Act fn 1st layer= sigmoid, o/p= vector of val btwn 0-1. O/p of 1st layer is * by weights and biases of 2nd layer. Act fn 2nd layer= sigmoid.

Non-linear layers can combine, transform data in more ways than LL, -> learn more complex relationships and patterns. W/o NL Act fn, DNN can only represent linear mappings, making them equivalent to a single LL.

Initializing all weights and biases to 1.0 is generally not a good idea- Symmetry Problem, Vanishing Gradients, Limited Expressiveness. To fix-use specialized weight init techniques:Xavier or He init, which consider arch, Act Fn, & no. of i/p o/p units to set suitable initial W & B || ***Dropout*** *switched off during test time- Tru (prevent overfitting)*

Local Minima: Points with lower loss in the parameter space, but not necessarily the global minimum. *Sol*- Use momentum or larger learning rate. || **Vanishing Gradients**: Gradients become very small during training, hindering learning. *Sol*- Use ReLU, or skip connections. **Exploding Gradients**: Gradients become very large, causing model instability. *Sol*- Gradient clipping. || **Identification**: Analyze loss curves, gradient magnitudes, and weight viz. **Pooling layers**-> to reduce the spatial Dim. of a feature map,while preserving no.of channels. **Saddle point**: A point where one dimension is a minimum and another is a maximum. Happens due to the nature of high-dimensional spaces. || **Leaky ReLU** prevents **dying ReLU** problem by allowing a small gradient for -ve inputs, ensuring more active neurons during training. **Limitations** of **Leaky**- It can't solve the dying ReLU problem (vanishing gradient) completely. Variants like Parametric ReLU(adapts alpha) can help. **Solution**: Use a dynamic leak parameter, use a different activation function, such as the ELU function or the Swish function.
Mini-Batch: Adv •Faster convergence •Better generalization •Suitable for large datasets – Dis• Adding additional hyperparameters• slower than SGD• May get stuck in bad spots.

f(x){0,10}Not recommended bcz recommended Act Fn is non-differentiable at x = 0 makes it unsuitable for gradient-based optz algo, such as backPP used in NN training
Should use Act Fn that are smooth, continuous, differentiable like ReLU, sigmoid, tanh

ReLU and He-good choice for NN-> helps maintain a balanced initialization, allowing ReLU neurons to activate effectively and promote faster convergence. Efficient, effective. Relatively easy to train NN with this combination, it produces good results.

If learning rate is too small (GDS)- Slow Convergence (long optimization time), Stagnation (may get stuck in local minima), Inefficient Training(impractical no of iterations for good soln), Sensitivity to Weight Initialization(struggle to escape poor local minima), Risk of Underfitting (model cn't capture complexity of data)

x1=x0-a.f'(x)=2.2;x2=1.72 || Overshoot & Diverge curve:*Reduce the learning rate*- cause network take smaller steps in the direction of the gradient, prevent it from overshooting the min of loss Fn during training. *Use a momentum term*-to keep the ntwk moving in the same direction, which can help to prevent it from diverging. *Use a different optimizer*-better suited for non-convex Fn than others.

Full batch GD using the entire training set (not stochastic GD). Can shuffle training data? - *No*, entire training DS will be updated.

KuttaVsBii*: Hyperparameter Transfer Learning:* Learning Rate, Number of Layers to Freeze Layer Architecture, Batch Size, Weight Initialization, Regularization Optimizer, No, of Training Epochs, Evaluation Metrics, Model Evaluation, Validation.
CNN based arch. Mini-batch GD- YES, To prevent the model from overfitting to the ordering of the data, To improve the performance of the model on the training data.
Transfer learning- When: When you have a smaller dataset and pre-trained models are available. How: Freeze layers of a pre-trained model and train only the top layers. Hyperparameters: Learning rate, unfreezing strategy, no. of trainable top layers, loss fn etc ||| **Improve your classifier**→ Use bigger NN, increase param. ||| **True abt Batch Norm**→speedup learning, noise to hidden layer. **64*64*16 (1*1)= 4097 ||| Correct->** reduce nc by using 1x1, std maxpooling to reduce nh,nw ||| **Regularization:** implicit tregularizing effect, your choise of regularization ||| **Helps get** weights out of local minima ||| Impose gradient clipping |||

