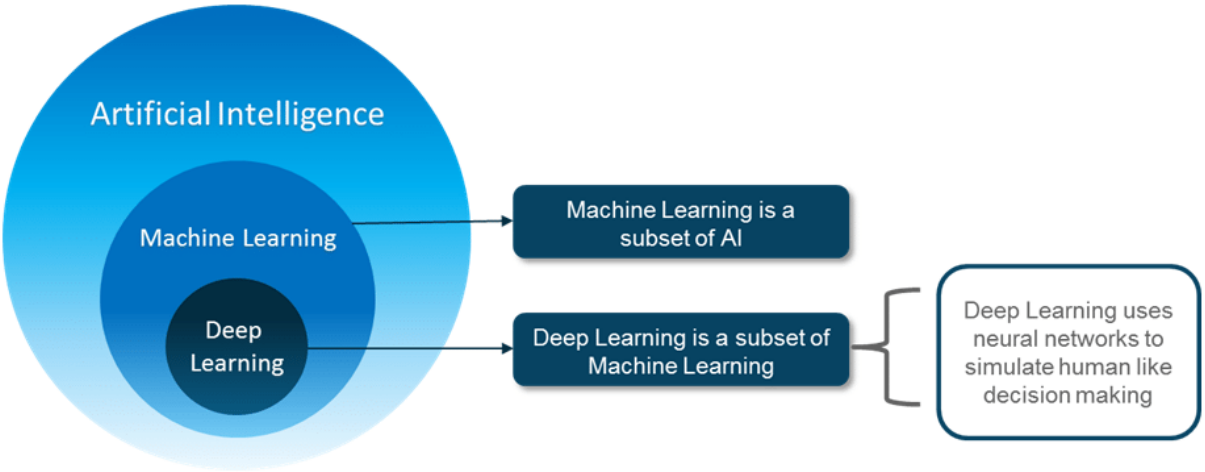
Course Title : Deep Learning Architectures

UNIT-1

MACHINE LEARNING WITH SHALLOW NEURAL NETWORKS

Deep Learning

Deep learning, a subset of machine learning, utilizes algorithms inspired by the human brain's structure and function. Through artificial neural networks, it constructs intelligent models capable of tackling intricate problems. Deep learning is primarily employed when working with unstructured data.



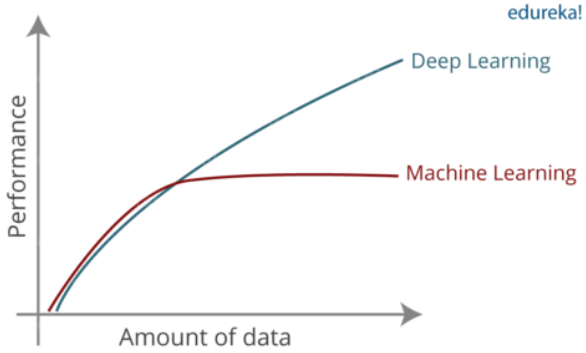
Artificial Intelligence is a technology that allows machines to mimic human behavior. The concept of AI is both simple and intriguing - to create machines capable of making decisions independently. For a long time, it was believed that computers would never be able to compete with the complex capabilities of the human brain.

With limited data and computing power in the past, the development of Artificial Intelligence was constrained. However, the emergence of Big Data and GPU technology has made AI a reality. Machine Learning is a branch of AI that leverages statistical approaches to enhance machine performance through learning from experience.

Deep learning is a branch of machine learning that enables the training of multi-layer neural networks. Its utilization of neural networks allows for the emulation of human decision-making processes.

The need for Deep Learning

- Machine Learning Algorithms are not equipped to handle High-Dimensional Data, characterized by a vast number of inputs and outputs spanning thousands of dimensions. Processing such data is laborious and resource-intensive, a phenomenon known as the **Curse of Dimensionality**.
- Machine Learning Algorithms are not equipped to handle High-Dimensional Data, characterized by a vast number of inputs and outputs spanning thousands of dimensions. Processing such data is laborious and resource-intensive, a phenomenon known as the Curse of Dimensionality.
- Machine Learning Algorithms are not equipped to handle High-Dimensional Data, characterized by a vast number of inputs and outputs spanning thousands of dimensions. Processing such data is laborious and resource-intensive, a phenomenon known as the Curse of Dimensionality.



This is where Deep Learning proved to be invaluable. Deep learning excels at managing high-dimensional data and autonomously pinpointing the relevant features efficiently.

What is Deep Learning?

Deep Learning is a branch of Machine Learning that employs comparable Algorithms to train Deep Neural Networks, resulting in enhanced accuracy. Essentially, Deep Learning mirrors the cognitive functions of the human brain by learning through experience.

As you're aware, the human brain consists of billions of neurons that enable us to accomplish remarkable tasks. Even a young child's brain can solve intricate problems that would challenge even the most powerful supercomputers. To replicate this level of functionality in a program, we turn to the concept of Artificial Neurons (Perceptrons) and Artificial Neural Networks.

What is a Neural Network?

The term "neural" is derived from "neuron", which refers to a single nerve cell. Essentially, a neural network is a network of neurons that carry out basic actions in our daily routines."

Example:

- Children learning to identify the appearance of an apple
- An animal distinguishing between its mother or owner
- Sensing the temperature of an object to determine if it is hot or cold

Our neural networks perform these complicated computations.

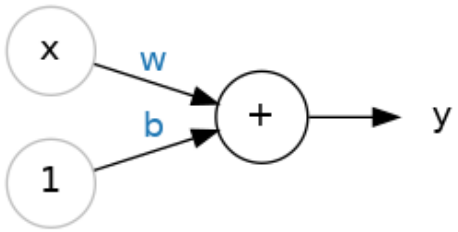
Humans have now been able to build a computation system that can perform in a manner similar to our nervous system. These are called artificial neural networks (ANNs).

An artificial neuron or Perceptron Model:

Perceptron is a supervised learning algorithm that classifies the data into two categories, thus it is a binary classifier.

The Linear Unit:

"Let us start by exploring the essential building block of a neural network: the individual neuron. A simple representation of a neuron (or unit) with a single input is shown below:"



The Linear Unit: $y = wx + b$

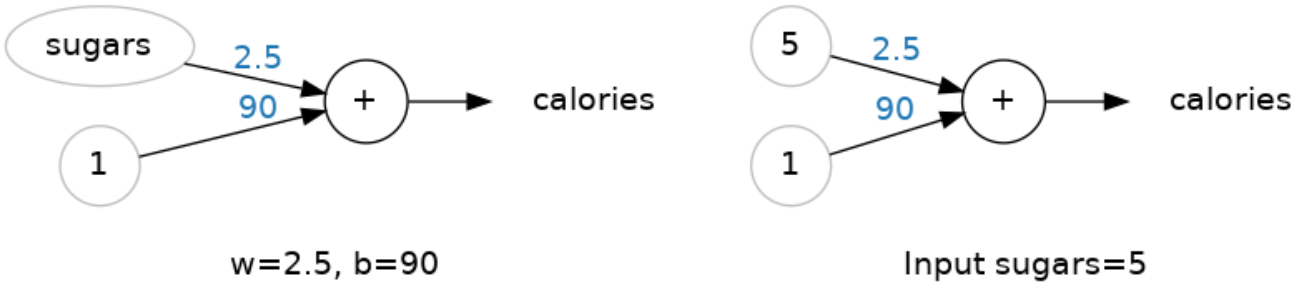
- The neuron receives input, denoted as x, with a weight connection of w. When a value travels through this connection, it is multiplied by the weight, resulting in w * x reaching the neuron.
- A neural network adjusts its performance by altering the weights.
- In addition to regular weights, we introduce a specialized weight called the bias denoted as b. Unlike other weights, the bias does not have an associated input; instead, a value of 1 is placed in the diagram so that the neuron receives only b (1 * b = b).
- The inclusion of the bias allows the neuron to adjust its output independently from its inputs.

- The output of a neuron, denoted as y, is determined by summing up all incoming values through its connections. The activation of the neuron can be represented by the formula y = wx + b, where w is the weight and x is the input value.
- Are you familiar with the formula y = wx + b? This equation represents a line, specifically the slope-intercept equation where "w" stands for the slope and "b" represents the y-intercept.

Example - The Linear Unit as a Model:

When considering the functionality of neurons within a network, it is beneficial to begin with a single-neuron model as a foundational reference point. These models are typically linear in nature, representing an individual neuron's behavior within a broader network context.

Consider the application of this concept on a dataset such as 80 Cereals. By training a model with 'sugars' (grams of sugars per serving) as the input and 'calories' (calories per serving) as the output, we may observe a bias of b=90 and a weight of w=2.5. An estimate for the calorie content of a cereal with 5 grams of sugar per serving can be calculated as follows:



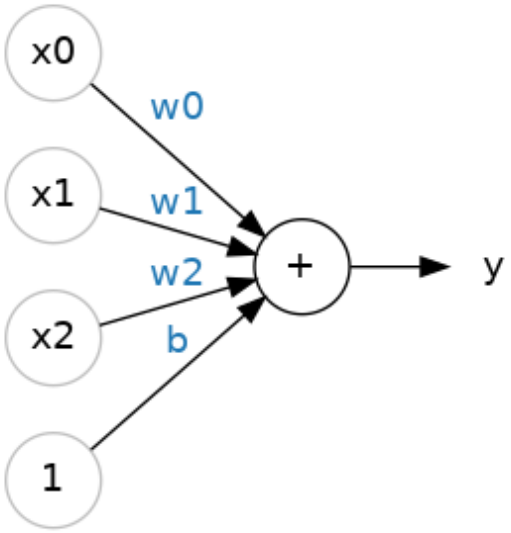
Computing with the linear unit.

Calculating using the linear unit, we find that our formula yields calories=2.5×5+90=102, as the anticipated result.

Multiple Inputs

The 80 Cereals dataset contains a wide range of attributes beyond just 'sugars'. Consider the possibility of enhancing our model by incorporating factors such as fiber or protein content. This can be achieved simply by increasing the number of input connections to the neuron and assigning one connection for each additional feature. Calculating the output entails multiplying

each input by its corresponding connection weight and then summing the results together.



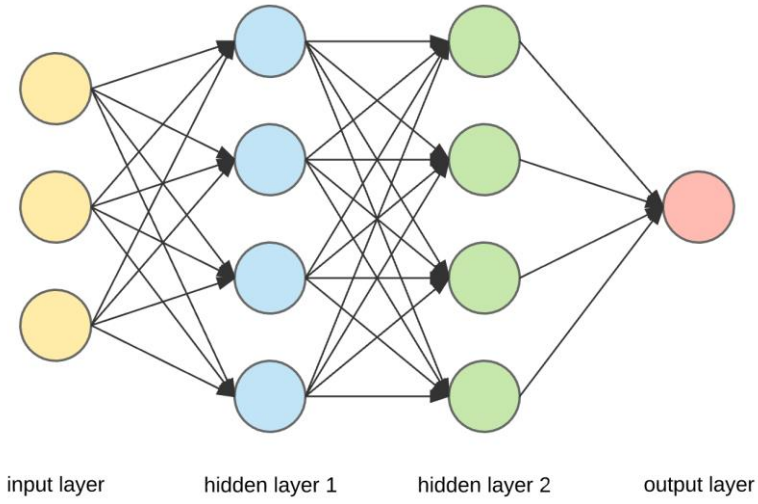
A linear unit with three inputs.

The equation representing the neuron can be written as y = w0x0 + w1x1 + w2x2 + b. A linear unit with two inputs can be used to model a plane, while a unit with additional inputs can fit a hyperplane.

What is an Artificial Neural Network Model?

An artificial neural network, or ANN, is a type of neural network that consists of multiple layers, including an input layer, hidden layers, and an output layer.

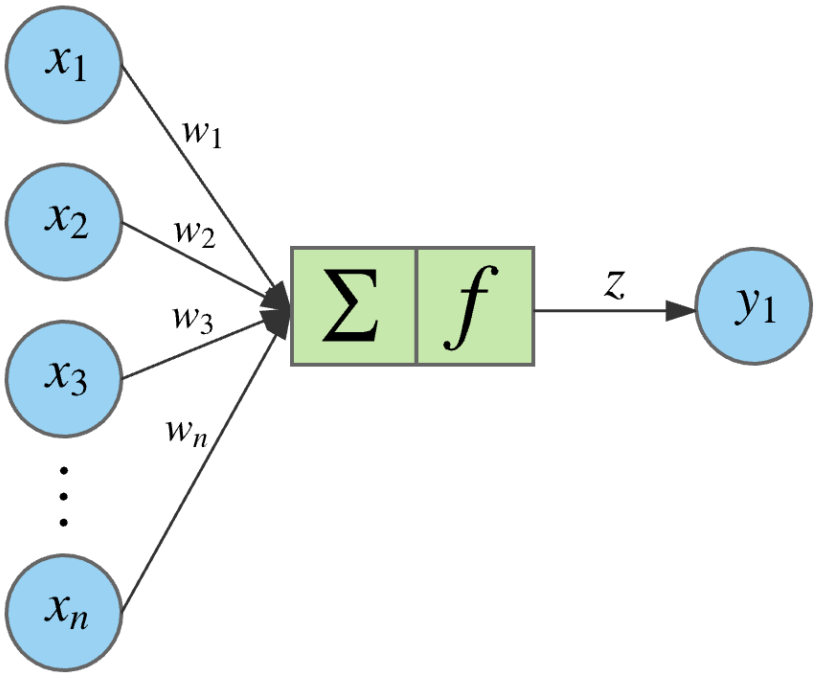
The image below represents an ANN.



Observing closely reveals that every node in a layer is intricately connected to each node in the adjacent layer.

As the number of hidden layers is augmented, the network gains depth.

Let's explore the characteristics of a single node within the output or hidden layer.



The node receives multiple inputs, calculates the sum of weights, and outputs the result through a non-linear activation function. This output becomes the input for the next layer's node. It's crucial to understand that the signal flows from left to right throughout the network. After all nodes have processed the data, the final output is produced. Below is the representation of a node's equation:

$$z = f(b + x \cdot w) = f\left(b + \sum_{i=1}^{n} x_i w_i\right)$$

$$x \in d_{1\times n},\ w \in d_{n\times 1},\ b \in d_{1\times 1},\ z \in d_{1\times 1}$$

The equation above states that b represents the bias, serving as the input for all nodes and consistently holding a value of 1. The bias is crucial in adjusting the activation function output towards the left or right direction.

Glossary of Artificial Neural Network Model

In this discussion, we will explore the fundamental terms essential for understanding an artificial neural network model.

Inputs

When data is initially inputted into a neural network from the source, it is referred to as the "input". The purpose of this input is to provide the network with data in order to facilitate the decision-making or prediction process. Typically, neural network models are designed to accept sets of real values as inputs, which must then be transmitted to a neuron within the input layer.

Training set

Training sets refer to inputs whose correct outputs are already known. They are utilized to assist in the training and memorization of results for a specific input set by the neural network.

Outputs

Each neural network produces an output in the form of real values or Boolean decisions based on the data it receives. The output is generated by one of the neurons in the output layer.

Neuron

The fundamental component of a neural network, known as a perceptron, is a neuron. Neurons take in input values and produce an output accordingly. Each neuron accepts a portion of the input, processes it through a non-linear activation function, and transmits the result to nodes in the subsequent layer. Common activation functions include TanH, sigmoid, and ReLu, which introduce non-linear elements crucial for effectively training the network.

Weight space

Each neuron in a neural network possesses a numerical weight that is combined with others to produce an output when sending input to another neuron. The process of training neural networks involves adjusting these weights through small increments. Fine-tuning the weights is crucial for determining the optimal set of weights and biases that yield the desired results. Backpropagation plays a key role in this optimization process.

Advantages of Artificial Neural Network (ANN)

- **Capability of Parallel Processing: Artificial neural networks possess a numerical value that allows them to execute multiple tasks concurrently.**
- **Data Storage on the Entire Network: In traditional programming, data is stored across the entire network rather than in a centralized database. The loss of a few pieces of data in one location does not impede the network's functionality.**
- **Ability to work with limited knowledge: Following training of an Artificial Neural Network (ANN), the system may still generate output with insufficient information. The impact on performance in this situation is determined by the importance of the missing data.**
- **Having a memory distribution:** To ensure the adaptability of an artificial neural network (ANN), it is imperative to establish a memory distribution by providing relevant examples and guiding the network towards the desired output through demonstration. The performance of the network is closely linked to the quality of the instances selected, as any incomplete or inaccurate information may lead to erroneous results.
- **Having fault tolerance:** Fault tolerance is a characteristic of artificial neural networks that allows them to continue generating output even if one or more cells are compromised. This resilience ensures that the network remains operational despite potential failures.

Disadvantages of Artificial Neural Network:

- **Ensuring the Correct Network Structure: Determining the structure of artificial neural networks does not have a set guideline. The appropriate network structure is achieved through practical knowledge, experimentation, and learning from mistakes.**
- **Unidentified Network Behavior: The issue of unrecognized behavior within Artificial Neural Networks (ANN) is a critical concern. When ANN generates a testing result, it often fails to offer explanations for the reasoning behind its decisions, leading to a decrease in confidence in the network.**
- **Dependency on Hardware: Artificial neural networks rely on processors with parallel processing capabilities, which are essential for their architecture.**

Consequently, the functionality of the equipment is contingent on this requirement.

- **Issue of Demonstrating to the Network: Artificial Neural Networks (ANNs) are designed to process numerical data. Therefore, any problems must first be transformed into numerical values before they can be fed into the ANN. The manner in which this presentation mechanism is handled will have a direct impact on the network's performance, as it heavily relies on the user's proficiency in this area.**
- **The duration of the network is unknown:** The length of the network is uncertain as it has been limited to a specific error value that is not providing optimal outcomes.

Neural Architectures for Binary Classification Models:

In this section, we will delve into fundamental architectures for machine learning models such as least-squares regression and classification. It is evident that the neural architectures corresponding to these models are subtle adaptations of the perceptron model in machine learning. The key disparity lies in the selection of the activation function utilized in the final layer, and the loss function employed on these results. This concept will be a recurring topic in this chapter, underscoring how slight modifications in neural architectures can yield distinct models from traditional machine learning approaches. Presenting traditional machine learning models in the framework of neural architectures also aids in understanding the inherent similarities among various machine learning models.

Least Square Regression Method:

Least squares is a widely employed technique in regression analysis to estimate unknown parameters by developing a model that minimizes the total squared errors between the observed and predicted data.

One commonly utilized method for fitting curves involves minimizing the sum of squared errors to achieve the smallest possible discrepancy. This technique allows for the creation of a line of best fit tailored to your specific data points.

Finding the Line of Best Fit Using Least Square Regression

When analyzing a set of paired numbers and its corresponding scatter plot, the line of best fit is the straight line that can be drawn through the scatter points to represent the relationship between them most accurately. The equation of the straight line is represented as:

Y = mX + c

where:

Y: Dependent Variable

m: Slope

X: Independent Variable

c: y-intercept

The objective is to determine the values of slope and y-intercept, and then substitute them into the equation along with the values of the independent variable X to calculate the values of the dependent variable Y. If we have 'n' data points, the slope can be calculated using the following formula.

$$m = \frac{\sum(x - \bar{x})(y - \bar{y})}{\sum(x - \bar{x})^2}$$

Then, the y-intercept is calculated using the formula:

$$c = \bar{y} - m * \bar{x}$$

At last, we substitute these values in the final equation Y = mX + c.

Logistic regression:

Logistic regression is a statistical technique utilized for making predictions when the output variable is categorical. It is employed when determining whether a given data point belongs to class 0 or class 1. The goal in logistic regression is to calculate the probability of the output being y=1 for a given input vector x. The predicted value y' represents the outcome when the input is x. This can be expressed mathematically as:

$$y' = p(y = 1|x)$$

Mathematical Model

Input: X is a matrix with dimensions n x m, where n represents the number of features and m denotes the number of training examples.

Parameters: W represents a Weight Matrix with dimensions n x 1, where n signifies the number of features within the dataset X. The Bias b serves the purpose of regulating the threshold value at which the activation function is activated.

Output:

$$y' = \sigma(W^T X + b)$$

Activation Function

Activation functions play a crucial role in the learning process of an Artificial Neural Network by adding non-linear properties to the network. Their primary function is to transform the input signal of a node into an output signal, which is then passed on as input to the next layer of the Neural Network. In this instance, the activation function applied is the **sigmoid activation function**. Mathematically, it can be defined as:

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

Loss Function

Loss is the discrepancy between the predicted output and the actual output. Minimizing the loss function is crucial to ensure the predicted value closely approximates the actual value. The choice of loss function for training a neural network depends on the specific case at hand. It is essential to select the appropriate loss function for the task to ensure effective training of the neural network. The mathematical definition of the loss function used for logistic regression is as follows:

$$L(y', y) = -[y \log y' + (1 - y) \log(1 - y')]$$

Let's examine the effectiveness of this loss function for logistic regression:

1. When y=1, the loss function is represented as L(y',y) = -log y'. To minimize this loss function, the value of log y' should be maximized. This can be achieved when y' approaches 1, resulting in predicted values that closely align with actual values.

2. When the loss function is calculated at y=0, it results in L(y',y) = -log(1-y'). In order to minimize the loss function, the value of log(1-y') should be maximized. This can be achieved by reducing the value of y', making it closer to 0. By doing so, the predicted value will be more aligned with the actual value.

3. The loss function described above exhibits convexity, ensuring the presence of a sole global minimum and preventing the network from getting trapped in local minima commonly found in non-convex loss functions.

Cost Function

The loss function is applied to individual input training examples throughout the training process, while the cost function is calculated for the entire training dataset in a single iteration. Essentially, the cost function represents the average of all loss values across the entire training dataset. This can be mathematically defined as:

$$J(W,b) = \sum_{i=1}^{m} L(y'^{i}, y^{i})$$

In the equation above, m represents the total number of training examples. The goal of training the network is to determine the Weight matrix **W** and Bias **b** values that minimize the cost function **J**.

Gradient Descent

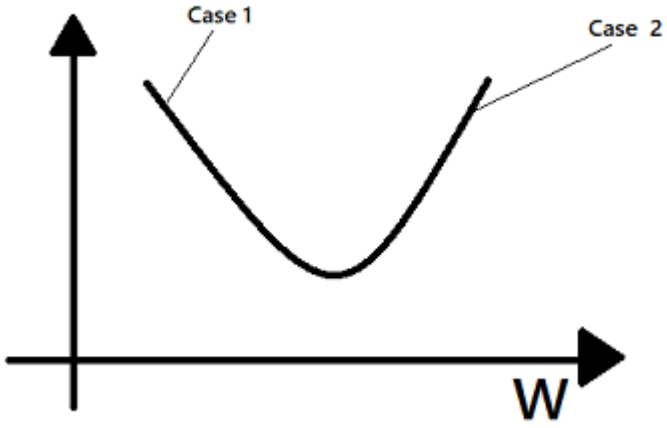
The Weight Matrix W is initially set with random values. We utilize gradient descent to reduce the Cost Function J in order to achieve the most favorable Weight Matrix W and Bias b. Gradient

descent is a primary iterative optimization technique used to discover the lowest point of a function. The application of gradient descent to Cost Function J aids in lowering the overall cost. This process can be expressed mathematically as:

$$\text{Repeat } \{$$
$$W = W - \alpha \frac{\partial}{\partial W} J(W)$$

$$b = b - \alpha \frac{\partial}{\partial b} J(b)$$
$$\}$$

The initial equation signifies the modification in Weight Matrix W, while the subsequent equation represents the adjustment in Bias b. The alterations in the values are dependent on the learning rate alpha and the cost J derivatives in relation to Weight Matrix W and Bias b. The process of updating W and b is repeated until the Cost Function J is minimized. Next, we will explore how Gradient Descent operates by analyzing the accompanying graph.



Case 1. Assuming that W was initialized with values below the threshold required for global minimum, the slope at that point (partial derivative of J with respect to W) will be negative. Consequently, the weight values will increase as per the equation of Gradient Descent.

Case 2. Assuming W has been initialized with values greater than those that correspond to the global minimum, it can be inferred that the slope at that point (i.e. the partial derivative of J with respect to W) is positive. Consequently, the weight values will decrease as per the equation of Gradient Descent.

Hence, both the variables W and b will reach their optimum values, leading to the minimization of the cost function J.

Logistic Regression using Gradient Descent

Introduction

Up to this point, we have comprehended the mathematical model of both logistic regression and gradient descent. In the upcoming section, we will explore the application of gradient descent in learning the Weight Matrix W and Bias b within the realm of logistic regression. Let us consolidate all the equations we have covered thus far.

$$z = W^T X + b$$
$$y' = a = \sigma(z)$$
$$\mathcal{L}(a, y) = -(y \log(a) + (1 - y) \log(1 - a))$$

1. Equation 1 represents the result of multiplying an input X with the Weight Matrix W and Bias b.
2. Equation 2 applies the sigmoid activation function to introduce non-linearity.
3. Equation 3 defines the loss function, which quantifies the difference between a given Y and the predicted Y'.

Widrow−Hoff Rule or Delta Learning Rule:

The Least Mean Square (LMS) method was developed by Bernard Widrow and Marcian Hoff. This algorithm aims to minimize errors across all training patterns in a supervised learning approach. It utilizes a continuous activation function.

Basic Concept − The foundation of this principle lies in the gradient-descent method, which persists indefinitely. The delta rule adjusts the synaptic weights in order to reduce the overall input to the output unit toward the desired target value.

Mathematical Formulation − To adjust the synaptic weights, the delta rule can be expressed as:

Δwi=α.xi.ej

where:

Δwi represents the change in weight for the ith pattern;

α is a positive and constant learning rate;

xi is the input value from the pre-synaptic neuron;

ej is the difference between the desired/target output and the actual output yin (ej = t−yin).

It is important to note that the above delta rule is intended for a single output unit only.

The updating of weight can be done in the following two cases −

Case-I:

If t does not equal y, then the new weight (w(new)) is equal to the old weight (w(old)) plus the change in weight (Δw).

Case-II:

If t equals y, then there is no change in weight.

Closed - form solutions:

The closed-form solution is preferable for smaller datasets if the cost of computing a matrix inverse is not a concern. For larger datasets, or those where the inverse of $\mathbf{X^TX}$ may not exist due to perfect multicollinearity, gradient descent (GD) or stochastic gradient descent (SGD) approaches are recommended. The linear regression model is defined as:

$$y = w_0 x_0 + w_1 x_1 + \ldots + w_m x_m = \sum_{j=0}^{m} \mathbf{w}^\top \mathbf{x}$$

The response variable is denoted as y, the sample vector with m dimensions is represented by x, and the weight vector (a vector of coefficients) is symbolized as w. It is important to acknowledge that w_0 signifies the intercept on the y-axis in the model, thus x_0 is equivalent to 1. By employing the closed-form solution, which is the normal equation method, we determine the weights of the model subsequently:

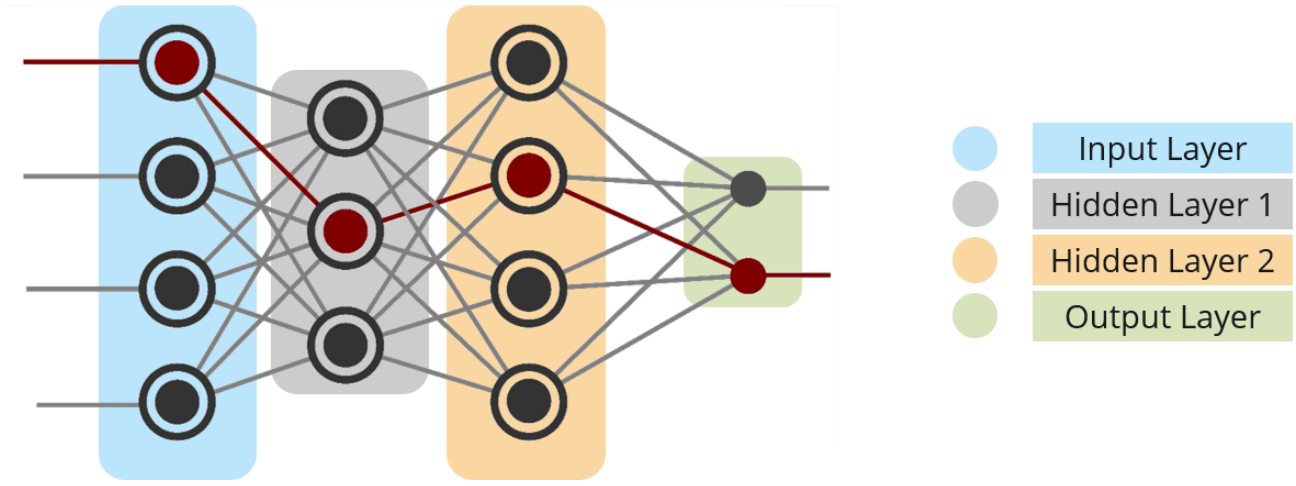
$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T y$$

Neural Architectures for Multiclass Models:

What is Multi-Layer Perceptron?

The human brain consists of millions of neurons. Therefore, a Neural Network is essentially a grouping of Perceptrons linked together in various configurations and utilizing various activation functions.

Consider the diagram below:



- **Input Nodes** – The input nodes in a neural network, collectively known as the "Input Layer," receive information from external sources and transmit it to the hidden nodes. No calculations are carried out by the input nodes; they solely pass on the received data.
- **Hidden Nodes** – The nodes known as "Hidden nodes" lack direct connections to external sources (hence their designation as "hidden"). These nodes carry out computations and relay information between input nodes and output nodes. A group of

hidden nodes comprises a "Hidden Layer." Although a network typically contains only one input layer and one output layer, it has the potential for zero or multiple Hidden Layers. A Multi-Layer Perceptron consists of at least one hidden layer.

