DEEP LEARNING

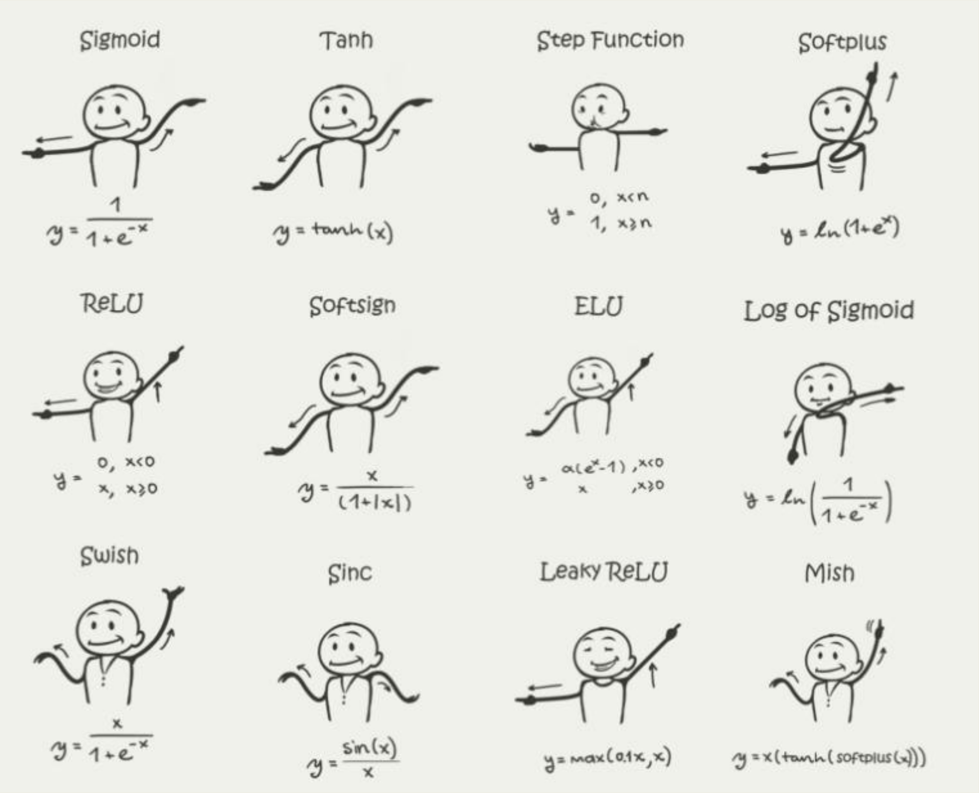
Lecture 5

Activation Functions

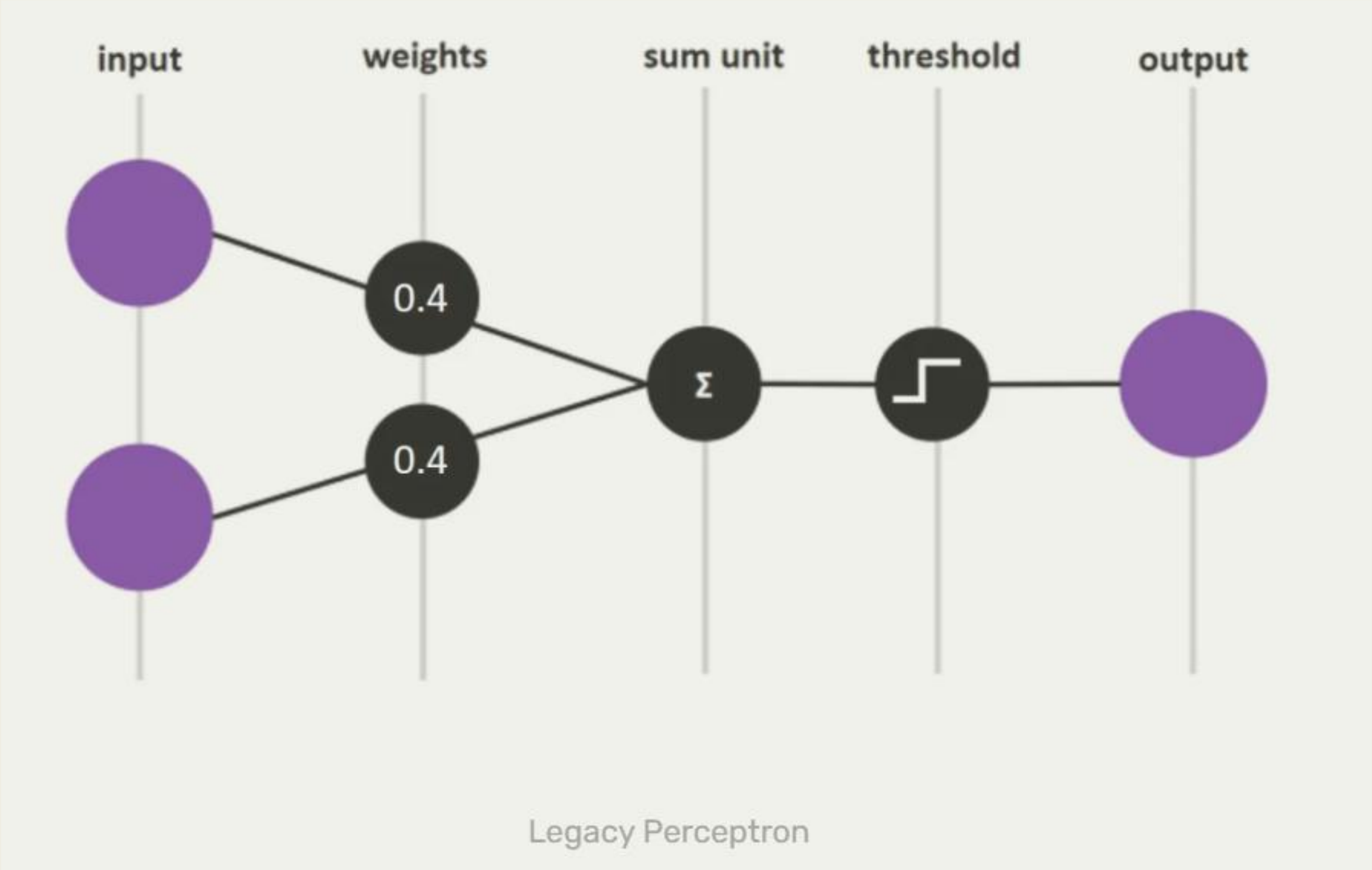
Instructor: Zafar Iqbal

Agenda

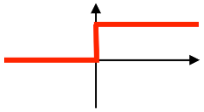
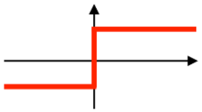
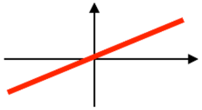
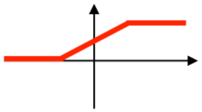
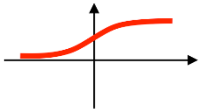
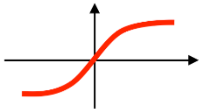
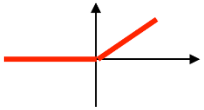
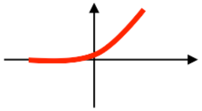
- What is Activation Function
- Sigmoid
- TanH
- ReLU
- Leaky ReLU
- Softmax



