

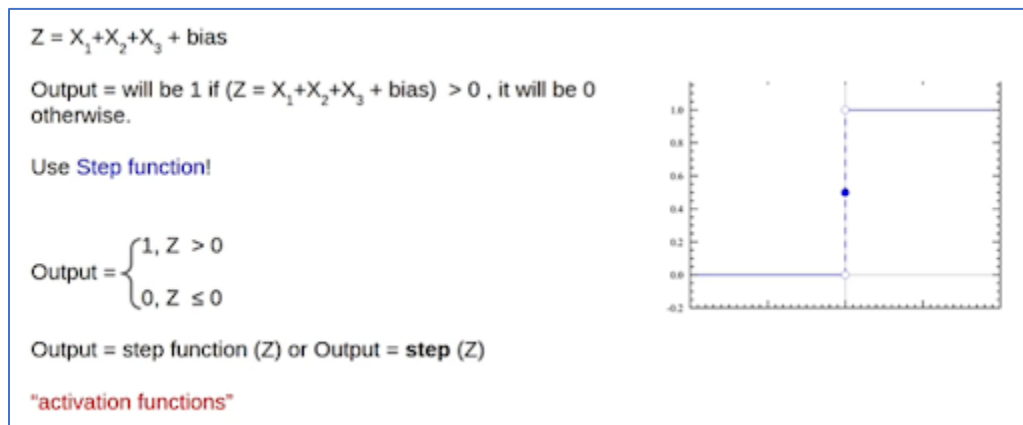
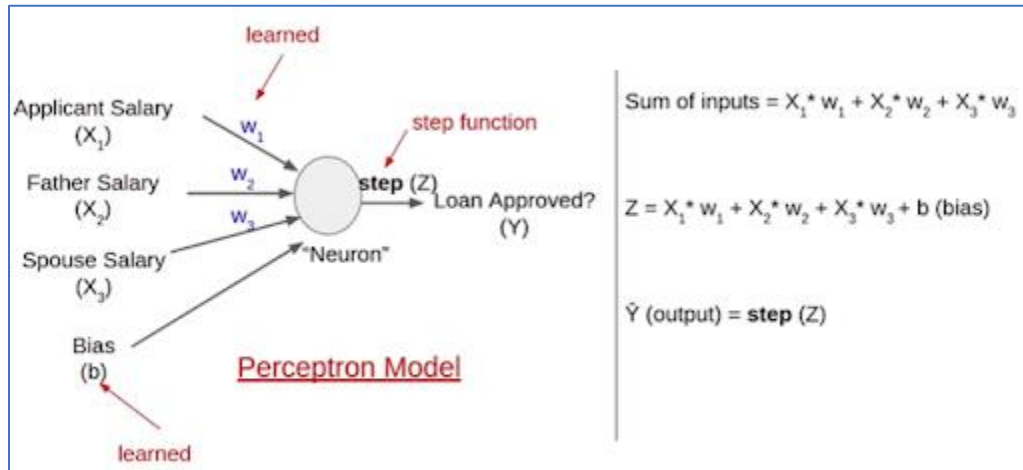
Table of Contents

1.	Basics of Neural Networks	2
1.1.	What is a Perceptron?	2
1.2.	What is a Sigmoid Neuron?	2
1.3.	What are the weights and biases?	3
1.4.	What is a Decision Boundary?	3
1.5.	What is the Gradient Descent Algorithm?.....	3
1.6.	What is Forward Propagation and How it will work?	4
1.7.	Backpropagation.....	4
1.8.	What is an activation function?	5
1.9.	Variants for Gradient Descent	10
1.10.	Optimizers	11
1.11.	Loss Functions.....	13
	What is batch normalization?	14
	Regularization Techniques used in Deep Neural Networks.....	15

1. Basics of Neural Networks

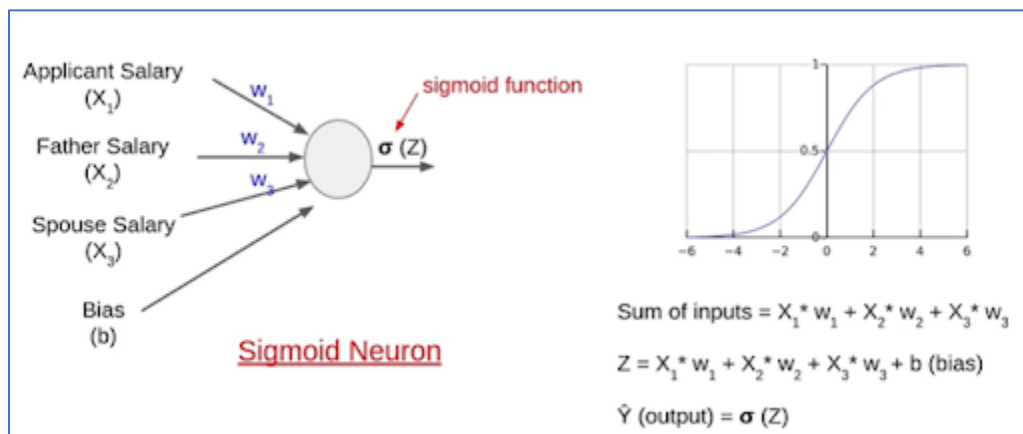
1.1. What is a Perceptron?

A Perceptron network is a simple type of artificial neural network used in machine learning for binary classification tasks. It is analogous to a human brain, although in a much simpler form. It takes in several inputs, multiplies them by respective weights (can be learned and updated from data during the learning process), and sums them up. This sum passes through an activation function, typically a step function, which converts the output into either 0 or 1, representing a binary classification.



1.2. What is a Sigmoid Neuron?

A sigmoid neuron is another type of neuron that has the sigmoid function as an activation function and returns the output from 0 to 1 instead of 0 or 1.



1.3. What are the weights and biases?

Weights and biases are the parameters in a neural network that are used to learn the relation between input features and output predictions. In a multi-layer neural network, each neuron output is connected to a neuron in the next layers through weights. Weights control the influence of each input on the output of a neuron. During the training process, the model adjusts the weights to minimize the difference between predicted and actual outputs. Bias is an additional parameter added to the neuron in a network. It acts as an offset or threshold for the weighted sum of the inputs. Adding the bias helps shift the activation function's output, allowing the neuron to better fit the data. Biases like weights, are adjusted during the training to improve the model's performance.

