Activation Functions

Importance of Activation Functions in Neural Networks

Activation functions play a pivotal role in the architecture of neural networks. These functions help determine the output of each neuron in a network, introducing non-linearity that is essential for deep learning models to capture complex patterns and relationships within the data. Without activation functions, a neural network would essentially behave as a linear regressor, unable to model intricate relationships in large datasets. By applying an activation function to the output of a neuron, the model gains the flexibility to approximate almost any kind of function, which is fundamental for deep learning's success.

Chain Rule and Vanishing Gradient Problem in Activation Functions

Activation functions, especially those with saturation regions like Sigmoid and Tanh, contribute to the vanishing gradient problem. This problem occurs because gradients diminish as they propagate back through layers during training, causing weight updates to be very small and slow, especially for deep networks. This challenge can be mitigated by using activation functions like ReLU and Leaky ReLU, which do not saturate for positive values.

The **chain rule** of derivatives is used to compute the gradient of the loss function with respect to each weight in the network. However, if the activation function saturates (like Sigmoid or Tanh), the gradient can become very small, making it difficult to propagate useful gradients backward through the network, which impedes learning.

Below is an explanation of some common activation functions used in deep learning, including their formulas and key features:

---

1. Linear Activation Function

Formula:

$$f(x) = x$$

The linear activation function outputs the input as-is, which means it does not introduce non-linearity. This makes it suitable for linear regression tasks but ineffective in deep neural networks, as it doesn't enable the model to learn complex patterns due to the lack of non-linearity.

Key Features:

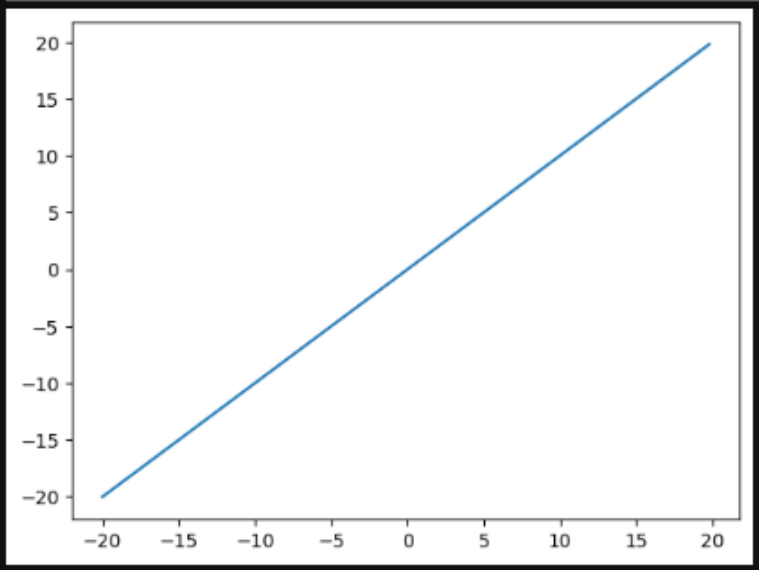
- **Unbounded**: Outputs can range from negative to positive infinity.

- **Limited Use**: Primarily used in the output layer of regression problems but rarely in hidden layers of deep networks.
- **Gradient**: The gradient is constant (1), leading to straightforward backpropagation.

Example:

Linear Activation Function

```
[16]: import numpy as np
      import matplotlib.pyplot as plt

      # Define the Linear Function
      def LinearFunction(number):
          a = []  # Initialize an empty list to store results
          # Loop through the input array
          for i in number:
              a.append(i)  # Append each number as is (linear transformation)
          return a  # Return the result list

      # Generate a sequence of numbers ranging from -20 to 20 with a step of 0.2
      randomNumber = np.arange(-20.0, 20.0, 0.2)

      # Apply the Linear Function to the numbers
      LinearLine = LinearFunction(randomNumber)

      # Plot the linear function
      plt.plot(randomNumber, LinearLine)

      # Get the current axis for potential customization (optional here)
      ax = plt.gca()

      # Display the plot
      plt.show()
```



---

2. Sigmoid Activation Function

Formula:

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

The sigmoid function maps any input to a value between 0 and 1. It's used for binary classification tasks where the output needs to represent a probability.

