# Perceptron

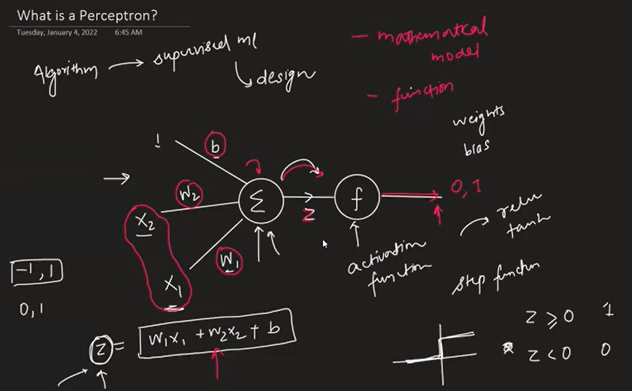
A perceptron is a basic building block of a neural network, representing a simplified model of a biological neuron. It is an algorithm used for supervised machine learning tasks, particularly binary classification problems.

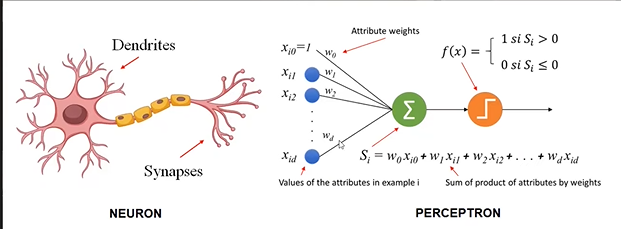
## Mathematical model

The perceptron’s design involves a mathematical model that takes multiple inputs and produces a single binary output (. Each input is associated with a weight representing the strength or importance of that input. Additionally, there is a bias term

In simpler terms, the perceptron takes the weighted sum of its inputs, adds a bias, and produces an output based on whether this sum is greater than zero or not. Mathematically, can be written as:

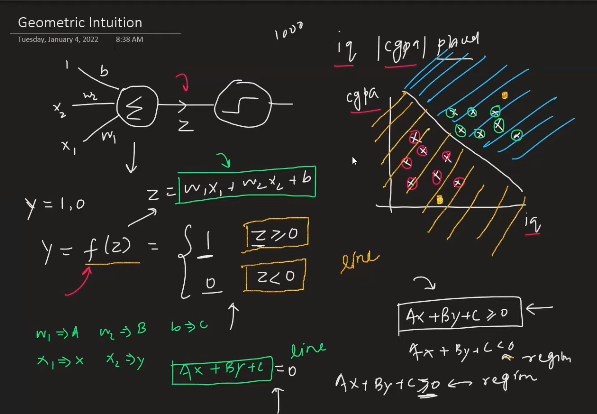
Here, the step function is a threshold function that output 1 if the sum is greater than zero and 0 otherwise.





Inputs are received through Dendrites and through Axon output is sent. Nucleus is where calculation is performed where step and activation function is performed.

The weights represent the strength of the feature and the output of the data is dependent on feature. From the below image we can say that the perceptron is the binary classification model which will separates the class basis on line and similarly in 3dimensional it will act as plane and in more than 4-dimensional space it will separate the data with hyperplane. In simple terms, Perceptron model is only giving good results if the binary classification problem where the data is linearly separable.

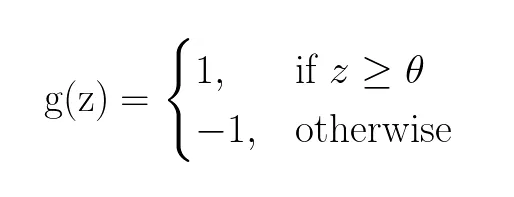


## Train Perceptron trick

Initialize with the random weights which will be assigned to each input variable and it will enter into loop which will classify the classes into two group. In every loop, the nearest misclassified point will adjust the weight of the independent variable which will correctly classify the misclassified point in succeeding loop and it will follow this until convergence.

The training process of a perceptron involves adjusting its weights iteratively to correctly classify training examples. The algorithm follows a loop until convergence, where weights are updated based on the errors made during the classification. Below is the step-by-step breakdown:

1. Initialization:
   * Initialize weights with random values
   * Initialize the bias term
2. Iterative training loop:
   * For each input example in the training dataset
   * Compute the weighted sum .
   * Apply the step function to to get the predicted output: .



* + Compare the predicted output to the actual output .
    - If is correct, no adjustment is needed.
    - If is incorrect, update the weights and bias:
      * Update weights: for each .
      * Update bias:

1. Convergence:
   * Repeat the iterative loop until convergence, where convergence could be defined by a certain number of iteration or until no misclassification occur.

## Loss functions:

In the context of perceptrons, the choice of a loss function is crucial for training the model. The goal during training is to adjust the weights and biases to minimize the chosen loss function, thereby improving the model's ability to make accurate predictions.

A common loss function used for the perceptrons is the perceptron loss function. The perceptron loss function is defined as:

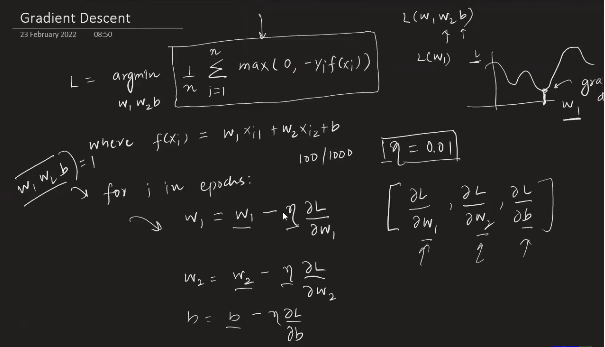
Here:

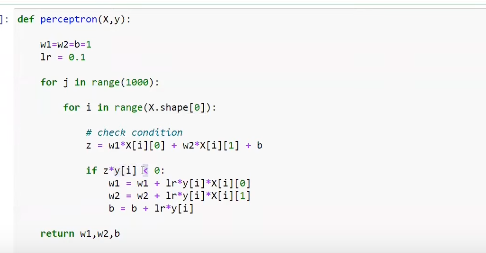
is the true output label (either 1 or -1).

is the predicted output from the perceptron.

The perceptron loss function penalizes the model when the predicted output is far from the true label. If the sign of the predicted and true outputs match, the loss is 0. If they have different sign, the loss is proportional to the magnitude of the difference.

The perceptron learning rules involves updating the weights based on the derivative of the perceptron loss with respect to the weights. Since, the perceptron loss is not differentiable, sub-gradient methods are employed.





