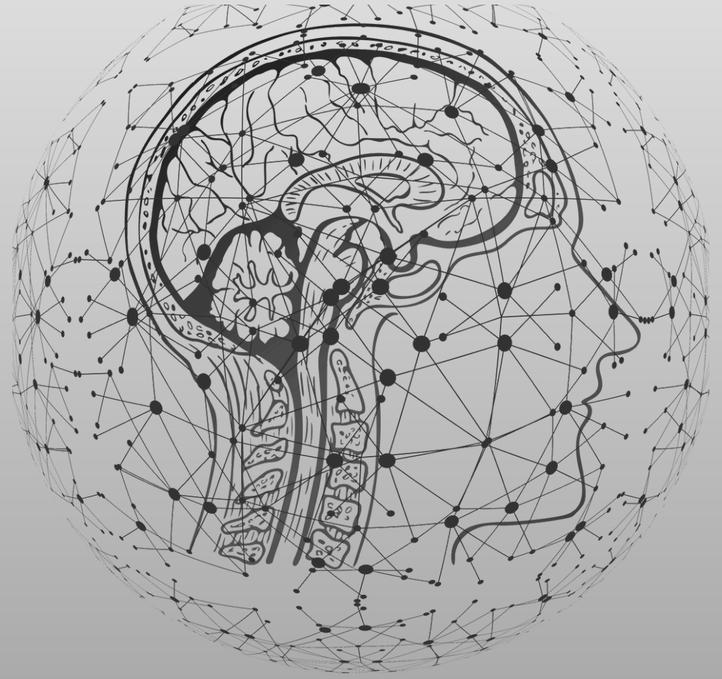
2.HAFTA

YAPAY SİNİR AĞLARI



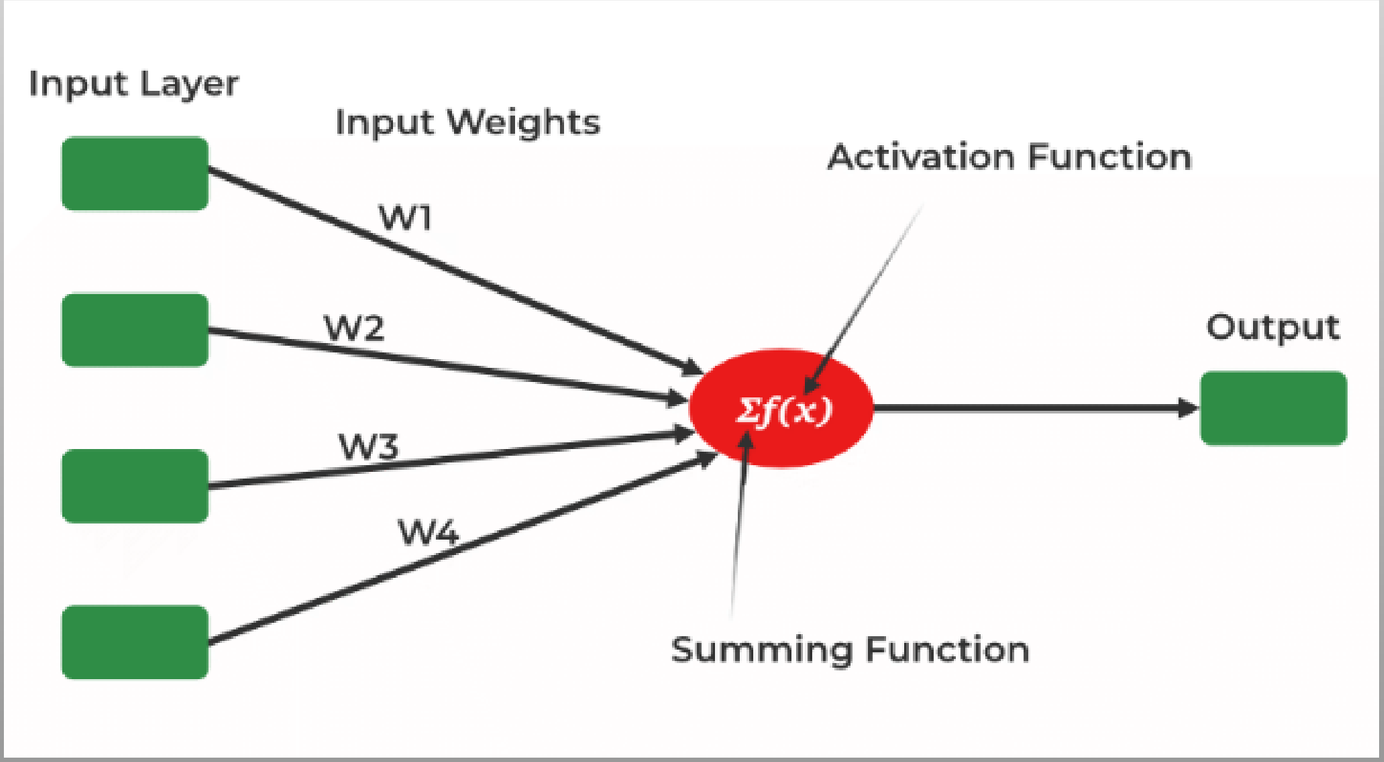
DR. ÖĞR. ÜYESİ ALPER TALHA KARADENİZ

SAMSUN ÜNİVERSİTESİ

PERCEPTRON VE AKTIVASYON FONKSIYONLARI:

1. Tek Katmanlı Perceptron (Single-Layer Perceptron)

- **Temel yapay sinir ağı birimidir.**
- **Girdileri alır, ağırlıklarla çarpar, bias ekler ve aktivasyon fonksiyonundan geçirerek çıktı üretir.**



Matematiksel Formül:

$$\text{Çıktı} = f(w_1x_1 + w_2x_2 + \ldots + w_nx_n + b)$$

- **w:** Ağırlıklar
- **x:** Girdiler
- **b:** Bias (Eşik değeri)
- **f:** Aktivasyon fonksiyonu

Python

```
import numpy as np

def perceptron(X, weights, bias, activation):
    z = np.dot(X, weights) + bias
    return activation(z)
```

2. Ağırlıklar, Bias ve Öğrenme Oranı

Ağırlıklar (Weights):

- Her bir girdinin ne kadar önemli olduğunu belirler.
- Başlangıçta rastgele küçük değerlerle başlar (np.random.randn).

Bias (Bias):

- Modelin esnekliğini artırır.
- Aktivasyon eşiğini ayarlar.

Öğrenme Oranı (Learning Rate – α):

- Ağırlık güncelleme adım büyüklüğünü kontrol eder.
- Çok büyükse: Salınım yapar, öğrenemez.
- Çok küçükse: Yavaş öğrenir.

Örnek Ağırlık Güncelleme:

Python

```
def update_weights(X, y_true, y_pred, weights, bias, lr=0.01):
    error = y_true - y_pred
    weights += lr * error * X  # Ağırlık güncelleme
    bias += lr * error         # Bias güncelleme
    return weights, bias
```

Örnek : Ağırlık Güncelleme

Bir yapay sinir ağı nöronuna verilen bilgiler:

- Girişler:

  $x_1 = 0.5,$

  $x_2 = -0.8$
- Ağırlıklar:

  $w_1 = 0.1,$

  $w_2 = -0.4$
- Öğrenme oranı: $\eta = 0.05$
- Hata sinyali (δ) = **0.6**

Ağırlıkları güncelleyiniz.

Ağırlık güncelleme formülü:

$$w_i^{\text{yeni}} = w_i + \eta \cdot \delta \cdot x_i$$

Güncellenmiş Ağırlıklar:

$$w_1^{\text{yeni}} = 0.1 + 0.05 \cdot 0.6 \cdot 0.5 = 0.1 + 0.015 = \boxed{0.115}$$

$$w_2^{\text{yeni}} = -0.4 + 0.05 \cdot 0.6 \cdot (-0.8) = -0.4 - 0.024 = \boxed{-0.424}$$

Sonuç:

Ağırlık	Yeni Değer
w_1	0.115
w_2	-0.424

AKTIVASYON FONKSIYONLARI:

Aktivasyon fonksiyonları , bir yapay sinir ağında bir nöronun çıktısını belirleyen matematiksel bir işlemlerdir.

Nörona gelen sinyallerin (girişlerin), doğrusal olmayan bir şekilde dönüştürülmesini sağlar ve bu sayede ağ, karmaşık ilişkileri öğrenebilir.

- Nörona gelen bilgiyi işler,
- Çıktının ne olacağına karar verir,
- Ve ağı daha esnek ve güçlü hale getirir.

Aktivasyon Fonksiyonlarının Genel Özellikleri :

1. Sıfır Merkezli Çıktı Aralığı:

Bu özellik, aktivasyon fonksiyonunun çıktılarının negatif ve pozitif değerler arasında (örneğin –1 ile +1) dağılmasını ifade eder. Böyle fonksiyonlar, öğrenme sürecinde ağırlıkların hem pozitif hem negatif yöne dengeli bir şekilde güncellenmesine olanak tanır.

Ne işe yarar?

