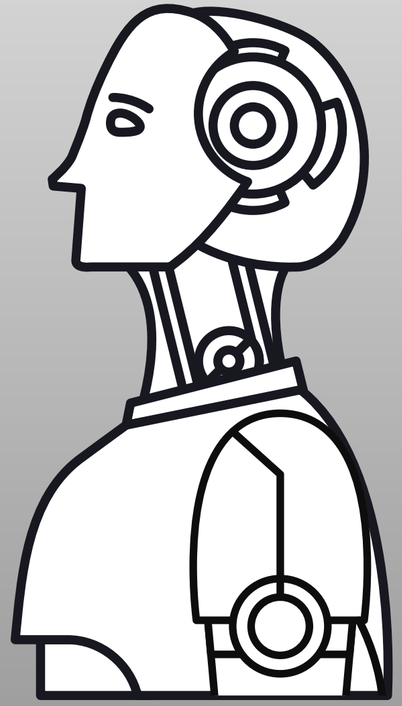
3.HAFTA

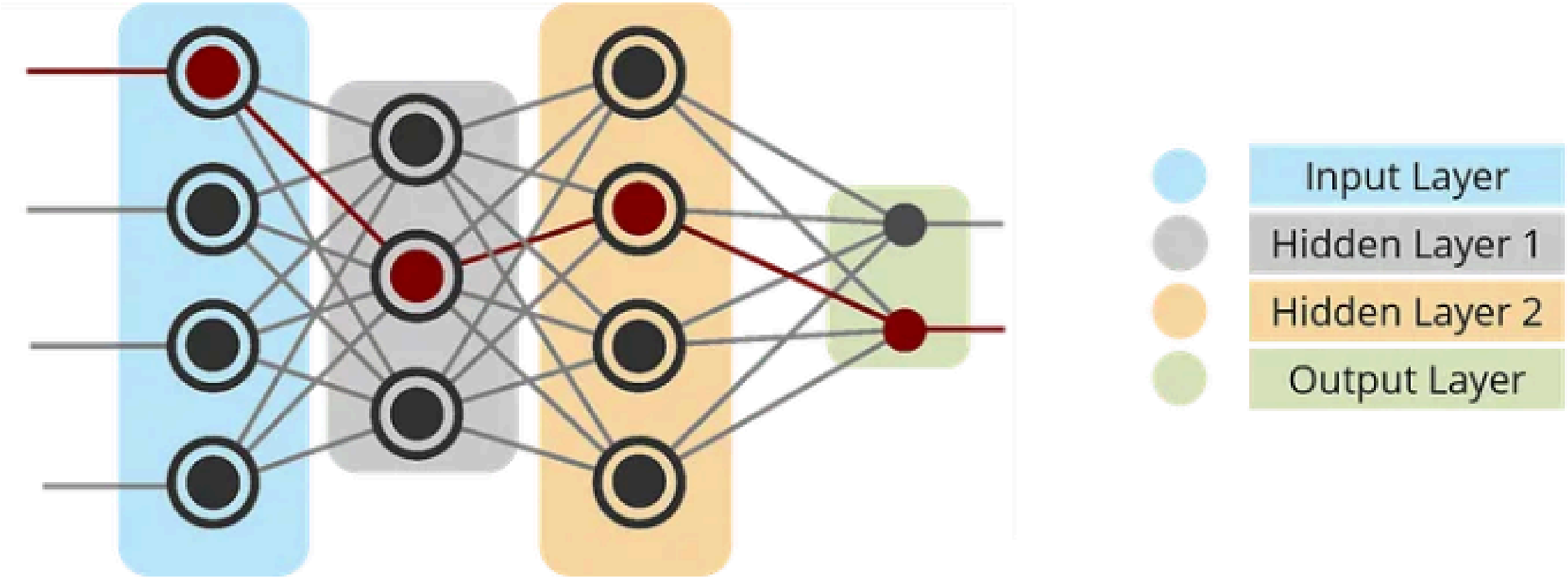
YAPAY SİNİR AĞLARI



SAMSUN ÜNİVERSİTESİ

DR. ÖĞR. ÜYESİ ALPER TALHA KARADENİZ

MLP (Multi-Layer-Perceptron**)**

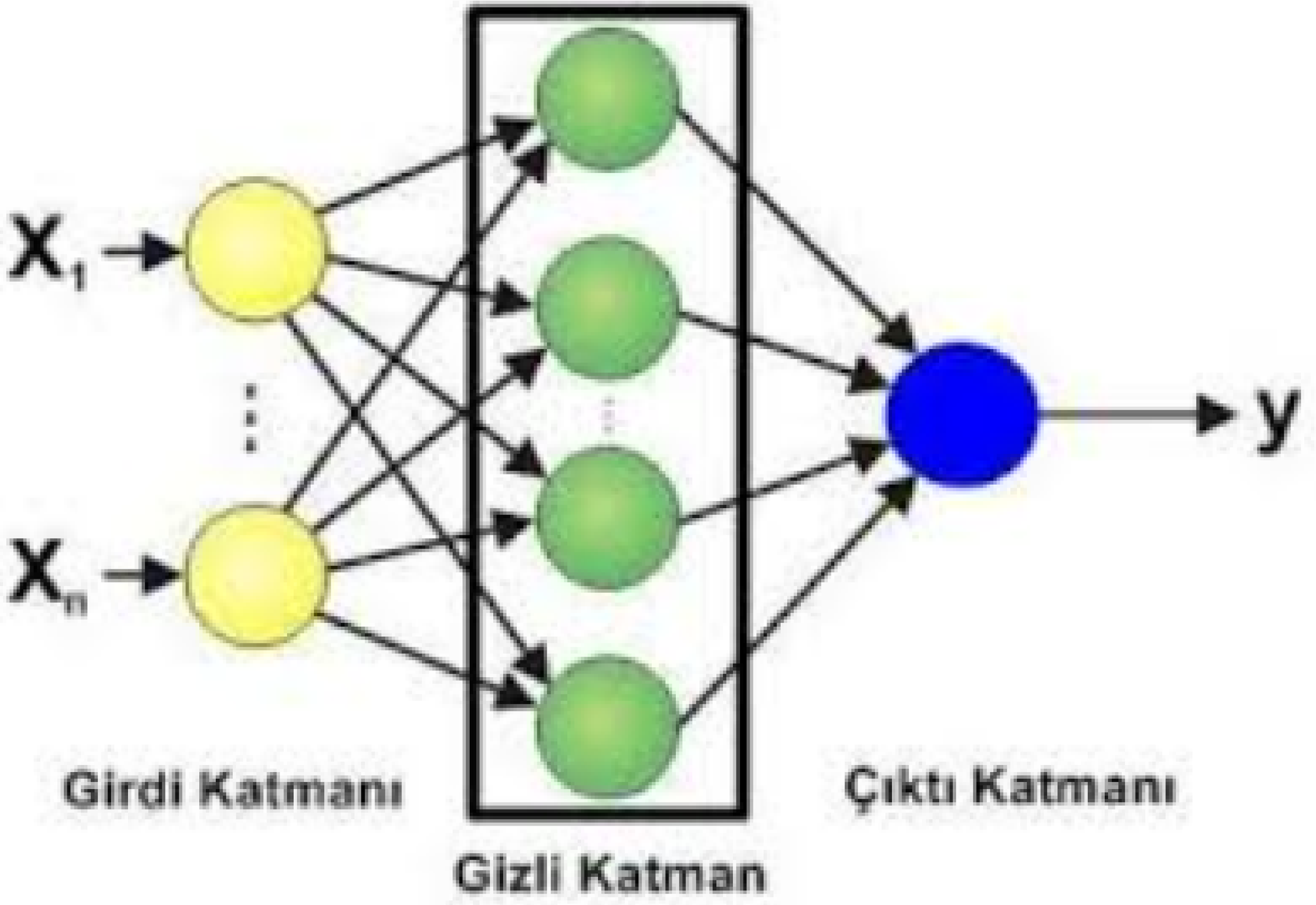


MLP :

MLP, yani Çok Katmanlı Algılayıcı (Multi-Layer Perceptron), yapay sinir ağlarının en temel ve yaygın formlarından biridir. Bu yapı, insan beyninin bilgi işleme şeklini taklit etmeye çalışan matematiksel bir modeldir. MLP, birbirine bağlı nöronlardan (yapay sinir hücrelerinden) oluşan birden fazla katmana sahiptir. Bu katmanlar genellikle bir giriş katmanı, bir veya daha fazla gizli katman ve bir çıktı katmanı olarak düzenlenir. MLP'nin temel özellikleri şunlardır:

1. Katmanlı Yapı:

- Giriş Katmanı: Verilerin model tarafından alındığı ilk noktadır.
- Gizli Katman(lar): Her biri bir dizi ağırlık ve bias içeren ve karmaşık özelliklerin öğrenildiği katmanlardır.
- Çıktı Katmanı: Sonuçların (tahminlerin) üretildiği katmandır.


X1
Xn
y
Girdi Katmanı
Gizli Katman
Çıktı Katmanı

2. Ağırlıklar ve Biaslar: Her bağlantı, bir ağırlığa (verinin önemini belirleyen) ve her nöron, bir biasa (aktivasyon eşiğini ayarlayan) sahiptir.

