Neural networks

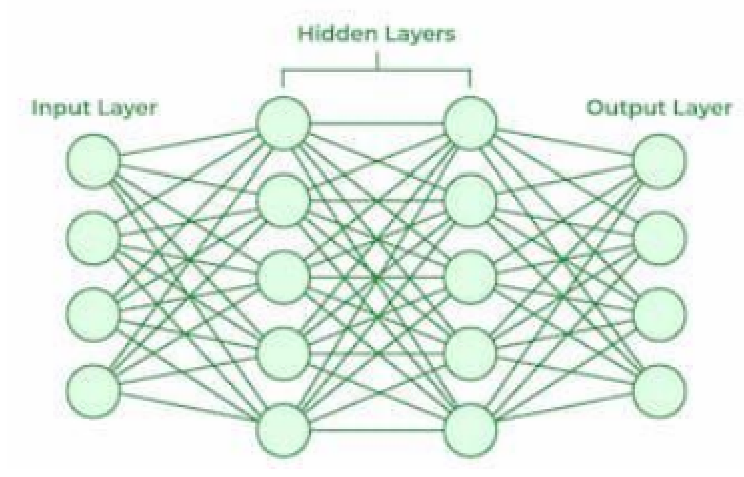
Neural networks are computational models inspired by the structure and function of the human brain. They are designed to recognize patterns, classify data, and make predictions based on input data. Neural networks consist of interconnected nodes or "neurons," organized into layers, and are used in various applications including image and speech recognition, natural language processing, and predictive analytics.

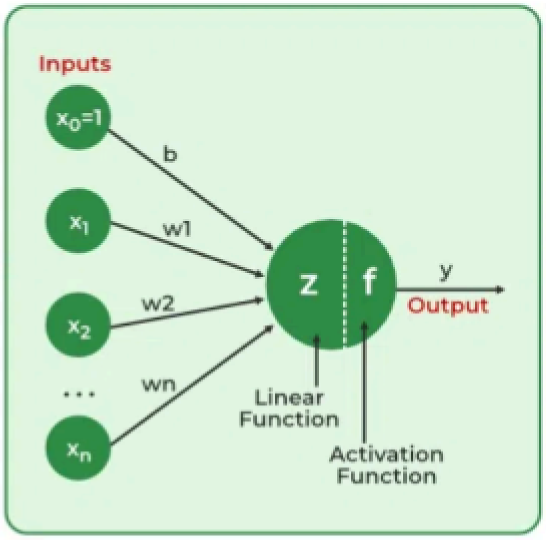
Importance of Neural Networks

● Pattern Recognition: Neural networks excel at identifying patterns and relationships in complex data, making them suitable for tasks like image and speech recognition.
● Adaptability: They can learn from data and adapt to new, unseen patterns, improving their performance over time with more data.
● Feature Extraction: Automatically extract and learn relevant features from raw data without needing manual feature engineering.
● Versatility: Applicable to a wide range of problems, from classification and regression to time series forecasting and anomaly detection.
● Deep Learning: Enable the development of deep learning models with multiple layers, capable of handling large-scale data and complex tasks.

How Neural Networks Work:

Neural networks operate through two main processes: forward propagation and backpropagation.





Forward Propagation(Propagating Inputs from input layer to output layer, through hidden layer)

Forward propagation is the process by which input data is passed through the network to generate predictions or outputs. It involves calculating the activations of neurons layer by layer from the input layer through the hidden layers to the output layer.

1. **Input Data:** Data is fed into the input layer, which passes it to the next layer.
2. **Weighted Sum:** Each neuron in a layer computes a weighted sum of its inputs. The weighted sum is calculated as:

$$z_i = \sum_j w_{ij}x_j + b_i$$

where **wij** is the weight from neuron **j** in the previous layer to neuron **i** in the current layer, **xj** is the input, and bi is the bias term.

3. **Activation Function:** The weighted sum is passed through an activation function to introduce non-linearity. The activation function **f(zi)** determines the neuron's output:

   $$a_i = f(z_i)$$

   Common activation functions include:

   - Sigmoid: $f(z) = \frac{1}{1+e^{-z}}$
   - ReLU (Rectified Linear Unit): $f(z) = \max(0, z)$
   - Tanh: $f(z) = \tanh(z)$
4. **Propagation Through Layers:** This process continues through all layers of the network until the final output layer is reached, producing the network's prediction.