Activation Functions

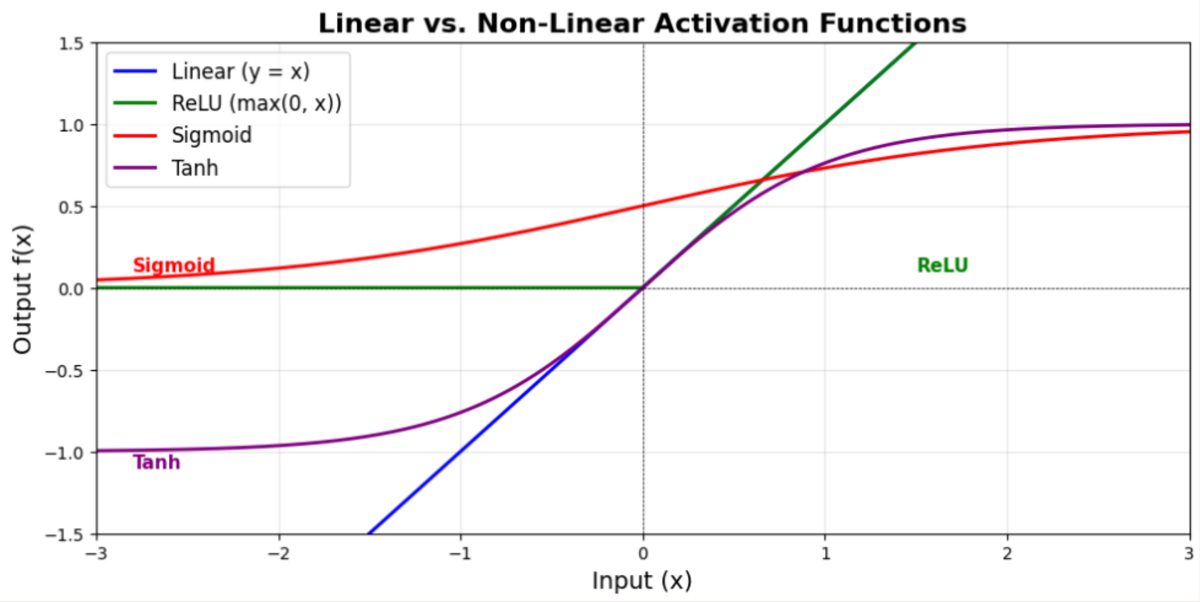


Activation Functions

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	Multi-layer Neural Networks	
Rectifier, ReLU (Rectified Linear Unit)	$\phi(z) = max(0, z)$	Multi-layer Neural Networks	
Rectifier, softplus	$\phi(z) = \ln(1 + e^{z})$	Multi-layer Neural Networks	

Activation Functions Overview

- Activation functions introduce *non-linearity* into neural networks, enabling them to learn and represent complex real-world patterns.
- Transform linear input combinations into non-linear outputs.
- Enable the network to approximate complex functions
- Help neurons decide when to activate - mimicking biological neurons.
- **Common types:** Sigmoid, Tanh, ReLU, Leaky ReLU, Softmax.

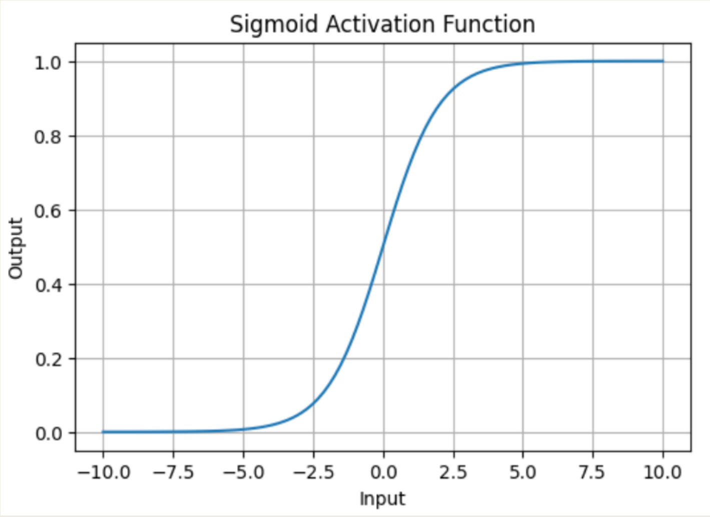


Sigmoid Activation Function

- The Sigmoid function has an *S-shaped* (logistic) curve.
- It introduces smooth non-linearity, enabling neural networks to model complex patterns beyond linear relationships.
- Mathematical Definition

$$A = \frac{1}{1 + e^{-x}}$$

- Produces continuous outputs between 0 and 1.
- Ideal for binary classification problems (e.g., yes/no, 0/1).



Sigmoid Activation Function

- Example 1: Calculate sigmoid for $x = 2$
- Step 1: Plug into formula
- $\sigma(2) = \frac{1}{1+e^{-2}}$
- Step 2: Calculate e^{-2}
- $e^{-2} \approx 0.1353$
- Step 3: Add 1
- $1 + 0.1353 = 1.1353$
- Step 4: Take reciprocal
- $\sigma(2) = \frac{1}{1.1353} \approx 0.8808$
- Final Answer:
- $\sigma(2) \approx 0.8808$

Sigmoid Activation Function

Input x	Full calculation step-by-step	Final Result σ(x)
x = 0	σ(0) = 1 / (1 + e^0) = 1 / (1 + 1) = 1 / 2	**0.5** (exactly)
x = 1	e^{-1} ≈ 0.367879 1 + 0.367879 = 1.367879 1 ÷ 1.367879 ≈ 0.7317	**≈ 0.7311**
x = 2	e^{-2} ≈ 0.135335 1 + 0.135335 = 1.135335 1 ÷ 1.135335 ≈ 0.8808	**≈ 0.8808**
x = 3	?	**≈ 0.9526**
x = 5	?	**≈ 0.9933**
x = 10	?	**≈ 0.99995**

Sigmoid Activation Function

```
import math
# Sigmoid function
def sigmoid(x):
    return 1 / (1 + math.exp(-x))
# Example "feature values" representing some measure of cat-likeness
# Let's say values > 0 mean likely cat, < 0 likely not cat
features = [2.0, -1.5, 0.0, 3.5, -2.2]
# Classify each example
for i, x in enumerate(features):
    prob = sigmoid(x)          # Compute probability of being a cat
    prediction = 1 if prob > 0.5 else 0  # Threshold at 0.5
    label = "Cat" if prediction == 1 else "Not Cat"
    print(f"Example {i+1}: feature={x}, probability={prob:.4f}, prediction={label}")
```