1.4. What is a Decision Boundary?

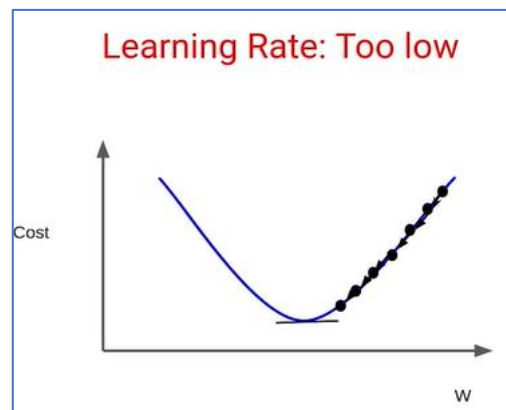
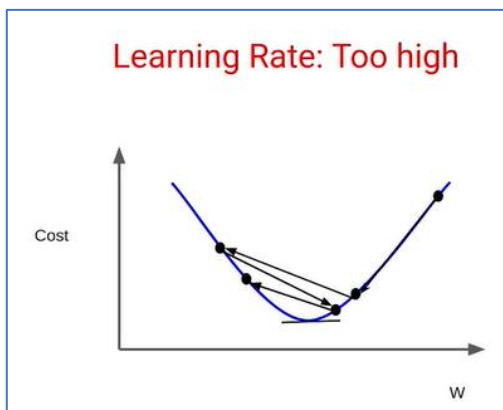
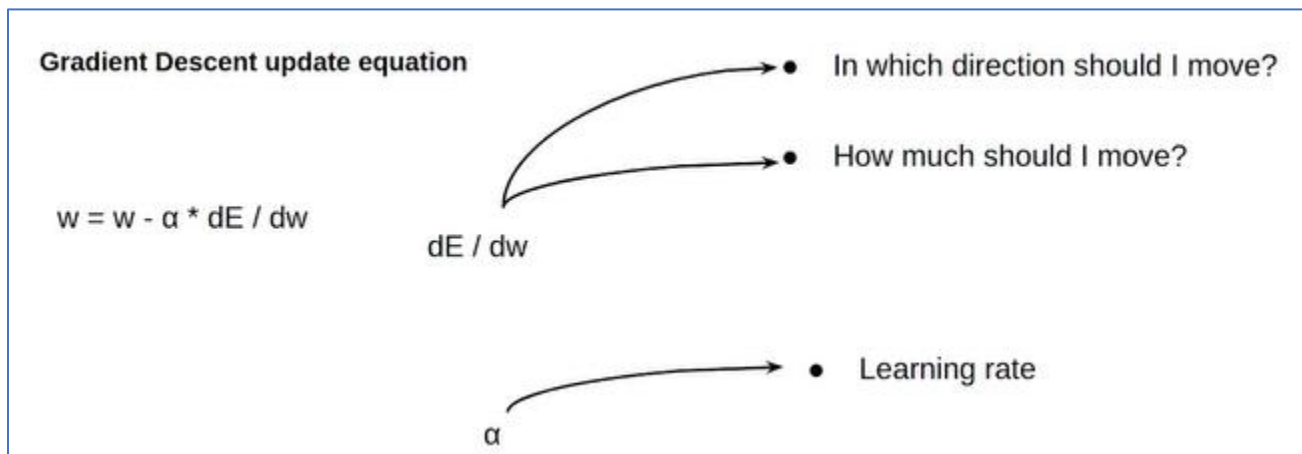
A Decision boundary is a line (in the case of 2D features) or a plane (in the case of n-D features) that separates two classes. The model adjusts its decision boundary during the training process based on the data and labels provided, so that it can better separate different classes and make accurate predictions.

1.5. What is the Gradient Descent Algorithm?

Gradient Descent Algorithm is an optimization algorithm used in machine learning and other mathematical optimization problems to find the minimum of a function. According to the gradient descent algorithm weights will be updated in the direction opposite to the direction of the tangent drawn at W_0 .

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

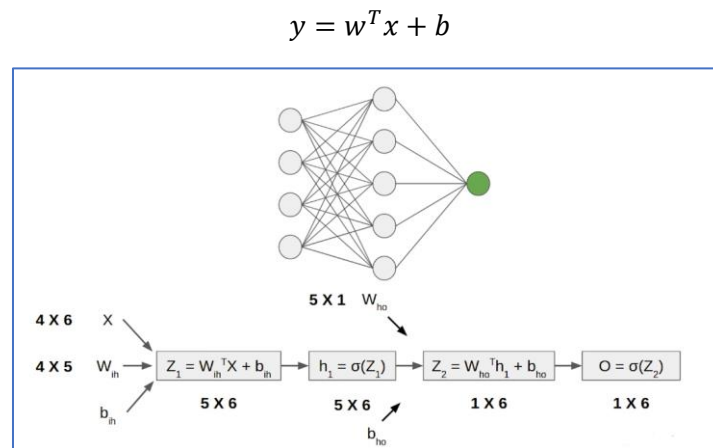
Learning rate η is a parameter, that controls how much the parameters are adjusted at each step. A high learning rate causes the algorithm to overshoot the minimum, while a low learning rate may lead to slow convergence.



