Inductive Learning

Inductive learning is a fundamental concept in machine learning where the model infers general patterns or rules from specific examples or observations. In inductive learning, the system is provided with a set of **input-output pairs** (training data), and the goal is to generalize from this data to make predictions or decisions about unseen instances.

How Inductive Learning Works:

1. **Training Data**:
   - The model is given a set of labeled examples (training data) where each instance consists of input features and an associated output (label or target).
   - For example, in a classification task, the model might be provided with images of animals (inputs) and their corresponding labels (e.g., "cat," "dog," "bird").
2. **Generalization**:
   - From the specific examples, the inductive learning algorithm tries to **generalize** a rule or pattern that can apply to new, unseen examples.
   - The model builds a **hypothesis** or function that maps inputs to the corresponding outputs.
3. **Prediction**:
   - Once the model has learned from the training data, it can be used to make predictions on new, unseen data. The goal is for the model to generalize well and make accurate predictions on data it has not encountered before.

Example of Inductive Learning:

- **Email Spam Classifier**:
  - Suppose you want to create a model that classifies emails as **spam** or **not spam**.
  - You provide the model with a set of training data where each email is labeled as either "spam" or "not spam" based on its content, sender, or other features.
  - The model learns patterns from these labeled emails (e.g., spam emails might contain certain words like "free" or "winner").
  - Once trained, the model can classify new, unseen emails based on the patterns it has learned from the training data.

Inductive Learning vs. Deductive Learning:

- **Inductive Learning**: Infers general rules from specific examples. It starts with **data** (examples) and tries to generalize to a rule or hypothesis that applies to all similar cases. Most machine learning algorithms, including decision trees, neural networks, and support vector machines, rely on inductive learning.
- **Deductive Learning**: Starts with a **general rule** and applies it to specific cases. In this type of learning, the system is provided with a pre-defined set of rules, and the goal is to apply those rules to new examples. Deductive learning doesn’t involve the same kind of generalization process as inductive learning.

Advantages of Inductive Learning:

1. **Generalization**:
   - Inductive learning is focused on generalizing from specific examples, making it powerful for predicting outcomes in **unseen data**.
2. **Adaptability**:
   - The system can learn patterns from a wide range of data, including structured data (e.g., tabular data) and unstructured data (e.g., images or text).
3. **Real-World Application**:
   - Most machine learning applications use inductive learning because it allows models to be trained on **historical data** and then applied to make predictions on future data.

Challenges of Inductive Learning:

1. **Overfitting**:
   - The model may learn the training data too well, including the noise or random fluctuations, leading to poor performance on new, unseen data.
2. **Bias-Variance Trade-off**:
   - Finding the right balance between **bias** (underfitting) and **variance** (overfitting) is key in inductive learning. A model with high bias will not capture the underlying patterns in the data, while a model with high variance will overfit the training data.
3. **Need for High-Quality Data**:
   - The success of inductive learning depends on the quality and quantity of training data. If the data is biased, incomplete, or noisy, the generalizations made by the model may not be accurate.

Examples of Inductive Learning Algorithms:

- **Decision Trees**: Learns rules by recursively splitting the data based on feature values to create decision boundaries.
- **Support Vector Machines (SVM)**: Finds the hyperplane that best separates different classes in the data.
- **Neural Networks**: Learns complex patterns by adjusting weights between layers of neurons based on error in predictions.
- **k-Nearest Neighbors (k-NN)**: Learns by storing all the training data and making predictions based on the closest neighbors to the input.

Naive Bayes

Naive Bayes Classifier

A Naive Bayes classifier is a probabilistic machine learning model used for classification tasks.

It is based on Bayes' Theorem.

Bayes' Theorem Formula:

$$P(A|B) = \frac{P(B|A) \times P(A)}{P(B)}$$

Where:

- $P(A|B)$: Posterior probability of event A occurring given that event B has occurred.
- $P(B|A)$: Likelihood of event B occurring given that event A has occurred.
- $P(A)$: Prior probability of event A occurring.
- $P(B)$: Marginal probability of event B occurring.

P(A) - Prior Probability:

- Represents our initial belief about the probability of event A before observing any new evidence.
- Example: The general probability of a person having a particular disease in the population.

P(B|A) - Likelihood:

- The probability of observing event B given that event A is true.
- Example: The probability of a positive test result if a person actually has the disease.

P(B) - Marginal Probability:

- The total probability of observing event B under all possible scenarios.
- Can be calculated using the Law of Total Probability:
  $P(B) = P(B|A)P(A) + P(B|\text{not } A)P(\text{not } A)$
- Example: The overall probability of getting a positive test result, whether or not a person has the disease.

P(A|B) - Posterior Probability:

- The updated probability of event A occurring after observing event B.
- Represents how our belief about A changes in light of new evidence B.
- Example: The probability that a person has the disease given that they have a positive test result.

Naive Bayes' Classifier

The classifier computes the posterior probability $P(C_k|X)$ of class C_k given a feature vector $X = (x_1, x_2, ..., x_n)$:

$$P(C_k|X) = \frac{P(C_k) \prod_{i=1}^{n} P(x_i|C_k)}{P(X)}$$

Where:

- $P(C_k)$ is the prior probability of class C_k.
- $P(x_i|C_k)$ is the likelihood of feature x_i given class C_k.
- $P(X)$ is the marginal probability of the feature vector X.

The Naive Independence Assumption

The core idea behind the Naive Bayes classifier is the assumption that all features (predictors) are conditionally independent of each other, given the class label.

This means that the probability of one feature occurring does not affect the probability of another feature occurring, once we know the class. Mathematically, this is represented as:

$$P(X_1, X_2, ..., X_n|C) = P(X_1|C) \times P(X_2|C) \times ... \times P(X_n|C)$$

Where:

- $X_1, X_2, ..., X_n$ are the features (predictors).
- C is the class label.

What This Means:

In a Naive Bayes classifier, the model treats each feature independently when calculating the likelihood $P(X_i|C)$ for each feature X_i.

For instance, if we're classifying an email as spam or not spam based on the presence of certain words, the model assumes that knowing the word "free" appears in the email gives no information about whether the word "win" also appears, given the email is spam or not spam.

Why the Assumption Is Considered "Naive":

- **Real-World Feature Dependence:**
  In most real-world datasets, features are not truly independent. For example, in email classification, the presence of certain words (like "win" and "free") is likely correlated, meaning they often occur together in spam emails. Ignoring these dependencies is what makes the assumption "naive."
- **Simplification for Computational Efficiency:**
  Despite the assumption being unrealistic, it drastically simplifies the calculations, making the Naive Bayes classifier computationally efficient and easier to implement. The naive assumption allows the model to decompose a complex joint probability distribution into a product of simpler, manageable terms. Without this assumption, calculating the joint distribution of the features would become exponentially more complex, especially as the number of features increases.

Implications of the Independence Assumption on Model Simplicity and Computational Efficiency:

- **Model Simplicity:**
  - **Reduced Complexity:** Because the Naive Bayes classifier assumes independence between features, it doesn't need to model the relationships or interactions between features. This reduces the complexity of the model, making it straightforward to train and interpret.
  - **Fewer Parameters:** Unlike more complex models, Naive Bayes doesn't require estimating parameters that describe feature dependencies. It only needs to estimate the probability distributions of individual features within each class, which results in far fewer parameters to compute, especially in high-dimensional datasets.
- **Computational Efficiency:**
  - **Scalability:** The independence assumption allows the classifier to scale well to large datasets with many features. Calculating the likelihood for each feature independently makes the model highly efficient, even with large feature sets.
  - **Fast Training and Prediction:** The training process is fast because the model doesn’t require iterative optimization, and the predictions are based on simple probability calculations. This makes Naive Bayes suitable for real-time applications, like spam filtering or text classification, where speed is crucial.

Mechanics of Naive Bayes Classification

1. Step 1: Calculate Prior Probabilities $P(C_k)$ for Each Class

The **prior probability** reflects how likely a particular class is to occur before considering any of the features. It is calculated based on the relative frequency of each class in the dataset.

- **Formula:**

$$P(C_k) = \frac{\text{Number of instances in class } C_k}{\text{Total number of instances}}$$

Example (Weather and Tennis Dataset):

Suppose we are trying to predict whether someone will play tennis (Yes/No) based on weather conditions. The dataset has two classes: "PlayTennis = Yes" and "PlayTennis = No." If we have 9 instances where the person plays tennis and 5 instances where they don't out of a total of 14, the prior probabilities are:

P(Yes) = 9/14 ≈ 0.64, P(No) = 5/14 ≈ 0.36

Step 2: Calculate Likelihood $P(x_i|C_k)$ for Each Feature x_i

The **likelihood** is the probability of observing a particular feature value x_i given that the class is C_k. For categorical features, it is calculated as the frequency of each feature value within a class. For continuous features (in Gaussian Naive Bayes), we assume the data follows a normal distribution and use the Gaussian probability density function.

- **Formula for Categorical Features:**

$$P(x_i|C_k) = \frac{\text{Number of times feature } x_i \text{ appears in class } C_k}{\text{Number of instances in class } C_k}$$

Example (Weather and Tennis Dataset):

Suppose we want to classify if someone will play tennis based on three features: Outlook, Temperature, and Humidity. Assume the current weather is:

- **Outlook:** Sunny
- **Temperature:** Hot
- **Humidity:** High

To calculate the likelihood of each feature given the class "Yes" (i.e., the person will play tennis), we look at how frequently each feature value appears when "PlayTennis = Yes."

- **Outlook = Sunny, given PlayTennis = Yes**
  If 2 out of the 9 "Yes" instances had Sunny weather:
  $P(\text{Sunny} \mid \text{Yes}) = 2/9 \approx 0.22$
- **Temperature = Hot, given PlayTennis = Yes**
  If 2 out of the 9 "Yes" instances had Hot temperature:
  $P(\text{Hot} \mid \text{Yes}) = 2/9 \approx 0.22$
- **Humidity = High, given PlayTennis = Yes:**
  If 3 out of the 9 "Yes" instances had High humidity:
  $P(\text{High} \mid \text{Yes}) = 3/9 \approx 0.33$

Repeat the same process for "No" (i.e., the person will not play tennis), calculating the likelihood of each feature value given the "No" class.

Step 3: Compute Posterior Probability $P(C_k|X)$ for Each Class Using Bayes' Theorem

Once the priors and likelihoods are calculated, we apply **Bayes' Theorem** to compute the **posterior probability**, which tells us how likely each class is given the observed feature values.

- **Bayes' Theorem:**

$$P(C_k|X) = \frac{P(C_k) \times P(X|C_k)}{P(X)}$$

Where:

- $P(C_k)$ is the prior probability of the class.
- $P(X|C_k) = P(x_1|C_k) \times P(x_2|C_k) \times ... \times P(x_n|C_k)$ is the likelihood of observing the feature set X given the class C_k.
- $P(X)$ is the marginal likelihood (constant across all classes, so it can be ignored when comparing classes).

Let's calculate $P(\text{Yes}|X)$ and $P(\text{No}|X)$, where X represents the weather conditions (Outlook = Sunny, Temperature = Hot, Humidity = High):

$$P(\text{Yes}|X) \propto P(\text{Yes}) \times P(\text{Sunny}|\text{Yes}) \times P(\text{Hot}|\text{Yes}) \times P(\text{High}|\text{Yes})$$

Substituting the values we calculated:

$$P(\text{Yes}|X) \propto 0.64 \times 0.22 \times 0.22 \times 0.33 \approx 0.0103$$

Similarly, compute for $P(\text{No}|X)$:

$$P(\text{No}|X) \propto P(\text{No}) \times P(\text{Sunny}|\text{No}) \times P(\text{Hot}|\text{No}) \times P(\text{High}|\text{No})$$

Assuming the likelihoods are:

- $P(\text{Sunny}|\text{No}) = 3/5 = 0.60$
- $P(\text{Hot}|\text{No}) = 2/5 = 0.40$
- $P(\text{High}|\text{No}) = 4/5 = 0.80$

Substituting the values:

$$P(\text{No}|X) \propto 0.36 \times 0.60 \times 0.40 \times 0.80 \approx 0.0691$$

Step 4: Select the Class with the Highest Posterior Probability

- After calculating the posterior probabilities for all classes, the final step is to select the class with the highest posterior probability. This is the predicted class.

Example:

In our example, we computed:

$$P(\text{Yes}|X) \approx 0.0103, \quad P(\text{No}|X) \approx 0.0691$$

Since $P(\text{No}|X) > P(\text{Yes}|X)$, the model predicts that the person will **not** play tennis under the given weather conditions.

Performance Evaluation Metrics

Confusion Matrix:

A **confusion matrix** provides a summary of the classification outcomes by showing the number of:

- **True Positives (TP):** Correctly classified positive instances.
- **False Positives (FP):** Incorrectly classified negative instances as positive.
- **True Negatives (TN):** Correctly classified negative instances.
- **False Negatives (FN):** Incorrectly classified positive instances as negative.

The matrix helps in calculating several other performance metrics by organizing the outcomes into these four categories.

Accuracy:

- **Formula:**

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

- **Interpretation:** Measures the proportion of correctly classified instances (both positives and negatives). It works well when classes are balanced.

Precision:

- **Formula:**

$$\text{Precision} = \frac{TP}{TP + FP}$$

- **Interpretation:** Indicates the accuracy of positive predictions, i.e., out of all predicted positives, how many were actually positive. Precision is important when false positives are costly.

Recall (Sensitivity):

- **Formula:**

$$\text{Recall} = \frac{TP}{TP + FN}$$

- **Interpretation:** Measures how well the model captures true positives, i.e., out of all actual positives, how many were correctly identified. Recall is crucial when false negatives are costly.

F1-Score:

- **Formula:**

$$F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

- **Interpretation:** The harmonic mean of precision and recall, balancing both metrics. It is useful when there is a trade-off between precision and recall, particularly in imbalanced datasets.

Advantages of Naive Bayes

1. **Easy and Fast to Implement:**
   - **Reason:** Naive Bayes classifiers are simple and rely on calculating probabilities directly from the training data. The model estimates the likelihood of each feature given the class label and multiplies these probabilities to make predictions.

2. **Performs Well with High-Dimensional Data:**
   - **Reason:** Naive Bayes works effectively with datasets that have a large number of features, even when the dataset contains more features than samples. Because the algorithm assumes feature independence, it does not suffer from the curse of dimensionality as much as other algorithms.

3. **Requires a Small Amount of Training Data:**
   - **Reason:** Naive Bayes relies on simple probability estimates rather than complex parameter tuning, allowing it to achieve good performance with a relatively small dataset. The underlying assumption of feature independence also reduces the need for large amounts of data to estimate joint probabilities.