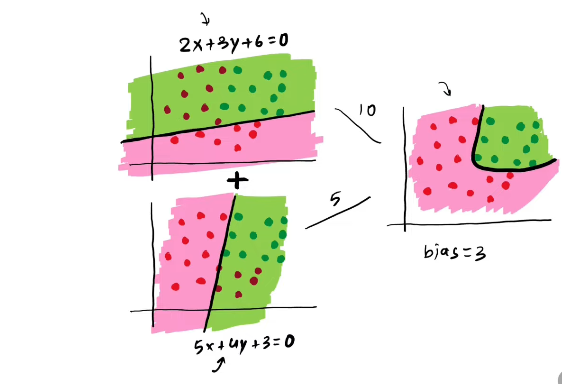
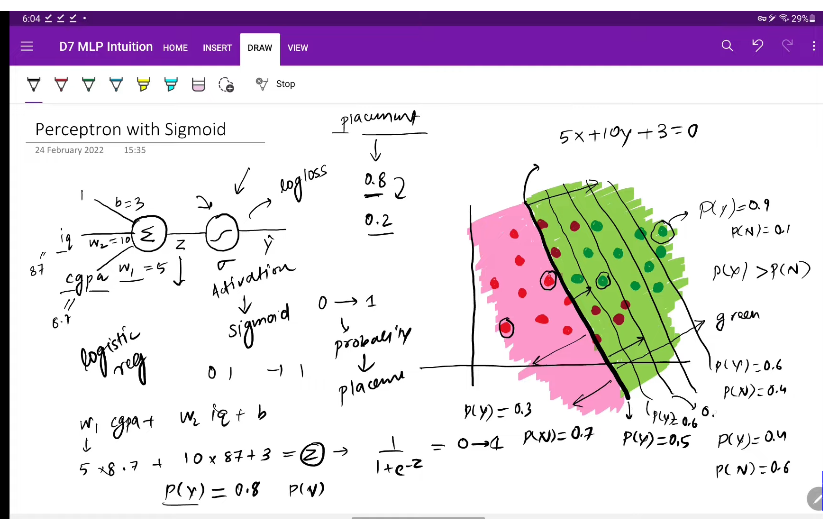
# Multi-Layer perceptron

## Multi-Layer Perceptron Notation

## Multi-Layer Perceptron-Intution

1. **Single Perceptron and Linearity:**
   * A single perceptron is a linear model, meaning it can only learn and represent linear decision boundaries.
   * It's limited to problems that are linearly separable, where classes can be distinguished by a straight line.
2. **Multi-Layer Perceptron (MLP) and Non-Linearity:**
   * An MLP consists of multiple layers, including an input layer, one or more hidden layers, and an output layer.
   * The addition of hidden layers allows the network to learn complex, non-linear relationships in the data.
3. **Sigmoid Activation Function:**
   * The use of the sigmoid activation function in the output layer allows the network to model probabilities. It squashes the output into the range (0, 1), representing probabilities of belonging to a particular class.
4. **Combining Probabilities:**
   * In multi-class classification, multiple nodes in the output layer correspond to different classes.
   * The softmax activation function is often used to convert raw scores (logits) into probabilities, ensuring that the probabilities sum to 1.
5. **Adjusting Weights in Hidden Layers:**
   * The backpropagation algorithm, with the help of optimization techniques like gradient descent, is used to adjust the weights in hidden layers during training.
   * This adjustment allows the network to learn and capture complex patterns, enabling it to create non-linear decision boundaries.
6. **Adding Nodes in Hidden Layers:**
   * The number of nodes in the hidden layers is a hyperparameter that can be adjusted based on the complexity of the problem.
   * Increasing the number of nodes can gives ability to capture intricate (complicated) patterns in the data.
7. **Hidden Layers for Non-Linear Relationships:**
   * The purpose of hidden layers is to capture and transform the input features into a higher-dimensional space, allowing the network to learn non-linear relationships.

In summary, the power of MLPs lies in their ability to model non-linear decision boundaries through the inclusion of hidden layers and activation functions. The training process involves adjusting the weights in these layers to optimize the model for the specific task at hand. This flexibility makes MLPs well-suited for a wide range of machine learning tasks, including complex pattern recognition and classification problems.



## Forward propogation

Forward propagation is the process by which the input data is passed through the layers of neural network, resulting int the computation of the final output. It involves a series of calculations in which the input is transformed, as it moves through the network, layer by layer, until the output is obtained.

* Input layer
  + The process begins with the input layer, which consists of nodes corresponding the features of the input data.
  + Each node in the input layer represents a feature, and the values of these nodes are the input features of the data
* Weighted Sum and Activation Function (Hidden layers):
  + For each node in the hidden layers and output layer, a weighted sum is computed.
  + The weighted sum for a node in hidden layer is calculated as:
* Here,is the weight matrix for layer , is the activation output of the previous layer, and is the bias term.
* The weighted sum is then passed through an activation function to introduce non-linearity. Common activation functions include sigmoid, tanh, and ReLU.
* Activation Output:
  + The result of applying activation function to weighted sum is the activation output for each node.
  + For example, if
* Repeat for each layer:
  + The process of computing the weighted sum, applying the activation function, and obtaining the activation output is repeated in each layer in the network until the final layer is reached.
* Output Layer:
  + The final layer’s activation output represents the network’s prediction or output.
  + Depending on the task (e.g., classification or regression), the activation function in the output layer may vary. For binary classification, a sigmoid activation function is often used, while softmax is common for multi-class classification.
* Forward Propagation Equation:
  + In a more compact form, the forward propagation equation for a layer can be expressed as:

Forward propagation is a crucial step in both training and making prediction with neural networks. During training, it computes the output that model predicts for a given input, and this output is compared to the actual target values to calculate the loss. The loss is then used in backward propagation step to update the model’s parameters and improve its performance over time.

## Loss function:

The loss function is a crucial element in training machine learning models. It measures how well the model's predictions align with the actual data. For example, in linear regression, Mean Squared Error (MSE) is a common loss function.

Here's why loss functions matter:

1. **Performance Evaluation:**
   * The loss function serves as a metric to evaluate how well the model is performing. High loss indicates poor performance, while low loss indicates better alignment with the data.
2. **Optimization and Learning:**
   * During training, the goal is to minimize the loss. Optimization algorithms like gradient descent adjust the model's parameters (weights and biases) to reduce the error and improve predictions.
3. **Feedback Signal:**
   * The loss function provides a feedback signal for learning. By computing gradients with respect to model parameters, it guides the model's updates in a direction that reduces the error.