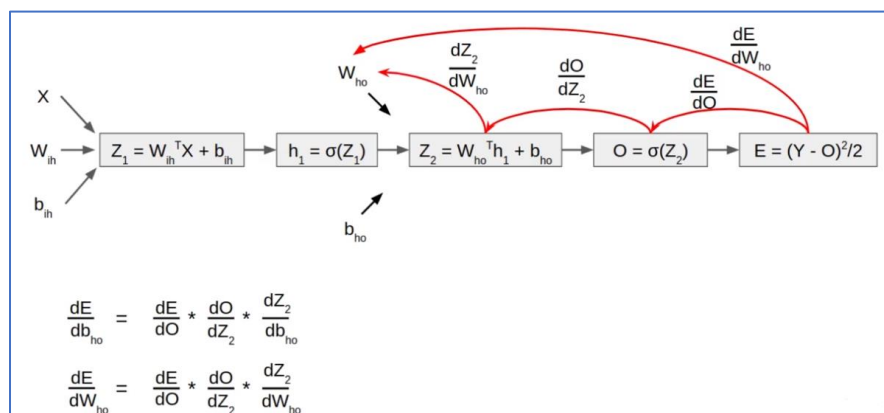
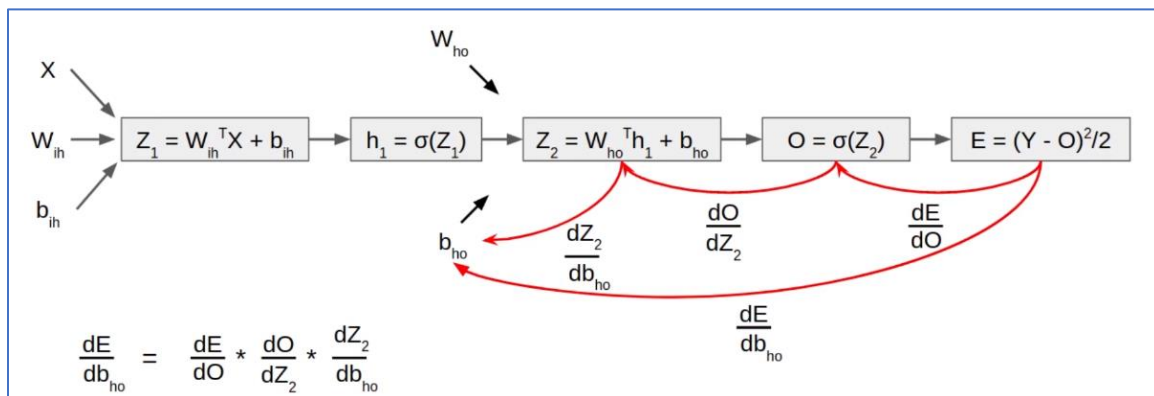
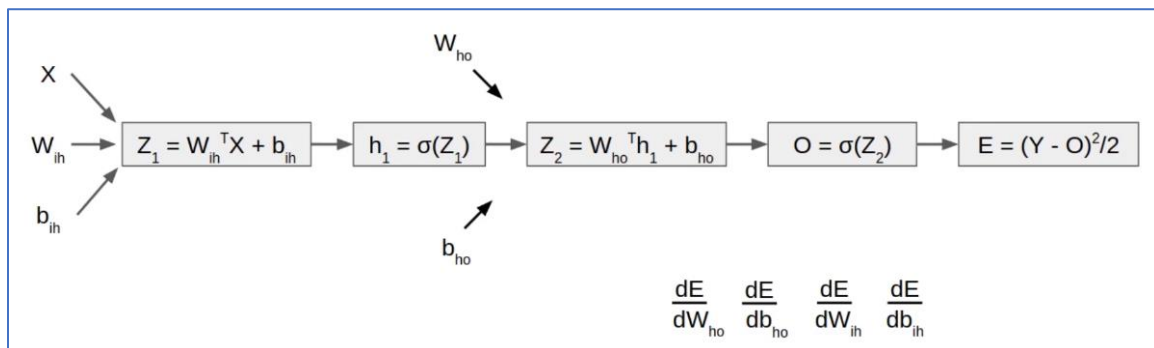
We stop updating the weights if the error does not change after a few iterations or when the number of iterations defined is completed.

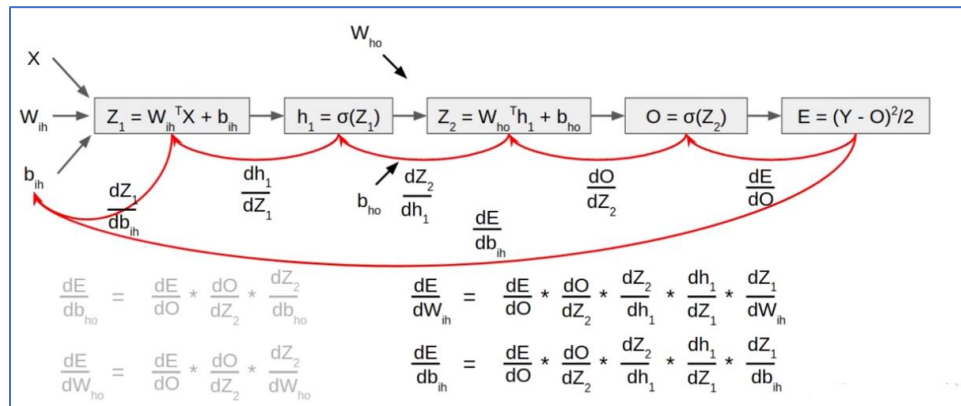
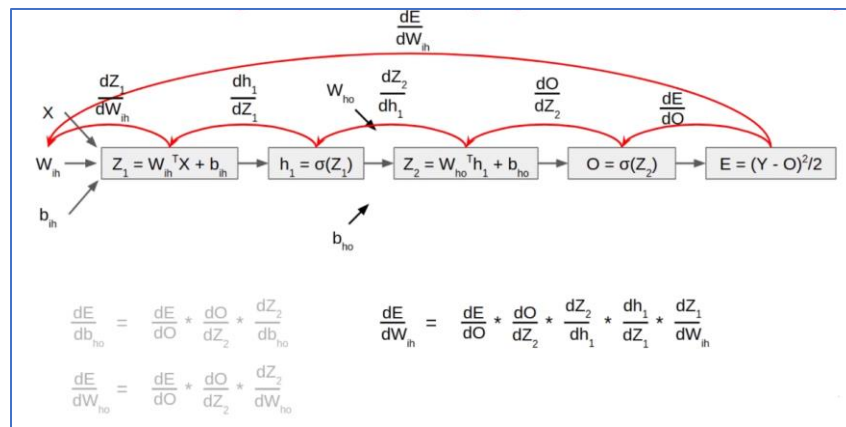
1.6. What is Forward Propagation and How it will work?

Forward propagation in a neural network is a mathematical operation of input features and weights/biases in the network to get the output.