Disadvantages of Naive Bayes

1. **Strong Feature Independence Assumption:**
   - **Reason:** The algorithm assumes that features are conditionally independent given the class label, which is rarely true in real-world scenarios. In many problems, features are often correlated or interact with each other, and this assumption leads to suboptimal predictions.
   - **Example:** In a medical diagnosis task, symptoms are often correlated (e.g., fever and sore throat), but Naive Bayes assumes that they contribute independently to the likelihood of a diagnosis. This can reduce its accuracy in such cases.
2. **Zero-Frequency Problem (Zero Probabilities):**
   - **Reason:** If a categorical variable in the test set has a category that was not observed in the training set, Naive Bayes will assign a probability of zero to that category, leading to incorrect predictions. This problem arises because the algorithm multiplies the likelihoods, and a zero probability eliminates the entire outcome.
   - **Example:** In text classification, if a word appears in the test data that did not appear in the training data, Naive Bayes will assign a zero probability to any class that includes that word in its prediction process.

Types of Naive Bayes Classifiers

1. Gaussian Naive Bayes:

- **Used for:** Continuous data.
- **Assumption:** Each feature follows a **normal (Gaussian) distribution** within each class.
- **Example:** Used for predicting continuous variables like age, height, or weight in classification tasks.

2. Multinomial Naive Bayes:

- **Used for:** Multinomially distributed data, particularly **discrete features** like counts or frequencies.
- **Common Application: Document classification** tasks where the features represent term frequencies (number of times a word appears in a document).
- **Assumption:** The features are multinomially distributed, and the classifier estimates the probability of each class based on word frequencies.
- **Example:** Classifying emails into spam or not spam based on word occurrence frequencies.

3. Bernoulli Naive Bayes:

- **Used for: Binary/Boolean features**, where each feature represents a binary outcome (1 or 0).
- **Common Application:** Feature values represent the presence (1) or absence (0) of a feature, such as whether a word is present in a document or not.
- **Example:** Useful in text classification where features are binary indicators of word presence, such as sentiment analysis using a bag-of-words model with word presence/absence.

Nearest Neighbour

Introduction to Nearest Neighbor

What is Nearest Neighbor (k-NN)?

- **k-NN** is a **lazy learning algorithm**:
  - Unlike eager learners (like decision trees or neural networks), it does not build an explicit model during the training phase. Instead, it stores the entire training dataset and makes decisions based on this stored data during inference.
  - In k-NN, predictions are made by finding the **k closest data points (neighbors)** to a new query point and then determining the output based on these neighbors.

Key Characteristics:

- **Non-parametric**: k-NN doesn't make assumptions about the data distribution.
- **Instance-based learning**: It memorizes the training data rather than generalizing from it.
- Can be applied to both **classification** and **regression** tasks.

How Does k-NN Work for Classification and Regression?

Classification:

- **Objective**: To assign a class label to a new data point.
- **Steps**:
  1. For a given query point, calculate the distance to every other point in the training dataset.
  2. Identify the **k nearest neighbors** based on the smallest distances.
  3. Perform a **majority vote** among the k neighbors: the query point is assigned the class that appears most frequently among the neighbors.
- **Example**: Predicting whether a patient has a disease (yes/no) based on health metrics like age, blood pressure, etc.

Regression:

- **Objective**: To predict a continuous output value.
- **Steps**:
  1. Calculate the distances between the query point and all other points.
  2. Select the **k nearest neighbors**.
  3. Take the **average** of the target values (or weighted average) of the k neighbors as the prediction.
- **Example**: Predicting the price of a house based on features like size, number of rooms, and location.

Working Mechanism of k-NN

Step 1: Calculate Distance Between the New Point and All Points in the Dataset

- The first step in k-NN is to determine how “close” each training data point is to the new query point.
- **Distance Metric**: Typically, the distance is calculated using a specific metric, such as:

Euclidean Distance (most common for continuous data):

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

This measures the straight-line distance between two points in an n-dimensional space.

Manhattan Distance: Useful when the distance needs to follow grid-like paths (city blocks).

$$d(x, y) = \sum_{i=1}^{n} |x_i - y_i|$$

Cosine Similarity: Often used in text data, where angles between high-dimensional vectors are more meaningful than raw distance.

Example: In a dataset containing the height and weight of individuals, the Euclidean distance between a new person (the query point) and all other individuals is calculated to determine how similar they are in these two features.

Step 2: Select the k-Nearest Neighbors Based on the Smallest Distance Values

- Once the distances between the query point and all other data points are calculated, the next step is to sort the distances in ascending order and select the **k closest points**.
  - If k=3, you select the **three closest points** from the dataset.
- **Key Concept: The Choice of k**:
  - A **small value of k** (e.g., k=1) might lead to overfitting or noise sensitivity, as the model bases its prediction on just one data point.
  - A **large value of k** (e.g., k=20) leads to a smoother prediction but may **overgeneralize** the results, reducing the model's ability to capture local patterns.
- **What Happens When k=1**:
  - In the extreme case where k=1, the algorithm makes a prediction based solely on the closest point in the dataset. While this ensures simplicity, it may result in poor generalization because any noise or outlier in the data can heavily influence the prediction.

Step 3: Assign the Class Label or Predict a Value

- After identifying the nearest neighbors, the k-NN algorithm makes predictions differently depending on whether it's a **classification** or **regression** problem.

For Classification (Majority Voting):

- **Majority Vote**: In classification problems, the algorithm checks the class labels of the k-nearest neighbors and assigns the label that occurs most frequently (majority voting).
    - **Example**: If k=3 and the nearest three neighbors are labeled as ["spam", "spam", "not spam"], the query point is classified as **"spam"** because it is the majority class.
- **Edge Cases**:
    - If there's a **tie** in voting (e.g., two "spam" and two "not spam" when k=4), tie-breaking mechanisms are needed. Common solutions include:
        - Reducing k to an odd number to avoid ties.
        - Selecting randomly among tied classes.

For Regression (Averaging):

- **Averaging Values**: In regression problems, k-NN takes the **average value** of the target variable from the k-nearest neighbors to make the prediction.
    - **Example**: Suppose you want to predict the price of a house based on the prices of similar houses in the neighborhood. If the prices of the 3-nearest neighbors are $300,000, $310,000, and $290,000, then the predicted price will be:

$$\text{Predicted Price} = \frac{300,000 + 310,000 + 290,000}{3} = 300,000$$

Edge Cases:

- **Weighted Average**: If one neighbor is significantly closer than the others, it might be weighted more heavily in the averaging process. More on **weighted voting** is discussed below.

Impact of k:

- **Small k**:
  - Pros: Captures local patterns, can be highly accurate if the data is clean and noise-free.
  - Cons: Sensitive to noise and outliers. It can lead to **overfitting** where the model memorizes the training data and fails to generalize well to new data.
- **Large k**:
  - Pros: More robust to noise, smoother decision boundaries, and better generalization.
  - Cons: Might oversimplify the problem and ignore local details, leading to **underfitting**.

What Happens When k=1:

- Discuss how k=1 results in predictions based solely on the nearest neighbor. While fast and simple, it is highly sensitive to outliers and noise, leading to **unstable predictions**.

Variations of k-NN

1-Nearest Neighbor (1-NN)

Overview: 1-Nearest Neighbor (1-NN) is the **simplest version** of the k-NN algorithm, where the prediction is made based on the **single nearest neighbor**.

How it Works:

- Given a query point, the algorithm calculates the **distance** between the query point and all points in the dataset.
- The **closest (nearest)** point in the dataset is selected, and the class label or value of this neighbor is used as the prediction.

Application:

- **Classification**: The class label of the nearest point is assigned to the query point.
- **Regression**: The value of the nearest point is used as the prediction.

Advantages and Disadvantages:

- **Advantages**:
  - Simple and computationally efficient for small datasets.
  - Works well when the nearest neighbor is highly representative of the query point.
- **Disadvantages**:
  - **Sensitive to noise**: If the nearest neighbor happens to be an outlier, the prediction will be highly inaccurate.
  - Can lead to **overfitting**, as the prediction is entirely dependent on a single data point.

Example:

- In **image classification**, if k=1, the algorithm looks at the nearest image based on pixel similarity and assigns the query image the same label as this nearest image. While simple, this approach can misclassify images if the nearest neighbor is an anomaly.

Distance-Weighted k-NN

Overview:

- In **distance-weighted k-NN**, closer neighbors have **more influence** on the prediction compared to farther neighbors. This modification helps when the nearest neighbors are not equally distant from the query point.

How it Works:

- The algorithm calculates the distance between the query point and all other points in the dataset.
- Instead of simply using majority voting or averaging, a **weight** is assigned to each neighbor based on its distance to the query point. Closer neighbors are given more weight in the prediction.

Weighted Voting Formula:

$$w_i = \frac{1}{d(x_i, x_q)}$$

Where:

- w_i is the weight assigned to neighbor i,
- $d(x_i, x_q)$ is the distance between neighbor i and the query point x_q.

Application:

- **Classification**: The class label is determined by a **weighted majority vote**, where closer neighbors have more influence on the vote.
- **Regression**: The predicted value is the **weighted average** of the values of the k-nearest neighbors, with closer neighbors contributing more to the final prediction.

Advantages and Disadvantages:

- **Advantages**:
  - Gives more importance to **closer, more relevant neighbors**, reducing the impact of irrelevant or distant points.
  - Useful when the data points are unevenly distributed, as closer points are more likely to be similar to the query point.
- **Disadvantages**:
  - Still computationally expensive as the distance to every data point must be calculated.
  - The effectiveness of weighted voting depends on the distance metric chosen.

Condensed Nearest Neighbor (CNN)

Overview: Condensed Nearest Neighbor (CNN) is used to **reduce the size of the training dataset** by eliminating data points that do not contribute to improving the classification of other points. This method helps to speed up predictions and reduce computational overhead.

How it Works:

- CNN starts by identifying a **core subset** of the original dataset that can classify the rest of the points as accurately as possible.
- The idea is to remove **redundant or unnecessary data points**, i.e., points that are not needed to correctly classify other points in the dataset.

Process:

- Initialize the core subset with a few representative points from the dataset.
- Iteratively add points to this core subset if they are misclassified by the current subset.
- Continue until all points are classified correctly by the core subset, or a stopping criterion is met.

Advantages and Disadvantages:

- **Advantages**:
  - **Reduces dataset size**, making k-NN faster and less computationally expensive during inference.
  - Helps in handling large datasets where computing distances to all points is infeasible.
- **Disadvantages**:
  - It may **not preserve all relevant data points**, leading to reduced accuracy if the condensed subset does not represent the full diversity of the data.
  - It can be **challenging to implement**, as choosing the initial core subset and defining stopping criteria can affect performance.

Example:

- In **pattern recognition**, such as handwriting recognition, CNN can be used to identify a small set of prototype characters that can classify all other handwritten characters accurately. This reduces the number of stored characters and speeds up classification.

Distance Metrics in k-NN

1. Euclidean Distance

Formula:

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

Where:

- x and y are two data points,
- x_i and y_i are the individual feature values of the data points in the i-th dimension,
- n is the number of features (or dimensions).

Explanation:

- **Euclidean distance** measures the straight-line distance between two points in a Euclidean space. It is the most commonly used distance metric in k-NN because of its simplicity and intuitive understanding of distance.
- It calculates the **geometric distance** between two points in a multi-dimensional space, meaning it finds the shortest path between these points.

When to Use:

- **Most suitable for continuous data**: Euclidean distance works best when the features are **continuous** (e.g., height, weight, temperature). It assumes that all dimensions contribute equally to the distance calculation.
- **Sensitive to scale**: One key drawback is that Euclidean distance is **sensitive to the scale** of the features. For example, if one feature is measured in kilometers and another in meters, the feature with larger values will dominate the distance calculation.
    - **Solution**: Feature scaling (such as normalization or standardization) is often necessary before applying Euclidean distance in high-dimensional spaces with different feature units.

2. Manhattan Distance

Formula:

$$d(x, y) = \sum_{i=1}^{n} |x_i - y_i|$$

Where:

- x and y are two data points,
- x_i and y_i are the individual feature values of the data points in the i-th dimension,
- n is the number of features (or dimensions).

Explanation:

- **Manhattan distance** (also known as L1 distance or **taxicab distance**) measures the distance between two points by summing the absolute differences along each dimension.
- Unlike Euclidean distance, which measures the shortest straight-line distance, Manhattan distance measures the distance by traversing **only along the axes** (think of a grid-like structure, such as city blocks, where you can only move horizontally or vertically, not diagonally).

When to Use:

- **Grid-like paths**: Manhattan distance is well-suited for scenarios where movements or paths are constrained to a grid, such as in **robotics** or **urban planning**, where you need to move along streets rather than cutting through blocks.
- **Insensitive to outliers**: It is less affected by outliers compared to Euclidean distance because it does not square the differences in each feature.
- **Works with both continuous and categorical data**: It can be applied in situations where features have a grid-like relationship or for discrete-valued features, but it's still most effective for continuous data.

3. Minkowski Distance

Formula:

$$d(x, y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{1/p}$$

Where:

- x and y are two data points,
- x_i and y_i are the individual feature values of the data points in the i-th dimension,
- n is the number of features (or dimensions),
- p is a parameter that defines the order of the distance metric.

Explanation:

- **Minkowski distance** is a generalization of both **Euclidean** and **Manhattan** distances. By adjusting the value of the parameter p, it behaves like different distance metrics:
  - When p=1, it becomes **Manhattan distance**.
  - When p=2, it becomes **Euclidean distance**.
- For other values of p, it represents a **normed distance** metric that blends the behavior of Euclidean and Manhattan distances.

When to Use:

- **General-purpose distance metric**: Minkowski distance allows you to adjust the p parameter depending on the nature of the data. This makes it flexible and adaptable to different types of problems.
- **Selecting p**: If the features are believed to have strong geometric relationships (e.g., in image data), then **p=2** (Euclidean) is appropriate. If you believe the features are better captured by a grid-like structure, use **p=1** (Manhattan).
- **Continuous data**: This distance metric works well when all the features are continuous.

4. Cosine Similarity

Formula:

$$\cos(\theta) = \frac{x \cdot y}{||x||\,||y||}$$

Where:

- $x \cdot y$ is the dot product of vectors x and y,
- $||x||$ and $||y||$ are the magnitudes (norms) of the vectors x and y,
- θ is the angle between the vectors.

Explanation: Cosine similarity measures the **cosine of the angle** between two vectors, rather than the distance between them. The cosine similarity ranges from **-1** to **1**:

- $\cos(\theta)=1$ indicates the vectors are pointing in the same direction (i.e., they are very similar).
- $\cos(\theta)=0$ indicates the vectors are orthogonal (no similarity).
- $\cos(\theta)=-1$ indicates the vectors are pointing in opposite directions (completely dissimilar).
- The key idea behind cosine similarity is that it measures the **orientation** or **angle** between the vectors, rather than their magnitude or distance. This is especially useful when the magnitude of the vectors (e.g., word counts) does not matter as much as their relative direction.