In deep learning:

1. **Initialization:**
   * Deep learning models start with random weights and biases. The loss function helps assess the model's performance with these initial parameters.
2. **Training Iterations:**
   * In each training iteration, forward propagation computes the output, the loss is calculated, and backward propagation adjusts the parameters using gradient descent.
3. **Iterative Improvement:**
   * The iterative process continues until the model converges to a state where the loss is minimized, indicating that the model has learned the patterns in the data.

In summary, the choice of a suitable loss function is essential for effective model training. It provides a measurable metric to guide improvements and ensures that the model learns to make accurate predictions. The quote "you can't improve what you can't measure" underscores the importance of having a measurable metric like the loss function in the learning process.

## Loss Function vs. Cost Function:

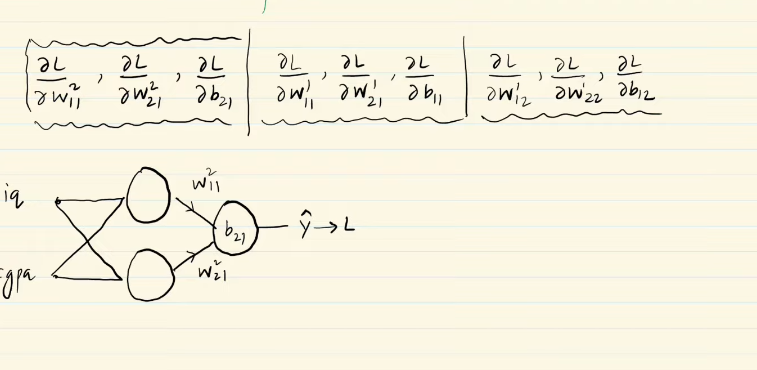
* **Loss Function:** Typically refers to the error for a single training example. In the context of neural networks, it measures the difference between the predicted output and the true target for a single data point.
* **Cost Function (or Objective Function):** The average loss over the entire training dataset. It's the sum of the losses for all instances divided by the number of instances. The cost function provides a global measure of how well the model is performing across the entire dataset.

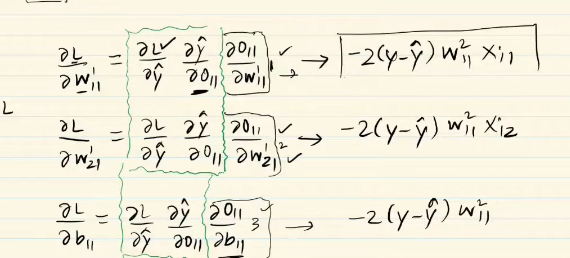
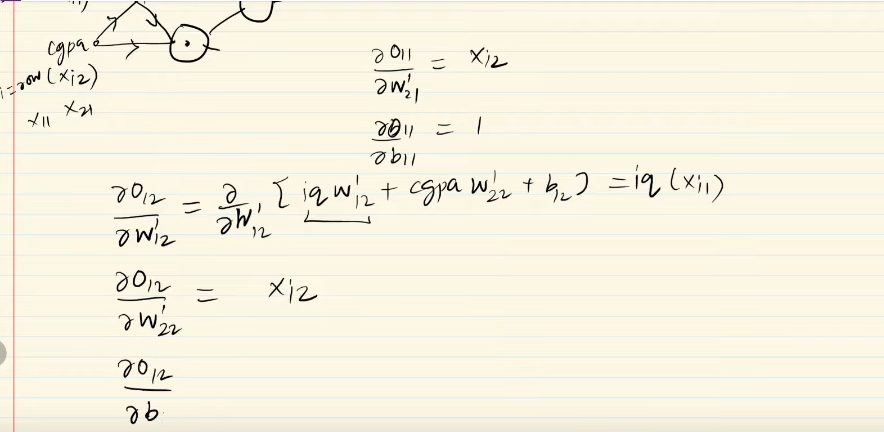
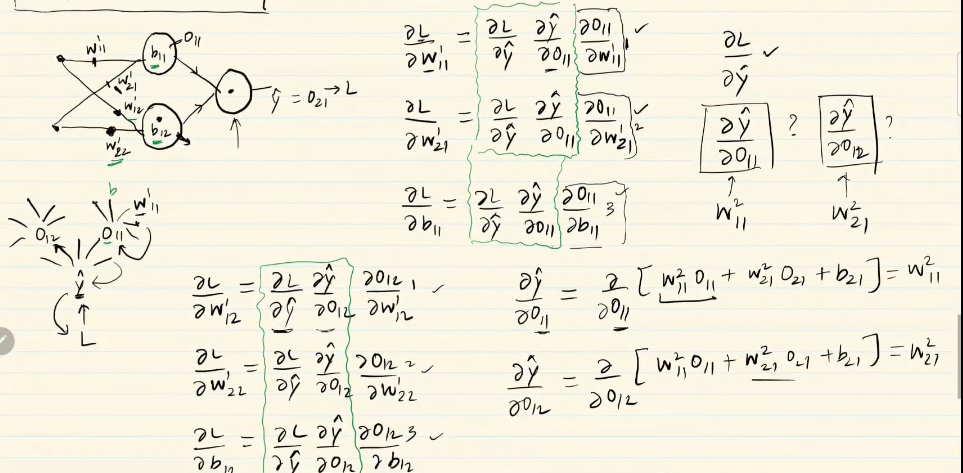
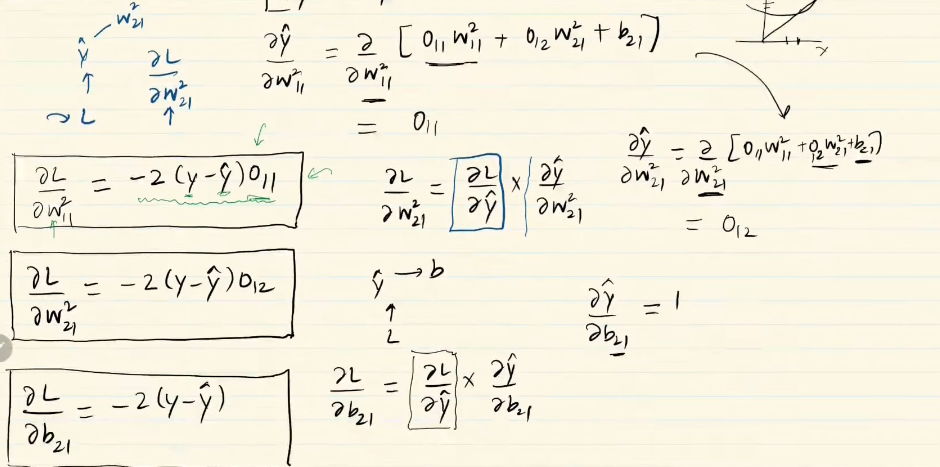
## Types of Loss Functions:

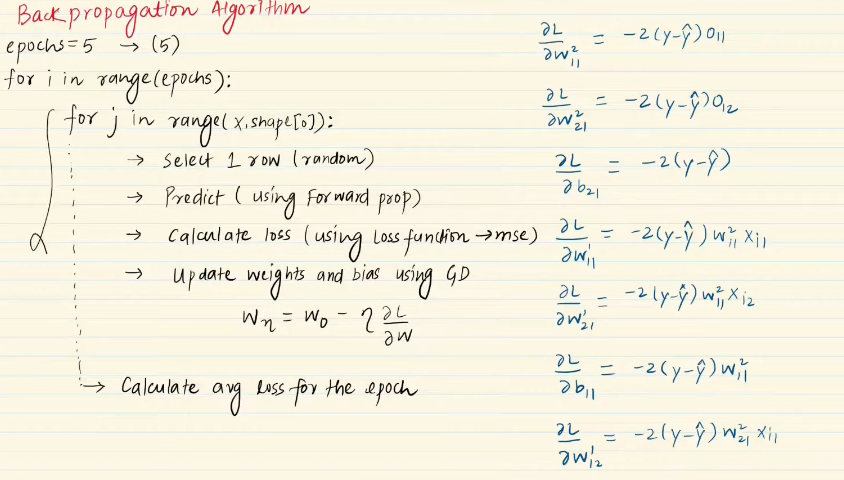
1. **Mean Squared Error (MSE) L2 Loss:**
   * **Advantages:**
     + Easy to interpret.
     + Differentiable.
     + Has only one local minimum.
   * **Disadvantages:**
     + Sensitive to outliers.
2. **Mean Absolute Error (MAE) L1 Loss:**
   * **Advantages:**
     + Intuitive and easy to understand.
     + Robust to outliers.
     + Units are the same as the output.
   * **Disadvantages:**
     + Not differentiable.
3. **Huber Loss:**
   * A compromise between MSE and MAE. It behaves like MSE for small errors and like MAE for large errors, making it more robust to outliers.
4. **Binary Cross Entropy:**
   * Used in binary classification problems.
   * **Advantages:**
     + Differentiable.
   * **Disadvantages:**
     + Multiple local minima.
     + Not intuitive.
5. **Categorical Cross Entropy:**
   * Used in multi-class classification problems (more than two classes).
   * Requires the use of softmax activation function.
   * Output neurons should be equal to the number of distinct classes.
6. **Sparse Categorical Cross Entropy:**
   * Similar to categorical cross entropy but is used when the target variable is represented as integers (class indices) rather than one-hot encoded vectors.

## Backpropagation

It is an algorithms to train neural network. Assume we have a regression problem. In this case initial weight is assigned randomly for weight and biases. Basis on it the prediction is made. Choose a loss function and evaluate it. To reduce the loss the weights and biases are corrected with the help of Gradient descent.







Backpropagation, short for "backward propagation of errors," is a fundamental algorithm used in training artificial neural networks, particularly multi-layer perceptrons (MLPs). It enables neural networks to learn from training data by iteratively adjusting their weights and biases to minimize a defined loss function.

Here's how backpropagation works:

1. **Forward Pass:** During the forward pass, input data is fed into the neural network, and computations are performed layer by layer to generate predictions. Each layer applies a linear transformation (weighted sum of inputs) followed by a non-linear activation function.
2. **Loss Calculation:** Once predictions are generated, they are compared to the actual targets, and a loss function is computed to quantify the discrepancy between predictions and targets. Common loss functions include mean squared error (MSE) for regression tasks and cross-entropy loss for classification tasks.
3. **Backward Pass (Gradient Calculation):** In the backward pass, gradients of the loss function with respect to the network's parameters (weights and biases) are computed using the chain rule of calculus. This involves propagating the error backwards through the network layer by layer.
4. **Gradient Descent:** Once gradients are computed, optimization algorithms such as gradient descent are used to update the parameters in the opposite direction of the gradient, aiming to minimize the loss function. The magnitude of the updates is determined by the learning rate hyperparameter.
5. **Iteration:** The process of forward pass, backward pass, and parameter updates is repeated iteratively for multiple epochs until convergence criteria are met (e.g., when the loss function converges or stops decreasing).

Backpropagation allows neural networks to learn complex patterns and relationships in data by iteratively adjusting their parameters based on observed errors. It is a foundational concept in deep learning and has enabled the training of deep neural networks with multiple layers, leading to breakthroughs in various fields such as computer vision, natural language processing, and reinforcement learning.

## Memoization

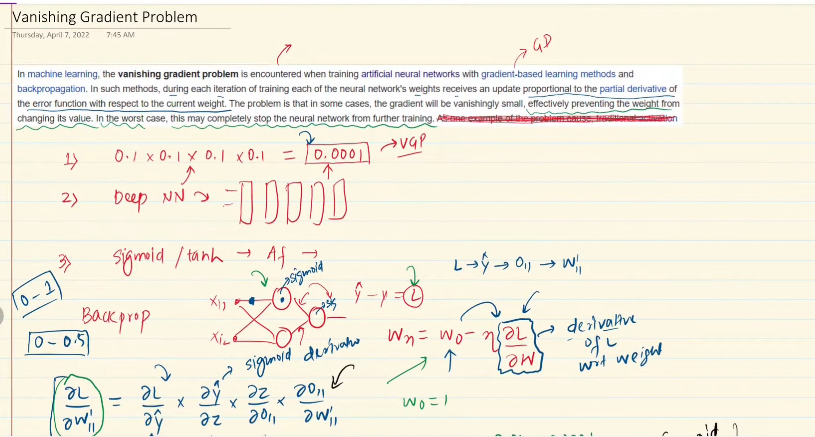
Memoization is an optimization technique used to primarily speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.

Working of memoization:

* Caching results: When a function is called with certain inputs, the result of that function call is stored (cached) in a data structure like a dictionary or an array
* Checking Cache: before performing a potentially expensive computation, the function first checks if the result for the given inputs is already available in the cache.
* Returning Cached Result: If the result is found in the cache, it is returned immediately without re-computations. This avoids redundant calculations and improves the overall performance of the algorithm.
* Computing and Storing New Results: If the result in not found in the cache, the function computes the result as usual and stores it in the cache for future use.