- **Output Nodes –** The output nodes, known collectively as the "Output Layer," are tasked with performing calculations and transmitting data from the network to external sources.

Weston-Watkins SVM (or) Multi-class SVM Loss:

Simply put, a loss function is utilized to measure the effectiveness of a predictor in classifying input data points within a dataset. A smaller loss indicates a stronger performance by our classifier in capturing the relationship between input data and output class labels. However, it is possible to overfit the model when the training data is closely mimicked, resulting in a loss of generalization capability.

Conversely, a larger loss signifies a need for improving classification accuracy through parameter adjustments, such as modifying the weight matrix W or bias vector b. This process involves optimizing parameter updates to enhance classification accuracy.The mathematics behind Multi-class SVM loss

The scoring function f assigns numerical scores to class labels based on our feature vectors. As implied by its name, a Linear SVM utilizes a straightforward linear mapping:

$$f(x_i, W, b) = Wx_i + b$$

Now that we have the scoring/mapping function f in place, it is imperative to assess its effectiveness in making predictions. This assessment will be based on the weight matrix W and bias vector b used in the function.

In order to ascertain the performance of the function, a loss function needs to be established. Let us move forward in defining this loss function.

It is worth noting that we already possess a matrix of feature vectors denoted as x. These feature vectors can be obtained from color histograms, Histogram of Oriented Gradients features, or raw pixel intensities. Regardless of the method chosen for quantifying our images, the essence lies in having a feature matrix extracted from the image dataset. The features associated with a particular image can then be accessed through the syntax x[i], thereby providing the i-th feature vector within x.

In the same manner, we possess a vector called y that holds the class labels corresponding to each instance x. These y values serve as our ground-truth labels and are the targets for accurate prediction by our scoring function. Just as we retrieve a specific feature vector using x, we can obtain the i-th class label using y[i]. For the sake of brevity, we will denote our scoring function as s.

$$s = f(x_i, W)$$

This means that we can calculate the anticipated score of the j-th category for the i-th data point using the following formula:

$$s_j = f(x_i, W)_j.$$

By utilizing this syntax, we are able to combine all the elements to derive the hinge loss function.

$$L_i = \sum_{j \neq y_i} max(0, s_j - s_{y_i} + 1)$$

So what is the above equation doing exactly?

In essence, the hinge loss function calculates the sum of errors across all misclassified classes. It directly compares the scoring function output for the incorrect class (j-th class label) with that of the correct class (t-th class).

We apply the *max* operation to clamp values to *0* — this is important to do so that we do not end up summing negative values.

A given x_i is classified correctly when the loss $L_i = 0$ (I'll provide a numerical example of this in the following section).

To derive the loss across our ***entire training set***, we simply take the mean over each individual L_i:

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i$$

Another related, common loss function you may come across is the ***squared hinge loss:***

$$L_i = \sum_{j \neq y_i} max(0, s_j - s_{y_i} + 1)^2$$

The use of the squared term in our loss function results in a heavier penalty by squaring the output, causing a quadratic increase in loss instead of a linear one.
The choice of a loss function should be determined by the characteristics of your dataset. The standard hinge loss function is commonly utilized, but in certain cases, better accuracy may be achieved using the squared variation. Ultimately, this decision is a hyper parameter that should be validated through cross-validation.

Softmax regression:

Softmax regression (or **multinomial logistic regression**)

In this discussion, we will present the softmax function and its application in multinomial logistic regression for situations involving more than two classes. It is important to differentiate this from multiple logistic regression, which pertains to scenarios with multiple predictors.

What is the Softmax Function?

In the sigmoid function, a probability threshold of 0.5 is applied. Observations falling below this threshold are assigned to class A, while those exceeding the threshold are classified as class B. As a result, predictions are constrained to two distinct classes.

When using multinomial logistic regression, the goal is to classify into multiple classes instead of just two. To achieve this, a function is needed that calculates the probability for each class. It is important that the probabilities for all classes add up to one. The softmax function is well-suited for meeting these criteria.
Softmax Function Formula

$$softmax(z) = \frac{e^{z(i)}}{\sum_{j=0}^{k} e^{z(j)}}$$

Consider a scenario in which z is a vector of inputs with a length equal to the number of classes k. To gain a better understanding of the softmax function, let's explore an example by inputting a vector of numbers.

For instance, let z=[1,3,4,7].

If we aim to compute the probability for the second entry, which is 3, we can do so by substituting our desired values into the formula.

$$softmax(z_2) = \frac{e^3}{e^1 + e^3 + e^4 + e^7} = 0.017$$

By using the softmax function on all elements in vector z, we obtain the resulting vector that adds up to 1:

softmax(z) = [0.002, 0.017, 0.047, 0.934]

Observe that the final element has a probability exceeding 90%. In a classification scenario, the observation would be allocated to the final class.Multinomial Logistic Regression

To conduct multinomial logistic regression, you develop a regression model in the format of:

$$z = \beta^t x$$

and applying the softmax function to it:

$$\hat{y} = softmax(\beta^t x)$$

Multinomial Logistic Regression Loss Function

The loss function for a multiple logistic regression model follows a general form, where y represents the vector of actual outputs. In the context of a classification problem, y is referred

to as a one-hot vector. This signifies that all elements in the vector are 0, except for the position corresponding to the class to which the observation belongs, which is marked as 1.

$$Cost(\beta) = -\sum_{i=j}^{k} y_j log(\hat{y}_j)$$

Let's demonstrate this with an example:

Consider a scenario where you need to categorize fruits into three categories, and the fruit in question is a banana. To simplify the discussion, we will focus on a single observation. The vector y can be represented as follows:

$$y = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} apple \\ banana \\ orange \end{bmatrix}$$

Once the logistic regression model is trained on your dataset, it generates a prediction vector containing probabilities.

$$\hat{y} = \begin{bmatrix} 0.2 \\ 0.7 \\ 0.1 \end{bmatrix}$$

Now, we plug this into our cost function:

$$Cost(\beta) = -(0 \times log(0.2) + 1 \times log(0.7) + 0 \times log(0.1))$$

One advantageous aspect of this function is that when the entries in y are 0, all terms unrelated to the true class will disappear.

$$Cost(\beta) = -log(0.7) = 0.36$$

In effectively minimizing costs, this function operates based on the probability of y_hat. A higher y_hat probability linked to the true probability results in lower costs. The gradient is

determined by obtaining the first derivative of the cost in relation to each entry β_j in β, which is a straightforward process. It turns out to be

$$\frac{\partial Cost(\beta)}{\partial \beta_j} = \hat{y} - y$$

To conclude, we are now able to utilize gradient descent to progressively reduce the cost by scaling it with a learning rate α.

$$Cost(\beta) = Cost(\beta) - \alpha \frac{\partial Cost(\beta)}{\partial \beta_j}$$

Hierarchical Softmax for Many Classes:

What is Hierarchical Softmax?

Before delving into the details of hierarchical softmax, it is important to grasp its functionality and significance.

In conventional softmax methodologies, the final layer of a neural network is tasked with forecasting the probability distribution across a defined set of classes. This is accomplished by executing the softmax function on the output layer, which entails computing the exponential of each output unit and subsequently normalizing the outcomes.

When applied to small datasets with a limited number of classes, this method performs effectively. However, its efficiency diminishes significantly when handling large datasets with numerous classes due to the computational cost associated with the softmax function. Calculating the exponential of all output units becomes increasingly burdensome as the number of classes grows.

Hierarchical softmax offers a solution to this challenge by reconfiguring the output layer into a hierarchical structure resembling a tree. Each node in the tree represents a class, and the path from the root to a node signifies a series of binary decisions affecting the likelihood of that

class. To determine the probability of a class, one must navigate the tree from the root to the relevant node, evaluating the probability of each decision made along the way.

By decomposing the output layer into a hierarchical structure, hierarchical softmax effectively lowers the computational load necessary for calculating the probability distribution across a wide range of classes. This approach is especially beneficial for handling extensive datasets and a high volume of classes. The rationale for breaking down the output layer into a binary tree was to streamline the process of deriving probability distributions, reducing the complexity from O(V) to O(log(V)).

Backpropagated Saliency for Feature Selection:

One common critique of neural networks has been their perceived lack of interpretability. Yet, it has been discovered that employing backpropagation can assist in identifying the key features influencing the classification of a specific test case. This method offers analysts insight into the importance of each feature in the classification process.

This method also possesses the advantageous ability to be utilized for feature selection.

Take into account a test instance X = (x1, . . . xd), where the neural network produces multilabel output scores o1 . . . ok.

In addition, consider the output of the class that wins among the k outputs as om, with m being one of the values in the set {1 . . . k}. The objective is to determine the features that are most significant for classifying this test instance.

In general, we aim to identify the impact of each attribute xi on the output om. Attributes with high sensitivity values play a crucial role in classifying the test instance. In order to achieve this goal, we would like to compute the absolute magnitude of

$$\frac{\partial o_m}{\partial x_i}.$$

The features with the highest absolute value of the partial derivative exert the strongest impact on determining the winning class during classification. Additionally, the sign of this derivative indicates whether a slight increase in xi from its current value will raise or lower the score of the winning class. While the derivative offers some insight into sensitivity for non-winning classes, its importance diminishes, especially in scenarios with a large number of classes.

The value of,

$$\frac{\partial o_m}{\partial x_i}$$

can be computed by a straightforward application of the backpropagation algorithm, in which one does not stop backpropagating at the first hidden layer but applies the process all the way to the input layer.

This technique can also be applied for feature selection by combining the absolute value of the gradient across all classes and properly classified training samples. The features exhibiting the highest combined sensitivity throughout the entire training set are deemed most significant. While it is not strictly necessary to combine this value across all classes, one can opt to focus only on the dominant class for properly classified training samples.

Autoencoders:

What are Autoencoders?

An autoencoder is a type of neural network in which the output layer has the same dimensionality as the input layer. In other words, the number of output units in the output layer equals the number of input units in the input layer. Autoencoders replicate data from the input to the output in an unsupervised manner, leading to their occasional classification as a replicator neural network.

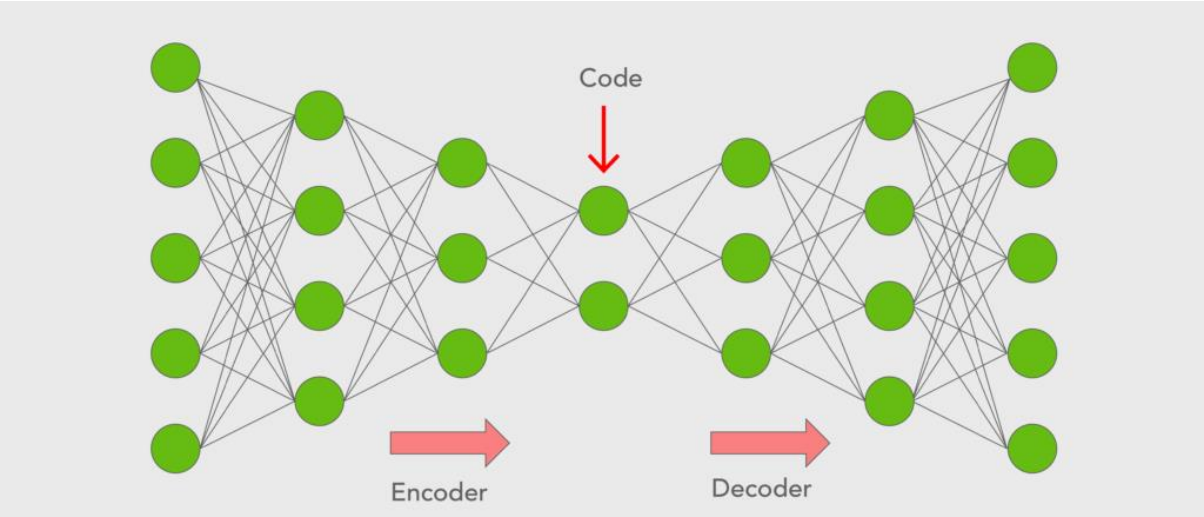
The autoencoders reconstruct individual input dimensions by sending them through the neural network. While using a neural network for input replication may appear straightforward, the process actually involves reducing the input size into a smaller representation. The middle layers of the neural network contain fewer units than the input or output layers, holding this reduced input representation. The output is then reconstructed based on this reduced representation of the input.

Architecture of autoencoders:

An autoencoder consists of three components:

- **Encoder: The encoder is a fully connected neural network that utilizes feedforward architecture to compress the input data into a latent space representation. This network encodes the input image into a compressed form with reduced dimensions, resulting in a distorted version of the original image.**
- **Code: This component of the network stores the reduced representation of the input data, which is then passed on to the decoder for reconstruction.**

- **Decoder: Similar to the encoder, the decoder is a feedforward neural network with a structure that mirrors that of the encoder. It plays a crucial role in reconstructing the input data back to its original dimensions from the encoded code.**



First, the input is processed through the encoder to compress and store it in the Code layer. Subsequently, the decoder decompresses the original input from the Code layer. The primary purpose of the autoencoder is to generate an output that is an exact replica of the input.

Note: In most cases, the decoder architecture is a mirror image of the encoder, although this is not a strict requirement. It is essential, however, that the dimensionality of the input and output remains the same.

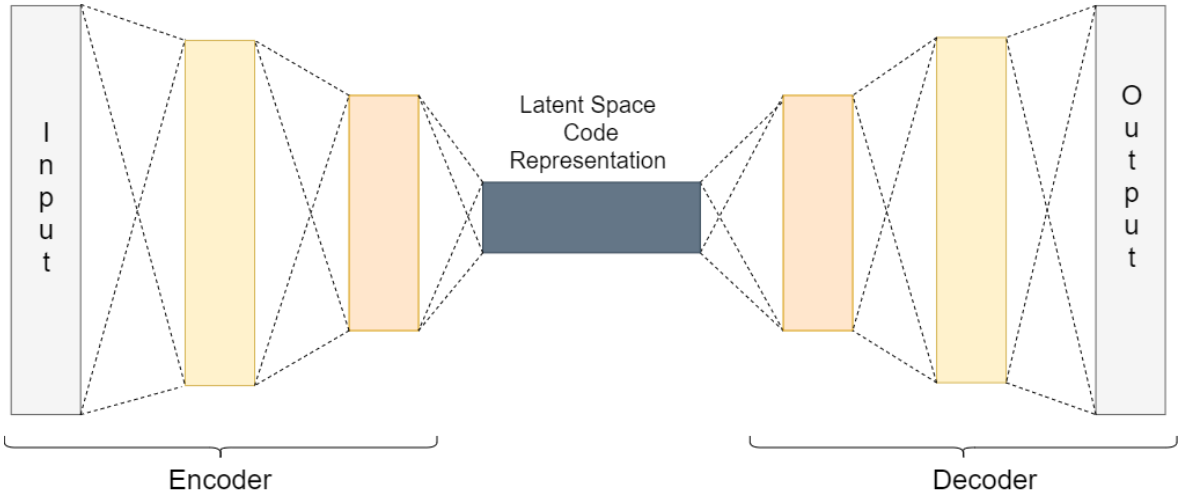
The Principle Behind Autoencoder:

An autoencoder comprises two components: an encoder and a decoder.

Initially, the encoder processes the input data, denoted as x, and encodes it using a function f(x).

Within the autoencoder architecture, there exists an internal hidden layer referred to as h. This hidden layer is responsible for learning the encoded representation of the input created by the encoder, leading to h = f(x).

Subsequently, the decoder function works to reconstruct the input data based on the coding stored in the hidden layer. If we designate the decoder function as g, the reconstruction can be expressed as r=g(f(x)) or simply r=g(h).



Simple Neural Network Architecture of Autoencoder

During the process of reconstructing an image, it is important to ensure that the neural network does not merely duplicate the input in the output. This kind of memorization can result in overfitting and reduced generalization capabilities. An autoencoder must efficiently reconstruct the input data by understanding its valuable features, rather than simply memorizing it.

Deep Autoencoders:

The true potential of autoencoders within the neural network field is fully utilized when deeper architectures are incorporated. By adding more intermediate layers, the neural network's ability to represent data is greatly enhanced. Notably, utilizing nonlinear activation functions within some of these deep autoencoder layers is crucial for maximizing their representation power. In contrast, the use of solely linear activations does not provide any additional advantage in a multilayer network. This principle holds true for all types of multilayer neural networks, including autoencoders.

Deep networks with multiple layers offer a substantial amount of representation power. The hierarchical layers within these networks generate progressively simplified representations of the data, which is especially natural for certain data domains such as images. Traditional machine learning lacks a precise equivalent to this type of model, but the backpropagation technique helps us address the challenges associated with computing the complex gradient-descent steps. Nonlinear dimensionality reduction techniques can map data from a manifold of

any shape into a simplified representation. While there are various methods for nonlinear dimensionality reduction in machine learning, neural networks possess certain advantages over these approaches:

1. Nonlinear dimensionality reduction methods often struggle to map out-of-sample data points to reduced representations without including these points in the training data initially. However, computing the reduced representation of an out-of-sample point is a straightforward process by passing it through the network.
2. Neural networks offer increased power and flexibility in nonlinear data reduction through the manipulation of the number and types of layers used in intermediate stages. Additionally, by selecting specific activation functions in particular layers, one can customize the reduction process to align with the data's properties. For instance, utilizing a logistic output layer with logarithmic loss is a logical choice for a binary data set.

This approach has the capability to produce remarkably compact reductions. For instance, it can compress a 784-dimensional image pixel representation into a 6-dimensional reduction using deep autoencoders.

Using nonlinear units in autoencoders results in greater reduction by implicitly transforming warped manifolds into linear hyperplanes. The enhanced reduction is attributed to the ease of navigating a warped surface through numerous points compared to a linear surface. This characteristic of nonlinear autoencoders is frequently leveraged for generating 2-dimensional visualizations of data through the construction of deep autoencoders with a compact hidden layer of only two dimensions. These two dimensions can then be projected onto a plane to visually represent the data points. Oftentimes, the class structure of the data becomes apparent through distinct clusters in the visualization.

Application to Outlier Detection:

Dimensionality reduction and outlier detection are interrelated, as encoding and decoding outlier points without loss of information is challenging.

It is commonly understood that by factorizing a matrix D as D ≈ D' = UV^T, the resulting low-rank matrix D' serves as a denoised representation of the data.

The compressed representation U only captures the regular patterns in the data, unable to account for exceptional variations in individual points.

Consequently, reconstructing D' fails to include these exceptional variations.

The outlier scores in a matrix can be calculated using the absolute values of the differences between the matrix and its corresponding reconstruction. This approach can be used to detect outlier entries or the outlier score of each row by summing the squared scores of the entries. Identifying outlier data points is crucial in various applications, such as feature selection in clustering. By summing the squared scores in each column of the matrix, outlier features can also be identified. This information can help in removing noisy features from clustering algorithms. While the example above uses matrix factorization, any type of autoencoder can be utilized for this purpose. De-noising autoencoders, in particular, are actively researched in the field.

UNIT–II: TRAINING DEEP NEURAL NETWORKS

Training deep neural networks (DNNs) is a complex process that involves several important aspects to ensure that the network learns effectively and generalizes well to new data. Here are the key components and considerations:

1. Data Quality and Preprocessing

- **Data Collection:** The quality, quantity, and diversity of the training data significantly impact the performance of the model.
- **Data Preprocessing:** Normalization, scaling, augmentation, and handling missing data are crucial steps to prepare the data for training.
- **Data Splitting:** Properly splitting data into training, validation, and test sets helps in evaluating the model's performance and avoiding overfitting.

2. Model Architecture

- **Layer Design:** Choosing the right type and number of layers (e.g., convolutional layers for image data, recurrent layers for sequence data).
- **Activation Functions:** Functions like ReLU, Sigmoid, or Tanh are used to introduce non-linearity, enabling the network to learn complex patterns.
- **Regularization Techniques:** Dropout, L2 regularization, and batch normalization help prevent overfitting and improve generalization.

3. Loss Function

- **Selection:** The choice of loss function depends on the problem (e.g., cross-entropy for classification, mean squared error for regression).
- **Optimization:** The goal is to minimize the loss function to improve the model’s predictions.

4. Optimization Algorithms

- **Gradient Descent:** The basic algorithm for minimizing the loss function by updating weights in the opposite direction of the gradient.
- **Variants of Gradient Descent:**
    - **SGD (Stochastic Gradient Descent):** Uses a single sample or a small batch for each iteration.
    - **Momentum:** Accelerates gradient vectors in the right directions, leading to faster converging.
    - **Adam (Adaptive Moment Estimation):** Combines the advantages of both AdaGrad and RMSProp, adapting learning rates based on estimates of lower-order moments.

5. Learning Rate

- **Importance:** A critical hyperparameter that controls the step size of weight updates.
- **Learning Rate Schedulers:** Techniques like learning rate annealing, and adaptive learning rates adjust the learning rate during training.

6. Batch Size

- **Trade-off:** Smaller batches provide a regularizing effect and lead to better generalization, while larger batches can speed up training.
- **Mini-batch Training:** A compromise between full-batch and stochastic training, often used in practice.

7. Overfitting and Underfitting

- **Overfitting:** Occurs when the model performs well on the training data but poorly on unseen data.
- **Underfitting:** Happens when the model is too simple to capture the underlying structure of the data.
- **Techniques to Combat Overfitting:** Data augmentation, early stopping, and regularization techniques.

8. Validation and Hyperparameter Tuning

- **Cross-Validation:** Helps in assessing the model's performance across different subsets of the data.
- **Hyperparameter Tuning:** Techniques like grid search, random search, and Bayesian optimization are used to find the best hyperparameters.

9. Evaluation Metrics

- **Accuracy, Precision, Recall, F1 Score:** Common metrics for classification problems.
- **Mean Absolute Error, Root Mean Squared Error:** Common metrics for regression problems.
- **Confusion Matrix:** Useful for visualizing the performance of classification models.

10. Transfer Learning

- **Pre-trained Models:** Leveraging models pre-trained on large datasets and fine-tuning them for specific tasks.
- **Feature Extraction:** Using features learned by a pre-trained model as input to a new model.

11. Scalability and Efficiency

- **Parallelization and Distributed Training:** Techniques to speed up training, especially for large datasets and complex models.
- **Hardware Acceleration:** Utilizing GPUs and TPUs to enhance the computational efficiency of training deep neural networks.

12. Monitoring and Debugging

- **Visualization Tools:** Tools like TensorBoard help in monitoring training progress, visualizing metrics, and debugging the model.
- **Logging:** Keeping track of experiments, hyperparameters, and results to facilitate reproducibility and comparison.

13. Ethical Considerations

- **Bias and Fairness:** Ensuring that the model is unbiased and does not perpetuate existing biases in the data.
- **Explainability:** Understanding and explaining the decisions made by the neural network, especially in critical applications.

14. Deployment and Maintenance

- **Model Deployment:** Considerations for deploying the model in a production environment, including scalability, latency, and resource management.
- **Model Retraining:** Periodic retraining with new data to ensure the model stays accurate and relevant over time.

Understanding and effectively managing these aspects is key to successfully training and deploying deep neural networks.

Backpropagation with Computational Graph:

Why computational graphs?

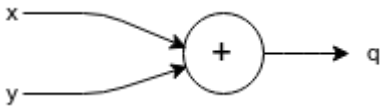
The implementation of backpropagation in deep learning frameworks such as Tensorflow, Torch, Theano, etc., is done through computational graphs. Understanding backpropagation on computational graphs is crucial as it brings together various backpropagation algorithms and their variants (e.g. backprop through time, backprop with shared weights). Once the concepts are converted into a computational graph, all these algorithms can be seen as the same - simply backpropagation on computational graphs.

Computational Graph, What is it?

A computational graph is a directed graph in which nodes represent mathematical operations. It provides a method for presenting and computing mathematical expressions. As an illustration, consider the following basic mathematical equation:

$q = x + y$

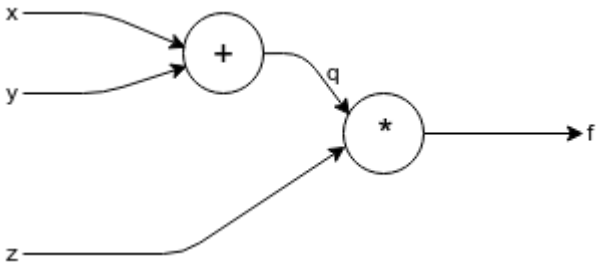
We can draw a computational graph of the above equation as follows.



The computational graph presented above features an addition node, denoted by the "+" sign, with input variables x and y, and output q. Now, consider a slightly more intricate example with the following equation.

$f=(x+y)*z$

Which is represented by the following computational graph.



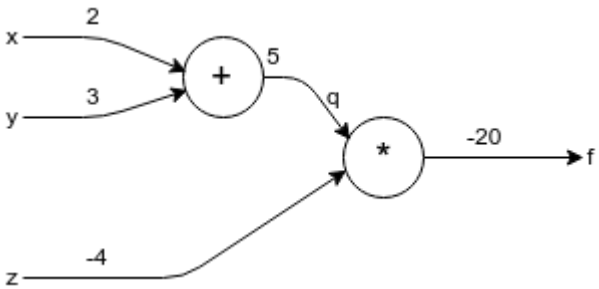
Forward Pass

The forward pass involves determining the value of a mathematical expression depicted in computational graphs. During the forward pass, values are passed from variables towards the node's output in the forward direction.

For instance, consider assigning values to all input variables as follows.

$x=2, y=3, z=-4$

By giving those values to the inputs, we could pass forward and get the following values for the outputs on each node.



First, we use the value of x = 2 and y = 3, to get q = 5. Then we use q = 5 and z = -4 to get f = -20. We go from left to right, *forwards*.

Objectives of Backward Pass (backpropagation)

In the backward pass, our objective is to determine the gradients for each input about the final output. These gradients are essential for training the neural network through gradient descent. As demonstrated in our ongoing example, we seek the specified gradients.

Desired gradients

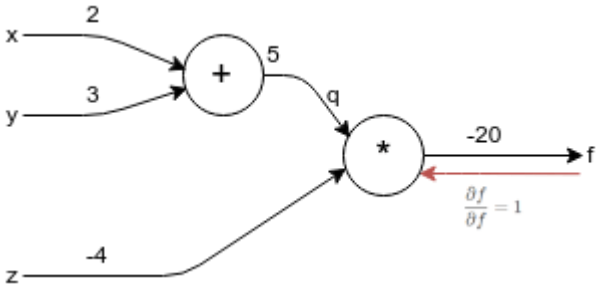
$$\frac{\partial x}{\partial f}, \frac{\partial y}{\partial f}, \frac{\partial z}{\partial f}$$

Backward pass (backpropagation)

We initiate the backward pass by calculating the derivative of the final output with respect to itself. This yields an identity derivative with a value of one.

$$\frac{\partial g}{\partial g} = 1$$

i.e. our computational graph now looks as follows,



Next, we will do the backward pass through the "*" operation, i.e. we'll calculate the gradients at q and z. Since g = p*z, we know that

$$\frac{\partial g}{\partial z} = p \qquad \frac{\partial g}{\partial p} = z$$

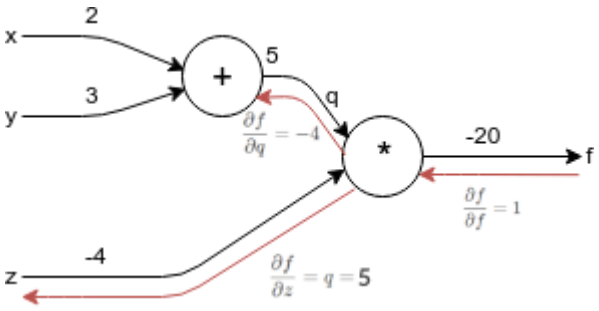
We already know the values of *z* and *q* from the forward pass. Hence, we get

$$\frac{\partial g}{\partial z} = p = 4$$

and

$$\frac{\partial g}{\partial p} = z = -3$$

So the graph now looks as follows:



We want to calculate the gradients at *x* and *y*, i.e.

$$\frac{\partial g}{\partial x}, \frac{\partial g}{\partial y}$$

To ensure optimal efficiency in our calculations, we will use the chain rule of differentiation even though x and f may appear to be distantly located in the graph despite being only two hops away. By applying the chain rule, we can accurately determine these values in a streamlined manner.

$$\frac{\partial g}{\partial x} = \frac{\partial g}{\partial p} * \frac{\partial p}{\partial x}$$

And

$$\frac{\partial g}{\partial y} = \frac{\partial g}{\partial p} * \frac{\partial p}{\partial y}$$

After confirming that df/dq = -4, which can be particularly challenging to calculate for large graphs, it is clear that dq/dx and dq/dy are simpler to determine due to the direct dependence of q on x and y. Thus, we can conclude that:

$$p = x + y \Rightarrow \frac{\partial x}{\partial p} = 1, \frac{\partial y}{\partial p} = 1$$

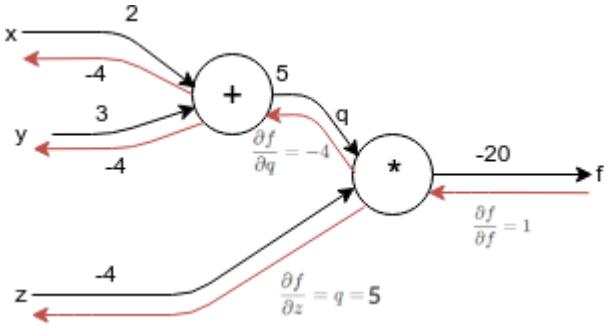
Hence, we get

$$\frac{\partial g}{\partial f} = \frac{\partial g}{\partial p} * \frac{\partial p}{\partial x} = (-3).1 = -3$$

And for the input y.

$$\frac{\partial g}{\partial y} = \frac{\partial g}{\partial p} * \frac{\partial p}{\partial y} = (-3).1 = -3$$

And our final graph.



