Artificial Neural Network

Introduction to Artificial Neural Networks (ANNs)

- Artificial Neural Networks (ANNs) are computational models inspired by the human brain's structure and function. They are composed of interconnected processing elements, called neurons, which work together to solve specific problems. ANNs are capable of learning from data, making them a fundamental tool in machine learning and artificial intelligence.
- Historical Background
- **1943:** The concept of an artificial neuron was introduced by Warren McCulloch, a neurophysiologist, and Walter Pitts, a mathematician, laying the groundwork for neural networks.
- **1958:** Frank Rosenblatt developed the Perceptron, the first model capable of supervised learning, at the Cornell Aeronautical Laboratory.
- **1980s:** The backpropagation algorithm was popularized by David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams, enabling the training of multi-layer networks and sparking a resurgence of interest in neural networks.
- **1990s-2000s:** Advances in computational power and the availability of large datasets facilitated significant improvements in ANN capabilities.
- **2010s-present:** Deep learning, a subset of machine learning based on deep neural networks, has led to breakthroughs in many domains, firmly establishing ANNs in both research and practical applications.

Overview of Applications

- **Image and Speech Recognition:** ANNs have been fundamental in developing systems that accurately recognize images and speech. Applications include facial recognition, automated tagging of photographs, and voice-activated assistants.
- **Natural Language Processing (NLP):** From translation services to sentiment analysis, ANNs have significantly advanced the ability of computers to understand and interact using human language.
- **Predictive Analytics:** In finance, healthcare, and marketing, ANNs are used for predicting stock market trends, patient diagnoses, and consumer behavior, respectively.
- **Autonomous Vehicles:** ANNs are key in processing the vast amounts of data from sensors in real-time, allowing for safe navigation of self-driving cars.
- **Game Playing and Robotics:** ANNs enable sophisticated decision-making in AI opponents in games and increase the autonomy of robots in complex tasks.
- The development and application of ANNs have transformed countless industries, enhancing automation, improving decision-making processes, and opening new avenues for research and innovation. By mimicking the learning process of the human brain, ANNs provide a powerful tool for solving complex, nonlinear problems that were previously beyond our reach.

Basics of Artificial Neural Networks

- Artificial Neural Networks (ANNs) are structured in layers composed of nodes or artificial neurons, each layer designed to perform specific functions. The architecture of an ANN typically includes three main types of layers:

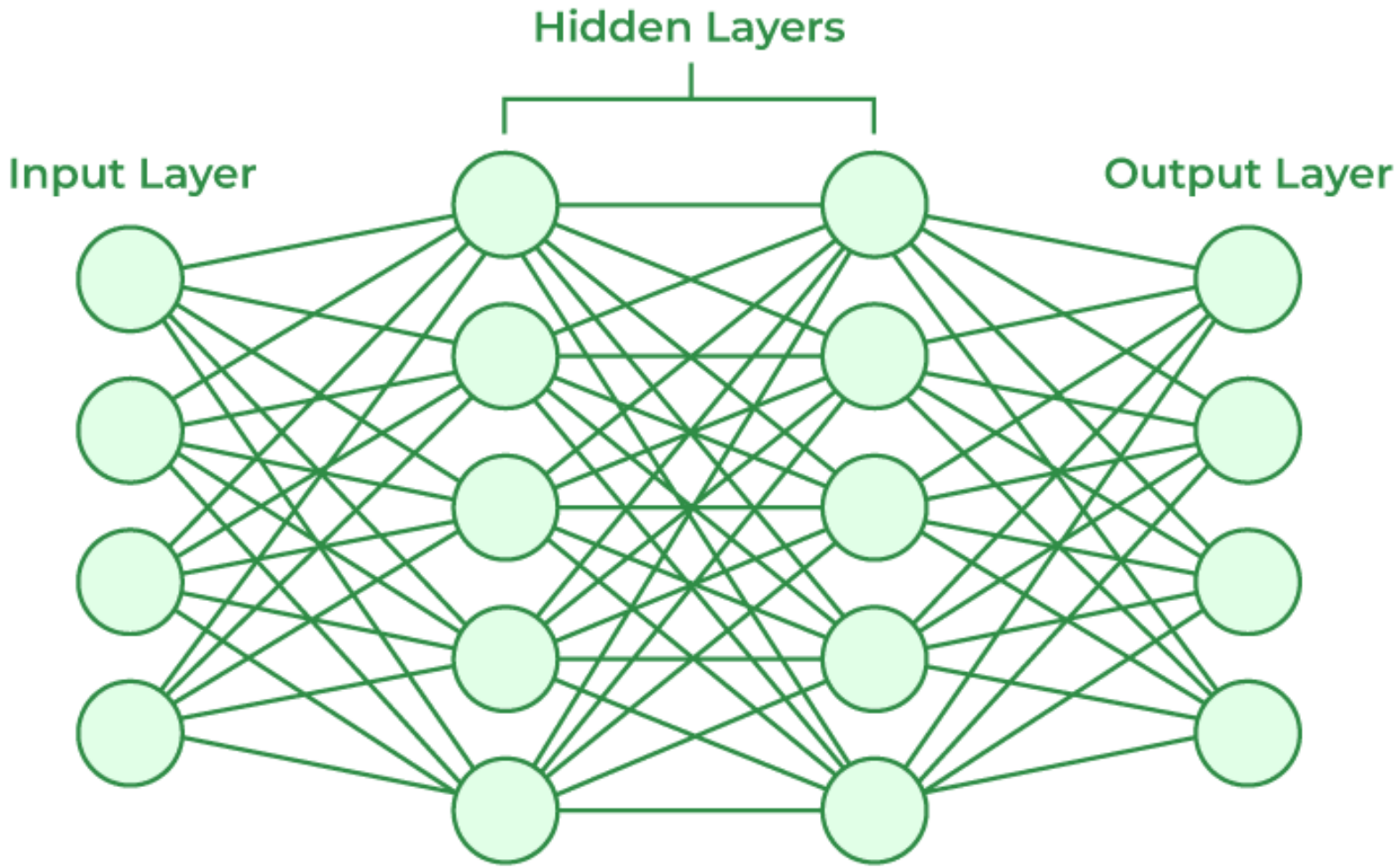
1. **Input Layer:** This is the first layer in the network, responsible for receiving the input from the external environment. Each neuron in the input layer represents a feature of the input dataset.
2. **Hidden Layers:** These layers lie between the input and output layers. Hidden layers perform various computations through their neurons, which are not directly exposed to the input or output. The complexity and capability of the ANN are largely determined by the number and size of its hidden layers. These layers enable the network to learn complex patterns by transforming the inputs into something that the output layer can use.
3. **Output Layer:** The final layer produces the output of the network. The design of the output layer—including the number of neurons—depends on the specific task (e.g., classification, regression). For classification, the output layer often uses a softmax function to represent the probability distribution over predicted classes.

How ANNs Mimic Decision Processes

- ANNs mimic the decision-making processes of the human brain through a combination of linear and non-linear transformations. Here's how:

1. **Learning from Examples:** Similar to how humans learn from experience, ANNs adjust their parameters (weights and biases) based on the input-output pairs provided during training. This adjustment is made to minimize the difference between the actual output and the predicted output by the network.
2. **Weighted Sum and Activation:** Each neuron in the network computes a weighted sum of its inputs and applies an activation function. The weighted sum represents the linear combination of inputs and their respective weights, akin to assessing the importance of each input. The activation function introduces non-linearity, allowing the network to learn complex patterns. This process is somewhat analogous to neurons in the brain that activate only when a certain threshold of signals is reached.
3. **Layer-wise Processing:** Information flows through the network from the input layer to the output layer, getting transformed at each step. This layered processing allows the ANN to build a hierarchy of learned features, starting from simple patterns in early layers to complex ones in deeper layers. It mirrors the human cognitive process, where simple perceptions gradually integrate into complex decisions.
4. **Feedback and Adjustment:** Through backpropagation and optimization algorithms like gradient descent, ANNs adjust their internal parameters in response to the error in their predictions. This continuous feedback loop enables the network to improve its accuracy over time, similar to how humans learn from mistakes and refine their decision-making processes.