## Gradient Descent

* Gradient descent is an optimization algorithm used to minimize an objective(cost) function by adjusting the parameters of a model in the opposite direction of the gradient of the objective function with respect to the parameters.
* The learning rate determines the size of steps taken during optimization.
* **Types of Gradient Descent:**
* **Batch GD:** Uses the entire dataset to compute the gradient and update parameters once per epoch.
* **Stochastic GD (SGD):** Updates parameters after each training example, making it faster but noisier.
* **Mini-batch GD:** Utilizes a subset of the dataset (batch) to compute the gradient and update parameters, striking a balance between batch GD and SGD.
* **Vanishing Gradient Problem:**
* Occurs when gradients become very small during backpropagation, hindering learning in deep neural networks.
* Commonly associated with sigmoid and tanh activation functions due to their limited range.
* **Handling Vanishing Gradient Problem:**
* Using ReLU activation function, which doesn't suffer from vanishing gradients.
* Use shallow network, but it doesn’t work if the objective is to identify the complex relationship in dataset.
* Proper weight initialization techniques.
* Batch normalization to stabilize gradients.
* Residual connections to facilitate gradient flow.



* **Improving Neural Network Performance:**
* Tuning hyperparameters like the number of layers, neurons, learning rate, optimizer, batch size, etc.
  + There should be sufficient number of neurons in initial hidden layer to capture the primitive features.
  + We can do warming up learning rate where we can increase the learning rate in training to improve the performance.
  + We can provide high number of epochs with help of Early stopping which will stop the iterations if there is no significant change in the loss.
* Monitoring loss rate and weight distributions.
* Utilizing transfer learning and unsupervised pre-training when data is limited.
* Employing regularization techniques like L1 and L2 regularization, dropout, for overfitting model.

**Other Common Challenges and Solutions:**

* Exploding gradient problem can be addressed by gradient clipping.
* Not enough data can be mitigated by data augmentation, transfer learning, or unsupervised pre-training.
* Slow training can be accelerated using more efficient optimizers or learning rate schedulers.

Why batch size is provided in multiple of 2 in Neural networks.

Providing a batch size that is a multiple of 2 (e.g., 32, 64, 128) is a common practice in neural networks for several reasons:

1. **Efficiency in Memory Utilization:** Modern hardware architectures, such as GPUs, are optimized for parallel processing and memory access. Using batch sizes that are powers of 2 can help maximize memory utilization and optimize data throughput during training.
2. **Vectorization and Parallelization:** Neural network operations, such as matrix multiplications and convolutions, are often implemented using highly optimized libraries that leverage parallel processing capabilities. Batch sizes that are powers of 2 can facilitate efficient vectorization and parallelization of these operations, leading to faster training times.

## Early Stopping

The early stopping mechanism allows training to halt at a point where the model's performance on the validation set is optimal, preventing further training that could lead to overfitting. By monitoring the specified quantity and observing changes over epochs, early stopping helps find the optimal balance between training performance and generalization to unseen data.

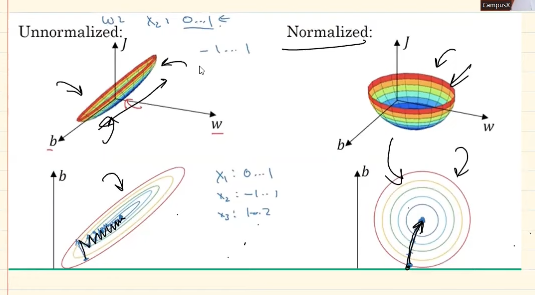
Implementing early stopping in training neural networks can help improve model performance, prevent overfitting, and save computational resources by stopping training when further improvement is unlikely.

Parameters of Early Stopping

1. **Min\_delta:** This parameter specifies the minimum change in the monitored quantity (e.g., validation loss) that is considered as an improvement. If the change in the monitored quantity is less than min\_delta, it is considered negligible, and the training may be stopped.
2. **Monitor:** This parameter specifies the quantity to be monitored during training to determine whether to stop early or continue. Common choices include validation loss, validation accuracy, or any other relevant metric that indicates model performance.
3. **Patience:** Patience determines the number of epochs to wait before considering early stopping. If the monitored quantity does not show improvement for a certain number of epochs (specified by patience), training may be stopped.

.

## Normalization:



1. **Balanced Update Steps:** When features have significantly different scales, the gradients for each feature may also vary widely. This can cause the optimization process to take longer to converge, as some features may dominate the updates while others are neglected. Feature scaling ensures that all features contribute more equally to the update steps, leading to more balanced progress towards the optimal solution.
2. **Avoidance of Oscillations:** In gradient descent, oscillations can occur when the learning rate is too high, causing the optimization process to overshoot the minimum and bounce back and forth around the optimal solution. Feature scaling helps mitigate oscillations by ensuring that the gradients are consistent across features, reducing the likelihood of large updates that lead to overshooting.
3. **Faster Convergence:** With balanced update steps and reduced oscillations, feature scaling generally leads to faster convergence of the optimization process. This means that the model reaches a satisfactory solution in fewer iterations, saving computational resources and training time.
4. **Improved Numerical Stability:** Feature scaling can also improve the numerical stability of optimization algorithms. Large differences in feature scales can result in numerical instabilities, such as overflow or underflow issues, which can disrupt the optimization process. Scaling features to a similar range helps maintain numerical stability throughout training.

## Dropout

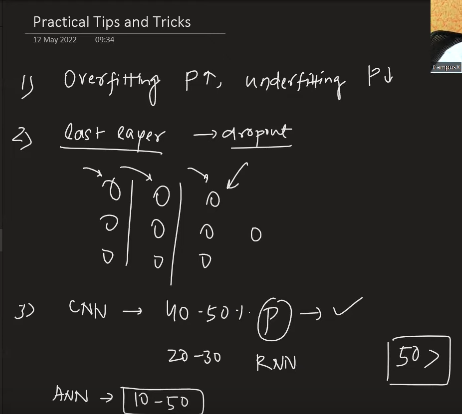
Neural networks are prone to overfitting due to complex architecture. To overcome with this solution is

* Add more data
* Reduce complexity of network by reducing layers and neurons.
* Early stopping
* Regularization

Dropout is a regularization technique commonly used in deep learning to prevent overfitting and improve the generalization performance of neural networks. It works by randomly dropping (i.e., setting to zero) a fraction of the neurons in a layer during training.