The primary rationale for working in backward order is that we only utilized previously calculated values to ascertain the gradient at x, as well as dq/dx (the derivative of the output of a node in relation to the input of the same node). Our approach involved leveraging local data to derive a comprehensive outcome.

Steps for training a neural network

To train a neural network, you should adhere to the following steps:

- **Forward Pass:** Take a data point x from the dataset as input and conduct a forward pass through the neural network. Calculate the cost c as the output of this pass.
- **Backward Pass:** Next, initiate a backward pass starting at the cost c. Calculate gradients for all nodes in the graph, including those representing the neural network weights.
- **Update Weights:** Upgrade the weights by utilizing the formula: W = W - learning rate * gradients.

- **Iterative Process:** Repeat these steps iteratively until the predefined stop criteria is fulfilled.

Backpropagation is utilized to determine the impact of changes in a weight w on the error within a neural network. In essence, it computes:

$\partial E/\partial w$,

where E is the error and w is the weight.

In traditional feed-forward neural networks, each neuron connection is assigned a weight. Backpropagation can be easily calculated using the chain rule. For instance, if you have the derivative of the error with respect to node yi (ie. $\partial E/\partial yi$), determining the impact of the pre-synaptic weights connected to that node can be expressed as:

$\partial E\partial w=\partial E/\partial yi\ \partial yi/\partial w$.

In convolutional neural networks, complexity arises from the fact that the weight w is employed across numerous nodes, typically encompassing most, if not all, nodes within the same layer.

Backpropagation with Post-Activation and Pre-Activation variables:

Backpropagation is the method through which neural networks reduce the error in their predictions by modifying the weights and biases of their neurons. This is a simplified explanation of backpropagation, but how exactly are these adjustments made? How is the error in the hidden layers computed? And what role does calculus play in this process? By the end of this article, all of these questions will be addressed. Let's begin by briefly reviewing the entire learning process before delving into the intricacies of backpropagation.

How does learning occur in a neural network?

The process of learning in a neural network takes place in three steps.

1. Input data points are passed through the neural network, flowing through its layers to produce an output in the final output layer. This process, known as forward propagation, will be further explained in detail below.

2. Following the generation of the output, the loss in the output must be calculated. Various methods, such as mean squared error or binary cross-entropy, can be used to determine the loss.
3. Once the loss has been calculated, adjustments must be made to the neural network's parameters (weight and bias) to minimize the loss. This iterative process is referred to as backpropagation.

Forward Propagation in an Artificial Neural Network:

An Artificial Neural Network (ANN) is composed of three main types of layers: the input layer, hidden layers, and the output layer. The process of data flow through the ANN can be summarized as follows:

- **Input layer:** The input or features that the neural network is being trained on are initially fed into the neurons of the input layer during the first pass, where data flows forward through the network.
- **Hidden layers:** The input values from the input layer are then passed through the neurons of the hidden layer. Here, the values undergo a pre activation function where they are multiplied by their respective weights and added with a bias.
- **Activation function:** Following the pre-activation function, an activation function is applied. There are various activation functions available such as sigmoid, hardtan, and relu.
- **Output layer:** The final layer is the output layer where the calculated output of the neural network is generated and displayed.

The Loss function:

After the inputs undergo forward propagation resulting in an output, the discrepancy between the predicted output and the ground truth can be determined as the error. However, in the context of a neural network, the error in the output is typically not directly computed. Instead, specific loss functions are utilized to calculate the loss, which is subsequently leveraged in optimization algorithms to minimize the loss value.

A variety of loss functions exist for determining the loss, such as mean squared error and binary cross entropy. Each of these functions possesses unique characteristics that can be strategically harnessed based on the specific problem being addressed.

The Gradient Descent Algorithm:

The primary goal of backpropagation is loss minimization, achieved through various optimization algorithms. To simplify, let's begin with the foundational Gradient Descent algorithm. This method involves calculating the loss's rate of change in relation to each parameter, then adjusting each parameter to reduce the loss. Essentially, a unit adjustment in a parameter, such as weight, impacts the loss. If the impact is negative, the weight should be increased to lower the loss; conversely, if the impact is positive, the weight should be decreased. This concept can be expressed mathematically as:

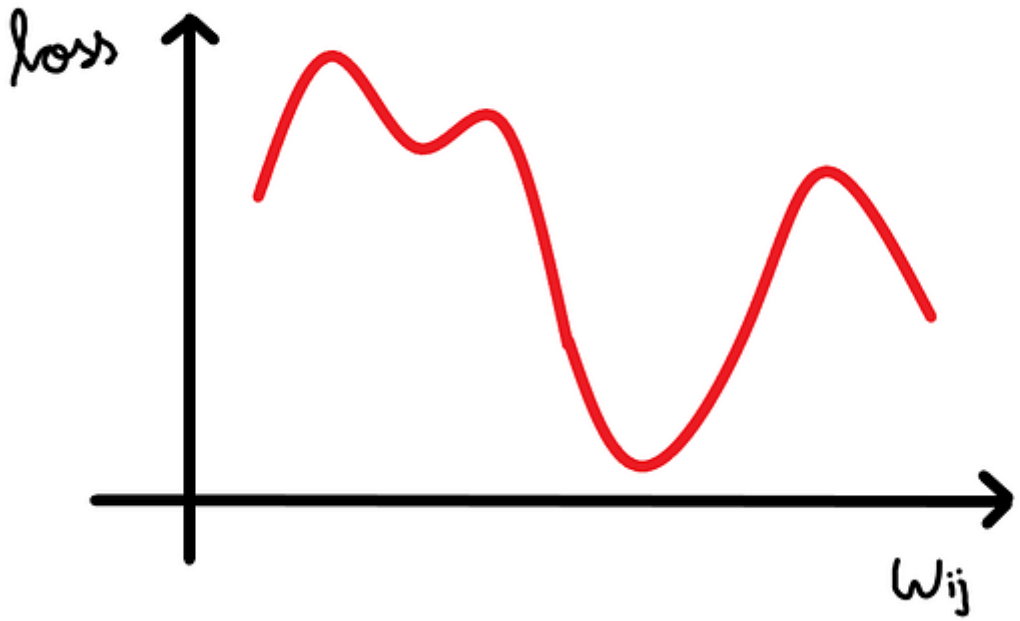
new_weight = old_weight - learning_rate * gradient

In the context of neural networks, the gradient is defined as the partial derivative of the loss function with respect to the weight. The learning rate acts as a scalar factor that either amplifies or reduces the gradient. It will be elaborated upon further in the following sections. The same principle applies to the bias as well, with the updated bias calculated using the formula:

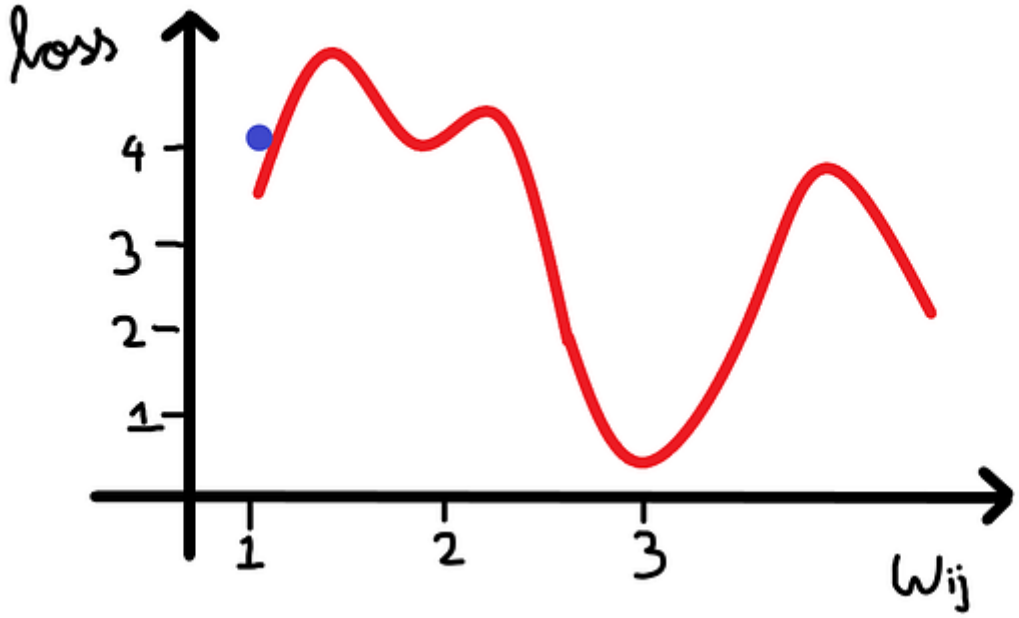
new_bias = old_bias - learning_rate * gradient,

where the gradient represents the partial derivative of the loss function regarding the bias.

The subsequent visualization shows the loss of a neural network concerning the variation in weight of a single neuron.



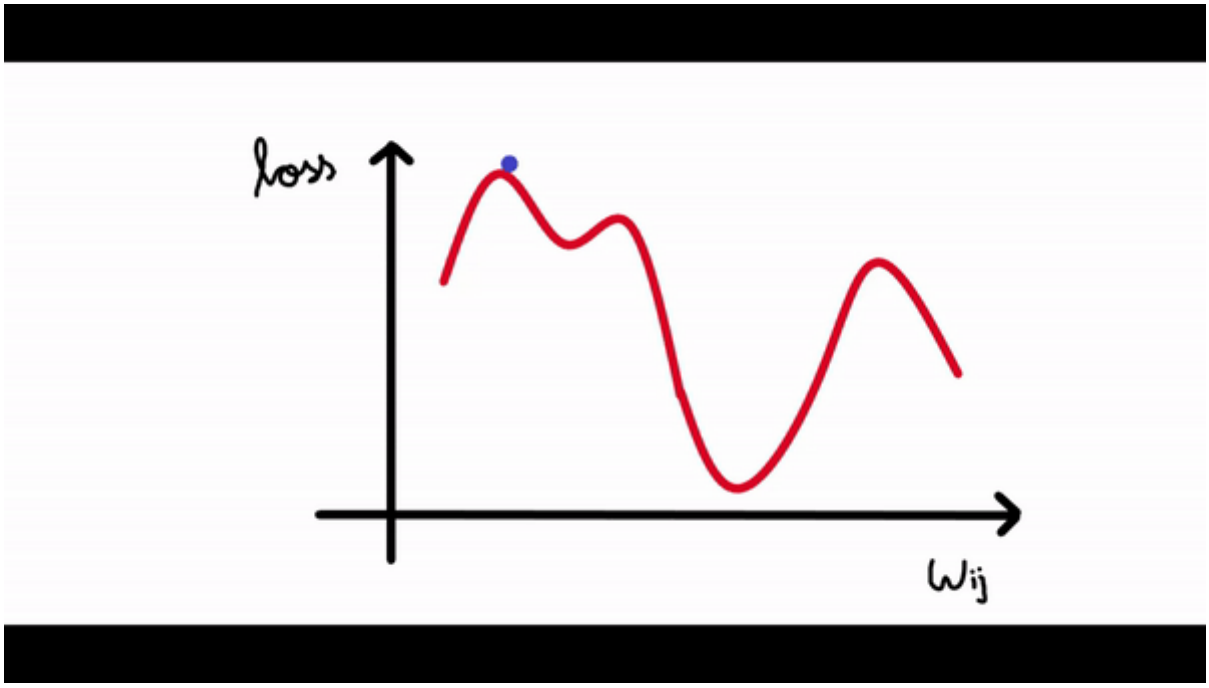
Given the numerous local minima present on the curve, our primary focus is on reaching the global minimum. Currently, our weight value is approximately at the origin, around 1, resulting in a loss of about 4.



To minimize the loss, it is essential to adjust the weight value to approximately 3. It is important to note that the weight should be updated proportionally to the loss. This is why the gradient is determined by taking the partial derivative of the loss with respect to the weight. The steps in the gradient descent algorithm involve:

- Calculating the gradient (a partial derivative of the loss function with respect to the weight or bias)
- Multiplying the gradient by a specified learning rate
- Subtracting the product of the gradient and learning rate from the weight or bias

These steps are repeated iteratively until the loss converges to the global minimum.



More about the Gradient:

The gradient of a straight line is usually calculated by using the general slope formula:

$$m = \frac{rise}{run} = \frac{y_2 - y_1}{x_2 - x_1}$$

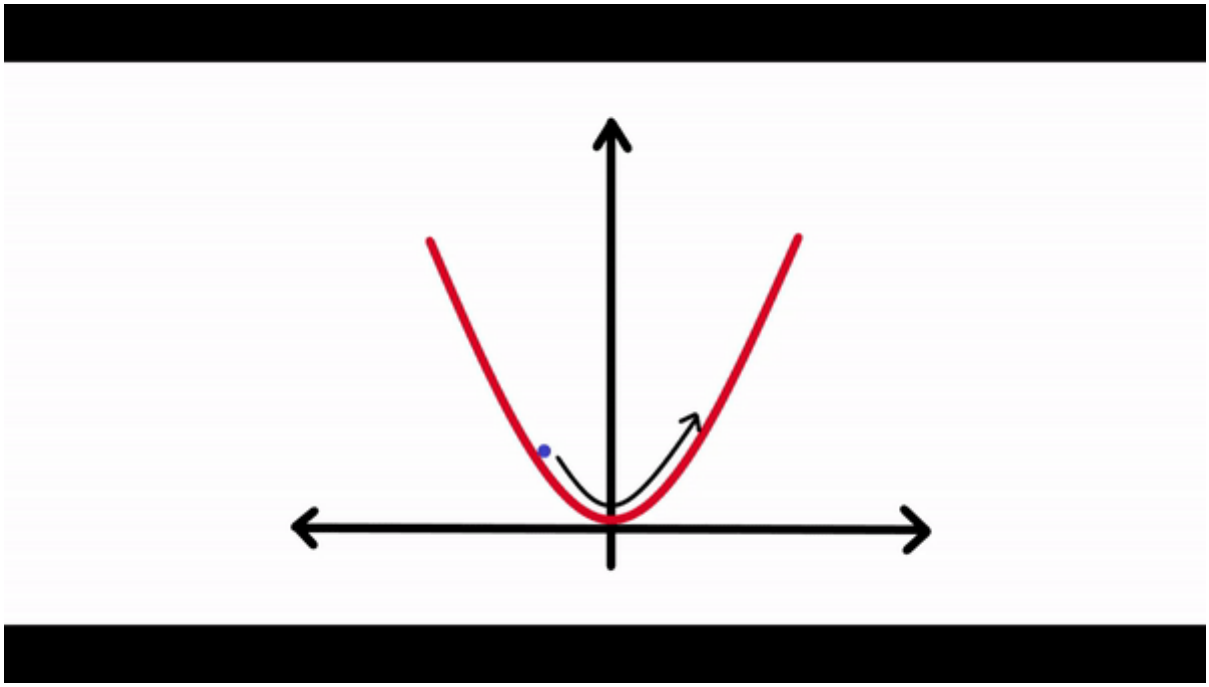
When two points are selected on a line with a certain distance between them, the slope can be calculated. This method is accurate when the graph is a straight line, but may not be ideal for uneven curves. The rise by run calculation may not provide precise results when dealing with irregular curves due to the changing slope at different points. To obtain the most accurate gradient value, considering an infinitesimally small neighborhood or distance for slope calculation would be beneficial.

By calculating the derivative of y with respect to x, we obtain the instantaneous rate of change in y with respect to x. This provides a more precise gradient than the traditional rise by run approach because it is instantaneous.

The same method applies when calculating the rate of change of loss with respect to a weight or bias. Computing the derivative of the loss function with respect to the weight gives us the instantaneous rate of change of loss in relation to the weight.

The Learning Rate:

After calculating the gradient, we must apply a scaling factor to adjust the size of the gradient. This is necessary because our neural network can take excessively large steps when trying to reach the lowest point in the loss curve, potentially preventing convergence to the global minimum. This behavior is illustrated in the graph provided below.



In neural network training, it is important to carefully choose the learning rate to efficiently optimize the model. The loss function may fluctuate in both directions without converging if the learning rate is too high. Conversely, using a learning rate that is too small can significantly prolong the convergence process. Therefore, selecting an optimal learning rate is essential for effective training.

To ensure the effective updating of model parameters, the learning rate is utilized to adjust the size of the gradient during each parameter update. The formula for updating parameters is as follows:

new_weight = old_weight - learning_rate * gradient

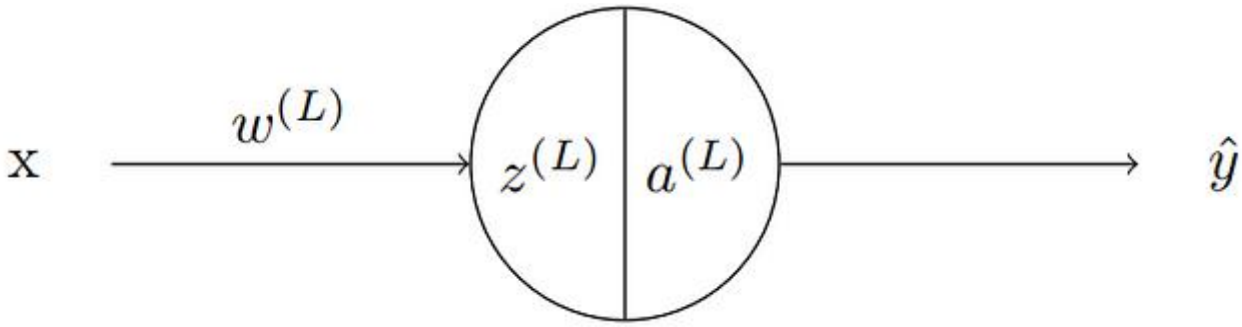
In this iterative process, the learning rate governs the magnitude of each step taken towards the minimum value of the loss function. This careful selection of the learning rate plays a crucial role in guiding the model towards convergence.

Calculating the Gradient:

The loss incurred can be attributed to the various weights and biases of the neurons in the network. Certain weights may have exerted more influence on the output, while others may have had minimal impact.

The primary objective now is to minimize the error in the output. This necessitates the calculation of the gradient in each neuron. Subsequently, this gradient is multiplied with the learning rate and then subtracted from the current weight (or bias) to make adjustments. This iterative process occurs in every neuron within the network.

Let us now consider a scenario where this neural network comprises only a single neuron.



where,

L- Layer number

w- weight

z- pre-activation function

a- activation function

y- output

The pre-activation z can be written as,

$$z^{(L)} = w^{(L)}x + b^{(L)}$$

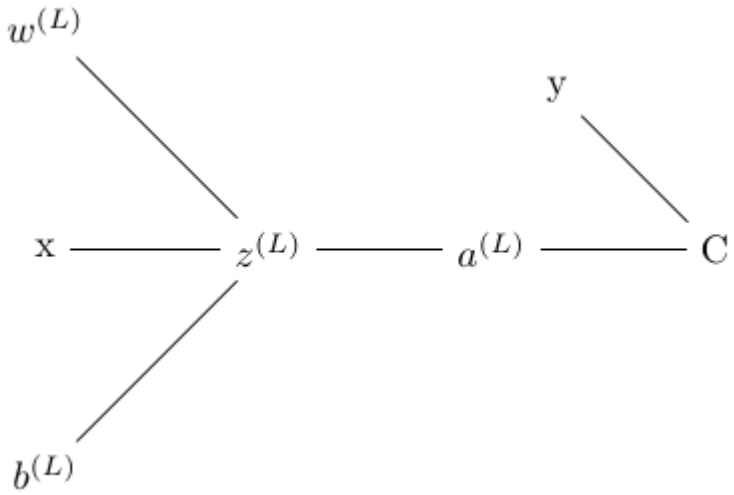
Let's temporarily set aside any bias to keep things simple. The value of z is then passed through an activation function, specifically the sigmoid activation function in this case, denoted by the symbol σ.

$$z^{(L)} = w^{(L)}x$$
$$a^{(L)} = \sigma(z^{(L)})$$
$$\hat{y} = a^{(L)}$$

The network generates the output y-hat, which is then used to calculate the loss using a selected loss function denoted by the letter C. Proceeding to the backpropagation process, the gradient of the loss function is calculated.

$$\text{Gradient} = \frac{\partial C}{\partial w}$$

The value indicates the impact of weight changes on loss. Gradient calculation involves applying the chain rule to derive derivatives. This rule is necessary because weight does not have a direct impact on error. Weight affects the pre-activation function, which then influences the activation function, output, and ultimately loss. The following diagram illustrates the interdependence of terms in the network.



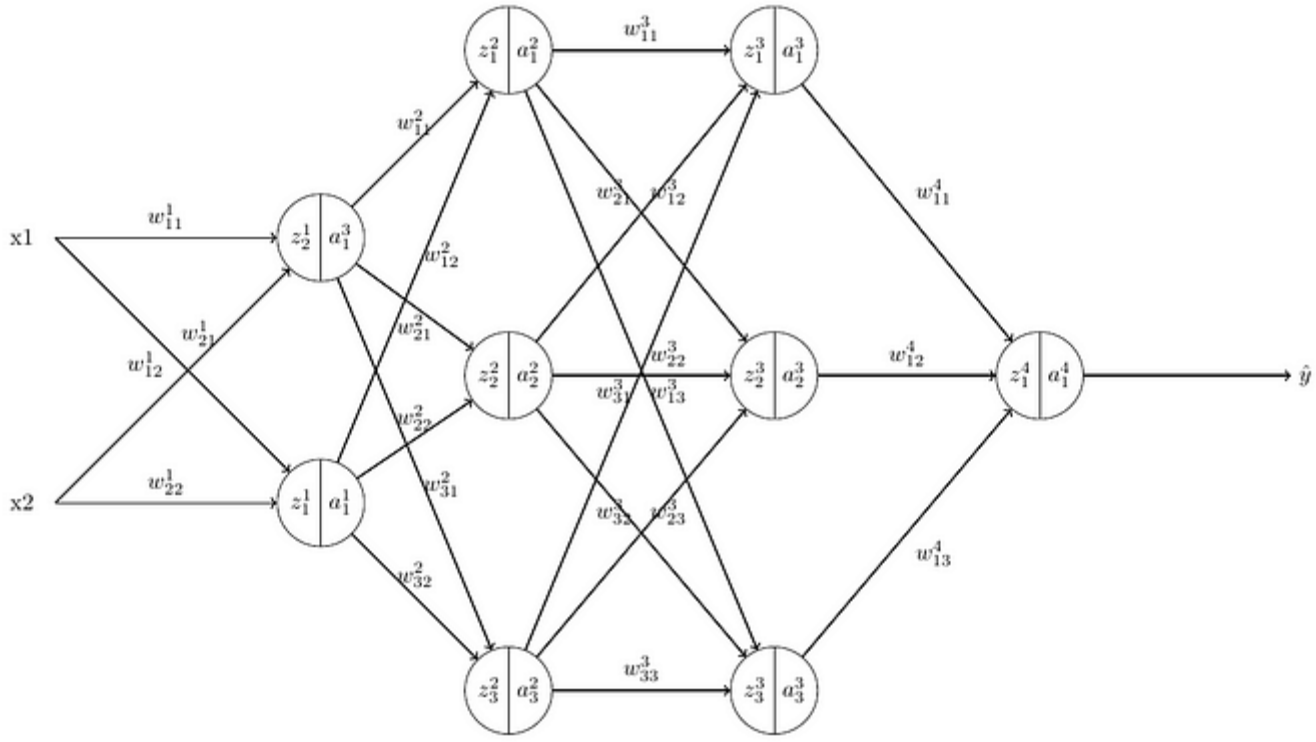
In this diagram, we observe that:

- The pre-activation function is determined by the input, weight, and bias.
- The activation function is in turn influenced by the pre-activation function.
- Finally, the loss is determined by the activation function.

The "y" symbol in the top right corner of the image represents the ground truth against which the predicted output is assessed to calculate the loss. Hence, when applying the chain rule, we arrive at the following outcome:

$$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial a^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial z^{(L)}}{\partial w^{(L)}}$$

We can also use the term "instantaneous rate of change of loss with respect to weight" to describe this gradient. Let's now apply this understanding, obtained from calculating the gradient of a single neuron network, to a complex neural network consisting of four layers: an input layer, two hidden layers, and an output layer.



The pre-activation function of each of these neurons is given by

$$z_j^{(L)} = w_{j0}^{(L)} a_0^{(L-1)} + w_{j1}^{(L)} a_0^{(L-1)} + w_{j2}^{(L)} a_0^{(L-1)} + \ldots + b_j^{(L)}$$

Where,

L- Layer number

j- index of the neuron for which the pre-activation function is being calculated

z- pre-activation function

w- weight of the neuron

a- activated output of the preceding neuron

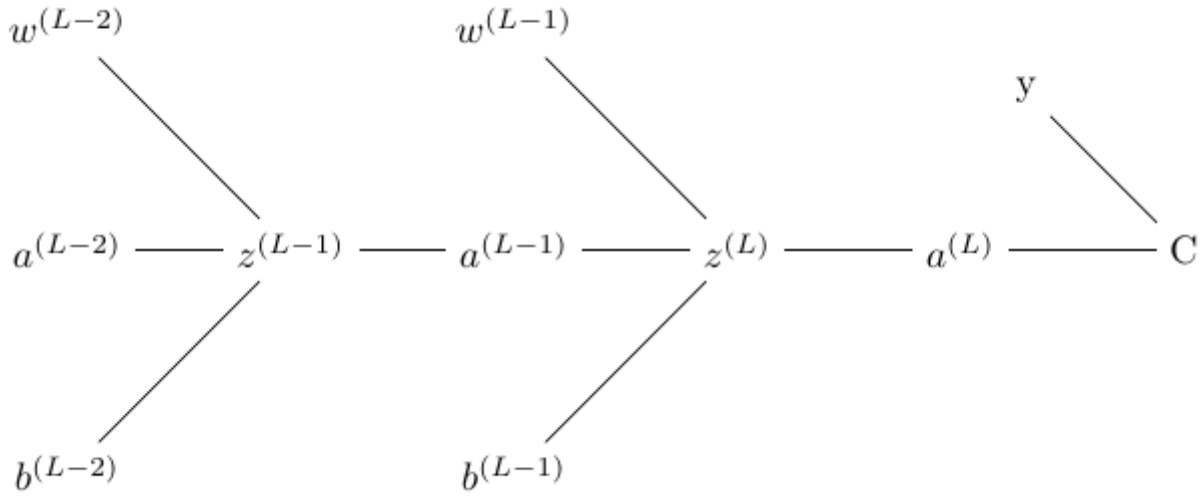
this is true for all the neurons except the input layer where we do not have an activation function. Therefore, in the input layer, z is just the sum of the inputs multiplied with their weights (not the previous neuron's activated output).

Here the gradient is given by,

$$\frac{\partial C}{\partial w_{jk}^{(L)}} = \frac{\partial C}{\partial a_j^{(L)}} \cdot \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \cdot \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}}$$

In the context of neural networks, the weight connecting nodes k and j in layers L-1 and L, respectively, is denoted as w. The preceding node is represented by k, while j signifies the succeeding node. This may prompt the question: why is the weight denoted as wjk instead of wkj? The answer lies in a naming convention that is commonly used when multiplying weights with inputs in matrices. For the purpose of this article, we will set this discussion aside.

To gain a better understanding of the relationship between these terms, refer to the tree diagram below.



Thus we can see that the output of the activation function of a node in the previous layer is given as the input of a node in the succeeding layer.

To determine the gradient at the output node, we require knowledge of the following terms:

- The error derivative concerning the activation function
- The activation function derivative concerning the pre-activated function
- The pre-activated function derivative concerning the weight.

Conversely, when computing the gradient in the hidden layer, it is necessary to calculate the loss function derivative concerning the activation function independently. This calculation must be completed prior to its utilization in the aforementioned formula.

$$\frac{\partial C}{\partial a_k^{(L-1)}} = \sum_{j=0}^{n_{(L-1)}} \frac{\partial C}{\partial a_j^{(L)}} \cdot \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \cdot \frac{\partial z_j^{(L)}}{\partial a_k^{(L-1)}}$$

The equation presented here is similar to the initial one, but with the inclusion of a summation. This is necessary due to the interconnected nature of neurons in a neural network. Unlike weights, the activation function of a single neuron can impact the outcomes of all neurons it is connected to in the subsequent layer.

When calculating the derivative of the loss function with respect to the activation function in the output layer, the chain rule need not be separately applied. This is because the activation function in the output layer directly influences the error, unlike the hidden and input layers whose effects are indirect and take different paths through the network.

After computing the gradient across all nodes in the network, it is scaled by the learning rate and subtracted from the corresponding weights. This process of backpropagation and weight adjustment is crucial for minimizing error and fine-tuning the neural network. This iterative process ultimately leads to improved model performance.

Wait a minute.. what about the bias?

The bias undergoes the same adjustments as the weight in a neural network. Similar to the weight, the bias influences the output of the network. During each training iteration, the gradient is computed for the loss function with respect to both the weight and the bias concurrently.

$$\frac{\partial C}{\partial b_j^{(L)}} = \frac{\partial C}{\partial a_j^{(L)}} \cdot \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \cdot \frac{\partial z_j^{(L)}}{\partial b_j^{(L)}}$$

In calculating the loss function with respect to the activation function of the previous layer for the hidden layers, it is essential to apply the chain rule separately. This process allows for backpropagation of the gradient and adjustment of the bias in each node.