Sigmoid Activation Function

Example 1:

feature=2.0, probability=0.8808, prediction=**Cat**

Example 2:

feature=-1.5, probability=0.1824, prediction=**Not Cat**

Example 3:

feature=0.0, probability=0.5000, prediction=**Not Cat**

Example 4:

feature=3.5, probability=0.9707, prediction=**Cat**

Example 5:

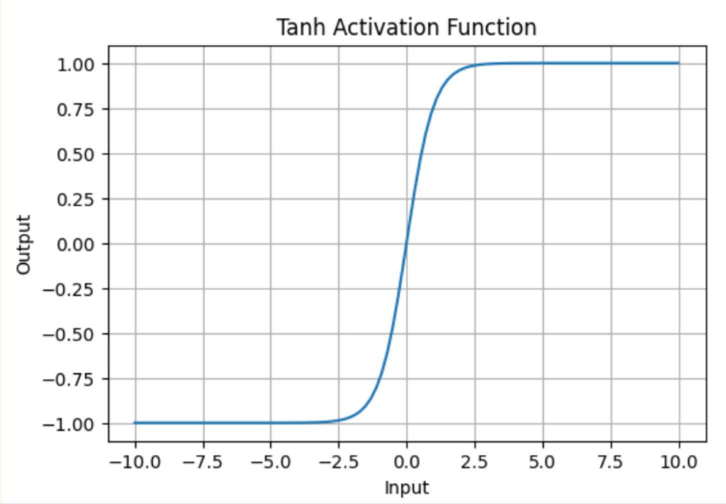
feature=-2.2, probability=0.0998, prediction=**Not Cat**

Tanh (Hyperbolic Tangent) Activation Function

- Tanh is a shifted and scaled version of the Sigmoid function.
- It stretches across the y-axis, making outputs zero-centered.
- Adds non-linearity, enabling networks to learn complex data relationships.
- Mathematical Definition

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

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



- Output Range: −1 to +1
- Zero-centered: Helps faster convergence during training
- Non-linear: Captures intricate data patterns
- Common Usage: Frequently used in hidden layers of neural networks

Tanh Activation Function

x	Step-by-step calculation using the formula	Final tanh(x)
0	$e^{-2(0)} = e^{0} = 1$ $1 + 1 = 2$ $2 / 2 = 1$ $1 - 1 = \mathbf{0}$	**0.0000**
1	$e^{-2(1)} = e^{-2} \approx 0.135335$ $1 + 0.135335 = 1.135335$ $2 \div 1.135335 \approx 1.7616$ $1.7616 - 1 = \mathbf{0.7616}$	**≈ 0.7616**
2	$e^{-4} \approx 0.0183156$ $1 + 0.0183156 = 1.0183156$ $2 \div 1.0183156 \approx 1.9640$ $1.9640 - 1 = \mathbf{0.9640}$	**≈ 0.9640**
3	?	**≈ 0.9951**
5	?	**≈ 0.9999**
−1	?	**≈ -0.7616**
−2	?	**≈ -0.9640**

Tanh Activation Function

```
import numpy as np
import matplotlib.pyplot as plt
# ---------------------------------
# Simple "Cat vs Not Cat" example using tanh
# ---------------------------------
# Fake but realistic scores from a neural network
# Positive score → more likely "Cat"
# Negative score → more likely "Not Cat"

# Step 1: Raw scores from the last layer (before activation)
scores = np.array([-5, -3, -1, -0.5, 0, 0.5, 1, 2, 4, 6])

# Step 2: Apply tanh using the exact formula you love
def tanh(x):
    return 2 / (1 + np.exp(-2*x)) - 1
probabilities = tanh(scores)     # Output between -1 and +1
# Step 3: Convert to probability of "Cat" (common trick)
# We map: -1 → 0% cat,  0 → 50% cat,  +1 → 100% cat
cat_probability = (probabilities + 1) / 2
# Step 4: Final prediction
prediction = cat_probability >= 0.5  # T = Cat, F = Not Cat
```

