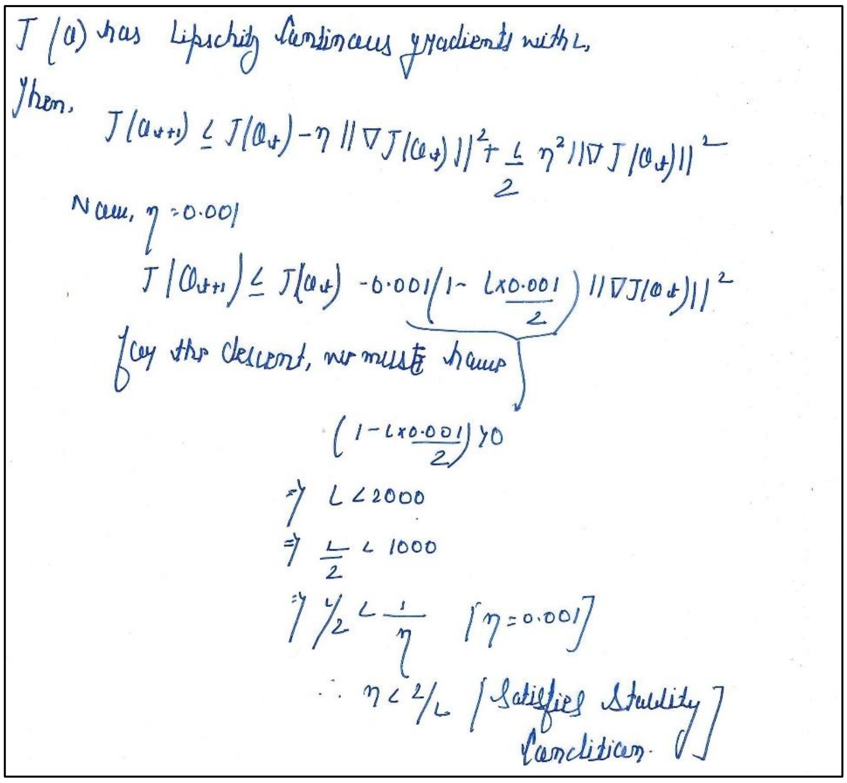
Hence, Stocastic Gradient Descent is better than the Batch Gradient Descent for the Complex Data Patterns as it involves unbiased noisy gradients, which makes the faster Convergence, better Computation than BGD, get rid from the problem of Saddle Point, and leads to better generalization than Batch Gradient Descent.

Question – 4 - Gradient Descent with Fixed Learning Rate

Mathematically write how gradient descent works with a fixed learning rate of 0.001. Create a training loop (pseudo-code is preferred, although a mathematical explanation would also suffice).

Answer – The mathematical model for why 0.001 is used as the learning rate, I can't determine that, but the following reasons might be suitable: -

i. It satisfies the stability condition, (*assuming the function is a smooth function and the gradient can't change randomly*): -



$J(\theta)$ has Lipschitz continuous gradients with L,

Then,

$$J(\theta_{t+1}) \leq J(\theta_t) - \eta \|\nabla J(\theta_t)\|^2 + \frac{L}{2}\eta^2 \|\nabla J(\theta_t)\|^2$$

Now, $\eta = 0.001$

$$J(\theta_{t+1}) \leq J(\theta_t) - 0.001 \underbrace{\left(1 - \frac{L \times 0.001}{2}\right)}_{} \|\nabla J(\theta_t)\|^2$$

for the descent, we must have

$$\left(1 - \frac{L \times 0.001}{2}\right) > 0$$

$$\Rightarrow L < 2000$$

$$\Rightarrow \frac{L}{2} < 1000$$

$$\Rightarrow L/2 < \frac{1}{\eta} \quad [\eta = 0.001]$$

$$\therefore \eta < 2/L$$ [Satisfies Stability Condition.]

ii. Learning rate is large enough to give meaningful descent.

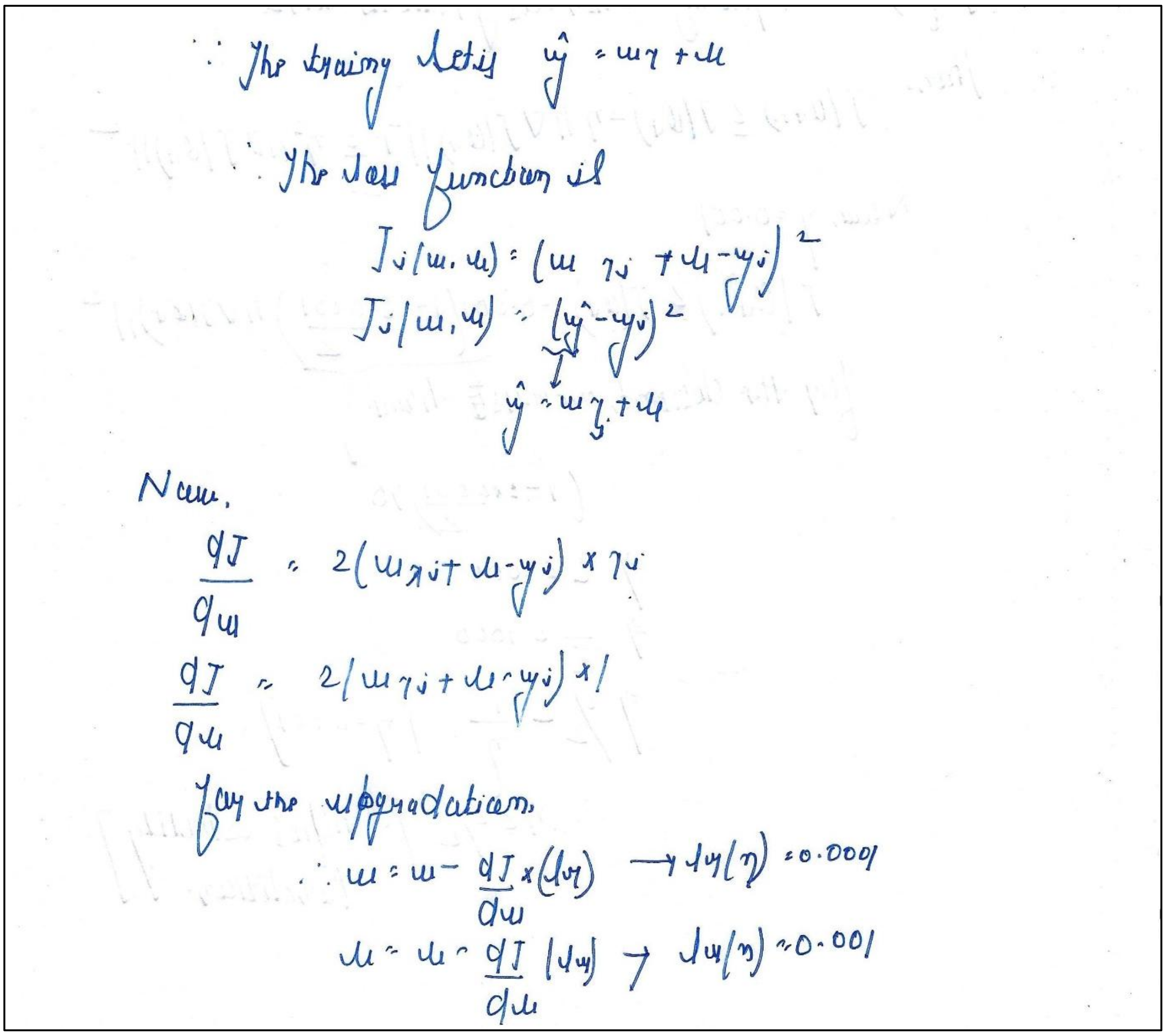
iii. Small enough to avoid Divergence.