Handling shared weights

In classical convolutional neural networks, shared weights are managed by summing the contributions of each instance where the weight appears during backpropagation, rather than averaging them. If layer yl is post-synaptic to the weight w and we have computed the impact of that layer on the error (∂E/∂yl), the weights are adjusted as follows:

$$\frac{\partial E}{\partial w} = \sum_i \frac{\partial E}{\partial y_i^l} \frac{\partial y_i^l}{\partial w},$$

where I specifies the node within layer l

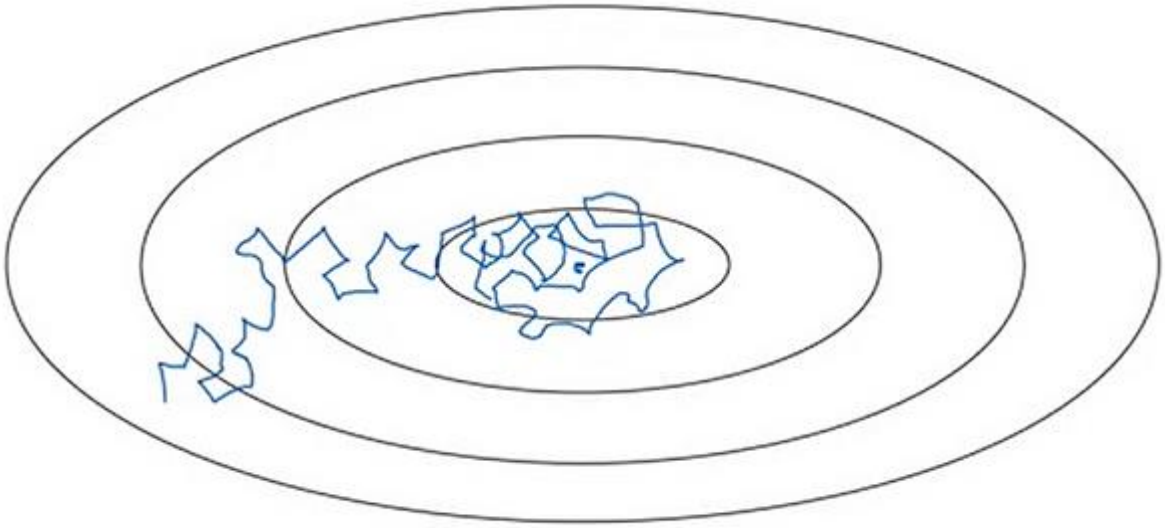
The reason why summation is the appropriate operation is because, when paths from a weight combine from various locations, they merge through summation. For instance, convolution entails summing the paths through the dot operation. Similarly, operations like max pooling and fully connected layers also incorporate the summation of individual paths.

Gradient based Strategies:

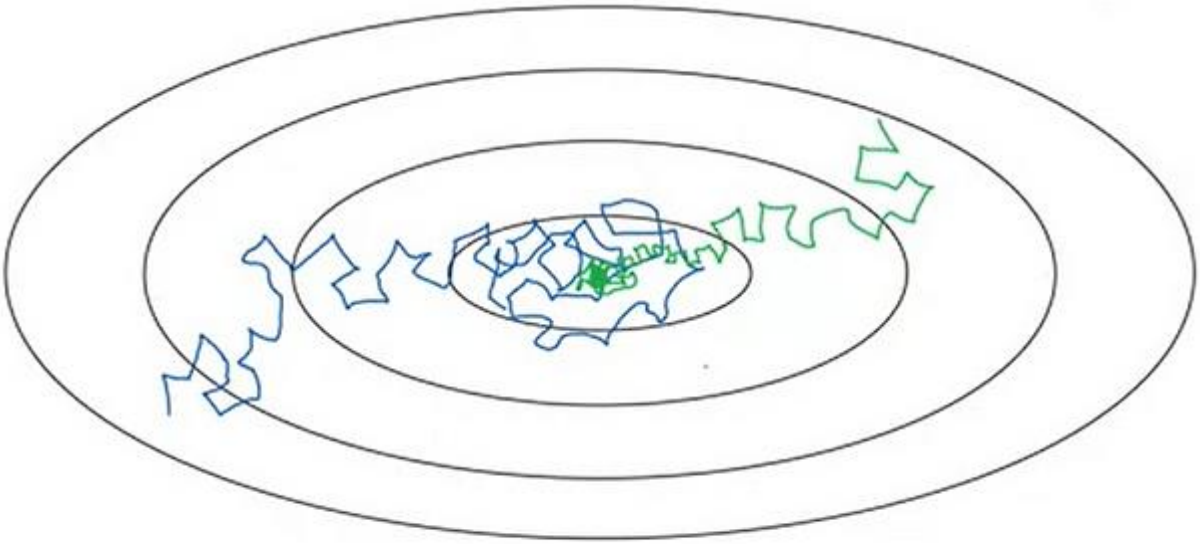
Learning Rate Decay

Learning rate decay is a method used in the training of contemporary neural networks. It involves initially training the network with a high learning rate, which is gradually decreased until a local minimum is reached. This technique has been shown through empirical evidence to benefit both the optimization process and the network's ability to generalize.

How does it work?



Algorithm converging with a constant learning rate (Noisy and represented with Blue)



The algorithm demonstrates convergence while decaying the learning rate over time, indicated by a less noisy trend and being represented with a green line.

In the initial image, where a constant learning rate is utilized, the algorithm's iterative steps towards the minimum exhibit significant noise. After a certain number of iterations, it appears as though the algorithm is meandering around the minimum rather than converging to it effectively.

Conversely, in the second image where the learning rate decreases over time (illustrated by a green line), the algorithm's initial large learning rate allows for relatively quick learning. As the algorithm approaches the minimum and the learning rate diminishes, it begins to oscillate within a narrower range around the minimum, as opposed to venturing far away from it.

Learning rate decay (common method):

$$\alpha = (1/(1+decayRate \times epochNumber)) * \alpha 0$$

1 epoch : 1 pass through data

α : learning rate (current iteration)

α0 : Initial learning rate

decayRate : hyper-parameter for the method

Let's take a rough example for the above method for better intuition :

Suppose we have α0 = 0.2 and decay rate=1 , then for the each epoch we can examine the fall in learning rate α as:

Epoch 1: alpha 0.1

Epoch 2: alpha 0.067

Epoch 3: alpha 0.05

Epoch 4: alpha 0.04

This is a typical method (commonly used) to apply learning rate decay for training neural networks

Gradient Descent

Gradient descent is an optimization algorithm known as a first-order optimization algorithm due to its explicit utilization of the first-order derivative of the target objective function.

First-order methods rely on gradient information to help direct the search for a minimum …

- The derivative of a function, also known as the first-order derivative, represents the rate of change or slope of the function at a given point. When dealing with a function that depends on multiple input variables, it is considered a multivariate function, with the inputs seen as a

vector. In this case, the derivative of the multivariate function is referred to as the "gradient," which is a vector representing the rate of change in each direction.

- **Gradient**: First-order derivative for a multivariate objective function.
  The derivative or gradient indicates the direction of greatest increase of the target function at a specific input. Gradient descent is an optimization algorithm that minimizes a target function by moving in the opposite direction of the gradient. This algorithm necessitates the existence of both the target function being optimized and its derivative. The target function, denoted as f(), provides a score for a set of inputs, while the derivative function, denoted as f'(), calculates the slope of the target function at those inputs. The gradient descent algorithm starts at a designated point (x) in the input space, typically chosen randomly.

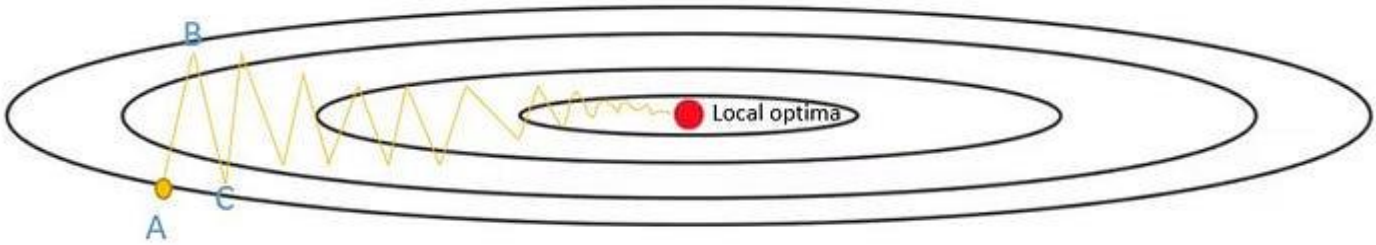
- The next step involves calculating the derivative and determining the appropriate direction in the input space to move in order to optimize the target function. This direction is determined by the gradient, calculated as the product of the step size (referred to as alpha or the learning rate) and the gradient itself. By subtracting this value from the current point, we ensure that we are moving in the opposite direction of the gradient, ultimately leading us downhill on the target function.
- ***x = x – step_size * f'(x)***
- As the gradient magnitude increases at a specific point in the objective function, the step taken in the search space also increases. This step size is determined by a step size hyperparameter.
- **Step Size** (*alpha*): The hyperparameter that dictates the magnitude of movement in the search space relative to the gradient during each iteration of the algorithm is known as the learning rate. An excessively small step size may result in minimal progress and prolong the search process. Conversely, an excessively large step size can cause the search to fluctuate within the space and overlook optimal points.
- Now that we are familiar with the gradient descent optimization algorithm, let's take a look at the momentum

  Momentum-Based Learning

  The use of Momentum in Gradient Descent always yields faster convergence compared to the Standard Gradient Descent algorithm. By computing the exponentially weighted average of gradients and utilizing this value for weight updates, Gradient Descent with momentum operates more efficiently than its traditional counterpart.

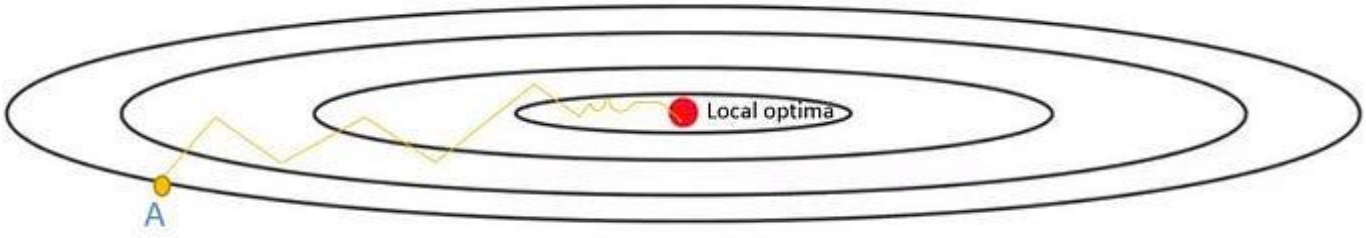
How it works ?

Consider an example where we are trying to optimize a cost function that has contours like the one below and the red dot denotes the local optima (minimum) location.



We initiate gradient descent from point 'A' and reach point 'B' after one iteration, which is on the opposite side of the ellipse. Subsequent descent leads us to point 'C'. Through multiple iterations of gradient descent, fluctuating between ascent and descent, we progress towards the local optimum. Increasing the learning rate intensifies the vertical oscillations, which in turn hinders our descent and restricts the possibility of using a significantly higher learning rate.

By using the exponentially weighted average dW and db values, we tend to average the oscillations in the vertical direction closer to zero as they are in both (positive and negative) directions. Whereas all the derivatives point to the right of the horizontal direction in the horizontal direction, the average in the horizontal direction will still be quite large. It enables our algorithm to take a straighter forward path to local optima and to damp out vertical oscillations. Because of this, the algorithm will end up with a few iterations at local optima.



Implementation

During backward propagation, we utilize dW and db to adjust our parameters W and b in the following manner:

W = W - learning rate * dW

b = b - learning rate * db

To incorporate momentum, we compute exponentially weighted averages of dW and db, instead of using them independently for each epoch:

VdW = β * VdW + (1 - β) * dW

Vdb = β * Vdb + (1 - β) * db

The hyperparameter beta (β) represents momentum and ranges from 0 to 1. By calculating the weighted average, we strike a balance between previous values and the current value. Updates to our parameters are made after computing the exponentially weighted averages:

W = W - learning rate * VdW

b = b - learning rate * Vdb

How to choose Beta?

- A higher momentum (beta) is recommended to alleviate the impact of update fluctuations, as greater emphasis is placed on the historical gradients.
- It is advised to use the default value of β = 0.9, but it can be adjusted within the range of 0.8 to 0.999 if necessary.
- Momentum factors in previous gradients to dampen the effect of gradient variations. It can be incorporated in batch gradient descent, mini-batch gradient descent, or stochastic gradient descent algorithms.

Parameter-Specific Learning Rates (Or)

Adagrad (Adaptive Gradient Descent) Deep Learning Optimizer

The term Adagrad stands for Adaptive Gradient Optimizer, a technique used alongside other optimizers such as Gradient Descent, Stochastic Gradient Descent, and mini-batch SGD to minimize the loss function with regards to the weights. The formula for updating weights is as described below:

$$(\mathrm{w})_{\mathrm{new}} = (\mathrm{w})_{\mathrm{old}} - \eta \frac{\partial \mathrm{L}}{\mathrm{dw}(\ \mathrm{old}\ )}$$

Based on iterations, this formula can be written as:

$$w_t = w_{t-1} - \eta \frac{\partial L}{\partial w(t-1)}$$

where

w(t) = value of w at current iteration, **w(t-1)** = value of w at previous iteration and **η** = learning rate.

In traditional SGD and mini-batch SGD, a fixed value of learning rate (η = 0.01) is commonly used for all weights or parameters. However, Adagrad Optimizer introduces a innovative concept where each weight is assigned a unique learning rate (η). This adaptation is particularly crucial when dealing with real-world datasets that consist of a combination of sparse and dense features. For instance, in scenarios such as Bag of Words where most features are zero (sparse), while others are non-zero (dense), applying the same learning rate to all weights can be suboptimal for the optimization process. The weight updating formula for Adagrad is:

$$\mathrm{w_t} = \mathrm{w_{t-1}} - \eta'_{\mathrm{t}} \frac{\partial \mathrm{L}}{\partial \mathrm{w(t-1)}}$$

Where **alpha(t)** denotes different learning rates for each weight at each iteration.

$$\eta'_{\mathrm{t}} = \frac{\eta}{\mathrm{sqrt}(\alpha_{\mathrm{t}} + \epsilon)}$$

In this scenario, η is a constant, while epsilon is a small positive value used to prevent division by zero error. If alpha(t) approaches 0, setting the learning rate to zero may result in w(old) = w(new) after multiplying by the derivative, leading to slow convergence.

$$\alpha_t = \sum_{i=1}^{t} g_i^2$$

$$g_i = \frac{\partial L}{\partial w(old)}$$

g_i is derivative of loss with respect to weight and g_i^2 will always be positive since its a square term, which means that alpha(t) will also remain positive, this implies that *alpha(t) >= alpha(t-1).*

It can be seen from the formula that as *alpha(t)* and $\eta_t^{'}$ is inversely proportional to one another, this implies that as **alpha(t)** will increase, $\eta_t^{'}$will decrease. This means that as the number of iterations will increase, the learning rate will reduce adaptively, so you no need to manually select the learning rate.

Advantages of Adagrad:

- No manual tuning of the learning rate required.
- Faster convergence
- More reliable

One main **disadvantage** of Adagrad optimizer is that alpha(t) can become large as the number of iterations will increase and due to this $\eta_t^{'}$ will decrease at the larger rate. This will make the old weight almost equal to the new weight which may lead to slow convergence.

What is Gradient Clipping and how does it occur?

Gradient clipping is a technique that involves restricting the error derivatives before passing them through the network. This helps prevent exploding gradients and stabilizes training. The capped gradients are then used to update the weights, leading to smaller weight updates. Gradients are capped by either scaling or clipping them to a specified threshold to prevent them from becoming too large or too small. This process ensures that the gradients remain within a certain range during training.

Types of Clipping Techniques

Gradient clipping can be applied in two common ways:

- Clipping by value
- Clipping by norm

Let’s look at the differences between the two.

Gradient Clipping-by-value

To limit the gradient by value, set a minimum and maximum threshold. If the gradient surpasses the maximum value, it will be kept at that maximum level. Likewise, if the gradient falls below the minimum, it will be raised to meet the minimum value.

Gradient Clipping-by-norm

Clipping the gradient by norm ensures that the gradient of every weight is clipped such that its norm won't be above the specified value.

"*…a norm is a function that accepts as input a vector from our vector space V and spits out a real number that tells us how big that vector is*. "

Gradient clipping in deep learning frameworks

Now that you understand what exploding gradients are and how to fix them, let's explore how gradient clipping can help in deep learning.

How to prevent exploding gradients with gradient clipping

Each deep learning framework applies gradient clipping in its own unique way, but the basic principles remain consistent across the board.

Examples of gradient clipping in action

Let's go over a few examples of how you can use gradient clipping to control gradients in your deep learning models. Let's dive in!

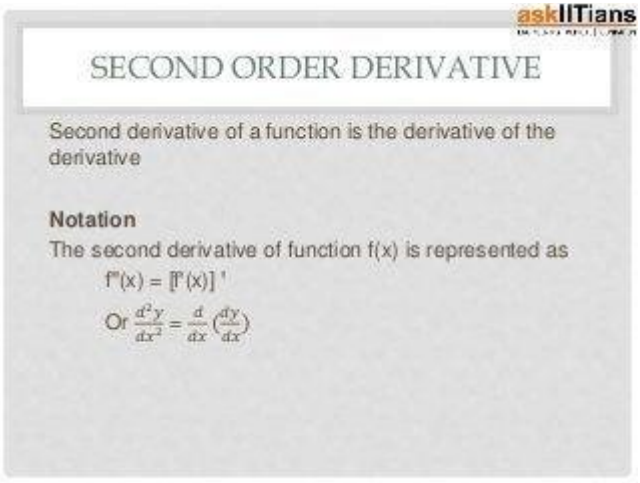
Second-Order Derivatives:

When we want to know how the graph of a function looks, we can use the second-order derivatives. This helps us see if the graph is curving upward or downward, based on its

concavity. Concavity is like the shape of the graph - it can be either concave up or concave down.

To find the second derivative, you just take the first derivative of the first derivative. And this idea keeps going - each time you take the derivative, you're looking at the previous derivative. Even if it sounds complicated, it all comes down to understanding how the function behaves.

The second derivative is the slope of the first derivative.



Typically, the concavity of a graph is determined by the second derivative of the function in question. A positive second derivative indicates upward concavity in the graph of the function, while a negative second derivative indicates downward concavity.

We are aware that the second derivative of a function plays a crucial role in determining the local maximum or minimum values, as well as inflection points. These can be identified using the following conditions:

- *If f''(x) is less than 0, then the function f(x) exhibits a local maximum at point x.*
- *If f''(x) is greater than 0, then the function f(x) displays a local minimum at point x.*

- *If f''(x) equals 0, then no definitive conclusions can be drawn about the point x, which may be a potential inflection point.*

Relation between first and second derivative

The relationship between the 1st and 2nd derivative mirrors that of a function and its 1st derivative. The 1st derivative signifies the rate of change, while the 2nd derivative signifies the rate of change of the rate of change.

One practical example is in measuring distance over time, where the 1st derivative represents velocity (speed) and the 2nd derivative represents acceleration.

To determine if a maximum or minimum is present in the 1st derivative, plug the x value into the equation. If the resulting y value is less than 0, it is a maximum. If the value is greater than 0, it is a minimum. Additionally, zeros of the function indicate inflection points for concavity (whether the graph is facing up or down). To test these points, enter values higher and lower than 0. If there is a sign change, it is an inflection point.

UNIT-III TEACHING DEEP LEARNERS

What is generalization and why is it needed?

The concept of generalization in deep learning refers to a model's capacity to effectively predict patterns in unseen or new data that align with the distribution of the training data. In other words, a model's generalization abilities determine its accuracy in analyzing and predicting outcomes in new data post training on a designated dataset. This discussion will delve into how the variance and bias of a model influence its generalization capabilities.

Variance-bias trade-off

In machine learning, two fundamental concepts to consider are variance and bias. Variance characterizes the range of predictions produced by a model, indicating how dispersed a set of data points is from their true values. On the other hand, bias measures the deviation of predictions from their true values.

Every machine learning model usually comes under any one of the following stages:

- Low bias - Low variance
- Low bias - high variance
- High bias - Low variance
- High bias - high variance

The stages mentioned above categorize models based on their bias and variance. Models with low bias and high variance are labeled as overfitted, whereas models with high bias and low variance are labeled as underfitted. The concepts of underfitting and overfitting can be further illustrated in the graph displayed below.

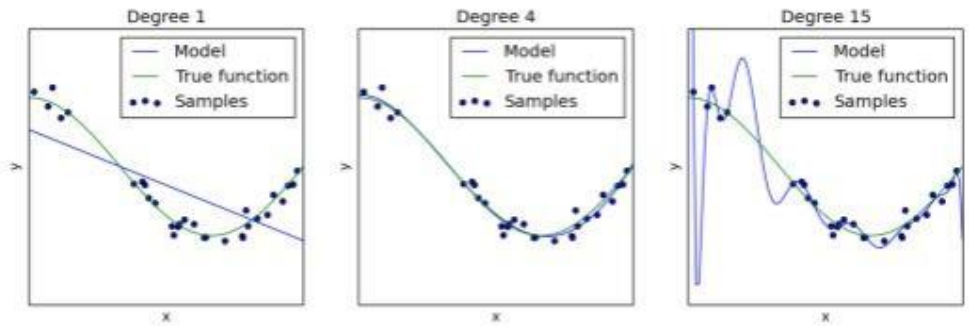


Image source: scikit-learn

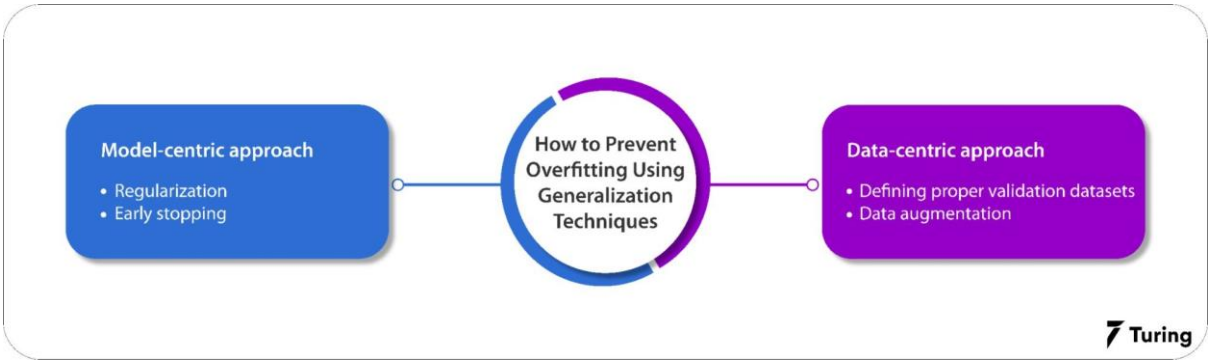
In the image above, the first graph illustrates an underfitted model, where the model has not effectively learned the patterns of the training data and struggles to generalize on new data. The second graph depicts a properly fitted model, which has successfully identified the

patterns within the training data. Finally, the third graph shows an overfitted model, where the model has memorized the exact patterns of the training data, leading to poor generalization on unseen data.

By aiming for optimal generalization, we can strike a balance between underfitting and overfitting, ensuring that a trained model performs as expected.

Generalization techniques to prevent overfitting in deep learning

This section will examine various methods for preventing overfitting in deep learning models. These techniques can be broadly classified as either data-centric or model-centric. The goal of these approaches is to train the model to generalize well on the validation dataset and to identify meaningful patterns within the training data.



Data-centric approach

The data-centric method focuses on tasks such as data cleaning, feature engineering, and creating accurate validation and testing datasets. We will explore key data-centric generalization techniques, including creating appropriate validation sets and performing data augmentation.

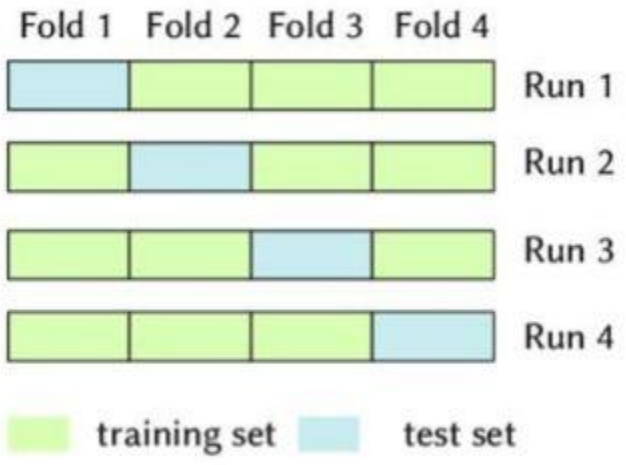
Defining proper validation datasets

The initial stage in predictive modeling is establishing a suitable validation dataset, as it is crucial for obtaining a accurate depiction of real-world data. A well-defined validation set enables effective evaluation of our machine learning model, aiding in the determination of its generalization capability.

It is important for the dataset used in training a machine learning model to contain a wide range of data samples to maximize the model's ability to detect patterns from the data. The performance of the model is influenced by the number of data samples available. Typically, deep learning models utilized in computer vision and natural language processing (NLP) tasks are trained on millions of data samples (images or text) to enhance the model's generalization capabilities.

To enhance learning on the training dataset, it is advisable to utilize cross-validation techniques such as K-fold or stratified K-fold during training. These techniques produce excellent results as they allow the model to learn from the entire dataset while using it for both training and validation simultaneously.

Below is an example of K-fold cross-validation technique:



Source:
https://www.researchgate.net/figure/The-technique-of-KFold-cross-validation-illustrated-here-for-the-case-K-4-involves_fig10_278826818

Penalty Terms

Regularization adjusts data towards specific values by introducing a tuning parameter to encourage those values.

L1 regularization: imposes an L1 penalty based on the absolute value of coefficients, limiting their size and potentially resulting in sparse models with some coefficients being eliminated. This method is utilized in Lasso regression.

L2 regularization: applies an L2 penalty based on the square of coefficients, shrinking all coefficients by the same factor without eliminating any. Ridge regression and SVMs utilize this approach.

Elastic nets: combine L1 and L2 methods while adding a hyperparameter for further customization.

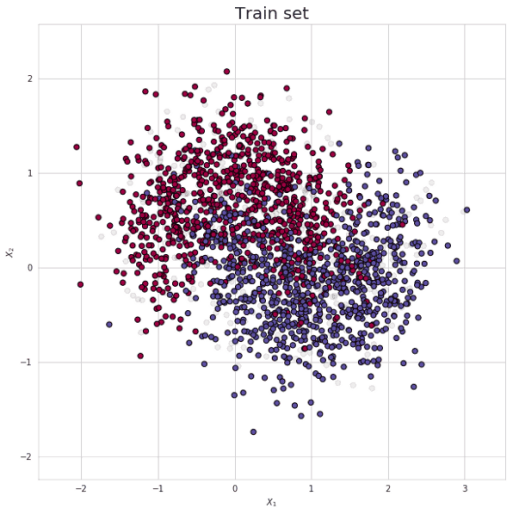
What is Regularization?

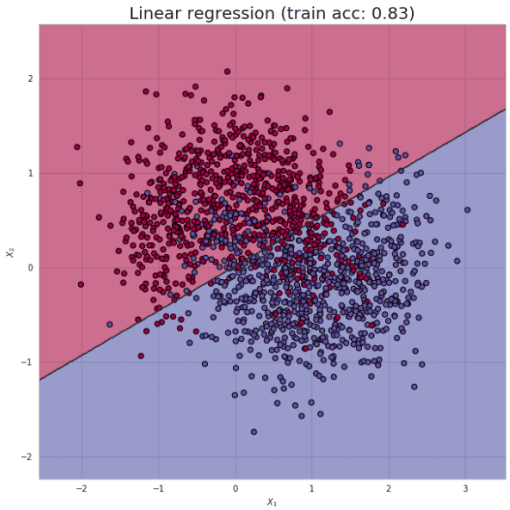
Regularization can be described as any adjustment or amendment made to the learning algorithm to minimize its error on a test dataset, also referred to as generalization error, rather

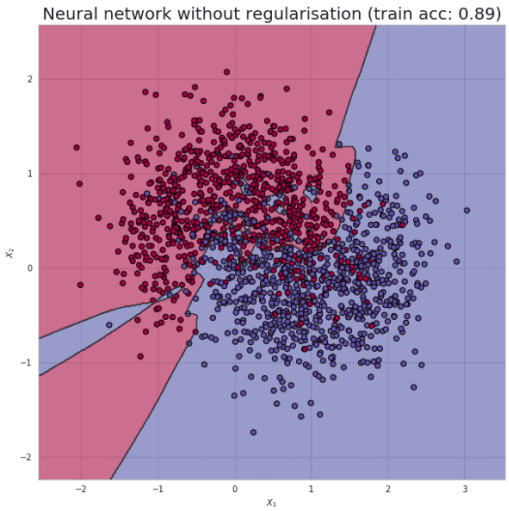
than the provided or training dataset. Within learning algorithms, there exist numerous forms of regularization techniques, each aimed at addressing various obstacles. These techniques can be classified based on the specific challenges they are designed to address:

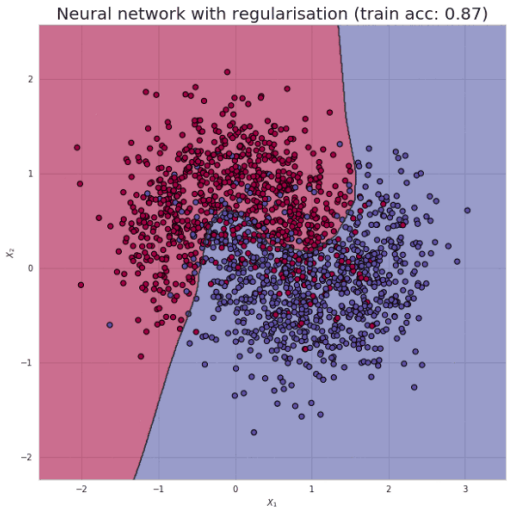
1. Some individuals may opt to impose additional restrictions on the training of a machine learning model by specifying limitations on the range or type of parameter values.
2. Alternatively, others may introduce additional terms in the objective or cost function, such as incorporating a soft constraint related to parameter values. Carefully selecting the appropriate constraints and penalties within the cost function often leads to significant improvements in the model's performance, especially on the evaluation dataset.
3. These supplemental terms can also be formulated based on prior knowledge that is closely related to the dataset or the problem at hand.
4. An established method of regularization involves creating ensemble models, which consider the collective decisions of multiple models that have been trained on distinct subsets of the data.