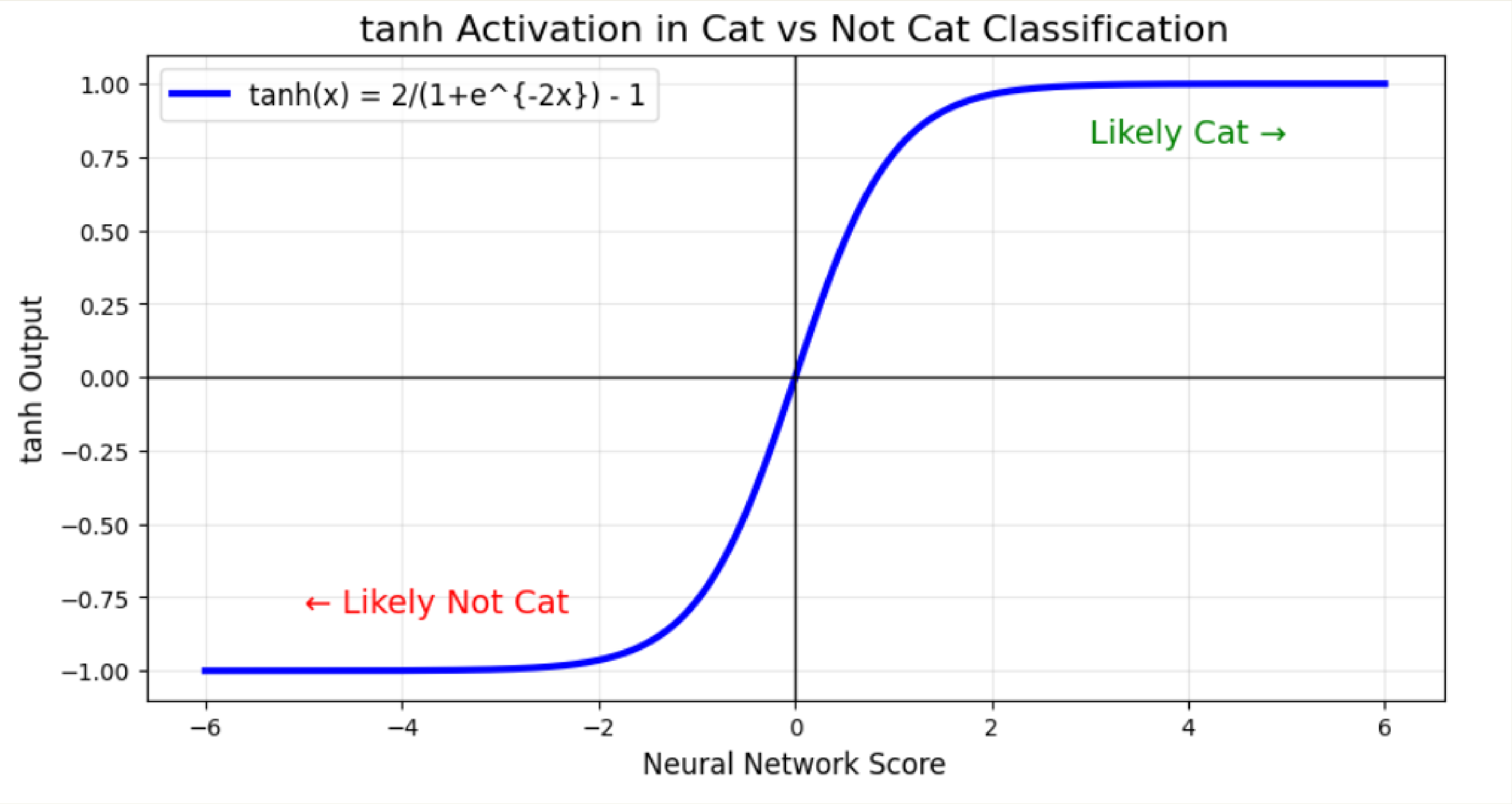
Tanh Activation Function

```
print("Score -->  tanh(score) -->  Cat Probability --> Final
Prediction")
print("-" * 70)
for s, t, p, pred in zip(scores, probabilities, cat_probability,
prediction):
    animal = "Cat" if pred else "Not Cat"
    print(f"{s:4.1f} -->  {t:6.3f} -->   {p:6.1%}-->{animal}")
plt.figure(figsize=(10, 5))
x = np.linspace(-6, 6, 500)
y = tanh(x)
plt.plot(x, y, 'b-', linewidth=3, label='tanh(x) = 2/(1+e^{-2x}) -
1')
plt.title('tanh Activation in Cat vs Not Cat Classification',
fontsize=16)
plt.xlabel('Neural Network Score', fontsize=12)
plt.ylabel('tanh Output', fontsize=12)
plt.axhline(0, color='black', linewidth=1)
plt.axvline(0, color='black', linewidth=1)
plt.grid(True, alpha=0.3)
plt.legend(fontsize=12)
plt.text(3, 0.8, 'Likely Cat →', fontsize=14, color='green')
plt.text(-5, -0.8, '← Likely Not Cat', fontsize=14, color='red')
plt.show()
```