iv. It is found that it works well with many of the models.

Mathematically it is used for the upgradation of the parameter for the cost function as

$$\boldsymbol{\theta} := \boldsymbol{\theta} - \boldsymbol{\eta} \boldsymbol{\nabla J i}(\boldsymbol{\theta})$$
$$\boldsymbol{\theta} := \boldsymbol{\theta} - \mathbf{0.001} \times \boldsymbol{\nabla J i}(\boldsymbol{\theta})$$

Here $\boldsymbol{\eta}$ is the learning rate, and $\boldsymbol{Ji}(\boldsymbol{\theta})$ is the Cost Function (*For the Calculation of the loss*).



∵ The training set is $\hat{y} = wx + b$

∴ The loss function is

$$J_i(w, b) = (w x_i + b - y_i)^2$$

$$J_i(w, b) = (\hat{y} - y_i)^2$$

$$\hat{y} = w x_i + b$$

Now,

$$\frac{dJ}{dw} = 2(w x_i + b - y_i) \times x_i$$

$$\frac{dJ}{db} = 2(w x_i + b - y_i) \times 1$$

For the upgradation

$$\therefore w = w - \frac{dJ}{dw} \times (lr) \rightarrow lr(\eta) = 0.0001$$

$$b = b - \frac{dJ}{db} (lr) \rightarrow lr(\eta) = 0.001$$

(** ***I am uploading the link of my notebook for the Code of the above model.***)

(***https://colab.research.google.com/drive/1T51d6a2lJfcrnvHoECCeSx72a_wCRATv?usp=drive_link***)

Question – 5 - Adam Optimizer

Read about the Adam optimizer. What sort of models is it mostly used for? Mathematically analyse the process. Write a few insights about your research.

Answer – Adam – Adaptive Momentum Estimation is one of the Optimisation Algorithms that is a combination of AdaGrad (Adaptive Gradient) and RMSprop (Root Mean Square Propagation). It is the Optimisation Technique that works well every type of datasets. Adam Optimiser works well with the larger Datasets as well as it uses the memory efficiently for the complex Data and adapts the Learning Rate according to each Parameter.

Adam Optisimer works on two Principals: -

i. **Momentum –** This is used to accelerate the Gradient Descent Process by the application of the Exponential weight moving through the average of past gradients.
   Mathematically - $m_t = (1 - \beta_1)(g_t + \beta_1 g_{t-1} + \beta_1^2 g_{t-2} + \cdots)$
   This method helps to smooth the trajectory of the optimisation, leading to the faster conversion, by reducing the oscillation near the global minima.
   The upgradation Rule for the Momentum is: -

$$w_{t+1} = w_t - \alpha m_t$$

   Here, m_t is the average of the Gradients at time t, which is changing.
   α is the learning Rate.
   w_t and w_{t+1} are weights respectively at time t and t + 1.
   The upgradation Rule for the Momentum is: -

$$\boxed{m_t = \beta_1 m_{t-1} + (1-\beta_1)\frac{\delta L}{\delta w_t}}$$

   Here, β_1 is the momentum Parameter.

   $\frac{\delta L}{\delta w_t}$ is the change gradient of the cost function (*Loss Function*).

ii. **Root Mean Square Propagation –** This is an adaptive learning rate method in which RMSprop uses the exponentially weighted average of the squared gradients, which solves the issue of diminishing learning rates.
   It adapts the learning rate for each parameter using the Root Mean Square of the past Gradients.
   The upgradation Rule for the Root Mean Square Propagation is: -

$$w_{t+1} = w_t - \frac{\eta_t}{\sqrt{v_t + \epsilon}} \frac{\delta L}{\delta w_t}$$

Here, v_t is the exponentially weight average of the gradient.
ε is a small constant (e.g., 10^{-8}) added to prevent division by zero.
The upgradation rule of the v_t is: -

$$V_t = \beta_2 v_{t-1} + (1-\beta_2)\left(\frac{\delta L}{\delta w_t}\right)^2$$

Advantages of using Adam Optimiser: -

- Adam optimiser allocates different learning rate to each parameter based on their past gradients. Due to this the optimiser avoids oscillation near the points of Global or Local Minima.
- By adjusting the initial bias, it helps to prevent the early – stage instability.
- It requires minimum hyperparameter tuning when compared to other optimiser algorithms like SGD, making it more convenient for the users.