   The primary goal of regularization is to decrease the excessive complexity of machine learning models and facilitate the learning of a simpler function to enhance generalization.







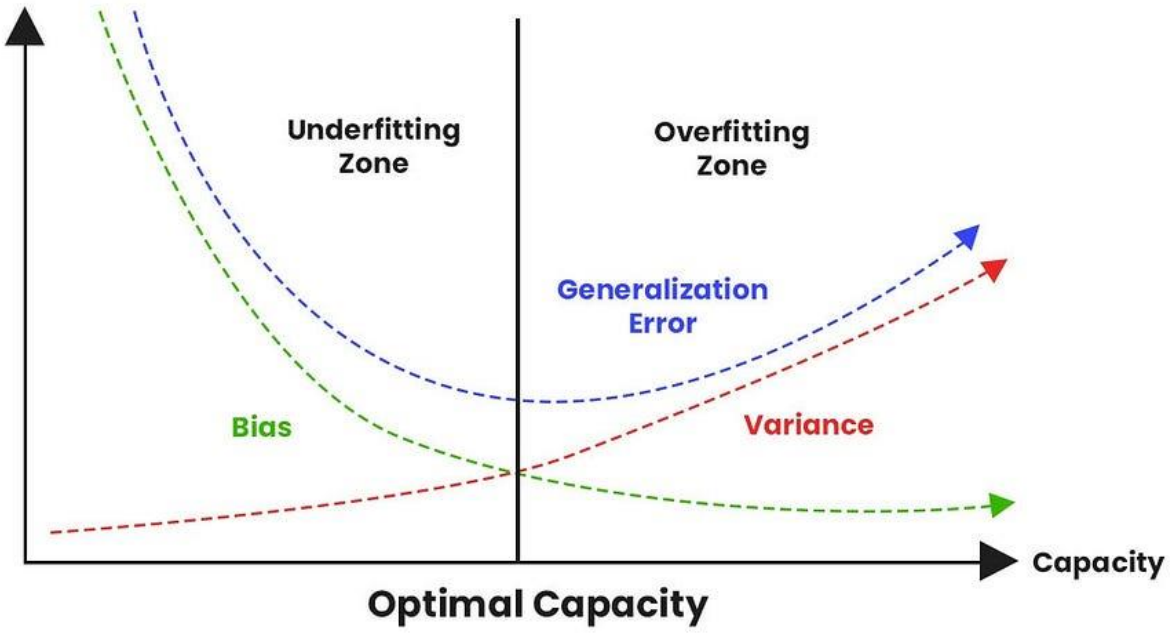


Impact of Regularization in Machine Learning

Regularization in Deep Learning:

The focus of regularization strategies in deep learning models typically involves the regularization of estimators. This leads us to the question: What exactly does it mean to regularize an estimator?

Bias vs variance tradeoff graph here sheds a bit more light on the nuances of this topic and demarcation:



Bias vs Variance tradeoff graph

Regularization is a technique used to control the variance of an estimator by introducing some bias. A successful regularization method strikes a balance between bias and variance, ultimately resulting in a substantial decrease in variance with minimal impact on bias. In essence, the goal is to achieve lower variance without significantly inflating the bias value. We consider two scenarios:

1. The actual data-generating process/function, known as F1, was responsible for producing the dataset.
2. By developing a generating process/function, referred to as F2, that emulates F1 while also investigating alternative generating scenarios/functions.

Regularization techniques assist in transitioning our model from F2 to F1 while avoiding excessive complexity in F2. Deep learning algorithms are typically employed in intricate domains such as images, audio, text sequences, and complex decision-making tasks. The accurate mapping of the True data-generation process - F1 is overwhelmingly challenging.

Therefore, through regularization, our objective is to align our model's F2 function as closely as possible to the original F1 function. The subsequent analogy serves to illustrate this concept more effectively:

Fitting our ML model, F2, onto our true data generation process, F1, is akin to trying to fit a square peg into a round hole with closed approximations. In practical deep learning training, we often discover that the best fitting model, with the least generalization error, is typically a larger model that has been appropriately regularized.

Let's now explore a particular regularization technique designed to construct a robust model with substantial regularization by leveraging parameter-based penalties, known as Parameter Norm Penalties.

When utilizing this regularization technique, the ability of models such as neural networks, linear regression, or logistic regression is constrained through the addition of a parameter norm penalty Ω(θ) to the objective function J. This can be illustrated by the following equation:

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha\Omega(\theta)$$

Parameter Norm Penaltes:

The hyperparameter α, residing in the range [0, ∞), determines the weight given to the norm penalty term, Ω, within the standard objective function J.

When α is set to 0, there is no regularization applied. Increasing the value of α corresponds to an increase in the level of regularization being applied.

This form of regularization penalizes only the weights of the affine transformation in each layer of the network, leaving the biases untouched. The rationale behind this approach is that biases generally require less data to adjust compared to weights. In the field of deep learning, there is a growing interest in utilizing a distinct parameter to achieve a similar outcome.

L1 Parameter Regularization:

L1 regularization is a technique used for regularization that is known for its specificity compared to gradient descent, although it still involves solving an optimization problem through gradient descent.

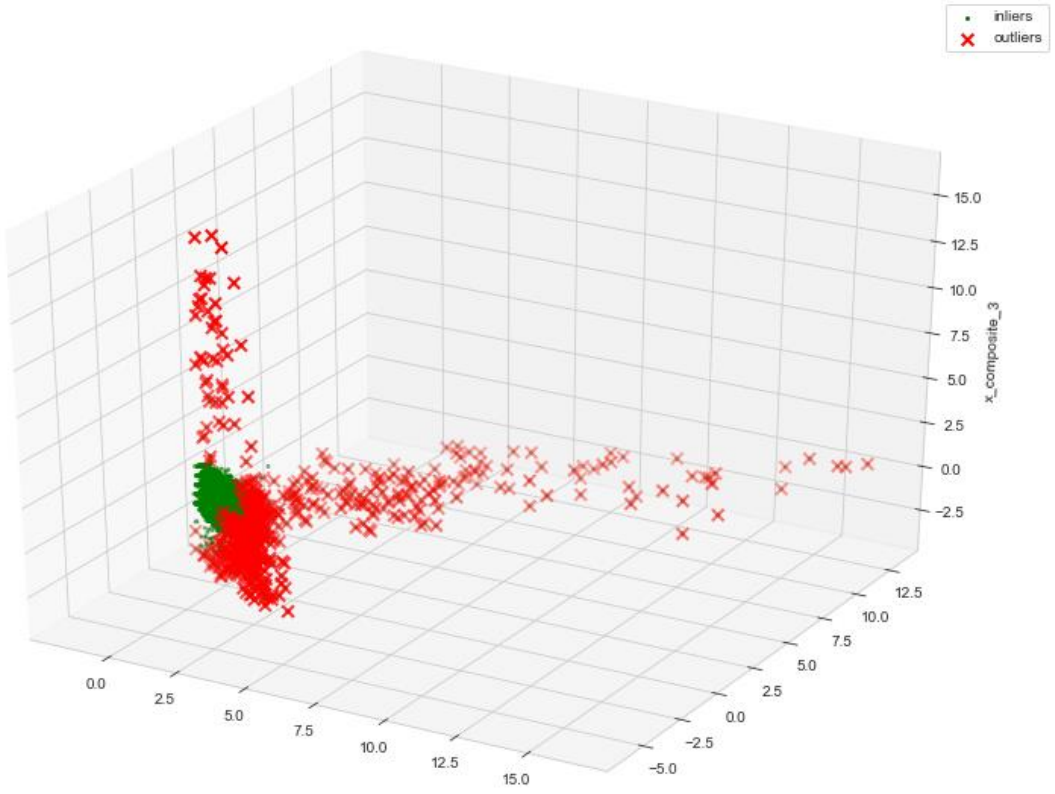
Formula and high-level meaning over here:

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} |w_i|$$

Formula for L1 regularization terms

Lasso Regression, also known as Least Absolute Shrinkage and Selection Operator, incorporates the "Absolute value of magnitude" of coefficients as a penalty term within the loss function.

Lasso eliminates less significant features by reducing their coefficients to zero, effectively removing them altogether. This technique is particularly effective for feature selection when dealing with a large number of features.
The L1 regularizer seeks to minimize the norm of the parameter vector, which represents the length of the vector. This optimization problem pertains to optimizing the parameters of individual neurons, single-layer neural networks, and specifically single-layer feed-forward neural networks.



Constructing a machine learning model that considers outliers for inclusion in cost penalties is a challenging endeavor. The accompanying image displays a visualization of a dummy dataset where the L1 method aids in detecting outliers located far from the rest of the data points.

A useful method for grasping this concept is viewing it as a tactic to expand the parameter hyperspace encompassing the true parameter vector. The objective is to pinpoint the most distinct edge closest to the parameter vector. Key factors to consider in L1 regularization are:

Benefits of L1 Regularization:

1. **L1 regularization is simple to implement and can be trained efficiently as a one-time process. Once trained, the parameter vector and weights can be readily utilized.**

2. **L1 regularization is effective in handling outliers as it induces sparsity in the solution, resulting in most coefficients being zero. This ensures that less important features or noise terms are eliminated, thereby enhancing robustness to outliers.**

To comprehend the point discussed above, we will explore the following example to grasp the concept of algorithms being influenced by outliers.

1. *Consider a scenario where we are working on the classification of images depicting various species of birds through a neural network equipped with multiple parameters.*
2. *Suppose we acquire an image sample featuring birds belonging to a particular species, without any noticeable distinction from the rest.*
3. *Upon incorporating this image into the training data and initiating the neural network training process, we encounter a situation akin to introducing an outlier amidst the existing dataset. This outlier becomes discernible at the periphery of the hyperspace, near the hyperplane. However, as we approach the hyperplane, the outlier becomes increasingly distant, positioning itself as an anomaly.*
4. *Performing iterative dropout is the recommended solution in such situations. While L1 regularization offers a quick fix, ultimately a precise decision must be made on where to trim the edges of the hyperspace.*
5. *Iterative dropout provides a method for determining the exact cutoff points. Although it may require more time during training, it could lead to a clearer understanding of the boundaries of the hyperspace in the end.*

In addition to reducing coefficients, the lasso algorithm conducts feature selection. This is evident in its full name, which includes the term 'selection'. By setting some coefficients to zero, specific features are effectively eliminated from the model.

L2 Parameter Regularization:

The Regression model that uses L2 regularization is called Ridge Regression.

$$J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right]$$

$$\min_\theta J(\theta)$$

Formula for Ridge Regression

Regularization introduces a penalty to the model as complexity grows. The regularization parameter (lambda) penalizes all parameters excluding the intercept to promote generalization and prevent overfitting. Ridge regression incorporates the "squared magnitude of the coefficient" as a penalty term in the loss function. In the accompanying image, the boxed section corresponds to the L2 regularization element/term.

$$\sum_{i=1}^{n}(y_i - \sum_{j=1}^{p}x_{ij}\beta_j)^2 + \lambda\sum_{j=1}^{p}\beta_j^2$$

Lambda is a hyperparameter.

If lambda is zero, then it is equivalent to OLS.

The Ordinary Least Squares (OLS) statistical model is useful for identifying significant features that strongly impact the output.

If lambda is excessive, it may result in excessive weight and cause underfitting. The following are key considerations about L2 regularization:

1. **Ridge regularization enforces small weights without driving them to zero, resulting in a more dense solution. However, it does not promote sparsity.**
2. **Ridge is sensitive to outliers, as the squared terms amplify the errors caused by outliers. The regularization term attempts to correct this by penalizing the weights.**
3. **Ridge regression excels when all input features impact the output and have weights of similar magnitudes.**
4. **L2 regularization is capable of capturing intricate data patterns.**

Differences, Usage and Importance:

Understanding the distinction between L1 and L2 regularization methods is crucial. Unlike L2 regularization, L1 regularization tends to produce a more sparse solution. Sparsity, in this context, denotes the occurrence of certain parameters reaching an optimal value of zero. The sparsity exhibited by L1 regularization showcases a unique behavior compared to L2 regularization.

The sparsity feature utilized in L1 regularization has been widely adopted as a method for selecting features in machine learning. Feature selection simplifies machine learning tasks by effectively determining which subset of available features should be included in the model. Lasso combines an L1 penalty with a linear model and a least-squares cost function. The L1 penalty results in some weights being reduced to zero, indicating that the corresponding features can be safely discarded.

Regularization as Bayesian inference?
Several regularization techniques can be seen as performing MAP Bayesian inferences:

1. **L2 regularization can be understood as being very similar to MAP Bayesian inference with a Gaussian prior distribution placed on the weights.**
2. **L1 regularization involves utilizing a penalty term to penalize the cost function, which can be likened to the log-prior term maximized in MAP Bayesian inference when utilizing an isotropic Laplace Distribution over the dataset of real numbers.**

Summary table

The complete post can also be condensed into concise bullet points, which can serve as a helpful tool for interview preparation or for quickly navigating through the content to locate specific information.

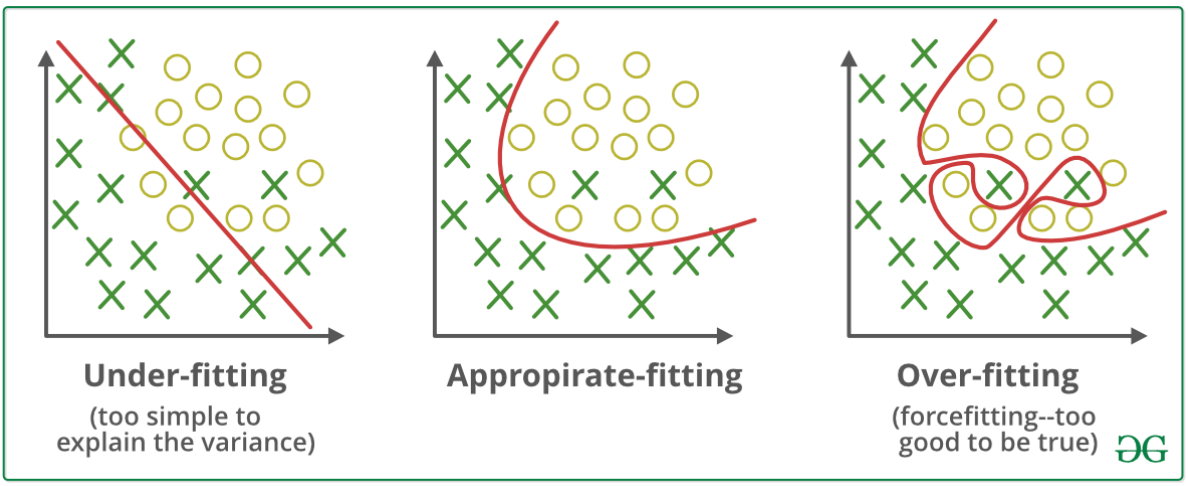
L1 Regularisation	L2 Regularisation
Sum of absolute value of weights	Sum of square of weights
Sparse solution is the outcome	Non-sparse (more segregated) solution
Multiple solutions are possible	Only one solution
Built-in feature selection in the penalty term	No specific feature selection mechanism
Robust to outliers	Not robots to outliers due to square term
Used in datasets with sparse features	Used in dataset with complex features

In the process of machine learning model development, you may have come across a scenario where the training accuracy is significantly higher than the validation or testing accuracy, indicating overfitting. This is a common issue in machine learning that practitioners strive to avoid. In this article, we will explore regularization as a method to address overfitting. But before delving into regularization, let us first understand the concepts of underfitting and overfitting.

What are Overfitting and Underfitting?

Overfitting is the result of a Machine Learning model being excessively tailored to the training set, hindering its performance on new data. In such instances, the model not only learns from the patterns in the training data but also picks up on irrelevant noise, a situation referred to as "memorization" of the training data.

Underfitting occurs when our model is unable to capture even the fundamental patterns present in the dataset. A model suffering from underfitting will struggle to perform adequately on both the training and validation data. To mitigate underfitting, it is necessary to enhance the complexity of the model or expand the feature set.

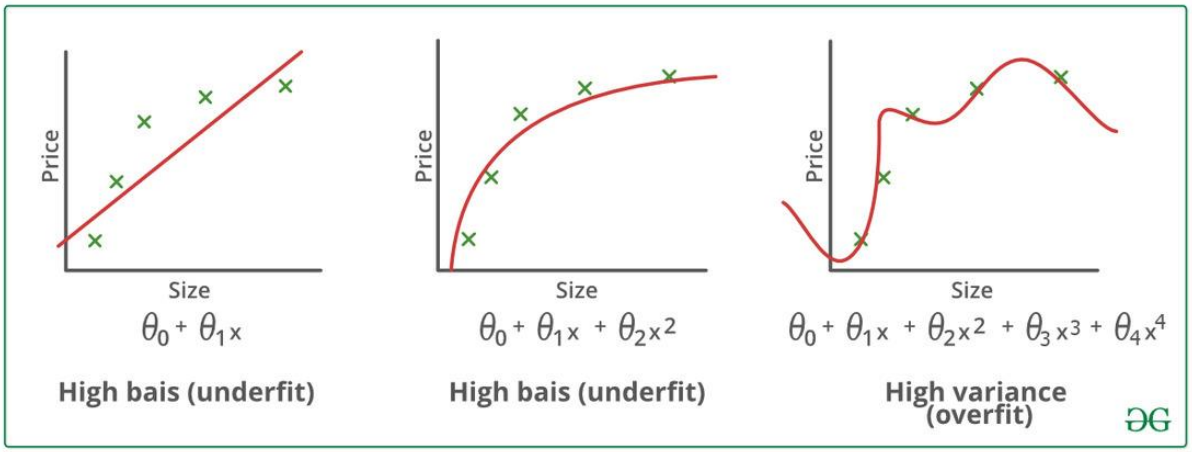


Overfitting and Underfitting in Machine Learning

What are Bias and Variance?

Bias occurs when attempting to fit a statistical model to real-world data that does not align perfectly with a mathematical model. Using an overly simplified model may result in High Bias, where the model is unable to effectively learn patterns within the data and performs poorly.

On the other hand, Variance pertains to the error that arises when making predictions with data that the model has not encountered before. High variance occurs when the model learns the noise present in the data, leading to potential inaccuracies in predictions.



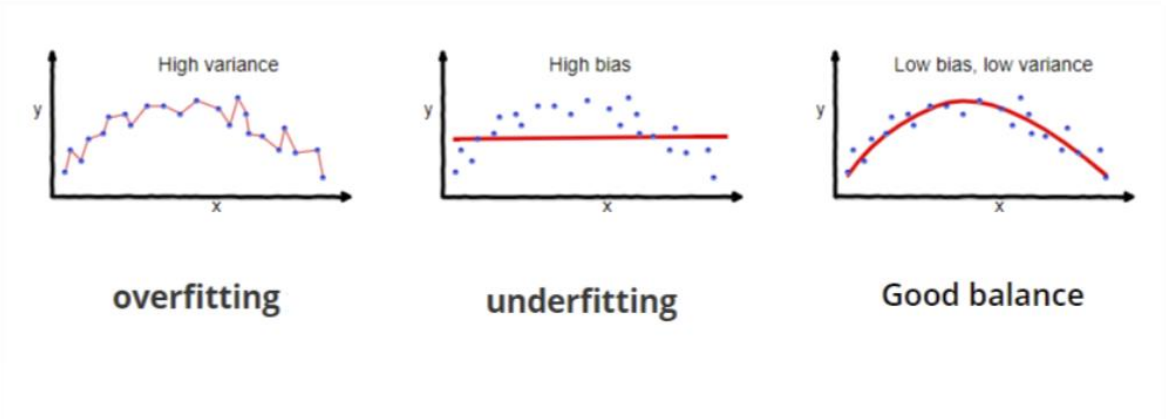
Bias and Variance in Machine Learning

Maintaining a suitable equilibrium between bias and variance, known as the Bias-Variance Tradeoff, can assist in preventing the model from becoming overfit to the training data.

Regularization in Machine Learning

Regularization is a method utilized to minimize errors by adjusting the function effectively on the provided training set and preventing overfitting. The most commonly employed regularization methods include:

1. Lasso Regularization - L1 Regularization
2. Ridge Regularization - L2 Regularization
3. Elastic Net Regularization - L1 and L2 Regularization



Regularized model to avoid underfitting as well as overfitting

Lasso Regression

LASSO (Least Absolute Shrinkage and Selection Operator) regression is a regression model that utilizes the L1 Regularization technique. This technique introduces a penalty term to the loss function (L) by adding the absolute value of the coefficient. By penalizing the weights to approach zero, Lasso Regression facilitates feature selection by ensuring that irrelevant features do not impact the model significantly.

where,

- ***m*** *– Number of Features*
- ***n*** *– Number of Examples*
- ***y_i*** *– Actual Target Value*
- ***y_i(hat)*** *– Predicted Target Value*

Ridge Regression

Ridge regression is a regression model that implements L2 regularization. It includes a penalty term in the loss function(L) which is based on the squared magnitude of the coefficients.

$$\text{Cost} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda \sum_{i=1}^{m} w_i^2$$

Elastic Net Regression

The model integrates both L1 and L2 regularization techniques, incorporating the absolute norm and squared measure of the weights. This is achieved by introducing an additional hyperparameter to adjust the balance between L1 and L2 regularization.

$$\text{Cost} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda \left((1-\alpha)\sum_{i=1}^{m}|w_i| + \alpha \sum_{i=1}^{m} w_i^2\right)$$

What is Ensemble Learning?

Ensemble learning is a method in machine learning that brings together multiple individual models to build a more robust and accurate predictive model. By leveraging the unique strengths of different models, ensemble learning strives to reduce errors, improve performance, and enhance the overall reliability of predictions. This approach leads to better outcomes in various machine learning and data analysis tasks.

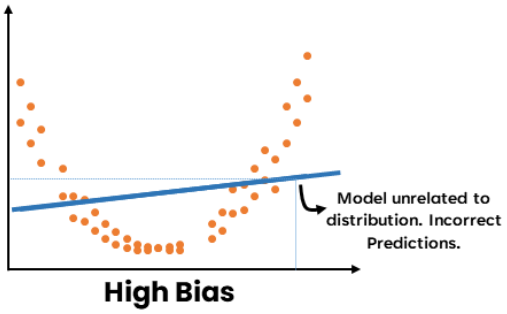
How Ensemble Learning Works?

Ensemble learning is a learning approach that involves merging multiple machine learning models. One common issue in machine learning is the subpar performance of individual models, resulting in low prediction accuracy. To address this challenge, we combine multiple models to create a more robust and accurate one.

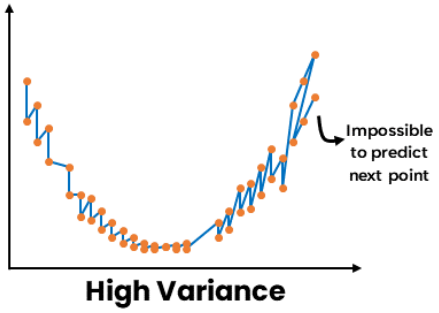
The models we merge are referred to as weak learners, as they typically exhibit either high bias or high variance. Due to their limitations in efficient learning and performance, weak learners are unable to excel on their own.

High-bias and High-variance Models

A high-bias model arises from insufficiently learning the data, rather than being dependent on the data distribution. As a result, subsequent predictions may not be aligned with the data, leading to inaccuracies.



A model with high variance occurs when the data is overfit. This leads to significant fluctuations with each data point, making accurate prediction of the next point impossible.

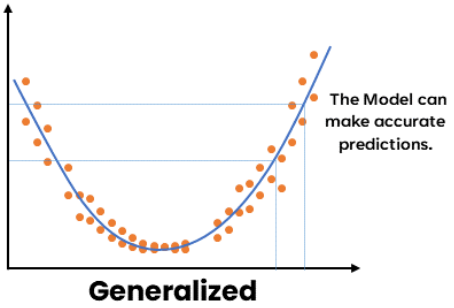


Models with high bias or high variance struggle to generalize effectively, leading weak learners to make inaccurate generalizations or fail to generalize at all. Consequently, relying solely on the predictions of weak learners is not advisable.

According to the bias-variance trade-off, an underfit model exhibits high bias and low variance, while an overfit model shows high variance and low bias. In both scenarios, there is an imbalance between bias and variance. Achieving a balance requires keeping both bias and variance low.

Ensemble learning aims to address this trade-off by reducing either bias or variance, ultimately striving for a balanced model.

Ensemble learning aims to address bias in weak models with high bias and low variance, as well as reduce variance in weak models with high variance and low bias. By combining multiple models, ensemble learning creates a more balanced, strong learner with low bias and variance. This results in a more generalized model that can make accurate predictions.



Monitoring Ensemble Learning Models

Ensemble learning enhances a model's effectiveness through three key mechanisms:

- Minimizing the variance of weak learners
- Minimizing the bias of weak learners
- Enhancing the overall accuracy of strong learners

Bagging is applied to decrease the variance of weak learners, while boosting is utilized to decrease their bias. Stacking, on the other hand, is employed to enhance the overall accuracy of strong learners.

Reducing Variance with Bagging

We employ bagging to aggregate weak learners with high variance. The goal of bagging is to create a model with reduced variance compared to the individual weak models. These weak learners are homogeneous, indicating they are of the same type.

Bagging is also referred to as Bootstrap aggregating and involves two main steps: bootstrapping and aggregation.

Bootstrapping

The process consists of sampling subsets of data with replacement from an original dataset. These subsets, referred to as bootstrapped datasets or simply bootstraps, are derived from the initial dataset. Sampling 'with replacement' allows for individual data points to be selected multiple times. Each bootstrap dataset serves as training data for a weak learner.

Aggregating

The weak learners are trained in isolation, with each learner generating independent predictions. These individual predictions are then combined at the end

to provide an overall prediction. The aggregation process can involve either max voting or averaging methods.

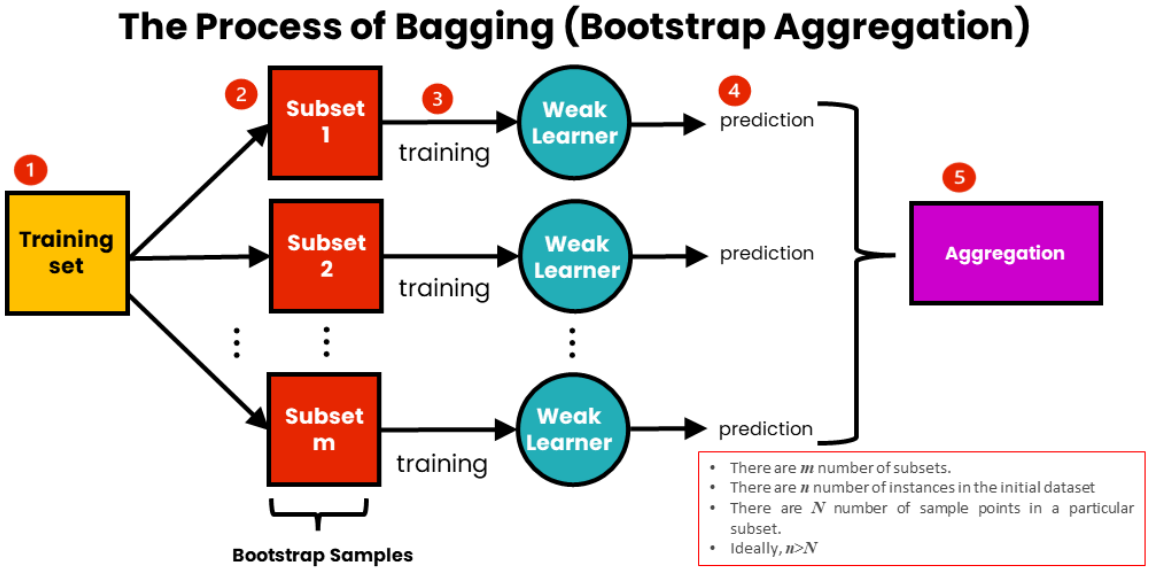
Max Voting

In classification problems, a commonly used method involves taking the mode of the predictions, which is the most frequently occurring prediction. This approach is known as voting, drawing a parallel with the idea of majority rule in elections. Each model generates a prediction, with each prediction counting as a single 'vote'. The prediction that occurs most frequently is selected as the representative for the combined model.

Averaging

The method is commonly applied in regression tasks, where it calculates the mean of the predictions to generate an overall prediction for the combined model.

Steps of Bagging



The steps of bagging are as follows:

1. An initial training dataset with n instances is available.
2. Multiple subsets of data are created from the training set, with each subset comprising N sample points sampled with replacement from the initial dataset.
3. The corresponding weak learners are trained independently for each subset of data. These models are homogeneous in nature, being of the same type.
4. Each model generates a prediction.
5. The predictions from all models are combined into a single prediction using either max voting or averaging.