1.7. Backpropagation



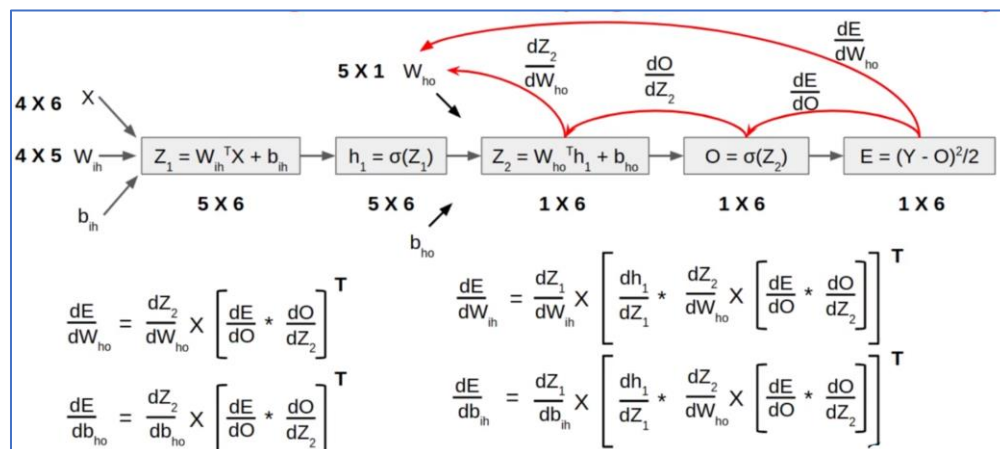


$$W_{ho} = W_{ho} - \alpha * \frac{dE}{dW_{ho}}$$

$$W_{ih} = W_{ih} - \alpha * \frac{dE}{dW_{ih}}$$

$$b_{ho} = b_{ho} - \alpha * \frac{dE}{db_{ho}}$$

$$b_{ih} = b_{ih} - \alpha * \frac{dE}{db_{ih}}$$



1.8. What is an activation function?

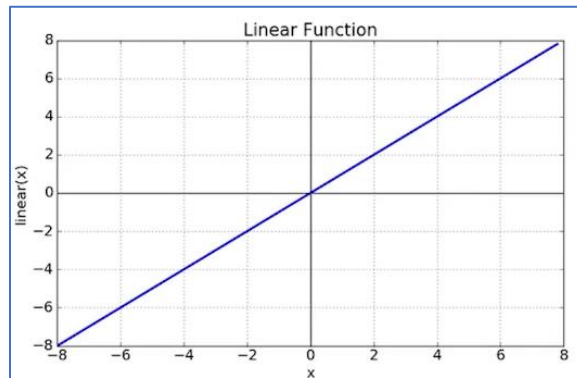
Let's say if we don't use the activation function, the output will become a linear relation of weights and inputs. Thus, it can solve linear problems and might not be able to solve complex relations between input and output.

Considerations for an activation function:

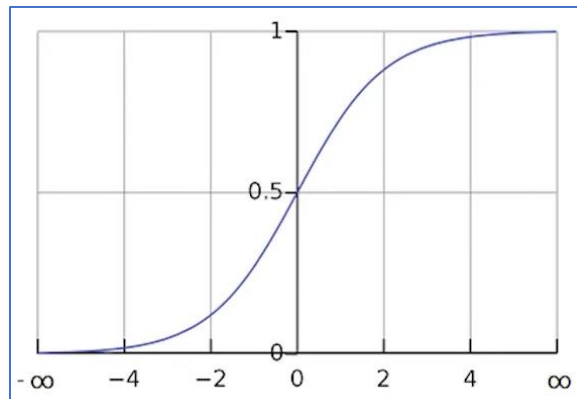
- Should be continuous at all points.
- Should be differentiable at all points. (Needed in backpropagation gradient calculations)

Different Activation functions:

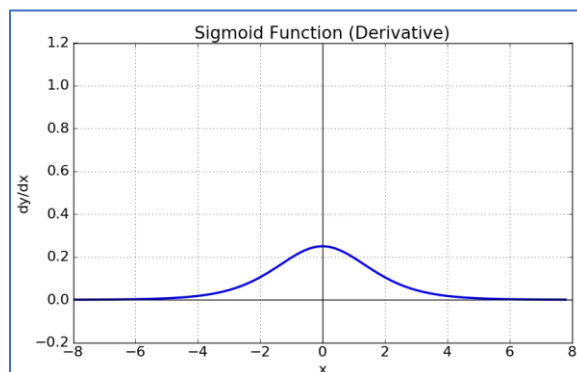
- Linear Activation Function
 - $(y = w^T x + b)$



- Ranges from -infinity to infinity $(-\infty \text{ to } +\infty)$
 - Generally used as an activation function of output neuron for a regression problem.
 - As it cannot capture non-linear relationships, not preferred in input and hidden layers.
- Sigmoid Activation Function
 - $\sigma(x) = \frac{1}{1+e^{-x}}$



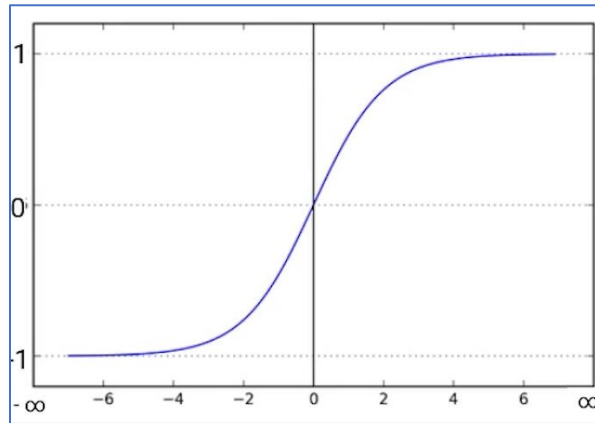
- Ranges from 0 to 1
 - These values can be considered as probabilities, and thus used as activation functions when we want to calculate the neighborhood of the input.
 - Sigmoid always returns the probability of a data point belonging to class 1.
 - Derivative of a sigmoid function $\frac{d\sigma}{dx} = \sigma(x)(1 - \sigma(x))$



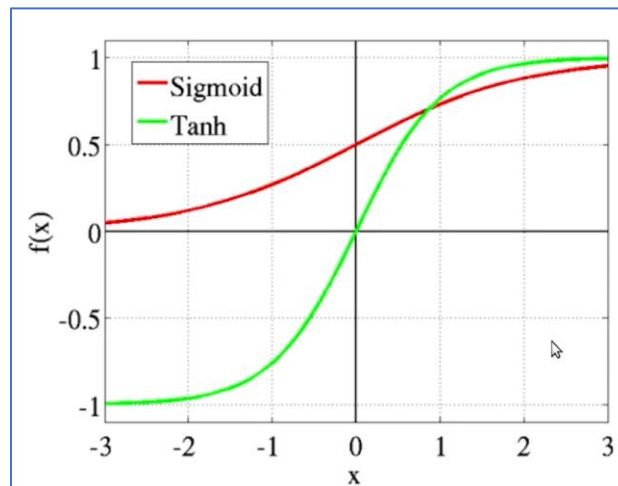
- As shown in the above figure derivative of sigmoid results in very small values.