The Adam Optimiser Algorithm is the combination of the Root Mean Square Propagation and Momentum Optimiser: -

The first Moment (mean): -

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\frac{\delta L}{\delta w_t}$$

The second Moment (variance): -

$$V_t = \beta_2 v_{t-1} + (1-\beta_2)\left(\frac{\delta L}{\delta w_t}\right)^2$$

The Adjustment of the Initial Bias, to remove any early-stage instability, as they might be having the value as zero: -

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

Now, the upgradation rule for the weights is: -

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

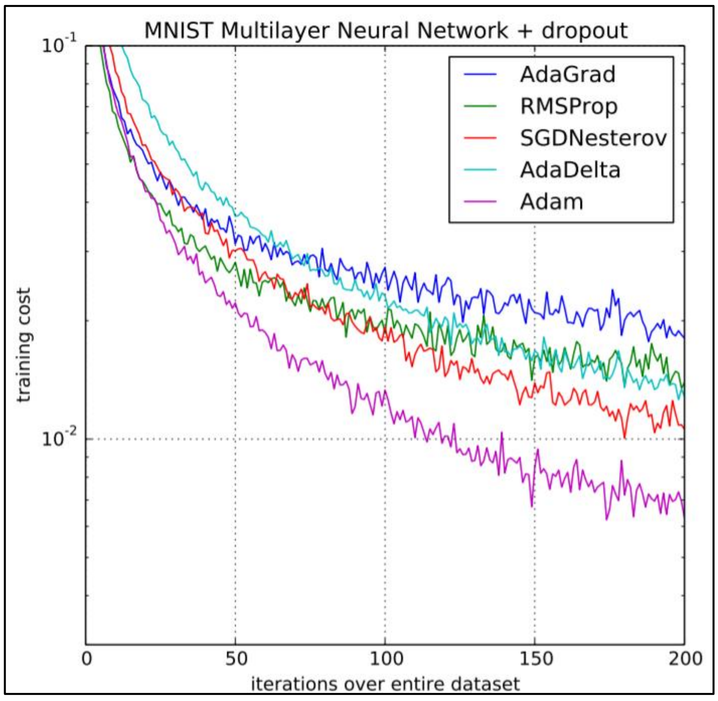
Here, β_1 and β_2 are the decay rates of the moving average of the Gradient, and the squared average of the Gradient.

α is the Learning rate = 0.001 (As it is considered the best for the Gradient Descent with Adam Optimiser).

ε is the constant which removes the chances of division by zero, as when v^t becomes extreme small nearly to zero.

$$v^t = \text{moving average of } g_t^2$$

The Graph presented below is the comparison of different optimiser algorithms, when the training of a multilayer neural network with MNSIT Dataset.



Following are the Models in which Adam Optimiser is Generally used for: -

i. **Deep Neural Networks** - Adam stabilizes and speeds up training.
ii. **Convolutional Neural Networks** - Adam handles different learning speeds across layers well.
iii. **Recurrent Neural Networks** - Adam copes well with vanishing gradients.
iv. **Natural Language Processing Models** - Adam works well with noisy gradients.
v. **Autoencoders & Variational Autoencoders** - Adam converges faster in high-dimensional spaces.
vi. **Generative Models** - Adam stabilizes adversarial training.

Question – 6 - Cross Entropy Loss

Read about Cross Entropy Loss and try to work out why it is better for classification models (refer to class notes and the internet for your research). Mathematical proofs or insights are required.

Answer – Classification problems are the problems in which the input is given and the ML model predicts the probability for each class.

The probability for true detection is 1 and for the wrong ones is 0.

Cross Entropy Loss is one of the ways for the detection of the closeness of the model's prediction with the actual one.

There are two types of Cross Entropy Loss function: -

i. **Binary Cross Entropy Loss Function**
The formula for the BCE Loss function is:

$$BCE = -\frac{1}{N}\Sigma_{i=1}^{N}(y_i.log(p_i) + (1 - y_i)log(1 - p_i))$$

- Here, N is number of samples,
- y_i true label for ith sample either 0 or 1 (Since, it is a binary model),
- p_i model-predicted probability for class 1 for the ith sample.

ii. **Multiclass Cross Entropy Loss Function**
It is also known as SoftMax Loss Function, it is used for the multiclass classification problems. Example – MNIST Dataset
The formula for MCE Loss function is: -

$$CE = -\frac{1}{N}\Sigma_{i=1}^{N}\Sigma_{j=1}^{C}(y_{i,j}.log(p_{i,j}))$$

- Here, N is number of samples,
- C is the number of classes.
- y_{ij} is 1 if class j is correct for ith sample or 0 otherwise.
- p_{ij} is model-predicted probability of ith sample being in class j.

