DEEP LEARNING ALGORITHMS

FNN | CNN | RNN | GAN | AUTOENCODERS | TRANSFORMERS | TENSORFLOW | KERAS | PYTOURCH

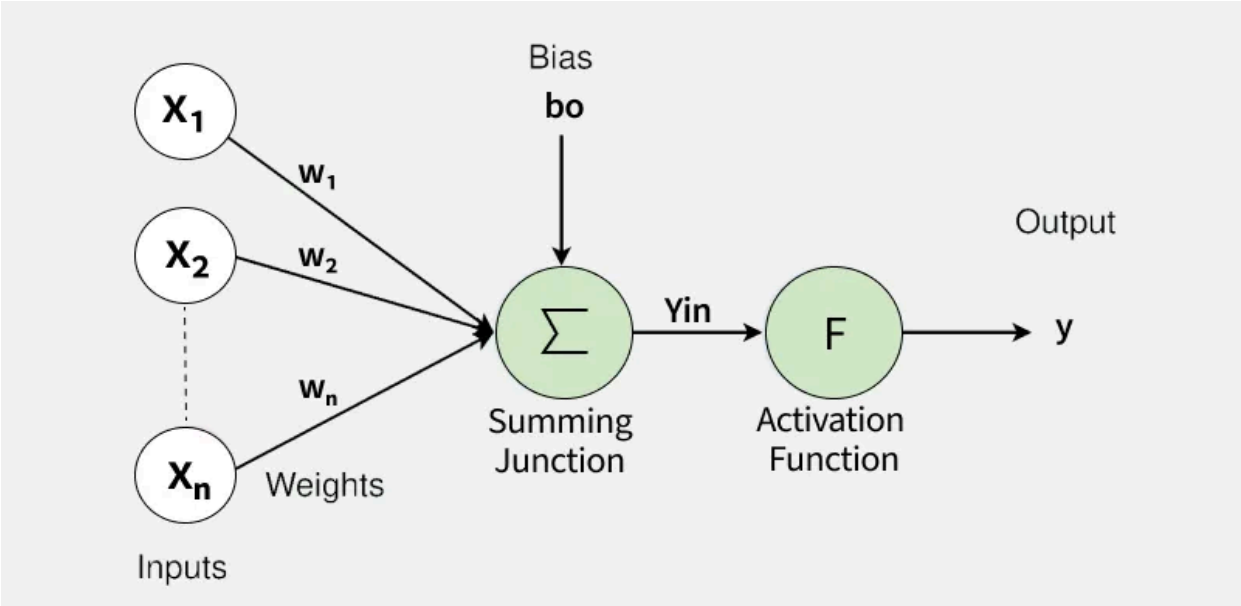
---

1. **Feedforward neural networks (FNNs)** are the simplest type of ANN, where data flows in one direction from input to output. It is used for basic tasks like classification.
2. **Convolutional Neural Networks (CNNs)** are specialized for processing grid-like data, such as images. CNNs use convolutional layers to detect spatial hierarchies, making them ideal for computer vision tasks.
3. **Recurrent Neural Networks (RNNs)** are able to process sequential data, such as time series and natural language. RNNs have loops to retain information over time, enabling applications like language modeling and speech recognition. Variants like LSTMs and GRUs address vanishing gradient issues.
4. **Generative Adversarial Networks (GANs)** consist of two networks—a generator and a discriminator—that compete to create realistic data. GANs are widely used for image generation, style transfer and data augmentation.
5. **Autoencoders** are unsupervised networks that learn efficient data encodings. They compress input data into a latent representation and reconstruct it, useful for dimensionality reduction and anomaly detection.
6. **Transformer Networks** have revolutionized NLP with self-attention mechanisms. Transformers excel at tasks like translation, text generation and sentiment analysis, powering models like GPT and BERT.

INPUT - HIDDEN LAYERS - SUMMING FUNCTION - ACTIVATION FUNCTION - OUTPUT - LOSS FUNCTION

Forward propagation

Forward propagation is the initial phase of processing input data through the neural network to produce an output or prediction. Let's see how it works:



1. Input Layer: The process starts with data entering the network's input layer. This could be anything from pixel values in an image to feature values in a dataset.
2. Weighted Sum: Each neuron calculates the weighted sum of the inputs. Each input is multiplied by its corresponding weight which shows the importance of that input.
3. Adding Biases: A bias is added to the weighted sum. Bias helps shift the output and provides flexibility, allowing the network to make better predictions even if all input values are zero.
4. Activation Function: The sum of the weighted inputs plus bias is passed through an activation function (e.g ReLU, sigmoid). The activation function decides if the neuron should activate which means it will pass information to the next layer or stay inactive.
5. Propagation: This process is repeated across multiple layers. The output of one layer becomes the input for the next, continuing until the network generates the final output or prediction.