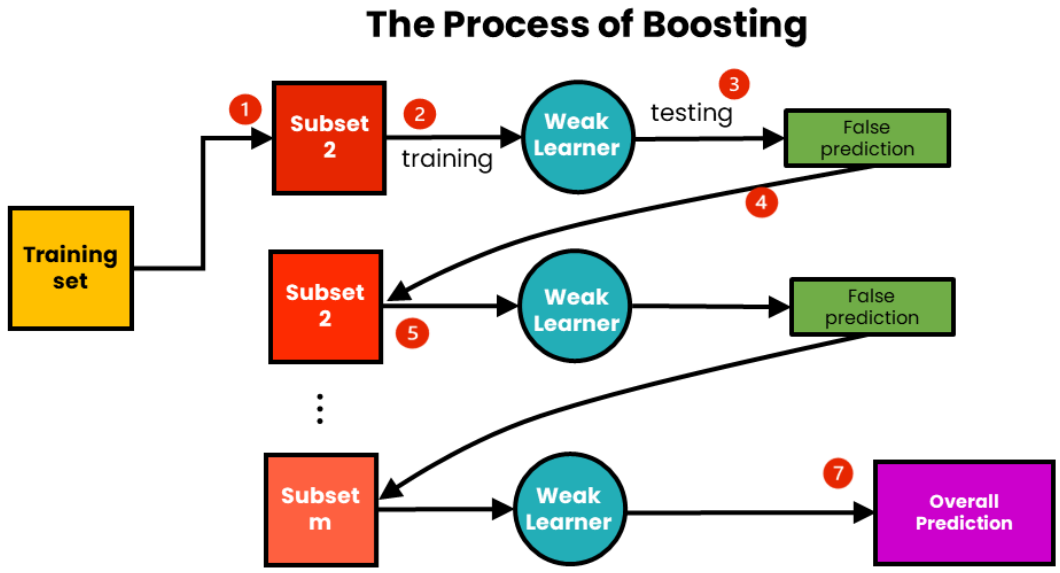
Reducing Bias by Boosting

In our approach, we utilize boosting to combine weak learners with high bias. The goal of boosting is to create a model with lower bias compared to the individual models. Similar to bagging, the weak learners are of a homogeneous nature.

Boosting involves training weak learners in a sequential manner. With each subsequent learner, the goal is to improve upon the errors of the previous learners in the sequence. Initially, a sample of data is extracted from the original dataset. This sample is used to train the first model, which then makes predictions. These predictions can be either correct or incorrect. The incorrectly predicted samples are then utilized for training the next model, allowing subsequent models to build upon the errors of their predecessors.

In contrast to bagging, where prediction results are aggregated at the end, boosting aggregates results after each step through weighted averaging. Weighted averaging assigns varying weights to models based on their predictive capabilities, giving higher weight to the model with the greatest predictive power. This prioritizes the learner with the highest predictive power as the most significant contributor to the overall outcome.

Steps of Boosting



Boosting works with the following steps:

1. **Sampling Subsets:** A set of m subsets is sampled from the initial training dataset.
2. **Training Weak Learners:** The first subset is used to train the initial weak learner.
3. **Testing Weak Learner:** The trained weak learner is tested using the training data, leading to incorrect predictions for some data points.
4. **Updating Subset**: Data points with incorrect predictions are sent to the second subset for updating.
5. **Training Second Weak Learner:** The updated subset is used to train and test the second weak learner.
6. **Continuation:** This process continues with each subsequent subset until all subsets have been processed.

7. **Aggregated Prediction:** The final prediction is derived from the aggregation of predictions made at each step, eliminating the need for further calculations.

Improving Model Accuracy with Stacking

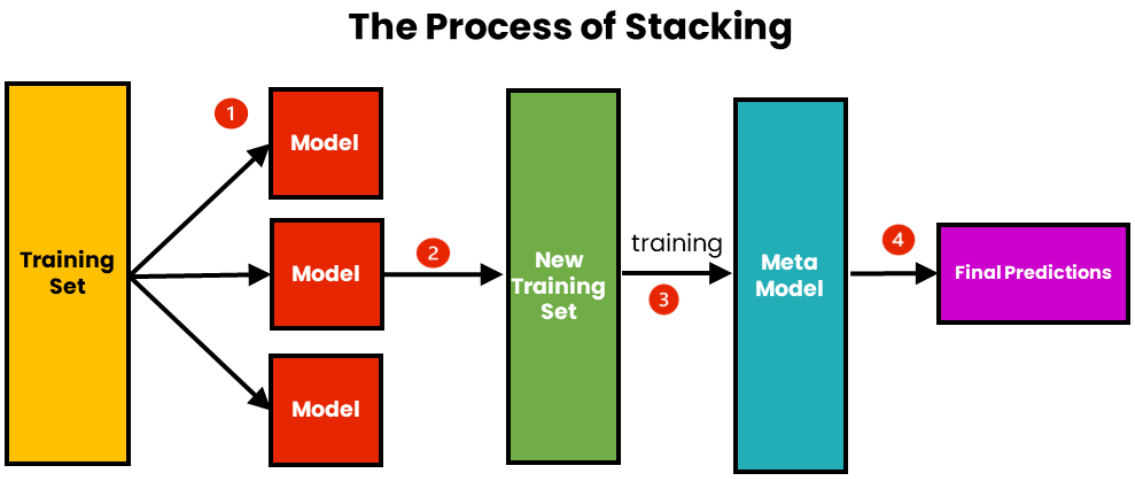
Utilizing stacking techniques enhances the predictive accuracy of powerful learners. The goal of stacking is to combine multiple diverse powerful learners into a single robust model.

Stacking sets itself apart from bagging and boosting by:

1. Collaborating strong learners
2. Combining heterogeneous models
3. Developing a Metamodel: A metamodel refers to a model generated using a fresh dataset.

In the process of stacking, diverse individual models are initially trained using an initial dataset. These models generate predictions, which are then aggregated to create a cohesive new dataset. Subsequently, this new dataset is utilized to train the metamodel, which ultimately yields the final prediction. The final prediction is achieved through a weighted averaging technique. Stacking is particularly advantageous as it can amalgamate both bagged and boosted models, thereby combining strong learners for improved predictive accuracy.

Steps of Stacking



The steps of Stacking are as follows:

1. Training data is used initially to train multiple algorithms.
2. The output of each algorithm is utilized to generate a new training set.
3. A meta-model algorithm is developed using the new training set.
4. Final predictions are made based on the results of the meta-model, with the results being combined through weighted averaging.

When to use Bagging vs Boosting vs Stacking?

	Bagging	Boosting	Stacking
Purpose	Reduce Variance	Reduce Bias	Improve Accuracy
Base Learner Types	Homogeneous	Homogeneous	Heterogeneous
Base Learner Training	Parallel	Sequential	Meta Model
Aggregation	Max Voting, Averaging	Weighted Averaging	Weighted Averaging

To decrease overfitting or variance in your model, leverage bagging; to lessen underfitting or bias, utilize boosting. For increased predictive accuracy, opt for stacking. Bagging and boosting are effective with homogeneous weak learners, while stacking operates with heterogeneous strong learners. All three methods can be applied to both classification and regression tasks.

One drawback of boosting is its susceptibility to variance and overfitting, making it less effective for reducing variance. In comparison, bagging tends to do a better job at reducing variance than boosting. Conversely, bagging is not recommended for reducing bias or underfitting, as it is more prone to bias and does not effectively address bias.

Stacked models offer improved prediction accuracy compared to bagging or boosting methods, yet they require a significantly greater amount of time and computational resources due to the combination of multiple models. If efficiency is a priority, it is recommended to avoid stacking. However, for those seeking maximum accuracy, stacking is the preferred approach.

Conclusion:

Ensemble learning techniques such as bagging, boosting, and stacking play a crucial role in enhancing the accuracy of machine learning models and mitigating the risks associated with inaccuracies. The main points discussed in the article include:

- Ensemble learning involves merging multiple machine learning models to create a more robust and accurate model.
- Bagging focuses on reducing variance and boosting target bias reduction, and stacking aims to enhance prediction accuracy.
- Bagging and boosting utilize similar weak learners, while stacking combines diverse strong learners.
- Bagging trains models concurrently, boosting trains models sequentially, and stacking establishes a meta-model.

Early Stopping

Early stopping is a regularization technique utilized during the training of models with iterative methods like Gradient Descent. Given that neural networks rely heavily on optimization algorithms such as gradient descent for learning, early stopping can be applied to all problems. This method helps prevent overfitting by continuously updating the model to improve its fit with the training data at each iteration. It is well recognized that excessive training in neural networks can lead to overfitting on the training data.

The model's performance on the test set has shown improvement up to a specific threshold. Prior to this threshold, enhancing the model's alignment with the training data results in a rise in generalization error. This regularization technique offers insight into the optimal number of iterations to prevent overfitting.

This technique is shown in the below diagram.

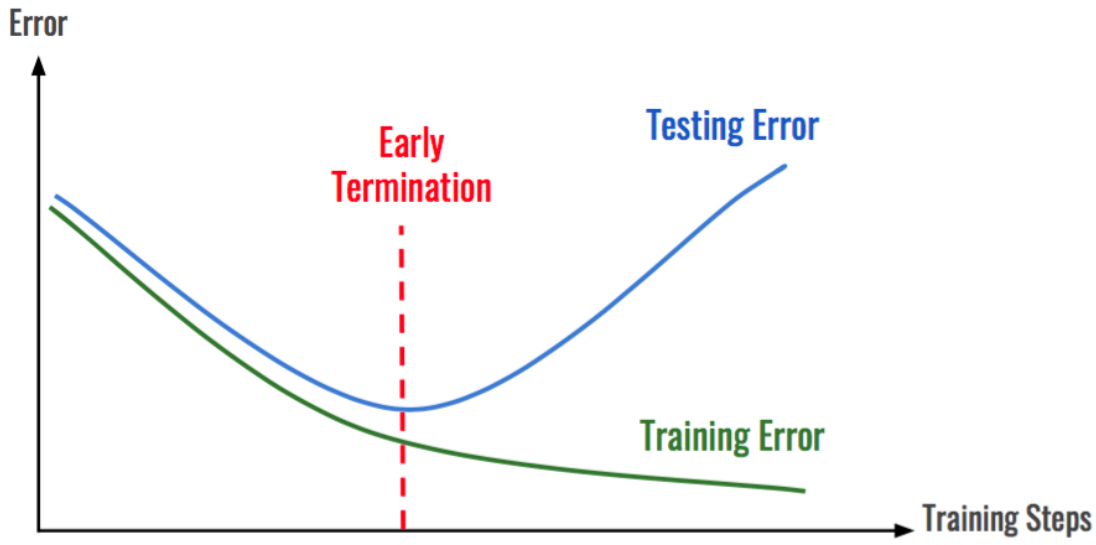


Image Source: Google Images

After multiple iterations, it is apparent that the test error is beginning to rise while the training error continues to decrease, indicating the model is overfitting. To address this issue, we halt the model when this trend emerges.

The network parameters at the time of early termination are deemed optimal for the model. To further reduce the test error post-early termination, the following strategies can be employed:

- Consider reducing the learning rate by implementing a learning rate scheduler algorithm.
- Utilize an alternative Optimization Algorithm.
- Implement weight regularization techniques such as L1 or L2 regularization for improved performance.

Unsupervised Pre-training

Unsupervised pre-training is a method in machine learning that utilizes unlabeled data to develop an initial model, which can later be refined with a limited amount of labeled data. This strategy proves valuable in situations where obtaining labeled data is limited or costly.

Definition

Unsupervised pre-training is a machine learning technique that leverages unlabeled data to build an initial model, which can then be fine-tuned with a small amount of labeled data. This approach is particularly beneficial in scenarios where acquiring labeled data is restricted or expensive.

During the fine-tuning process, the pre-trained model undergoes additional training using a reduced set of labeled data. This allows the model to optimize its parameters for the specific task at hand by utilizing the insights acquired during the initial pre-training phase. Notably, this approach typically results in faster training times and necessitates fewer labeled data compared to training a model from the ground up.

Importance

Unsupervised pre-training is a valuable asset for data scientists, particularly when faced with extensive amounts of unlabeled data. This technique enables the identification of intricate patterns and features within the data, ultimately enhancing the efficiency of subsequent tasks.

Additionally, unsupervised pre-training helps alleviate the difficulties associated with acquiring labeled data, including the time and resources required for manual labeling. By utilizing unlabeled data, it facilitates the creation of resilient models even in instances where labeled data is scarce.

Use Cases

Unsupervised pre-training has proven to be effective across various domains:

- Natural Language Processing (NLP): Unsupervised pre-training has been utilized in NLP to advance models such as BERT and GPT, which are pretrained on extensive text corpora and subsequently fine-tuned for tasks like sentiment analysis and question answering.
- Computer Vision: Unsupervised pre-training is instrumental in enhancing the performance of tasks such as image classification and object detection in computer vision, by pre-training models on large sets of unlabeled images.
- Reinforcement Learning: In reinforcement learning, unsupervised pre-training is valuable for acquiring a robust environmental representation, which can then be refined using reward signals.

Limitations

Although unsupervised pre-training brings numerous advantages, it also poses certain limitations. The computational demands of the pre-training phase can be extensive, necessitating substantial resources and time. Furthermore, the effectiveness of the pre-training greatly impacts the outcomes of the subsequent fine-tuning phase. If the pre-training fails to capture relevant features, the fine-tuning process may not result in substantial enhancements.

Notwithstanding these obstacles, unsupervised pre-training remains a valuable strategy for harnessing unlabeled data and enhancing model performance across various tasks and domains.

UNIT - IV RECURRENT NEURAL NETWORKS

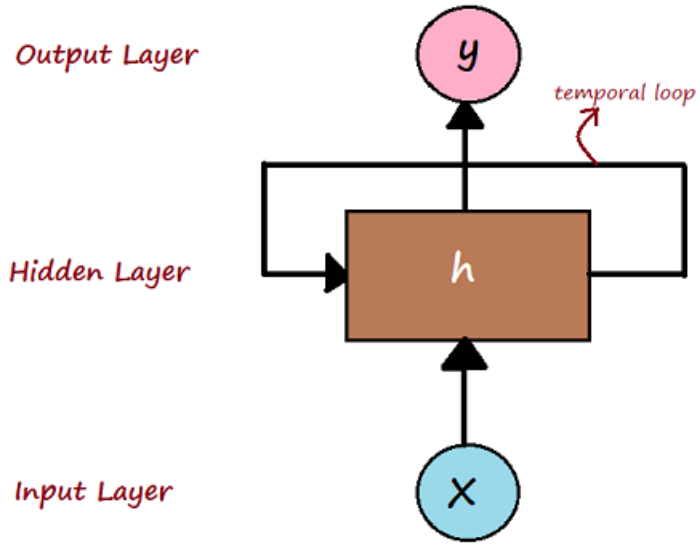
What is this RNN ?

The recurrent neural network (RNN) is a form of deep learning architecture commonly utilized for predicting future sequences. What sets RNN apart from other learning structures? The key distinction lies in its ability to retain memory. Unlike other deep learning models, RNNs have the capacity to remember past inputs.

Moreover, while traditional neural networks treat each input as independent, RNNs establish connections between inputs to infer the subsequent step. This relational approach enables RNNs to comprehend and predict future outcomes based on the interdependencies among inputs.

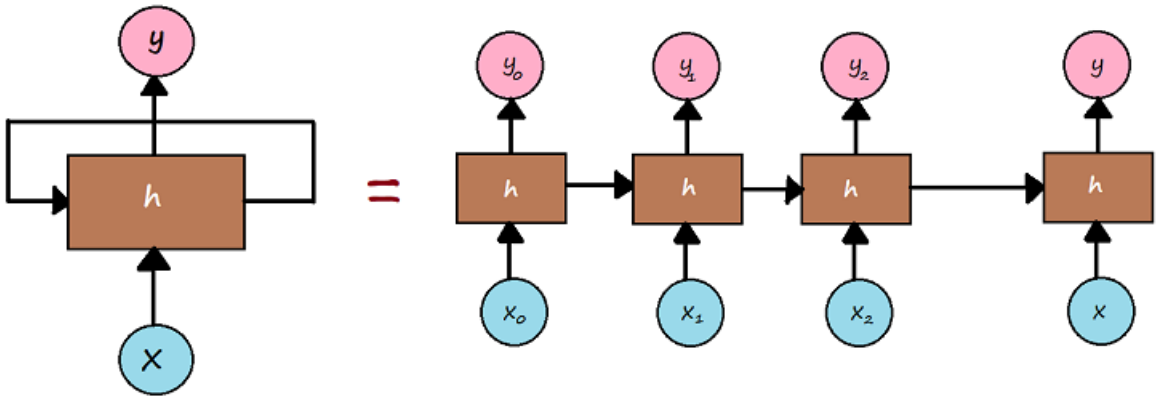
In our previous discussion, we mentioned that RNNs have the ability to remember information. But how exactly does this process occur?

RNNs achieve memory retention through a loop-like structure that enables them to continuously refer back to previous information, thus solidifying the relationships they have formed. To delve deeper into this concept, let's examine a visual representation of the RNN structure.



- Upon reviewing the image above, it is evident that there are nodes present in the hidden layer. This can be observed in the preceding image.
- These nodes possess temporal cycles that sustain them. This loop allows the input value to be retained in their memory even after an output is provided (Short memory).

To understand this better, let's open this picture a little more.



- **As we understand from the picture. For example, y0 obtained in the previous step and x1, our current input, are used to output y1.**

Why Recurrent Neural Networks

The image above shows a forward neural network that is limited in its capabilities:

- Not capable of handling sequential data
- Only consider the current input, ignoring past inputs and a lack of memory retention.

To address these limitations, Recurrent Neural Networks (RNN) offer a solution. RNNs excel at processing sequential data, taking into account both current and past inputs. With the ability to remember previous entries due to internal memory, RNNs provide a more comprehensive approach to data analysis.

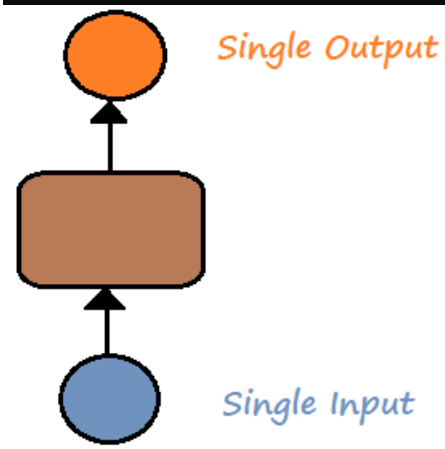
What are the application areas of RNN

- **Image Captioning**
- **Time Series Prediction**
- **Natural Language Processing**
- **Machine Translation**

What are the types of RNN

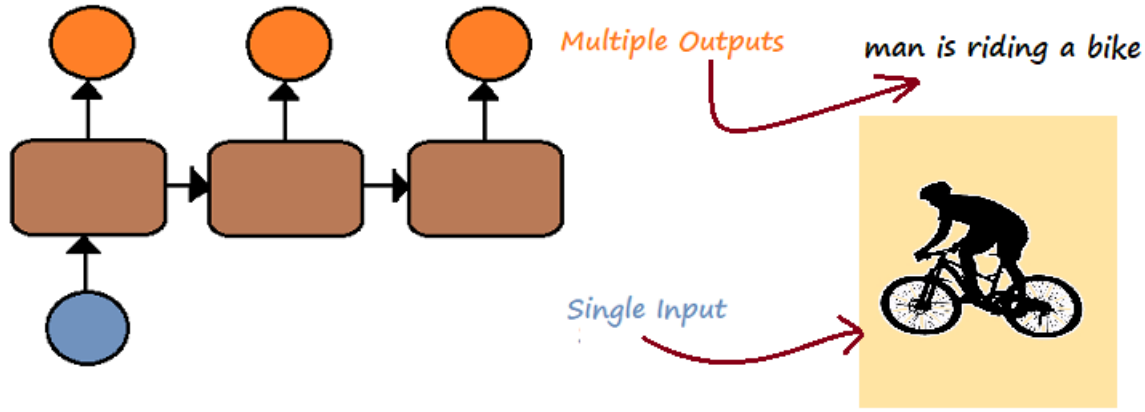
1. **One to One**
2. **One to Many**
3. **Many to One**
4. **Many to Many**

One to One !?



This type of neural network can be called the vanilla neural network. It has a single entrance and a single exit. It is often used for machine learning problems.

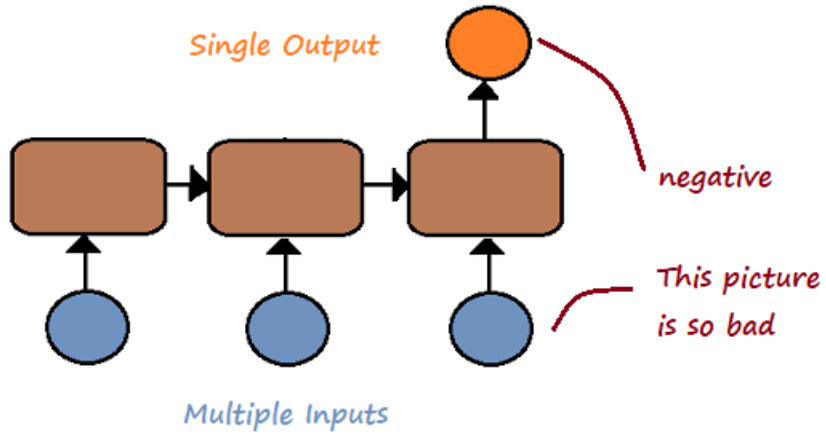
One to Many !?



this neural network has one input and multiple outputs.

Example : An example is the picture and the sentence that describes it. A picture is input and a sentence describing it comes out.

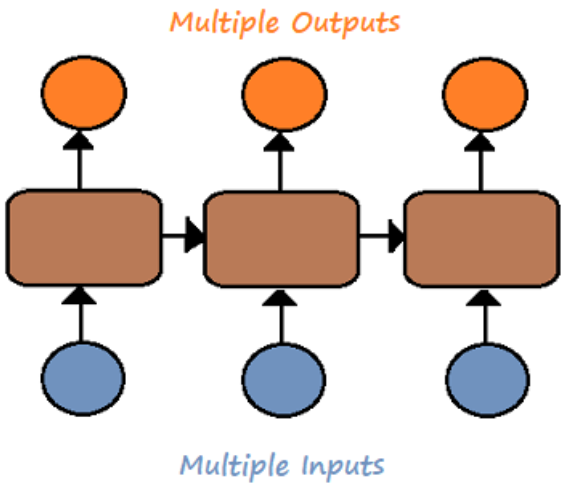
Many to One !?



This RNN takes multiple inputs and yields a single output.

Example: Suppose it takes a sentence as input. It can also give the emotion of this sentence as output. as in the picture above.

Many to Many !?



It is a type of RNN that gives many inputs and many outputs.

Example : An example is machine translation from one language to another.

Benefits of RNN

Benefits of Recurrent Neural Networks:

- Handles sequential data: Processes data in sequence
- Memorizes and stores past results: Retains previous information for future computations
- Considers current and previous results: Incorporates both current and past data for generating new output
- Maintains fixed model size: Does not increase in size with larger inputs
- Shares weights across time: Utilizes shared weights for efficient computation across different time units

Limitations of RNN

Below are some of the limitations of Recurrent Neural Networks:

1. Computation time slows down due to recurrence.
2. Inability to handle lengthy information sequences when utilizing tanh or ReLU activation functions.
3. Future data cannot be considered in the computation of current data.
4. Training process is complex.

Concern of Exploding Gradient: The growth of model weights is driven by the accumulation of significant gradient errors, which exhibit exponential growth over time.

Issue of Vanishing Gradient: The gradients diminish to a point where they no longer cause substantial adjustments in the model weights.

What is a Gradient?

The Gradient can be defined as the derivative of the loss function with respect to the weights. It is crucial in updating the weights to effectively minimize the loss function while conducting backpropagation in neural networks.

Vanishing Gradients?

The issue of Vanishing Gradient is observed when the derivative or slope decreases with each layer during backpropagation. This can result in significantly longer training times or even halt the training process altogether when the weights are updated very minimally.

Vanishing Gradient commonly arises when using the sigmoid and tan h activation functions, as their derivatives fall within the range of 0 to 0.25 and 0 to 1, respectively. Consequently, the updated weight values become exceedingly small, leading to minimal variation between old and new weight values and exacerbating the Vanishing Gradient problem.

To mitigate this issue, one can opt for the ReLU activation function, where the gradient remains at 0 for negative and zero inputs, and at 1 for positive inputs. This can help prevent the occurrence of Vanishing Gradient and facilitate smoother training of neural networks.

Exploding Gradients?

The occurrence of exploding gradient arises when the derivatives grow significantly as we move backwards through each layer in backpropagation. This is in stark contrast to vanishing gradients. The source of this issue is the weights, not the activation function. High weight values lead to increased derivatives, causing drastic differences between

new and old weights, preventing the gradient from converging. Consequently, the model may oscillate around minima without reaching a global minimum point.

LSTMs

Recurrent neural networks typically have a limited short-term memory, unless using Long Short-Term Memory (LSTM) cells. LSTMs utilize a 3-gate module for memory management:

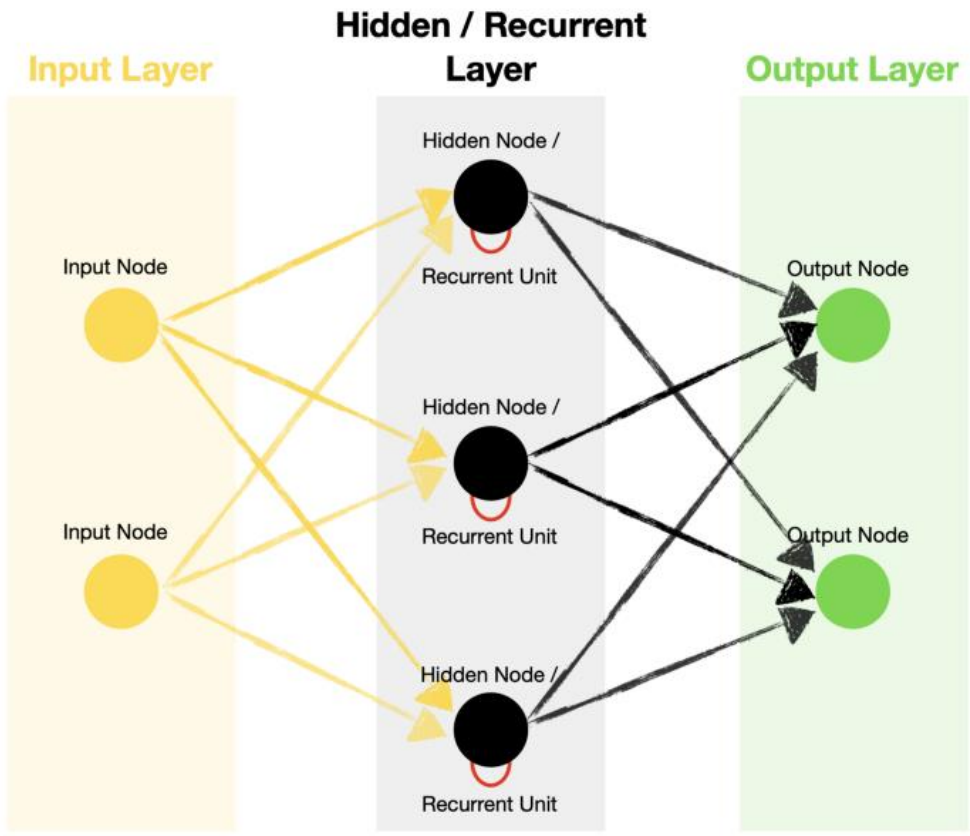
Forget gate: Determines the amount of past information to retain and discard.
Input gate: Controls how much of the current input is integrated into the current state.
Output gate: Regulates the amount of the current state that is transmitted to the output.

Intro

Standard Recurrent Neural Networks (RNNs) are hampered by short-term memory issues, which stem from a vanishing gradient problem arising when processing extended data sequences. Fortunately, there exist more sophisticated iterations of RNNs capable of retaining crucial information from earlier segments of the sequence and propelling it forward. Two prominent options include Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU).

What makes LSTM different from standard RNNs and how does LSTM work?

To begin, we will provide a brief overview of a basic RNN architecture. An RNN is comprised of various layers akin to a Feed-Forward Neural Network, including the input layer, hidden layer(s), and output layer.

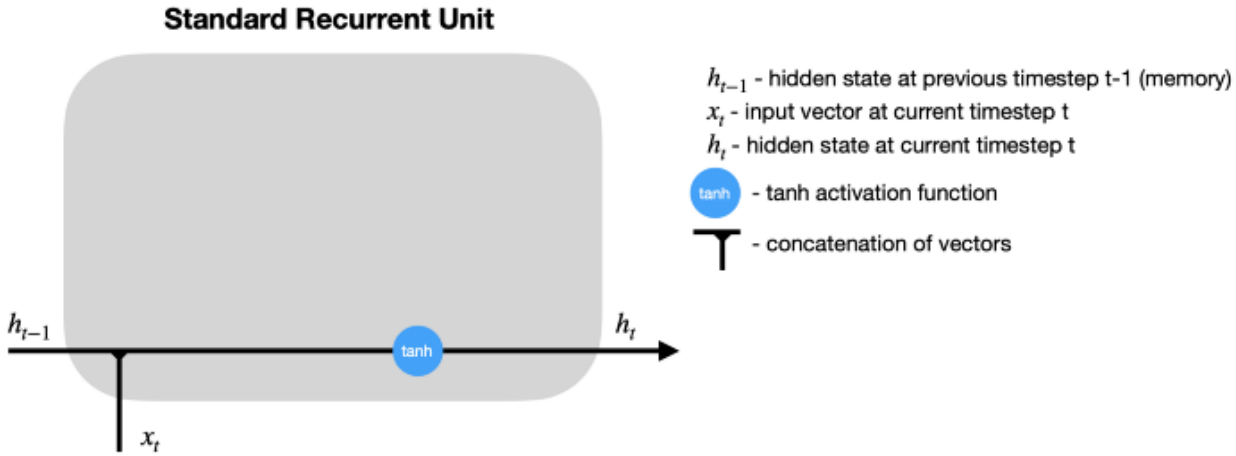


Standard Recurrent Neural Network architecture

The Recurrent Neural Network (RNN) comprises recurrent units within its hidden layer, enabling it to handle sequential data. This is achieved by iteratively transmitting a hidden state from the previous time step and integrating it with the current input. Each individual iteration of input processing in an RNN is referred to as a time step, with the total number of time steps corresponding to the length of the sequence being processed.