The Reason why Cross Entropy Loss works better with the Classification Models: -

i. In the case of Classification Models, we are giving the probabilities for different classes.
Let,
True Label Distribution be **y**;
And the Model Predicted Distribution be $\boldsymbol{\hat{y}}$**;**
- $\boldsymbol{\hat{y}_i}$ is 1 if the ith sample is correct or shows zero otherwise.
- The predicted probabilities ($\boldsymbol{\hat{y}_i}$) is = $P(class\ i \mid x)$

ii. The **cross-entropy** between true distribution **y** and predicted distribution **ŷ** is:

$$\boxed{\mathcal{L}_{CE} = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)}$$

From this formula it could be inferred that the Cross Entropy directly maximises the probability of the correct class. As,

$$\mathcal{L}_{CE} = -\log(\hat{p}_k) \qquad \log(\hat{p}_k) \uparrow \quad \Rightarrow \quad \hat{p}_k \uparrow$$

iii. **Information theory**: - Information Theory was introduced by Claude Shannon in 1948, is a mathematical framework for quantifying information, data compression, and transmission.
From information theory:

$$H(p, q) = H(p) + D_{KL}(p \parallel q)$$

Here,

- $H(p, q)$ is the cross-entropy
- $H(p)$ is the entropy of true distribution (constant)
- $D_{KL}(p \parallel q)$= **KL divergence**

(***KL Divergence*** *- Kullback Leibler Divergence is a measure from the information theory that quantifies the difference between two probability distributions. It tells us about the how much information is lost when we approximate a True Distribution with P with another Distribution Q.*)

Since in this $H(p)$ is constant, hence, $\min H(p,q) \iff \min D_{KL}(p \| q)$

From this, it could be inferred that minimizing the cross entropy is directly proportional to the minimizing the KL Divergence. Hence, model's predicted distribution becomes as close as possible to the true distribution.

iv. According to SoftMax Function

$$\hat{y}_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

Z_i are the inputs (logits). Using SoftMax function for finding the probability.

According to Cross Entropy Loss Gradient,

$$\mathcal{L}_{CE} = -\sum_i y_i \log(\hat{y}_i)$$

And, $\frac{\partial \mathcal{L}_{CE}}{\partial z_i} = \hat{y}_i - y_i$

From here it could be inferred that,

- The Entropy Loss Gradient is Linear.
- The Gradient is Stable.
- The Gradient is large when the prediction is wrong.
- The Gradient is small (nearly to zero), when the prediction is correct.

According to Mean Squared Error (MSE) gradient,

$$\mathcal{L}_{MSE} = \sum_i (\hat{y}_i - y_i)^2$$

And, $\frac{\partial \mathcal{L}_{MSE}}{\partial z_i} = (\hat{y}_i - y_i)\,\hat{y}_i(1 - \hat{y}_i)$

Hence, the gradient becomes zero when $\hat{y}_i$ is 0 or 1.

From here it could be inferred that,

- The gradients are vanishing.
- Slow and stalled learning Rate.
- Gives wrong prediction values.

v. Maximum Likelihood Estimation

Assuming y ~ $\hat{y}_i$

Likelihood:

$$P(y \mid x) = \prod_i \hat{y}_i^{y_i}$$

→ (**)

Taking log both side of (**)

Therefore, negative log likelihood $-\log P(y \mid x) = \mathcal{L}_{CE}$.

Hence, the Cross – Entropy is inversely proportional to the likelihood estimation.

Minimizing cross-entropy leads to maximum likelihood estimation.

Hence, the Cross Entropy Loss works better for the Classification Models.

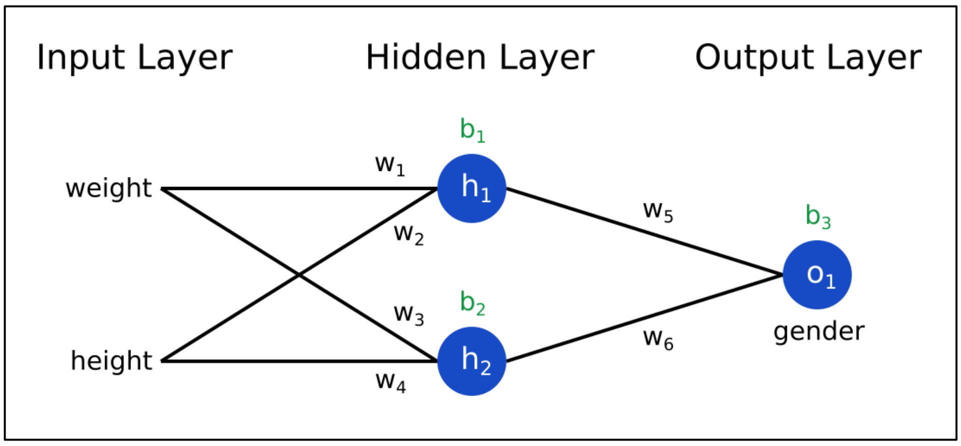
Question – 7 - Activation Layers

What are activation layers? Define a few types and what they are mostly used for. Mention a few mathematical requirements for the same (properties).

Answer – Activation Layers (*Activation Functions which are used as layers*) are components of the Neural Network that adds non – linearity into the Dataset model. This enables the model to learn the complex data patterns (*Example – Images, Audio, etc*). Without this even a neural network would simply behave as a simple linear regression model.

Activation functions decide whether a neuron should be activated based on the weighted sum of inputs and a bias term.

An example of the Neural Network: -



Here, hidden layer is consisting of neurons h1 and h2, but it may consist of many other neurons and many different layers.

Here, b1 and b2 are the biases. w1, w2, w3, w4, w5, w6 are different weights, and the output layer is consisting of a single neuron.

$$h_1 = i_1.w_1 + i_2.w_3 + b_1$$

$$h_2 = i_1.w_2 + i_2.w_4 + b_2$$

Hence, the output is consisting of,

$$output(O1) = h_1.w_5 + h_2.w_6 + b_3$$

Without the use of activation function, these deep neural network works as a simple linear regression.