- Tanh Activation Function

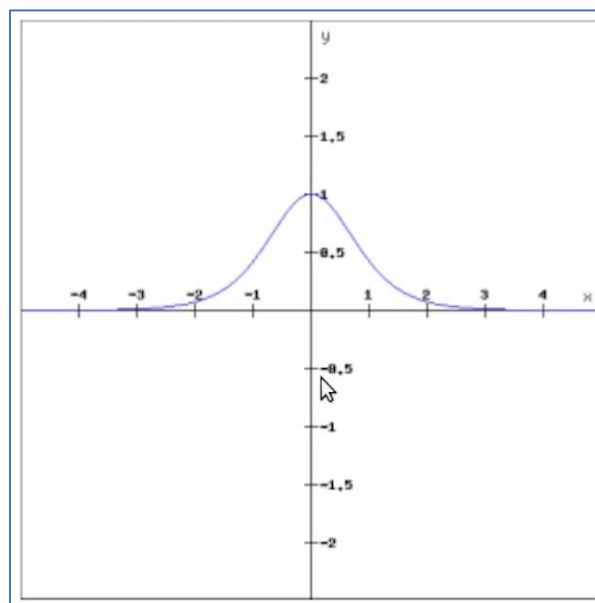
- $\tanh(x) = \frac{2}{1+e^{-2x}} - 1$ can also be written as $\tanh(x) = 2\sigma(2x) - 1$



- Ranges from -1 to 1



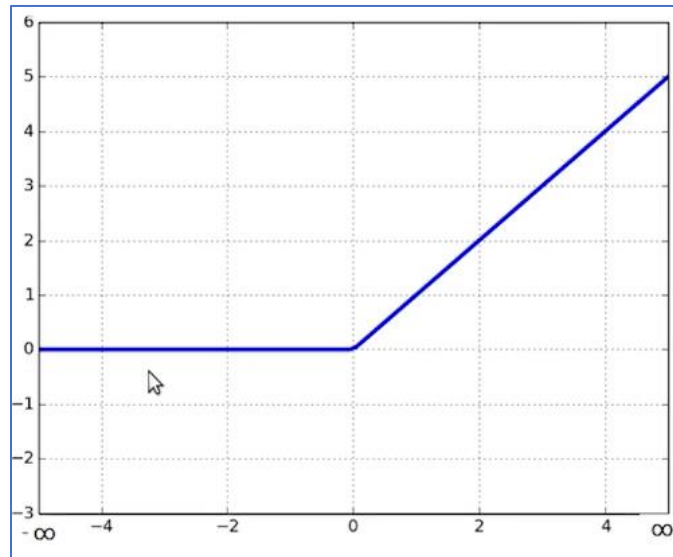
- Derivative of the tanh is $\frac{d\tanh}{dx} = 1 - \tanh^2(x)$



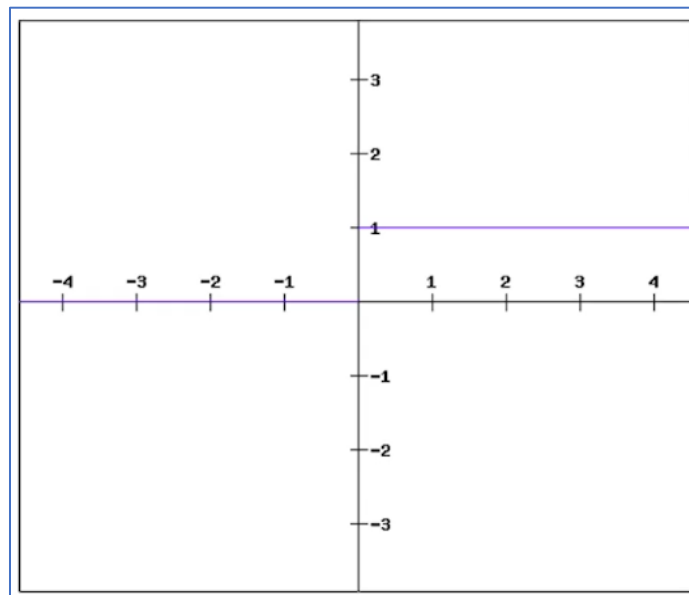
- Derivatives of tanh are larger than sigmoid and hence gradients will be larger (update process will be faster)

- ReLU: Rectified Linear Unit

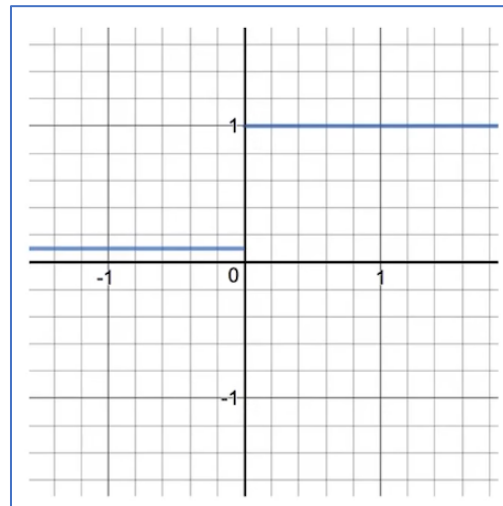
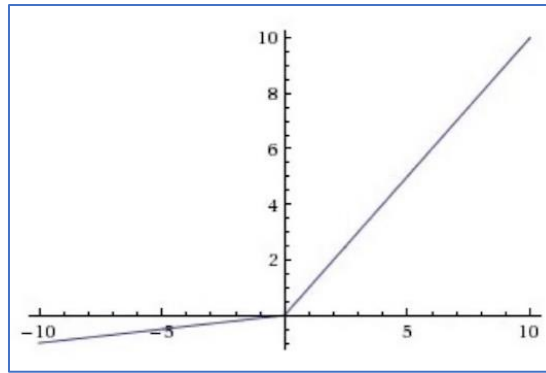
- $ReLU(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$ or also can be written as $ReLU(x) = \max(0, x)$