- Gradyan iniş algoritmasında simetrik çıktı, daha verimli öğrenme sağlar.
- Aksi takdirde, ağırlık güncellemeleri hep aynı yönde kayabilir ve öğrenme yavaşlar.

2. Gradyan Büyüklüğü:

Gradyan, bir fonksiyonun girişe göre ne kadar değiştiğini gösteren türev değeridir. Bir aktivasyon fonksiyonunun gradyanı ne kadar büyükse, öğrenme adımları o kadar etkili olur.

Ne işe yarar?

- Büyük gradyanlar, daha hızlı ve sağlıklı öğrenme sağlar.
- Küçük gradyanlar, özellikle derin ağlarda "vanishing gradient" (kaybolan gradyan) problemlerine yol açabilir.

3. Simetrik Dağılım:

Fonksiyon çıktılarının pozitif ve negatif yönde eşit veya dengeli dağılmasıdır. Bu denge, öğrenme sırasında ağın farklı yönlere sapmasını engeller.

Ne işe yarar?

- Ağırlık güncellemelerinin bir yönde "kıtalanmasını" (tek yönlü hareket etmesini) önler.
- Özellikle RNN gibi sıralı ağlarda öğrenmeyi daha stabil hâle getirir.

4. Diferansiyellenebilirlik:

Aktivasyon fonksiyonunun her noktasında türev alınabilir olmasıdır. Bu matematiksel özellik, öğrenme algoritmalarının çalışabilmesi için zorunludur.

Ne işe yarar?

- Geri yayılım (Backpropagation) algoritması, türev değerlerini kullanarak ağırlıkları günceller.
- Eğer fonksiyon türevlenemezse, bu süreç işlemez ve öğrenme durur.

Aktivasyon Fonksiyonları:

1. Sigmoid Fonksiyonu:

Matematiksel Formül :

$$\sigma(z) = 1 / (1 + e^{-z})$$

Özellikler :

- Çıktıyı 0 ile 1 arasına sıkıştırır
- Özellikle ikili sınıflandırma problemlerinde kullanılır

Matematiksel Özellikler :

- Türevi: $\sigma'(z) = \sigma(z)(1 - \sigma(z))$
- Limiti: $\lim_{z \to \infty} \sigma(z) = 1$, $\lim_{z \to -\infty} \sigma(z) = 0$
- İkinci türevi: $\sigma''(z) = \sigma(z)(1 - \sigma(z))(1 - 2\sigma(z))$

Örnek :

Bir yapay sinir ağı nöronuna bir giriş x= –1 geliyor. Bu girişe sigmoid fonksiyonu uygulanırsa çıktı ne olur?

Adım Adım Çözüm:

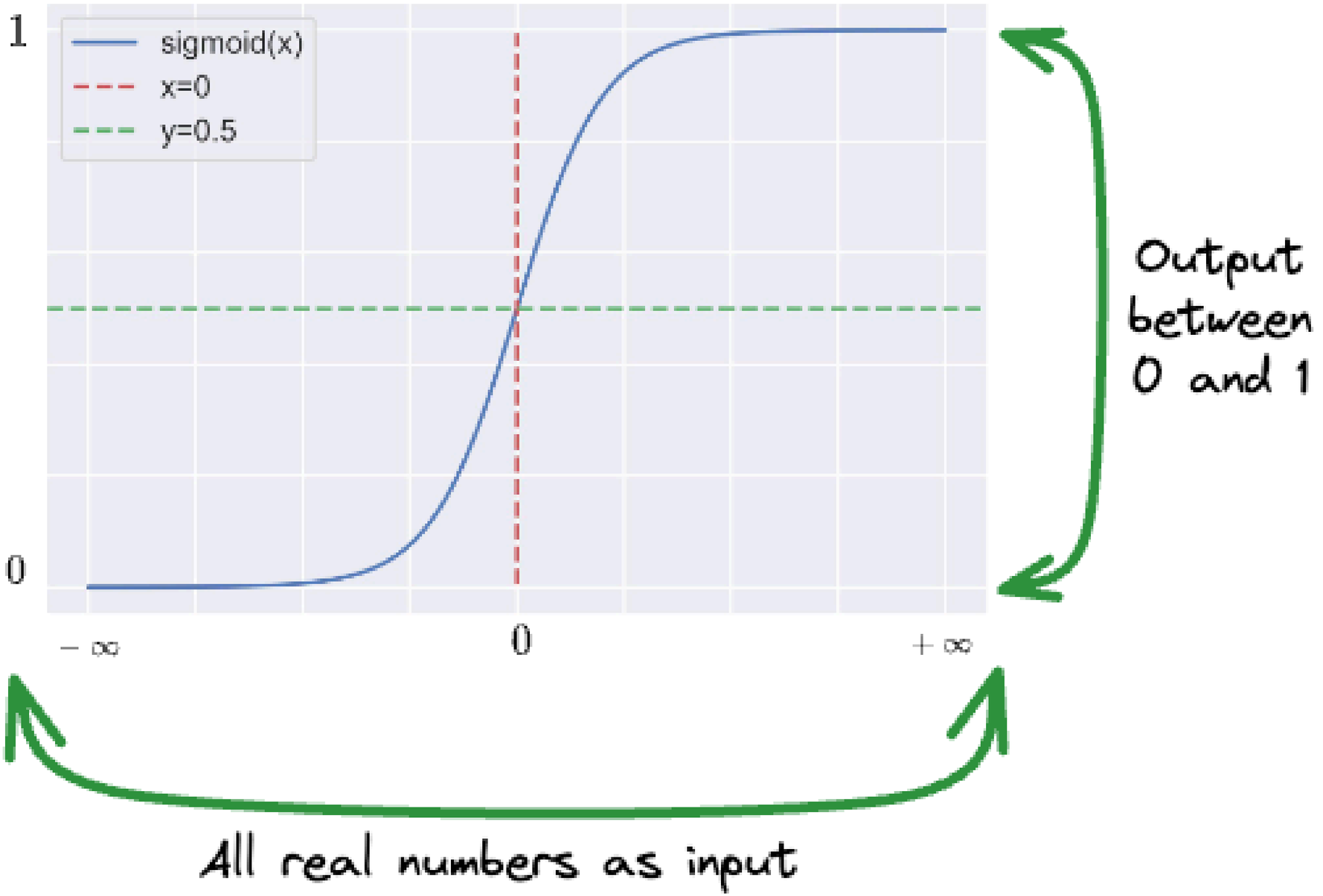
$$\sigma(-1) = \frac{1}{1+e^{1}} = \frac{1}{1+2.718} \approx \frac{1}{3.718} \approx \boxed{0.2689}$$

Sonuç:

$$\sigma(-1) \approx 0.2689$$

x değeri	Sigmoid(x)
-3	≈ 0.0474
-1	≈ 0.2689
0	0.5

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



```python
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Örnek kullanım
print("Sigmoid(0):", sigmoid(0))  # 0.5
print("Sigmoid(5):", sigmoid(5))  # 0.9933
print("Sigmoid(-2):", sigmoid(-2))  # 0.1192
```

Nöral Ağlardaki Davranışı:

- Gradyan kaybı problemi: $|\sigma'(z)| \leq 0.25$ olduğundan, zincir kuralıyla çarpılan türevler hızla küçülür
- Çıktı aralığı: $(0, 1)$ aralığı olasılık yorumuna izin verir

```python
def sigmoid(x):
    # Pozitif ve negatif girdiler için stabil hesaplama
    mask = x >= 0
    positive = np.exp(-x[mask])
    negative = np.exp(x[~mask])

    result = np.empty_like(x)
    result[mask] = 1.0 / (1.0 + positive)
    result[~mask] = negative / (1.0 + negative)
    return result
```

2. Hiperbolik Tanjant (Tanh) Fonksiyonu:

$$\tanh(x) = (e^x - e^{-x})/(e^x + e^{-x}) = 2\sigma(2x) - 1$$

Matematiksel Analiz:

1. Türev: $\tanh'(x) = 1 - \tanh^2(x) = \text{sech}^2(x)$
2. Taylor Serisi: $\tanh(x) \approx x - x^3/3 + 2x^5/15 - \_$ ($|x| < \pi/2$ için)
3. Lipschitz Sabiti: 1 (türevin mutlak değeri ≤ 1)

Avantajları:

- Sıfır merkezli (-1,1) aralığı, özellikle RNN'lerde daha iyi yakınsama sağlar
- Sigmoid'e göre daha büyük gradyanlar üretir (maksimum türev değeri 1)

Örnek:

Bir nörona giriş olarak x=1 geliyor. Bu girişe hiperbolik tanjant aktivasyon fonksiyonu uygulanırsa, nöronun çıktısı ne olur?