Tanh Activation Function

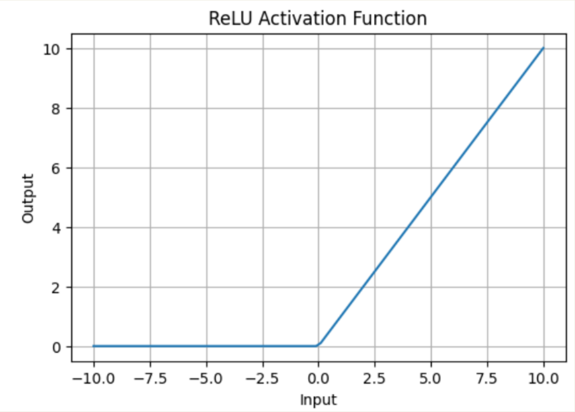
```
Score --> tanh(score) --> Cat Probability --> Final Pred
---------------------------------------------------------
5.0  --> -1.000       --> 0.0%            --> Not Cat
-3.0  --> -0.995      --> 0.2%            --> Not Cat
-1.0  --> -0.762      --> 11.9%           --> Not Cat
-0.5  --> -0.462      --> 26.9%           --> Not Cat
0.0   -->  0.000      --> 50.0%           --> Cat
0.5   -->  0.462      --> 73.1%           --> Cat
1.0   -->  0.762      --> 88.1%           --> Cat
2.0   -->  0.964      --> 98.2%           --> Cat
4.0   -->  0.999      --> 100.0%          --> Cat
6.0   -->  1.000      --> 100.0%          --> Cat
```

Tanh Activation Function



ReLU (Rectified Linear Unit) Activation Function

- The most widely used activation in deep learning networks.
- Introduces non-linearity while being computationally simple.
- **Mathematical Definition**
  - $A(x) = \max(0, x)$
  - If $x > 0$, output is x.
  - If $x \leq 0$, output is 0.



- **Output Range:** $[0, \infty)$
- **Non-linear:** Enables learning of complex relationships.
- **Efficient Backpropagation:** Simplifies gradient computation.
- **Advantages Over Sigmoid & Tanh**
  - Computationally Efficient: Uses simple thresholding, no exponentials.
  - Sparse Activation: Only a few neurons activate at a time → faster and more efficient training.
  - Reduced Vanishing Gradient: Maintains stronger gradients for positive values.

ReLU Activation Function

Input x	Apply ReLU rule step-by-step	ReLU(x) result
x = 5	3 > 0 → return x itself	**3**
x = 10	10 > 0 → return 10	**10**
x = 0.7	0.7 > 0 → return 0.7	**0.7**
x = 0	x = 0 → rule says ≥ 0, so return 0	**0**
x = -1	-1 < 0 → return 0	**0**
x = -4.5	-4.5 < 0 → return 0	**0**
x = -100	-100 < 0 → return 0	**0**

In code/backpropagation:
→ if input > 0 → gradient = 1
→ if input ≤ 0 → gradient = 0 (neuron is dead)

ReLU Activation Function

```python
import numpy as np

# Raw scores from your neural network
scores = np.array([-4.2, -1.5, -0.3, 0.0, 0.8, 2.1, 5.7])

# Apply ReLU activation (what happens inside hidden layers)
relu_output = np.maximum(0, scores)   # this is ReLU!

print("Score → After ReLU → Meaning")
print("────────────────────────────")
for score, out in zip(scores, relu_output):
    if out == 0:
        meaning = "Neuron completely silent (dead for this image)"
    else:
        meaning = f"Neuron active → sends {out:.1f} to next layer"
    print(f"{score:5.1f} →    {out:4.1f}    → {meaning}")
```

ReLU Activation Function

```
Score → After ReLU → Meaning
──────────────────────────────
 -4.2 →      0.0      → Neuron completely silent (dead for this image)
 -1.5 →      0.0      → Neuron completely silent (dead for this image)
 -0.3 →      0.0      → Neuron completely silent (dead for this image)
  0.0 →      0.0      → Neuron completely silent (dead for this image)
  0.8 →      0.8      → Neuron active → sends 0.8 to next layer
  2.1 →      2.1      → Neuron active → sends 2.1 to next layer
  5.7 →      5.7      → Neuron active → sends 5.7 to next layer
```

Activation Functions Behavior at a Glance

Activation	Is it zero-centered?	Average value
Sigmoid	No	≈ 0.5
Tanh	Yes	≈ 0
ReLU	No	> 0

Some Important questions

What is the derivative (gradient) of ReLU?