Backpropagation (Propagating error from output layer to input layer through hidden layer and updating the biases accordingly)

Backward propagation (backpropagation) is the process used to update the network's weights and biases based on the error calculated from the forward propagation. It involves calculating the gradients of the loss function with respect to each weight and bias and using these gradients to adjust the weights and biases to minimize the loss.

1. **Compute Loss:** The network's output is compared to the actual target value using a loss function (e.g., Mean Squared Error for regression or Cross-Entropy Loss for classification). The loss function measures the discrepancy between the predicted and actual values.
2. **Calculate Gradients:** Using the loss function, the gradient of the loss with respect to each weight is calculated. This involves applying the chain rule of calculus to compute how changes in weights affect the loss.
3. **Update Weights:** The weights are updated using an optimization algorithm such as Gradient Descent. The update rule for a weight **wij** is:

   $$w_{ij} = w_{ij} - \eta \frac{\partial L}{\partial w_{ij}}$$

   where η is the learning rate, and **∂L\∂wij** is the gradient of the loss function with respect to the weight.
4. **Iterate:** This process of forward propagation and backpropagation is repeated iteratively through multiple epochs (complete passes through the training dataset) until the network's performance improves and converges to a satisfactory level.

[or]

1. **Compute Loss(Error):**
   - Calculate the loss (error) between the predicted output and the actual target using a loss function.
2. Compute the updated values of all weights and biases, from:

a. Output Layer —--------> Hidden Layer
b. Hidden Layer —----------> Input Layer

Issues in Training Neural Networks

1. Overfitting

🔴 Issue:

- The model performs well on training data but poorly on unseen test data.
- Happens when the network learns noise instead of general patterns.

✅ Solutions:

- **Regularization**: Use **L1/L2 (Lasso/Ridge) regularization** or **Dropout** to prevent over-reliance on specific neurons.
- **Early Stopping**: Stop training when validation loss stops improving.
- **Data Augmentation**: Increase dataset size artificially (e.g., rotating/flipping images).
- **Reduce Model Complexity**: Use fewer layers or neurons.

---

2. Underfitting

🔴 Issue:

- The model fails to capture patterns in the training data.
- Happens when the model is too simple for the problem.

✅ Solutions:

- **Increase Model Complexity**: Add more layers or neurons.
- **Train Longer**: Ensure proper convergence before stopping.
- **Use Better Features**: Improve input data quality.
- **Reduce Regularization**: Too much regularization can cause underfitting.

---

3. Vanishing & Exploding Gradients

🔴 Issue:

- In deep networks, gradients can become **too small (vanishing)** or **too large (exploding)**, preventing proper weight updates.
- Common in deep networks with **sigmoid or tanh activation functions**.

✅ Solutions:

- **Use ReLU Activation**: Avoids small gradients (instead of sigmoid/tanh).
- **Batch Normalization**: Normalizes activations to keep values stable.
- **Gradient Clipping**: Restricts the size of large gradients.
- **Proper Weight Initialization**: Use **Xavier (Glorot) or He Initialization**.

---

4. Slow Convergence

🔴 Issue:

- Training takes too long due to poor weight updates.
- Large datasets and deep architectures make training inefficient.

✅ Solutions:

- **Use a Good Optimizer**: Adam, RMSprop, or Momentum-based optimizers speed up learning.
- **Learning Rate Scheduling**: Adjust learning rate dynamically (e.g., decay after epochs).
- **Batch Normalization**: Speeds up convergence by stabilizing activations.

---

5. Poor Choice of Learning Rate

🔴 Issue:

- **Too High** → Model diverges (loss increases instead of decreasing).
- **Too Low** → Training is very slow and may get stuck.

✅ Solutions:

- **Learning Rate Annealing**: Reduce the learning rate gradually.
- **Use Adaptive Learning Rates**: Adam or RMSprop adjust the learning rate automatically.
- **Grid Search**: Experiment with different learning rates.

---

6. Data Imbalance

🔴 Issue:

- When one class dominates, the model is biased toward that class.
- Example: In fraud detection, **99% transactions are normal, 1% are fraud**.

✅ Solutions:

- **Class Weighting**: Assign higher weight to the minority class.
- **Oversampling**: Duplicate rare class samples (SMOTE algorithm).
- **Undersampling**: Remove excess majority class samples.