$x = 1$ için:

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

$$e^{1} \approx 2.718, \quad e^{-1} \approx 0.3679$$

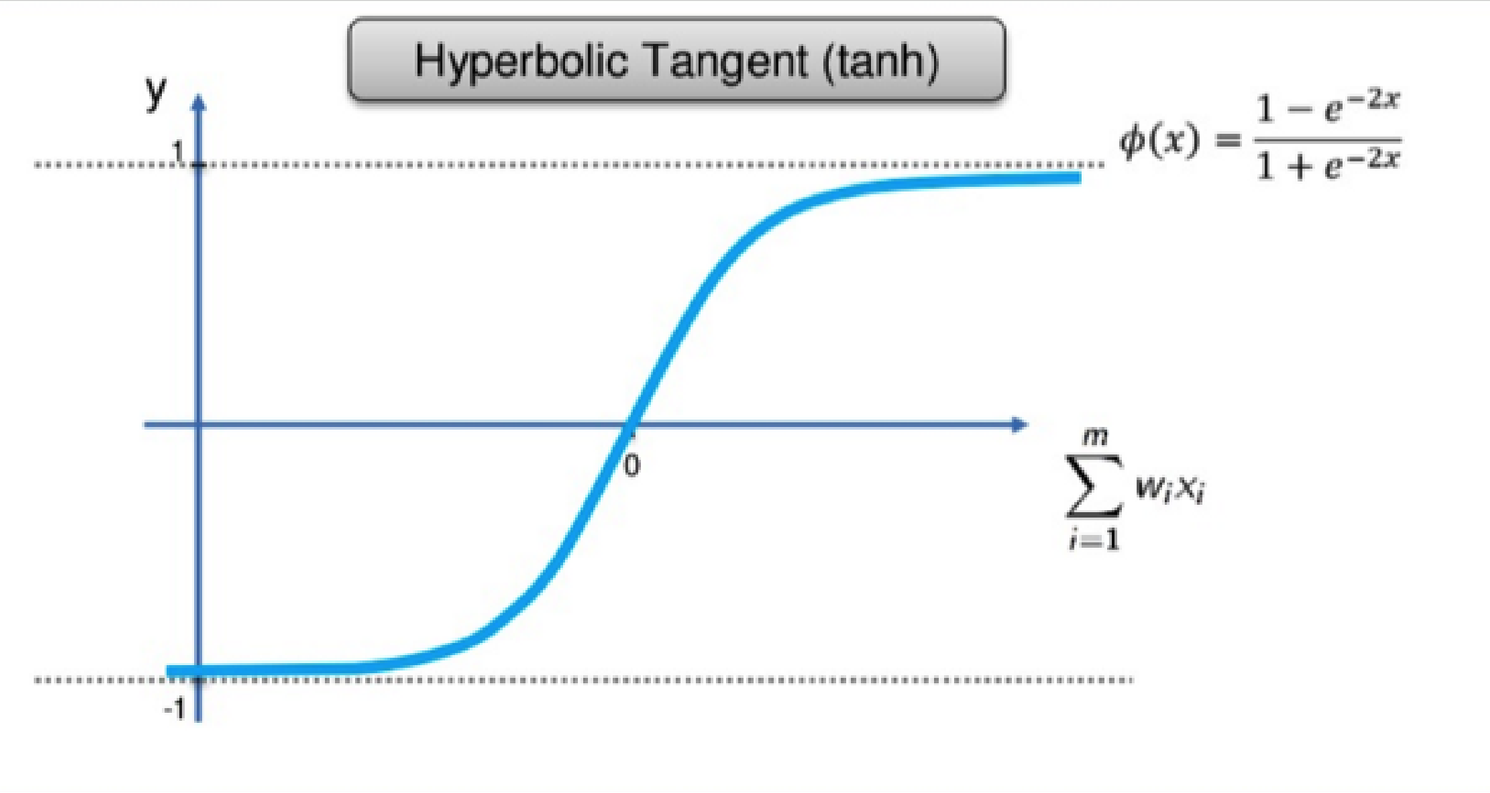
$$\tanh(1) = \frac{2.718 - 0.3679}{2.718 + 0.3679} = \frac{2.3501}{3.0859} \approx \boxed{0.7616}$$

Sonuç:

$$\tanh(1) \approx 0.7616$$

Karşılaştırmalı Değerler :

x	tanh(x)
-3	-0.995
-1	-0.7616
0	0



Nöral Ağlardaki Avantajları :

- Sıfır Merkezli Çıktı Aralığı:

Çıktıların (−1, 1) aralığında olması, özellikle tekrarlayan ağlarda (RNN) daha iyi yakınsama sağlar.

- Gradyan Büyüklüğü:

Maksimum türev değeri 1'dir (x=0'da), sigmoidin maksimum türev değeri 0.25'e kıyasla daha büyük gradyanlar üretir.

- Simetrik Dağılım:

Pozitif ve negatif aktivasyonların dengeli dağılımı, öğrenme sürecini stabilize eder.

- Diferansiyellenebilirlik:

Her noktada türevlenebilir olması, optimizasyon algoritmaları için uygun bir özelliktir.

```python
def tanh_func(x):
    return np.tanh(x)

# Örnek 1: Basit kullanım
print("Tanh örnekleri:")
values = [-3, -1, 0, 1, 3]
for val in values:
    result = tanh_func(val)
    print(f"tanh({val}) = {result:.3f}")

# Örnek 2: Sigmoid vs Tanh karşılaştırma
x = 2
print(f"x={x}: Sigmoid={sigmoid(x):.3f}, Tanh={tanh_func(x):.3f}")
```

Neden sigmoid'den daha iyi:

- Çıktılar −1 ile +1 arasında → daha iyi gradient flow
- Sıfır merkezli → ağırlıklar hem artabilir hem azalabilir

3. ReLU (Rectified Linear Unit) ve Varyasyonları :

Temel ReLU:

ReLU(x) = max(0,x)

Matematiksel Özellikler :

- Türev: ReLU'(x) = I(x > 0) (indikatör fonksiyonu)
- Kıvrım Noktası: x = 0'da türev tanımsız (subgradient: [0,1])

Gelişmiş Varyasyonlar:

1.Leaky ReLU:

LReLU(x) = max(αx,x) (genellikle α=0.01)

2.Parametrik ReLU:

PReLU(x) = max(αx,x) (α öğrenilir)

3.ELU:

ELU(x) = { x (x > 0), $\alpha(e^{x}-1)$ (x ≤ 0) }

KARŞILAŞTIRMALI ANALIZ:

Metrik	RELU	Leaky RELU	ELU
Ölü Nöron	Var	Azaltır	Önler
Negatif Değerler	0	αx	$\alpha(e^x-1)$
Ortalama Aktivasyon	>0	≈ 0	≈ 0

Örnek:

Bir yapay sinir ağı nöronuna 3 adet giriş veriliyor. Bu girişler ve ağırlıkları aşağıdaki gibidir:

- $x_1 = 1.2, w_1 = -0.4$
- $x_2 = -0.7, w_2 = 1.5$
- $x_3 = 2.0, w_3 = 0.3$

Bias (b) değeri: **0.2**

Bu nöronun net girdisini hesapla, ReLU aktivasyon fonksiyonu uygula ve çıkışı bulunuz.

1. Net girdi hesaplama (weighted sum + bias):

$$\text{net} = x_1 \cdot w_1 + x_2 \cdot w_2 + x_3 \cdot w_3 + b$$

$$= (1.2)(-0.4) + (-0.7)(1.5) + (2.0)(0.3) + 0.2$$

$$= -0.48 + (-1.05) + 0.6 + 0.2 = -0.73$$

2. ReLU fonksiyonu uygula:

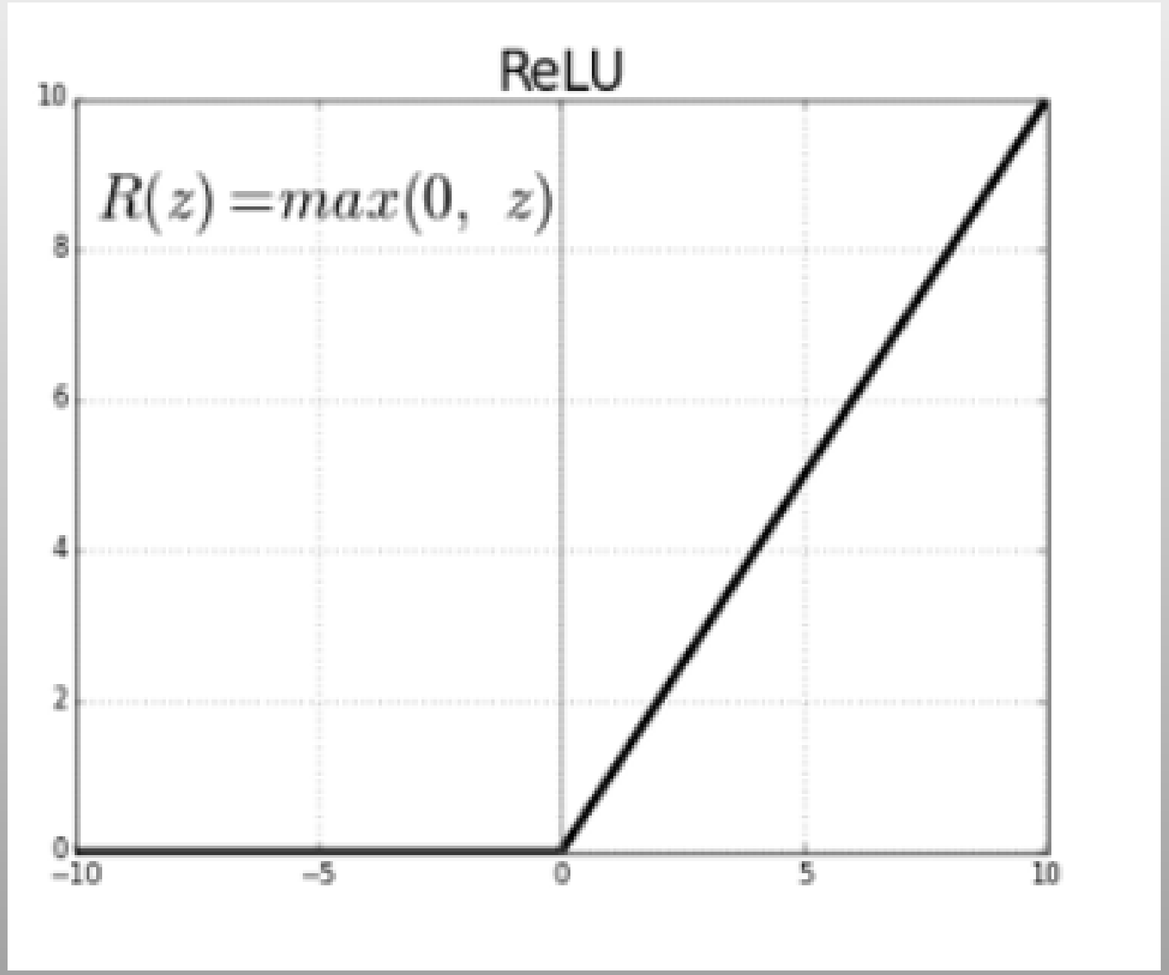
$$\text{ReLU}(-0.73) = 0 \quad (\text{çünkü negatif})$$