3. Aktivasyon Fonksiyonları: Her nöronun çıktısı, genellikle bir aktivasyon fonksiyonu (örneğin, sigmoid, ReLU) tarafından işlenir. Bu fonksiyonlar, nöronların lineer olmayan karmaşık örüntüleri öğrenmesini sağlar.

4. Öğrenme Süreci: MLP, genellikle arka yayılım (backpropagation) ve gradyan inişi gibi yöntemler kullanarak eğitilir. Bu süreçte, ağın çıktıları ile gerçek değerler arasındaki hata hesaplanır ve bu hata, ağırlıkları ve biasları ayarlamak için kullanılır.

Ne zaman ortaya çıktı?

Çok Katmanlı Algılayıcılar (MLP), yapay sinir ağlarının ilk formasyonlarından biridir ve 1980'lerin başlarında popülerlik kazanmıştır. Frank Rosenblatt'ın 1958'de icat ettiği basit algılayıcı kavramı üzerine inşa edilmiştir. Bu modelin gelişimi, arka yayılım (backpropagation) algoritmasının 1986'da Rumelhart, Hinton ve Williams tarafından tanıtılmasıyla ivme kazanmıştır.

Neden ortaya çıktı?

MLP'nin temel amacı, insan beyninin karmaşık öğrenme süreçlerini taklit ederek, makine öğrenmesi problemlerinde daha etkili çözümler sunmaktır. Yapay sinir ağları, girdi (input) ve çıktı (output) arasındaki karmaşık ilişkileri modellemek için kullanılır ve MLP, bu ilişkileri birden fazla katman aracılığıyla daha derinlemesine öğrenme yeteneğine sahiptir.