When to Use:

- **High-dimensional data**: Cosine similarity works best in cases where the data is high-dimensional, and the **magnitude** of the data points (vectors) is not as important as their orientation.
- **Text classification and document comparison**: Cosine similarity is widely used in **Natural Language Processing (NLP)** to compare the similarity of text documents. Each document is represented as a vector of word counts (or term frequencies), and cosine similarity is used to measure the similarity between documents based on the angle between their vectors.
- **Sparse data**: When dealing with sparse data (e.g., text or term frequency vectors where most values are zero), cosine similarity is more effective than distance-based metrics like Euclidean or Manhattan.

Advantages of k-NN

1. **Simple and Intuitive:** k-NN is one of the simplest machine learning algorithms. It is easy to understand conceptually: the algorithm classifies a query point based on the classes of its nearest neighbors.
2. **No Training Phase (Lazy Learning):** k-NN is a **lazy learning** algorithm, meaning there is **no explicit training phase** where a model is built. The entire dataset is stored, and all computations (distance calculation and class assignment) are deferred until the classification of a new data point.
3. **Effective for Small Datasets:** k-NN performs well on **small datasets** where each class is easily distinguishable by the closest neighbors. The algorithm's performance is strong when there is a clear relationship between features and the class labels.
4. **Adaptable to Multi-Class and Multi-Label Problems:** k-NN can naturally extend to **multi-class classification** (handling more than two classes) and **multi-label classification** (where each instance can belong to multiple classes).

Limitations of k-NN

1. **Computationally Expensive:** k-NN requires calculating the distance between the query point and **all data points** in the training set. For large datasets, this becomes computationally expensive, especially when nnn (the number of data points) is large and ddd (the number of features) is high.
2. **Sensitive to Irrelevant Features:** k-NN is heavily influenced by the features used for distance calculation. If there are **irrelevant features** in the dataset, they can distort the distance measurements, making it difficult to identify true neighbors.
3. **Curse of Dimensionality:** As the number of features (**dimensionality**) increases, the distance between points in high-dimensional space becomes **less meaningful**. In high dimensions, all points tend to become equidistant, making it difficult to distinguish between neighbors.

Decision Trees

Introduction to Decision Trees

1. What is a Decision Tree?

- **Definition**:
  - A **supervised learning algorithm** used for both **classification** and **regression** tasks.
  - A decision tree makes decisions by **recursively splitting the dataset** based on specific feature values.
  - At each step, it selects a feature that best splits the data into more homogeneous groups (for classification) or predicts a target value (for regression).
- **Tree-like Structure**:
  - **Root Node**: Represents the entire dataset and the first feature split.
  - **Decision Nodes**: Intermediate points where the data is split based on feature values.
  - **Leaf Nodes**: Represent the final output (a predicted class for classification or a value for regression).

How Decision Trees Work

1. Tree Structure

A **decision tree** is made up of three main types of nodes:

- **Root Node**:
  - The **starting point** of the decision tree, which contains the entire dataset. All subsequent decisions stem from the root node.
- **Decision Nodes**:
  - Points in the tree where the dataset is split based on the value of a specific feature. Each split creates branches leading to further decision nodes or leaf nodes.
  - These nodes represent **questions** or **decisions** about which way to split the data (e.g., "Is the age greater than 30?").
- **Leaf Nodes**:
  - **Terminal nodes** that provide the final output of the decision tree. In a classification task, leaf nodes contain class labels, and in regression tasks, they hold predicted values.
  - Once data reaches a leaf node, no further splitting occurs, and the decision process ends.

2. Splitting the Data

At each **decision node**, the tree splits the data based on a specific feature. The main objective of each split is to create **subsets of the data** that are as **homogeneous** as possible with respect to the target variable (i.e., the class label in classification tasks or the predicted value in regression).

- The splitting process is **recursive**, meaning the tree continues to split the data until a stopping criterion is met.
- For example, in a classification task predicting whether someone will buy a car, the first decision node might split the data based on **income** (e.g., "Is the income greater than $50,000?"). The resulting subsets are then split further using features like age or credit score.

3. Choosing Splitting Criteria

To split the data at each node, decision trees use specific criteria depending on whether they are solving a classification or regression problem.

For Classification: Gini Impurity:

Gini impurity measures the **probability of misclassifying** a randomly chosen data point if it were labeled according to the class distribution in the subset.

$$Gini = 1 - \sum_{i=1}^{n} (p_i)^2$$

Where p_i is the proportion of data points belonging to class i.

- A Gini score of 0 means the subset is **perfectly homogeneous** (all points belong to the same class), while higher values indicate more impurity (i.e., mixed classes).

Information Gain (Entropy):

- **Entropy** measures the **uncertainty** or **disorder** in a dataset. The goal is to reduce uncertainty by splitting the data.
- The formula for entropy is:

$$Entropy = -\sum_{i=1}^{n} p_i \log_2(p_i)$$

Where p_i is the proportion of data points belonging to class i.

  - Information Gain is calculated as the **reduction in entropy** after the data is split. The feature that provides the highest information gain (i.e., reduces uncertainty the most) is chosen for the split.

For Regression:

- **Mean Squared Error (MSE)**:
  - For regression tasks, the splitting criterion aims to minimize the **Mean Squared Error** between the predicted values and actual values.

The formula for MSE is:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Where y_i is the actual value, $\hat{y}_i$ is the predicted value, and n is the number of data points in the subset.

Key Concepts

- **Greedy Algorithm**:
  - Decision trees use a **greedy algorithm** to construct the tree.
  - This means they make the locally optimal decision (best split at each node) without considering the overall structure of the tree.
  - While this method is efficient, it can lead to **overfitting** if the tree grows too deep.
- **Binary Trees**:
  - Most decision trees are **binary trees**, meaning each decision node creates **two branches** by splitting the data into two subsets.
  - Binary splits simplify the structure and make the tree easier to interpret.

Decision Tree Construction

Step 1: Selecting the Best Split

The first step in constructing a decision tree is to select the **best feature** to split the data at each decision node. This is done by evaluating each feature based on certain criteria like **Information Gain** or **Gini Impurity**. The feature that best separates the data (i.e., produces the most homogeneous groups) is chosen.

Step 2: Recursion

Once the best feature has been selected based on Information Gain, Gini Impurity, or MSE, the dataset is **split** into subsets based on the values of this feature. The tree then **recursively applies** the same process to each subset.

- **Recursive Process**:
    1. **Split the data**: Based on the feature with the highest Information Gain or lowest Gini Impurity.
    2. **Recompute** the splitting criteria (Information Gain, Gini, or MSE) for the new subsets.
    3. Continue splitting until a stopping criterion is met.

For example, in a classification task, if the feature "Income" has the highest information gain, the data might be split into two groups: **Income > $50,000** and **Income <= $50,000**. Then, for each group, the process is repeated using the remaining features (e.g., Age, Credit Score).

Stopping Criteria

To avoid excessive growth and overfitting, the construction of a decision tree is controlled by **stopping criteria**:

1. **All records at a node belong to the same class**:
    - If all data points at a node are from the same class, the node becomes a **leaf node**, and no further splits are made.
2. **Maximum Tree Depth**:
    - A predefined **maximum depth** can be set to limit the growth of the tree.
    - Once the tree reaches this depth, no further splits are made, even if subsets are still heterogeneous.
3. **Minimum Samples per Leaf**:
    - A node is not allowed to split if the number of samples in a node is below a certain threshold (e.g., less than 5 samples).
    - This prevents nodes with very few samples from splitting further, which helps reduce overfitting.

Evaluating a Decision Tree

1. Classification Evaluation Metrics

- **Accuracy**
- **Precision, Recall, and F1-Score**
- **Confusion Matrix**

2. Regression Evaluation Metrics

- **Mean Squared Error (MSE)**: The average of the squared differences between the actual and predicted values.
  - **Usage**: Penalizes larger errors more than smaller errors.
- **Mean Absolute Error (MAE)**: The average of the absolute differences between actual and predicted values.
  - **Usage**: Less sensitive to outliers than MSE.

3. Cross-Validation

- **k-Fold Cross-Validation**: A technique to evaluate model performance by splitting the data into k subsets (folds). The model is trained on k−1 folds and tested on the remaining fold, repeating the process k times.
  - **Importance**: Helps assess the model's ability to generalize to new data and prevent **overfitting** by using different portions of the dataset for training and validation.
  - **Why it Matters**: Ensures that the decision tree's performance is **not overly tuned** to one subset of the data, making the model more robust and reliable.

Pruning Techniques in Decision Trees

Why Pruning is Necessary

Overfitting:

- **Overfitting** occurs when a decision tree becomes too complex by fitting not only the **true patterns** in the data but also the **noise**. This happens when the tree is allowed to grow too deep, creating many branches and leaf nodes that reflect nuances in the training data but do not generalize well to new, unseen data.
- Overfitting leads to poor performance on **test data** or real-world scenarios, where the model fails to correctly classify or predict because it has become too specific to the training data.

How Pruning Helps:

- **Pruning** is the process of **simplifying the decision tree** by removing branches and leaf nodes that contribute little to improving the model's accuracy. By doing this, pruning reduces the tree's complexity, allowing it to better generalize to unseen data and thus reduce overfitting.

A. Pre-Pruning (Early Stopping)

- **What is Pre-Pruning?**
  - Pre-pruning (also known as **early stopping**) involves **stopping the growth of the tree early**, before it becomes too complex.
  - This prevents the tree from overfitting by limiting how deep it can grow or by controlling other factors such as the number of samples in a node.
- **How it Works**:
  - **Stopping criteria** are set before training begins. These criteria determine when to stop the tree from growing further. Pre-pruning decisions are made while building the tree, based on:
    - **Maximum Depth**: Limits the number of levels the tree can grow. For example, if a tree's depth is restricted to 3, no node can split beyond the third level.
    - **Minimum Samples Required to Split**: Limits splitting based on the number of samples available in a node. For example, you might prevent further splits if a node contains fewer than 5 samples.
    - **Minimum Samples per Leaf**: Requires each leaf node to contain a minimum number of samples, ensuring that leaf nodes are not too small.
- **Advantages**:
  - **Efficiency**: The tree-building process is faster since it avoids unnecessary splits.
  - **Overfitting Prevention**: By restricting the size of the tree, pre-pruning reduces the chances of capturing noise in the data.
- **Disadvantages**:
  - **Risk of Underfitting**: Since pre-pruning halts tree growth before exploring all splits, it may stop too early and miss important patterns in the data.

B. Post-Pruning

- **What is Post-Pruning?**
  - Post-pruning involves **growing the decision tree to its full depth** and then **removing branches** that do not improve the model's performance. Unlike pre-pruning, the entire tree is first constructed, and pruning is done after the full tree is built.
- **How it Works**:
  - The tree is initially allowed to grow to its maximum possible size, capturing all patterns and even noise in the data.
  - Afterward, branches and leaf nodes that contribute little to improving the model's performance are pruned. The goal is to **prune back nodes that don't add value**.
  - **Evaluation Metrics**: Post-pruning typically uses **cross-validation** or a **validation set** to evaluate whether removing certain branches leads to better model performance.
  - **Process**:
    1. Evaluate the tree's performance (e.g., using accuracy, MSE, or Gini).
    2. Remove the least significant branches that lead to minimal improvement.
    3. Re-evaluate the tree and keep pruning until the model's performance on the validation set starts to degrade.
- **Advantages**:
  - **More Accurate**: Post-pruning allows the tree to fully explore all splits before deciding what to remove, reducing the risk of missing important splits.
  - **Better Generalization**: By pruning unnecessary branches, the tree becomes more general and less likely to overfit.
- **Disadvantages**:
  - **Complexity**: Post-pruning is computationally more expensive because it requires building the full tree and then testing different pruned versions to see which performs best.
  - **Time-Consuming**: Pruning each branch and re-evaluating the tree can be time-intensive, especially for large datasets.

C. Cost Complexity Pruning (CCP)

- **What is Cost Complexity Pruning?**
  - **Cost Complexity Pruning (CCP)** is a specific post-pruning technique that balances the trade-off between the **complexity** of the tree and its ability to fit the data. CCP adds a **penalty** for having too many branches, encouraging simpler trees that generalize better.
- **How it Works**:
  - CCP minimizes the following cost function:

$$R_\alpha(T) = R(T) + \alpha|T|$$

Where:

- $R(T)$ is the **error rate** of the tree,
- $|T|$ is the number of leaf nodes (a measure of tree complexity),
- α is a **penalty parameter** that controls how much weight is given to the complexity of the tree.

As α increases, the algorithm prunes more aggressively, reducing the size of the tree.

The algorithm prunes branches that **minimally reduce the training error** but add significant complexity to the tree.

- **Advantages**:
  - **Balances complexity and performance**: CCP ensures the tree is not too deep, by controlling the trade-off between how well the tree fits the data and its complexity.
  - **Flexibility**: The tuning of the α\alphaα parameter allows for flexible control over the level of pruning, making the tree simpler without sacrificing too much accuracy.
- **Disadvantages**:
  - **Choosing the right α**: Determining the optimal penalty α\alphaα requires careful cross-validation, which can be computationally expensive.

Advantages of Decision Trees

a. Easy to Understand and Interpret: Decision trees are one of the most intuitive and interpretable machine learning algorithms. They represent decision-making in a **graphical, flowchart-like structure**, making it easy for non-technical stakeholders to understand how the model reaches its decisions.

Example: In a medical diagnosis system, a decision tree might classify patients based on symptoms, with each decision (e.g., "Is age > 50?") clearly laid out, making the model highly interpretable.

b. No Need for Feature Scaling: Unlike algorithms like k-NN or SVM, decision trees are **unaffected by the scale** of features because they rely on **splitting the data** based on feature values, not on distances between data points.

Example: A feature like "income" can be measured in thousands, while "age" is measured in years, and this will not affect the tree's decision-making.

c. Handles Both Numerical and Categorical Data: Decision trees can naturally handle **both continuous (numerical)** and **discrete (categorical)** data types without the need for one-hot encoding or standardization. The algorithm treats categorical values just as it would continuous values by splitting based on the distinct categories.

Example: A decision tree can easily split on a categorical variable like "City" or a numerical variable like "Age" within the same model.

d. Non-parametric: Decision trees are **non-parametric**, meaning they make no assumptions about the underlying data distribution. This flexibility allows them to perform well on a variety of datasets, including those that do not follow standard assumptions like linearity or normal distribution.