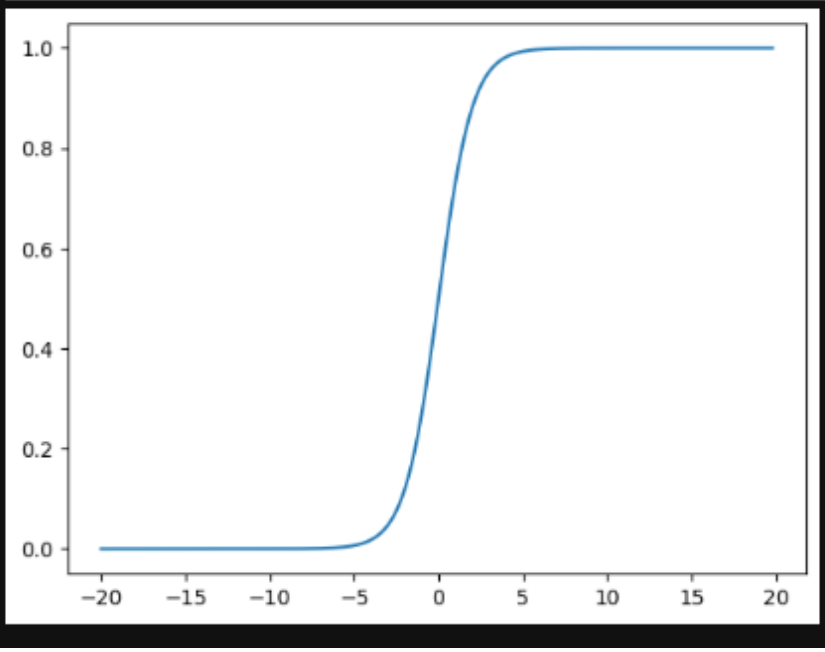
Key Features:

- **Saturating**: At extreme values, the gradient approaches 0, which can lead to vanishing gradients during backpropagation (vanishing gradient problem).
- **Smooth Output**: Output values between 0 and 1, making it useful for probabilities.
- **Not Zero-Centered**: Outputs can only be positive, making optimization harder.

Example:

Sigmoid Activation Function

[17]:
```python
import numpy as np
import matplotlib.pyplot as plt
import math

# Define the Sigmoid Function
def SigmoidFunction(number):
    sigmoidDataStore = []  # Initialize an empty list to store results
    # Loop through the input array
    for i in number:
        # Calculate sigmoid: 1 / (1 + e^(-i))
        each_result = 1 / (1 + math.exp(-i))
        sigmoidDataStore.append(each_result)  # Store the result
    return sigmoidDataStore  # Return the list of sigmoid values

# Generate a sequence of numbers ranging from -20 to 20 with a step of 0.2
randomNumber = np.arange(-20.0, 20.0, 0.2)

# Apply the Sigmoid Function to the numbers
SigmoidLine = SigmoidFunction(randomNumber)

# Plot the sigmoid function
plt.plot(randomNumber, SigmoidLine)

# Get the current axis for potential customization (optional here)
ax = plt.gca()

# Display the plot
plt.show()
```



---

3. ReLU (Rectified Linear Unit) Activation Function

Formula:

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

ReLU is one of the most commonly used activation functions in hidden layers. It outputs 0 for negative inputs and the input itself for positive values. This property helps ReLU introduce non-linearity and accelerate the convergence of training in deep networks.

Key Features:

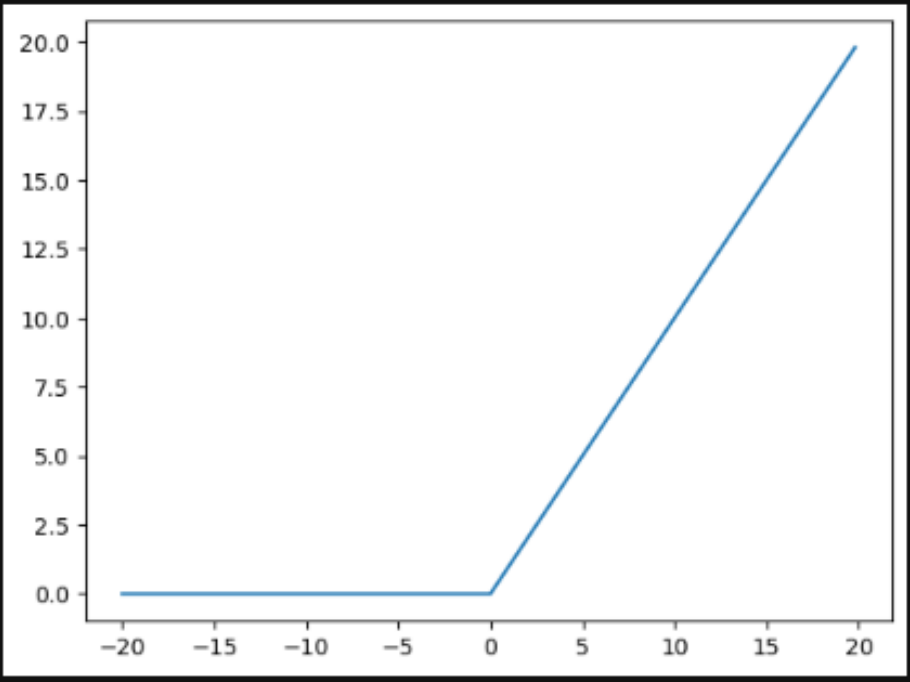
- **Non-Saturating**: The gradient is constant (1) for positive values, avoiding vanishing gradients.

- **Sparse Activation**: Outputs 0 for negative inputs, creating sparse activations, which can make the model more efficient.
- **Dying ReLU Problem**: Neurons can get "stuck" if they only output 0s, especially when the weights are poorly initialized.

Example:

ReLU (Rectified Linear Unit)

[18]:
```python
import numpy as np
import matplotlib.pyplot as plt

# Define the ReLU Function
def ReluFunction(number):
    numberStore = []  # Initialize an empty list to store results
    # Loop through the input array
    for i in number:
        initialvalue = 0  # Default value is 0
        # Apply ReLU: if the number is positive, retain it; otherwise, use 0
        if i > 0:
            initialvalue = i
        numberStore.append(initialvalue)  # Store the result
    return numberStore  # Return the list of ReLU values

# Generate a sequence of numbers ranging from -20 to 20 with a step of 0.2
randomNumber = np.arange(-20.0, 20.0, 0.2)

# Apply the ReLU Function to the numbers
ReluLine = ReluFunction(randomNumber)

# Plot the ReLU function
plt.plot(randomNumber, ReluLine)

# Get the current axis for potential customization (optional here)
ax = plt.gca()

# Display the plot
plt.show()
```



---

4. Leaky ReLU Activation Function

Formula:

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

Where α is a small constant (e.g., 0.01).

Leaky ReLU is a modification of ReLU that allows a small, non-zero slope for negative inputs. This helps address the "dying ReLU" problem, where neurons can become inactive.

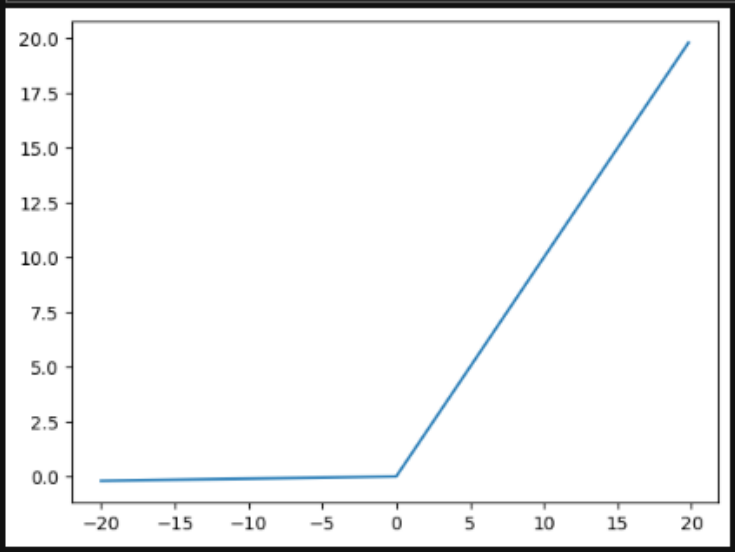
Key Features:

- **Non-Saturating**: Similar to ReLU for positive values.
- **Small Negative Slope**: Allows a small negative gradient when the input is less than 0, preventing neurons from "dying".
- **Efficiency**: Introduces more active neurons and keeps the training stable.