Kullanım alanları

- **Görüntü ve Ses İşleme:** Görüntü tanıma, ses tanıma gibi alanlarda etkili sonuçlar verir.
- **Finans:** Kredi skorlaması, piyasa analizi gibi finansal tahminlerde kullanılır.
- **Tıp:** Hastalık teşhisi, ilaç keşfi gibi alanlarda biyomedikal verilerin analizinde etkilidir.
- **Robotik ve Kontrol Sistemleri:** Otomatik kontrol sistemlerinde ve robotik uygulamalarda karar verme süreçlerinde kullanılır.

Avantajları

- **Esneklik:** Farklı tipteki veri setleriyle çalışabilme yeteneği.
- **Genelleme Kabiliyeti:** Yeni ve görülmemiş verilere adapte olabilme.
- **Derin Öğrenme:** Çok katmanlı yapısı sayesinde karmaşık örüntüleri ve ilişkileri öğrenebilme.

Dezavantajları

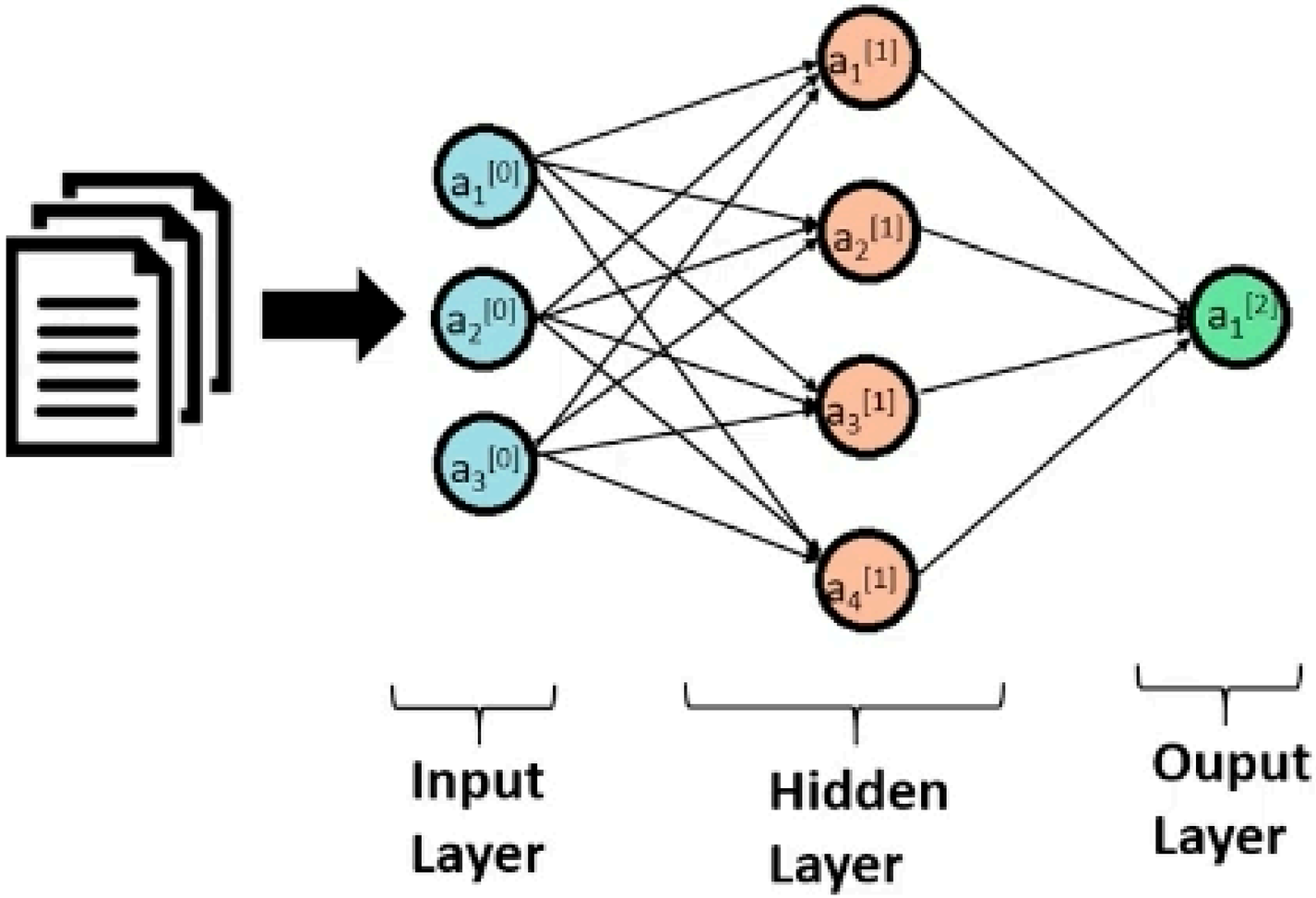
- **Aşırı Uyuma (Overfitting):** Eğitim verisine aşırı uyum sağlayarak genel veri seti üzerinde kötü performans.
- **Eğitim Zorluğu:** Çok sayıda hiperparametre ve uzun eğitim süreleri.
- **Yerel Minimum Sorunu:** Optimizasyon sırasında yerel minimumlara takılma olasılığı.

FORWARD PROPAGATION:

İleri yayılım, bir sinir ağının eğitilmesinde ilk adımdır; burada girdi verileri, bir tahmin üretmek için ağ üzerinden geçirilir.