---

7. Lack of Data

🔴 Issue:

- Neural networks need large datasets to generalize well.
- Small datasets lead to high variance (overfitting).

✅ Solutions:

- **Data Augmentation**: Create synthetic samples (for images, rotate/flip).
- **Transfer Learning**: Use a pre-trained model instead of training from scratch.

---

8. Internal Covariate Shift

🔴 Issue:

- **Distribution of activations** changes during training, making optimization unstable.
- Happens when earlier layers' weights change drastically.

✅ Solutions:

- **Batch Normalization**: Normalizes inputs of each layer.
- **Smaller Learning Rates**: Reduce large changes in weights.

---

9. Dead Neurons (Dying ReLU Problem)

🔴 Issue:

- In ReLU activation, some neurons always output **0** and stop learning.
- Happens when neurons get stuck with large negative inputs.

✅ Solutions:

- **Use Leaky ReLU or ELU** instead of standard ReLU.
- **Reduce Learning Rate** to avoid large updates.

---

10. Exploding Memory Usage

🔴 Issue:

- Large models require high RAM/GPU memory.
- Training with large batch sizes can lead to **Out of Memory (OOM) errors**.

✅ Solutions:

- **Use Smaller Batches**: Reduce batch size if OOM occurs.
- **Model Pruning & Quantization**: Reduce number of parameters.
- **Gradient Checkpointing**: Reduce memory footprint in deep models.

SVM for Classification:

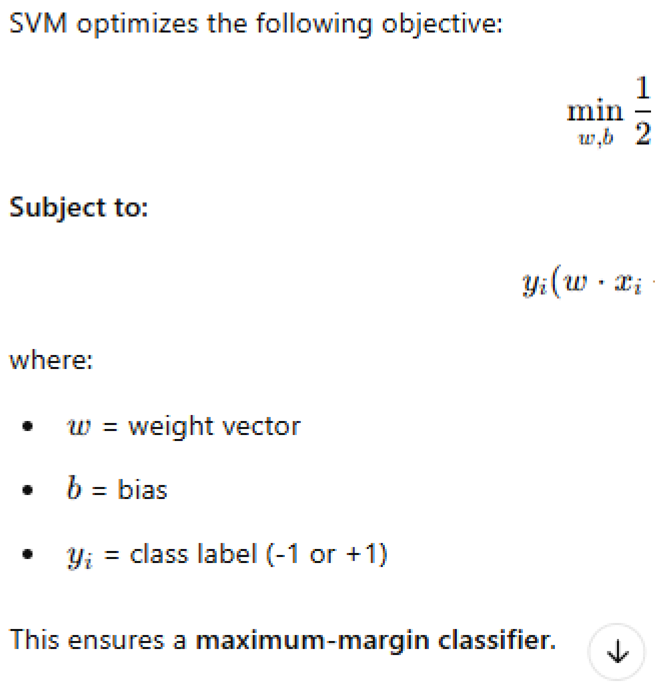
Support Vector Machine (SVM) is a supervised learning algorithm used for classification and regression. It is particularly effective for high-dimensional datasets and cases where the decision boundary is not easily separable.

(Refer the notes for SVM)

How SVM Works for Classification

SVM classifies data by finding the **optimal hyperplane** that best separates the classes.

- **Margin**: The distance between the hyperplane and the closest data points (support vectors).
- **Support Vectors**: The data points that are closest to the hyperplane and influence its position.
- **Kernel Trick**: SVM can handle non-linearly separable data using kernels (e.g., polynomial, RBF).



SVM optimizes the following objective:

$$\min_{w,b} \frac{1}{2}||w||^2$$

Subject to:

$$y_i(w \cdot x_i + b) \geq 1$$

where:

- w = weight vector
- b = bias
- y_i = class label (-1 or +1)

This ensures a **maximum-margin classifier.**

SVM for Regression:

Support Vector Regression (SVR) is the regression counterpart of SVM, used to predict continuous values rather than categorical labels. It works by finding a function that deviates at most by a small margin (ϵ\epsilonϵ) from the actual target values.

How SVR Works

SVR aims to fit a function that keeps most of the training points within a certain margin (ϵ\epsilonϵ) while minimizing the model complexity.