- The structure and operation of ANNs are designed to emulate the human brain's ability to make decisions. By simulating the process of learning from examples, performing complex transformations, and adjusting based on feedback, ANNs have become a powerful tool for modeling and solving a wide range of problems that require nuanced decision-making capabilities.


Hidden Layers
Input Layer
Output Layer

The Perceptron

- The Perceptron is a fundamental building block in the field of neural networks, often referred to as the simplest form of a neural network. Invented in 1958 by Frank Rosenblatt, the Perceptron marked the beginning of machine learning by demonstrating that machines could learn from data. A Perceptron is essentially a single-layer neural network with a step activation function, designed to classify inputs into two possible categories (binary classification).

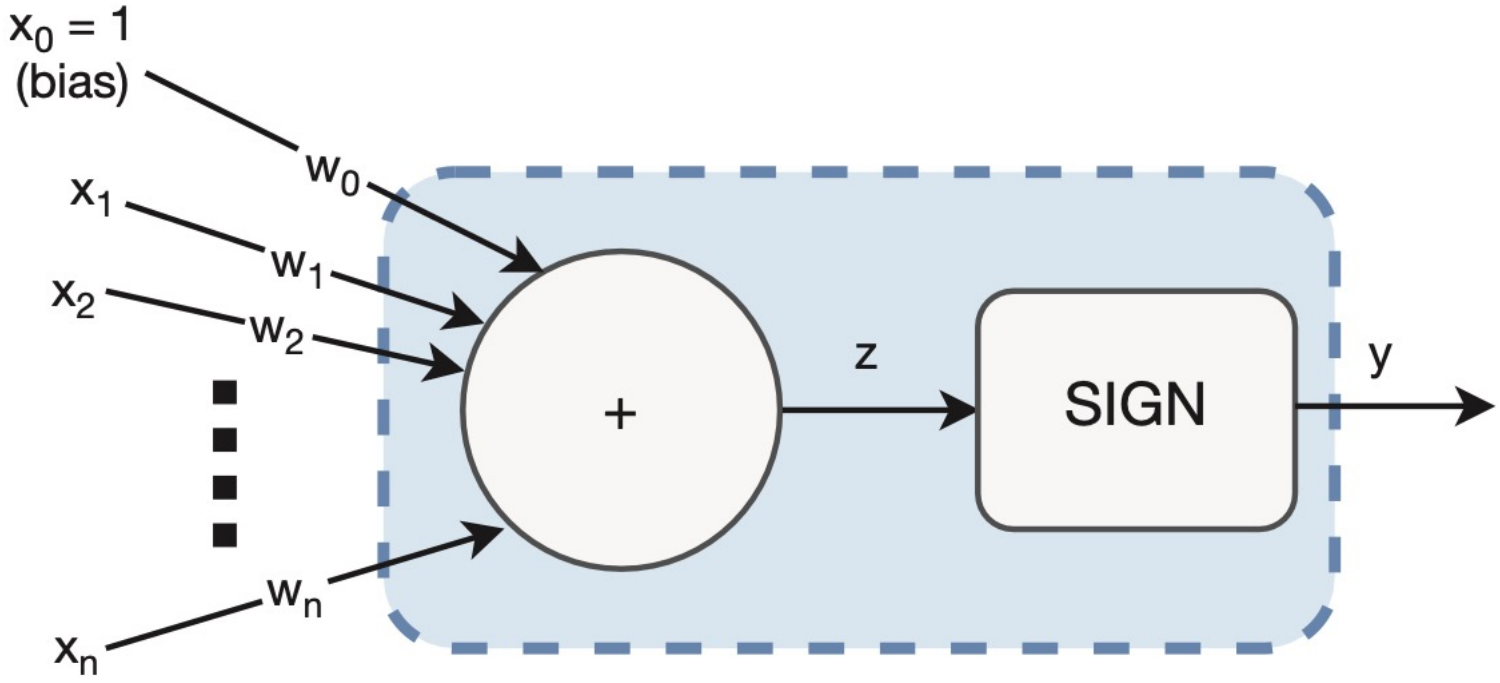
Mathematical Model

- At its core, the Perceptron model is based on a weighted sum of inputs, mathematically represented as:
- Where:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

- y is the output,
- w_i represents the weight associated with the i^{th} input x^i,
- b is the bias term that adjusts the threshold,
- f is the activation function, typically a step function that outputs either 0 or 1.

- The process involves multiplying each input by its corresponding weight, summing all these products together, adding a bias, and finally applying an activation function to determine the output.



Linear Separability

- A key concept related to the Perceptron is linear separability. A dataset is considered linearly separable if it is possible to separate its classes with a single straight line (in two dimensions) or a hyperplane in higher dimensions. The Perceptron algorithm finds this line (or hyperplane) by adjusting the weights and bias through a process known as learning.

Limitations and Significance

- The simplicity of the Perceptron comes with limitations, notably its inability to solve problems that are not linearly separable, such as the XOR problem. This limitation led to the development of Multi-layer Perceptrons (MLPs) and the exploration of deeper neural network architectures.
- Despite its limitations, the Perceptron remains a fundamental concept in neural network design, illustrating the principles of learning, weight adjustment, and decision boundaries.

- To summarize, the perceptron will output -1 if the weighted sum is less than zero,
- and otherwise it will output 1. Written as an equation, we have the following:

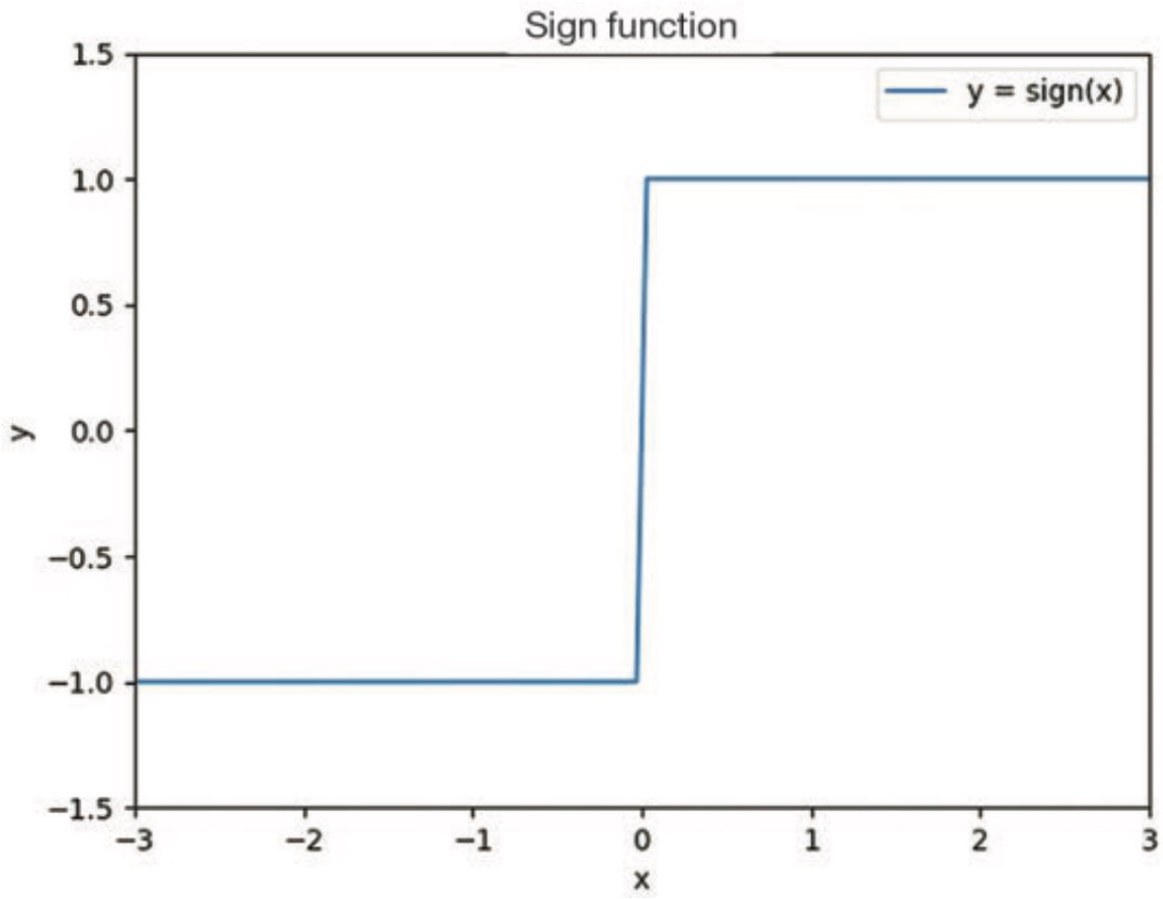
$$y = f(z), \text{ where}$$

$$z = \sum_{i=0}^{n} w_i x_i$$

$$f(z) = \begin{cases} -1, & z < 0 \\ 1, & z \geq 0 \end{cases}$$

$$x_0 = 1 \text{ (bias term)}$$

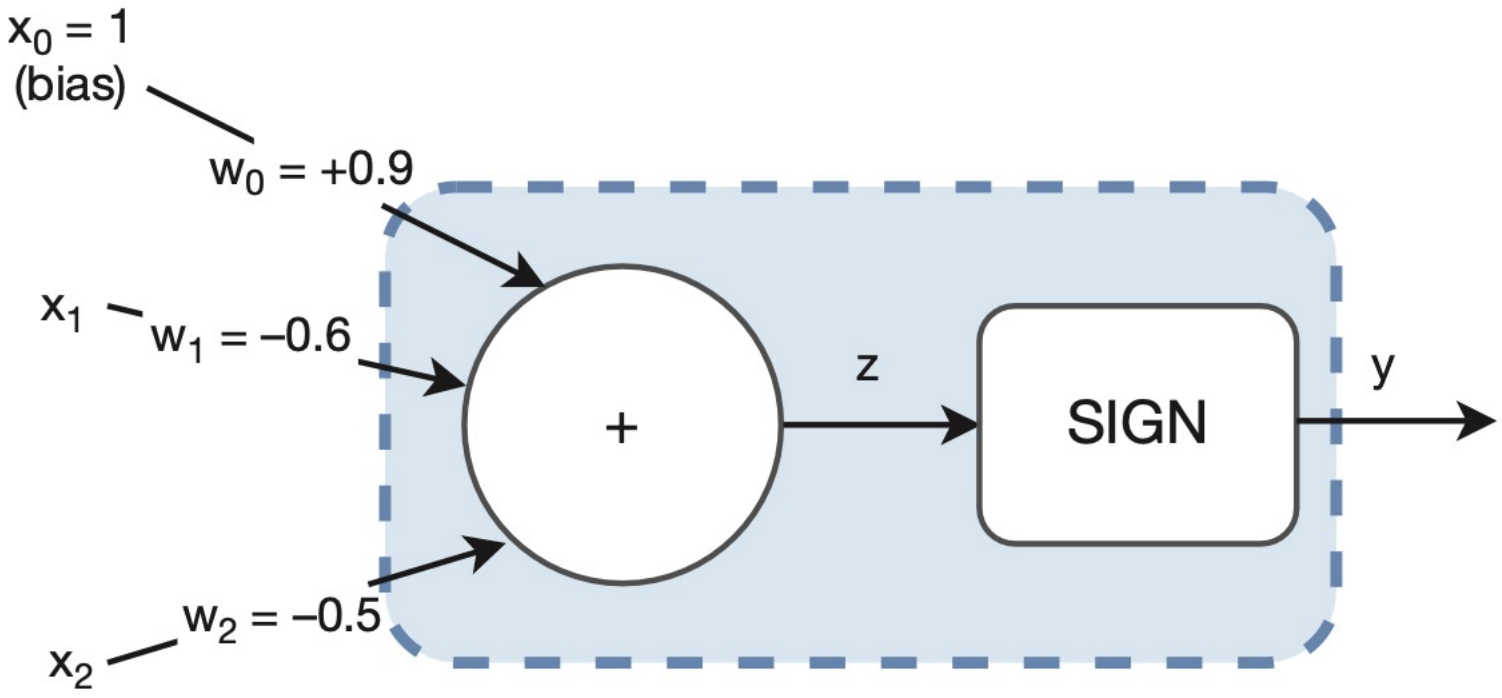
Activation Function



Python Implementation of Perceptron Function

```
# First element in vector x must be 1.
# Length of w and x must be n+1 for neuron with n inputs.
def compute_output(w, x):
    z = 0.0
    for i in range(len(w)):
        z += x[i] * w[i] # Compute sum of weighted inputs
    if z < 0: # Apply sign function
        return -1
    else:
        return 1
```

Example of a Two-Input Perceptron



X_0	X_1	X_2	$W_0 * X_0$	$W_1 * X_1$	$W_2 * X_2$	Z	Y
1	−1 (False)	−1 (False)	0.9	0.6	0.5	2.0	1 (True)
1	1 (True)	−1 (False)	0.9	−0.6	0.5	0.8	1 (True)
1	−1 (False)	1 (True)	0.9	0.6	−0.5	1.0	1 (True)
1	1 (True)	1 (True)	0.9	−0.6	−0.5	−0.2	-1 (False)

- In our example, we have four sets of input/output data, each corresponding to one row in Table 1-1. The algorithm works as follows:

1. Randomly initialize the weights.
2. Select one input/output pair at random.
3. Present the values $x_1, \ldots, x_n$ to the perceptron to compute the output y.
4. If the output y is different from the ground truth for this input/output pair, adjust the weights in the following way:

a. If $y < 0$, add η *(The Learning Rate)* x_i to each w_i.