Hence, let us now introduce an activation function (σ),

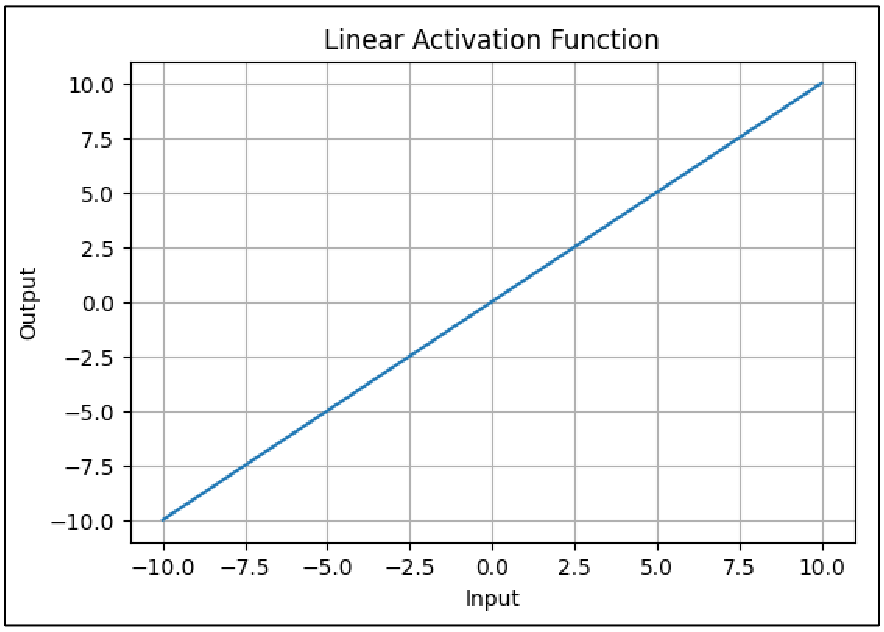
Therefore, the final output becomes

$$output(O1) = \sigma(h_1.w_5 + h_2.w_6 + b_3)$$

Types of Activation Layers (or Activation Functions): -

1. **Linear Activation Function** – In the Linear Activation function the neural network gives just the linear combination of input as their output. The function matches with y = x cases.
   - Linear activation function is used at just one place i.e. output layer.
   - Using linear activation across all layers makes the network's ability to learn complex patterns limited.

Using the Linear Activation function doesn't increase the capacity of the Neural Network to learn complex Data Patterns.

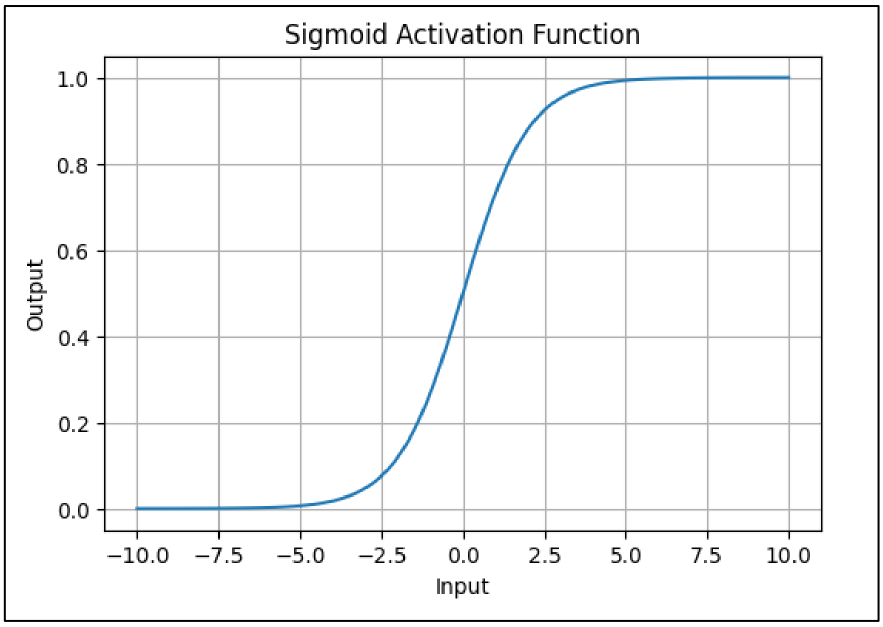


2. **Non – Linear Activation Functions**

   i. **Sigmoid Function** – Mathematically it is defined as ${}^{1}/_{1\,+\,e^{-x}}$
   This formula ensures smooth and continuous output which is important for gradient algorithms.

   - By the use of these activation functions Neural Network can handle complex data patterns (Eg – Images, Audio, etc).
   - The range of the σ (Activation function) is from 0 to 1.

   - The function exhibits a steep gradient when x values are between -2 and 2. This sensitivity means that small changes in input x can cause significant changes in output y. (*The graph below shows all these properties*).
   - The sigmoidal function is strictly increasing $\frac{d\sigma(x)}{dx} > 0$.
   - It is important that the sigmoidal function is Non – Linear as being linear wouldn't help much in the understanding of the Neural Network. $(\sigma(ax + b) \neq a\sigma(x) + b)$

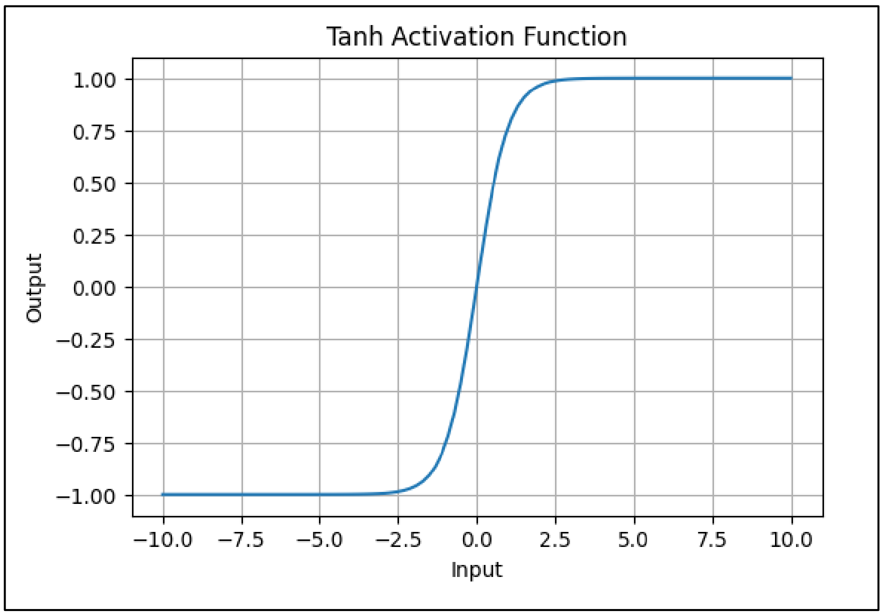


ii. **Tanh Activation Function** – It is a hyperbolic tangent function and is a shifted version of the sigmoid, allowing it to stretch across the y-axis. Mathematically, it is defined as:

$$tanh(x) = {}^{2}/_{1+e^{-2x}} - 1$$

- The range of the function is from -1 to +1.
- The tanh(x) is a non – linear function, important for the neural networks.
- Commonly used in hidden layers due to its zero – centred output, facilitating easier learning for subsequent layers.
- The derivative of the tanh(x) is in a finite range. 0 to 1.
- The saturation at the end points explains the vanishing gradient problem.

$$\lim_{|x|\to\infty} \tanh'(x) = 0$$



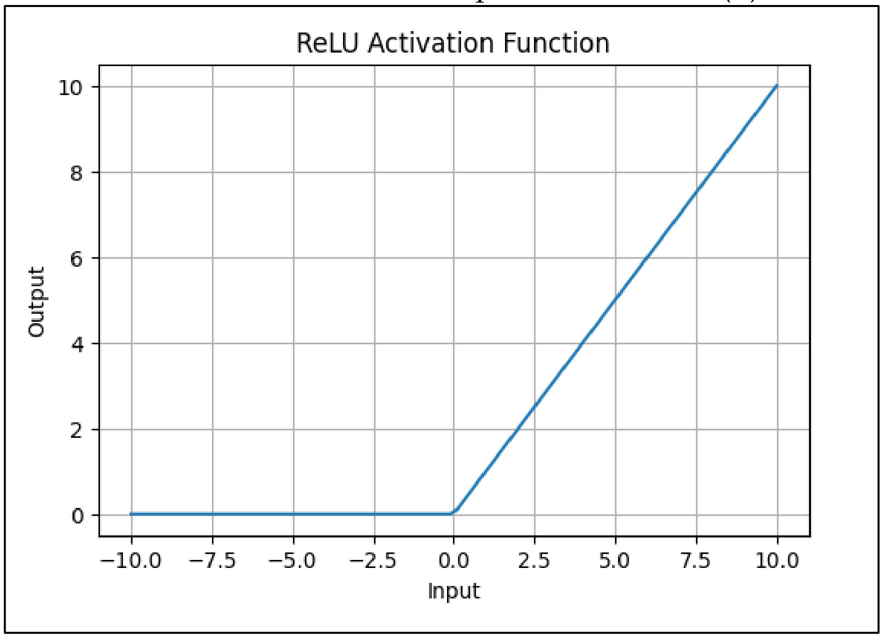
iii. **Rectified Linear Unit Function (ReLU Function)**

Mathematically it is defined as A(x) = max(0,x), it states that if x is negative the value of the function is 0 and when x is positive then the function value is x.

- Range of the function is [0,∞).
- The nature of the function is Non – Linear.
- The function is better than the other two (Sigmoid and Tanh), as it uses very simple mathematical operations. At one time only few neurons are active making the process faster and requires less memory for each computation.
- The constant derivate of the function (1) prevents the problem of vanishing gradients.
- ReLU function also ensures Lipschitz continuity,

$$|\mathrm{ReLU}(x_1) - \mathrm{ReLU}(x_2)| \leq |x_1 - x_2|$$

Here, Lipschitz Constant (L) = 1



iv. **Leaky ReLU** – Mathematically it is defined as,

$$f(x) = \begin{cases} x, & x > 0 \\ \alpha x, & x \leq 0 \end{cases}$$

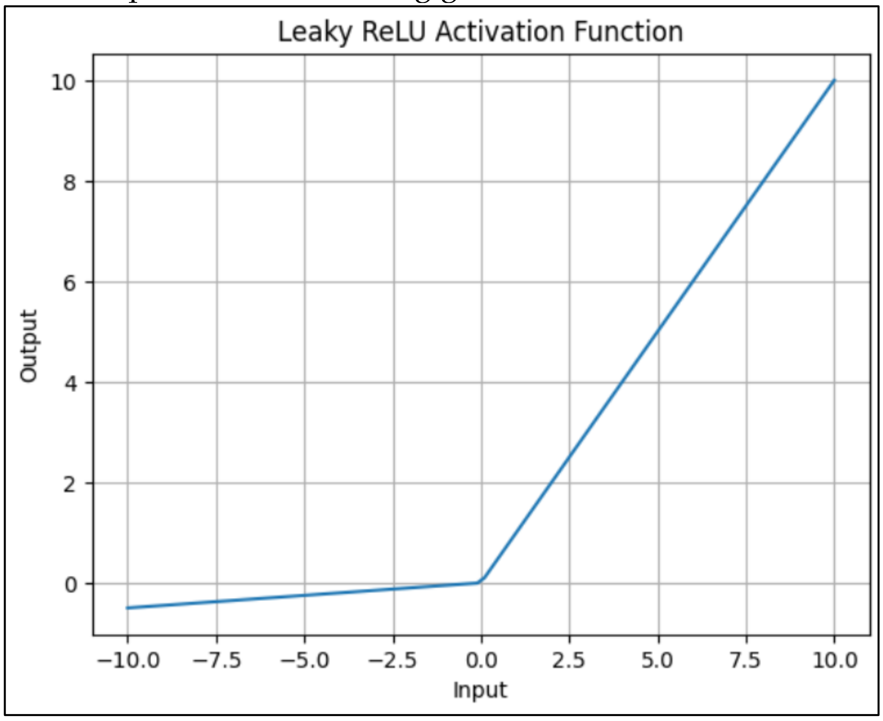
The formula of the function allows a small negative slope α into the function.