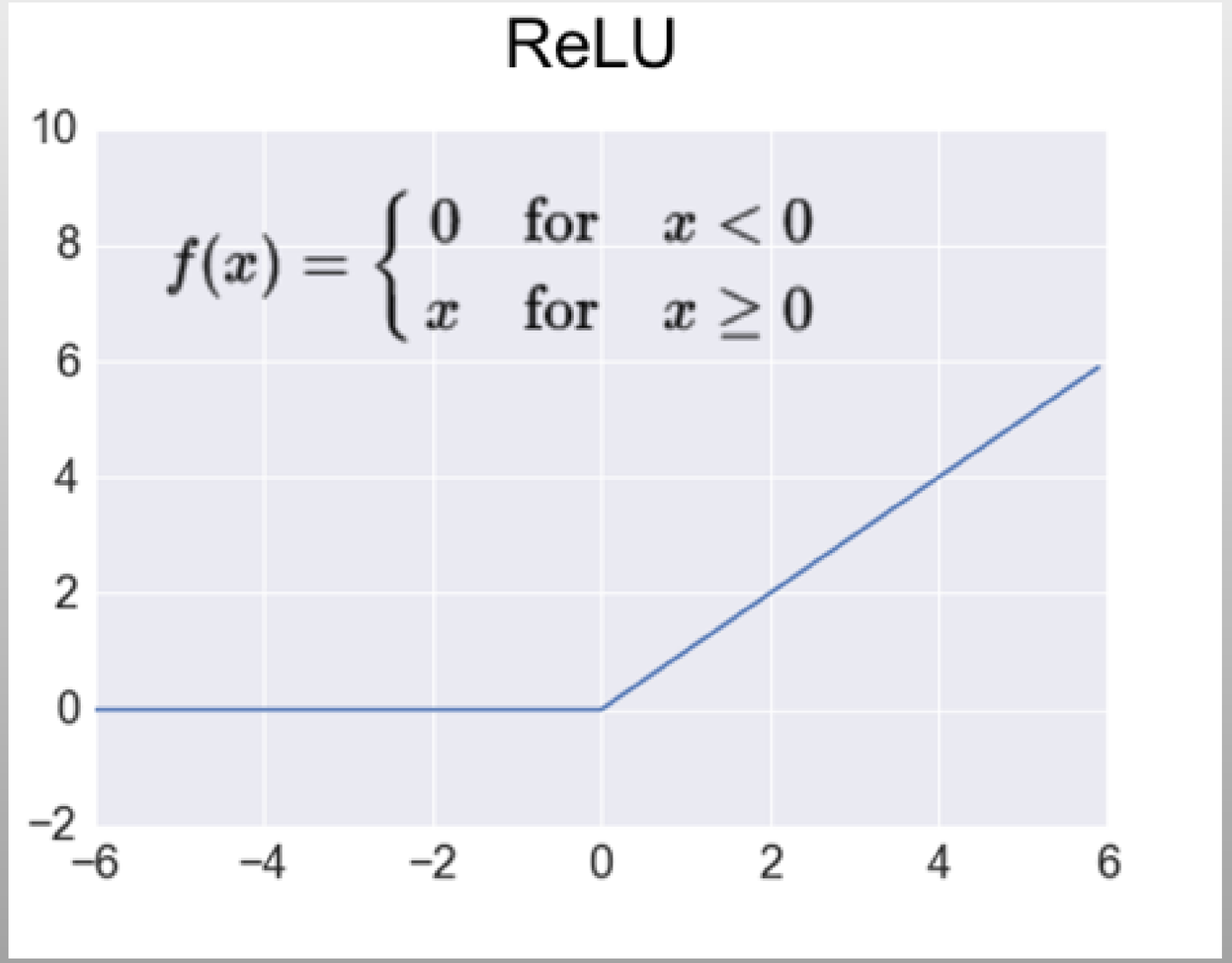
Nöronun çıktısı: **0**

Aynı soruda farklı bias olsaydı:

Örneğin bias = +2 olsaydı:

$$\text{Yeni net} = -0.73 + (2 - 0.2) = 1.07 \Rightarrow \text{ReLU}(1.07) = \boxed{1.07}$$


ReLU
$R(z) = max(0,\ z)$
10
8
6
4
2
0
−10
−5
0
5
10


ReLU
$f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$
10
8
6
4
2
0
−2
−6
−4
−2
0
2
4
6

```python
def relu(x):
    return np.maximum(0, x)

# Örnek 1: Basik kullanım
print("ReLU örnekleri:")
values = [-3, -1, 0, 1, 3]
for val in values:
    result = relu(val)
    print(f"relu({val}) = {result:.3f}")

# Örnek 2: Hidden layer simulasyonu
hidden_input = np.array([-0.5, 1.2, -2.0, 0.8])
hidden_output = relu(hidden_input)
print(f"Hidden giriş: {hidden_input}")
print(f"Hidden çıkış: {hidden_output}")
```

Neden popüler :

- Hesaplama hızı: Sadece max işlemi
- Vanishing gradient yok: Pozitif bölgede türev = 1
- Basitlik: Implement etmesi kolay

Ne yapar :

Negatif değerleri 0 yapar, pozitif değerleri olduğu gibi bırakır

Ne zaman kullanılır :

Hidden layer'larda, hızlı eğitim istediğimizde

4.SOFTMAX FONKSIYONU :

$$\text{Sofmax}(z)_i = e^{zi}/\sum_j e^{zj}$$

Özellikler :

- Çıktı vektörü bir olasılık dağılımıdır ($\sum_i p_i = 1$)
- Sıcaklık parametresi (T) ile ölçeklenebilir: Softmax(z/T)
- Jacobian matrisi: $\partial \text{Softmax}_i/\partial z_j = \text{Softmax}_i(\delta_{ij} - \text{Softmax}_j)$

Ne yapar :

Sayı listesini olasılık dağılımına dönüştürür (toplamları 1)

Ne zaman kullanılır :

Multi-class sınıflandırmanın son katmanında

Örnek:

Bir sinir ağının son katmanında 3 adet çıkış nöronu var. Bu nöronların **net giriş (logit)** değerleri şu şekilde:

- $z_1 = 1.2$
- $z_2 = 0.9$
- $z_3 = 0.4$

Bu değerlere **Softmax aktivasyon fonksiyonu** uygulandığında her bir çıkışın olasılık değeri ne olur?