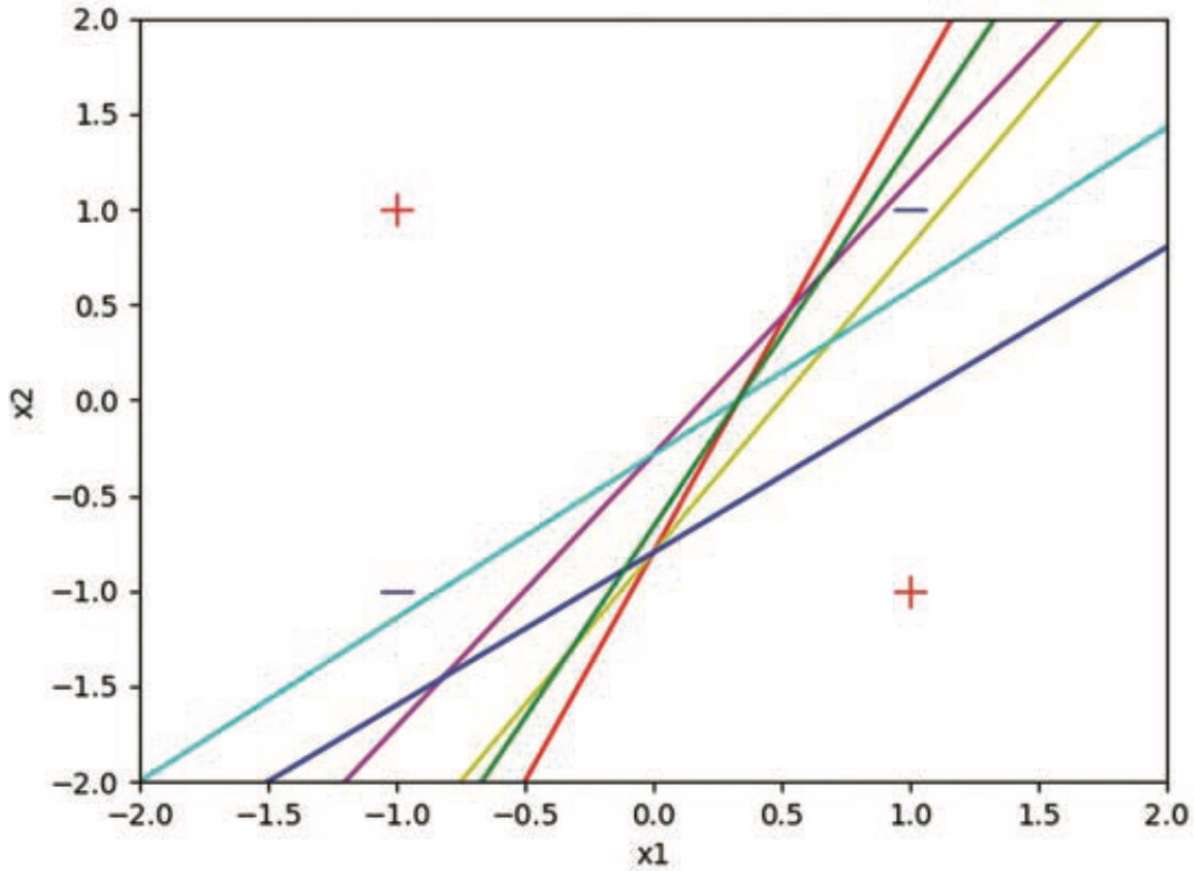
b. If $y > 0$, subtract η x_i from each w_i.

5. Repeat steps 2, 3, and 4 until the perceptron predicts all examples correctly.

See ANN.pynb

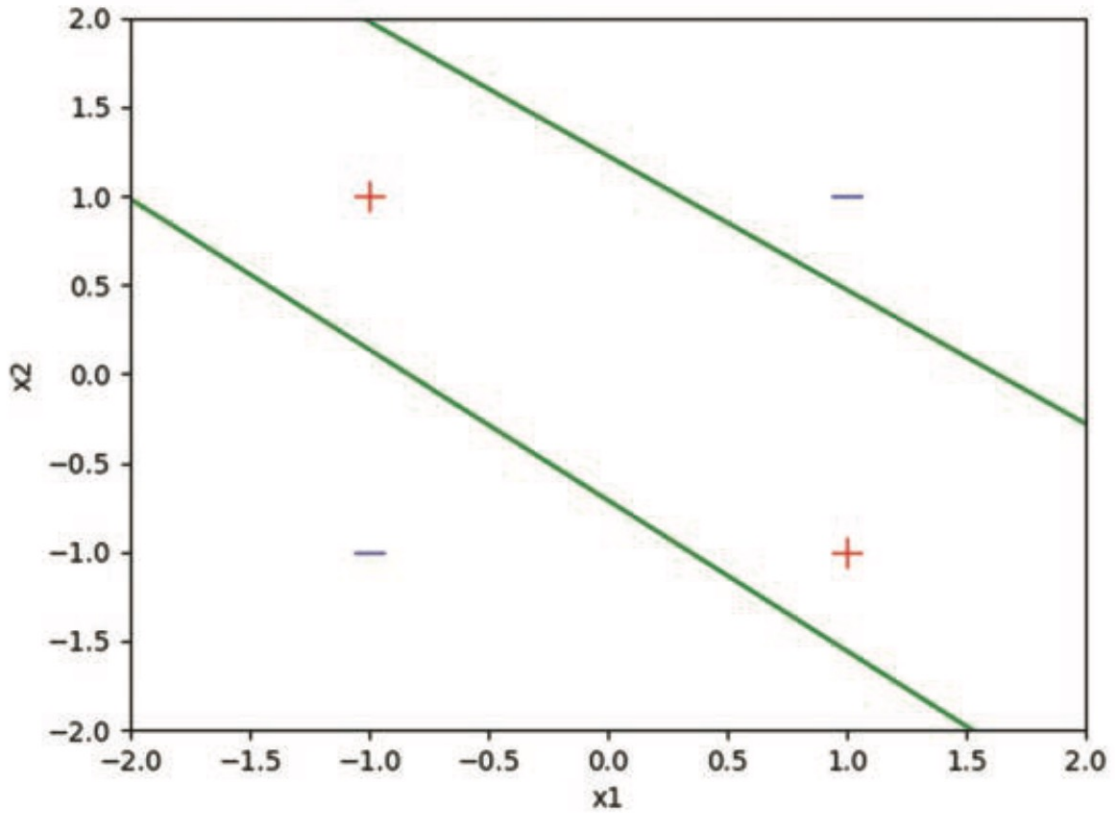
- Code Cell
- `6.1 Perceptron`

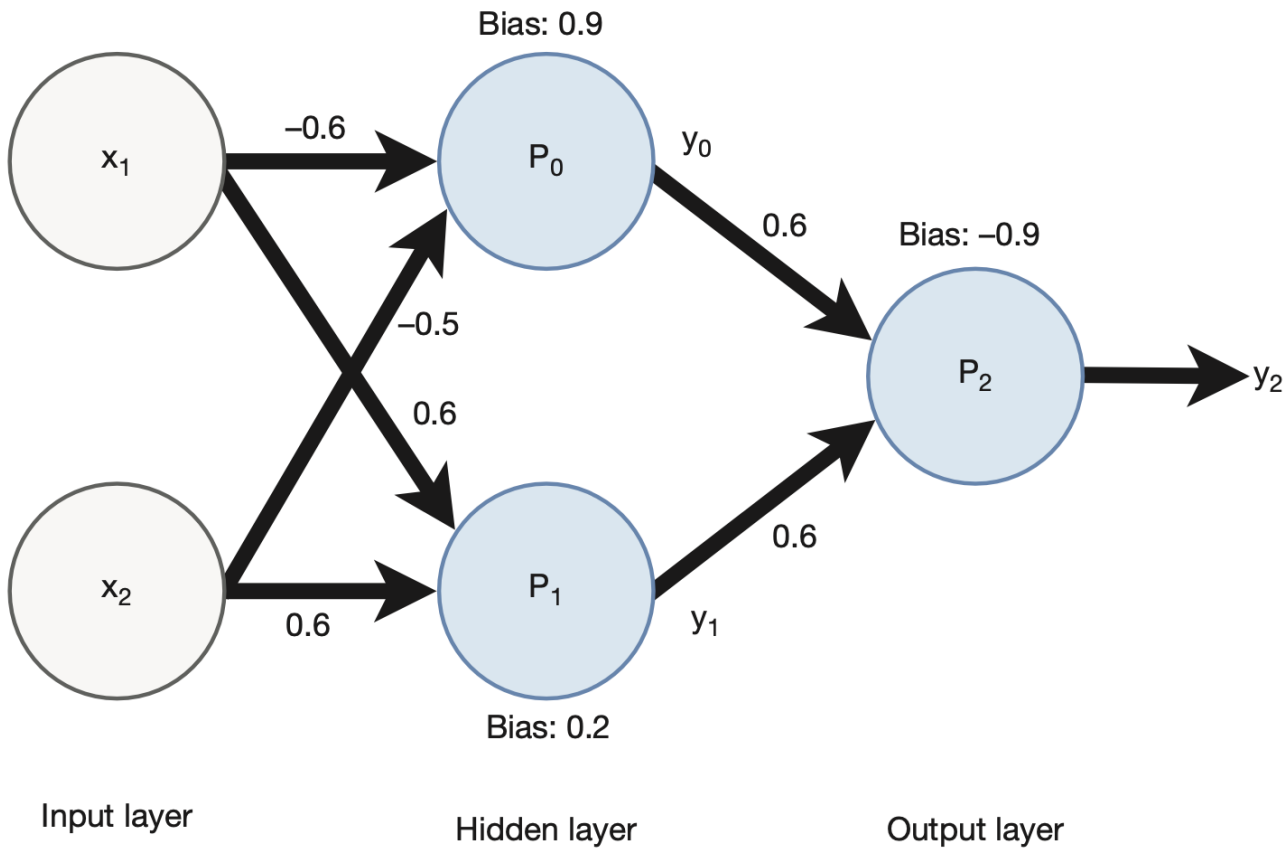
XOR Gate



Combining Multiple Perceptrons

- As shown previously, a single perceptron can separate the chart into two regions, illustrated by drawing a straight line on the chart. That means that if we add another perceptron, we can draw another straight line.