How does LSTM differ from standard RNN?

RNNs and LSTMs both rely on recurrent units to analyze sequence data, although their internal mechanisms differ significantly. While RNNs and LSTMs both involve combining the previous hidden state with new input and passing it through an activation function, the specific operations within the recurrent unit vary between the two models. In a standard RNN, for example, the unit diagram simplifies these operations to just two major steps.



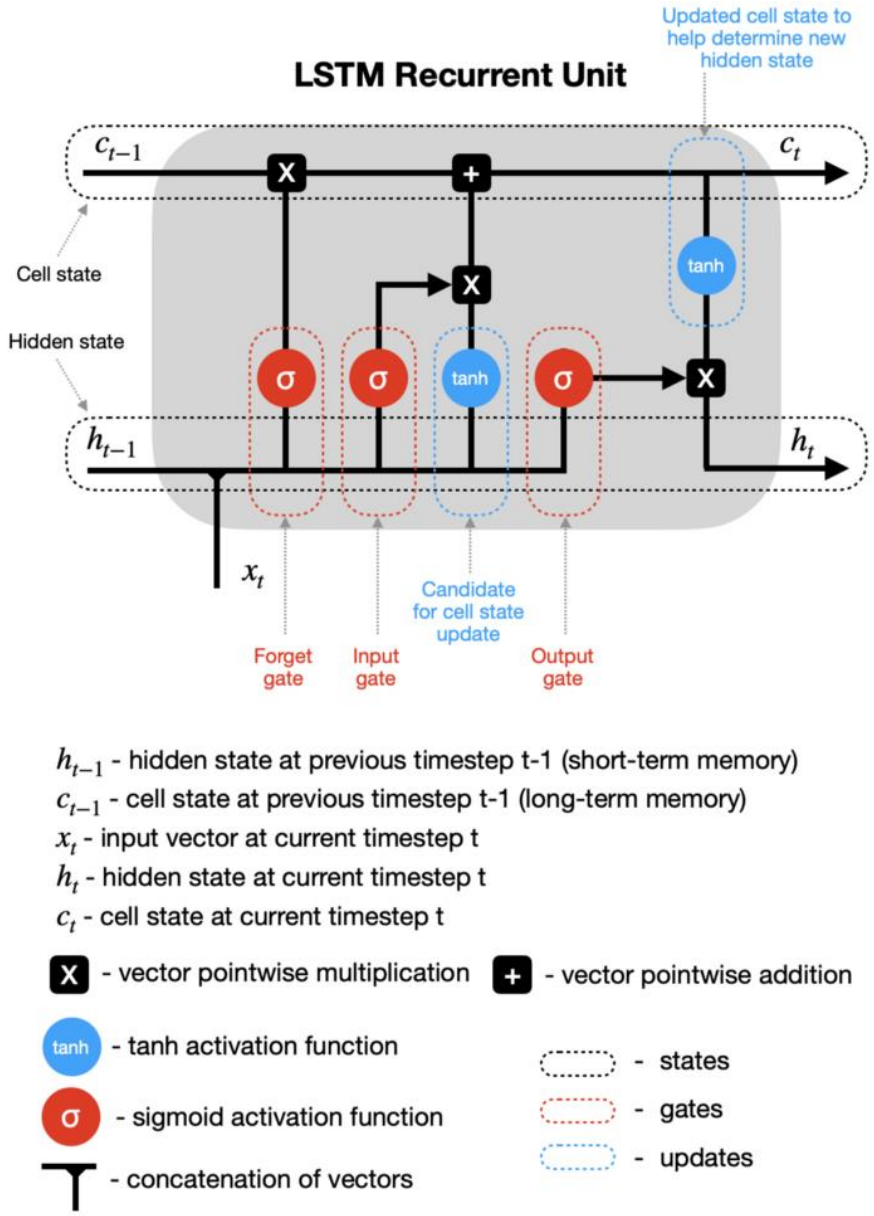
Standard RNN recurrent unit.

After calculating the hidden state at timestep t, it is sent back to the recurrent unit and combined with the input at timestep t+1 to determine the new hidden state at timestep t+1. This iterative process continues for t+2, t+3, and so on until the specified number (n) of timesteps is achieved.

In contrast, LSTM utilizes different gates to determine which information should be retained or discarded. Additionally, it incorporates a cell state, serving as a form of long-term memory within the LSTM structure. Let us delve further into these concepts.

How does LSTM work?

LSTM recurrent unit is much more complex than that of RNN, which improves learning but requires more computational resources.



Let's review the simplified diagram (weights and biases excluded) to understand the processing of information by the LSTM recurrent unit.

1. **Combination of Hidden State and New Inputs:** The hidden state from the previous timestep (h_t-1) and the input at the current timestep (x_t) are merged before being passed through various gates.
2. **Control Mechanism of Forget Gate:** The forget gate determines which information should be retained or discarded. Through the sigmoid function, values in the cell state are either discarded (multiplied by 0), stored (multiplied by 1), or partially retained (multiplied by a value between 0 and 1).
3. **Identification of Important Elements by Input Gate:** The input gate assists in identifying crucial elements that need to be incorporated into the cell state. Only the

information deemed important by the input gate is added to the cell state after being multiplied by the cell state candidate.

4. **Cell State Update Process:** Initially, the previous cell state (c_t-1) is multiplied by the output of the forget gate. New information from [input gate × cell state candidate] is then added to obtain the updated cell state (c_t).
5. **Updating Hidden State:** The final step involves updating the hidden state by passing the latest cell state (c_t) through the tanh activation function and then multiplying it by the output gate results.

The latest cell state (c_t) and hidden state (h_t) are fed back into the recurrent unit, repeating the process at timestep t+1. This loop persists until the sequence is exhausted.

Gated Recurrent Unit (GRU) is a type of recurrent neural network (RNN) developed by Cho et al. in 2014 as a more simplified alternative to Long Short-Term Memory (LSTM) networks. Similar to LSTM, GRU is capable of handling sequential data such as text, speech, and time-series information.

GRU employs gating mechanisms to selectively update the hidden state of the network at each time step. These gating mechanisms regulate the flow of information within the network. The GRU includes two key gating mechanisms known as the reset gate and the update gate.

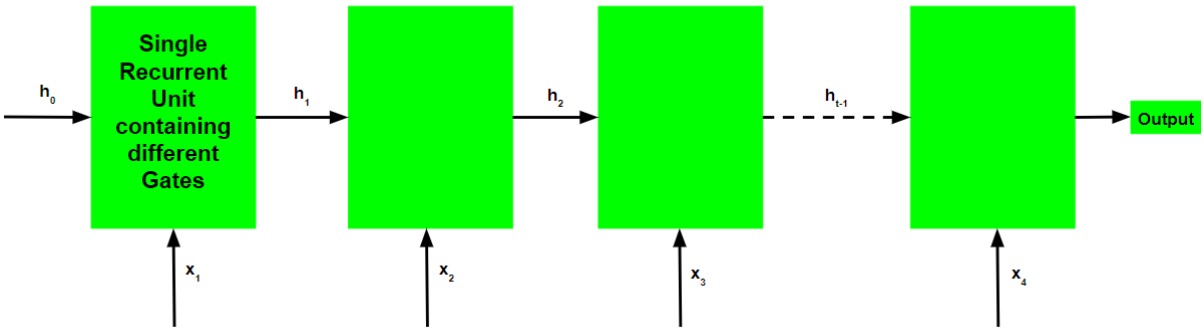
The reset gate controls the amount of the prior hidden state to be discarded, and the update gate determines the extent to which new input should modify the hidden state. The GRU output is computed based on the adjusted hidden state. The equations for calculating the reset gate, update gate, and hidden state in a GRU are as follows:
Reset gate: ***r_t = sigmoid(W_r * [h_{t-1}, x_t])***
Update gate: ***z_t = sigmoid(W_z * [h_{t-1}, x_t])***
Candidate hidden state: ***h_t' = tanh(W_h * [r_t * h_{t-1}, x_t])***
Hidden state: ***h_t = (1 – z_t) * h_{t-1} + z_t * h_t'***
where W_r, W_z, and W_h are learnable weight matrices, x_t is the input at time step t, h_{t-1} is the previous hidden state, and h_t is the current hidden state.

To summarize, GRU networks are a type of Recurrent Neural Network (RNN) that utilize gating mechanisms to selectively update the hidden state at each time step, enabling them to effectively model sequential data. They have demonstrated effectiveness in various natural language processing tasks such as language modeling, machine translation, and speech recognition.
In contrast to LSTM, GRU networks consist of only three gates and do not maintain an Internal Cell State. Instead, the information typically stored in the Internal Cell State of an

LSTM unit is incorporated into the hidden state of the GRU. This amalgamated information is then passed on to the next GRU unit. The different gates of a GRU include the following:

1. **Update Gate (z):** This gate determines the amount of relevant past knowledge to carry forward into the future. It can be compared to the Output Gate of an LSTM recurrent unit.
2. **Reset Gate (r):** This gate decides which past information to discard. It is similar to the combination of the Input and Forget Gates in an LSTM recurrent unit.
3. **Current Memory Gate:** This aspect is often overlooked in discussions about Gated Recurrent Unit Networks. It is integrated into the Reset Gate, serving as a means to introduce non-linearity and Zero-mean into the input. By incorporating it into the Reset Gate, the impact of previous information on current data being transferred forward is minimized.

The foundational structure of a Gated Recurrent Unit (GRU) Network closely resembles that of a standard Recurrent Neural Network (RNN) when visually depicted. The primary distinction lies in the internal mechanisms of each recurrent unit. GRU networks are equipped with gated components that regulate the flow of information between the current input and the preceding hidden state.



Applications of RNN Networks:

1. Machine Translation:

RNNs have the capability to create a sophisticated deep learning system for automatically translating text between different languages, eliminating the need for manual intervention. For instance, it is possible to easily translate text from your native language to English.

2. Text Creation:

Recurrent Neural Networks (RNNs) have the capability to construct a deep learning framework for generating text. By analyzing the preceding sequence of words/characters in a text, a trained model can understand the probability of a word/character occurring. Models can be trained at various levels such as character, n-gram, sentence, or paragraph.

3. Captioning of images:

The act of generating text that explains the contents of an image is referred to as image captioning. This caption can depict both the object and the action taking place within the image. In the given example, a proficient deep learning model utilizing RNN is able to describe the image as "A woman wearing a green coat is reading a book under a tree".

4. Recognition of Speech:

This technology, also referred to as Automatic Speech Recognition (ASR), has the ability to transcribe human speech into written or text form. It's important to differentiate between speech recognition and voice recognition: the former concentrates on converting spoken words into text, while the latter identifies the user based on their voice.

Commonly utilized speech recognition tools include Alexa, Cortana, Google Assistant, and Siri.

5. Forecasting of Time Series:

An RNN can be trained on historical time-stamped data to develop a time series prediction model, enabling it to forecast future outcomes, as exemplified by its application in the stock market.

Stock market data can be utilized to develop a machine learning model capable of predicting future stock prices by analyzing past data. This can support investors in making informed investment choices.

Application to Automatic Image Captioning

In image captioning, training data comprises pairs of images and captions. For instance, the image displayed on the left side of Figure 7.9 was sourced from the National Aeronautics and Space Administration website and captioned as "cosmic winter wonderland." Multiple image-caption pairs like this can be utilized for training the weights within a neural network. Subsequently, once training is finished, the network can predict captions for unfamiliar test cases. This methodology can be perceived as an example of image-to-sequence learning.

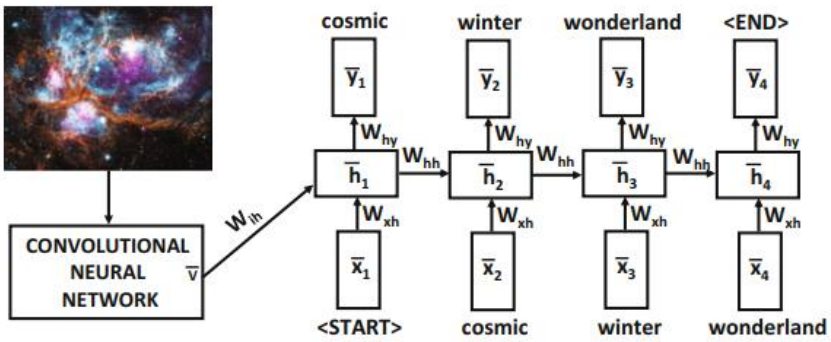


Figure 7.9: Example of image captioning with a recurrent neural network. An additional convolutional neural network is required for representational learning of the images. The image is represented by the vector $\overline{v}$, which is the output of the convolutional neural network. The inset image is by courtesy of the National Aeronautics and Space Administration (NASA).

When automatically captioning images, a challenge arises in that a distinct neural network is necessary to grasp the representation of the images. The convolutional neural network is a prevalent architecture utilized to understand image representations.

Sequence-to-Sequence Learning and Machine Translation

By combining a convolutional neural network and a recurrent neural network, image captioning can be performed. Similarly, two recurrent networks can be integrated to translate text from one language to another. This approach is known as sequence-to-sequence learning, where a sequence in one language is translated into a corresponding sequence in another language. Sequence-to-sequence learning has potential applications beyond machine translation, including question-answering systems.

In the discussion that follows, we offer a straightforward solution for machine translation using recurrent neural networks. While traditional RNNs are not commonly utilized for this specific task, we instead employ a modified version known as the Long Short-Term Memory (LSTM) model. The LSTM model excels in capturing long-term relationships, making it particularly effective for processing longer sentences. Given that the principles of RNNs also apply to LSTMs, we will proceed to explore machine translation with the (simpler) RNN.

In the machine translation application, a combination of two different RNNs is utilized, similar to how a convolutional neural network and a recurrent neural network are combined for image captioning. The first RNN receives input from the source language and gradually accumulates knowledge about the source sentence in its hidden state over successive time stamps. Once the end-of-sentence symbol is reached, the second RNN begins by outputting the first word of the target language. Subsequent states in the second RNN output each word of the target language one by one, using words of the target

language as input. During training instances, the model has access to the actual target language words, but during test instances, predicted values are used instead.

Application to Sentence-Level Classification

When analyzing this issue, every sentence serves as a training or test sample for classification tasks. Sentence-level classification is commonly considered more challenging than document-level classification due to the brevity of sentences and the limited evidence in vector space representations. Nevertheless, the sequence-centric perspective is a potent tool that can enhance the accuracy of classification outcomes.

Sentence-level classification in sentiment analysis is commonly utilized to determine the sentiment of users towards certain topics based on the content of individual sentences [6]. An example of this is determining if a sentence conveys a positive sentiment by categorizing the sentiment polarity as the class label.

Time-Series Forecasting and Prediction

Recurrent neural networks are commonly utilized for time-series forecasting and prediction due to their inherent suitability. Unlike traditional text applications, these networks have input units represented as real-valued vectors instead of discrete one-hot encoded vectors. In the context of real-valued prediction, the output layer typically employs linear activations rather than the softmax function. However, if the output is a discrete value, such as the identifier of a specific event, it is also feasible to utilize discrete outputs with softmax activation. While various forms of recurrent neural networks, such as LSTM or GRU, can be employed for such tasks, it is important to note that a common challenge faced in time-series analysis is the potential length of the sequences involved.

While LSTM and GRU models offer some level of protection as time-series length increases, they do have limitations in terms of performance. This is due to the fact that the effectiveness of LSTMs and GRUs diminishes when applied to series with very long lengths. It is common for time-series data to have numerous time-stamps

with a range of short- and long-term dependencies. As a result, predicting and forecasting in such cases present distinct challenges.

Several practical solutions are available for forecasting time series data, especially when dealing with a moderate number of time series. One particularly effective approach is employing echo-state networks, which have been shown to accurately predict both continuous and discrete observations using only a small number of time series. It is important to note that echo-state networks rely on random expansion of the feature space through hidden units, making it crucial to keep the number of inputs low. If working with a large number of original time series, it may not be feasible to sufficiently expand the hidden space to accommodate all relevant features. It is worth mentioning that the majority of time-series forecasting models in the literature are univariate in nature.

End-to-End Speech Recognition

In end-to-end speech recognition, the goal is to transcribe raw audio files directly into character sequences, with minimal intermediate processing steps. Some preprocessing is necessary to prepare the data for input, such as converting the raw audio files into spectrograms using the specgram function from the matplotlib python toolkit (work cited as [157]).

The width utilized was 254 Fourier windows with a 127-frame overlap and 128 inputs per frame. The output comprises a character in the transcription sequence, such as a character, punctuation mark, space, or null character. The label may vary depending on the specific application being utilized. For instance, labels could encompass characters, phonemes, or musical notes.

A bidirectional recurrent neural network is ideal for this situation as it leverages context from both preceding and succeeding characters to enhance accuracy. However, a key challenge lies in ensuring alignment between audio frame representations and transcription sequences in this setting.

The alignment of this type is not predetermined and is actually a result of the system. This creates a challenge of circular dependency between segmentation and recognition, known as Sayre's paradox. To address this issue, connectionist temporal classification is employed. This method combines a dynamic programming algorithm with the probabilistic outputs of the recurrent network to identify the alignment that maximizes the overall probability of generation.

UNIT–V: CONVOLUTIONAL NEURAL NETWORKS

A Convolutional Neural Network (CNN) is a Deep Learning neural network architecture frequently utilized in Computer Vision, a branch of Artificial Intelligence dedicated to enabling computers to comprehend and interpret visual data. Artificial Neural Networks, particularly Convolutional Neural Networks, demonstrate noteworthy efficacy in various Machine Learning tasks involving datasets such as images, audio, and text. Different types of Neural Networks serve specific functions; for instance, Recurrent Neural Networks, specifically Long Short-Term Memory (LSTM) networks, excel in predicting word sequences, while Convolutional Neural Networks are optimal for image classification purposes.

In a regular Neural Network there are three types of layers:

1. **Input Layers:** The input layer is where we provide input to our model. The number of neurons in this layer should match the total number of features in our dataset, such as the number of pixels in an image.
3. **Hidden Layer:** The input from the Input layer is passed to the hidden layer, with the potential for multiple hidden layers depending on the model and data size. Each hidden layer may contain varying numbers of neurons, typically exceeding the number of features. Output from each layer is computed through matrix multiplication of the previous layer's output with the layer's learnable weights, followed by addition of learnable biases and activation function to introduce nonlinearity to the network.
4. **Output Layer:** The hidden layer output is next passed through a logistic function, such as sigmoid or softmax, to transform each class output into a probability score for that class.

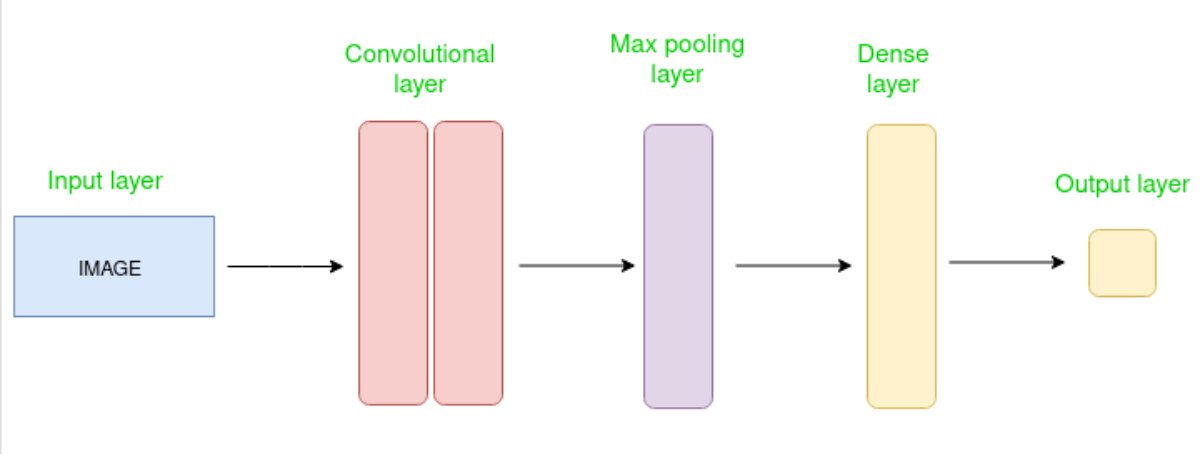
The input data is processed through the model, generating outputs at each layer in a process known as feedforward. Subsequently, we evaluate the model's performance by calculating the error using various error functions such as cross-entropy and square loss error. The error function gauges the network's efficiency. Following this, we update the model by computing the derivatives through a process referred to as Backpropagation, aimed at minimizing the loss.

Convolution Neural Network

The Convolutional Neural Network (CNN) is an expanded form of artificial neural networks (ANN) commonly utilized for extracting features from matrix datasets with a grid-like structure. Specifically, it is well-suited for analyzing visual datasets such as images or videos where identifying data patterns is crucial.

CNN architecture

Convolutional Neural Network consists of multiple layers like the input layer, Convolutional layer, Pooling layer, and fully connected layers.



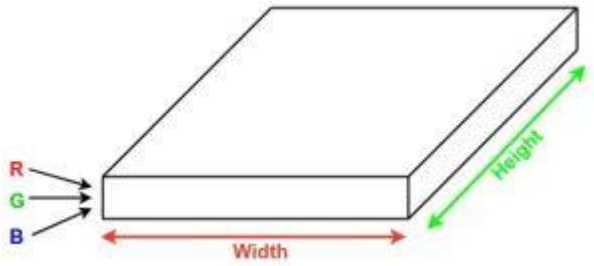
Simple CNN architecture

The Convolutional layer applies filters to the input image for feature extraction, the Pooling layer downsamples the image to decrease computation, and the fully

connected layer makes the final prediction. The network optimizes filters through backpropagation and gradient descent to improve performance.

How Convolutional Layers works

Convolutional Neural Networks, also known as covnets, are neural networks that utilize shared parameters. Visualize an image as a three-dimensional cuboid with dimensions for length, width (representing the image's size), and height (corresponding to channels such as red, green, and blue).



Consider the scenario of extracting a small segment from an image and applying a filter or kernel - a small neural network - with K outputs arranged vertically. By sliding this neural network across the entire image, a new image of varying dimensions is created, with more channels but reduced width and height compared to the original image. This process, known as Convolution, expands the image beyond just the standard R, G, and B channels.

If the size of the patch matches that of the image, it essentially functions as a typical neural network. The advantage of this approach is the reduction in the number of weights due to the utilization of a smaller patch.

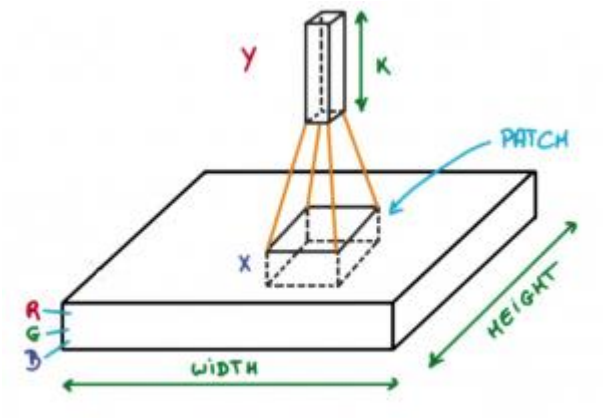


Image source: Deep Learning Udacity

Mathematics that is involved in the whole convolution process.

- Convolution layers are comprised of a collection of learnable filters (or kernels) with small widths and heights matching the depth of the input volume (3 in the case of an image input layer).
- To illustrate, when performing convolution on an image with dimensions of 34x34x3, the filters can have sizes of axax3, where 'a' can vary (e.g. 3, 5, or 7) but must be smaller than the image dimensions.
- During the forward pass, each filter is systematically slid across the entire input volume using a specified stride (e.g. 2, 3, or 4 for high-dimensional images). The dot product between the kernel weights and the input volume patches is computed at each step.
- Sliding the filters generates a 2-D output for each filter, which are then stacked together to form an output volume with a depth equal to the number of filters. The network effectively learns all the filters through this process.

Layers used to build ConvNets

An architecture consisting of complete Convolution Neural Networks is commonly referred to as covnets. Each covnets is comprised of a sequence of

layers that utilize differentiable functions to transform one volume to another.

Types of layers:

Let's take an example by running a covnets on of image of dimension 32 x 32 x 3.

- **Input Layers:** The input layer is where we provide data to our model. In Convolutional Neural Networks (CNN), typically the input consists of images or a series of images. This layer houses the unprocessed image data, which typically has dimensions of 32x32 pixels and a depth of 3 channels.
- **Convolutional Layers:** The layer responsible for extracting features from the input dataset utilizes learnable filters, also known as kernels. These filters are applied to the input images in the form of smaller matrices with dimensions typically 2x2, 3x3, or 5x5. The filters slide over the input image data, computing the dot product between the kernel weight and the corresponding input image patch. The result of this process is referred to as feature maps. If 12 filters are used for this layer, the output volume will have dimensions of 32 x 32 x 12.
- **Activation Layer:** Activation layers introduce nonlinearity to the network architecture by applying an activation function to the output of the previous layer. Common activation functions include RELU (max(0, x)), Tanh, Leaky RELU, and others. The dimensions of the output volume from the activation layer will remain the same, resulting in a volume size of 32 x 32 x 12.
- **Pooling layer:** The pooling layer in convolutional neural networks (covnets) is intermittently integrated to shrink the volume size, enabling faster computation, conserving memory, and avoiding overfitting. Two prevalent forms of pooling layers include max pooling and average pooling. By implementing a max pool with 2 x 2 filters and a stride of 2, the resulting volume will measure 16x16x12.

- **Max Pooling:** When the filter is applied to the input data, it identifies the pixel with the highest value and sends it to the output array. Notably, this method is commonly preferred over average pooling in many applications.
- **Average Pooling:** The average value within the receptive field is calculated by the filter as it traverses the input, and this value is then transmitted to the output array.

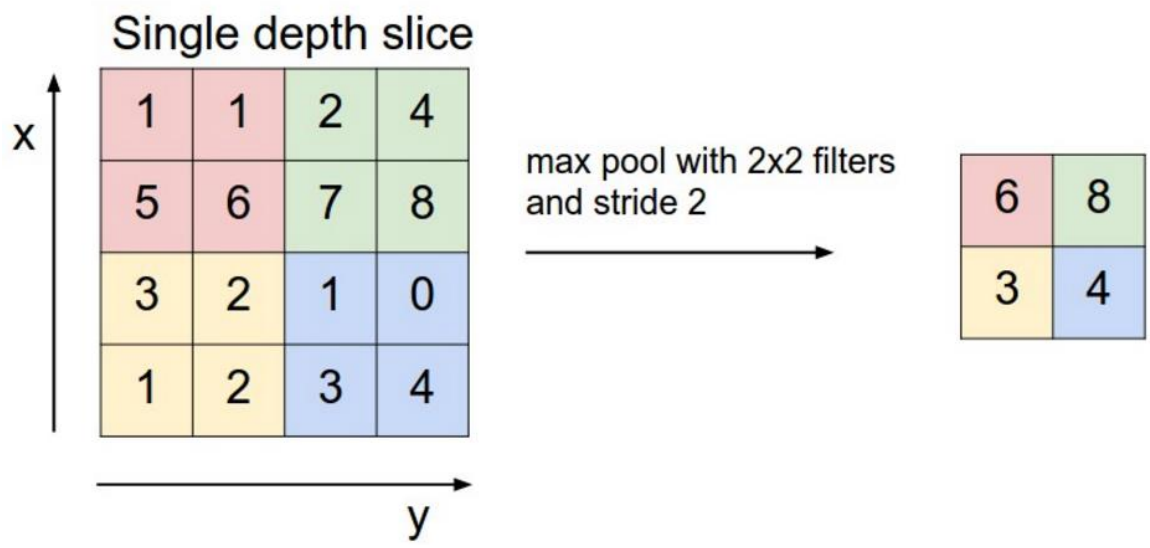


Image source: cs231n.stanford.edu

- **Flattening:** The feature maps produced from the convolution and pooling layers are then transformed into a one-dimensional vector for easy input into a fully connected layer for classification or regression tasks.
- **Fully Connected Layers:** The final classification or regression task is computed by taking the input from the previous layer.