GhodaVsZ: **Geometric transformations** *(flipping, rotating, cropping, translating)*. **Color space transformations** *(changing brightness, contrast, saturation)*. **Noise injection**, Partial Occlusion, Erasing, Image Mixing | ***Data Augment.***- NO. bcoz the test set is used to evaluate the final perf. of model on unseen data. If you apply data augmentation to the test set, it will not be a fair evaluation of the model's perf. D'Aug used to artificially increase the size of the training set. || **Learnable parameters in Batch norm -** γ (scale) and β (shift) for each feature

Model achieves (high)98% training accuracy and 54%(low) test accuracy -> Data augmentation, Regularization, Dropout, Early stopping, Model architecture.

Error fn measures difference between predicted and actual. **Loss function** quantifies how well the prediction did vs actual. The loss fn is a specific form of the error fn used in training, typically computed for a single data point or batch. Both are minimized during training.
Chain Rule: if If we have fn composed within another fn, such as y=g(f(x)), then the chain rule.

$$\frac{dy}{dx} = \frac{dy}{df} \cdot \frac{df}{dx}$$

CNN for Img- **Local Connectivity**- CNNs are effective for image tasks due to their ability to capture local features. **Translation Invariance**- They provide translation invariance, recognizing patterns regardless of location. **Hierarchical Feature Learning**- enables them to learn complex image representations. **Parameter Sharing**- reduces the number of model parameters and aids generalization. CNNs leverage **Spatial Hierarchy**, detecting local patterns and entire objects. **Convolutional Filters**- capture features at different scales. **Pooling**- layers reduce dimensions for computational efficiency. CNNs have achieved **State-Of-The-Art Performance** in image-related tasks.

Sigmoid – Binary Classification || ***ReLU***- Linear Regg || ***Softmax***- Multiclass classification
Batch **GD**: Uses entire dataset; stable but slow. || ***Stochastic*** **GD**: One sample at a time; faster but noisier. || ***Mini-Batch GD***: Middle-ground; benefits of both. **Derivatives**: Sigmoid: f'(x) = sigmoid(x)(1 - sigmoid(x)) || ReLU: f'(x) = 1 if x > 0, else 0 || Tanh: f'(x) = 1 - tanh^2(x)
Activation Fn- Sigmoid, ReLU, Tanh || **Pros**- Smooth output, - Fast to compute, - Smooth output || **Cons** - Outputs are saturated, meaning that they are close to 0 or 1, - Dead neurons if the input is negative, - Outputs are centered around 0

CNN Reason for loss→ It is possible that the weights are incorrectly initialized. • It is also possible that the learning rate is too low. • Some other answers may also be possible (for eg X not correlated with Y at all is a possibility) ||| **Criminal**- F1 Score > Precision & Recall || Pooling layers cause spatial dimensions to shrink and allow us to use fewer parameters to obtain smaller and smaller hidden representations of the input.

Firing rule determines whether a neuron should fire for any input pattern. **ANN Loss Fn-** Popular Loss Functions *–>MSE or L2 Loss* • Real no prediction • Regression • Sensitive to outliers→*–>MAE or L1 Loss* • Less sensitive to outliers *–>Huber Loss* (Smooth L1 Loss) • Combination of L1 and L2 loss *–>Binary Cross Entropy* (Logistic) • Output is probability • Mainly used for Binary classification (output 0 or 1) • How the BCE works for Multilabel – classification? *–>Cross-entropy* • Output is probability • Mainly used for multi-class Classification • Ground truth should be in one hot vector format.

$$\mathrm{MSE} = \frac{1}{2}\sum_{i=1}^{n}(\hat{y_i} - y_i)^2 \quad \mathrm{MAE} = \frac{1}{2}\sum_{i=1}^{n}|\hat{y_i} - y_i| \quad L_\delta(y,\hat{y_i}) = \begin{cases} \frac{1}{2}(\hat{y_i} - y_i)^2, & \text{if } |\hat{y_i} - y_i| < 0 \\ \delta|\hat{y_i} - y_i| - \frac{1}{2}\delta^2, & \text{otherwise} \end{cases}$$

$$L = \frac{1}{n}\sum_{j=1}^{n}\sum_{i=1}^{c} -(y_i \log(\hat{y_i}) + (1 - y_i)\log(1 - \hat{y_i})) \quad L = \frac{1}{n}\sum_{j=1}^{n}\sum_{i=1}^{c} -(y_i \log(\hat{y_i}))$$