Example:

Leaky ReLU

[19]:
```python
import numpy as np
import matplotlib.pyplot as plt

# Define the Leaky ReLU Function
def LeakyReluFunction(number, alpha=0.01):
    numberStore = []  # Initialize an empty list to store results
    # Loop through the input array
    for i in number:
        initialvalue = i if i > 0 else alpha * i  # Apply Leaky ReLU: If the number is negative, scale it by alpha
        numberStore.append(initialvalue)  # Store the result
    return numberStore  # Return the list of Leaky ReLU values

# Generate a sequence of numbers ranging from -20 to 20 with a step of 0.2
randomNumber = np.arange(-20.0, 20.0, 0.2)

# Apply the Leaky ReLU Function to the numbers
LeakyReluLine = LeakyReluFunction(randomNumber)

# Plot the Leaky ReLU function
plt.plot(randomNumber, LeakyReluLine)

# Get the current axis for potential customization (optional here)
ax = plt.gca()

# Display the plot
plt.show()
```



---

5. Tanh (Hyperbolic Tangent) Activation Function

Formula:

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

Tanh is similar to the sigmoid function but outputs values in the range of -1 to 1. It is zero-centered, which makes it easier to optimize, as both positive and negative outputs are allowed.

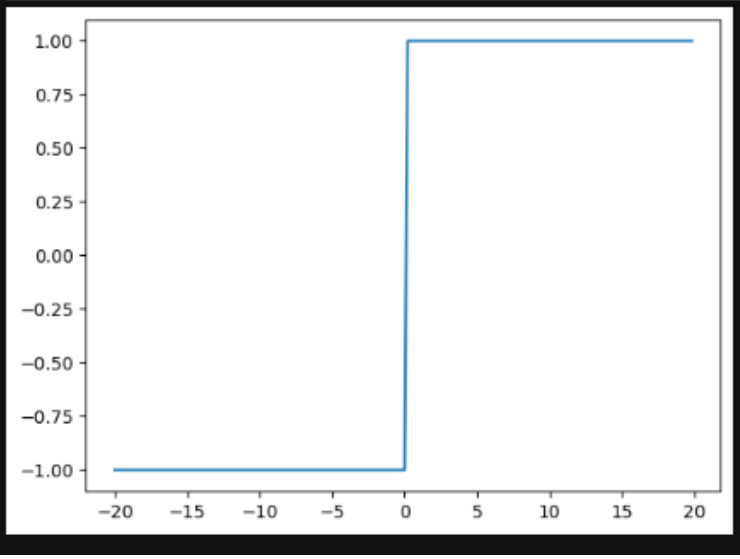
Key Features:

- **Saturating**: Like the sigmoid, it suffers from the vanishing gradient problem for very large or small values.
- **Zero-Centered**: Helps with optimization since the outputs can be both positive and negative.
- **Useful in RNNs**: Often used in recurrent neural networks to maintain state over time.

Example:

Tanh (Hyperbolic Tangent)

[20]:
```python
import numpy as np
import matplotlib.pyplot as plt

# Define the Tanh Function (discrete approximation)
def TanhFunction(number):
    tanhDataStore = []  # Initialize an empty list to store results
    # Loop through the input array
    for i in number:
        initialvalue = 0  # Default value is 0
        # Check the range of the number and assign corresponding tanh value
        if i < 0:  # For negative numbers
            initialvalue = -1
            tanhDataStore.append(initialvalue)
        elif i > 0:  # For positive numbers
            initialvalue = 1
            tanhDataStore.append(initialvalue)
    return tanhDataStore  # Return the list of tanh values

# Generate a sequence of numbers ranging from -20 to 20 with a step of 0.2
randomNumber = np.arange(-20.0, 20.0, 0.2)

# Apply the Tanh Function to the numbers
TanhLine = TanhFunction(randomNumber)

# Plot the tanh function
plt.plot(randomNumber, TanhLine)

# Get the current axis for potential customization (optional here)
ax = plt.gca()

# Display the plot
plt.show()
```



---

6. Softmax Activation Function

Formula:

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Softmax is used primarily in the output layer of a neural network for multi-class classification problems. It converts logits (raw network outputs) into probabilities, ensuring that the sum of all outputs is 1.