How Dropout works:

* **Random Dropout**: During each training iteration, dropout randomly selects a subset of neurons in a layer and temporarily removes them from the network. The dropout rate determines the probability of dropping out each neuron, typically ranging from 0.2 to 0.5
* **Training Phase**: in the forward pass, the network operates with the dropout applied, an only a fraction of neurons is active. This means that the networks predictions are based on different subnetwork configuration in each iteration.
* **Inference Phase**: During inference/testing, dropout is turned off, and all neurons are active. However, the weights of neurons are scaled by the dropout rate to compensate for the increased activity, ensuring consistent behaviour between training and testing.
* **Regularization Effect**: By randomly dropping neurons during training, dropout introduces noise and variability into the network, forcing it to learn more robust and generalized features. This prevents individual neurons from relying too heavily on specific features and encourages the network to learn redundant representations, which helps prevent overfitting.
* **Ensemble Effect**: Dropout can be viewed as training multiple subnetworks simultaneously, each corresponding to a different combination of active neurons. During inference, these subnetworks are effectively combined, leading to an ensemble effect that improves the model’s generalization performance.
* **Scaling Weights in testing:**
  + To compensate for the dropout applied during training, the weights of each neuron are scaled by the dropout rate (probability of dropout) during testing.
  + For example, if the dropout rate used during training was 0.5 (50% dropout probability), the weights of each neuron are multiplied by 0.5 during testing.



## Regularization L2 & L1

Regularization techniques such as L1 and L2 regularization are commonly used in deep learning to prevent the overfitting and improve the generalization performance of neural networks.

* L1 Regularization (Lasso Regularization):
  + L1 regularization adds a penalty term to the cost function proportion to the absolute values of the weights (parameters) of the neural network.
  + The regularization is term is calculated as the sum of absolute values of the weights where is the regularization parameter and are the weights.
  + L1 regularization encourages sparsity in the weight matrix by pushing some of the weights exactly zero
* L2 Regularization (Ridge Regularization):
  + L2 regularization adds a penalty term to the cost function proportional to the square of the Euclidean norm (L2 norm) of the weights of the neural network.
  + The regularization term is called as the sum of the square of the weights: , where is the regularization parameter and are the weights.
  + L2 regularization penalized large weights more heavily than small weights, encouraging the network to spread influence of all features rather than relying heavily on a few.

Both L1 and L2 regularization techniques add a regularization term to the cost function, which penalizes complex models with large weights. The regularization parameter λλ controls the strength of regularization: larger values of λ result in stronger regularization, which can help prevent overfitting but may also lead to underfitting if set too high.

In practice, a combination of L1 and L2 regularization, known as Elastic Net regularization, can be used to benefit from the advantages of both techniques. Regularization techniques like L1 and L2 are essential tools in the deep learning practitioner's toolbox for building more robust and generalizable neural networks.

## Activation function

In Neural networks, each neuron forms a weighted sum of its inputs and passes the resulting scalar value through a function referred to as an activation function or transfer function. If a neuron has n inputs, then the output or activation of a neuron is

The function is referred to as the activation function.

### Ideal activation function

1. **Non-Linear:** Capable of capturing nonlinear relationships between inputs and outputs, allowing the network to learn complex patterns and make accurate predictions.
2. **Differentiable:** Must be differentiable to facilitate the use of gradient-based optimization algorithms like backpropagation, which rely on computing derivatives to update weights during training.
3. **Computationally Inexpensive:** Calculating activations and their derivatives should be computationally efficient to enable faster training and inference.
4. **Zero-Centered:** Having a mean-centered around zero helps with the convergence of optimization algorithms, particularly gradient descent.
5. **Non-Saturating:** Avoids saturation issues where the gradient becomes close to zero, preventing learning. Non-saturating activation functions can sustain learning even with large input values.

### Common Activation Functions:

1. **Sigmoid Function (Logistic Activation):**
   * Advantages:
     + Maps inputs to a range between 0 and 1, making it suitable for binary classification problems.
     + Non-linear and differentiable, allowing for gradient-based optimization.
   * Disadvantages:
     + Prone to vanishing gradient problem due to saturation, limiting its effectiveness in deep networks.
     + Computationally expensive due to exponentiation operations.
2. **Tanh Function (Hyperbolic Tangent Activation):**
   * Advantages:
     + Similar to the sigmoid but maps inputs to a range between -1 and 1, aiding in better convergence.
     + Non-linear, differentiable, and zero-centered, making it suitable for hidden layers.
   * Disadvantages:
     + Shares some drawbacks with the sigmoid, including vanishing gradient and computational expense.
3. **ReLU (Rectified Linear Unit):**
   * Advantages:
     + Simple and computationally efficient, replacing negative values with zero.
     + Non-linear and non-saturating, promoting faster convergence and alleviating vanishing gradient issues.
   * Disadvantages:
     + Not zero-centered, which may lead to dead neurons during training (dying ReLU).
     + Not differentiable at zero, although this rarely poses a problem in practice.

The "dying ReLU" problem refers to a scenario in neural networks where ReLU (Rectified Linear Unit) activation functions result in neurons always outputting zero, effectively becoming "dead" and not contributing to the learning process. This issue arises when the input to a ReLU neuron is consistently negative, causing the neuron to output zero due to the rectification operation.

**Causes:**

* **High Learning Rate:** Rapid changes in weights during training can push ReLU neurons into negative territory, leading to dead neurons.
* **High Negative Bias:** Biases that heavily bias activations towards negative values can also contribute to dead neurons.

**Solutions:**

* **Lower Learning Rate:** Using a lower learning rate can help stabilize weight updates and prevent abrupt changes that lead to dead neurons.
* **Positive Bias Initialization:** Initializing biases with positive values can prevent neurons from starting in a negative regime, reducing the likelihood of dying ReLU.
* **Alternative Activation Functions:** Consider using alternative activation functions that do not suffer from the dying ReLU problem, such as Leaky ReLU, Parametric ReLU, or ELU.

**Variants of Relu**

1. **Leaky ReLU:**
   * Introduces a small slope (typically 0.01) for negative inputs to prevent dead neurons.
   * Advantages: Non-saturating, easily computed, and overcomes the dying ReLU problem.
   * Disadvantages: Fixed leakage parameter, which may not be optimal for all scenarios.
2. **Parametric ReLU:**
   * Introduces a learnable parameter aa that determines the slope for negative inputs.
   * Advantages: Allows the network to learn the optimal leakage parameter, potentially addressing the dying ReLU problem more effectively.
   * Disadvantages: Adds additional trainable parameters, increasing model complexity.
3. **ELU (Exponential Linear Unit):**
   * Applies an exponential function to negative inputs to ensure smoothness and prevent dead neurons.
   * Advantages: Zero-centered, faster convergence, and no dying ReLU problem.
   * Disadvantages: Computationally more expensive due to the exponential function.
4. **SELU (Scaled Exponential Linear Unit):**
   * Self-normalizing variant of ELU that maintains mean and variance stability across layers.
   * Advantages: Self-normalizing behaviour, faster convergence, and prevention of dead neurons.
   * Disadvantages: Requires careful initialization and scaling to maintain stability.