- **Answer:**
- ReLU'(x) = { 1 if x > 0

  0 if x < 0

  0 or 1 or undefined at exactly x = 0 (in practice we use 0)
- Or in short form:
- Gradient = 1 when input is positive
- Gradient = 0 when input is negative or zero

Some Important questions

Why did ReLU almost completely replace sigmoid and tanh in hidden layers after 2012?

- **Answer:** ReLU replaced sigmoid/tanh because:
  - No vanishing gradient for x > 0 (gradient = 1, not → 0) → trains much faster and deeper networks
  - Computationally cheaper (just max(0,x), no expensive exp())
  - Induces sparsity (negative inputs → 0) → better generalization
  - Works perfectly with He/Kaiming initialization

Some Important questions

If 40% of neurons output 0 forever after training, what problem is this and how can you fix it?

- **Answer:** This is the dying ReLU problem.
  - *Use Leaky ReLU or Parametric ReLU (P-ReLU)*
  - *Use He/Kaiming initialization*
  - *Lower the learning rate*
  - *Use ReLU6 or ELU*
  - *Add Batch Normalization before ReLU*

Dying ReLU Problem

- Imagine a smart robot that identifies animals.
  - When it sees a **dog**, it gives positive scores.
  - When it sees a **cat**, it gives negative scores.
- If the robot sees too many cats first:
  - ReLU makes all negative values **0**.
  - Robot's "dog detector neuron" becomes **silent forever**.
  - Later, even if you show a dog, it no longer reacts.
- This is the **Dying ReLU problem**.
- **Leaky ReLU Fixes It**

With Leaky ReLU:

- Even negative values send a *small* signal.
- The dog-detector neuron is still alive.
- It can recover and learn dog features later.

Leaky ReLU (Leaky Rectified Linear Unit)

Overview

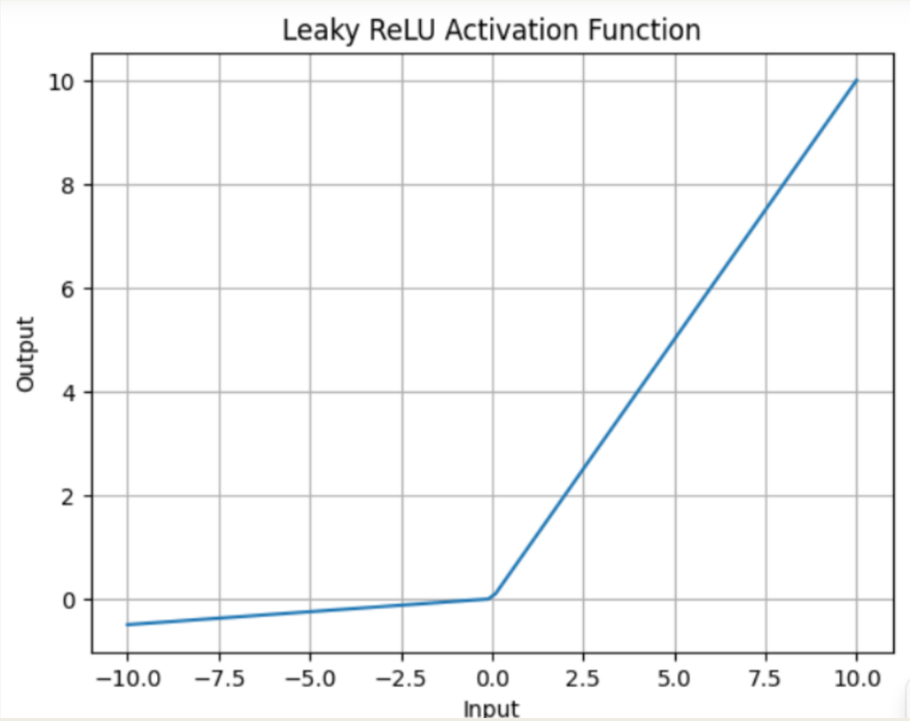
- A variant of ReLU that allows a **small negative slope** instead of outputting zero for negative inputs.
- Helps prevent the **"dying ReLU"** problem, where neurons stop learning due to zero gradients.
- **Mathematical Definition**
- $f(x) = \begin{cases} x, & x > 0 \\ \alpha x, & x \leq 0 \end{cases}$
- Where α is a small constant (e.g., 0.01).

Key Characteristics

- **Output Range:** $(-\infty, \infty)$
- **Non-linearity:** Enables complex mapping and learning.
- **Improved Gradient Flow:** Maintains non-zero gradient for negative inputs.