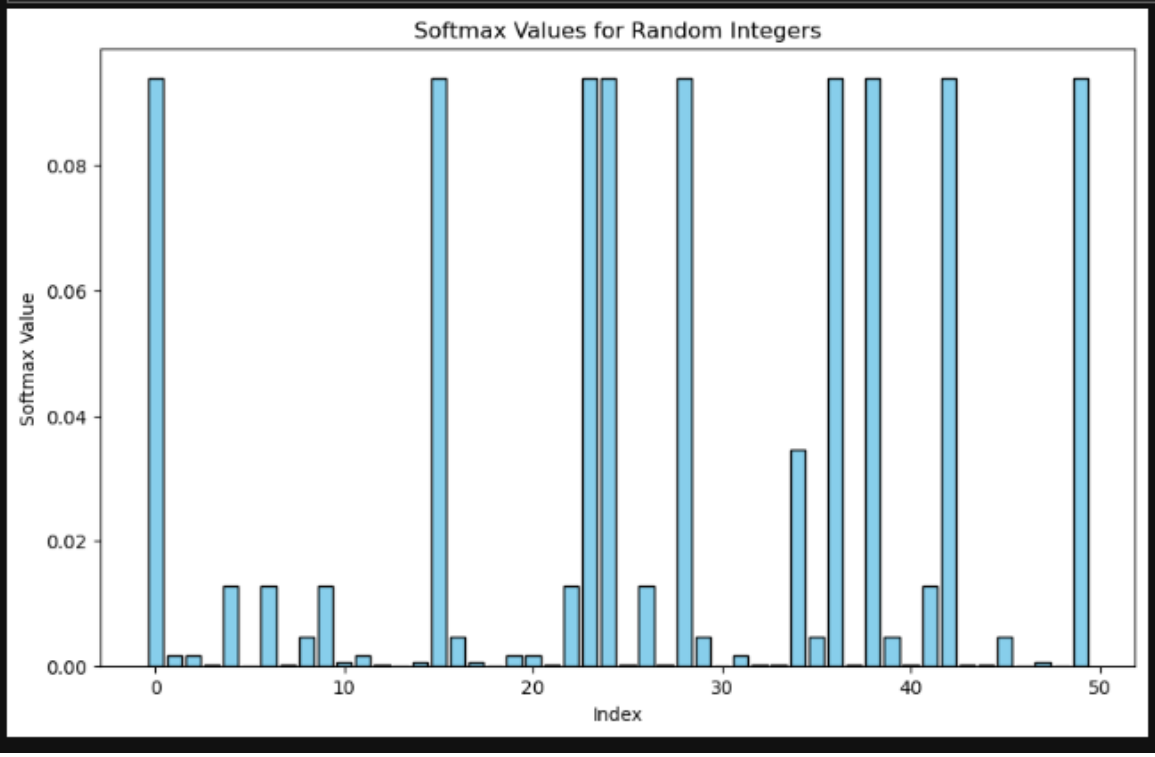
Key Features:

- **Probability Distribution**: The output values represent probabilities, useful for classification tasks.
- **Normalization**: Ensures that all output values sum to 1.
- **Exponentially Weighted**: Emphasizes larger values in the input vector and suppresses smaller values.

Example:

Softmax Activation Function

```
[22]: import numpy as np
      import matplotlib.pyplot as plt

      # Generate random numbers
      arr_number = np.random.randint(1, 10, 50)  # Array of random integers

      # Calculate softmax values
      soft_maxNum = np.exp(arr_number) / np.sum(np.exp(arr_number))  # Softmax formula

      # Plot the softmax values
      plt.figure(figsize=(10, 6))  # Set figure size
      plt.bar(range(len(arr_number)), soft_maxNum, color='skyblue', edgecolor='black')  # Bar chart
      plt.xlabel('Index')
      plt.ylabel('Softmax Value')
      plt.title('Softmax Values for Random Integers')
      plt.show()
```



By understanding the behavior of activation functions and their impact on training, deep learning models can be designed more effectively, ensuring stable learning and better performance across tasks like image recognition, sequence prediction, and more.