Example: Whether your data is normally distributed or highly skewed, decision trees can still be applied without needing to transform the data.

Limitations of Decision Trees

a. Overfitting: One of the most common issues with decision trees is **overfitting**. As the tree grows deeper, it starts capturing **noise and irregularities** in the training data, making it fit too closely to the training data and perform poorly on unseen data.

Example: If a tree is allowed to split endlessly, it may reach pure leaf nodes with just one or two data points, effectively memorizing the training set. This makes the tree highly specific to the training data, reducing its generalization ability.

b. Instability: Decision trees can be highly **unstable** because small changes in the training data (e.g., removing or adding a few samples) can lead to a **completely different tree structure**. This is because the algorithm is greedy, choosing the best split at each step, which may not lead to the globally optimal solution.

Example: A small perturbation in data, such as a single feature value changing slightly, can cause a different tree structure and entirely different predictions, making the model sensitive to small variations.

c. Bias Toward Splits on Categorical Variables: Decision trees tend to **favor splits** on categorical variables with many levels (e.g., zip codes), which can introduce **bias**. The algorithm may choose such features for a split because they inherently create more branches, even if they are not the most informative feature.

Example: A tree might split on a variable like "Zip Code" just because it has many categories, even though this split might not be as predictive as a simpler feature like "Age" or "Income."

d. Limited Expressiveness: Decision trees create **orthogonal decision boundaries** because they split the data along feature axes. This can be limiting in cases where the true decision boundary between classes is more complex or non-linear.

Example: If the relationship between features is curved or diagonal, a decision tree may struggle to capture this, as it can only split data vertically or horizontally, potentially missing more complex patterns.

Applications of Decision Trees

Decision trees are widely used across multiple industries due to their versatility. Some prominent applications include:

a. Medical Diagnosis

- **Use Case**: Decision trees help classify patients based on symptoms and medical test results to predict diseases or health conditions.
- **Example**: Classifying whether a patient has a heart disease based on features like age, cholesterol level, and blood pressure. The decision tree makes predictions by splitting the data based on these symptoms and lab results.

b. Customer Segmentation

- **Use Case**: Businesses use decision trees to identify different types of customers based on their purchasing behavior and demographic information.
- **Example**: A marketing company might use a decision tree to segment customers based on income, age, and spending patterns. This helps in targeting specific products to relevant customer groups.

c. Fraud Detection

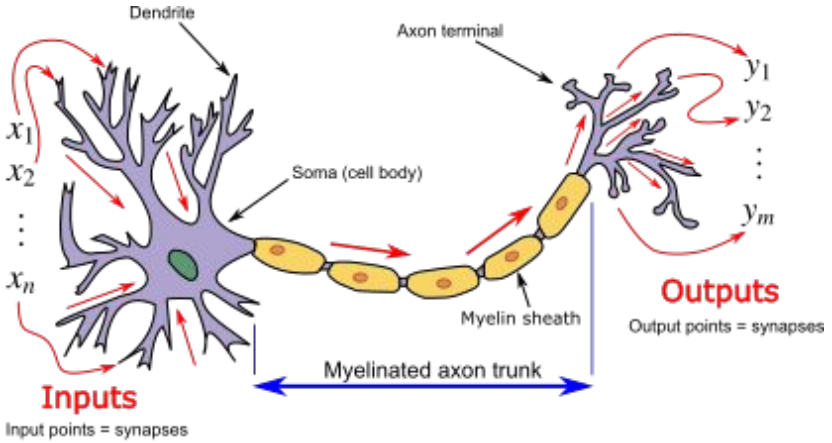
- **Use Case**: Decision trees are used to detect fraudulent transactions based on transaction attributes such as transaction amount, location, and frequency.
- **Example**: A decision tree may flag unusual credit card transactions (based on location, time, or amount) as potentially fraudulent by identifying patterns in past fraudulent transactions.

Neural Network Basics

Biological Neuron

A typical neuron consists of four major parts:

1. **Soma (Cell Body)**:
   - The soma is the central part of the neuron that contains the cell's nucleus.
   - **Function**: The soma integrates incoming signals received from the dendrites.
2. **Dendrites**:
   - Dendrites are branching extensions from the cell body, resembling tree-like structures.
   - **Function**: Dendrites act as the neuron's input receivers.
3. **Axon**:
   - The axon is a long, thin extension from the soma that transmits electrical signals away from the cell body towards other neurons or effector cells (like muscles).
   - **Function**: The axon conducts the electrical signal (known as an **action potential**) from the soma to other neurons. At the end of the axon are terminal branches that connect to other neurons at **synapses**.
4. **Synapses**:
   - The synapse is the small gap or junction between the terminal branches of one neuron and the dendrites or soma of another neuron.
   - **Function**: The synapse is where the neuron communicates with other neurons.

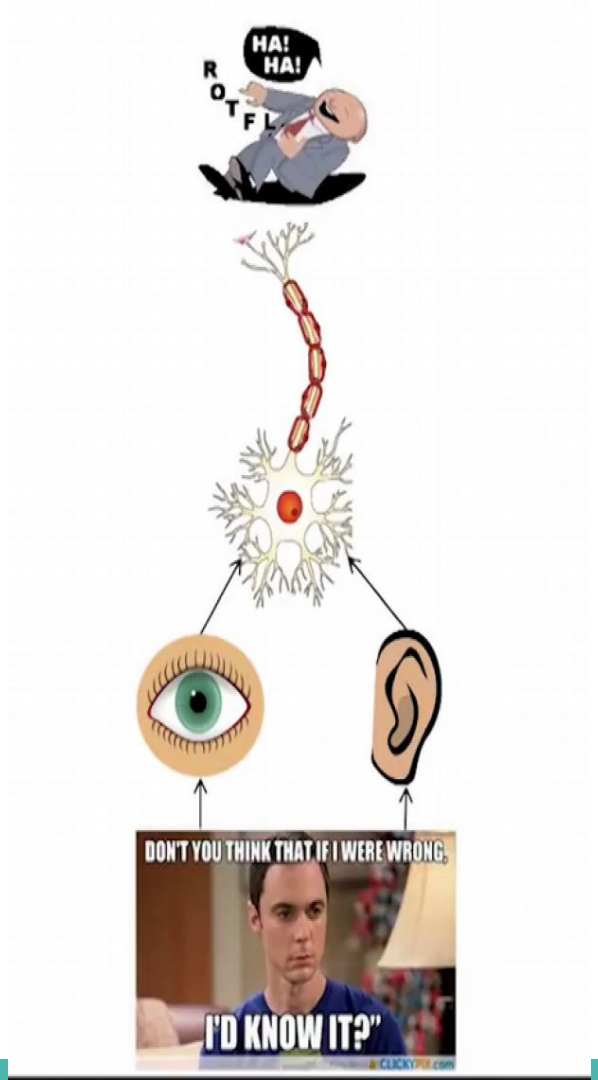


Here we have a cartoonish illustration of how a neuron works.

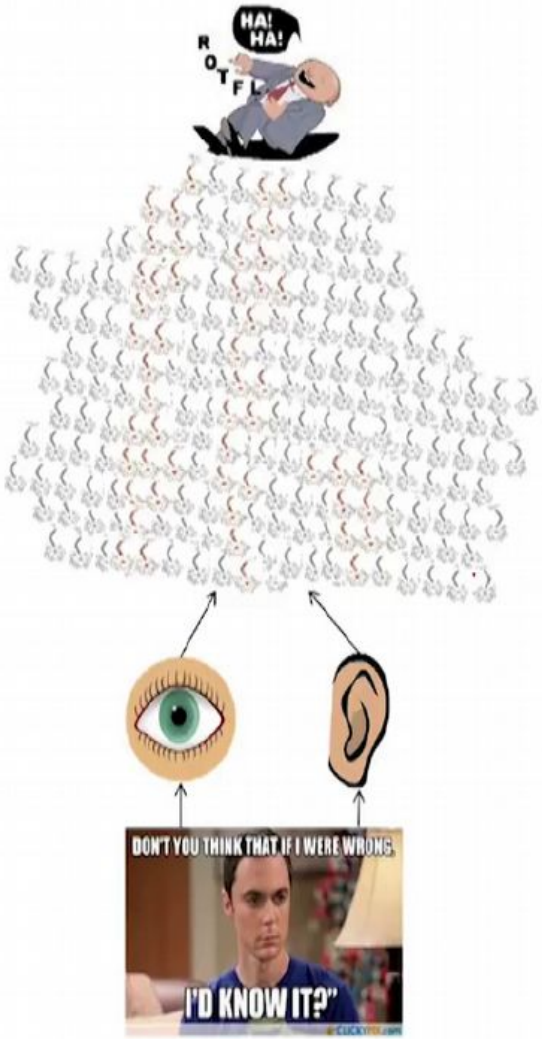
Our sense organs interact with the outside world.

They relay information to the neurons.

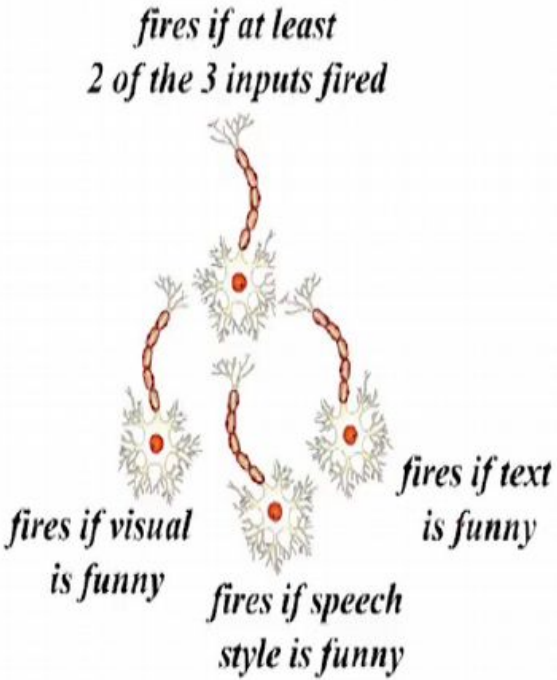
The neurons (may) get activated and produces a response (laughter in this case).



- Ofcourse, in reality, it is not just a single neuron which does all this.
- There is a massively parallel interconnected network of neurons.
- The sense organs relay information to the lowest layer of neurons.
- Some of these neurons may fire (in red) in response to this information and in turn relay information to the other neurons they are connected to.
- These neurons may also fire (again, in red) and the process continues eventually resulting in a response (laughter in this case)
- An average human brain has around 10^{11} (100 billion) neurons!

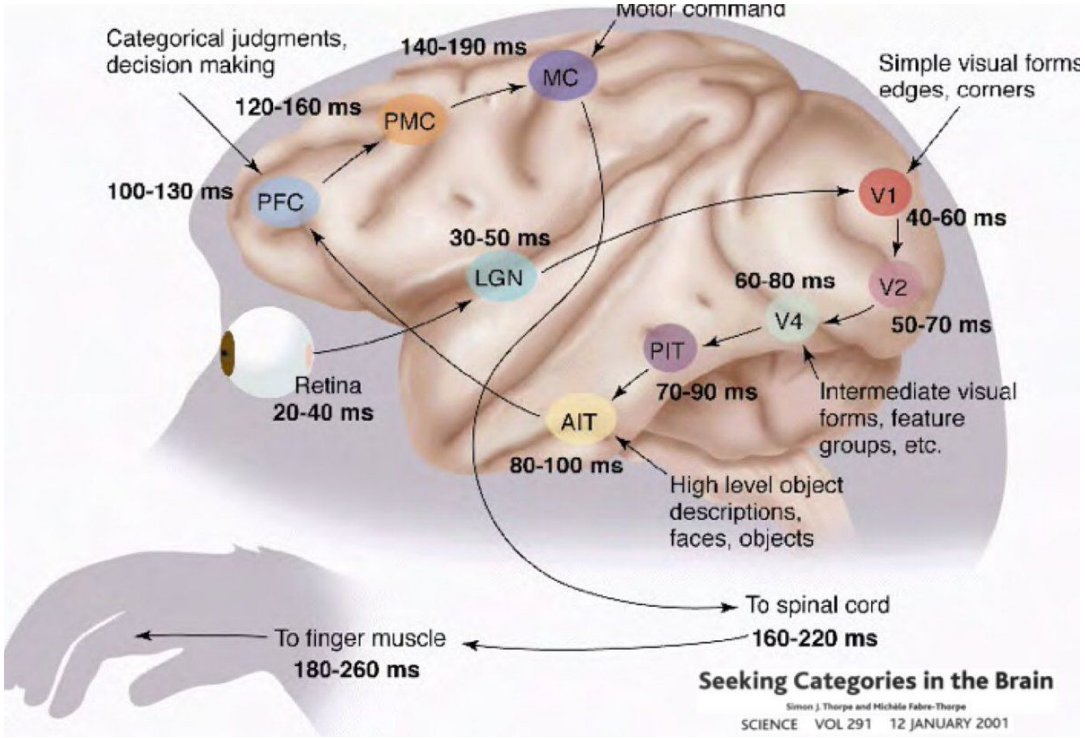


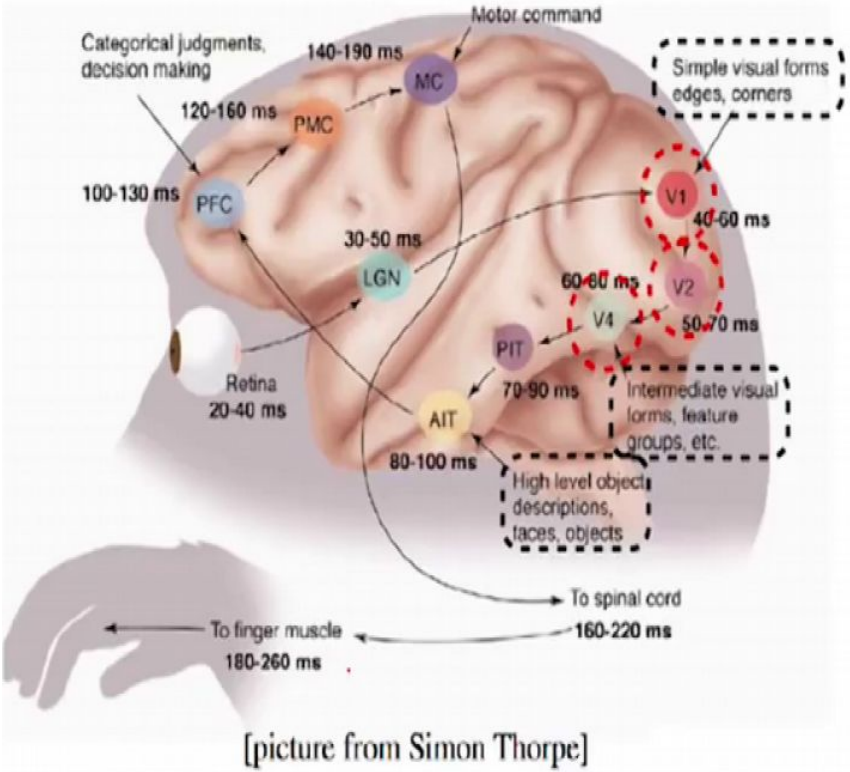
- This massively parallel network also ensures that there is division of work
- Each neuron may perform a certain role or respond to a certain stimulus.