CNN- Img shape: [28 × 28 × 1]→ i)*CL 32 filters with kernel (3, 3), stride = (1, 1), padding = same*->O/p shape: (Input Shape-Kernel Size+2*Padding)/Stride+1= (28-3+2*1)/1+1=28x28x32 *Param*=32*[(3 x 3 x 1) + 1]= 320 || **ii)***Max Pooling with pool size(2,2), stride= (2,2)-> Output Shape: (Input Shape - Pool Size) / Stride + 1* →O/p: (28-2)/2+1=14x14x32 *Param=0* || **iii**) *CL with 64 filters with kernel (3, 3), stride = (1, 1), padding = same*→ o/p: (14 - 3 + 2 * 1) / 1 + 1 = 14 x 14 x 64 || Param = 64[(3 x 3 x 32) + 1]= 18,496 || (**iv**) *Max Pooling Layer with pool size (2, 2), and stride = (2, 2)*→ *O/p= (14 - 2) / 2 + 1 = 7 x 7 x 64* || (**v**) Flatten Layer *=7x7x64 (vi) Dense* →o/p= 10, param= 10* [(7 x 7 x 64) + 1] = 31370

The value of "c" must be equal to the no. of channels in the feature map. To perform a **(1 x 1 x c)** convolution on a **(7 x 7 x 64)** feature map without errors, c should be 64. The output shape will remain (7 x 7 x 64). || YOLO - 7* (7 x 7 grid 5 anchor)

No, the output vector is not correct. The **softmax** Fn is a non-linear Act Fn that converts a vector of real values into a prob distribution. The o/p values of the softmax function sum to 1.

Regularization method leads to weight sparsity-> "(L1) Lasso regularization" bcz L1 adds a penalty term to the loss fn that is proportional to the absolute values of the weights (L1 norm of the weights). The objective of L1 regularization is to encourage the model to learn a sparse set of weights, effectively driving some of the weights to become exactly 0 during training.
Regularization Term, Effect on Gradient Descent, Sparse Weight Selection.

Cancer- ReLU vs Sigmoid→ "Sigmoid" is the standard for binary classification. It maps the ntwk o/p between 0 and 1, providing a clear probability interpretation for the positive class(cance), making decisions at a 0.5 threshold.