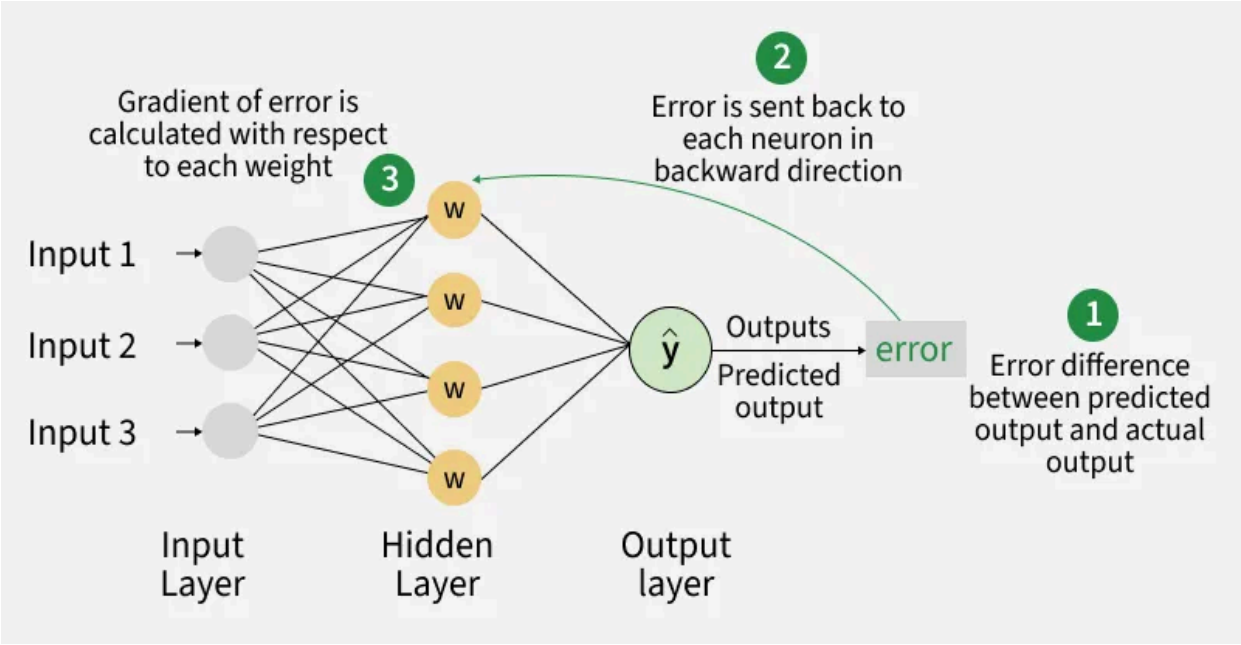
Backward propagation | Chain rule | Gradient descent

Back propagation is also known as "Backward Propagation of Errors" is a method used to train neural networks . Its goal is to reduce the difference between the model's predicted output and the actual output by adjusting the weights and biases in the network.

It works iteratively to adjust weights and bias to minimize the cost function. In each epoch the model adapts these parameters by reducing loss by following the error gradient. It often uses optimization algorithms like **gradient descent** or **stochastic gradient descent**. The algorithm computes the gradient using the chain rule from calculus allowing it to effectively navigate complex layers in the neural network to minimize the cost function.

Backpropagation

Once the network has made a prediction, it's important to evaluate how accurate that prediction is and make adjustments to improve future predictions. This is where backpropagation comes:



1. **Error Calculation:** Once the network generates an output, it's compared to the actual result (the target). The difference between the predicted and actual values is the error also called the loss.
2. **Gradient Calculation**: The error is propagated back through the network and the gradient or slope of the error with respect to the weights and biases is calculated. This tells the network how to adjust the parameters to minimize the error.

3. **Updating Weights and Biases**: Using the gradient, the network adjusts the weights and biases. The goal is to reduce the error in future predictions. This step is done through an optimization algorithm like gradient descent.
4. **Iteration**: This process of forward and backward propagation is repeated many times on different batches of data. With each iteration, the network's weights and biases get closer to the optimal values, improving the model's performance.

Types of Hidden Layers in Artificial Neural Networks:

1. Dense (Fully Connected) Layer
Dense (Fully Connected) Layer is the most common type of hidden layer in an ANN. Every neuron in a dense layer is connected to every neuron in the previous and subsequent layers. This layer performs a weighted sum of inputs and applies an activation function to introduce non-linearity. The activation function (like ReLU, Sigmoid, or Tanh) helps the network learn complex patterns.

- **Role**: Learns representations from input data.
- **Function**: Performs weighted sum and activation.

2. Convolutional Layer
Convolutional layers are used in Convolutional Neural Networks (CNNs) for image processing tasks. They apply convolution operations to the input, capturing spatial hierarchies in the data. Convolutional layers use filters to scan across the input and generate feature maps. This helps in detecting edges, textures, and other visual features.

- **Role**: Extracts spatial features from images.
- **Function**: Applies convolution using filters.

3. Recurrent Layer
Recurrent layers are used in Recurrent Neural Networks (RNNs) for sequence data like time series or natural language. They have connections that loop back, allowing information to persist across time steps. This makes them suitable for tasks where context and temporal dependencies are important.

- **Role**: Processes sequential data with temporal dependencies.
- **Function**: Maintains state across time steps.

4. Dropout Layer

Dropout layers are a regularization technique used to prevent overfitting. They randomly drop a fraction of the neurons during training, which forces the network to learn more robust features and reduces dependency on specific neurons. During training, each neuron is retained with a probability p.

- **Role**: Prevents overfitting.
- **Function**: Randomly drops neurons during training.

5. Pooling Layer

Pooling Layer is used to reduce the spatial dimensions of the data, thereby decreasing the computational load and controlling overfitting. Common types of pooling include Max Pooling and Average Pooling.

Use Cases: Dimensionality reduction in CNNs

6. Batch Normalization Layer

A **Batch Normalization Layer** normalizes the output of a previous activation layer by subtracting the batch mean and dividing by the batch standard deviation. This helps in accelerating the training process and improving the performance of the network.

Use Cases: Stabilizing and speeding up training

Summing function = weighted sums + bias

Neural networks learn from data and identify complex patterns that make them important in areas such as image recognition, natural language processing and autonomous systems. It has two fundamental components: weights and biases that help in how neural networks learn and make predictions.

Weights:

Weights are numerical values assigned to the connections between neurons. They find how much influence each input has on the network's final output.