- The range of the function is from - ∞ to + ∞.
- Solves the problem of Dying ReLU, in which the neurons get stuck with zero outputs.
- The function is a Non – Linear Function.
- The function ensures the Lipschitz continuity,

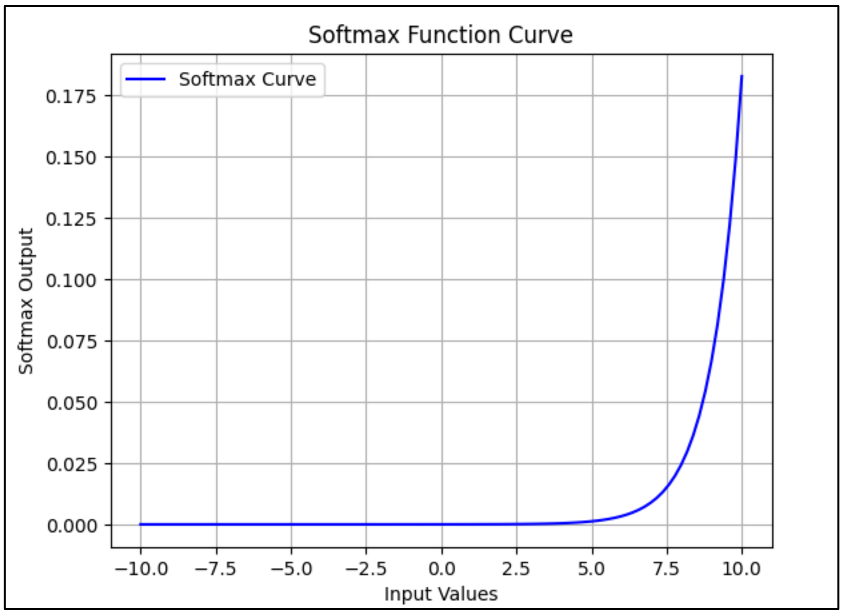
$$|\mathrm{ReLU}(x_1) - \mathrm{ReLU}(x_2)| \leq |x_1 - x_2|$$

Here, Lipschitz Constant (L) = 1

- The derivate of the function is not equal to zero, which solves the problem of vanishing gradients.



3. **Exponential Linear Units** - An Exponential Linear Unit (ELU) is a neural-network activation function designed to improve learning speed and stability by allowing smooth negative outputs instead of hard zeros.
   It combines both ReLU and Leaky ReLU.
   i. **SoftMax Function** – This function is defined for the problem of multiclass classifications. It transforms raw output scores from a neural network into probabilities. It works by squashing the output values of each class into the range of 0 to 1 while ensuring that the sum of all probabilities equals 1.
      - The SoftMax function is a non – linear function.
      - The function assigns probability to each of class, so to identify which class does the input belongs.

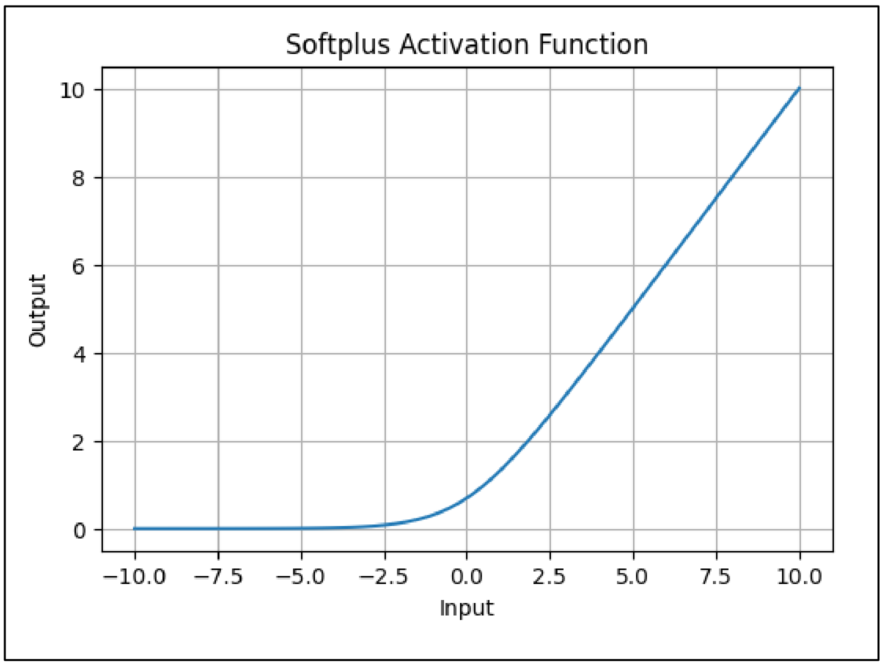


ii. **SoftPlus Function** – Mathematically it is defined as,

$$A(x) = \log(1 + e^x)$$

According to the function, it could be seen that the output is always positive and differentiable, which is a benefit added over ReLU function.

- The range of the function is from 0 to ∞.
- The nature of the function is non – linear.
- SoftPlus is a smooth and continuous function, as it avoids the sharp discontinuities of ReLU (*which lead to problems during optimization*).



Question – 8 - … Explain how the strictly positive nature of the Sigmoid inputs ($x > 0$) mathematically constrains the direction in which all weights feeding into this neuron can be updated for a single data point. Discuss why this constraint makes the optimization inefficient (often resulting in a "zig-zagging" path toward the minimum).

Answer – Let us consider a sigmoid function for a neural network, having a neuron,

$$\hat{y} = \sigma\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

Here, x_i = Inputs send through the neural network,

W_i = Weights

b = biases

The sigmoidal function formula = $1/_{1+e^{-x}}$

Let $L(y, \hat{y})$ be a cost function and z = $\sum_{i=1}^{n} w_i x_i + b$,

The Gradient of the Cost Function,

$$\frac{\partial \mathcal{L}}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_i}$$

$$\frac{\partial \mathcal{L}}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \underbrace{\sigma(z)(1-\sigma(z))} \cdot \underbrace{x_i} \rightarrow (***)$$

This is also positive as the range of the sigmoidal function is between 0 and 1. And x_i is considered as positive.