1. Her e^{z_i}'yi hesapla:

$$e^{z_1} = e^{1.2} \approx 3.3201$$

$$e^{z_2} = e^{0.9} \approx 2.4596$$

$$e^{z_3} = e^{0.4} \approx 1.4918$$

2. Toplamı hesapla:

$$\text{Toplam} = 3.3201 + 2.4596 + 1.4918 = 7.2715$$

3. Softmax çıktılarını hesapla:

$$\text{Softmax}(z_1) = \frac{3.3201}{7.2715} \approx \boxed{0.4567}$$

$$\text{Softmax}(z_2) = \frac{2.4596}{7.2715} \approx \boxed{0.3382}$$

$$\text{Softmax}(z_3) = \frac{1.4918}{7.2715} \approx \boxed{0.2051}$$

Sonuç (Olasılık Dağılımı):

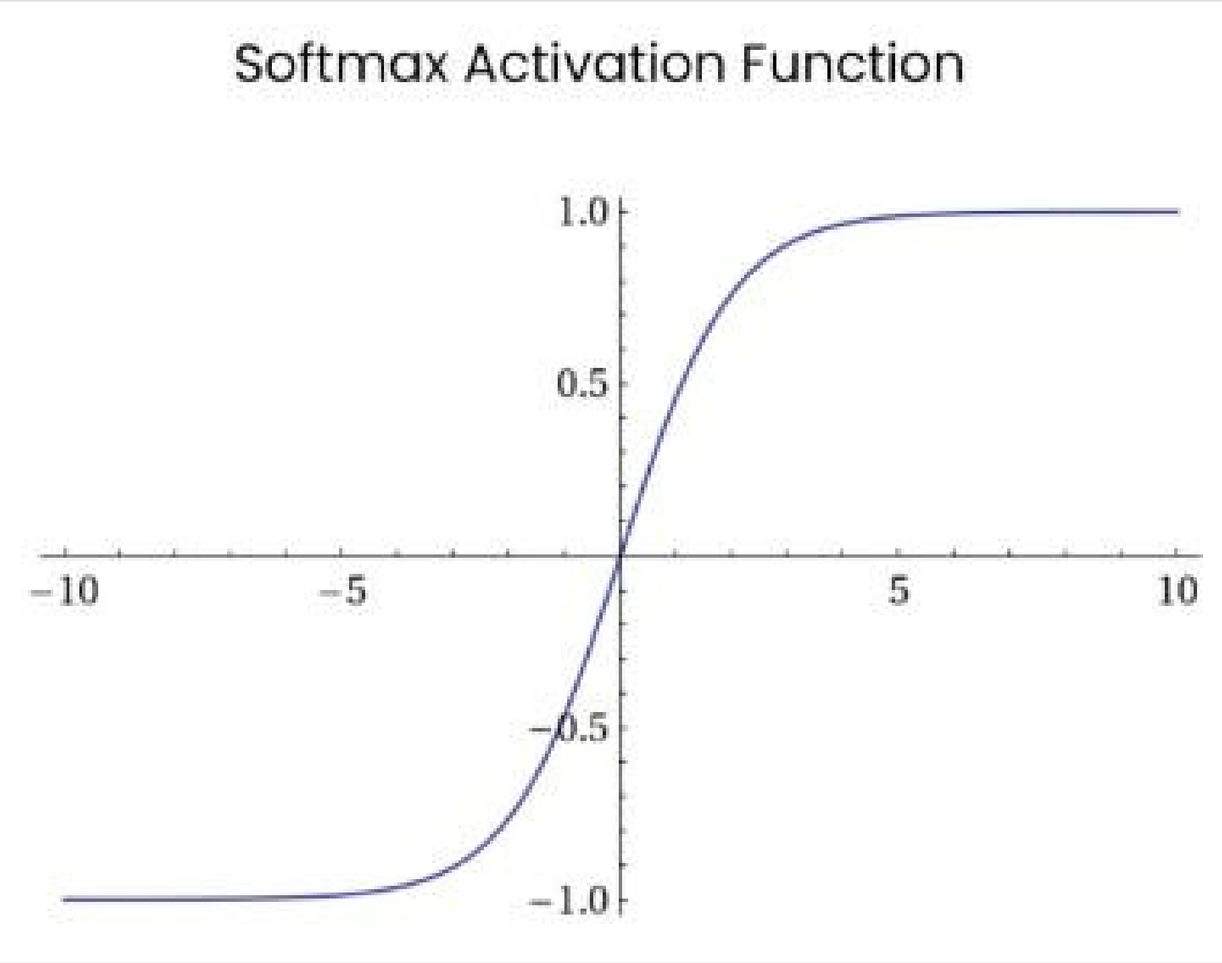
Sınıf	Logit (z)	Softmax Çıkışı (Olasılık)
1	1.2	0.4567
2	0.9	0.3382
3	0.4	0.2051

Toplam:

$$0.4567 + 0.3382 + 0.2051 = \boxed{1.0000}$$

Nerede Kullanılır?

- Çok sınıflı sınıflandırma problemleri (örneğin: el yazısı rakam tanıma)
- CNN'lerin sonunda
- NLP modellerinde (örneğin: kelime tahmini)



```python
def softmax(x):
    exp_vals = np.exp(x - np.max(x))  # Sayısal kararlılık için
    return exp_vals / np.sum(exp_vals)

# Örnek 1: 3 sınıflı problem
class_scores = [2.0, 1.0, 0.1]
probabilities = softmax(class_scores)
print("Sınıf skorları:", class_scores)
print("Olasılıklar:", probabilities)
print("Toplam:", np.sum(probabilities))

# Örnek 2: Tahmin yapma
classes = ['Köpek', 'Kedi', 'Kuş']
predicted_class = np.argmax(probabilities)
confidence = probabilities[predicted_class]
print(f"Tahmin: {classes[predicted_class]} (Güven: {confidence:.2f})")
```

Önemli özellikler:

- Tüm çıktılar 0–1 arasında
- Toplamları tam 1
- En yüksek skor → en yüksek olasılık

```
def softmax(x):
    # Batch işlemler için genişletilmiş versiyon
    if x.ndim == 1:
        x = x.reshape(1, -1)

    max_x = np.max(x, axis=1, keepdims=True)
    exp_x = np.exp(x - max_x)  # Numerik stabilite
    return exp_x / np.sum(exp_x, axis=1, keepdims=True)
```

Özet Tablo :

Aktivasyon	Aralık	Kullanım Yeri	Avantaj	Dezavantaj
Sigmoid	(0,1)	Son katman (binary)	Smooth, olasılık	Vanishing gradient
Tanh	(-1,1)	Hidden layer	Sıfır merkezli	Vanishing gradient
ReLU	[0,∞)	Hidden layer	Hızlı, basit	Dead neurons
Softmax	(0,1)	Son katman (multi-class)	Olasılık dağılımı	Sadece çıkış için

Aktivasyon Seçimi:

- Hidden layer'lar için: ReLU (varsayılan seçim)
- Binary classification çıkış: Sigmoid
- Multi-class classification çıkış: Softmax
- Regression çıkış: Lineer (aktivasyon yok)

PERCEPTRON VE XOR PROBLEMI

XOR (Exclusive OR) mantık kapısı, yapay sinir ağları tarihinde dönüm noktası oluşturan önemli bir problemdir. Bu problem, basit görünmesine rağmen tek katmanlı perceptronlarla çözülemeyen bir yapıya sahiptir.

1.Tek Katmanlı Perceptron Sorunu

Problem:

XOR fonksiyonu doğrusal olarak ayrılamaz

XOR Truth Table:

0 XOR 0 = 0

0 XOR 1 = 1

1 XOR 0 = 1

1 XOR 1 = 0

Bu veriyi tek bir çizgi ile ayırmak imkansızdır.
Gizli katman ekleyerek çözebiliriz

Çözüm Yaklaşımı : Gizli Katmanın Önemi (Multi–Layer Perceptron)

XOR problemini çözmek için bir gizli katman eklememiz gerekir

Bu yaklaşım şu avantajları sağlar :

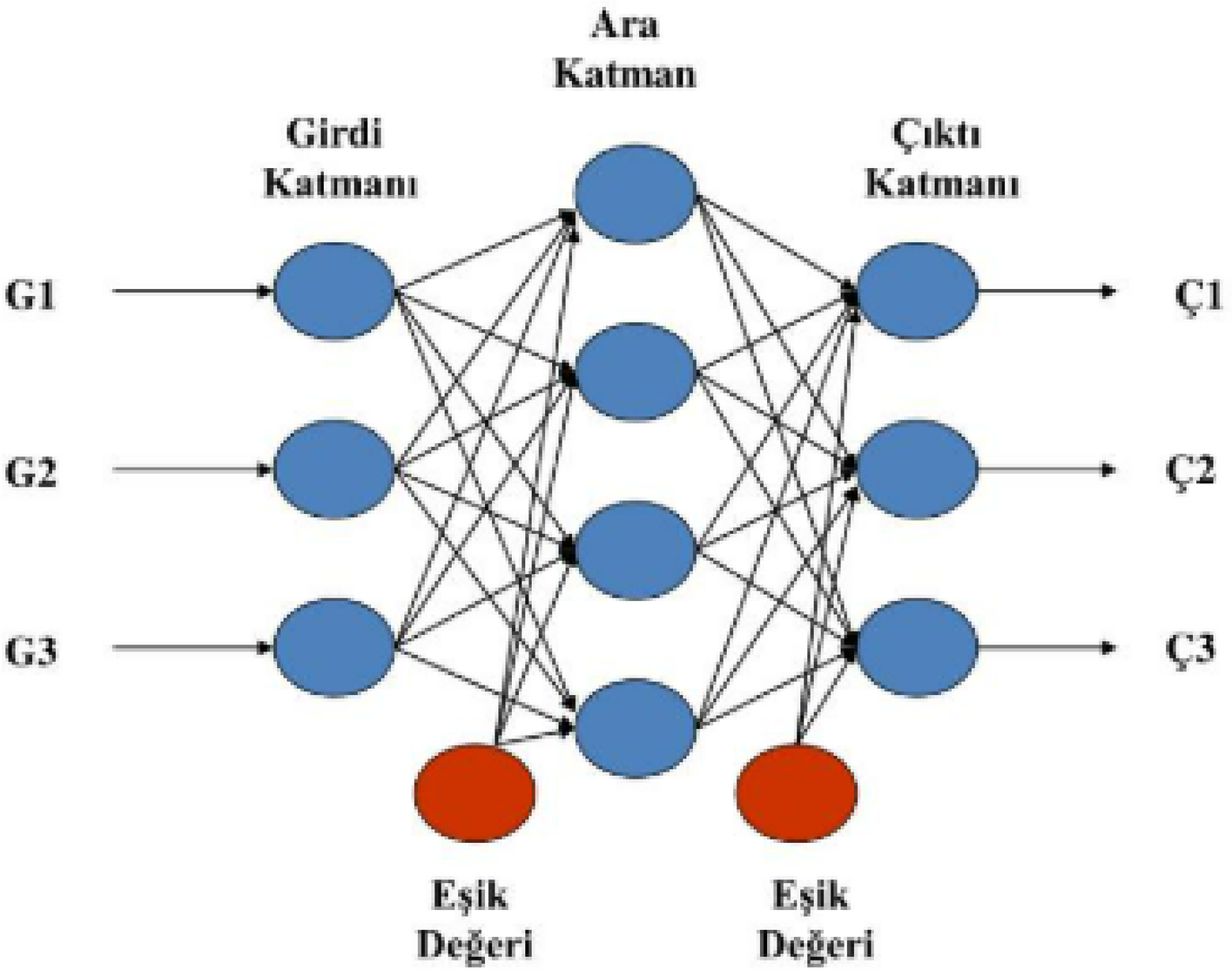
- Doğrusal Olmayan Dönüşüm:

  Gizli katman, girdi verisini daha yüksek boyutlu bir özellik uzayına eşler
- Ayrıştırma Kapasitesi:

  Veri noktalarını doğrusal olarak ayrılabilir hale getirir
- Hiyerarşik Öğrenme:

  Basit özelliklerin kombinasyonlarını öğrenir

Çok Katmanlı Ağ Modelinin Yapısı



Ağ Mimarisinin Detaylı Analizi :

Kullandığımız ağ mimarisi üç temel bileşenden oluşur:

Girdi Katmanı:

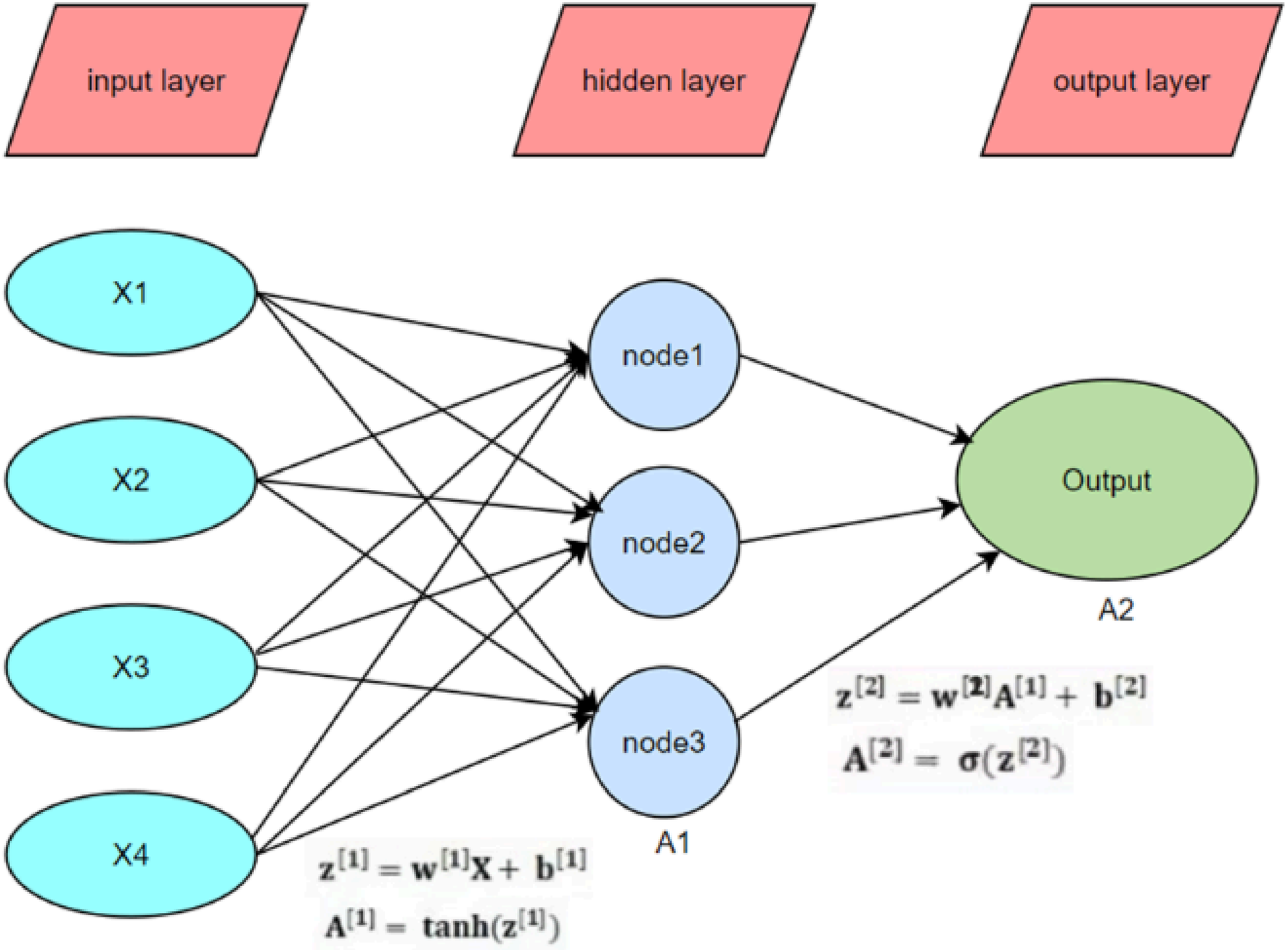
- 2 nöron (x_1 ve x_2 değerleri)
- Herhangi bir aktivasyon fonksiyonu yok

Gizli Katman:

- 2 nöron
- ReLU (Rectified Linear Unit) aktivasyonu
    1. Matematiksel formül: $f(x) = \max(0,x)$
    2. Avantajları: Hızlı hesaplama, gradyan kaybı sorununu azaltma

Çıktı Katmanı:

- 1 nöron
- Sigmoid aktivasyonu
    1. Matematiksel formül: $\sigma(x) = 1/(1+e^{-x})$
    2. Amacı: Çıktıyı 0–1 aralığına sıkıştırmak


input layer
hidden layer
output layer
X1
X2
X3
X4
node1
node2
node3
Output
A1
A2
$z^{[1]} = w^{[1]}X + b^{[1]}$
$A^{[1]} = \tanh(z^{[1]})$
$z^{[2]} = w^{[2]}A^{[1]} + b^{[2]}$
$A^{[2]} = \sigma(z^{[2]})$

Eğitim Süreci:

İleri Yayılım (Forward Propagation):

- Girdi verisi ağırlıklarla çarpılır ve bias eklenir:

$$z_1 = XW_1 + b_1$$

- ReLU aktivasyonu uygulanır:

$$a_1 = \text{relu}(z_1)$$

- Gizli katman çıktıları çıktı katmanına iletilir:

$$z_2 = a_1W_2 + b_2$$

- Sigmoid aktivasyonu uygulanır:

$$a_2 = \sigma(z_2)$$

Örnek:

Bir yapay sinir ağı iki katmandan oluşmaktadır.

- Girdi vektörü:

  $X = [1, -1]$

- Birinci katman ağırlıkları ve bias:

$$W_1 = \begin{bmatrix} 2 & -3 \\ 1 & 4 \end{bmatrix}, \quad b_1 = [0, 1]$$

- Birinci katmanda ReLU aktivasyonu uygulanmaktadır:

  $a_1 = \mathrm{ReLU}(XW_1 + b_1)$

- İkinci katman ağırlıkları ve bias:

$$W_2 = \begin{bmatrix} 3 \\ -2 \end{bmatrix}, \quad b_2 = [0.5]$$

- İkinci katmanda sigmoid aktivasyonu uygulanmaktadır:

  $a_2 = \sigma(a_1 W_2 + b_2)$

İleri yayılımı yaparak çıktı a2'yi bulunuz.