A simplified illustration

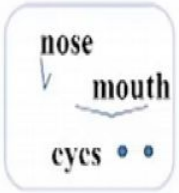
- The neuron in the brain are arranged in a hierarchy
- We illustrate this with the help of visual cortex (part of the brain) which deals with processing visual information.
- Starting from the retina, the information is relayed to several layers (follow the arrows)
- We observe that the layer V1, V2 to AIT form a hierarchy (from identifying simple visual forms to high level objects)





[picture from Simon Thorpe]

Layer 1: detect edges & corners



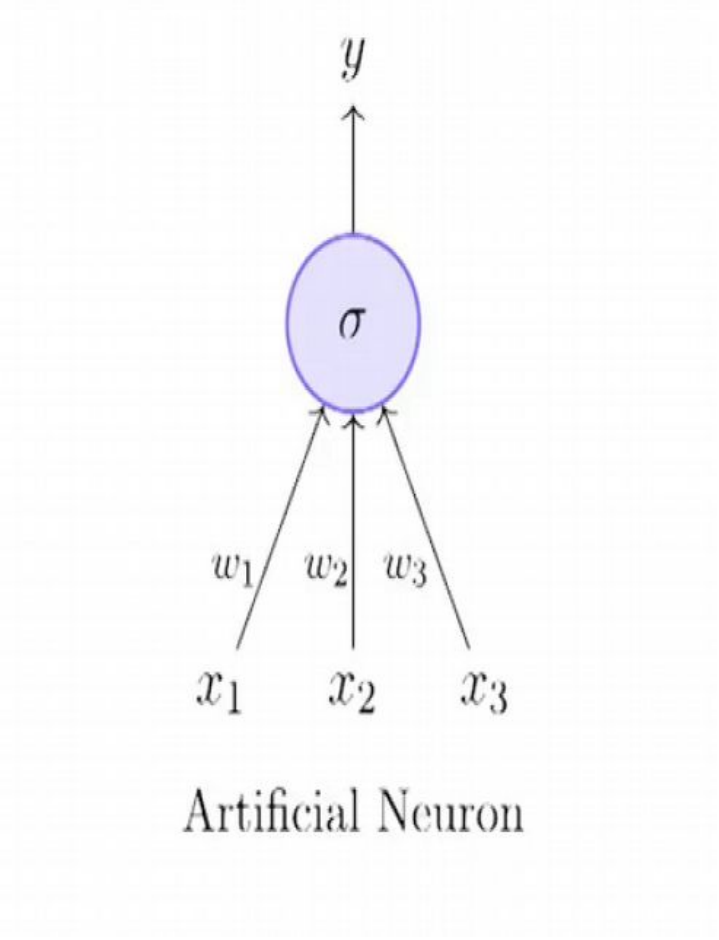
Layer 2: form feature groups

Layer 3: detect high level objects, faces, etc.

Sample illustration of hierarchical processing

Artificial Neuron

The most fundamental unit of a deep neural network is called an artificial neuron.



McCulloch Pitts Neuron

McCulloch (neuroscientist) and Pitts (logician) proposed a highly simplified computational model of the neuron (1943)

All the inputs are binary.

MP neuron is divided into two parts: *g* aggregates the inputs and the function *f* takes a decision based on this aggregation

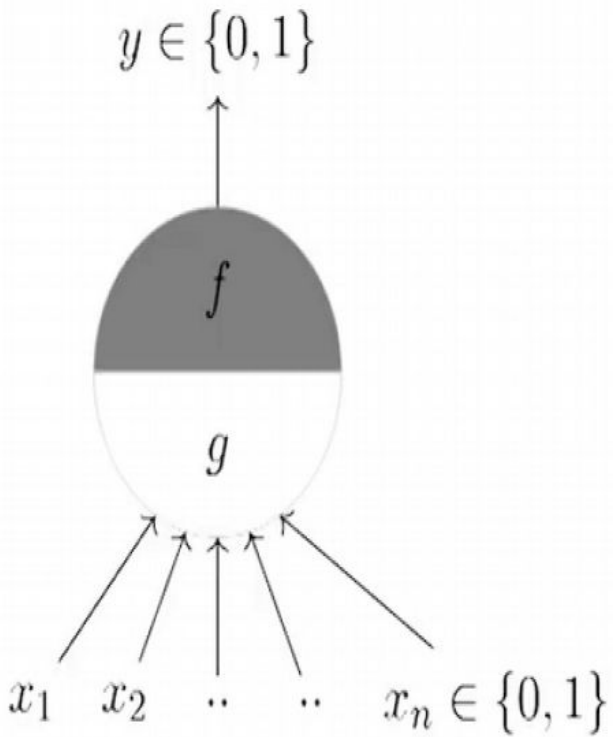
The output is again Boolean.

If it's 0, then neuron does not fire. If it's 1, the neuron fires.

So, suppose I am trying to make a decision whether I should watch a movie or not. So, x_1 could be is the genre of the movie thriller.

Similarly, there could be another variable say x_2 which says is the actor Matt Damon.

So, these are all various such factors that I could take is the director Christopher Nolan, and so on.



$x_1 = isActorDamon$

$x_2 = isGenreThriller$

$x_3 = isDirectorNolan$

The inputs can be excitatory or inhibitory

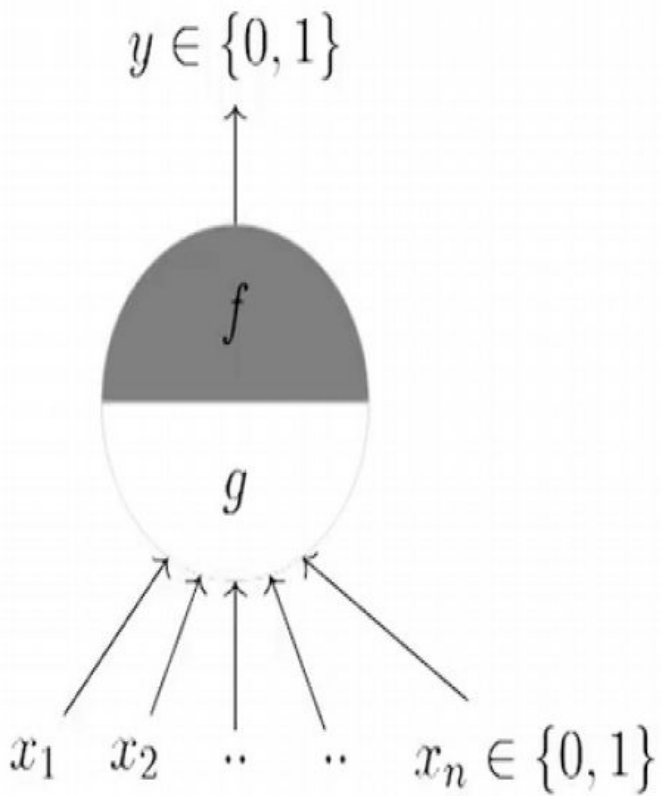
Inhibitory: inhibitory inputs irrespective of what else is on in your input features. If this input is on, your output is always going to be 0. That means, the neuron is never going to fire.

y = 0 if any x_i is inhibitory, else

Excitatory: is not something which will cause the neuron to fire on its own, but it combine with all the other inputs that you have seen could cause the neuron to fire

$$g(x_1, x_2, ..., x_n) = g(\mathbf{x}) = \sum_{i=1}^{n} x_i$$

$$y = f(g(\mathbf{x})) = 1 \quad if \quad g(\mathbf{x}) \geq \theta$$
$$= 0 \quad if \quad g(\mathbf{x}) < \theta$$



$\boldsymbol{\theta}$ is called the thresholding parameter
This is called Thresholding Logic

Perceptron

Frank Rosenblatt, an American psychologist, proposed the classical perceptron model (1958)

A more general computational model than McCulloch-Pitts neurons

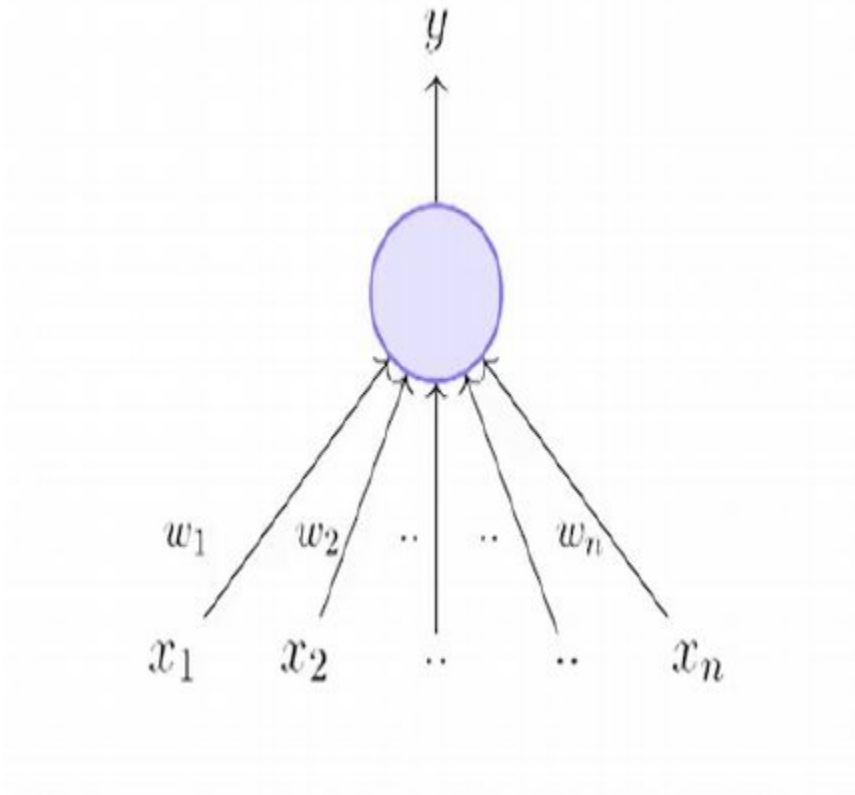
Main differences: Introduction of numerical weights for inputs and a mechanism for learning these weights.

Inputs are no longer limited to boolean values

Refined and carefully analyzed by Minsky and Papert (1969) - their model is referred to as the perceptron model here

It is a type of single-layer neural network.

The perceptron operates as a linear classifier used for binary classification tasks.



- The perceptron takes **multiple inputs** and multiplies each by a corresponding **weight**.
- The inputs are combined into a **weighted sum**, and a **bias** is added.
- This weighted sum is passed through an **activation function** (usually a **step function**).
- The activation function outputs a **1** or **0**, depending on whether the weighted sum crosses a **threshold**.

Inputs: $x_1, x_2, ..., x_n$.

Weights: $w_1, w_2, ..., w_n$ applied to each input.

Weighted sum: $z = w_1x_1 + w_2x_2 + \dots + b$, where b is the bias.

Activation function (Step function):

$$y = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

Output: Binary result — **1** (positive class) or **0** (negative class).

Structure of a Single-Layer Perceptron

The perceptron is a basic unit of a neural network, consisting of the following key components:

a. Inputs (x_1, x_2, …, x_n)

- The perceptron takes **multiple input features**, which are represented as x_1, x_2, …,x_n.
- These inputs represent the features of the data to be classified.
  Example:
  - For a student exam classification task:
    - x_1: Number of hours studied.
    - x_2: Number of practice exams taken.

b. Weights (w_1, w_2, …, w_n)

- Each input has an associated **weight**. Weights determine the importance of each input in making the classification decision.
- These weights are learned during the training process.
  Example:
  - w_1: Weight for hours studied.
  - w_2: Weight for the number of practice exams.

c. Bias (b)

- The **bias** term is a constant that is added to the weighted sum of the inputs. The bias allows the perceptron to shift the decision boundary, helping the model adjust its predictions.
- Bias is particularly important when the data points are not centered around the origin.
  Example:
  - In the student exam classification task, the bias term can help adjust the threshold for determining whether a student passes or fails.

d. Summation of Inputs

- The perceptron takes all input features and multiplies them by their corresponding weights.
- Then, the bias b is added to the weighted sum of the inputs.
  The **weighted sum** is calculated as:

$$z = w_1x_1 + w_2x_2 + \cdots + w_nx_n + b$$

  Example:
  - Suppose the weights and inputs are:
    - $x_1 = 10$ (hours studied).
    - $x_2 = 3$ (practice exams taken).
    - $w_1 = 0.8$ (importance of hours studied).
    - $w_2 = 0.5$ (importance of practice exams).
    - Bias $b = -5$.
- The weighted sum is:
  $z = (0.8\times10) + (0.5\times3) + (-5) = 8 + 1.5 - 5 = 4.5$

Supervised Learning in the Perceptron

- **Supervised Learning**:
  - The perceptron is trained using **labeled data** where both the input features and their correct output labels are provided. The perceptron learns from these **input-output pairs** to minimize classification errors.
  - Each training example consists of an input vector $x = [x_1, x_2, \ldots, x_n]$ and its corresponding correct class label y_{true}, which can be either 0 or 1 in binary classification.
- **Example**:
  - Given input features (e.g., hours studied, number of practice exams taken), the correct label might be "Pass" (1) or "Fail" (0) for each student.

Perceptron Learning Rule

The **learning rule** for the perceptron is based on adjusting the weights and bias whenever the perceptron makes an incorrect prediction. The goal is to reduce the error in classification for future predictions by modifying the weights in proportion to the error.

a. Correct Prediction: No Weight Change

- If the perceptron's predicted label y_{pred} matches the true label y_{true}, no change is made to the weights or bias.
  Example:
  - If the perceptron predicts **Pass (1)** and the true label is also **Pass (1)**, the perceptron has made the correct prediction, and the weights remain the same.

b. Incorrect Prediction: Update Weights

- If the perceptron's prediction is incorrect ($y_{pred} \neq y_{true}$), the weights are updated according to the difference between the true label and the predicted label.