- **Purpose**: During forward propagation, inputs are multiplied by their respective weights before being passed through an activation function. This helps decide how strongly an input will affect the output.
- **Learning Mechanism**: During training, weights are updated iteratively through optimization algorithms like gradient descent to minimize the difference between predicted and actual outcomes.
- **Generalization**: Well-tuned weights help the network not only make accurate predictions on training data but also generalize to new, unseen data.

- **Example:** In a neural network predicting house prices, the weight for the "size of the house" finds how much the house size influences the price prediction. The larger the weight, the bigger the impact size will have on the final result.

Biases:

Biases are additional parameters that adjust the output of a neuron. Unlike weights, they are not tied to any specific input but instead shift the activation function to better fit the data.

- **Purpose**: Biases help neurons activate even when the weighted sum of inputs is not enough. This allows the network to recognize patterns that don't necessarily pass through the origin.
- **Functionality**: Without biases, neurons would only activate when the input reaches a specific threshold. It makes the network more flexible by enabling activation across a wider range of conditions.
- **Training**: During training, biases are updated alongside weights through backpropagation. Together, they fine-tune the model, improving prediction accuracy.
- **Example**: In a house price prediction network, the bias might ensure that even for a house with a size of zero, the model predicts a non-zero price. This could reflect a fixed value such as land value or other baseline costs.

Activation functions

While building a neural network, one key decision is selecting the Activation Function for both the hidden layer and the output layer. It is a mathematical function applied to the output of a neuron. It introduces non-linearity into the model, allowing the network to learn and represent complex patterns in the data. Without this non-linearity feature a neural network would behave like a linear regression model no matter how many layers it has.

Activation function decides whether a neuron should be activated by calculating the weighted sum of inputs and adding a bias term. This helps the model make complex decisions and predictions by introducing non-linearities to the output of each neuron.

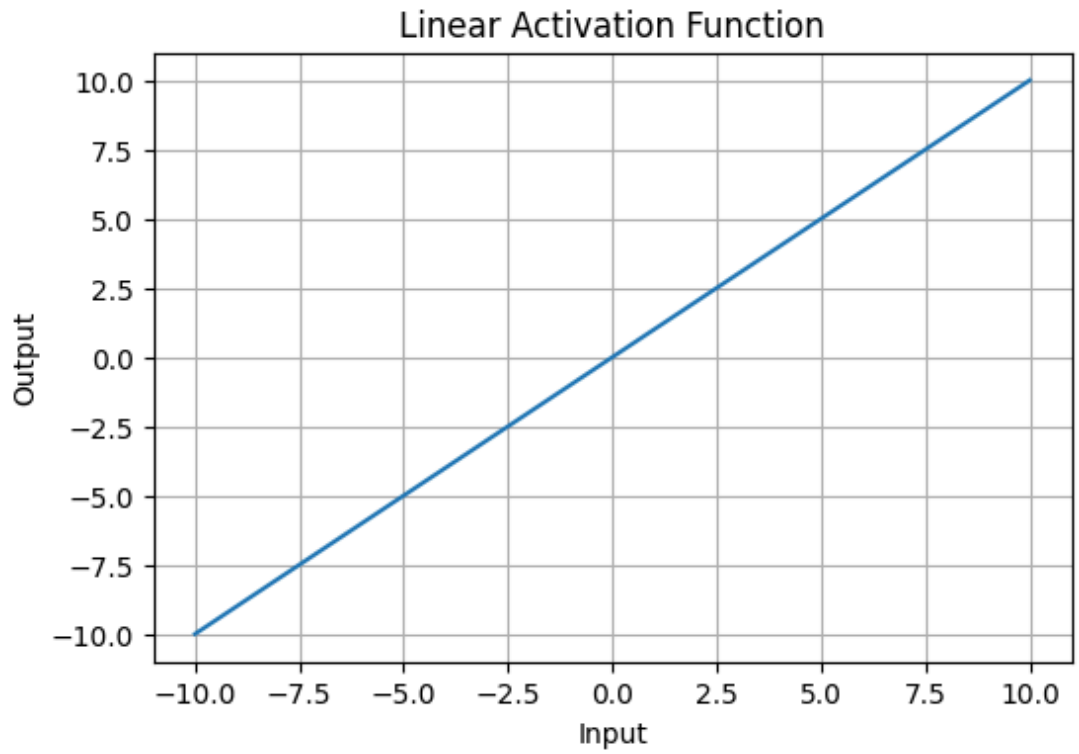
1. Linear Activation Function

Linear Activation Function resembles a straight line defined by y=x. No matter how many layers the neural network contains, if they all use linear activation functions the output is a linear combination of the input.

- The range of the output spans from $(-\infty$ to $+\infty)$
- Linear activation function is used at just one place i.e. output layer.

- Using linear activation across all layers makes the network's ability to learn complex patterns limited.

Linear activation functions are useful for specific tasks but must be combined with non-linear functions to enhance the neural network's learning and predictive capabilities.