Verilenler:

$$X = [1, -1]$$

$$W_1 = \begin{bmatrix} 2 & -3 \\ 1 & 4 \end{bmatrix}, \quad b_1 = [0, 1]$$

$$W_2 = \begin{bmatrix} 3 \\ -2 \end{bmatrix}, \quad b_2 = [0.5]$$

1. Adım: $z_1 = XW_1 + b_1$

$$XW_1 = [1, -1] \times \begin{bmatrix} 2 & -3 \\ 1 & 4 \end{bmatrix} = [(1)(2) + (-1)(1), (1)(-3) + (-1)(4)] = [2 - 1, -3 - 4] = [1, -7]$$

$$z_1 = [1, -7] + [0, 1] = [1, -6]$$

2. Adım: ReLU uygula

$$a_1 = \mathrm{ReLU}(z_1) = [\max(0, 1), \max(0, -6)] = [1, 0]$$

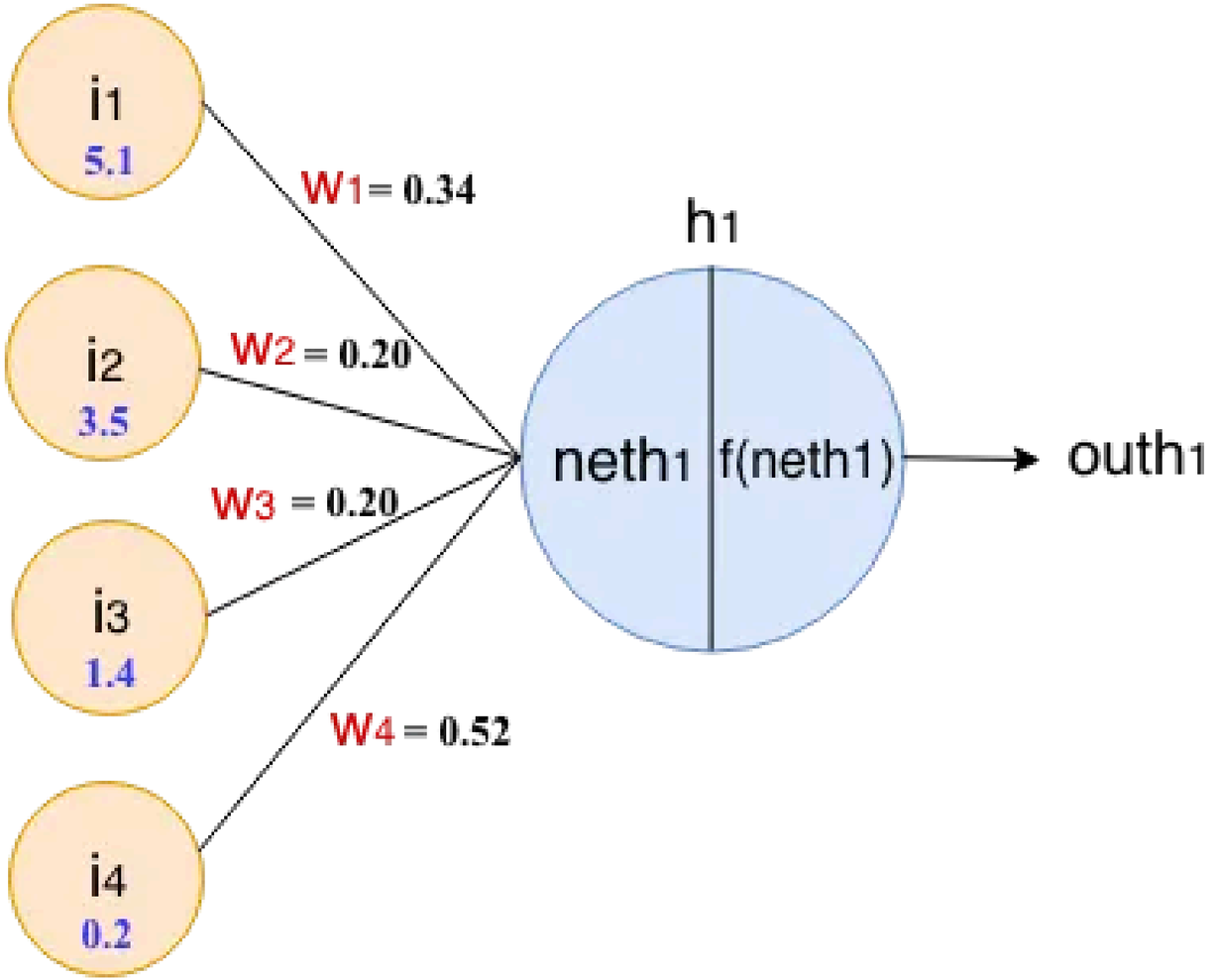
3. Adım: $z_2 = a_1 W_2 + b_2$

$$z_2 = [1, 0] \times \begin{bmatrix} 3 \\ -2 \end{bmatrix} + 0.5 = (1 \times 3) + (0 \times -2) + 0.5 = 3 + 0 + 0.5 = 3.5$$

4. Adım: Sigmoid uygula

$$a_2 = \frac{1}{1 + e^{-3.5}} \approx \frac{1}{1 + 0.0302} = 0.97$$

Modelin çıktısı $a_2 \approx \boxed{0.97}$



$$net = \sum w * i$$

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

out= f(net)

Hata Hesaplama:

- Binary Cross Entropy kaybı kullanılır : (Binary Cross-Entropy Loss (BCE))

- y: Gerçek sınıf etiketi (0 veya 1)
- a_2: Modelin tahmin ettiği olasılık (0 ile 1 arasında, genellikle sigmoid çıktı)
- L: Kayıp değeri (loss), küçük olması iyidir

Formül:

$$L = -\left[y \cdot \log(a_2) + (1 - y) \cdot \log(1 - a_2)\right]$$

- Eğer gerçek sınıf $y = 1$ ise, loss formülü:

$$L = -\log(a_2)$$

Yani modelin tahmini a_2'nin 1'e ne kadar yakın olduğuna bakar.

- Eğer gerçek sınıf $y = 0$ ise, loss formülü:

$$L = -\log(1 - a_2)$$

Modelin tahmini 0'a ne kadar yakınsa, loss o kadar küçük olur.

Örnek:

- Gerçek: $y = 1$, Tahmin: $a_2 = 0.9$

$$L = -[1 \cdot \log(0.9) + 0 \cdot \log(0.1)] = -\log(0.9) \approx 0.105$$

- Gerçek: $y = 1$, Tahmin: $a_2 = 0.1$

$$L = -\log(0.1) \approx 2.302$$

Yani, kötü tahmin yüksek kayıp verir.

Geri Yayılım (Backpropagation):

- Çıktı katmanı gradyanı:

$$\partial L/\partial z_2 = a_2 - y$$

- Gizli katman gradyanı:

$$\partial L/\partial z_1 = (\partial L/\partial z_2 \cdot W_2^T) \odot \text{relu}'(z_1)$$

- Ağırlık güncellemeleri:

$$W_2 -= \eta \cdot a_1^T \cdot \partial L/\partial z_2$$

$$b_2 -= \eta \cdot \Sigma \partial L/\partial z_2$$

$$W_1 -= \eta \cdot X^T \cdot \partial L/\partial z_1$$

$$b_1 -= \eta \cdot \Sigma \partial L/\partial z_1$$

1. Çıktı Katmanı Gradyanı

$$\frac{\partial L}{\partial z_2} = a_2 - y$$

- a_2: Modelin çıktı aktivasyonu (örneğin sigmoid sonrası değer)
- y: Gerçek etiket
- z_2: Çıktı katmanının toplam girdisi (aktivasyon öncesi)
- Bu ifade, çıktı katmanındaki hata sinyalini verir.

2. Gizli Katman Gradyanı

$$\frac{\partial L}{\partial z_1} = \left(\frac{\partial L}{\partial z_2} \cdot W_2^T \right) \odot \mathrm{ReLU}'(z_1)$$

- W_2^T: Çıktı katmanının ağırlık matrisinin transpozu
- $\odot$: Eleman bazlı (Hadamard) çarpım
- $\mathrm{ReLU}'(z_1)$: ReLU fonksiyonunun türevi (z1'de)
- Bu formül, hatayı gizli katmana geri yayar ve o katmandaki gradyanı hesaplar.