1. Interface 3 x 3 x 3, pass 20 filters, no. of learnable parameters? → 20 * (3 * 3 * 3 + 1) = 560.
2. **Step size impact training**- Too large can overshoot minima. Too small can slow down convergence. **3. Image input shape** is 16 x 16, can you apply max pooling, 5 consecutive times of the size 2. No, 16 -> 8 -> 4 -> 2 -> 1 -> 0.5. After 4 times, size is less than pooling window.**4.** ***Dropout vs Drop Connect***: Both are used for generalization || Dropout: Regularization technique, Deactivates neurons during training. Use it to prevent overfitting. Drop Connect: Drops out weights. It can be more powerful but is used less commonly. Again, use for regularization. **5.** Apply 1 x 1 convolution layer, how you can control the no of channels? A 1x1 convolution layer; pointwise convolution, controls the number of output channels without changing spatial dimensions. It applies a linear transformation to every individual pixel in the input. The two main purposes are: ***Dimensionality Reduction***: Decreases the number of channels, reducing computational needs while retaining key features. Transform a 4x4x64 input feature map to 4x4x32. ***Dimensionality Expansion***: Increases the channels, helping to identify more intricate patterns. Can change a 4x4x64 input to 4x4x128. **6.** You can reshape **feature maps for concatenation** by ensuring that the dimensions align in the axis along which you want to concatenate. Downsampling, one by one convolution layer. **7. Apply Back propogation** for 1 weight 1 iteration. → **Forward - z** = wixi + b, a = sigmoid(z)=1/(1+e^-z), || **Loss(MSE)** = (1/n)(y-y`)^2 -> y` = a(last) || **Back - dE/dw** = dE/da *dz/dw, || **dE/da** = -(y-y`), **da/dz** = sigmoid(1-sigmoid) -> **sigmoid** = y`, **dz/da** = d/da(wa + b) || **Wnew** = wold - (n*dE/dw)
8. Use **gradient descent** to update the weight for 1 iteration. For y=x^2, with x=2, gradient is dy/dx=2x=4. If learning rate is 0.01, x_new = 2 - 0.01 * 4 = 1.96.
9. Good combination of weight matrix and optimizer (Activation function)- Sigmoid or Hyperbolic Tangent (tanh) Activations with Xavier (Glorot) Initialization - Both sigmoid and tanh activations have a similar S-shaped curve, though tanh is zero-centered. The Xavier initialization takes into account the size of the previous layer, ensuring the variance remains the same across layers. ||| **10. Symmetry breaking problem:** When neurons in the same layer learn the same features due to identical initialization. **Solution**: Random weight initialization, Use an asymmetric activation Fn, (ReLU or ELU Fn), Use a regularization technique, such as dropout or L1 regularization, Use data augmentation to increase the diversity of the training data. ||| **11. Momentum:** Momentum in advanced optimization helps accelerate convergence by reducing oscillations. Using Adam with momentum can be a good suggestion to speed up training.
Benefits of momentum: -Faster convergence -Reduced noise sensitivity -Improved stability
Yes, **Adam** is a very effective optimizer that uses **momentum** and adaptive learning rates. It is often used to train large datasets efficiently. |||| **12.** NN architecture with batch size of 528, & batch size of 16 or 32 is better. **A smaller batch size is often better**, as it can lead to better generalization and convergence. A larger batch size may converge faster but could risk overfitting. **advantage and disadvantage of smaller batch size-** *Disadvantage*: Slower training speed; Worse hardware utilization; Problem with batch normalization *Advantage*: Less memory consumption; Small batches can offer a regularizing effect that provides better generalization |||| ***For a 10-class multi-class classification****, you should have 10 output nodes in the final layer.* |||| **Ground truth** (Levels) - S*S (B*5+#number of classes) **S x S**: This represents the grid size into which the image is divided. **YOLO** divides an image into an S x S grid, and each cell in this grid is responsible for detecting objects whose center falls within it. **B:** Represents the number of bounding boxes each grid cell predicts. Each bounding box contains 5 values.

Why is scaling (γ) and shifting (β)→ Simply normalizing the inputs of a layer may change what the layer can represent. For instance, normalizing the inputs of a sigmoid function would constrain them to the linear regime of the nonlinearity. Inserting learnable trans- formation in the batch norm layer restores the representation power of the network. ||| Model does really well on the dev set, but fails upon deployment (can be thought of as test set). Class imbalance in **dev** set, Distribution shift between dev and **deployment** ||| **3x3 conv (stride 2) - 2x2 Pool - 3x3 conv (stride 2) - 2x2 Pool** → 1x1 ← 2x2 ← 5x5 ← 10x10 ← 21x21. The support is 21x21=441 pixels. |||| **False**- **Discarded variables** might have strong correlation with a non discarded variable and only one of them might have a non zero coefficient with l1 regularization at random

Sigmoid:
σ(x)' = d/dx ((1/1 + e-x)-1)
σ(x)' = – 1/ (1 + e-x)2 d/dx (1+e-x)
σ(x)' = – 1/ (1 + e-x)2 d/dx (e-x)
σ(x)' = – e-x / (1+ e-x)2 d/dx (-x)
σ(x)' = e-x / (1 + e-x)

tanh(x)=ex−e−xex+e−x
It is now possible to derive using the rule of the quotient and the fact that:
derivative of ex is ex and
derivative of e−x is −e−x
So you have:
ddxtanh(x)=(ex+e−x)(ex+e−x)−(ex−e−x)(ex−e−x)(ex+e−x)2
=1−(ex−e−x)2(ex+e−x)2=1−tanh2(x)