Giriş Verileri

İleri beslemeli bir sinir ağında, veriler tek bir yönde akar: girdiden çıktıya. Giriş katmanı, ağa girdiğiniz verilerden oluşur. Bu, probleminize uygun herhangi bir veri türü olabilir; örneğin görseller, metinler veya sayısal veriler. Veriler genellikle vektörleştirilir veya ağın işleyebileceği dizilere dönüştürülür.



$$a_1^{[1]} = \text{activation\_function}(W_{11}^{[1]} * a_1^{[0]} + W_{12}^{[1]} * a_2^{[0]} + W_{13}^{[1]} * a_3^{[1]} + B1)$$

$$a_2^{[1]} = \text{activation\_function}(W_{21}^{[1]} * a_1^{[1]} + W_{22}^{[1]} * a_2^{[1]} + W_{23}^{[1]} * a_3^{[1]} + B1)$$

Ağırlıklar ve Önyargılar

Ağırlıklar ve önyargılar, bir sinir ağının öğrenilebilir parametreleridir. Ağırlıklar, belirli bir giriş nöronunun çıktı üzerindeki etkisinin gücünü belirlerken, önyargılar çıktının sabit bir değer kadar kaydırılmasına olanak tanır.

- **Ağırlıklar:** Nöronlar arasındaki her bağlantıya, göreceli öneminin bir ölçüsü olan bir ağırlık atanır. Bir nörondan bir girdi geçtiğinde, bu ağırlıkla çarpılır. Ağırlıklar hem pozitif (uyarıcı) hem de negatif (engelleyici) olabilir ve eğitimin başlangıcında rastgele başlatılırlar.
- **Önyargı:** Önyargı, nöronlara ekstra bir girdidir ve her zaman 1'dir ve kendi bağlantı ağırlığına sahiptir. Bu önyargı ağırlığı, aktivasyon fonksiyonunu sola veya sağa kaydırma yeteneği sağlar ve bu da başarılı öğrenme için kritik olabilir.

Nöronlar ve Katmanlar

Giriş verileri, nöron veya düğüm dediğimiz yapılara aktarılır. Bu nöronlar, bir giriş katmanı, gizli katmanlar ve bir çıkış katmanı olmak üzere birden fazla katman halinde düzenlenir.

- **Giriş Katmanı:** Ağın veri kümenizden girdi aldığı yer burasıdır. Genellikle ağ için uygun bir forma dönüştürülür.
- **Gizli Katman(lar):** Bunlar, giriş ve çıkış katmanları arasında yer alan katmanlardır. Mevcut görev için faydalı özellikleri öğrenme amacıyla giriş verileri üzerinde dönüşümler gerçekleştirirler. Derin öğrenmede "derin" terimi, ağda birden fazla gizli katman bulunması anlamına gelir.
- **Çıktı Katmanı:** Bu son katmandır. Ağın eğitilerek ürettiği tahminleri veya sınıflandırmaları sağlar.

Doğrusal Dönüşüm

İleri yayılımın ilk adımı, girdilerin ve ilişkili ağırlıkların ağırlıklı toplamının hesaplanmasını ve ardından sapmanın eklenmesini içerir. Bu aynı zamanda doğrusal dönüşüm olarak da bilinir. Denklem genellikle şöyledir:

> Bir katmanda **giriş vektörü** $\mathbf{x}$ (örneğin, önceki katmanın çıktısı) ağırlık matrisi $\mathbf{W}$ ile çarpılır ve bir **bias vektörü** $\mathbf{b}$ eklenir:
>
> $$\mathbf{z} = \mathbf{W} \cdot \mathbf{x} + \mathbf{b}$$
>
> Burada:
>
> - $\mathbf{x}$ — giriş vektörü (boyut: $n \times 1$)
> - $\mathbf{W}$ — ağırlık matrisi (boyut: $m \times n$)
> - $\mathbf{b}$ — bias vektörü (boyut: $m \times 1$)
> - $\mathbf{z}$ — lineer dönüşüm sonucu (boyut: $m \times 1$)

Örnek :

Bir katmanda 3 giriş nöronu ve 2 çıkış nöronu var

Verilenler:

$$\mathbf{x} = \begin{bmatrix} 2 \\ -1 \\ 3 \end{bmatrix}, \quad \mathbf{W} = \begin{bmatrix} 1 & 0 & -1 \\ 2 & -1 & 1 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Katmana uygulanan lineer dönüşüm işleminin sonucu ne olur?

Çözüm:

1. Matris çarpımı:

$$\mathbf{W} \cdot \mathbf{x} = \begin{bmatrix} 1 & 0 & -1 \\ 2 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ -1 \\ 3 \end{bmatrix} = \begin{bmatrix} 1*2+0*(-1)+(-1)*3 \\ 2*2+(-1)*(-1)+1*3 \end{bmatrix} = \begin{bmatrix} 2+0-3 \\ 4+1+3 \end{bmatrix} = \begin{bmatrix} -1 \\ 8 \end{bmatrix}$$

2. Bias ekleme:

$$\mathbf{z} = \begin{bmatrix} -1 \\ 8 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 8 \end{bmatrix}$$

Sonuç:

$$\mathbf{z} = \begin{bmatrix} 0 \\ 8 \end{bmatrix}$$

Ağırlıkların ve Önyargıların Başlatılması

Bir sinir ağını eğitmeden önce, her nöron için ağırlıkları ve sapmaları başlatmamız gerekir. Başlangıçta bu değerler rastgele ayarlanır. Bu, farklı nöronlar arasındaki simetriyi bozduğu ve eğitim sırasında farklı özellikler öğrenmelerine olanak tanıdığı için çok önemlidir. Tüm ağırlıklar aynı değere başlatılsaydı, bir katmandaki tüm nöronlar aynı özellikleri öğrenirdi ki bu da istediğimiz bir şey değildir.