Bias: 0.9
x_1
−0.6
P_0
y_0
0.6
Bias: −0.9
−0.5
P_2
y_2
0.6
x_2
P_1
0.6
y_1
Bias: 0.2
Input layer
Hidden layer
Output layer

X_0	X_1	X_2	Y_0	Y_1	Y_2
1	–1 (False)	–1 (False)	1.0	–1.0	–1.0 (False)
1	1 (True)	–1 (False)	1.0	1.0	1.0 (True)
1	–1 (False)	1 (True)	1.0	1.0	1.0 (True)
1	1 (True)	1 (True)	–1.0	1.0	–1.0 (False)

Combinations of Perceptrons and Input Examples and the Corresponding Linear Algebra Operation

NUMBER OF PERCEPTRONS	NUMBER OF INPUT EXAMPLES	LINEAR ALGEBRA OPERATION
One	One	Dot product
Multiple	One	Matrix-vector multiplication
Multiple	Multiple	Matrix-matrix multiplication

Gradient-Based Learning

- Weight Update Step of Perceptron Learning Algorithm

```
for i in range(len(w)):
   w[i] += (y * LEARNING_RATE * x[i])
```

- w is an array representing the weight vector, x is an array representing the input vector, and y is the desired output.

- Gradient Descent is a cornerstone optimization algorithm within the realm of Artificial Neural Networks and broader machine learning domains. It is elegantly simple yet profoundly effective, aimed at iteratively minimizing the cost (or loss) function, which measures the difference between the model's predictions and the actual data. By navigating the multi-dimensional landscape of the cost function, Gradient Descent seeks the lowest point, analogous to a ball rolling down a hill, which corresponds to the model's optimal parameters.
- At the heart of Gradient Descent lies the cost function, a mathematical formulation that quantifies the error of the model's predictions. In ANNs, common cost functions include Mean Squared Error (MSE) for regression tasks and Cross-Entropy for classification. The choice of the cost function is crucial as it directly influences the direction and efficiency of the descent through the error landscape.

- Imagine standing on a foggy mountain range, where your goal is to find the lowest valley. Your visibility (data) is limited, and you can only feel the slope (gradient) under your feet to decide your next step. Gradient Descent operates under a similar principle, where it uses the gradient of the cost function with respect to each parameter (weight and bias in the context of ANNs) to guide its steps towards the global minimum. Each step is determined by the gradient multiplied by the learning rate, a hyperparameter that dictates the size of the steps.
-