- **ϵ\epsilonϵ-Insensitive Tube:**
  - Unlike traditional regression, SVR **ignores small errors** within a threshold ϵ\epsilonϵ.
  - Only **support vectors** outside this margin affect the model.
- **Optimization Objective:**
  SVR minimizes the error while keeping the model complexity low.

$$\min_{w,b} \frac{1}{2}||w||^2$$

Subject to constraints:

$$|y_i - (w \cdot x_i + b)| \leq \epsilon$$

where:

- w = weight vector
- b = bias
- y_i = actual target value
- x_i = input features
- ϵ = margin of tolerance

Reproducing Kernels

Reproducing kernels are a fundamental concept in functional analysis and machine learning, particularly in **Support Vector Machines (SVM)** and **Kernel Methods**. They enable algorithms to operate in **high-dimensional feature spaces** without explicitly computing transformations.

1. What is a Kernel Function?

A **kernel function** is a function $K(x, x')$ that computes the **dot product** in a higher-dimensional space **without explicitly mapping data** into that space.

$$K(x, x') = \Phi(x) \cdot \Phi(x')$$

where:

- x, x' are input vectors.
- $\Phi(x)$ is a mapping to a high-dimensional space.
- $K(x, x')$ is the kernel function.

✅ **Benefit:** Saves computational cost by avoiding direct computation in high-dimensional spaces.

2. Reproducing Kernel Hilbert Space (RKHS)

A **Reproducing Kernel Hilbert Space (RKHS)** is a Hilbert space where evaluation of functions is done using kernel functions.

Reproducing Property

For a function f in RKHS $\mathcal{H}$, the kernel function satisfies:

$$f(x) = \langle f, K(x, \cdot) \rangle_{\mathcal{H}}$$

This means the kernel function **acts as a "basis function"** for the space.

✅ **Why is this important?**

- Ensures that kernel methods work mathematically.
- Guarantees that function values can be computed efficiently.

3. Common Reproducing Kernels

A. Linear Kernel

$$K(x, x') = x \cdot x'$$

✅ **Use case:** When data is **linearly separable.**

B. Polynomial Kernel

$$K(x, x') = (x \cdot x' + c)^d$$

✅ **Use case:** When data has **non-linear relationships.**

C. Radial Basis Function (RBF) Kernel

$$K(x, x') = \exp(-\gamma ||x - x'||^2)$$

✅ **Use case:** Works well for **most problems.**

D. Sigmoid Kernel

$$K(x, x') = \tanh(\alpha x \cdot x' + c)$$

✅ **Use case:** Inspired by **neural networks.**

4. Why Are Reproducing Kernels Important?

✔ Enable **SVM, SVR, and PCA** in high-dimensional spaces.
✔ **Avoid explicit computation** of feature transformations.
✔ **Guarantee mathematical validity** of kernel-based learning.
✔ Provide a **powerful alternative** to deep learning in structured data.

Image Scene Classification Using K-Nearest Neighbors (KNN Classifier)

K-Nearest Neighbors (KNN) is a **simple yet effective** algorithm for **image scene classification** based on similarity measures. It classifies an image by finding the **K most similar images** in the training dataset and assigning the most common label.

---

1. Steps for Image Scene Classification Using KNN

Step 1: Collect Dataset

- Use a labeled dataset containing images of different scenes (e.g., **mountains, forests, beaches, cities**).
- Example datasets: **Places365, CIFAR-10, ImageNet**.

Step 2: Feature Extraction

- Images cannot be directly compared as vectors.
- Extract meaningful features such as:
  - **Color histograms** (RGB distribution)
  - **Edge detection** (Sobel, Canny)
  - **Texture features** (HOG, Gabor Filters)
  - **Deep Features** (ResNet, VGG, CNN)

Step 3: Normalize the Data

- Features are normalized to ensure **fair distance computation**.
- Common normalization techniques:
  - **Min-Max Scaling**
  - **Z-score Normalization**

Step 4: Train the KNN Classifier

- Store all training image feature vectors.
- Use **Euclidean distance** (or **cosine similarity**) to compare test images with training images.
- Choose the most frequent label among **K neighbors**.

Step 5: Classify Test Images

- Extract features from a new image.
- Compute distances to all training images.
- Assign the **majority label** among the K-nearest neighbors.