1. Giriş Katmanı (Input Layer)

- **Görev:** Modelin dış dünyadan aldığı ham veriyi temsil eder.
- **Yapısı:** Her bir nöron, bir giriş özelliğini (feature) temsil eder.
- **Örnek:**
  - MNIST veri setinde, her resim 28x28 piksel, yani toplam 784 piksel.
  - Bu yüzden giriş katmanında 784 nöron olur; her nöron bir pikselin parlaklık değerini temsil eder.
- **Not:** Giriş katmanı sadece veriyi alır, herhangi bir ağırlıkla işleme yapmaz.

2. Gizli Katmanlar (Hidden Layers)

- **Görev:** Girdi veriden anlamlı özellikleri çıkarır, modelin karmaşık örüntüleri öğrenmesini sağlar.
- **Özellikler:**
  - Sayısı ve her katmandaki nöron sayısı **hiperparametre** olarak adlandırılır.
  - Modelin kapasitesini ve karmaşıklığını belirler.
  - Çok fazla gizli katman ve nöron fazla öğrenmeye (overfitting) sebep olabilir.
  - Çok az olursa model yetersiz öğrenir (underfitting).

Gizli Katman Sayısı ve Boyutunun Belirlenmesi

- **Deneme-yanılma (trial and error):**
  - Farklı sayıda katman ve nöron denenir, performans karşılaştırılır.
- **Hiperparametre optimizasyonu yöntemleri:**
  - **Grid Search:** Belirlenen parametre aralıkları sistematik denenir.
  - **Random Search:** Parametreler rastgele seçilerek test edilir.
  - **Bayesian Optimization:** Akıllı tahminlerle optimum parametre bulunmaya çalışılır.
- **Veri ve problem karmaşıklığı:**
  - Basit problemlerde az sayıda gizli katman ve nöron yeterli olur.
  - Karmaşık problemler için derin (çok katmanlı) yapılar gerekir.

Derinlik Kavramı (Deep Learning)

- **Derinlik:** Gizli katman sayısıdır.
- **Derin öğrenme:** Genellikle 2 ve üzeri gizli katmanı olan ağlar için kullanılır.
- Derin ağlar, karmaşık ve yüksek boyutlu verilerde çok güçlü sonuçlar verir.
- Ancak eğitimleri zor olabilir, çok veri ve hesaplama gücü gerektirir.

3. Çıkış Katmanı (Output Layer)

- **Görev:** Modelin sonuçlarını üretir.
- **Yapısı:**
  - Problem tipine göre katmandaki nöron sayısı ve aktivasyon fonksiyonu seçilir.

Aktivasyon Fonksiyonu Seçimi:

Problem Türü	Çıkış Nöron Sayısı	Aktivasyon Fonksiyonu	Notlar
İkili Sınıflandırma	1	Sigmoid	0-1 arasında olasılık verir
Çoklu Sınıflandırma	Sınıf sayısı kadar	Softmax	Sınıflar olasılık olarak çıktı
Regresyon	1 veya daha fazla	Doğrusal (aktivasyonsuz) veya ReLU	Sürekli değer tahmini

- **İkili Sınıflandırma:** Hasta/hasta değil, spam/normal e-posta.
  - Çıkış katmanı 1 nöron, sigmoid fonksiyonuyla çıktı olasılık verilir.
- **Çoklu Sınıflandırma:** MNIST el yazısı rakam sınıflandırması (0-9).
  - Çıkış katmanı 10 nöron, softmax aktivasyonuyla her rakam için olasılık verilir.
- **Regresyon:** Ev fiyatı tahmini.
  - Çıkış katmanı 1 nöron, aktivasyon fonksiyonu kullanılmayabilir.

4. Özet Tablosu

Katman Türü	Görev	Örnek	Not
Giriş Katmanı	Ham veriyi almak	784 nöron (MNIST)	Veriyi doğrudan alır, işlem yapmaz
Gizli Katmanlar	Özellik çıkarımı ve örüntü öğrenme	1-100+ nöron, 1-10+ katman	Sayı ve boyut deneyimle ve optimizasyonla belirlenir
Çıkış Katmanı	Model tahminini üretir	1 nöron (binary), 10 nöron (multiclass)	Problem türüne göre aktivasyon seçilir

Örnek :

Bir yapay sinir ağının yapısı bu şekildedir

- **Giriş katmanı:** 2 nöron (x_1 ve x_2)
- **1 gizli katman:** 2 nöron (h_1, h_2)
- **Çıkış katmanı:** 1 nöron (y)
- **Aktivasyon fonksiyonu: Sigmoid**

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

Verilenler:

Girişler:

$$x_1 = 1, \quad x_2 = 2$$

Gizli katman ağırlıkları ve bias:

$$w_{x1,h1} = 0.5, \quad w_{x2,h1} = 0.3, \quad b_{h1} = 0.1$$

$$w_{x1,h2} = -0.4, \quad w_{x2,h2} = 0.2, \quad b_{h2} = -0.2$$

Çıkış katmanı ağırlıkları ve bias:

$$w_{h1,y} = 0.6, \quad w_{h2,y} = -0.1, \quad b_y = 0.05$$

1. Gizli katmandaki h_1 ve h_2 nöronlarının ağırlıklı toplam (z) ve aktivasyon (a) değerlerini hesapla.
2. Çıkış katmanındaki y nöronunun ağırlıklı toplam (z) ve aktivasyon (a) değerini hesapla.