ANN Loss Function- What is ERROR- Error is a general term that represents the difference between the predicted output of a model and the actual target output. It quantifies how far off the model's predictions are from the true values. Error is a more general term and can be used interchangeably with terms like "residuals" or "discrepancy." It's often used in a broader sense to describe the overall performance of a model. **What is Loss Function-** A loss function (or cost function) is a specific mathematical function that quantifies the difference between the predicted output and the true target values. It is used during the training of a machine learning model to guide the optimization process. The goal during training is to minimize the value of the loss function. The choice of the loss function depends on the type of task the model is designed for. For example, mean squared error (MSE) is commonly used for regression tasks, while categorical cross entropy is used for classification tasks. **What is Cost Function -** The terms "cost function" and "loss function" are often used interchangeably. Both refer to the same concept—a mathematical function that measures the difference between the model's predictions and the actual target values. In the context of neural networks, you'll often hear the term "loss function" more frequently. During the training process, the overall cost or loss is the aggregated value of the loss function across all training samples. The optimization algorithm adjusts the model's parameters to minimize this aggregated cost, thus improving the model's performance. **Mean Square Error(MSE)-** Real number prediction, Regression (e.g., house price prediction), MSE is sensitive to outliers. Mean Squared Error (MSE), also known as L2 Loss, is a common loss function used in regression problems. It measures the average squared difference between the predicted values and the actual target values. The formula for MSE is as follows: 1/n ©(Ÿi-Y1)^2. The Mean Squared Error (MSE) is sensitive to outliers due to the squaring operation in its calculation. The squaring of errors amplifies the impact of larger errors, leading to a disproportionate influence of outliers on the overall loss. **Mean Absolute Error (MAE) or L1 Loss-** Less sensitive to outliers**.** Mean Absolute Error (MAE), also known as L1 Loss, is another commonly used loss function in regression problems. Unlike Mean Squared Error (MSE), MAE measures the average absolute difference between the predicted values and the actual target values. The formula for MAE is as follows: 1/n ©|Ÿi-Y1|. **Huber Loss (Smooth L1 Loss)-** Combination of L1 and L2 loss. Huber Loss, also known as Smooth L1 Loss, is a robust loss function commonly used in neural networks, especially in regression tasks where the goal is to predict continuous values. It is designed to combine the best properties of Mean Absolute Error (MAE) and Mean Squared Error (MSE), providing both the robustness of MAE in the presence of outliers and the differentiability of MSE.

$$L_\delta(y,\hat{y_i}) = \begin{cases} \frac{1}{2}(\hat{y_i} - y_i)^2, & if\ |\hat{y_i} - y_i| < 0 \\ \delta|\hat{y_i} - y_i| - \frac{1}{2}\delta^2, & otherwise \end{cases}$$

Binary Cross Entropy (Logistic) - Output is probability, Mainly used for Binary classification (output 0 or 1), How the BCE works for Multilabel – classification? Binary Cross Entropy (Logistic) Loss is a commonly used loss function in binary classification problems within neural networks. It is particularly well-suited for scenarios where each example belongs to one of two classes (positive or negative), and the network outputs a probability distribution over these two classes.

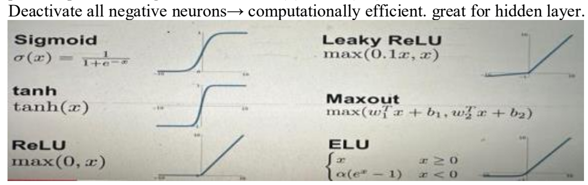
$$L = \frac{1}{n}\sum_{j=1}^{n}\sum_{i=1}^{c} -(y_i \log(\hat{y_i}) + (1-y_i)\log(1-\hat{y_i}))$$

. Cross-Entropy- Output is probability, Mainly used for multi-class classification, Ground truth should be in one hot vector format. Cross-Entropy Loss, also known as categorical cross-entropy, is a commonly used loss function in neural networks, particularly in multi-class classification tasks. It measures the dissimilarity between the predicted probability distribution and the true probability distribution (one-hot encoded labels).