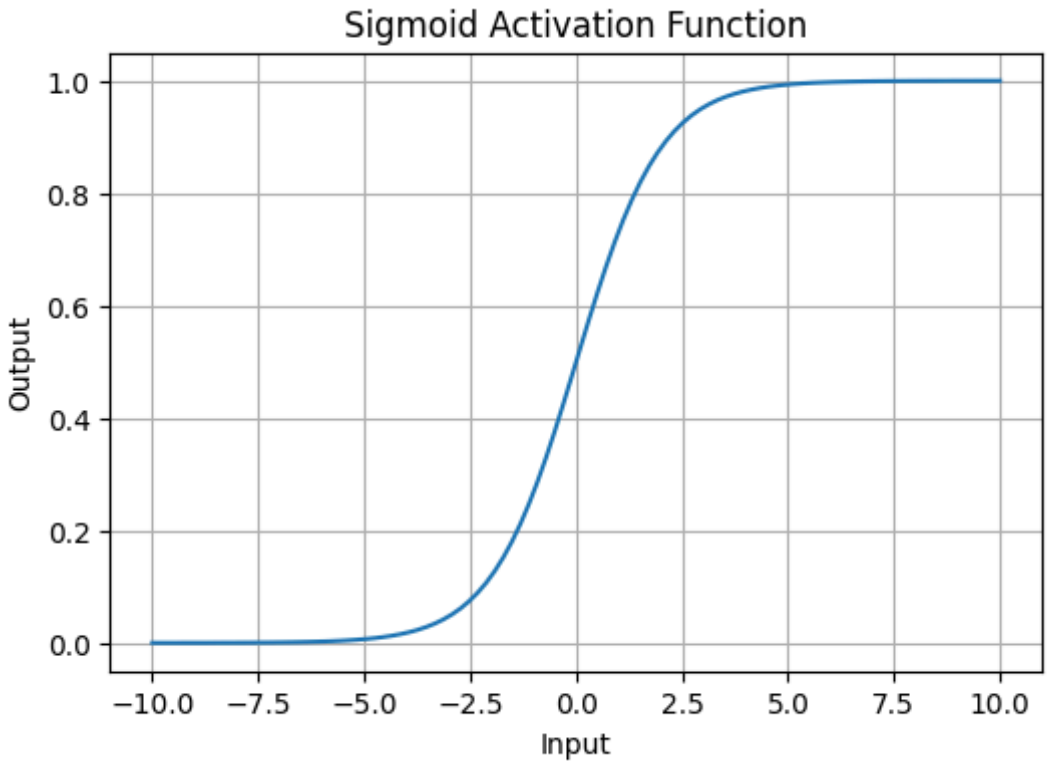
2. Non-Linear Activation Functions

1. Sigmoid Function

Sigmoid Activation Function is characterized by 'S' shape. It is mathematically defined as $A = \frac{1}{1+e^{-x}}$. This formula ensures a smooth and continuous output that is essential for gradient-based optimization methods.

- It allows neural networks to handle and model complex patterns that linear equations cannot.
- The output ranges between 0 and 1, hence useful for binary classification.

- The function exhibits a steep gradient when x values are between -2 and 2. This sensitivity means that small changes in input x can cause significant changes in output y which is critical during the training process.



2. Tanh Activation Function

Tanh function (hyperbolic tangent function) is a shifted version of the sigmoid, allowing it to stretch across the y-axis. It is defined as:

$$f(x)=\tanh(x)=f(x)=\tanh(x)=\frac{2}{1+e^{-2x}}-1$$

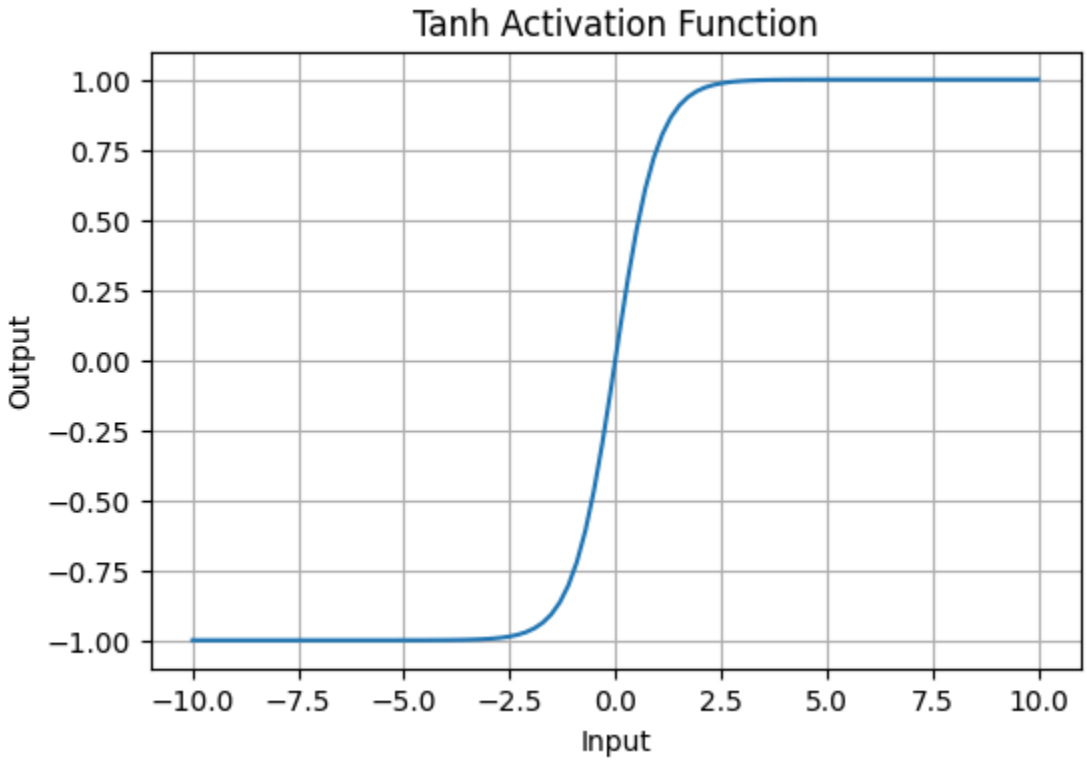
Alternatively, it can be expressed using the sigmoid function:

$$\tanh(x)=2\times\text{sigmoid}(2x)-1$$

$$\tanh(x)=2\times\text{sigmoid}(2x)-1$$

- **Value Range**: Outputs values from -1 to +1.
- **Non-linear**: Enables modeling of complex data patterns.