Gizli Katman Hesaplamaları

h_1 için ağırlıklı toplam:

$$z_{h1} = w_{x1,h1} \times x_1 + w_{x2,h1} \times x_2 + b_{h1} = 0.5 \times 1 + 0.3 \times 2 + 0.1 = 0.5 + 0.6 + 0.1 = 1.2$$

h_1 aktivasyon:

$$a_{h1} = \sigma(1.2) = \frac{1}{1 + e^{-1.2}} \approx \frac{1}{1 + 0.301} = 0.7685$$

h_2 için ağırlıklı toplam:

$$z_{h2} = w_{x1,h2} \times x_1 + w_{x2,h2} \times x_2 + b_{h2} = -0.4 \times 1 + 0.2 \times 2 - 0.2 = -0.4 + 0.4 - 0.2 = -0.2$$

h_2 aktivasyon:

$$a_{h2} = \sigma(-0.2) = \frac{1}{1 + e^{0.2}} \approx \frac{1}{1 + 1.221} = 0.4502$$

Çıkış Katmanı Hesaplamaları

Ağırlıklı toplam:

$$z_y = w_{h1,y} \times a_{h1} + w_{h2,y} \times a_{h2} + b_y = 0.6 \times 0.7685 + (-0.1) \times 0.4502 + 0.05$$

$$z_y = 0.4611 - 0.0450 + 0.05 = 0.4661$$

Aktivasyon:

$$a_y = \sigma(0.4661) = \frac{1}{1 + e^{-0.4661}} \approx \frac{1}{1 + 0.627} = 0.6144$$

Sonuçlar:

Nöron	z Değeri	Aktivasyon $a = \sigma(z)$
h_1	1.2	0.7685
h_2	-0.2	0.4502
y	0.4661	0.6144

Python'da Manuel Hesaplama: 3 Katmanlı MLP Örneği

Amaç: NumPy kullanarak bir giriş katmanı, bir gizli katman ve bir çıkış katmanından oluşan bir Çok Katmanlı Perceptron'un (MLP) ileri yayılımını (forward propagation) manuel olarak hesaplama

Adım 1: Problem Tanımı ve Katman Boyutları

- **Giriş Verisi:** 3 özellikli (feature) bir örnek:

```python
X = np.array([0.5, -1.2, 2.1])  # shape: (3,)
```

- **Katmanlar:**
  - Giriş katmanı: 3 nöron (X'in boyutu).
  - Gizli katman: 4 nöron (`hidden_size = 4`).
  - Çıkış katmanı: 1 nöron (binary classification).

Adım 2: Ağırlık ve Bias Matrislerinin Oluşturulması

- **Ağırlıklar:** Rastgele başlatma (gerçek uygulamada Xavier/Glorot init kullanılır).

python

```python
import numpy as np

# Giriş -> Gizli Katman
W1 = np.random.randn(4, 3)  # shape: [hidden_size, input_size]
b1 = np.zeros(4)            # bias, shape: (hidden_size,)

# Gizli -> Çıkış Katmanı
W2 = np.random.randn(1, 4)  # shape: [output_size, hidden_size]
b2 = np.zeros(1)            # bias, shape: (output_size,)
```

Adım 3: Aktivasyon Fonksiyonları

- **ReLU** (Gizli katmanda):

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

- **Sigmoid** (Çıkış katmanında, binary classification için):

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

Adım 4: İleri Yayılım (Forward Propagation)

1. **Giriş → Gizli Katman:**

```
z1 = np.dot(W1, X) + b1  # (4,3) dot (3,) -> (4,)
a1 = relu(z1)             # Aktivasyon
```

   - `z1` örneği: `[-0.3, 1.1, 0.5, -2.0]` → `a1 = [0, 1.1, 0.5, 0]` (ReLU sonrası).

2. **Gizli → Çıkış Katmanı:**

```
z2 = np.dot(W2, a1) + b2  # (1,4) dot (4,) -> (1,)
output = sigmoid(z2)       # Çıkış aktivasyonu
```

   - `output` örneği: `0.73` (1 sınıfına ait olasılık).

Adım 5: Tüm Kod Özeti

python

```python
import numpy as np

# Aktivasyon fonksiyonları
def relu(x):
    return np.maximum(0, x)

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

# Giriş verisi
X = np.array([0.5, -1.2, 2.1])

# Ağırlık ve bias'lar
W1 = np.random.randn(4, 3)  # Gizli katman ağırlıkları
b1 = np.zeros(4)
W2 = np.random.randn(1, 4)  # Çıkış katmanı ağırlıkları
b2 = np.zeros(1)

# İleri yayılım
z1 = np.dot(W1, X) + b1
a1 = relu(z1)
z2 = np.dot(W2, a1) + b2
output = sigmoid(z2)

print("Çıkış:", output)  # Örnek: 0.73
```

Kontrol Listesi ve Hata Ayıklama

- **Boyut Uyumsuzlukları:**
  - `W1.dot(X)` için `X.shape` = (3,) ve `W1.shape` = (4,3) olmalı.
  - `W2.dot(a1)` için `a1.shape` = (4,) ve `W2.shape` = (1,4) olmalı.
- **Aktivasyon Seçimi**:
  - Çıkış katmanında **sigmoid** (binary) veya **softmax** (multi-class).