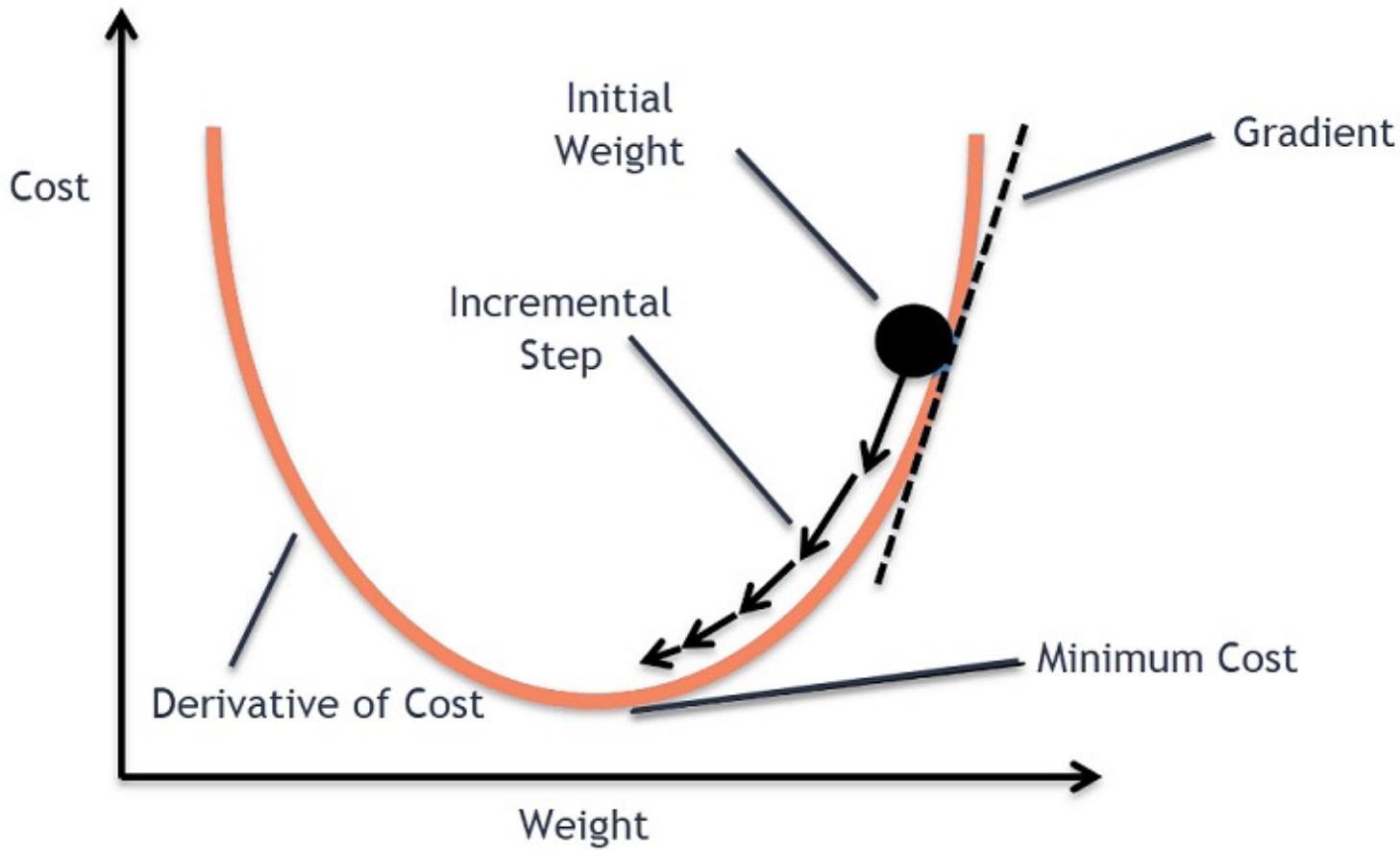
Common Cost Functions in ANNs

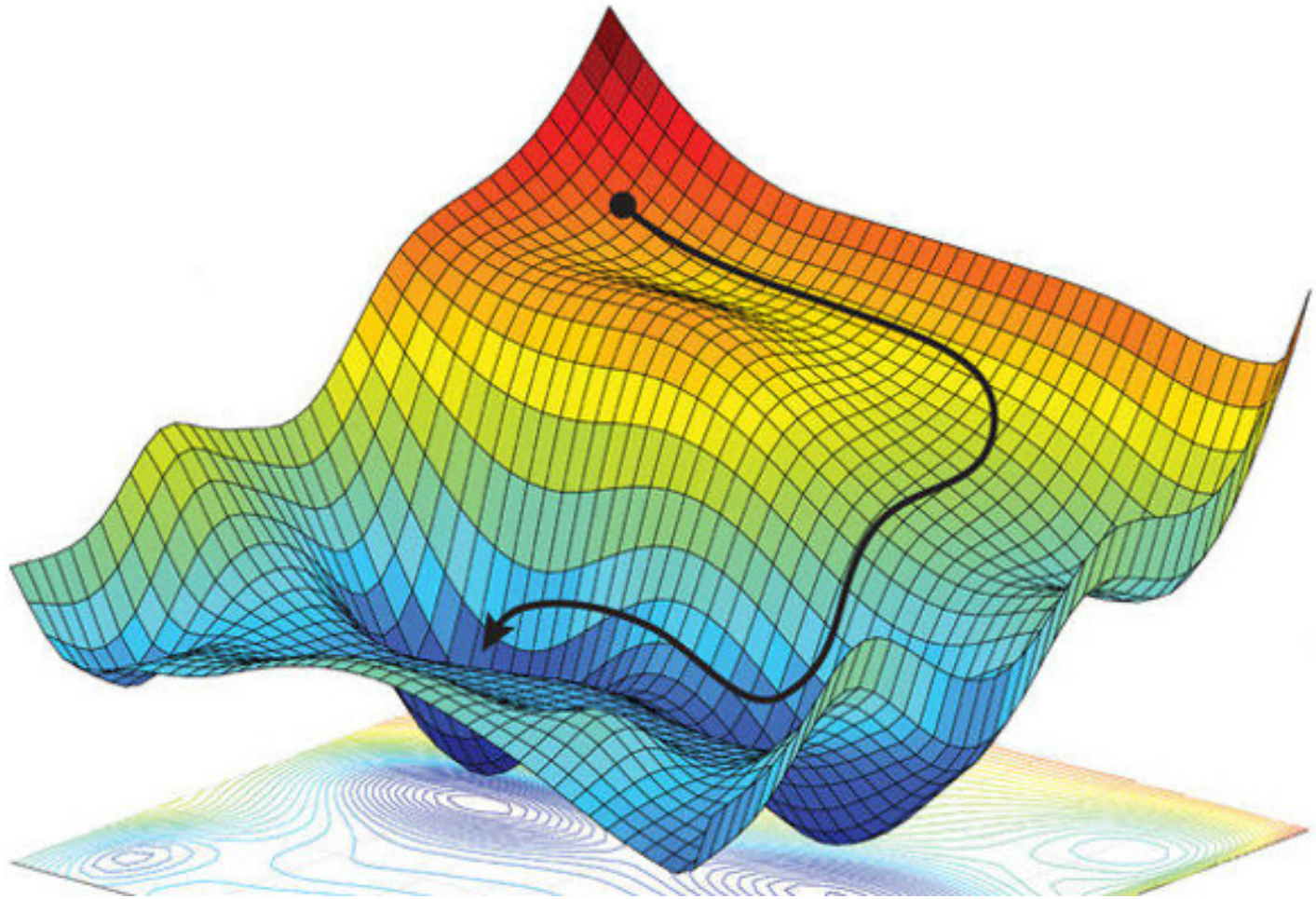
- **Mean Squared Error (MSE):** Predominantly used in regression problems, MSE calculates the average of the squares of the differences between the predicted and actual values. It is mathematically represented as:

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

- where y_i are the actual values, $\hat{y}_i$ are the predicted values, and n is the number of observations. MSE is favored for its simplicity and the fact that it penalizes larger errors more severely, encouraging more accurate predictions.

- **Cross-Entropy:** Also known as Log Loss, Cross-Entropy is crucial for classification tasks. It measures the dissimilarity between the true label distribution and the predictions, providing a more nuanced understanding of classification performance than mere accuracy. For binary classification, its formula is: $$H(y, \hat{y}) = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$
- Cross-Entropy is particularly effective because it exacerbates the penalty for predictions that are confidently wrong, steering the model more strongly toward correct classification.
- The Role of Cost Functions in Model Optimization
- The choice of cost function goes beyond mere convention; it is a strategic decision that aligns the model's learning objective with the problem at hand. By quantifying error in a way that is most penalizing to the types of mistakes most costly to the task, the cost function ensures that the model not only learns but learns correctly. In the landscape of ANNs, where models must navigate through high-dimensional data to make predictions, the cost function is the beacon that lights the path to optimal performance.


Cost
Initial
Weight
Gradient
Incremental
Step
Derivative of Cost
Minimum Cost
Weight



- The gradient, denoted as $\nabla J(\theta)$, where $J(\theta)$ is the cost function and θ represents the parameters of the model (e.g., weights and biases in ANNs), is a vector containing all the partial derivatives of the cost function with respect to each parameter. It points in the direction of the steepest ascent of J. By moving in the opposite direction, we aim to find the lowest point of the function, akin to descending a mountain in the steepest path to the valley.

- The core of Gradient Descent is encapsulated in the update rule, a formula that iteratively adjusts the parameters to minimize the cost function:
- Here: $\theta_{new} = \theta_{old} - \alpha \cdot \nabla J(\theta_{old})$
- θ_{new} and θ_{old} are the values of the parameters before and after an update,
- α is the learning rate, a scalar that determines the size of the step to take in the direction opposite to the gradient,
- $\nabla J(\theta_{old})$ is the gradient of the cost function evaluated at θ_{old}.
- The learning rate α is a critical hyperparameter in Gradient Descent. It controls how big a step is taken on each iteration. If α is too small, the descent will be slow, potentially requiring many iterations to converge to the minimum. If α is too large, there's a risk of overshooting the minimum, resulting in divergence. Finding the right balance for α is crucial for efficient and effective optimization.

- Convergence Criteria
- Gradient Descent iterates this process until a convergence criterion is met, which could be a small change in cost function value between iterations, a small change in parameter values, or reaching a predetermined number of iterations. This iterative nature allows the algorithm to be applied to very large datasets and complex models, including deep neural networks, by incrementally improving model parameters based on the available data.

The Gradient of the Loss Function

- To minimize the MSE loss, we use gradient descent, which requires calculating the gradient of the loss function with respect to the weights vector *w*. The gradient tells us the direction to adjust *w* to decrease the loss.

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

- The partial derivative of the loss function with respect to each weight w_j is:

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

- where x_{ij} is the value of feature *j* for sample *i*. This can be vectorized for all weights as:

$$\nabla_w L = \frac{-2}{n} X^T (y - \hat{y})$$

The compute_gradient Function

- The compute_gradient function implements the vectorized formula for the gradient of the MSE loss. Here's a step-by-step breakdown:

1. **Predictions**: First, it calculates $\hat{y}=X.w$, which are the predictions made by the model given the current weights w.
2. **Errors**: Next, it computes the error vector $y-\hat{y}$, which represents the difference between the actual values and the predictions.
3. **Gradient Calculation**: Finally, it calculates the gradient of the loss function with respect to the weights using the formula:

$$\nabla_w L = \frac{-2}{n} X^T (y - \hat{y})$$

- X^T is the transpose of the input matrix, which when multiplied by the error vector $y-\hat{y}$, gives a vector where each element corresponds to the partial derivative of the loss with respect to each weight.
- The multiplication by $\frac{-2}{n}$ scales the sum of the errors and normalizes it by the number of samples, which helps in controlling the step size in the gradient descent update.
- This gradient is used in the gradient descent optimization algorithm to adjust the weights w iteratively to minimize the loss function L.