- **Use in Hidden Layers**: Commonly used in hidden layers due to its zero-centered output, facilitating easier learning for subsequent layers.



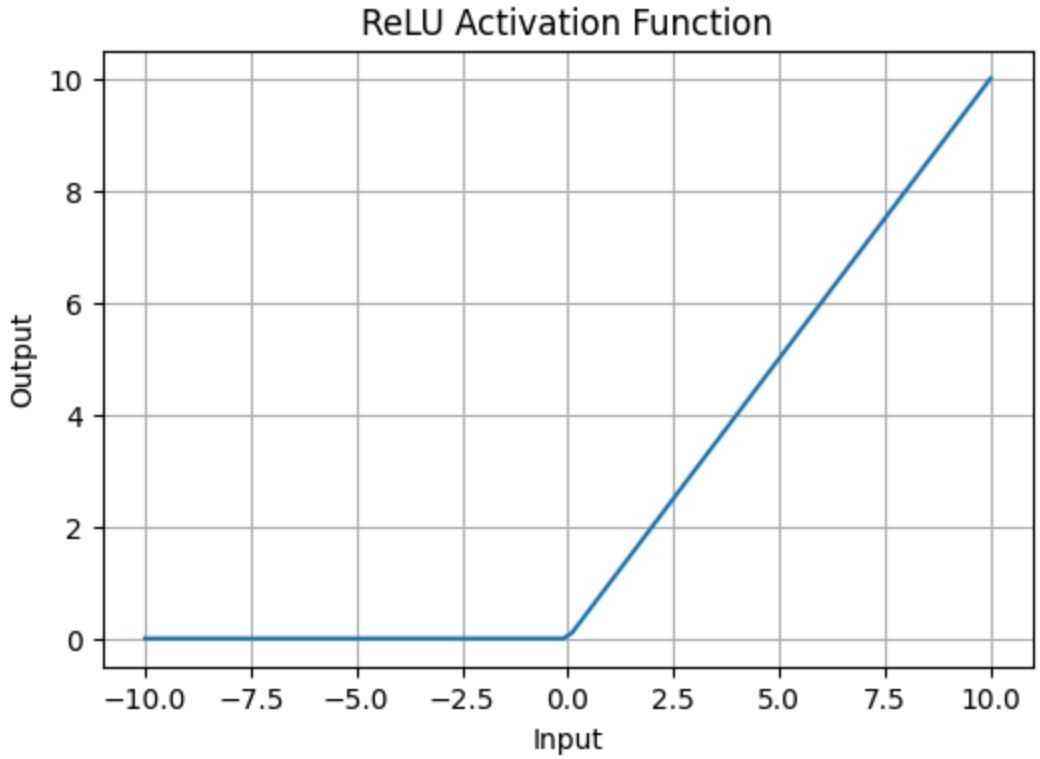
3. ReLU (Rectified Linear Unit) **Function**

ReLU activation is defined by

A(x)=max(0,x)

$A(x)=\max(0,x)$, this means that if the input x is positive, ReLU returns x, if the input is negative, it returns 0.

- **Value Range**: $[0,\infty)$, meaning the function only outputs non-negative values.
- **Nature**: It is a non-linear activation function, allowing neural networks to learn complex patterns and making backpropagation more efficient.
- **Advantage over other Activation:** ReLU is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations. At a time only a few neurons are activated making the network sparse making it efficient and easy for computation.