Uygulama: NumPy ile 2 Gizli Katmanlı MLP

Amaç: 2 gizli katmana sahip bir MLP'nin ileri yayılımını (forward propagation) NumPy ile implemente edeceğiz. Binary classification için sigmoid çıkış kullanacağız.

Adım 1: Giriş Verisi ve Katman Boyutları

- **Giriş Verisi:** 4 özellikli (feature) bir örnek:

```python
X = np.array([1.0, -0.5, 2.3, 0.1])  # shape: (4,)
```

- **Katman Boyutları:**
  - Giriş: 4 nöron
  - **1. Gizli Katman:** 5 nöron
  - **2. Gizli Katman:** 3 nöron
  - Çıkış: 1 nöron

Adım 2: Ağırlık ve Bias Matrislerini Başlatma

python

```python
import numpy as np

# 1. Gizli Katman (Giriş -> Hidden1)
W1 = np.random.randn(5, 4)  # shape: [hidden1_size, input_size]
b1 = np.zeros(5)

# 2. Gizli Katman (Hidden1 -> Hidden2)
W2 = np.random.randn(3, 5)  # shape: [hidden2_size, hidden1_size]
b2 = np.zeros(3)

# Çıkış Katmanı (Hidden2 -> Çıkış)
W3 = np.random.randn(1, 3)  # shape: [output_size, hidden2_size]
b3 = np.zeros(1)
```

Adım 3: Aktivasyon Fonksiyonları

python

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

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

Adım 4: İleri Yayılım Fonksiyonu

python

```python
def forward(X, W1, b1, W2, b2, W3, b3):
    # 1. Gizli Katman
    z1 = np.dot(X, W1.T) + b1  # (4,) dot (5,4).T -> (5,)
    a1 = relu(z1)

    # 2. Gizli Katman
    z2 = np.dot(a1, W2.T) + b2  # (5,) dot (3,5).T -> (3,)
    a2 = relu(z2)

    # Çıkış Katmanı
    z3 = np.dot(a2, W3.T) + b3  # (3,) dot (1,3).T -> (1,)
    output = sigmoid(z3)
    return output
```

Adım 5: Örnek Çalıştırma

python

```python
# Rastgele ağırlıklarla test
np.random.seed(42)  # Reprodüksibilite için
output = forward(X, W1, b1, W2, b2, W3, b3)
print("Çıkış Olasılığı:", output)  # Örnek: 0.876
```

Adım 6: Boyut Kontrolü ve Hata Ayıklama

- **Hata:** `ValueError: shapes (x,) and (y,z) not aligned` → **Dot product uyumsuzluğu.**
  - Çözüm: `W.T` ile transpoz almayı unutmayın (çünkü `X` ve `a1`, `a2` vektörleri **row vector** olarak kabul edilir).
- **Örnek Boyutlar:**
  - `W1.shape = (5,4)`, `X.shape = (4,)` → `z1.shape = (5,)`.
  - `W2.shape = (3,5)`, `a1.shape = (5,)` → `z2.shape = (3,)`.

Genişletme Önerileri

1. **Batch İşleme:**
   - `X`'i (N,4) boyutunda bir batch olarak almak:

```python
X_batch = np.array([[1.0, -0.5, 2.3, 0.1], [0.2, -1.0, 1.1, 0.9]])
```

     - `forward` fonksiyonunda `np.dot` yerine `np.matmul` kullanın.
2. **Softmax Çıkış:**
   - Multi-class için sigmoid yerine softmax ekleyin:

```python
def softmax(x):
    exps = np.exp(x - np.max(x))  # Numerik stabilite
    return exps / np.sum(exps)
```

3. **Backpropagation:**
   - Loss fonksiyonu (örn. BCE) ve gradyan hesaplama ekleyerek eğitim yapın.

Örnek Çıktı :

Çıkış Olasılığı: 0.876

1 sınıfına ait tahmin

Sonuç ve Genişletme :

Amaç: MLP implementasyonunda karşılaşılabilecek potansiyel hataları ve çözümlerini özetlemek, modeli geliştirmek için genişletme önerileri sunmak.

Potansiyel Hatalar & Kontrol Listesi:

1. Boyut Uyumsuzlukları (Shape Errors)

- **Belirtiler:**
  - `ValueError: shapes (x,) and (y,z) not aligned`
  - `np.dot` veya `np.matmul` hataları.
- **Nedenler:**
  - Ağırlık matrislerinin transpozunun unutulması (`W.T`).
  - Giriş verisinin boyutunun (N, features) yerine (features,) olması.

Çözümler:

- **Debug Adımları:**

```
print("W1 shape:", W1.shape)  # (hidden1_size, input_size)
print("X shape:", X.shape)    # (input_size,) veya (batch_size, input_size)
```

- **Düzeltmeler:**
  - Batch işleme için `X`'i `(1, input_size)` şeklinde genişletin:

```
X = np.expand_dims(X, axis=0)  # (4,) -> (1, 4)
```

  - `np.dot(X, W1.T)` kullanarak boyut uyumunu sağlayın.