- Ranges from 0 to ∞
 - Computationally very efficient than sigmoid and tanh activation functions
 - Not differentiable at point 0 and hence we modify the derivative for simplification.
 - $\frac{d ReLU}{dx} = \begin{cases} 0 & \text{for } x \leq 0 \\ 1 & \text{for } x > 0 \end{cases}$



- For negative inputs derivative of ReLU becomes 0 which makes no updates in the weights.
 - This can be solved using Leaky ReLU
- Leaky Rectified Linear Unit (Leaky ReLU)
 - $Leaky ReLU(x) = \begin{cases} 0.01x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$ this can be written as $Leaky ReLU(x) = \max(0.01x, x)$
 - Ranges from $-\infty$ to ∞
 - Now derivatives of Leaky ReLU can be $\frac{d Leaky ReLU}{dx} = \begin{cases} 0.01 & \text{for } x \leq 0 \\ 1 & \text{for } x > 0 \end{cases}$



- Softmax

- Popularly used for multi-class classification problem
- As we know the sigmoid returns the probabilities, and the output probabilities are independent of each other. That means the probability of data point 1 belonging to class 1 is independent of the probability of data point 1 belonging to class 2. This is why the sigmoid activation function is not preferred for multi-class classification problems.

In the two-class logistic regression, the predicted probabilities are as follows, using the sigmoid function:

$$\Pr(Y_i = 0) = \frac{e^{-\beta \cdot \mathbf{X}_i}}{1 + e^{-\beta \cdot \mathbf{X}_i}}$$

$$\Pr(Y_i = 1) = 1 - \Pr(Y_i = 0) = \frac{1}{1 + e^{-\beta \cdot \mathbf{X}_i}}$$

In the multiclass logistic regression, with K classes, the predicted probabilities are as follows, using the softmax function:

$$\Pr(Y_i = k) = \frac{e^{\beta_k \cdot \mathbf{X}_i}}{\sum_{0 \leq c \leq K} e^{\beta_c \cdot \mathbf{X}_i}}$$

- SoftMax also returns the probability for each class.
- $\text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$
- If the number of classes is 2 SoftMax acts as a sigmoid function.

One can observe that the softmax function is an extension of the sigmoid function to the multiclass case, as explained below. Let's look at the multiclass logistic regression, with $K = 2$ classes:

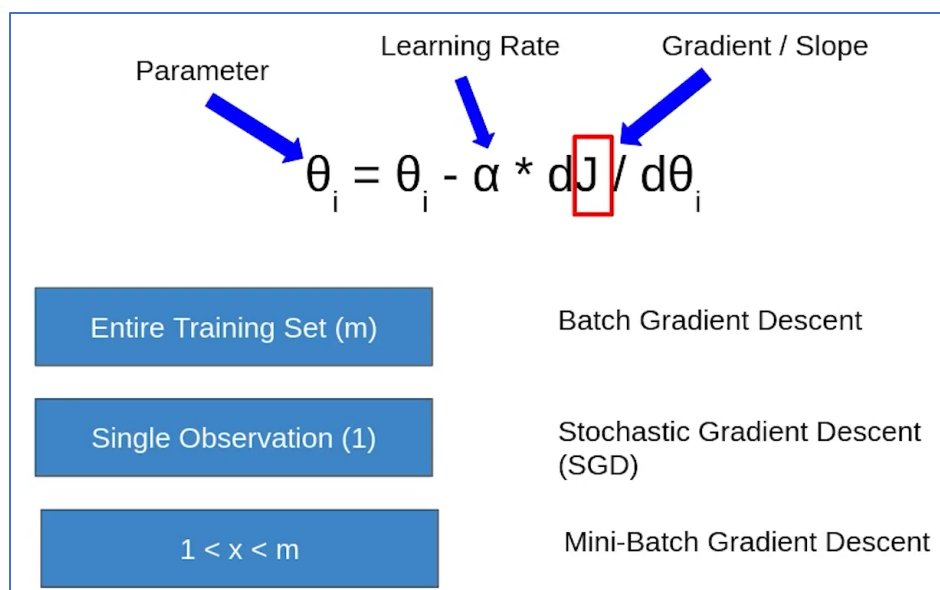
$$\Pr(Y_i = 0) = \frac{e^{\beta_0 \cdot \mathbf{X}_i}}{\sum_{0 \leq c \leq K} e^{\beta_c \cdot \mathbf{X}_i}} = \frac{e^{\beta_0 \cdot \mathbf{X}_i}}{e^{\beta_0 \cdot \mathbf{X}_i} + e^{\beta_1 \cdot \mathbf{X}_i}} = \frac{e^{(\beta_0 - \beta_1) \cdot \mathbf{X}_i}}{e^{(\beta_0 - \beta_1) \cdot \mathbf{X}_i} + 1} = \frac{e^{-\beta \cdot \mathbf{X}_i}}{1 + e^{-\beta \cdot \mathbf{X}_i}}$$