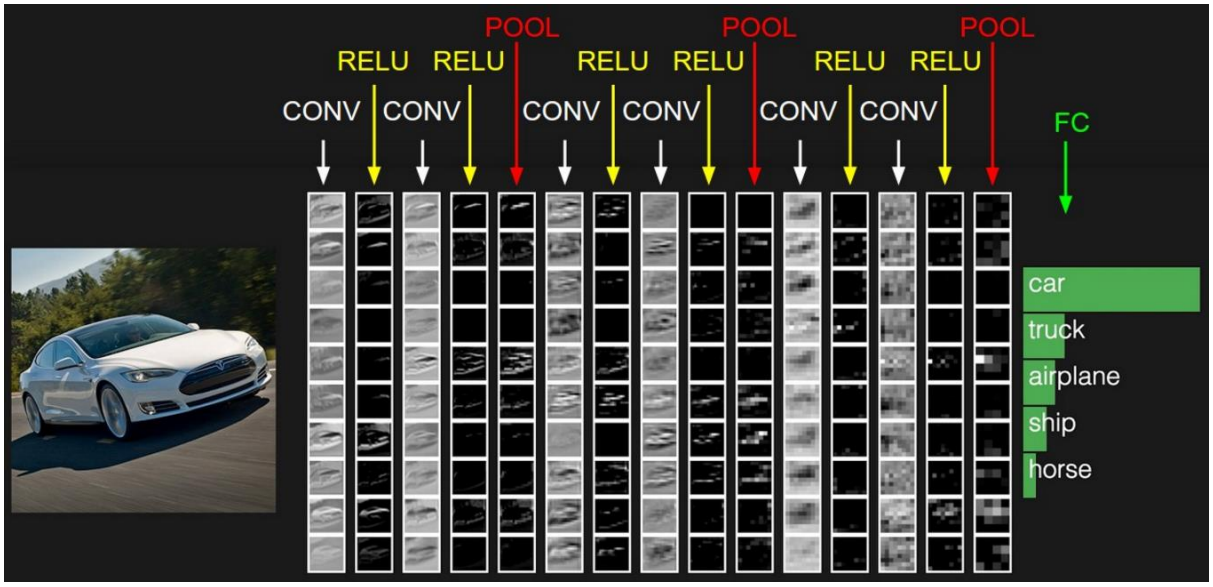
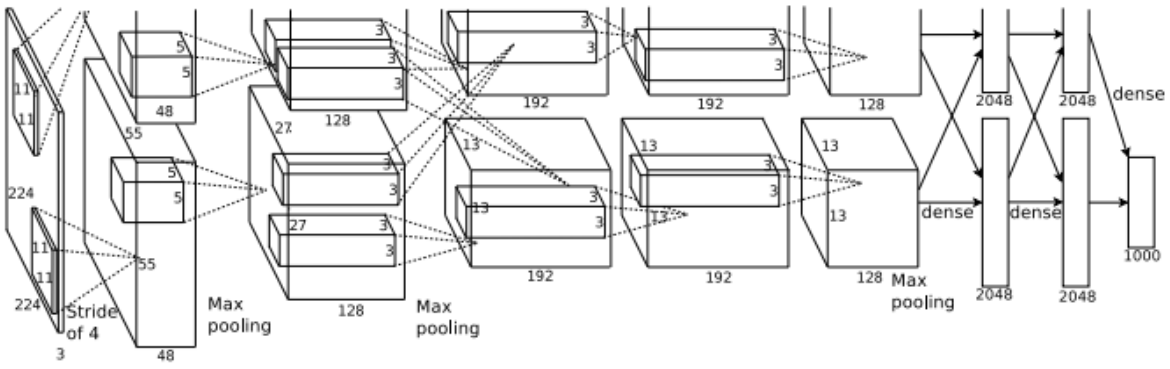


Image source: cs231n.stanford.edu

- **Output Layer:** The fully connected layers' output is utilized in a logistic function for classification purposes, such as sigmoid or softmax. This function transforms the output of each class into a probability score for each class.

- **Convolutional Architectures-AlexNet,VGG,GoogleNet,ResNet:**

In 2012, AlexNet surpassed all previous competitors and emerged victorious by effectively lowering the top-5 error rate from 26% to 15.3%. The non-CNN variant in second place had a top-5 error rate of approximately 26.2%.

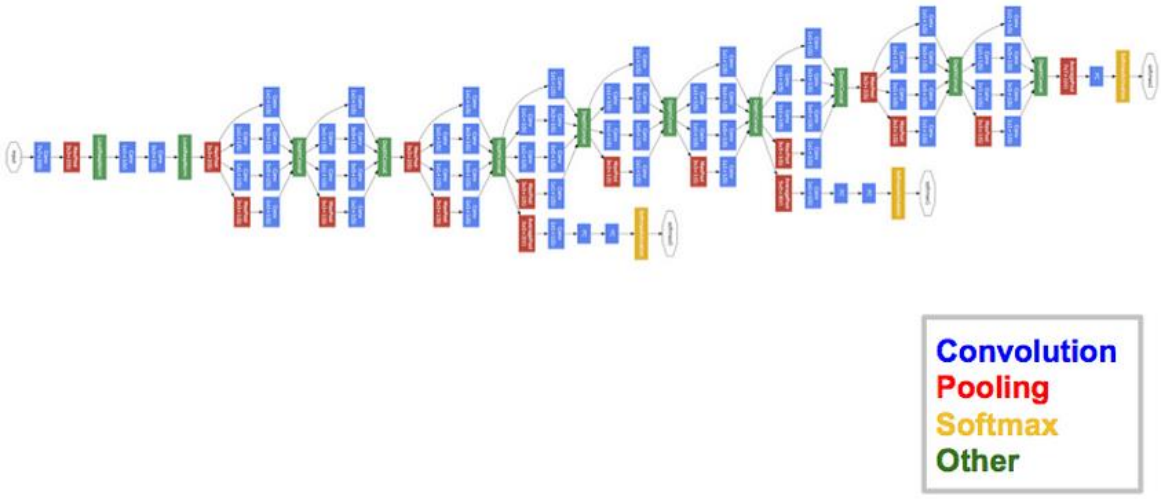


The network was designed with a similar architecture to LeNet by Yann LeCun and team, but it featured a greater depth, more filters per layer, and stacked convolutional layers. It included 11x11, 5x5, and 3x3 convolutions, as well as max pooling, dropout, data augmentation, and ReLU activations with SGD and momentum. ReLU activations were applied after each convolutional and fully-connected layer. AlexNet was trained over 6 days using two Nvidia Geforce GTX 580 GPUs, leading to the network being split into two pipelines. The design of AlexNet was led by the SuperVision group, which included Alex Krizhevsky, Geoffrey Hinton, and Ilya Sutskever.

GoogLeNet/Inception(2014)

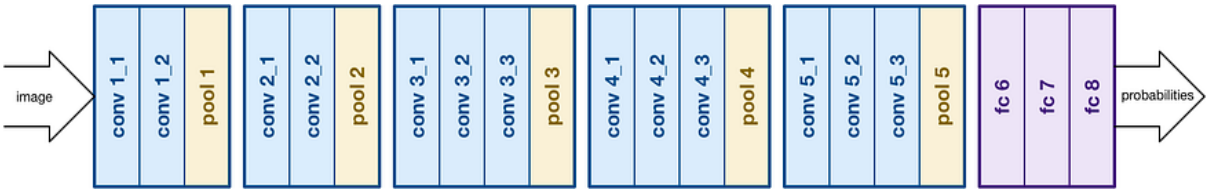
The ILSVRC 2014 competition was won by GoogLeNet, also known as Inception V1, developed by Google. It achieved a top-5 error rate of 6.67%, which was very close to human-level performance. As a result, the competition organizers were prompted to assess the model's capabilities. However, evaluating its accuracy proved to be a challenge and required human training. After several days of training, the human expert, Andrej Karpathy, was able to achieve a top-5 error rate of 5.1% (single model) and 3.6% (ensemble).

GoogLeNet's network architecture utilized a convolutional neural network (CNN) inspired by LeNet, but incorporated a novel element called an inception module. The model also implemented batch normalization, image distortions, and RMSprop to enhance performance. The inception module utilized multiple small convolutions to significantly reduce the number of parameters. Despite being a 22-layer deep CNN, the model successfully decreased the number of parameters from 60 million (AlexNet) to 4 million.



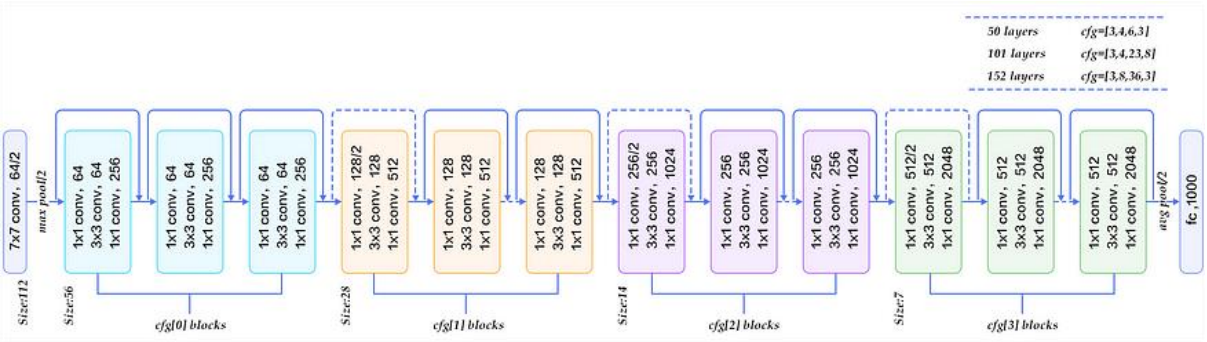
VGGNet (2014)

The runner-up in the ILSVRC 2014 competition, known as VGGNet in the community, was created by Simonyan and Zisserman. VGGNet features 16 convolutional layers and is highly regarded for its consistent architecture. Similar to AlexNet, it utilizes 3x3 convolutions with a high number of filters. The model was trained on 4 GPUs for 2-3 weeks and is currently a popular choice for image feature extraction within the community. The weight configuration of VGGNet is publicly available and has been utilized as a baseline feature extractor in various applications and challenges. However, with 138 million parameters, handling VGGNet can be somewhat challenging.

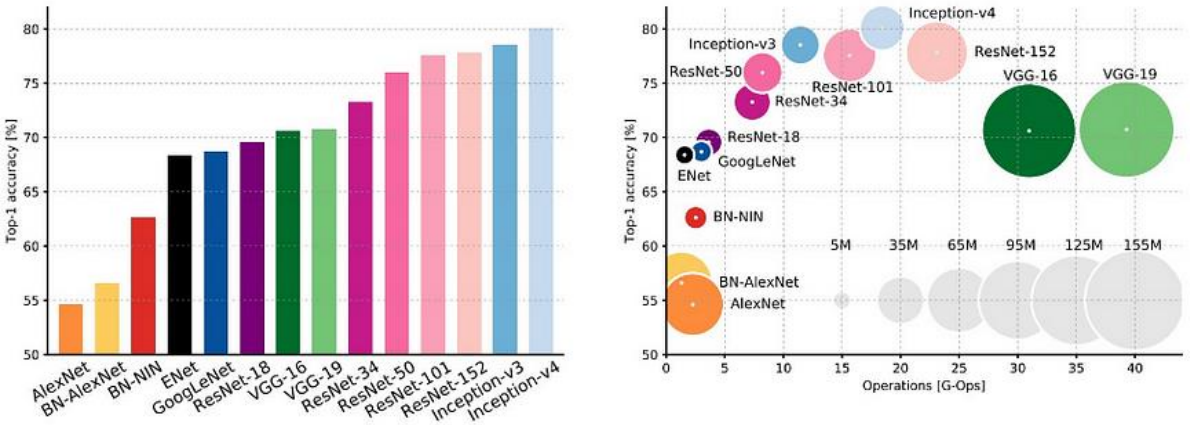


ResNet(2015)

In 2015, at the ILSVRC, Kaiming He and his colleagues introduced a novel architecture called the Residual Neural Network (ResNet). This groundbreaking model utilized "skip connections" and incorporated heavy batch normalization. These skip connections, also referred to as gated units or gated recurrent units, bear a resemblance to successful elements seen in RNNs. By employing this technique, the researchers were able to successfully train a neural network with an impressive 152 layers, all while maintaining lower complexity compared to VGGNet. The ResNet achieved a top-5 error rate of 3.57%, surpassing human-level performance on the dataset.



AlexNet employs two parallel CNN lines trained on separate GPUs with cross-connections, GoogleNet incorporates inception modules, and ResNet utilizes residual connections.



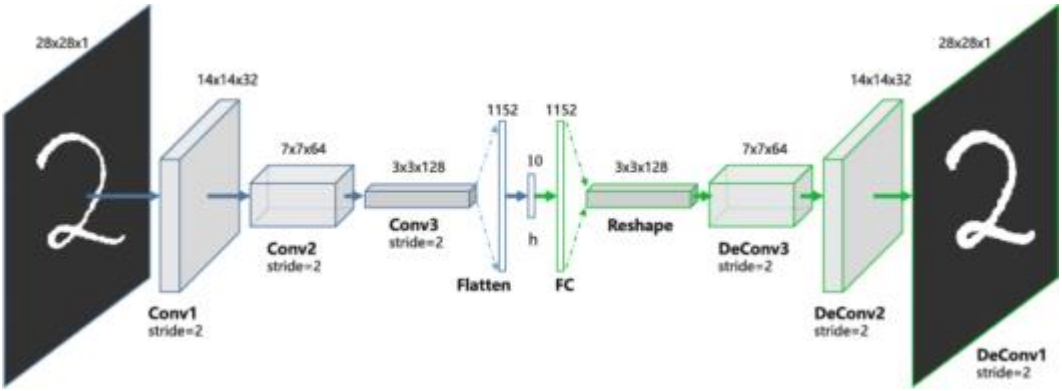
An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Summary Table

Year	CNN	Developed by	Place	Top-5 error rate	No. of parameters
1998	LeNet(8)	Yann LeCun et al			60 thousand
2012	AlexNet(7)	Alex Krizhevsky, Geoffrey Hinton, Ilya Sutskever	1st	15.3%	60 million
2013	ZFNet()	Matthew Zeiler and Rob Fergus	1st	14.8%	
2014	GoogLeNet(19)	Google	1st	6.67%	4 million
2014	VGG Net(16)	Simonyan, Zisserman	2nd	7.3%	138 million
2015	ResNet(152)	Kaiming He	1st	3.6%	

Convolution Autoencoders(CAE)

Autoencoders, when traditionally formulated, do not consider the concept of signals being a combination of multiple other signals. Convolutional Autoencoders leverage the convolution operator to capitalize on this insight. These models are trained to encode the input as a collection of basic signals and subsequently attempt to reconstruct the input by altering the image's geometry or reflectance.



Use cases of CAE:

- Image Reconstruction
- Image Colorization
- latent space clustering
- generating higher resolution images

Applications of CNN:

Below are some applications of Convolutional Neural Networks used today:

Image Classification

Convolutional Neural Networks (CNNs) are instrumental in image classification across multiple sectors. By leveraging CNNs, computers can effectively analyze and organize visual data, opening up a range of possibilities.

One prominent use case is image tagging, a process that assigns labels or tags to images according to their content. This functionality is widely employed on online platforms such as social media, eCommerce, and photo-sharing websites to improve user interactions and assist in discovering relevant content.

Recommender Systems

CNNs can also be utilized in recommendation engines, leveraging image data to suggest products or services to consumers. For instance, an eCommerce platform could employ image classification to recommend clothing items that align with a customer's style or preferences.

Image Retrieval

Image classification is utilized in various industries for efficient searching based on visual content rather than text-based queries. This is especially beneficial in sectors like fashion, where users seek items that align with specific styles or color palettes.

Additionally, image classification finds application in object detection, aiming to spot and locate objects within images, and semantic segmentation, assigning labels to each pixel in an image. These technologies have diverse use cases, including enhancing self-driving cars, bolstering security and surveillance systems, and improving medical imaging procedures.

Face Recognition

Facial recognition, a component of image recognition, is designed to detect and identify human faces within images or videos. This technology is frequently utilized in social media platforms to facilitate tasks like tagging friends in photos and videos, as well as for enhancing security measures and verifying user accounts.

Optical Character Recognition

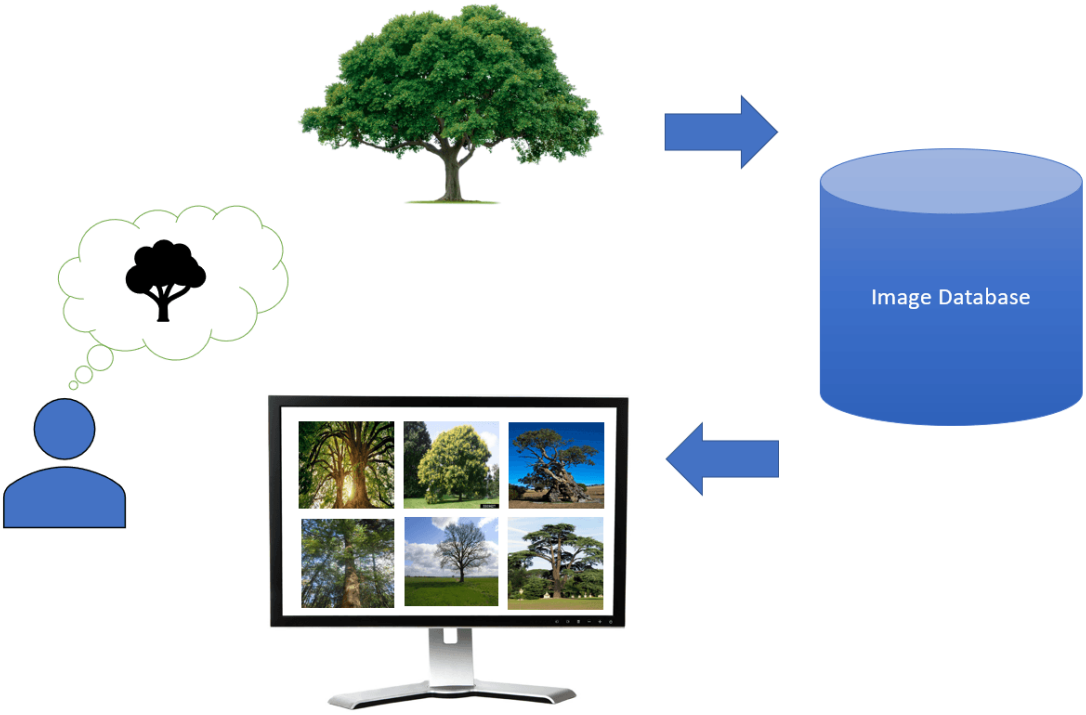
Optical Character Recognition (OCR) is a technology that allows for the recognition and interpretation of printed or handwritten text from various sources, including images, scanned documents, and more. OCR algorithms make use of machine learning techniques, such as convolutional neural networks, to analyze the shape and structure of characters and patterns in order to transcribe the text accurately.

OCR has a wide range of applications in fields such as document management, digital archiving, and data entry. It can streamline the conversion of paper documents into searchable and editable digital text, automating the process efficiently. Additionally, OCR is commonly utilized in automated systems for

processing forms and invoices, as well as in the creation of ebooks and digital libraries.

Content-Based Image Retrieval (CBIR)?

Content-Based Image Retrieval (CBIR) is a method used to retrieve images from a database by comparing visual features such as shapes, colors, texture, and spatial information. Essentially, a user inputs a query image, and CBIR identifies and retrieves images in the database that are similar to the query image based on these visual features. The comparison process involves analyzing the content of the query image and measuring its similarity to the database images in terms of the aforementioned features:



The image provided showcases a query example that represents the user's information requirement alongside a vast dataset of images. The goal of the CBIR system is to rank all images in the dataset based on their likelihood of meeting the user's information need.

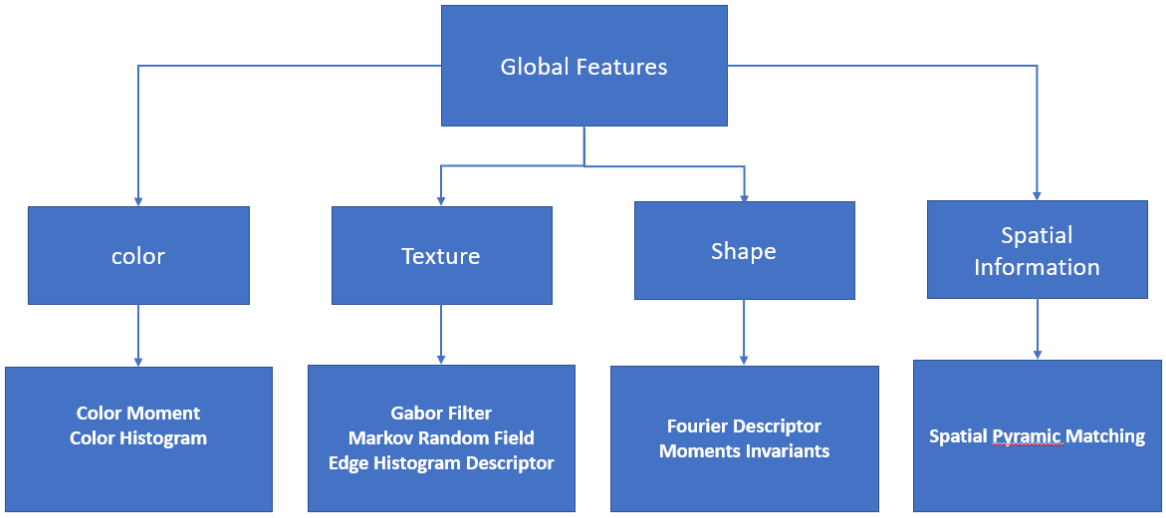
Feature Extraction Methods in CBIR

There are two main categories of visual features: global and local.

Global Features

Global features encompass information that characterizes the entirety of an image. Examples of such features include color descriptors like color moments and color histograms, which provide insight into the overall color spaces within an image. Additionally, other global features extend to visual elements such as shapes and textures.

In this diagram, we find various methods for global feature extraction:



Local Features

While global features offer several advantages, they are subject to changes in scaling and rotation. As a result, local features are considered to be more dependable in various conditions.

Local features pertain to the description of visual patterns or structures that can be identified within small groups of pixels. These include edges, points, and a variety of image patches.

The descriptors utilized for extracting local features take into account the regions surrounding the identified visual structures. These descriptors convert a local pixel neighborhood into a vector representation.

One of the most commonly utilized local descriptors is SIFT, which stands for Scale-Invariant Feature Transform. This feature consists of both a descriptor and a key point detector. It remains unchanged even when the image is rotated. However, there are certain drawbacks associated with SIFT, such as the requirement for a fixed vector for encoding and a significant amount of memory.

Object Detection

Object detection is a computer vision method used to pinpoint objects within images or videos. Harnessing machine learning or deep learning, object detection algorithms provide valuable outcomes. Humans possess the inherent ability to swiftly identify and locate objects within visual content. The objective of object detection is to emulate this cognitive prowess through computational means.

Why Object Detection Matters

Object detection plays a crucial role in the development of advanced driver assistance systems (ADAS), allowing vehicles to identify driving lanes and perform pedestrian detection for enhanced road safety. Additionally, object detection is valuable in industries like video surveillance and image retrieval systems.

How It Works

Object Detection Using Deep Learning

Various techniques are available for object detection, with popular options including deep learning methods that utilize convolutional neural networks (CNNs) like R-CNN and YOLO v2. These advanced approaches are designed to automatically recognize and detect objects within images.

You can choose from two key approaches to get started with object detection using deep learning:

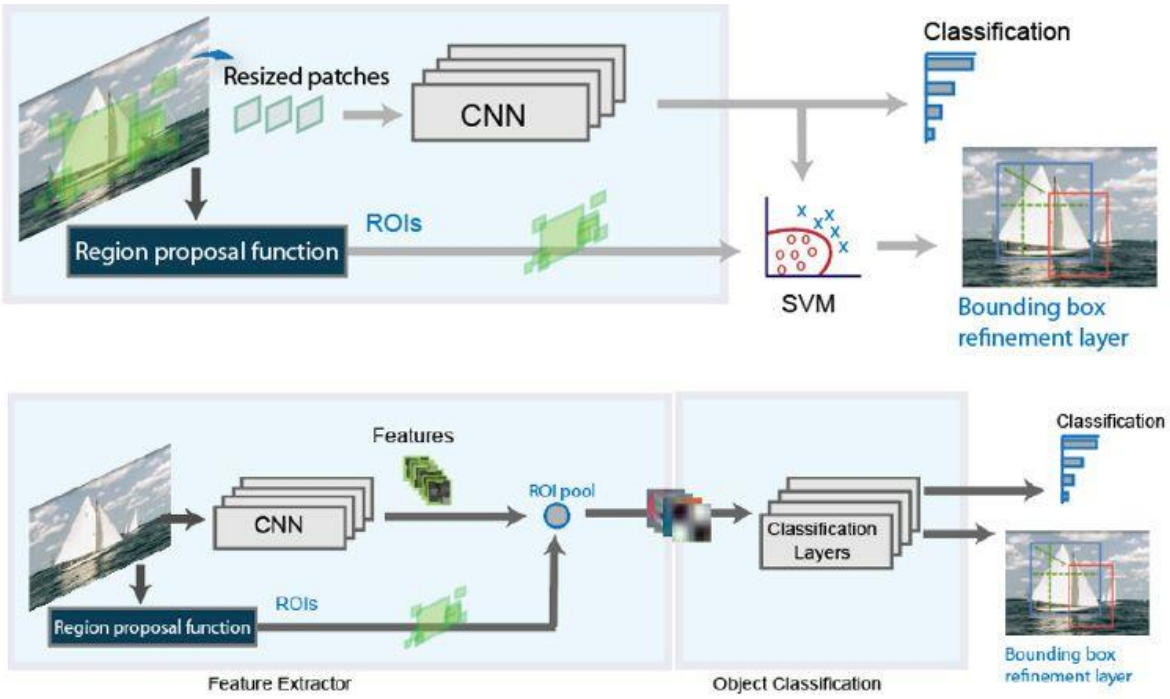
- **Create and train a custom object detector.** To develop a custom object detector from the ground up, it is essential to create a network architecture tailored to extract features relevant to the objects in focus. Moreover, a substantial amount of labeled data is indispensable for training the Convolutional Neural Network (CNN) effectively. The outcomes of a custom object detector can be extraordinary; however, it is worth noting that configuring the layers and weights in the CNN manually is a painstaking process that demands considerable time and a sufficient amount of training data.
- **Use a pretrained object detector.** Several object detection workflows in deep learning utilize transfer learning, a technique that allows you to begin with a pre-trained network and then adjust it to meet the needs of your specific use case. This strategy can yield quicker outcomes as the object detectors have already undergone training on tens of thousands, or even millions, of images.

Detecting a stop sign using a pretrained R-CNN.

To decide on the type of object detection network to use, you must choose between a two-stage network or a single-stage network, regardless of whether you create a custom object detector or opt for a pretrained one.

Two-Stage Networks

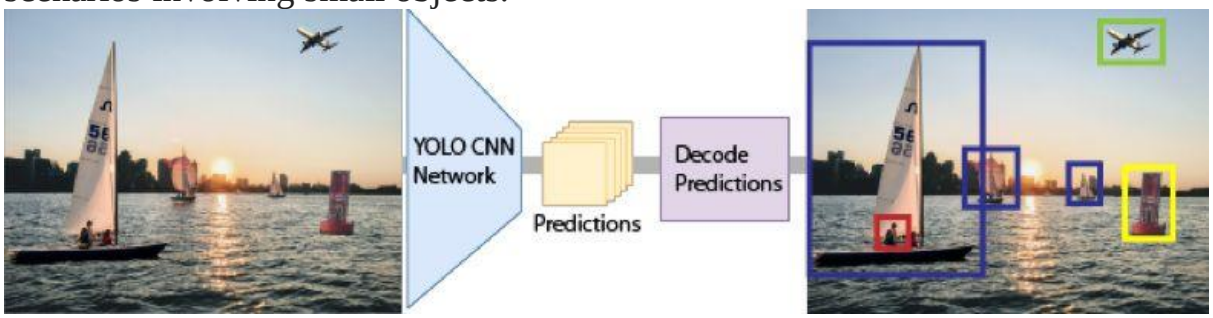
The first phase in two-stage networks, like R-CNN and its derivatives, involves detecting region proposals, which are specific areas in the image that could potentially contain an object. The subsequent phase focuses on categorizing the objects found within these region proposals. While two-stage networks can produce highly precise object detection outcomes, they generally operate slower compared to single-stage networks.



High-level architecture of R-CNN (top) and Fast R-CNN (bottom) object detection.

Single-Stage Networks

In single-stage networks, such as YOLO v2, the CNN generates network predictions for regions throughout the image using anchor boxes. These predictions are then decoded to form the ultimate bounding boxes for the objects. While single-stage networks are typically faster than two-stage networks, they may not achieve the same level of accuracy, particularly in scenarios involving small objects.



Overview of YOLO v2 object detection.

Natural Language and Sequence Learning

Although recurrent neural networks are typically the go-to for machine learning with text sequences, the use of convolutional neural networks has gained traction in recent years.

Initially, convolutional neural networks may not appear to be the most intuitive choice for text-mining tasks.

Initially, shapes in an image are perceived uniformly regardless of their location within the image, unlike text where the position of a word in a sentence holds significance. Additionally, the challenges of handling position translation and shift differ between image and text data. While neighboring pixels in an image tend to be similar, neighboring words in text are typically distinct. Despite these discrepancies, convolutional networks have demonstrated enhanced performance in recent years. Much like an image is represented as a 2-dimensional object with depth determined by the number of color channels, a text sequence is represented as a 1-dimensional object with depth determined by its representation dimensionality. For text sentences, the representation dimensionality is equivalent to the lexicon size in the case of one-hot encoding. Hence, instead of 3-dimensional boxes with spatial and depth aspects, a lightweight markdown language such as Markdown or CommonMark may be utilized for text representation.

In the subsequent layers of the convolutional network, depth is determined by the quantity of feature maps instead of the size of the lexicon. Additionally, the number of filters in a specific layer determines the amount of feature maps in the following layer, similar to how it works with image data. When working with image data, convolutions are carried out at every 2D position, whereas with text

data, convolutions are applied at every 1D point within the sentence using the identical filter.

An issue with this method is that employing one-hot encoding can lead to a significant increase in the number of channels, resulting in a high number of parameters in the filters of the initial layer. The vocabulary size of a typical dataset is usually around 106. To address this, pretrained word embeddings like word2vec or GLoVe are utilized instead of one-hot encodings. These word embeddings offer semantic richness, allowing for a reduction in dimensionality to a few thousand from a hundred-thousand. This approach not only decreases the number of parameters in the initial layer by an order of magnitude but also provides a meaningful representation. Other operations, such as max-pooling or convolutions, for text data are similar to those used for image data (see Chapter 2 for more information).