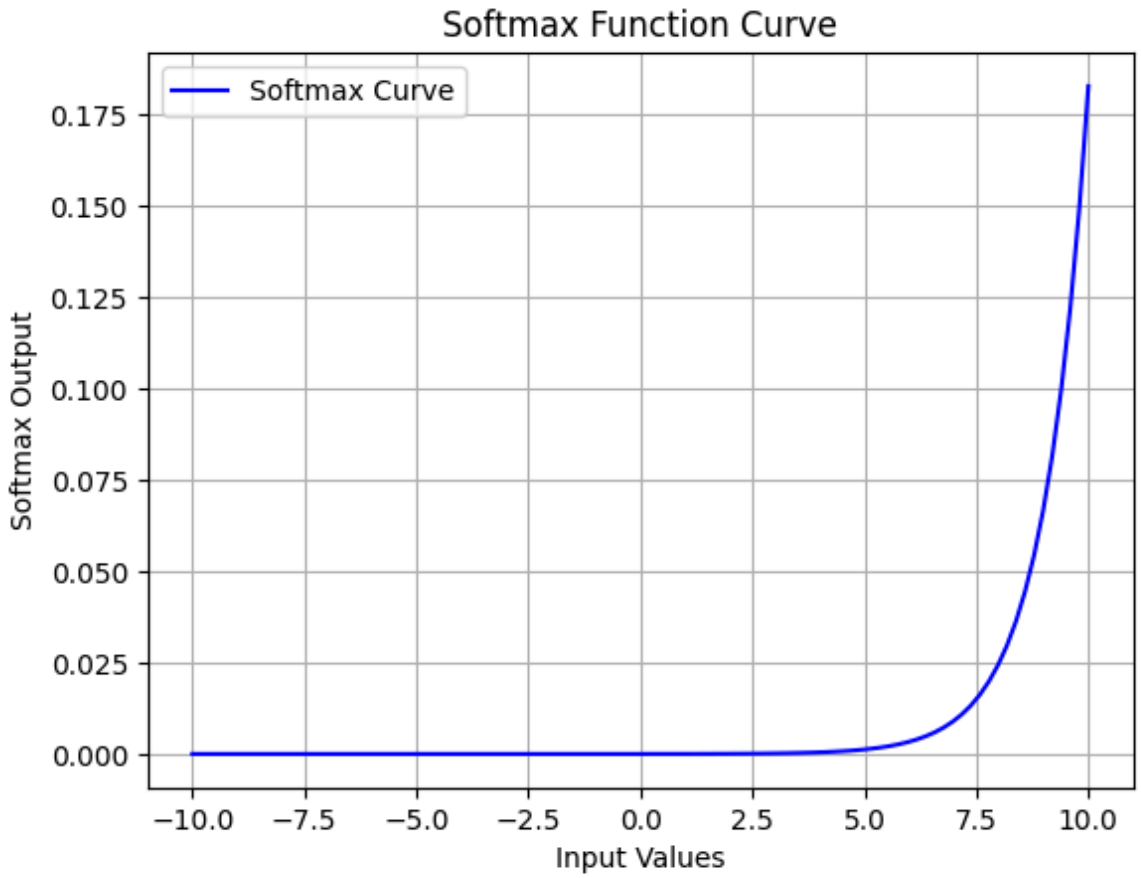


3. Exponential Linear Units

1. Softmax Function

Softmax function is designed to handle multi-class classification problems. It transforms raw output scores from a neural network into probabilities. It works by squashing the output values of each class into the range of 0 to 1 while ensuring that the sum of all probabilities equals 1.

- Softmax is a non-linear activation function.
- The Softmax function ensures that each class is assigned a probability, helping to identify which class the input belongs to.

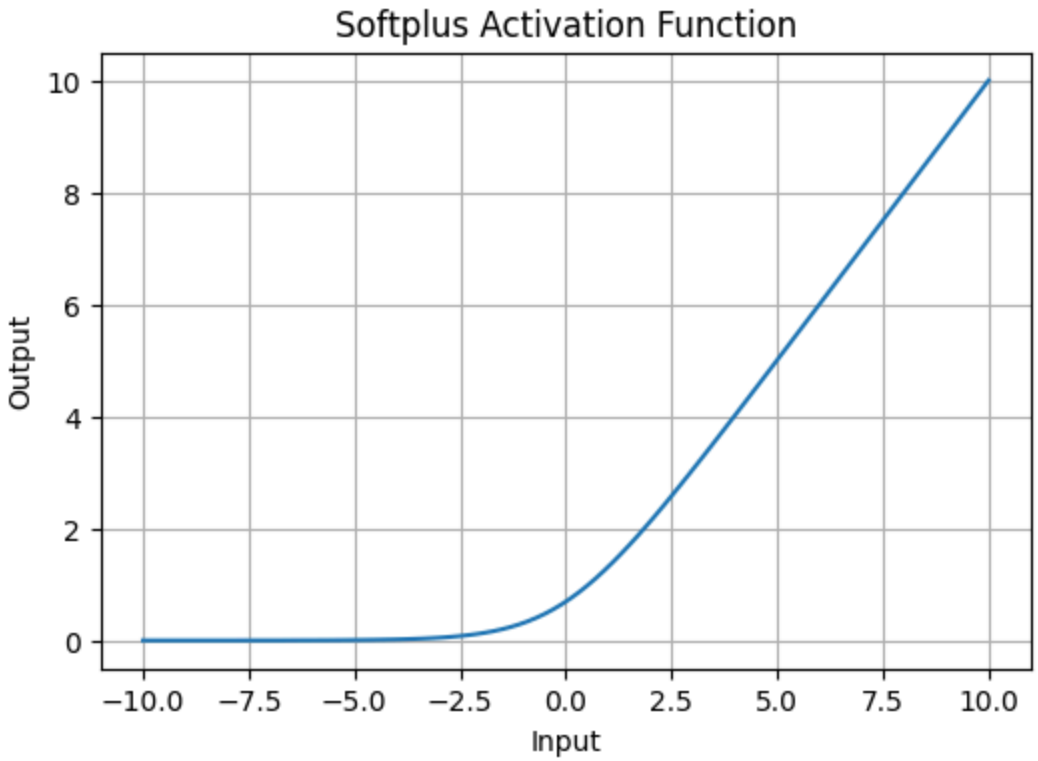


2. SoftPlus Function

Softplus function is defined mathematically as: $A(x)=\log(1+e^{x})$

This equation ensures that the output is always positive and differentiable at all points which is an advantage over the traditional ReLU function.

- **Nature**: The Softplus function is non-linear.
- **Range**: The function outputs values in the range $(0,\infty)$, similar to ReLU, but without the hard zero threshold that ReLU has.
- **Smoothness**: Softplus is a smooth, continuous function, meaning it avoids the sharp discontinuities of ReLU which can sometimes lead to problems during optimization.



Impact of Activation Functions on Model Performance

The choice of activation function has a direct impact on the performance of a neural network in several ways:

1. **Convergence Speed:** Functions like ReLU allow faster training by avoiding the vanishing gradient problem while Sigmoid and Tanh can slow down convergence in deep networks.
2. **Gradient Flow:** Activation functions like ReLU ensure better gradient flow, helping deeper layers learn effectively. In contrast Sigmoid can lead to small gradients, hindering learning in deep layers.
3. **Model Complexity:** Activation functions like Softmax allow the model to handle complex multi-class problems, whereas simpler functions like ReLU or Leaky ReLU are used for basic layers.