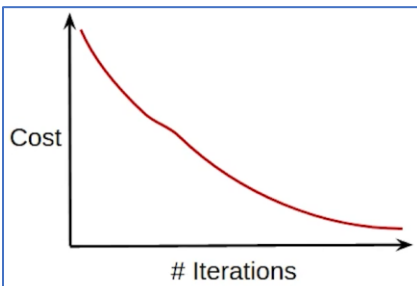
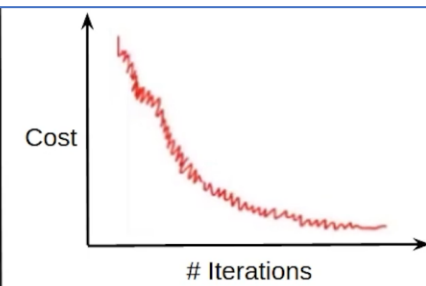
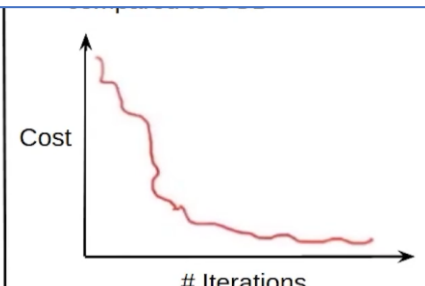
$$\Pr(Y_i = 1) = \frac{e^{\beta_1 \cdot \mathbf{X}_i}}{\sum_{0 \leq c \leq K} e^{\beta_c \cdot \mathbf{X}_i}} = \frac{e^{\beta_1 \cdot \mathbf{X}_i}}{e^{\beta_0 \cdot \mathbf{X}_i} + e^{\beta_1 \cdot \mathbf{X}_i}} = \frac{1}{e^{(\beta_0 - \beta_1) \cdot \mathbf{X}_i} + 1} = \frac{1}{1 + e^{-\beta \cdot \mathbf{X}_i}}$$

with $\beta = -(\beta_0 - \beta_1)$. We see that we obtain the same probabilities as in the two-class logistic regression using the sigmoid function. [Wikipedia](#) expands a bit more on that.

- Tips and Tricks to select an activation function.
 - A linear activation function:
 - Used for a regression problem.
 - It cannot capture the non-linear relation between input and output.
 - Used at the output layer.
 - A Sigmoid activation function:
 - Returns probabilities.
 - Used at the output layer.
 - ReLU and Tanh activation functions:
 - Non-linear functions
 - Used at the hidden layers.
 - ReLU is the most preferable hidden layer activation function.
 - SoftMax Activation function:
 - Returns probabilities for each class.
 - Used for multi-class classification.
 - Used at the output layer.

1.9. Variants for Gradient Descent

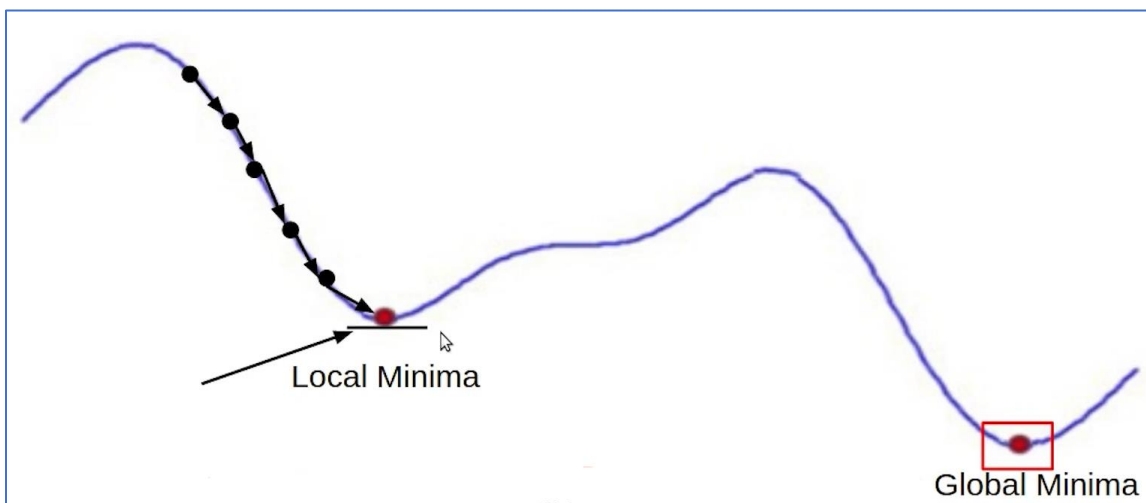


Batch Gradient Descent	Stochastic Gradient Descent (SGD)	Mini-Batch Gradient Descent
<ul style="list-style-type: none"> Entire dataset for updation Cost function reduces smoothly Computation cost is very high 	<ul style="list-style-type: none"> Single observation for updation Lot of variations in cost function Computation time is more 	<ul style="list-style-type: none"> Subset of data for updation Smoother cost function as compared to SGD Computation time is lesser than SGD Computation cost is lesser than Batch Gradient Descent
		

1.10. Optimizers

Problems with Gradient Descent:

- Getting stuck at local minima



- Gradient Descent faces a problem of local minima which can be solved using Momentum.
- Momentum is the accumulation of the previous gradients, and the equation is:

$$V_t = \beta V_{t-1} + (1 - \beta) \frac{dJ}{d\theta}$$
- With this accumulated speed there will be a chance of getting out of the local minima.
- Even at the local minima where $\frac{dJ}{d\theta}$ is equals to 0, there will be some factor to get out of the local minima.
- Now the updating equation becomes: $w_t = w_{t-1} - \alpha V_t$
- Generally, we consider $\beta=0.9$ which means providing 10% of weightage to the current gradient and 90% to previously accumulated weights.

- Same learning rate throughout the training process:
 - In the training process some variables might update a bit faster than other variables, but when we use the same learning rate, we are forcing them to be in sync.
 - So, as the training progresses, the learning rate will be changed as per the cost function.
 - Let's consider the cost function and define the process:

$$\eta = \sum_{i=1}^{t-1} \left[\frac{dJ}{d\theta} \right]^2$$

$$\theta_i = \theta_{i-1} - \frac{\alpha}{\sqrt{\eta}} \frac{dJ}{d\theta}$$

- In the above equation, we considered the square of gradients because the gradients might be positive and negative and can cancel each other. To avoid that we used the square.
- But when we use this equation, the gradient is always positive, and the second term will become infinitely small after a few iterations. This leads to $\theta_t \cong \theta_{t-1}$
- So, like momentum we use:

$$\mu_t = \beta \mu_{t-1} + (1-\beta) \left[\frac{dJ}{d\theta} \right]^2$$

$$\theta_i = \theta_i - \frac{\alpha}{\sqrt{\mu_t + \varepsilon}} \frac{dJ}{d\theta}$$

- Now, if the square of the gradients is high then the learning rate will be low and vice versa. It helps to reduce the learning rate when is nearing the minimum.
- Adam = SGD with Momentum + RMSProp