Weight Update Rule:

- For each weight w_i, the update rule is:

$$w_i \leftarrow w_i + \eta(y_{\text{true}} - y_{\text{pred}})x_i$$

Where:

- w_i: The weight associated with the input feature x_i.
- η: The **learning rate**, a small positive constant that controls how much the weights are adjusted in response to the error.
- y_{true}: The correct label (either 0 or 1).
- y_{pred}: The perceptron's predicted label.
- x_i: The value of the input feature associated with weight w_i.

Explanation:

- If the prediction is **incorrect**, the weight is adjusted by an amount proportional to both the learning rate and the input feature. The adjustment moves the weights in the direction that would reduce the error on this particular example.

Example:

- Suppose the perceptron predicts **Fail (0)** for a student, but the true label is **Pass (1)**.
- Input feature $x_1 = 10$ (hours studied) and weight $w_1 = 0.8$.
- The learning rate $\eta=0.1$
- The weight update for w_1 would be:

$$w_1 \leftarrow w_1 + \eta\,(y_{true} - y_{pred})\,x\,1$$

$$w_1 \leftarrow 0.8 + 0.1\,(1-0) \times 10 = 0.8 + 1 = 1.8$$

- The weight w_1 is increased because the perceptron under-predicted (it predicted 0 when the true label was 1).

Bias Update

Just as the weights are updated when the perceptron makes an incorrect prediction, the **bias** is also updated to shift the decision boundary. The bias update rule follows a similar pattern to the weight update rule.

Bias Update Rule:

$$b \leftarrow b + \eta\,(y_{true} - y_{pred})$$

- b: The bias term.
- η: The learning rate.
- y_{true}: The correct label (either 0 or 1).
- y_{pred}: The perceptron's predicted label.

Explanation:

- If the perceptron under-predicts (predicts 0 when it should predict 1), the bias is **increased** to make it easier for the perceptron to output 1 in the future.
- If the perceptron over-predicts (predicts 1 when it should predict 0), the bias is **decreased** to make it harder for the perceptron to output 1.

Example:

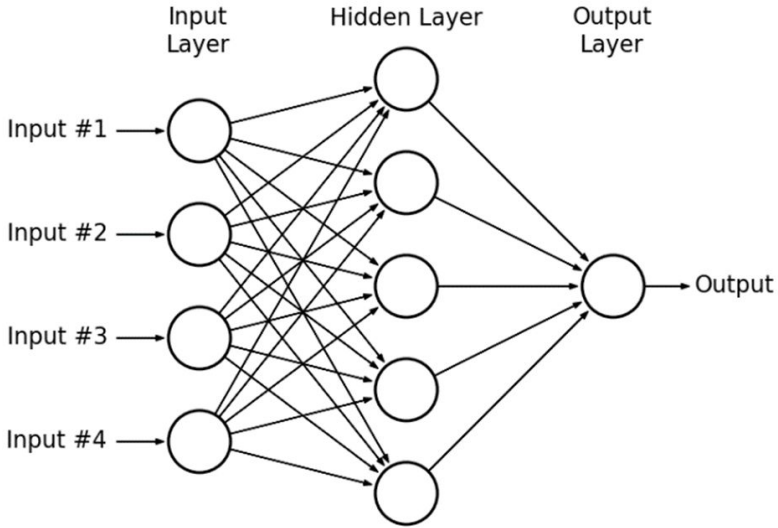
- If the bias $b = -1$ and the perceptron incorrectly predicts **Fail (0)** when the true label is **Pass (1)**, and the learning rate $\eta = 0.1$, the bias is updated as follows:

$$b \leftarrow -1 + 0.1\,(1-0) = -1 + 0.1 = -0.9$$

- The bias is increased, making it easier for the perceptron to predict 1 next time.

What is a Multilayer Perceptron (MLP)?

- The **Multilayer Perceptron (MLP)** is a type of **feedforward artificial neural network (ANN)**.
- In an MLP, data flows in one direction—**forward**—from the input layer, through the hidden layer(s), to the output layer, without any feedback loops.
- Unlike a **single-layer perceptron**, which only has an input and output layer, the MLP includes one or more **hidden layers**, enabling it to learn more complex patterns and relationships.



Fully Connected Layers

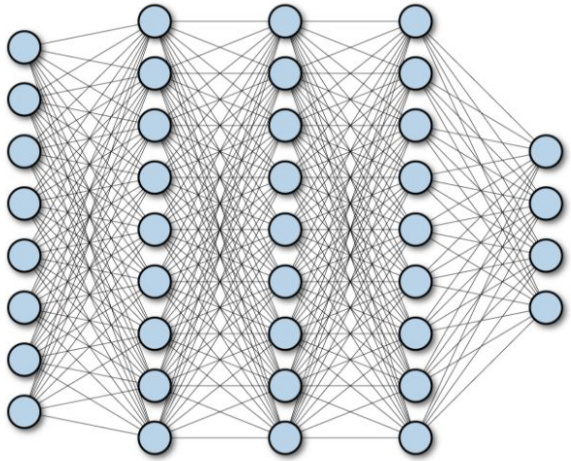
Fully connected means that each neuron in one layer is connected to every neuron in the following layer. This architecture ensures that information flows smoothly from the input to the output layer.

Connections:

- In an MLP, every neuron in one layer applies a **weighted sum** of the outputs of all neurons from the previous layer and then applies an **activation function** (such as ReLU, sigmoid, or tanh) to introduce non-linearity.

Example:

- If a hidden layer has 4 neurons, and the next hidden layer has 5 neurons, each of the 4 neurons will be connected to each of the 5 neurons in the next layer, resulting in 20 connections.



Layers of the Multilayer Perceptron: In detail

An MLP is structured into three primary types of layers, each playing a critical role in processing and transforming input data:

1. Input Layer:

- **Function**: The input layer is responsible for receiving the raw data or input features that will be processed by the network.
- **Number of Neurons**: The number of neurons in the input layer corresponds to the number of input features in the dataset. Each neuron in the input layer takes one feature from the input data.
  Example:
  - In an image classification problem, each pixel from the image (e.g., 28x28 pixels for MNIST) would be an input feature, resulting in 784 input neurons (one for each pixel).

In NLP tasks, the input to the MLP often comes from the **text** that needs to be classified. However, raw text cannot be fed directly into an MLP, so it first needs to be converted into numerical representations.

Common Techniques for text representation:

- **Bag of Words**: Represents a text as a vector of word counts or term frequency-inverse document frequency (TF-IDF).
- **Word Embeddings**: Converts words into dense vectors that capture semantic meaning (e.g., using pre-trained embeddings like Word2Vec or GloVe).

Example:

- Consider the sentence: "**This movie is amazing!**"
- Using **word embeddings**, each word in the sentence ("This," "movie," "is," "amazing") is transformed into a vector (e.g., 300-dimensional vector). These word vectors are concatenated or averaged to form a single vector representing the entire sentence.

Input Layer:

- The **number of neurons** in the input layer will correspond to the dimensionality of the input vector (e.g., if the sentence is represented as a 300-dimensional word embedding, there will be 300 neurons in the input layer).

2. Hidden Layer(s):

- **Function**: Hidden layers contain neurons that apply **non-linear transformations** to the inputs. These transformations enable the MLP to learn complex patterns and relationships that are not linearly separable.
- **Number of Neurons**: The number of neurons in the hidden layers can vary depending on the complexity of the problem. More neurons in a hidden layer or more hidden layers increase the capacity of the network to learn intricate patterns.
  Non-Linearity: Without hidden layers, an MLP would behave like a simple linear classifier. The non-linear activation functions in the hidden layers allow the model to solve more complex tasks by introducing non-linearity into the network's decision-making process.

The hidden layers in an MLP transform the input representations (in this case, the sentence vector) to learn deeper, more complex patterns in the data.

- The neurons in the hidden layers apply **non-linear transformations** to the input, allowing the MLP to capture complex linguistic relationships, such as the interaction between words (e.g., "movie" and "amazing" indicate positive sentiment).

Example:

- The hidden layers can help the model learn that words like "amazing," "great," or "fantastic" are highly correlated with positive sentiment, even if these words weren't explicitly seen in the training data.

Number of Neurons:

- The number of neurons in the hidden layers is flexible, depending on the complexity of the task. For example, 128 or 256 neurons may be used in a hidden layer to capture the nuances in the text.

3. Output Layer:

- **Function**: The output layer provides the final result based on the transformations performed by the hidden layers. The output could be a predicted class for classification problems or a continuous value for regression problems.
- **Number of Neurons**:
    - **For classification**: If the task involves classifying objects into multiple categories (e.g., recognizing digits from 0 to 9), the output layer will have one neuron for each class.
    - **For regression**: If the task involves predicting a continuous value, there may be a single output neuron representing the predicted value.

The output layer in an MLP produces the final prediction, which, in this case, is the **sentiment** of the text.

Number of Neurons:

- For binary classification (positive or negative sentiment), there will be **one neuron** in the output layer with a **sigmoid activation function**. This neuron will output a value between 0 and 1, representing the probability of the sentence being positive or negative.

Example:

- For the sentence "**This movie is amazing!**", the output neuron might output a value close to 1, indicating **positive sentiment**.

How MLPs Overcome The Limitation

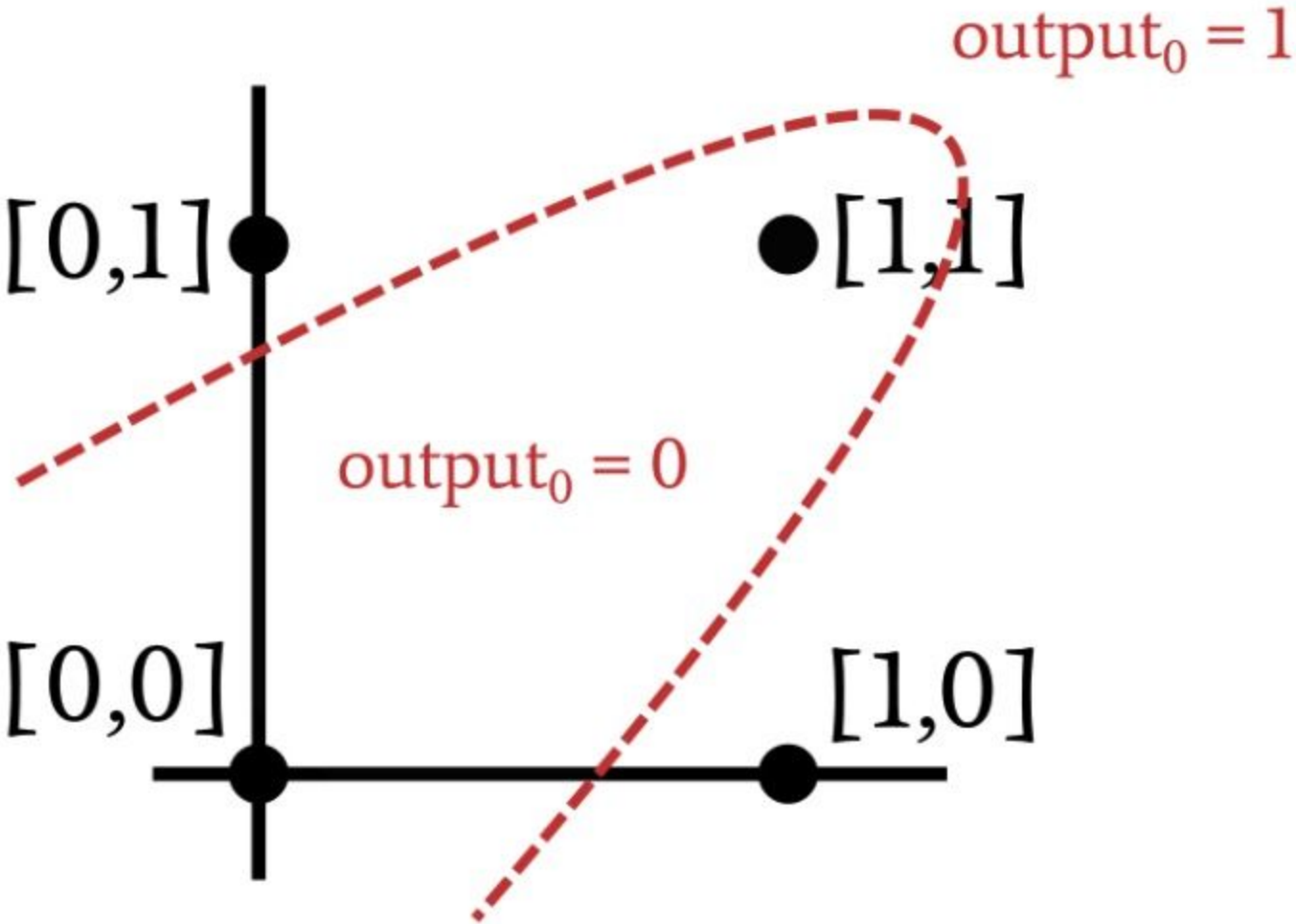
The key difference between a **single-layer perceptron** and an **MLP** is the introduction of **hidden layers** and **non-linear activation functions**. These components allow MLPs to solve **non-linearly separable problems**, like XOR, by applying complex transformations to the input data.

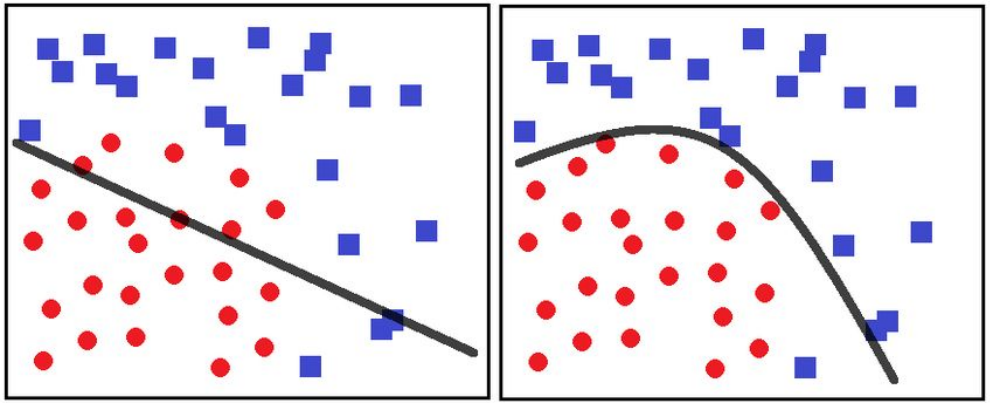
a. Hidden Layers and Non-Linearity

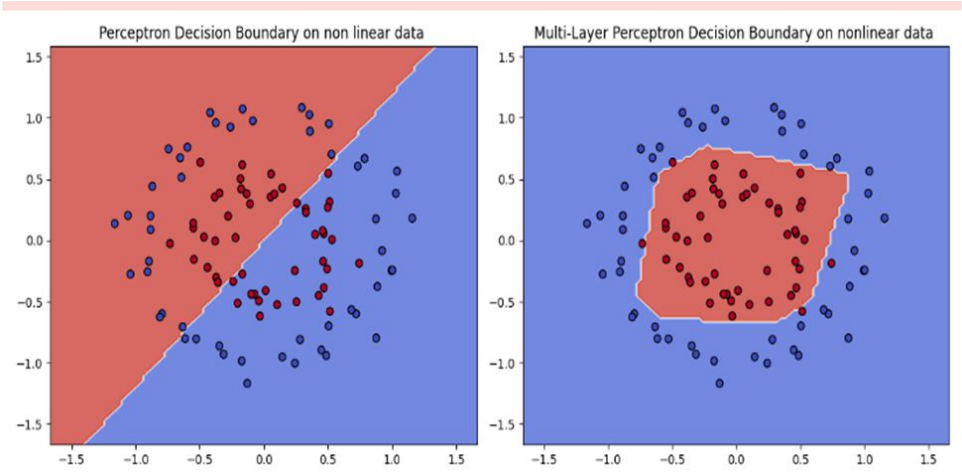
- **Hidden layers** play a crucial role in transforming input data in a way that makes it linearly separable in a new, higher-dimensional space.
- In an MLP, each hidden layer applies a **non-linear transformation** to the data, meaning it can model more complex decision boundaries than a straight line.