Optimization | Vanishing and Exploding Gradients Problems in Deep Learning

Optimization algorithms in deep learning are used to minimize the loss function by adjusting the weights and biases of the model. The most common ones are:

- Optimization algorithms in deep learning
- Gradient Descent
- Stochastic Gradient Descent (SGD)
- Batch Normalization
- Mini-batch Gradient Descent
- Adam (Adaptive Moment Estimation)
- Momentum-based Gradient Optimizer
- Adagrad Optimizer
- RMSProp Optimizer

Loss Functions

A **loss function** is a mathematical way to measure how good or bad a model's predictions are compared to the actual results. It gives a single number that tells us how far off the predictions are. The smaller the number, the better the model is doing. Loss functions are used to train models. Loss functions are important because they:

1. **Guide Model Training:** During training, algorithms such as Gradient Descent use the loss function to adjust the model's parameters and try to reduce the error and improve the model's predictions.
2. **Measure Performance:** By finding the difference between predicted and actual values, it can be used for evaluating the model's performance.
3. **Affect learning behavior:** Different loss functions can make the model learn in different ways depending on what kind of mistakes they make.

Learning Rate

The learning rate is a key hyperparameter in neural networks that controls how quickly the model learns during training. It determines the size of the steps taken to minimize the loss function. It controls how much change is made in response to the error encountered, each time the model weights are updated. It determines the size of the steps taken towards a minimum of the loss function during optimization.

In mathematical terms, when using a method like **Stochastic Gradient Descent (SGD)**, the learning rate (often denoted as α or η) is multiplied by the gradient of the loss function to update the weights:

$$w = w - \alpha \cdot \nabla L(w)$$

$$w = w - \alpha \cdot \nabla L(w)$$

Where:

- w represents the weights
- α is the learning rate
- $\nabla L(w)$ is the gradient of the loss function

Impact of Learning Rate on Model

The learning rate is a critical hyperparameter that directly affects how a model learns during training by controlling the magnitude of weight updates. Its value significantly affects both convergence speed and model performance.

Low Learning Rate:

- Leads to slow convergence
- Requires more training epochs
- Can improve accuracy but increases computation time

High Learning Rate:

- Speeds up training
- Risks of overshooting optimal weights
- May cause instability or divergence of the loss function

Optimal Learning Rate:

- Balances training speed and model accuracy
- Ensures stable convergence without excessive training time

Best Practices:

- Fine-tune the learning rate based on the task and model
- Use techniques like **learning rate scheduling** or **adaptive optimizers** to improve performance and stability

Identifying the ideal learning rate can be challenging but is important for improving performance without wasting resources.

Techniques for Adjusting the Learning Rate

1. Fixed Learning Rate

- A constant learning rate is maintained throughout training.
- Simple to implement and commonly used in basic models.

- Its limitation is that it lacks the ability to adapt to different training phases which may create sub optimal results.

2. Learning Rate Schedules

These techniques reduce the learning rate over time based on predefined rules to improve convergence:

- **Step Decay**: Reduces the learning rate by a fixed factor at set intervals (every few epochs).
- **Exponential Decay**: Continuously decreases the learning rate exponentially over training time.
- **Polynomial Decay**: Learning rate decays polynomially, offering smoother transitions compared to step or exponential methods.

3. Adaptive Learning Rate Methods

Adaptive methods adjust the learning rate dynamically based on gradient information, allowing better updates per parameter:

- **AdaGrad**: AdaGrad adapts the learning rate per parameter based on the squared gradients. It is effective for sparse data but may decay too quickly.
- **RMSprop:** RMSprop builds on AdaGrad by using a moving average of squared gradients to prevent aggressive decay.
- **Adam (Adaptive Moment Estimation):** Adam combines RMSprop with momentum to provide stable and fast convergence; widely used in practice.

4. Cyclic Learning Rate

- The learning rate oscillates between a minimum and maximum value in a cyclic manner throughout training.
- It increases and then decreases the learning rate linearly in each cycle.
- Benefits include better exploration of the loss surface and leading to faster convergence.

5. Decaying Learning Rate

- Gradually reduces the learning rate as training progresses.
- Helps the model take more precise steps towards the minimum. This improves stability in later epochs.

Achieving an optimal learning rate is essential as too low results in long training times while too high can lead to model instability. By using various techniques we optimize the learning process, ensuring accurate predictions without unnecessary resource expenses.