Adam

=

SGD with Momentum

+

RMSProp

$$V_t = \beta_1 V_{t-1} + (1-\beta_1) \frac{dJ}{d\theta}$$

$$\mu_t = \beta_2 \mu_{t-1} + (1-\beta_2) \left[\frac{dJ}{d\theta} \right]^2$$

$$\theta_i = \theta_{i-1} - \frac{\alpha}{\sqrt{\mu_t + \varepsilon}} * V_t$$

- One of the most used optimizers.
- There are other optimizers like
 - Nestrov Accelerated Gradient Descent
 - $W_look_ahead = W_{t-1} - \text{beta} * \text{velocity}_{t-1}$
 - $\text{Velocity}_t = \text{beta} * \text{velocity}_{t-1} + \text{eta} * dJ \text{ wrt } W_look_ahead$
 - $W_t = W_{t-1} - \text{velocity}_t$

$$v_t = \gamma v_{t-1} + \eta \nabla J(\theta_{t-1} - \gamma v_{t-1})$$

$$\theta_t = \theta_{t-1} - v_t$$

- AdaGrad

1.11. Loss Functions

For Regression:

- Mean Squared Error: $MSE = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$
- Mean Absolute Error: $MAE = \frac{1}{N} \sum_{i=1}^N |y_i - \hat{y}_i|$
- Root Mean Squared Error: $RMSE = \sqrt{\frac{\sum_{i=1}^N (y_i - \hat{y}_i)^2}{N}}$

For Classification:

- Binary Cross Entropy/Log Loss: $BCE = -\frac{1}{N} \sum_{i=1}^N (y_i * \log p_i + (1 - y_i) * \log(1 - p_i))$
 - We use log here to less penalize the smaller values and more penalize the high values
- Categorical Cross Entropy: $logloss = -\frac{1}{N} \sum_i \sum_j y_{ij} * \log(p_{ij})$

1.12.

What is batch normalization?

Batch normalization is a technique used in deep neural networks to improve training stability and accelerate convergence. It involves normalizing the activations of each layer within a mini-batch, typically before applying the activation function. The normalization is performed by subtracting the batch mean and dividing by the batch standard deviation, followed by scaling and shifting using learnable parameters.

How Batch Normalization Works:

- 1. Normalization:**
 - For each feature in the input, calculate the mean and standard deviation across the mini-batch.
 - Normalize the feature by subtracting the mean and dividing it by the standard deviation.
 - This centers the distribution of each feature around zero and scales it to have unit variance.
- 2. Scaling and Shifting:**
 - Introduce learnable parameters (γ and β) for each feature.
 - Scale and shift the normalized features using these parameters.
 - This allows the model to learn the optimal scaling and shifting for each feature.
- 3. Batch Statistics:**
 - During training, the mean and standard deviation are computed for each mini-batch.
 - During inference, running averages of these statistics are used to normalize the input.

Benefits of Batch Normalization:

- 1. Stabilizes Training:**
 - Helps mitigate the vanishing or exploding gradients problem by ensuring that activations are within a reasonable range throughout the network.
- 2. Accelerates Convergence:**
 - Reduces the dependence of gradients on the scale of parameters, allowing for faster convergence and reduced training time.
- 3. Improves Generalization:**
 - Acts as a form of regularization by adding noise to the network, which can help prevent overfitting and improve generalization performance.
- 4. Enables Higher Learning Rates:**
 - Allows for the use of higher learning rates, which can speed up training and potentially lead to better performance.
- 5. Reduces Sensitivity to Initialization:**
 - Makes deep networks less sensitive to the choice of initialization parameters, allowing for more stable training across different architectures.

Implementation Considerations:

- Batch normalization is typically applied before the activation function in each layer of the network.
- It is often used in conjunction with other regularization techniques such as dropout to further improve generalization.
- Batch normalization can be applied to different types of layers, including fully connected layers, convolutional layers, and recurrent layers.

Overall, batch normalization is a powerful technique that plays a crucial role in training deep neural networks by addressing issues related to internal covariate shift and improving the stability and efficiency of the optimization process.

Regularization Techniques used in Deep Neural Networks

Regularization techniques are essential in preventing overfitting and improving the generalization performance of neural networks. Here are some common regularization techniques used in neural networks:

1. **L1 and L2 Regularization (Weight Decay):**
 - L1 regularization adds a penalty term to the loss function proportional to the absolute value of the weights.
 - L2 regularization adds a penalty term proportional to the square of the weights.
 - Both techniques discourage large weights and encourage the model to learn simpler patterns.
 - The regularization term is typically added to the loss function with a hyperparameter (λ) controlling its strength.
2. **Dropout:**
 - Dropout is a regularization technique that randomly sets a fraction of input units to zero during training.
 - This prevents units from co-adapting too much and acts as an ensemble method by training multiple models with different subsets of the input units.
 - Dropout is applied independently to each layer during training and turned off during inference.
3. **Batch Normalization:**
 - Batch normalization not only stabilizes training by normalizing activations but also acts as a form of regularization.
 - By reducing the internal covariate shift and adding noise to the network, batch normalization helps prevent overfitting and improves generalization.
4. **Early Stopping:**
 - Early stopping involves monitoring the validation loss during training and stopping training when the validation loss starts to increase.
 - This prevents the model from overfitting to the training data by halting training before it starts to memorize noise.
5. **Data Augmentation:**
 - Data augmentation involves generating additional training examples by applying transformations such as rotations, translations, scaling, and flipping to the existing data.
 - This helps increase the size and diversity of the training dataset, making the model more robust to variations in the input data.
6. **DropConnect:**
 - DropConnect is a generalization of dropout that randomly sets a fraction of weights to zero instead of input units.
 - It encourages sparsity in the weight matrix and prevents co-adaptation of weights.
7. **Weight Sharing and Parameter Tying:**
 - Weight sharing and parameter tying constrain the model by sharing weights or tying parameters across different parts of the network.
 - This reduces the number of trainable parameters and helps prevent overfitting, especially in cases of limited training data.
8. **Noise Injection:**
 - Noise injection involves adding random noise to the input data or hidden layers during training.
 - This helps regularize the model by adding stochasticity and preventing it from overfitting to the training data.

By employing these regularization techniques, practitioners can effectively combat overfitting and build neural network models that generalize well to unseen data. It's common to use a combination of these techniques to achieve better performance and robustness.