Non-Linearity in MLPs:

- The **activation functions** in the hidden layers introduce non-linearity. This allows the MLP to create **curved decision boundaries** that can separate data points that are not linearly separable in the original input space.
  Example:
  - If you're trying to classify circular data, a single-layer perceptron might fail because a straight line cannot separate the data points. An MLP, however, can create a curved boundary using non-linear transformations.


output$_0$ = 1
[0,1]
[1,1]
output$_0$ = 0
[0,0]
[1,0]




Perceptron Decision Boundary on non linear data
Multi-Layer Perceptron Decision Boundary on nonlinear data
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5

Forward Propagation in Multilayer Perceptron

Forward propagation involves moving the inputs through the MLP, layer by layer, to produce the final output. Each layer in the network applies transformations to the data, enabling the model to learn patterns and make predictions.

1. **Input Layer**:
   - The input layer receives the raw input data (e.g., features from an image, text, or numerical values) and passes it to the next layer (the first hidden layer).
2. **Hidden Layers**:
   - Each hidden layer consists of neurons that compute **weighted sums** of the inputs from the previous layer and apply **activation functions** to introduce non-linearity.
3. **Output Layer**:
   - The output layer produces the final prediction, which could be a class label in classification tasks or a numerical value in regression tasks.

Activation Functions

Activation functions are **mathematical functions** applied to the output of each neuron in a neural network. They transform the weighted sum of inputs to produce the final output of a neuron, which is then passed to the next layer or, in the case of the output layer, used as the network's prediction.

Key Points:

- The activation function introduces **non-linearity** into the neural network, which allows it to model complex patterns.
- Activation functions are applied after the neuron calculates the **weighted sum** of its inputs and adds a **bias** term.

Mathematically, if a neuron computes a weighted sum of inputs $z = w_1x_1 + w_2x_2 + \cdots + w_nx_n + b$, the activation function $f(z)$ transforms z into the neuron's final output:

$$y = f(z)$$

Ideal Activation Functions: Characteristics

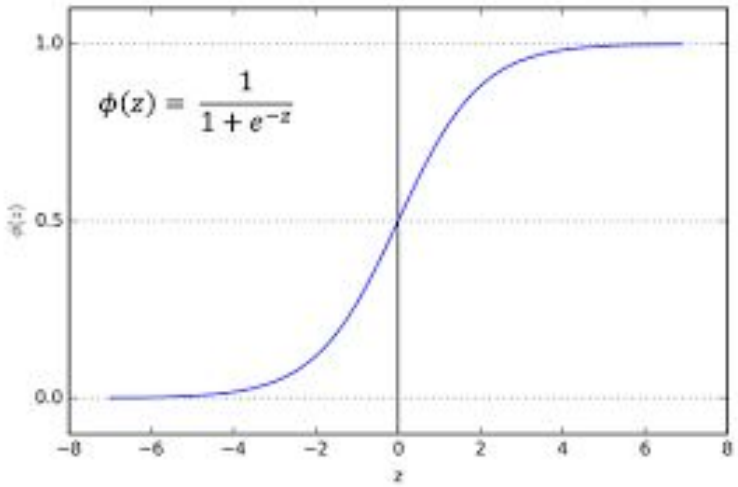
- Non-linear
- Differentiable
  - Easy to calculate the derivative for gradient descent
- Computationally Inexpensive
  - Calculating the derivative is simple, easy and fast leading to faster training
- Zero-Centered
  - Or normalized.
  - Empirically its proven, if we have normalized values (for i/p and corresponding layers) then the NN converges faster
  - Mean of the activation function will be 0, e.g., Tanh
- Non-saturating
  - Doesn't squeezes the value in a particular range hence avoiding vanishing gradient problem
  - Saturating functions, e.g., Sigmoid, tanh
  - E.g., Relu is non-saturating

Sigmoid Activation Function

Mathematical Form:

The **sigmoid activation function** is defined as:

$$f(z) = \frac{1}{1 + e^{-z}}$$



Where z is the weighted sum of inputs to the neuron.

Properties:

- **Range**: The output of the sigmoid function is always between 0 and 1. This makes it ideal for problems where the output represents probabilities, particularly in binary classification tasks.
- **Smooth and Differentiable**: The sigmoid function is **smooth** and **differentiable**, which is crucial for backpropagation and optimization in neural networks. The smooth nature allows for efficient computation of gradients.

Advantages:

- **Probabilistic Interpretation**: The sigmoid function is often used when the output needs to be interpreted as a probability, as the output is bounded between 0 and 1.
- **Normalization**: The sigmoid function **squashes** the input values into a small range, which can help with normalization and prevent extremely large values.

Disadvantages:

- **Vanishing Gradient Problem**: For very large or very small input values, the sigmoid function outputs values close to 0 or 1. This causes the gradient of the function to become very small (close to 0), which leads to the **vanishing gradient problem**. As a result, the network struggles to learn and adjust the weights, particularly in deep layers.
- **Saturated Neurons**: When the input z is very large or very small, the sigmoid neuron becomes **saturated**, meaning small changes in the input do not significantly affect the output, leading to poor learning.

Use Case:

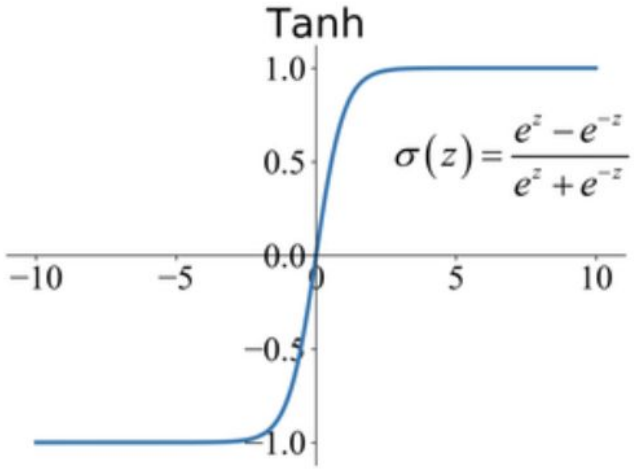
- The sigmoid activation function is primarily used in **binary classification problems**, where the output is either 0 or 1. For example, in logistic regression and binary classifiers, the sigmoid function is often used in the **output layer**.

Tanh Activation Function

Mathematical Form:

The **tanh activation function** (short for **hyperbolic tangent**) is defined as:

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



Properties:

- **Range**: The output of the tanh function ranges between **-1 and 1**, which makes it centered around zero.
- **Zero-Centered Outputs**: Unlike the sigmoid function, which outputs values between 0 and 1, the tanh function is symmetric around the origin. This can be helpful in **optimization** since the network does not have to deal with only positive values.

Advantages:

- **Stronger Gradient**: Compared to the sigmoid function, the **tanh function** has a **stronger gradient** for inputs that are closer to 0. This makes it more effective in training deeper neural networks because it can propagate stronger gradients.
- **Zero-Centered**: Since the output is centered around zero, the tanh function helps in avoiding problems with **bias shifts** during training, which may be present when using the sigmoid function.

Disadvantages:

- **Vanishing Gradient Problem**: Like the sigmoid function, tanh also suffers from the vanishing gradient problem when the input values are very large or very small. For input values close to -1 or 1, the gradient becomes very small, leading to slower learning.

Use Case:

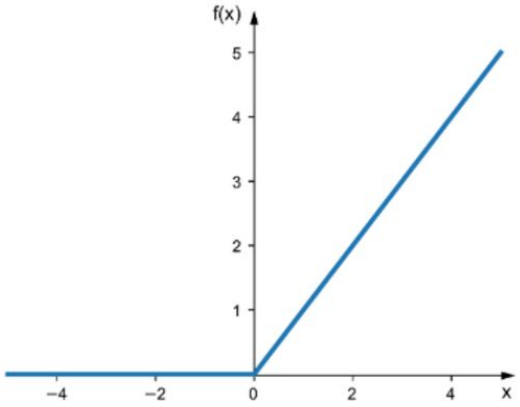
- Tanh is commonly used in the **hidden layers** of neural networks, especially in applications such as **natural language processing (NLP)** and **time series analysis**, where the output is expected to be centered around zero.

ReLU (Rectified Linear Unit) Activation Function

Mathematical Form:

The **ReLU activation function** is defined as:

$$f(z) = \max(0, z)$$



Where z is the weighted sum of inputs to the neuron.

Properties:

- **Range**: The output of ReLU is **0** for all negative input values and the input value itself for positive inputs. Therefore, the output ranges from 0 to infinity.
- **Non-Linearity**: Despite its simplicity, ReLU introduces **non-linearity** into the network, which allows the network to learn complex patterns. It preserves the linear property for positive inputs, but for negative inputs, it forces the neuron to output 0.

Advantages:

- **Efficient Computation**: ReLU is computationally very efficient because it only requires a comparison between the input value and zero. This simplicity leads to faster training times.
- **Avoids Vanishing Gradient Problem**: Unlike sigmoid and tanh, ReLU does not saturate for positive inputs. This helps prevent the vanishing gradient problem, as the gradients for positive values are large and do not approach zero.

Disadvantages:

- **Dying ReLU Problem**: One of the main problems with ReLU is that neurons can become inactive or "die" during training if they consistently receive negative inputs. Once a neuron becomes inactive, it only outputs 0 and stops learning, leading to **dead neurons**.
- **Unbounded Output**: For large input values, ReLU can produce very large outputs, which may cause instability during training, especially if the learning rate is too high.

Use Case:

- ReLU is widely used in **deep neural networks** and **convolutional neural networks (CNNs)**, particularly in tasks such as **image recognition** and **speech recognition**. Its computational efficiency and ability to avoid the vanishing gradient problem make it the most popular activation function in these fields.

Leaky ReLU and Parametric ReLU

Leaky ReLU Mathematical Form:

The **Leaky ReLU** activation function is a variation of ReLU and is defined as:

$$f(z) = \begin{cases} z & \text{if } z > 0 \\ \alpha z & \text{if } z \leq 0 \end{cases}$$

Where α is a small positive constant, usually set to a value like 0.01.

Parametric ReLU:

The **Parametric ReLU (PReLU)** activation function is similar to Leaky ReLU, but instead of using a fixed α, this parameter is **learned during training**. This gives the network more flexibility to adapt the slope for negative values.

Advantages:

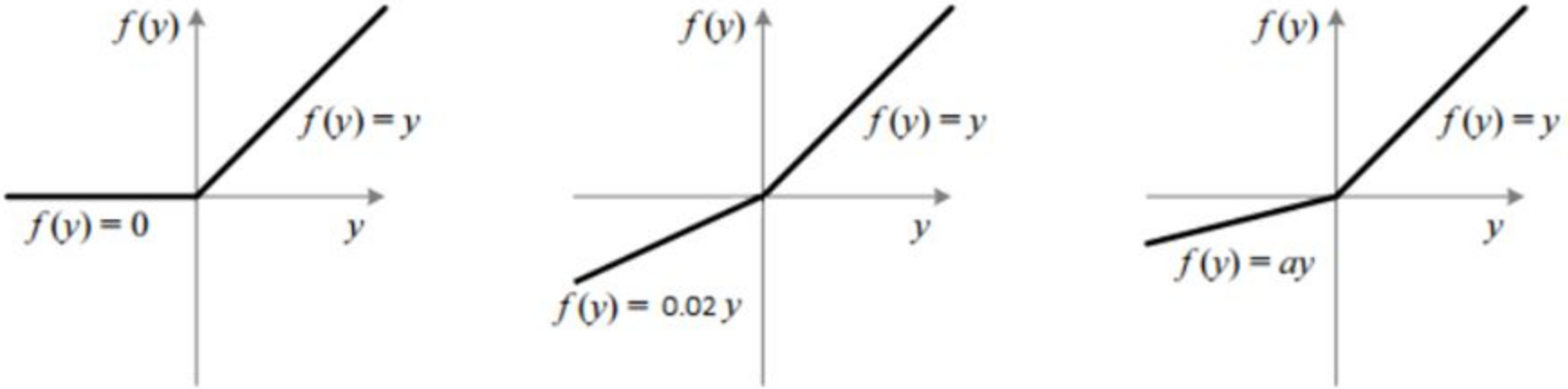
- **Solves Dying ReLU Problem**: Leaky ReLU addresses the dying ReLU problem by allowing a small, negative slope for negative inputs. This ensures that neurons do not become completely inactive and can still learn from negative inputs.
- **Flexibility with PReLU**: Parametric ReLU adds even more flexibility by making α\alphaα a learnable parameter, allowing the network to determine the best slope for negative values during training.

Disadvantages:

- **Minor Instability**: Introducing a negative slope for negative inputs can lead to **minor instability** during training in some cases, particularly if the value of α is too large or if it is not well-tuned.

Use Case:

- Leaky ReLU and PReLU are typically used in **deeper neural networks** where the dying ReLU problem becomes more prevalent. These functions are beneficial in networks where standard ReLU fails to activate a sufficient number of neurons.



(Left) ReLU, (Middle) LeakyReLU and (Last) PReLU

Advanced Activation Functions: Softmax

Mathematical Form:

The **softmax activation function** is used in multi-class classification problems and is defined as:

$$\text{softmax}(z_j) = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

Where:

- z_j is the weighted sum for class j, and
- The denominator is the sum of exponentials across all classes, k.