From the equation (***) it could be inferred that if the input x_i is positive then, the cost function gradient also becomes positive, which means that ***all the weights are being pushed in the same direction for a single data point***.

As the sign of $\left(\frac{\partial \mathcal{L}}{\partial w_1}\right)$ is same as the sign of $\left(\frac{\partial \mathcal{L}}{\partial w_2}\right)$ and the same carries for the rest of the network.

This thing lead us towards a conclusion that: -

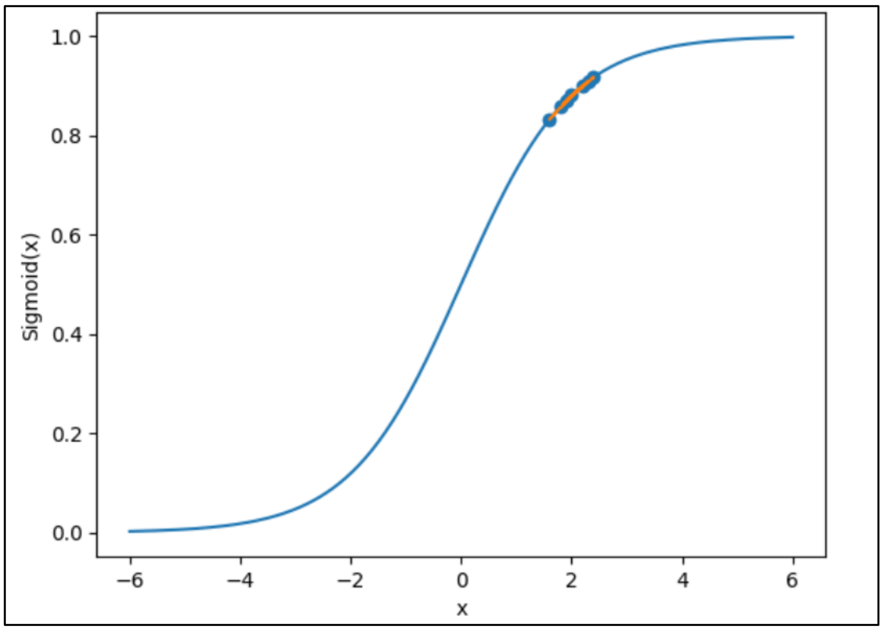
i. We can't move independently in different directions along different weights.
ii. All the upgradation in the parameters are correlated.

The reason behind the Zig – Zag Motion near the points of minimum: -

In the previous case it was assumed all the things for a single data point, but in the case of many data points, some of them have different sign and they may get negative sign.

Then in this scenario all the gradients also change their sign and becomes negative (*since the sign of one gradient is directly proportional to the sign of the other gradient*).

Hence, the optimiser moves in one direction with respect to one weight and then changes sign with respect of the other. This makes the optimisation process insufficient and hence, the path becomes Zig Zag instead of a smooth descent path.



The Zig Zac Motion near the points of minimum in Sigmoid Function

Question – 9 - A proposed "modern" activation function is the Swish function, defined as f(x) = $x \cdot$ σ(x), where σ(x) is the sigmoid function. Derive the analytical expression for the derivative f'(x) exclusively in terms of f(x) and σ(x). Then, argue why computing this gradient during backpropagation is computationally more expensive (in terms of Floating-Point Operations) than computing the gradient of the ReLU function.

Answer – The Swish Activation Function is defined as: -

$$f(x) = x \cdot \sigma(x)$$

Here, the Sigmoidal Function is $\sigma(x) = {}^{1}/_{1\, +\, e^{-x}}$

Finding f'(x),

$$\boxed{\begin{gathered} f'(x) = \frac{d}{dx}[x] \cdot \sigma(x) + x \cdot \frac{d}{dx}[\sigma(x)] \\ f'(x) = \sigma(x) + x \cdot \sigma'(x) \end{gathered}}$$

Since the derivative of the Sigmoidal Function is, $\sigma'(x) = \sigma(x)(1 - \sigma(x))$

Hence, $\boxed{f'(x) = \sigma(x) + x \cdot \sigma(x)(1 - \sigma(x))}$

Now from the definition of the Swish Function,

$$\boxed{\begin{gathered} f(x) = x\sigma(x) \quad \Rightarrow \quad x = \frac{f(x)}{\sigma(x)} \\ f'(x) = \sigma(x) + \frac{f(x)}{\sigma(x)} \cdot \sigma(x)(1 - \sigma(x)) \end{gathered}}$$

Hence, f'(x) = $f(x) + \sigma(x)(1 - f(x))$

In this case, f'(x) = $f(x) + \sigma(x)(1 - f(x))$

As, in this it could be seen that for the computation of f'(x) (*For the Cost Function*) during the back proportion through the neural network, requires a lot of computational requirements,

- Compute the Sigmoidal Function ($\sigma(x)$), in which the exponential form of e^{-x} needs to be computed.
- Simultaneously, we have to also deal with the addition, multiplication and subtraction.

- Multiple floating-point multiplications and additions.

During the Back Proportion the gradient is computed for every neuron, for each layer and for every training sample. The cost function is directly proportional to FLOPs per activation, the neurons and the layers.

This overall makes the computation very expensive and inefficient for the optimisation algorithms.

When Comparing the ReLU Activation Function with Swish Activation Function: -

In the case of ReLU function only one comparison is required to check whether the input value x is positive or negative, and no Multiplication, Addition and Subtraction (Floating Point Operations) is required.

This makes the computation of the gradient during the back proportion very cheap, simultaneously making it efficient as well.