See ANN.pynb

- Code Cell
- 6.2 Gradient Descent
- 6.3 ANN Library

Activation Functions

- They introduce non-linearities into the network, allowing it to learn complex patterns. Without non-linearity, a neural network, regardless of how many layers it had, would behave just like a single-layer network.
- **1. Sigmoid**
- **Formula**: $f(x) = \frac{1}{1 + e^{-x}}$
- **Characteristics**: Smooth, outputs between 0 and 1. Historically popular for output layers in binary classification.
- **Best Use**: Binary classification tasks in the output layer. Less used in hidden layers due to vanishing gradient problems.

- **2. Hyperbolic Tangent (Tanh)**
- **Formula**: $f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- **Characteristics**: Outputs between -1 and 1, making it zero-centered and better for convergence in some cases than sigmoid.
- **Best Use**: Hidden layers in many architectures, especially when the data is centered around zero.
-

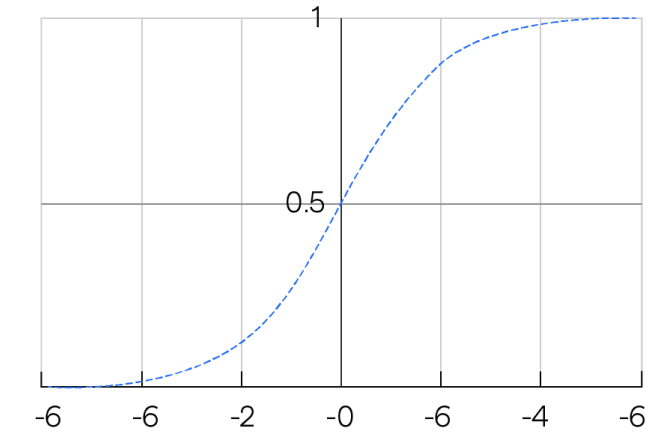
- **3. ReLU (Rectified Linear Unit)**
- **Formula**: $f(x)=\max(0,x)$
- **Characteristics**: Outputs the input directly if it is positive, otherwise it will output zero. It has become very popular due to its computational efficiency and sparsity.
- **Best Use**: Widely used in hidden layers across various network architectures due to its effectiveness and efficiency.

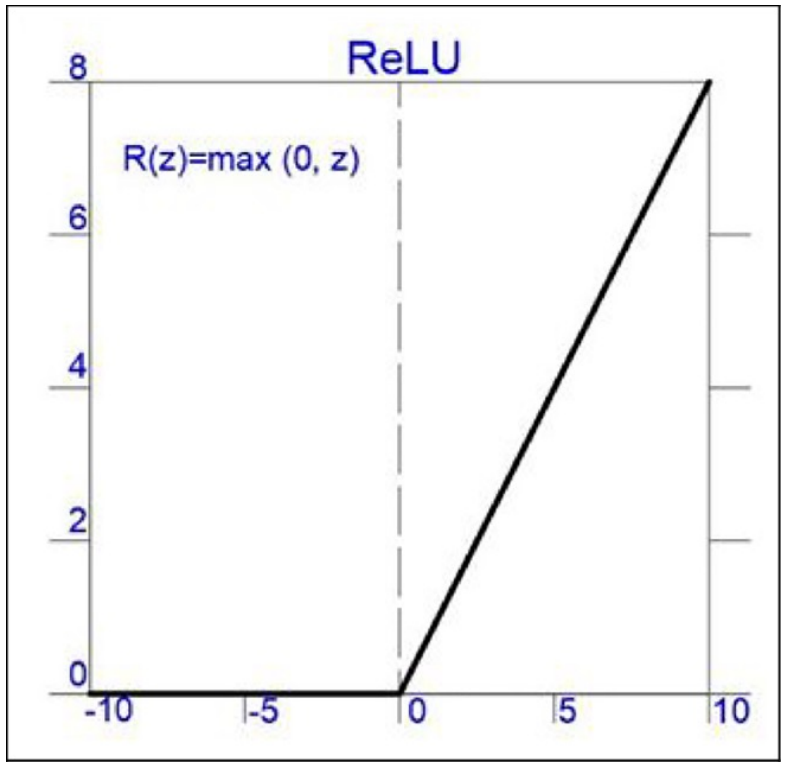
- **4. Softmax**
- **Formula**: $f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$
- **Characteristics**: Converts logits, the raw outputs of the last layer of a network, into probabilities by taking the exponential of each output and then normalizing these values by dividing by the sum of all the exponentials.
- **Best Use**: Output layer of a multi-class classification problem where outputs are mutually exclusive.

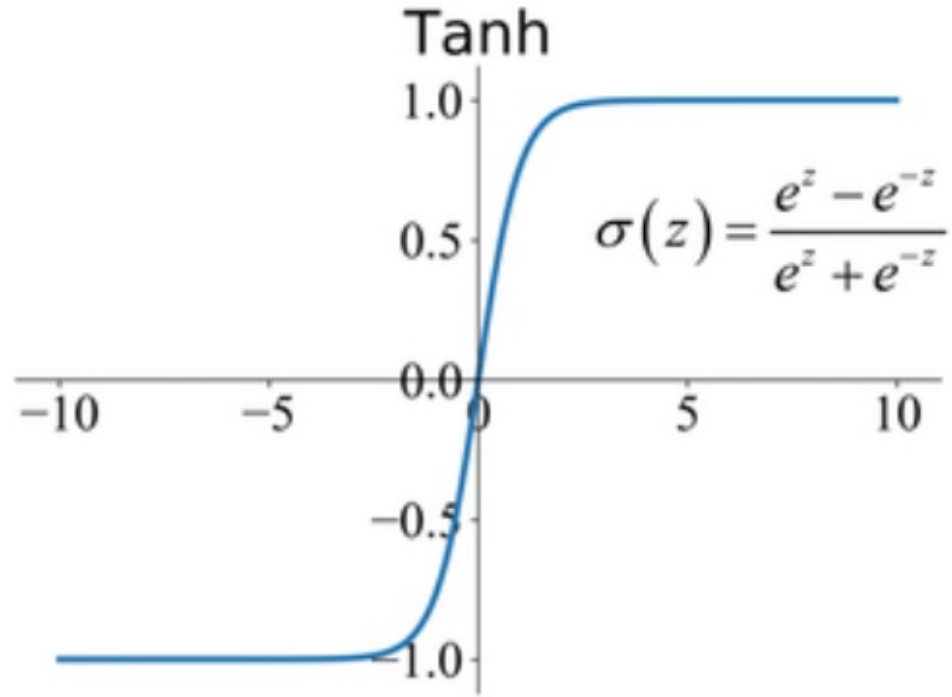
Activation Function	Output Range	Pros	Cons	Best Use Case
Sigmoid	(0, 1)	Easy interpretation as probability	Vanishing gradient, not zero-centered	Binary classification (output layer)
Tanh	(-1, 1)	Zero-centered	Vanishing gradient	Hidden layers when data is centered around 0
ReLU	[0, ∞)	Computationally efficient, sparsity	Dying ReLU problem	General hidden layers
Softmax	(0, 1)	Outputs interpreted as probabilities	Computationally intensive	Multiclass classification (output layer)