Properties:

- **Range**: The output of the softmax function is a **probability distribution**, with each output value between **0 and 1**.
- **Normalization**: The softmax function normalizes the output so that the sum of all output values equals **1**. This makes it ideal for interpreting the output as a probability distribution over multiple classes.

Advantages:

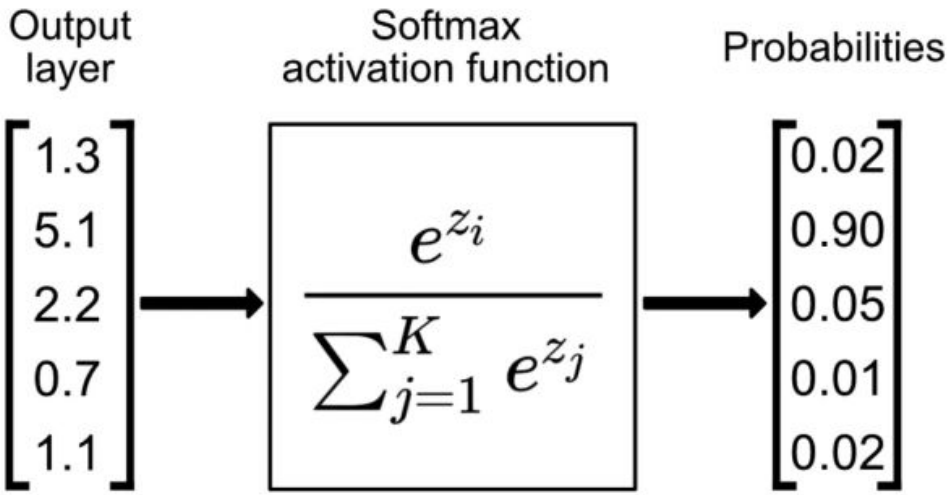
- Softmax ensures that the network produces output that can be interpreted as the **probability** of each class. This is essential for multi-class classification tasks, where the model must assign probabilities to each possible class and make predictions based on the highest probability.

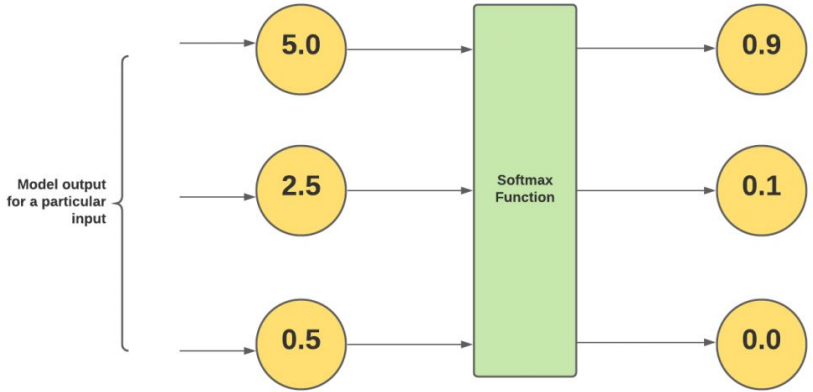
Disadvantages:

- **Numerical Instability**: The exponential nature of softmax can lead to numerical instability when dealing with large input values. A common solution is to subtract the maximum value from each input before applying the softmax function to stabilize the computation.

Use Case:

- Softmax is most commonly used in the **output layer** of neural networks designed for **multi-class classification** tasks, such as image classification (e.g., **CIFAR-10** dataset) or NLP tasks where multiple possible outcomes exist (e.g., part-of-speech tagging, text classification).


Output layer
Softmax activation function
Probabilities
1.3
5.1
2.2
0.7
1.1
$\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$
0.02
0.90
0.05
0.01
0.02


Model output for a particular input
5.0
2.5
0.5
Softmax Function
0.9
0.1
0.0

Backpropagation

Backpropagation Intuition

Backpropagation is the algorithm that makes **gradient descent** efficient for neural networks. Instead of calculating gradients for each parameter individually, backpropagation efficiently computes the gradient of the loss function with respect to every weight and bias in the network.

Chain Rule of Calculus:

- The core of backpropagation is the **chain rule** from calculus, which allows us to compute the gradient of a composition of functions.
- In neural networks, each layer's output depends on the previous layer's output. The chain rule allows us to compute how changing a weight in one layer affects the final output (and thus the loss).

Backpropagation works in two main steps:

1. **Forward Pass**:
    - The input data passes through the network, producing an output.
    - The loss is calculated based on the predicted output and the actual target.
2. **Backward Pass**:
    - The algorithm starts from the output layer and computes the gradient of the loss with respect to each weight and bias by applying the chain rule.
    - These gradients are propagated backward through the network (hence the name "backpropagation").

Why Backpropagation is Efficient:

- **Backpropagation** allows for the computation of all the gradients in a single pass through the network, as opposed to computing the gradient for each parameter individually.
- It reuses computations from the forward pass and applies the chain rule to distribute the error through the layers.

Update Weights Using Gradients:

Once the gradients are calculated, the weights are updated in the direction that reduces the loss using the gradient descent update rule:

$$W := W - \eta \cdot \frac{\partial J(W)}{\partial W}$$

Where:

- W is the weight matrix.
- η is the learning rate.
- $\frac{\partial J(W)}{\partial W}$ is the gradient of the loss function w.r.t. W.

For a simple neural network with one hidden layer, the gradient of the loss w.r.t. the weight connecting two neurons is:

$$\frac{\partial L}{\partial W} = \delta \cdot a$$

Where:

- δ is the error term propagated from the output layer.
- a is the activation from the previous layer.

Need for Backpropagation:

- Without backpropagation, computing the gradients for all the weights would be computationally expensive, especially in deep networks with many layers and parameters.
- Backpropagation ensures that the network learns efficiently by providing a systematic way to compute the gradients for all weights simultaneously.
- It has been instrumental in the success of deep learning by enabling the training of deep networks with many layers, where manual gradient computation would be infeasible.

Mathematical Formulation of Backpropagation

1. Goal: Find the Gradients of the Loss Function w.r.t. Weights and Biases

The fundamental objective of backpropagation is to compute the gradients of the loss function L with respect to each parameter (i.e., the weights $W^{(l)}$ and biases $b^{(l)}$ in the network. These gradients allow us to update the parameters in the direction that minimizes the loss function using an optimization algorithm like gradient descent.

2. Chain Rule: Compute Gradients Layer by Layer

The chain rule from calculus plays a critical role in backpropagation. Since the output of each layer depends on the previous layer, the chain rule allows us to propagate the error backward through the network, adjusting each layer's weights and biases accordingly.

Backpropagation computes the gradient for each layer starting from the output layer and moving back through the hidden layers, finally reaching the input layer.

Notation

To understand the derivation of backpropagation, let's define some essential notations:

- **Loss function**: Let L denote the loss function, which measures the difference between the network's predictions and the true target values.
- **Input to activation function in layer l**: Let $z^{(l)}$ represent the weighted sum of inputs to the activation function in layer l.
- **Output of activation function in layer l**: Let $a^{(l)}$ represent the output of the activation function in layer l.
- **Weight matrix in layer l**: Let $W^{(l)}$ represent the weight matrix in layer l.
- **Bias term in layer l**: Let $b^{(l)}$ represent the bias term in layer l.

Each layer computes the following:

$$z^{(l)} = W^{(l)} \cdot a^{(l-1)} + b^{(l)}$$

$$a^{(l)} = f(z^{(l)})$$

Where:

- $z^{(l)}$ is the input to the activation function in layer l,
- $a^{(l-1)}$ is the output of the activation function from the previous layer,
- f is the activation function applied to $z^{(l)}$.

Why Backpropagation is Efficient:

- **Backpropagation** allows for the computation of all the gradients in a single pass through the network, as opposed to computing the gradient for each parameter individually.
- It reuses computations from the forward pass and applies the chain rule to distribute the error through the layers.

Update Weights Using Gradients:

Once the gradients are calculated, the weights are updated in the direction that reduces the loss using the gradient descent update rule:

$$W := W - \eta \cdot \frac{\partial J(W)}{\partial W}$$

Where:

- W is the weight matrix.
- η is the learning rate.
- $\frac{\partial J(W)}{\partial W}$ is the gradient of the loss function w.r.t. W.

Backpropagation Steps

Step 1: Forward Pass

In the **forward pass**, data flows through the network from the input layer to the output layer. For each layer l, we compute:

1. **Weighted sum of inputs**:

$$z^{(l)} = W^{(l)} \cdot a^{(l-1)} + b^{(l)}$$

Here, $W^{(l)}$ is the weight matrix of layer l, and $a^{(l-1)}$ is the output from the previous layer.

2. **Apply activation function**:

$$a^{(l)} = f(z^{(l)})$$

The output of the activation function becomes the input to the next layer.

Step 2: Compute the Loss

At the output layer, after the forward pass, we compute the loss using a chosen loss function. For regression problems, a common loss function is **Mean Squared Error (MSE)**:

$$L = \frac{1}{2} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Where:

- n is the number of data points,
- y_i is the actual target value for the i-th data point,
- $\hat{y}_i$ is the predicted output.

For classification problems, we typically use **Cross-Entropy Loss**, but MSE is used here as an example for simplicity.

Step 3: Backward Pass

The **backward pass** involves computing the gradients of the loss function with respect to each parameter (weights and biases) using the chain rule.

3.1 Output Layer

At the output layer, we first compute the error term $\delta^{(L)}$, which represents the gradient of the loss function with respect to the input of the activation function in the output layer.

$$\delta^{(L)} = \frac{\partial L}{\partial z^{(L)}} \qquad \frac{\partial L}{\partial z^{(L)}} = \frac{\partial L}{\partial a^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} \qquad \frac{\partial L}{\partial z^{(L)}} = \frac{\partial L}{\partial a^{(L)}} \cdot f'(z^{(L)})$$

Where $f'(z^{(L)})$ is the derivative of the activation function (e.g., the derivative of the sigmoid activation function).

Now, we compute the gradient of the loss with respect to the weights $W^{(L)}$ and biases $b^{(L)}$ in the output layer:

$$\frac{\partial L}{\partial W^{(L)}} = \frac{\partial L}{\partial z^{(L)}} \cdot \frac{\partial z^{(L)}}{\partial W^{(L)}} \qquad \frac{\partial L}{\partial W^{(L)}} = \delta^{(L)} \cdot (a^{(L-1)})^T$$

$$\frac{\partial L}{\partial b^{(L)}} = \delta^{(L)}$$

3.2 Hidden Layers

For each hidden layer, the error term $\delta^{(l)}$ is computed recursively by propagating the error backward. The error term for a hidden layer depends on the error from the next layer:

$$\delta^{(l)} = (W^{(l+1)})^T \cdot \delta^{(l+1)} \cdot f'(z^{(l)})$$

Where:

- $(W^{(l+1)})^T$ is the transpose of the weight matrix of the next layer,
- $\delta^{(l+1)}$ is the error from the next layer,
- $f'(z^{(l)})$ is the derivative of the activation function in layer l.

The gradients with respect to the weights and biases in hidden layer l are:

$$\frac{\partial L}{\partial W^{(l)}} = \delta^{(l)} \cdot (a^{(l-1)})^T$$

$$\frac{\partial L}{\partial b^{(l)}} = \delta^{(l)}$$

This process continues recursively for all layers, starting from the output layer and moving back through the network to the input layer.

Example: Simple Network with One Hidden Layer

Let's consider a simple neural network with one hidden layer to demonstrate how backpropagation works. We have:

- **Input Layer**: $a^{(0)}$,
- **Hidden Layer**: $W^{(1)}, b^{(1)}, z^{(1)}, a^{(1)}$,
- **Output Layer**: $W^{(2)}, b^{(2)}, z^{(2)}, a^{(2)}$.

Forward Pass:

1. Hidden Layer:

$$z^{(1)} = W^{(1)} \cdot a^{(0)} + b^{(1)}$$

$$a^{(1)} = f(z^{(1)})$$

2. Output Layer:

$$z^{(2)} = W^{(2)} \cdot a^{(1)} + b^{(2)}$$

$$a^{(2)} = f(z^{(2)})$$

Loss Function (MSE):

$$L = \frac{1}{2}(y - a^{(2)})^2$$

Backward Pass:

1. Output Layer:

$$\delta^{(2)} = \frac{\partial L}{\partial z^{(2)}} = (a^{(2)} - y) \cdot f'(z^{(2)})$$

$$\frac{\partial L}{\partial W^{(2)}} = \delta^{(2)} \cdot (a^{(1)})^T$$

$$\frac{\partial L}{\partial b^{(2)}} = \delta^{(2)}$$

2. Hidden Layer:

$$\delta^{(1)} = (W^{(2)})^T \cdot \delta^{(2)} \cdot f'(z^{(1)})$$

$$\frac{\partial L}{\partial W^{(1)}} = \delta^{(1)} \cdot (a^{(0)})^T$$

$$\frac{\partial L}{\partial b^{(1)}} = \delta^{(1)}$$

By computing these gradients, we can update the weights and biases using gradient descent.

Example: Gradient Calculation for a Sigmoid Neuron

Let's walk through an example to illustrate the gradient calculation for both weights and biases:

Given:

- Neuron in layer L uses the sigmoid activation function.
- We have a single input $x = 0.5$, weight $W = 0.4$, bias $b = 0.1$.
- The true label is $y = 1$.

Step 1: Forward Pass

1. Compute the weighted sum:

$$z = W \cdot x + b = 0.4 \cdot 0.5 + 0.1 = 0.3$$

2. Apply the sigmoid activation function:

$$a = \frac{1}{1 + e^{-0.3}} \approx 0.574$$

Step 2: Compute the Loss

Let's assume Mean Squared Error (MSE) as the loss function:

$$L = \frac{1}{2}(y - a)^2 = \frac{1}{2}(1 - 0.574)^2 \approx 0.091$$

Step 3: Backward Pass - Gradient Calculation

1. Compute the error term:
$$\delta = (a - y) \cdot a \cdot (1 - a) = (0.574 - 1) \cdot 0.574 \cdot (1 - 0.574) \approx -0.103$$
2. Compute the gradient w.r.t. the weight:
$$\frac{\partial L}{\partial W} = \delta \cdot x = -0.103 \cdot 0.5 = -0.0515$$
3. Compute the gradient w.r.t. the bias:
$$\frac{\partial L}{\partial b} = \delta = -0.103$$

Step 4: Update Weights and Biases

Let's use a learning rate $\eta = 0.1$.

- Update the weight:

$$W := W - \eta \cdot \frac{\partial L}{\partial W} = 0.4 - 0.1 \cdot (-0.0515) = 0.40515$$

- Update the bias:

$$b := b - \eta \cdot \frac{\partial L}{\partial b} = 0.1 - 0.1 \cdot (-0.103) = 0.1103$$