## Weight Initialization

Weight initialization in deep learning refers to the process of assigning initial values to the weights of the neural network layers. It is a crucial step in training neural networks as it can significantly affect the convergence speed, stability, and overall performance of the model. Proper weight initialization helps prevent issues such as vanishing or exploding gradients, which can hinder the training process.

When initializing weights in a neural network, it's essential to strike a balance between random initialization and ensuring that the weights are not too large or too small. Here are some common techniques used for weight initialization:

**Common Weight Initialization Techniques:**

1. **Zero Initialization:**
   * Initializing all weights to zero is generally not recommended as it leads to symmetry in the network and may cause all neurons to compute the same output.
2. **Random Initialization:**
   * Assigning random values to weights from a uniform or normal distribution.
   * Small random weights can lead to vanishing gradients, while large random weights can cause exploding gradients.
3. **Xavier/Glorot Initialization:**
   * The weights are initialized using a Gaussian or uniform distribution with variance scaled based on the number of input and output units. Xavier initialization is well-suited for activation functions like tanh.
     1. **Uniform Xavier initialization**: draw each weight w from a random uniform distribution in [-x,x] for x =
     2. **Normal Xavier initialization:** draw each weight, w, from a normal distribution with a mean of 0 and a standard deviation
4. **He Normal Initialization:**
   * A variant of Xavier initialization specifically designed for ReLU activation functions. It scales the initial weights using a Gaussian distribution with variance adjusted based on the number of input units. He initialization helps prevent dead ReLU issues and promotes faster convergence.
     1. **Normal He initialization:** draw each weight w from a random normal distribution in [-x,x] for x =

## Batch Normalization

It is an algorithmic method which makes the training of Deep Neural Networks faster and more stable. It consists of normalizaing activation vectores from hidden layers using the mean and variance of the current batch this normalization step is applied before (or right after) the nonlinear function.

There is internal covariate shift which is reduces with the batch normalization.

Batch normalization is applied based on mini-batch GD and it’s done by layer by layer.

### **How Batch Normalization Works**

Suppose we have a neural network with multiple hidden layers and we want to apply batch normalization to improve its training. Here's how batch normalization would work in this scenario

* **Normalization Step:**
  + During the training process, we divide the training dataset into mini-batches. Let's say each mini-batch contains 64 images.
  + For each mini-batch, we compute the mean and variance of the activations for each hidden layer.
  + Then, we normalize the activations within the mini-batch using the computed mean and variance.
  + Mathematically, the normalization process can be represented as: where is the normalized activation, is the original activation, is the mean, is the variance, and ϵ is a small constant to prevent division by zero.
* **Application Before Activation Function:**
  + We apply batch normalization before the activation function of each hidden layer. This ensures that the normalized activations are passed through the activation function.
  + Common activation functions used in this context include ReLU, sigmoid, or tanh.
* **Training Process:**
  + During training, as we propagate forward through the network, batch normalization normalizes the activations of each hidden layer using the statistics computed from the current mini-batch.
  + This normalization step helps stabilize the training process and prevents issues such as vanishing or exploding gradients.
* **Advantages:**
  + Faster Convergence: Batch normalization accelerates the training process by reducing internal covariate shift and enabling higher learning rates.
  + Improved Stability: It stabilizes the training process by preventing the gradients from vanishing or exploding, leading to more stable and reliable learning.
  + Better Performance: Overall, batch normalization often leads to better performance in terms of training speed and final accuracy, especially for deeper neural networks.

### Internal covariate shift

Internal covariate shift refers to the phenomenon where the distribution of network activations changes as the parameters of the preceding layers are updated during training. In simpler terms, it's the change in the distribution of the input to each layer as the parameters of the preceding layers change.

Here's why internal covariate shift can occur and its implications:

1. **Changing Activations:** As the parameters (weights and biases) of the preceding layers are updated during training, the activations of each layer will change. This can lead to a shift in the distribution of the input to each layer, known as internal covariate shift.
2. **Impact on Training:** Internal covariate shift can make training neural networks more challenging. When the distribution of activations changes significantly during training, it can slow down the learning process and make it harder for the network to converge to an optimal solution.
3. **Vanishing or Exploding Gradients:** Internal covariate shift can also contribute to issues like vanishing or exploding gradients. If the distribution of activations changes too drastically, it can lead to gradients becoming very small or very large, which can hinder the training process.
4. **Stabilizing Training with Batch Normalization:** Batch normalization is a technique used to mitigate the effects of internal covariate shift by normalizing the activations of each layer. By normalizing the activations within each mini-batch, batch normalization helps stabilize the training process and prevent issues like vanishing or exploding gradients.

## Optimizers in Deep Learning

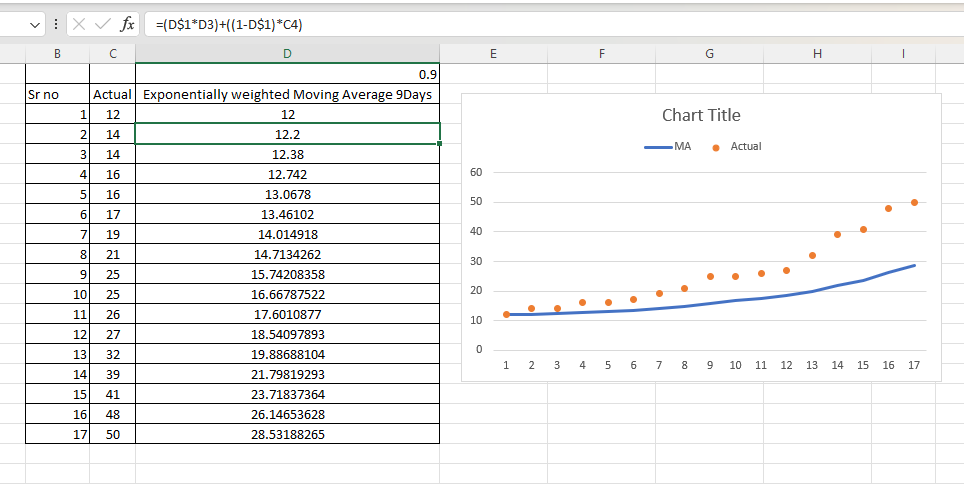
To increase the speed of Deep learning is weight initialization, batch normalization and activation function. Apart from this we have optimizers. In deep learning we draw an architecture where we have weights and biases of layers which will predict the output closer to the real value. Neural network is an optimization problem itself where we reduce the loss. To complete this process weights and bias are initially randomly assigned. The algorithm aims to find the global minima where we have optimizer techniques to find the lowest cost function. In Gradient descent we have three types Batch, Mini-batch and stochastic gradient descent