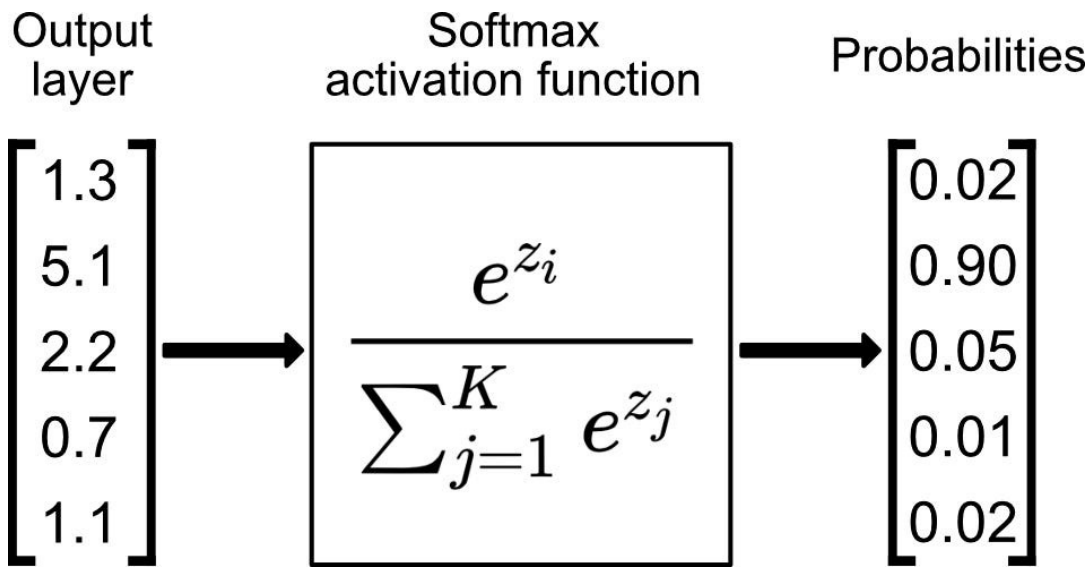
Sigmoid Function

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









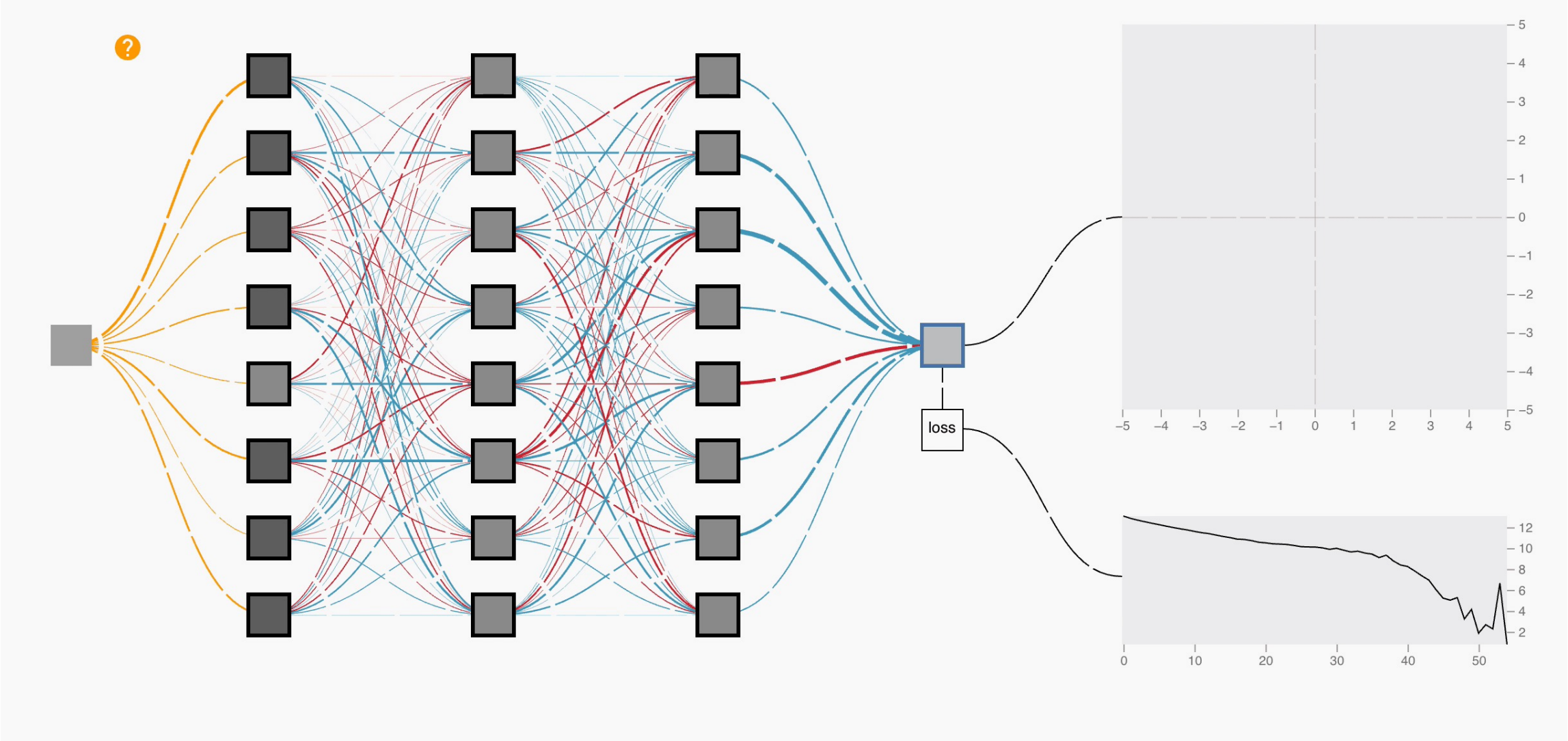
Backpropagation

- It's the mechanism by which neural networks learn from the error of their predictions and adjust their parameters (weights and biases) accordingly. The goal of backpropagation is to minimize the difference between the predicted output and the actual output (often referred to as the loss or cost) by adjusting the model's parameters. Here's a step-by-step explanation suitable for grad students:
- **1. Forward Pass**
- **Initialization**: Begin with a neural network with initialized weights and biases.
- **Input**: Feed the input data into the network.
- **Computation**: Progress through the layers of the network, calculating the output of each neuron. This is done by applying the weights and biases to the inputs and passing the result through an activation function.
- **Output**: Obtain the final output of the network, which is the prediction for the given input.
- **2. Calculate Loss**
- Compare the predicted output to the actual target output using a loss function. This function quantifies how far the prediction is from the target. Common loss functions include Mean Squared Error (MSE) for regression tasks and Cross-Entropy Loss for classification tasks.

- **3. Backward Pass (Backpropagation)**
- **Gradient of the Loss Function**: Compute the gradient of the loss function with respect to the output. This tells us how to change the output layer weights to reduce the loss.
- **Propagation of the Gradient**: Move backward through the network, layer by layer, using the chain rule of calculus to recursively calculate the gradient of the loss with respect to the weights and biases of each layer. This process effectively determines the contribution of each weight and bias to the total error.
- **Update Weights and Biases**: Adjust the weights and biases in the direction that minimally reduces the error, using the gradients calculated during backpropagation. The size of the step taken in this direction is determined by the learning rate, a hyperparameter that needs to be chosen carefully.
- **4. Iterate**
- Repeat the forward pass, loss calculation, and backpropagation for many iterations (epochs) over the training data. With each iteration, the network weights and biases are adjusted to minimize the loss, and hence, improve the model's accuracy.

- **Key Concepts:**
- **Gradient Descent**: The process of adjusting parameters in the direction that minimally reduces the error is known as gradient descent.
- **Learning Rate**: A hyperparameter that controls how much we are adjusting the weights of our network with respect to the loss gradient. Too small a learning rate can make a model learn very slowly, while too large a learning rate can make the model's steps too large, potentially overshooting the minimum or failing to converge.
- **Chain Rule of Calculus**: Essential for efficiently computing gradients through layers. It allows the gradient of the loss to be backpropagated through the network by localizing the computation of gradients at each node.
- **Practical Considerations:**
- **Initialization**: The initial values of weights can significantly affect the training process and the final model performance.
- **Overfitting**: A model that has learned the training data too well and performs poorly on unseen data. Techniques like regularization and dropout are used to prevent overfitting.
- **Optimization Algorithms**: While the basic form of backpropagation typically uses gradient descent, more sophisticated optimization algorithms like Adam and RMSprop can lead to faster convergence.

https://xnought.github.io/backprop-explainer/



Learning the XOR Function

- See ANN.pynb
- Code Cell
- `6.4 Backpropagation`

We did it! We have now reached the point in neural network research that was state-of-the art in 1986!