Leaky ReLU (Leaky Rectified Linear Unit)

- **Advantages**
- Prevents neurons from becoming inactive (unlike standard ReLU).
- Helps achieve **better convergence** and **stable training.**
- Useful in **deep networks** where vanishing gradients can occur.



Leaky ReLU

```
import numpy as np

scores = np.array([-5.0, -2.0, -0.5, 0.0, 1.2, 3.8])

relu = np.maximum(0, scores)

leaky_relu = np.where(scores > 0, scores, 0.01 * scores)

print("Score | ReLU | LeakyReLU | Meaning")
print("-------------------------------------")
for s, r, l in zip(scores, relu, leaky_relu):
    if r == 0 and l != 0:
        meaning = "ReLU kills neuron, Leaky keeps it alive"
    elif r == 0:
        meaning = "Dead neuron"
    else:
        meaning = "Neuron active"
    print(f"{s:5.1f} | {r:4.1f} | {l:9.3f} | {meaning}")
```

Leaky ReLU

Score	ReLU	LeakyReLU	Meaning
-5.0	0.0	-0.050	ReLU kills neuron, Leaky keeps it alive
-2.0	0.0	-0.020	ReLU kills neuron, Leaky keeps it alive
-0.5	0.0	-0.005	ReLU kills neuron, Leaky keeps it alive
0.0	0.0	0.000	Dead neuron
1.2	1.2	1.200	Neuron active
3.8	3.8	3.800	Neuron active

Softmax Activation Function

- Designed for **multi-class classification problems.**
- Converts the raw network outputs (logits) into **probabilities.**
- Ensures that the **sum of all output probabilities equals 1**.

Mathematical Definition

- $\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$
- Where:
- z_i =score (logit) for class *i*
- K= total number of classes
- e^{z_i} =exponentiation (makes all values positive)

Key Characteristics

- **Range:** (0, 1)
- **Non-linear:** Allows learning of complex class relationships.
- **Probabilistic Interpretation:** Each output represents the **likelihood** of belonging to a specific class.

Softmax Activation Function

Applications

- Commonly used in the **output layer** of classification networks (e.g., image or text classification).
- Enables comparison between multiple classes effectively.

Insight

- Softmax converts neural network outputs into a **probability distribution**, helping models make **interpretable decisions** for multi-class tasks.

Three-class classification (Cat, Dog, Horse)

- Suppose your neural network outputs these **raw scores (logits):**
  - Cat = **2.0**
  - Dog = **1.0**
  - Horse = **0.1**
- Let's apply **softmax step-by-step.**
- **STEP 1 — Take exponent of each score**
  - $e^{2.0} = 7.389$
  - $e^{1.0} = 2.718$
  - $e^{0.1} = 1.105$
- **STEP 2 — Sum all exponent values**
  - $7.389 + 2.718 + 1.105 = 11.212$

Three-class classification (Cat, Dog, Horse)

- STEP 3 — Divide each exponent by the sum
- Cat probability
  - $P(Cat) = \frac{7.389}{11.212} = 0.659 \approx 65.9\%$
- Dog probability
  - $P(Dog) = \frac{2.718}{11.212} = 0.242 \approx 24.2\%$
- Horse probability
  - $P(Horse) = \frac{1.105}{11.212} = 0.098 \approx 9.8\%$

Class	Logit	Softmax Probability
Cat	2.0	**0.659**
Dog	1.0	**0.242**
Horse	0.1	**0.098**

Example (3-class classifier)

- logits = [2.0, 1.0, 0.1]

1. Let's compute Softmax:

- Exponentiate each:
- $[e^2, e^1, e^{0.1}] \approx [7.39, 2.71, 1.10]$

2. Sum them:

- $7.39 + 2.71 + 1.10 = 11.20$

3. Divide each by total:

- $\text{Softmax} = [0.66, \ , 0.24 \ 0.10]$

Final probabilities:

- Class 1 → **66%**
- Class 2 → 24%
- Class 3 → 10%

- The model chooses **Class 1.**

Softmax Example

```python
import numpy as np
# Logits (raw scores from the model)
scores = np.array([2.0, 1.0, 0.1])   # [Cat, Dog, Horse]
# Softmax calculation
exp_scores = np.exp(scores)
probabilities = exp_scores / np.sum(exp_scores)

# Print results
classes = ["Cat", "Dog", "Horse"]
print("Class   Logit   Softmax Probability")
print("-------------------------------------")
for c, s, p in zip(classes, scores, probabilities):
    print(f"{c:6} {s:5.1f}          {p:.3f}")
```

```
Class    Logit    Softmax Probability
-------------------------------------
Cat        2.0          0.659
Dog        1.0          0.242
Horse      0.1          0.099
```