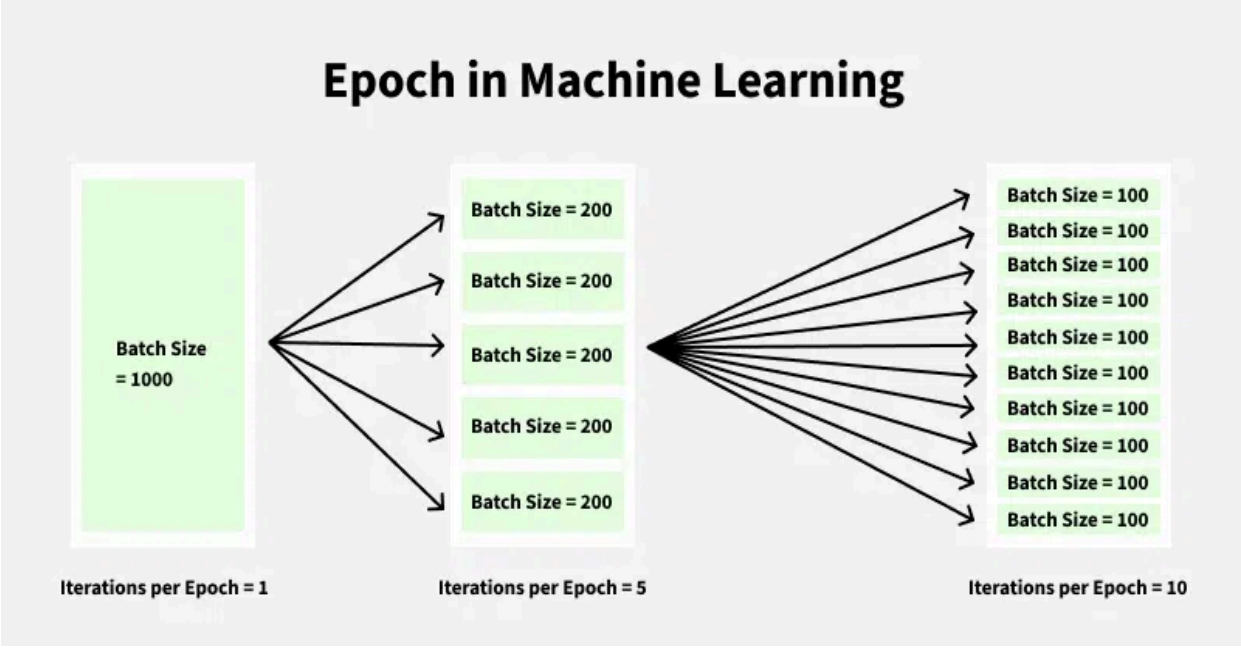
Epochs

In machine learning, an epoch refers to one complete pass through the entire training dataset where every data sample is passed through the model and its parameters are updated based on the calculated error. The training process requires multiple epochs, allowing the model to improve iteratively by adjusting its parameters based on the calculated error.

Example of an Epoch

In deep learning, datasets are usually divided into smaller subsets known as **batches**. The model processes these batches sequentially, updating the parameters after each batch. Batch size is a hyperparameter that plays an important role in determining how many samples are processed together which affects the frequency of updates.

- For example, if the training dataset has 1000 samples, one epoch would involve processing and updating the model with all 1000 samples in sequence.
- If the dataset has 1000 samples but a batch size of 100 is used then there would be only 10 batches in total. In this case, each epoch would consist of 10 iterations with each iteration processing one batch of 100 samples.



Typically, when training a model, the number of epochs is set to a large number like 100 and an early stopping method is used to determine when to stop training. This means that

the model will continue to train until either the validation loss stops improving or the maximum number of epochs is reached.

Now let's see how the data is fed to the model during training, this process involves splitting the data into smaller batches which are then processed in multiple iterations.

How Epochs, Batches and Iterations Work Together?
Understanding the relationship between epochs, batch size and iterations is important to optimize model training. Let's see how they work together:

- **Epochs Ensure Data Completeness:** An epoch represents one complete pass through the entire training dataset, allowing the model to refine its parameters with each iteration.
- **Batch Size affects training efficiency:** The batch size refers to how many samples are processed in each batch. A larger batch size allows the model to process more data at once, smaller batches on the other hand provide more frequent updates.
- **Iterations update the model:** An iteration occurs each time a batch is processed where the model finds the loss, adjusts its parameters and updates its weights based on that loss.

Learning Rate Decay and Its Role in Epochs

In addition to adjusting the number of epochs, the learning rate decay is an important technique that can further enhance model performance over time.

- Learning rate is a hyperparameter that controls how much the model's weights are adjusted during training. A high learning rate might cause the model to overshoot the optimal weight while a low learning rate can make the training slow.
- Learning rate decay is a technique where the learning rate gradually decreases during training. This helps the model make large adjustments at the start and more refined, smaller adjustments as it nears the optimal solution.

Using learning rate decay with multiple epochs ensures that the model doesn't overshoot during later stages of training. It helps the model to get an optimal solution which improves its performance.

Advantages of Using Multiple Epochs in Model Training
Using multiple epochs in machine learning is key to effective model training:

1. **Parameter Optimization:** Multiple epochs allow the model to refine its parameters over time, improving performance and accuracy especially for complex datasets where subtle patterns may emerge across multiple passes.
2. **Convergence Monitoring:** Training over multiple epochs allows for continuous monitoring of loss and performance which ensures the model is progressing toward the best solution.
3. **Early Stopping:** By tracking the model's performance over multiple epochs, early stopping helps stop the training when there's no significant improvement which prevents overfitting and saves computational resources.

Disadvantages of Overusing Epochs in Model Training

Training a model for too many epochs can lead to some common issues which are as follows:

1. **Overfitting Risk:** Training for too many epochs can cause the model to overfit where it memorizes the training data and loses the ability to generalize to new, unseen data.
2. **Increased Computational Cost:** Training for too many epochs, especially with large datasets can be computationally expensive and time-consuming.
3. **Model Complexity:** The optimal number of epochs varies depending on the complexity of the model and dataset. Too few epochs may lead to underfitting while too many can result in overfitting**.**
4. **Resource Drain:** Excessive epochs require more computational resources and time, potentially leading to inefficiencies especially with limited hardware.
5. **Balancing Act:** Finding the right number of epochs requires careful experimentation as an excessive or insufficient number can affect the model's performance.

By understanding epochs, batches and iterations, we can optimize our model's training process and fine-tune it for better performance.

Call backs

Callbacks are functions or blocks of code that are executed at specific stages of the training process. They allow you to interact with the model at various points such as:

- At the start and end of an epoch
- Before and after a batch is processed
- At the start and end of training

These interactions can be used to implement custom behavior such as early stopping, learning rate scheduling, saving model checkpoints, logging metrics, and more.

Example CNN Implementation

Lung Cancer Detection using Convolutional Neural Network (CNN)