$$L = \frac{1}{n}\sum_{j=1}^{n}\sum_{i=1}^{c} -(y_i \log(\hat{y_i}))$$

. Gradient Descent- Learning rater scheduling - Change the values of the LR during training Learning rate decay: reducing learning rate over time- Approach 1 - reduce the LR by some constant systematically eg - reduce 50% every 10 epochs. Approach 2 - gradually reduce the learning rate over time,eg - Exponential decay, cosine decay. Approach 3 - reduce learning rate by a constant whenever validation loss stops improving. Warm up: gradually increase the learning rate initially and then cool down until the training. **Gradient-based dynamic learning rate-** a dynamic learning rate strategy that adjusts the learning rate during training based on the magnitude of the gradient. The idea is to use a learning rate that is proportional to the absolute value of the gradient. This approach is designed to address the challenges associated with training when the initial parameter values are far from the minima of the optimization problem. 1. **Starting Value is Far from Minima**: When the optimization process starts, the model's parameters are initialized with some values. If these initial values are far from the optimal values (minima), the gradient of the objective function is often large. **Gradient Approaches 0 as Model Approaches Minima:** As the optimization algorithm progresses, the model's parameters are adjusted to minimize the objective function. During this process, the gradient approaches zero as the model gets closer to the optimal values. **Learning Rate = Learning Rate * abs(Gradient):** The new learning rate at each step is calculated as the product of the original learning rate and the absolute value of the gradient. New Learning Rate=Original Learning Rate×abs(Gradient). This means that when the gradient is large (indicating that the optimization is far from convergence or the parameters are far from the optimal values), the learning rate is increased. This allows for larger steps in the parameter space to expedite convergence. **Benefits**- 1- Faster convergence during the initial stages of training when the parameters are far from the optimal values.2- Improved stability during later stages of training when the model is fine-tuning its parameters. **Time-based dynamic learning rate-** The time-based dynamic learning rate is a strategy that involves adjusting the learning rate during the training process based on the number of epochs completed. The idea is to systematically decrease the learning rate as training progresses, allowing for larger steps in the parameter space initially and finer adjustments as the optimization process approaches convergence. The formula for the time-based learning rate adjustment is often expressed as: Learning rate = learning rate 1 − ($epoch\ number / total\ epochs$). **Benefits**- 1- Stability: Gradual reduction in the learning rate enhances the stability of the optimization process, especially during the later stages of training. 2- Fine-Tuning: Smaller learning rates in the later epochs allow the optimization algorithm to make finer adjustments to the model parameters. **Challenges with gradient descent- Local Minima and Saddle Points:** Gradient descent can get stuck in local minima or saddle points, where the gradient is zero but the point is not an optimal solution. **Solution**: Techniques like momentum and random initialization can help escape saddle points.