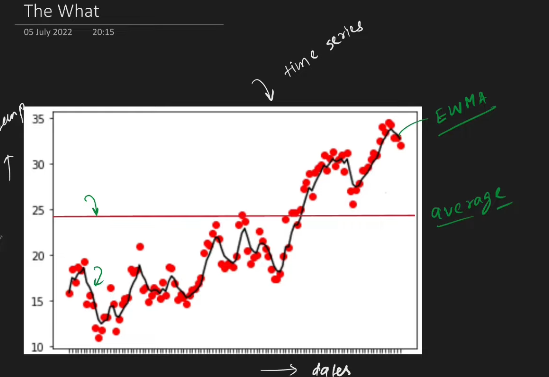
**Challenges with Gradient Descent:**

1. **Learning Rate Selection:** Choosing an appropriate learning rate is critical for the convergence of gradient descent. A learning rate that is too high may cause oscillations or divergence, while a learning rate that is too low may result in slow convergence.
2. **Learning Rate Scheduling:** Deciding on the schedule for adjusting the learning rate during training can be challenging, as it requires prior knowledge or experimentation and need to be set before the training of the model.
3. **Optimal Parameter Values:** Gradient descent requires traversing the parameter space to find optimal values for weights and biases, which can be computationally expensive, especially in high-dimensional spaces.
4. **Local Minima:** Gradient descent may converge to local minima instead of the global minimum of the loss function, depending on the initialization and landscape of the loss function.
5. **Saddle Points:** Saddle points in the loss landscape can slow down convergence, as the gradient near saddle points may be close to zero, leading to slow progress.

### Exponentially Weighted Moving Average

EWMA is technique to find the hidden trend in time series data. The weighted average of the next point would be more than the previous point.





### SGD with Momentum:

SGD with momentum is an optimization algorithm commonly used in training neural networks. It buils upon the basic Stochastic Gradient Descent algorithm by introducing momentum, which helps accelerate convergence, especially in the presence of high curvature, small but consistent gradients, or noisy gradients.

#### Concept of Momentum

In SGD with momentum, the update rule for the parameter at iteration is modified to incorporate a momentum term. The update rule is as follows:

 is the learning rate.

 is the momentum coefficient, typically set between 0 and 1. It is also called as decay factor

 is the momentum term at iteration .

 is the gradient of the objective function with respect to the parameter at iteration .

#### Mathematics behind Momentum:

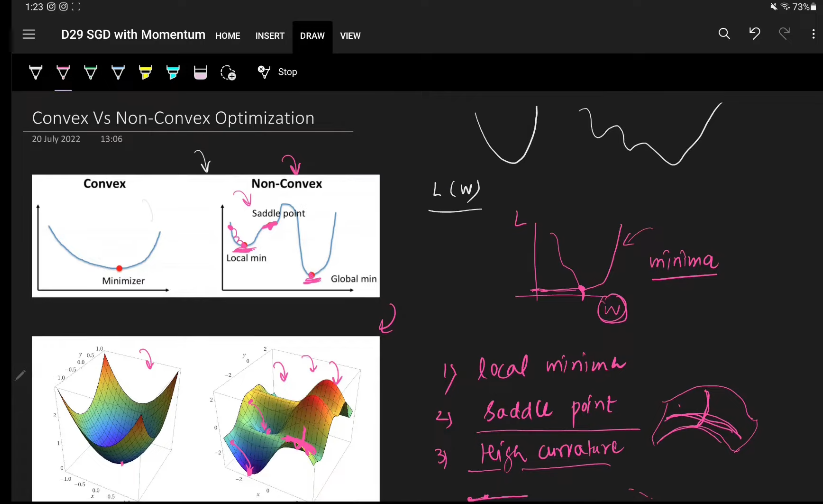
* The momentum term is a moving average of past gradients, which accumulates the gradient vectors over time. It acts as a velocity term, indicating the direction and speed of the optimization process.
* When the gradients have consistent signs and magnitudes, the momentum term grows in the direction of the gradients, allowing the optimizer to move faster towards the optimum.
* Conversely, when the gradients fluctuate or change direction frequently, the momentum term smooths out these variations, providing more stable updates and preventing oscillations.
* The momentum coefficient controls the influence of past gradients on the current update. A higher value leads to more inertia and smoother trajectories, while a lower ββ value allows faster adaptation to changing gradients.

#### Convex and Non-Convex Optimization

* In convex optimization problems, the objective function has a single global minimum, and SGD with momentum helps accelerate convergence towards this minimum by reducing oscillations and efficiently navigating the search space.
* In non-convex optimization problems, the objective function may have multiple local minima, saddle points, or plateaus. SGD with momentum help escape shallow local minima and navigate complex landscapes by providing momentum to overcome barriers and explore promising regions of the search

#### Disadvantages

1. **Overshooting**: The momentum term accumulates gradients over time, which can lead to overshooting the minimum or bouncing around it. This can cause the optimizer to oscillate and take longer to converge, especially if the momentum coefficient is too high.
2. **Smoothing Effect**: In some cases, the momentum term can smooth out gradients too aggressively, causing the optimizer to miss sharp changes in the objective function landscape. This can slow down convergence, particularly in regions with high curvature or steep gradients.
3. **Sensitivity to Hyperparameters**: The performance of SGD with Momentum can be sensitive to the choice of hyperparameters, such as the momentum coefficient (ββ) and the learning rate (αα). Selecting inappropriate values for these hyperparameters may result in suboptimal convergence or instability during training.
4. **Memory Requirements**: The momentum term requires additional memory to store the accumulated gradients for each parameter. In deep neural networks with millions of parameters, this can increase memory consumption and training time, especially when using high-dimensional inputs or large batch sizes.
5. **Inertia**: The momentum term introduces inertia into the optimization process, which can cause the optimizer to struggle to adapt quickly to sudden changes in the gradient landscape. This can be problematic in dynamic environments or when the objective function changes rapidly over time.
6. **Potential for Divergence**: If the momentum coefficient is too high, SGD with Momentum may overshoot the minimum and diverge, leading to instability and poor convergence. Finding an appropriate balance between momentum and learning rate is crucial to avoid this issue.



### Nesterov Accelerated Gradient (NAG)