3. Ağırlık ve Bias Güncellemeleri

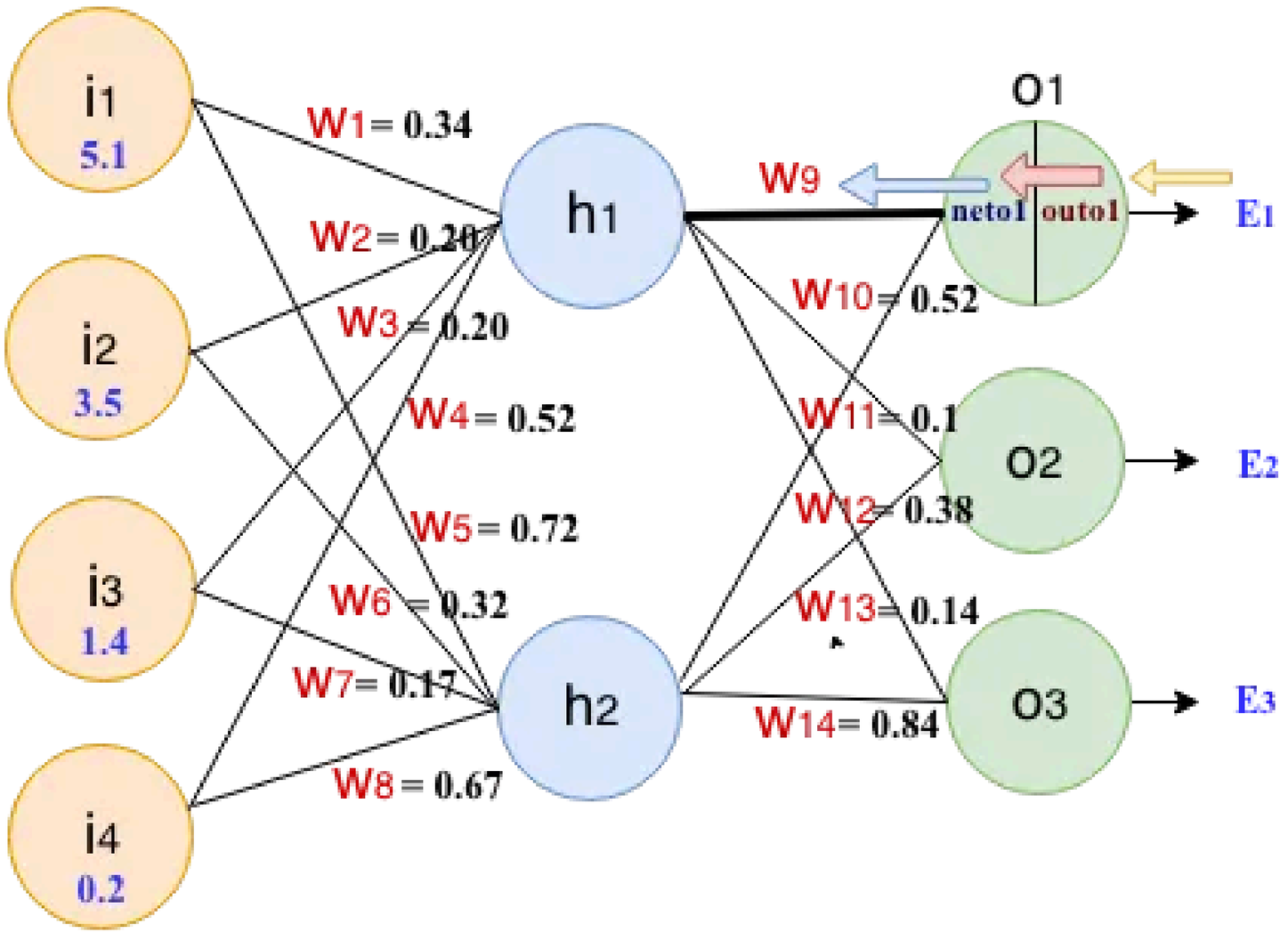
$$W_2 := W_2 - \eta \cdot a_1^T \cdot \frac{\partial L}{\partial z_2}$$

$$b_2 := b_2 - \eta \cdot \sum \frac{\partial L}{\partial z_2}$$

$$W_1 := W_1 - \eta \cdot X^T \cdot \frac{\partial L}{\partial z_1}$$

$$b_1 := b_1 - \eta \cdot \sum \frac{\partial L}{\partial z_1}$$

- η: Öğrenme oranı
- a_1^T: Gizli katman aktivasyonlarının transpozu
- X^T: Giriş vektörünün transpozu
- Her parametre, gradyanların yönünde ve büyüklüğünde güncellenir.


i1
5.1
i2
3.5
i3
1.4
i4
0.2
W1 = 0.34
W2 = 0.20
W3 = 0.20
W4 = 0.52
W5 = 0.72
W6 = 0.32
W7 = 0.17
W8 = 0.67
h1
h2
W9
W10 = 0.52
W11 = 0.1
W12 = 0.38
W13 = 0.14
W14 = 0.84
O1
neto1
outo1
O2
O3
E1
E2
E3

```python
class XORSolver:
    def __init__(self):
        # 2 giriş -> 3 hidden -> 1 çıkış
        self.W1 = np.random.randn(2, 3) * 0.5  # Hidden weights
        self.b1 = np.zeros(3)                   # Hidden bias
        self.W2 = np.random.randn(3, 1) * 0.5  # Output weights
        self.b2 = np.zeros(1)                   # Output bias

    def forward(self, X):
        # Hidden layer: giriş -> sigmoid dönüşümü
        z1 = np.dot(X, self.W1) + self.b1
        a1 = sigmoid(z1)

        # Output layer: hidden -> final sigmoid
        z2 = np.dot(a1, self.W2) + self.b2
        output = sigmoid(z2)
        return output, a1  # Geriye dönük için a1 lazım
```

```python
    def train_step(self, X, y, lr=0.5):
        # İleri geçiş
        output, hidden = self.forward(X)

        # Hata hesapla
        error = y - output

        # Çıkış katmanı güncelleme
        output_delta = error * output * (1 - output)  # Sigmoid türevi
        self.W2 += lr * np.dot(hidden.T, output_delta)
        self.b2 += lr * np.sum(output_delta, axis=0)

        # Hidden katman güncelleme
        hidden_error = np.dot(output_delta, self.W2.T)
        hidden_delta = hidden_error * hidden * (1 - hidden)
        self.W1 += lr * np.dot(X.T, hidden_delta)
        self.b1 += lr * np.sum(hidden_delta, axis=0)

# XOR veri seti
X_xor = np.array([[0,0], [0,1], [1,0], [1,1]])
y_xor = np.array([[0], [1], [1], [0]])
```

```python
# Model eğitimi
model = XORSolver()
print("XOR öğrenme süreci:")

for epoch in range(0, 3001, 1000):
    for _ in range(1000):
        model.train_step(X_xor, y_xor)

    # Test sonuçları
    predictions, _ = model.forward(X_xor)
    avg_error = np.mean((y_xor - predictions)**2)
    print(f"Epoch {epoch}: Ortalama hata = {avg_error:.4f}")

# Final test
print("\nFinal XOR sonuçları:")
final_pred, _ = model.forward(X_xor)
for i in range(4):
    x1, x2 = X_xor[i]
    pred = final_pred[i][0]
    target = y_xor[i][0]
    print(f"{x1} XOR {x2} = {pred:.3f} (hedef: {target})")
```

Gerçek Hayat Uygulaması:

Akıllı Ev Aydınlatma Kontrol Sistemi (XOR Tabanlı)

Problem Senaryosu:

Bir evde iki hareket sensörü (A ve B) ve bir akıllı lamba bulunuyor. Sistemin çalışma mantığı:

- Sensör A: Kapı girişi
- Sensör B: Pencere bölgesi

Kural:

"Sadece bir sensör hareket algılarsa lamba yansın"

(Yani XOR mantığı: 0–1→1, 1–0→1, diğer durumlarda 0)

Python Uygulaması:

```python
import numpy as np
import time

class AkilliLambaSistemi:
    def __init__(self):
        # XOR problemi için eğitilmiş ağırlıklar
        self.W1 = np.array([[20, -20], [20, -20]])  # Girdiden gizli katmana
        self.b1 = np.array([[-30, 10]])             # Bias değerleri
        self.W2 = np.array([[20], [20]])            # Gizliden çıktıya
        self.b2 = np.array([[-10]])                 # Çıktı bias'ı

    def relu(self, x):
        return np.maximum(0, x)

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def lamba_kontrol(self, sensor_A, sensor_B):
        """Sensör durumuna göre lambayı kontrol eder"""
        # Girdi vektörü oluştur
        X = np.array([[sensor_A, sensor_B]])

        # İleri yayılım
        hidden = self.relu(X.dot(self.W1) + self.b1)
        output = self.sigmoid(hidden.dot(self.W2) + self.b2)

        # Eşik değeri kontrolü (0.5)
        return output > 0.5
```

```
# Sistemi test edelim
sistem = AkilliLambaSistemi()

print("Akıllı Lamba Kontrol Sistemi Başlatılıyor...")
print("(Sensör A | Sensör B) => Lamba Durumu")

test_durumlari = [(0,0), (0,1), (1,0), (1,1)]
for durum in test_durumlari:
    sensor_A, sensor_B = durum
    lamba_durumu = sistem.lamba_kontrol(sensor_A, sensor_B)
    print(f"({sensor_A} | {sensor_B}) => {'AÇIK' if lamba_durumu else 'KAPALI'}")
    time.sleep(1)
```

Çıktı:

```
Akıllı Lamba Kontrol Sistemi Başlatılıyor...
(Sensör A | Sensör B) => Lamba Durumu
(0 | 0) => KAPALI
(0 | 1) => AÇIK
(1 | 0) => AÇIK
(1 | 1) => KAPALI
```

Sistemin Çalışma Mantığı :

Girdiler:

- sensor_A: Kapıdaki hareket (1: hareket var, 0: yok)
- sensor_B: Penceredeki hareket (1: hareket var, 0: yok)

XOR Mantığı:

- Sadece bir sensör hareket algılarsa (0–1 veya 1–0) lamba yanar
- Her ikisi de algılarsa (1–1) hırsız alarmı tetiklenebilir (güvenlik nedeniyle lamba kapalı)
- Hareket yoksa (0–0) lamba kapalı

Nöral Ağ Yapısı:

- Gizli katman: Sensör verilerini doğrusal olmayan şekilde birleştirir
- Çıktı katmanı: Lamba kontrol kararını verir (sigmoid > 0.5 ise açık)