Convergence to Local Minima: Gradient descent is sensitive to the initial conditions, and it may converge to local minima instead of the global minimum. **Solution:** Techniques like random initialization, learning rate tuning, and using different optimization algorithms can help escape local minima. **Vanishing and Exploding Gradients**: In deep neural networks, the gradients can become very small (vanishing gradients) or very large (exploding gradients) during backpropagation, leading to slow convergence or instability. **Solution**- Weight initialization techniques, gradient clipping, and the use of activation functions that mitigate vanishing/exploding gradients (e.g., ReLU) can help address this challenge. **Gradient Descent with Momentum**: Gradient Descent with Momentum is an optimization algorithm that improves upon basic gradient descent by introducing a momentum term. The momentum helps accelerate the optimization process, especially when the gradient changes direction frequently or oscillates. It Decrease the number of iterations to reach the optima➔converges faster than gradient descent. Involves adding additional hyperparameter that controls the amount of history (momentum) to include the update equation. Momentum hyperparameter ranges from 0 to 1. Momentum 0 is the same as the gradient descent.**Advantages**- 1- The momentum term helps the optimization process to maintain and gain speed, especially in the presence of oscillations or irregularities in the cost landscape.2- Momentum helps dampen oscillations and navigate through regions with small gradients more efficiently. This can lead to a smoother and more direct path to the optimum.**Disadvantages**- 1- The momentum parameter (®)needs to be tuned, and choosing an inappropriate value may lead to suboptimal performance.2- In some cases, the momentum term may cause overshooting, where the optimizer gains too much speed and skips over the optimal solution. This can be mitigated by careful tuning of the momentum parameter. **Adaptive Gradient Descent (AdaGrad)** is an optimization algorithm that adapts the learning rates for each parameter based on their historical gradients. It aims to give smaller learning rates to frequently occurring parameters and larger learning rates to infrequently occurring ones. It modifies the general learning rate at each time step for every parameter based on the past gradients. It eliminates the need to manually tune the learning rate. **Advantages**- 1- AdaGrad adapts the learning rates based on the historical information of gradients, allowing it to automatically adjust to the characteristics of the optimization landscape.2- AdaGrad performs well when dealing with features that have varying importance or when working with sparse data. It assigns smaller learning rates to frequently occurring features. **Disadvantages-** 1- Over time, the accumulated squared gradients in the denominator can become very large, leading to very small effective learning rates. This can result in slow convergence or premature convergence.2- AdaGrad requires additional memory to store and update the accumulated squared gradients for each parameter. This may become an issue for very large models**. AdaDelta** is an extension of AdaGrad that seeks to address some of its limitations, particularly the issue of the monotonically decreasing learning rates. In AdaDelta, the learning rates are adaptively adjusted based on a running average of the second moments of the gradients. **Advantages-**1 AdaDelta adapts learning rates on the fly based on the historical information of gradients, making it less sensitive to the choice of hyperparameters.2- Similar to AdaGrad, AdaDelta eliminates the need for manual tuning of the learning rate. **Disadvantages**-1- AdaDelta requires additional memory to maintain running averages of second moments for each parameter. This can be a limitation for large models.2- While AdaDelta is effective for many problems, it may not always outperform other optimization algorithms, and its performance can depend on the characteristics of the optimization landscape. **Activation Function-** Activation functions are mathematical operations applied to the output of a neuron in a neural network. They introduce non-linearities to the network, allowing it to learn complex patterns and relationships in data. Activation functions play a crucial role in determining the output of a neural network and enabling it to learn and make predictions. The linear activation function (known as "no activation") makes no changes to the weighted sum of the input; it just returns the value it was given. All deep learning models with linear activation functions, regardless of depth or width, are just single layer perceptron. **What makes an activation function good?-**Hidden layers: Nonlinearity, Computationally simple and efficient, Avoid limited range, Avoid vanishing/exploding gradient. Output layer: Nonlinear (classification problem), Linear (regression problem). **Linear Function**: the output of the function is the same as the input. Mathematically, it performs a linear transformation on the input data, and there is no non-linearity introduced. The linear activation function is often used in the output layer of regression models. The function doesn't do anything to the weighted sum of the input, it simply spits out the value it was given. If value of c = 1➔f(x) = x➔identity activation function. One of the significant drawbacks is its lack of non-linearity. Neural networks with only linear activation functions in hidden layers cannot capture complex, non-linear relationships in the data. **Sigmoid:** The sigmoid activation function, also known as the logistic function, is a widely used non-linear activation function in neural networks. It squashes input values to the range between 0 and 1, making it suitable for binary classification problems. ʃ(x)= 1/1+e^-x.**Advaantages**- 1-the sigmoid function provides a smooth and continuous transition between 0 and 1. This smoothness can be advantageous during the optimization process.2- Well-suited for binary classification problems where the goal is to predict probabilities of belonging to a particular class. **Disadvantages- 1-** The sigmoid function suffers from the vanishing gradient problem, especially in deep neural networks. Gradients become very small for extreme input values. (close to 0 or 1), leading to slow convergence and difficulties in learning.2- The output of the sigmoid function is not zero-centered, which can lead to issues during optimization, especially when used in hidden layers. **Hyperbolic Tangent (tanh)-** The hyperbolic tangent (tanh) activation function is a non-linear activation function that squashes input values to the range between -1 and 1. It is an improvement over the sigmoid activation function, as it is zero-centered, making it more suitable for certain scenarios.tanh(x)= (e^x – e^-x) / (e^x + e^-x). **RELU:** The Rectified Linear Unit (ReLU) is a popular non-linear activation function used in neural networks. It is defined as the positive part of its argument, i.e., ReLu(x) = max(0,x). Takes a real valued number and threshold at zero. Deactivate all negative neurons→ computationally efficient. great for hidden layer.



Leaky ReLU: The Leaky Rectified Linear Unit (Leaky ReLU) is a variant of the Rectified Linear Unit (ReLU) activation function. It allows a small, non-zero gradient when the input is negative. The purpose of introducing this non-zero slope is to address the "dying ReLU" problem where neurons can become inactive for negative inputs during training. F(x)=max(ax,x). **Regularization**- Regularization refers to techniques that are used to calibrate machine learning models in order to minimize the adjusted loss function and prevent overfitting or underfitting. Using Regularization, we can fit our machine learning model appropriately on a given test set and hence reduce the errors in it. Helps the model to generalize to unseen (test) data. Can increase or decrease training time based on the regularizer. Generally, works better for larger network and with sufficient data.