2. Gradyan Problemleri (Vanishing/Exploding Gradients)

- **Belirtiler:**
  - Model eğitiminde loss **sabit kalıyor** (vanishing).
  - Loss **NaN** değeri veriyor (exploding).
- **Nedenler:**
  - **Vanishing:** Aktivasyon fonksiyonu olarak sigmoid/tanh kullanımı (türevleri küçük).
  - **Exploding:** Ağırlıkların rastgele başlatılmasında büyük değerler (örn. `randn` ile aşırı büyük/std sapma).

Çözümler:

- **Ağırlık İnitializasyonu:**

```
# Xavier/Glorot Initialization (sigmoid/tanh için)
W1 = np.random.randn(5, 4) * np.sqrt(1/4)  # input_size=4
# He Initialization (ReLU için)
W1 = np.random.randn(5, 4) * np.sqrt(2/4)
```

- **Aktivasyon Fonksiyonu:**
  - Gizli katmanlarda **ReLU** veya **LeakyReLU** kullanın.
  - Çıkış katmanında probleme göre **sigmoid** (binary) veya **softmax** (multi-class).

Gradyan Kırpma (Gradient Clipping):

```
max_grad_norm = 1.0
grads = [dW1, db1, dW2, db2, ...]
for grad in grads:
    grad = np.clip(grad, -max_grad_norm, max_grad_norm)
```

Genişletme Önerileri :

1. Batch İşleme Desteği Ekleme

- **Problem:** Şu anki kod tek örnek (`X.shape = (4,)`) ile çalışıyor.
- **Çözüm:**

```
# Batch input (N örnek)
X_batch = np.array([[1.0, -0.5, 2.3, 0.1],
                    [0.2, -1.0, 1.1, 0.9]])  # shape: (2, 4)

# Forward fonksiyonunu güncelle:
def forward(X, W1, b1, W2, b2, W3, b3):
    z1 = np.dot(X, W1.T) + b1  # (2,4) dot (4,5) -> (2,5)
    a1 = relu(z1)
    z2 = np.dot(a1, W2.T) + b2 # (2,5) dot (5,3) -> (2,3)
    a2 = relu(z2)
    z3 = np.dot(a2, W3.T) + b3 # (2,3) dot (3,1) -> (2,1)
    return sigmoid(z3)
```

2. Backpropagation ve Eğitim Döngüsü

- **Loss Fonksiyonu (Binary Cross-Entropy):**

```
def binary_cross_entropy(y_true, y_pred):
    return -np.mean(y_true * np.log(y_pred) + (1-y_true) * np.log(1-y_pred))
```

- **Gradyan Hesaplama:**
  - Chain rule ile türevleri elle hesaplayın veya otomatik diff (Autograd/JAX) kullanın.

3. Hiperparametre Optimizasyonu

- **Önerilen Denemeler:**
  - Gizli katman nöron sayıları: `[32, 64, 128]`
  - Aktivasyonlar: `ReLU` vs `LeakyReLU`
  - Optimizasyon: SGD, Adam

Hiperparametre Nedir?

Bir modeli eğitirken iki tür ayar vardır:

1. **Model Parametreleri:** Modelin eğitim sırasında veriden **kendi kendine öğrendiği** değerlerdir. Bir yapay sinir ağının katmanlarındaki ağırlıklar ve sapmalar (weights and biases) buna örnektir. Biz bunları doğrudan belirleyemeyiz.
2. **Hiperparametreler:** Modeli eğitmeye başlamadan **bizim belirlediğimiz** ayar değerleridir. Bunlar, modelin yapısını ve öğrenme sürecini kontrol eder. Örneğin:
    - **Öğrenme Oranı (Learning Rate):** Modelin ağırlıklarını her adımda ne kadar değiştireceğini belirler.
    - **Katman Sayısı:** Yapay sinir ağındaki gizli katmanların (hidden layers) sayısı.
    - **Nöron Sayısı:** Her bir katmandaki nöron sayısı.
    - **Dropout Oranı:** Eğitim sırasında rastgele devre dışı bırakılacak nöronların oranı.
    - **Batch Boyutu:** Bir eğitim adımında kullanılan veri örneklerinin sayısı.

Neden Hiperparametre Optimizasyonu Yapılır?

Bir modelin performansı, doğru hiperparametrelerin seçilmesine büyük ölçüde bağlıdır. Yanlış bir öğrenme oranı, modelin ya çok yavaş öğrenmesine ya da hiç öğrenememesine neden olabilir. Benzer şekilde, aşırı düşük bir dropout oranı aşırı öğrenmeye yol açabilirken, çok yüksek bir dropout oranı modelin yeterince öğrenmesini engelleyebilir.

Hiperparametre optimizasyonu, deneme yanılma yöntemini ortadan kaldırarak en iyi performansı veren hiperparametre kombinasyonunu **sistematik bir şekilde** bulmamızı sağlar. Bu, manuel olarak saatlerce veya günlerce farklı değerleri denemekten çok daha verimlidir.

En Yaygın Optimizasyon Yöntemleri

Hiperparametre optimizasyonu için en çok kullanılan üç yöntem şunlardır:

1. Grid Search (Izgara Arama)

Bu yöntem, en basit ve en kolay anlaşılır olanıdır. Belirlediğiniz her hiperparametre için olası değerlerin bir listesini oluşturursunuz. Grid Search, bu listelerdeki **tüm olası kombinasyonları tek tek dener** ve her kombinasyonla modeli eğitir. Sonunda en iyi doğrulama (validation) performansını veren kombinasyonu seçer.

- **Avantajı:** Belirlenen aralıklar içinde en iyi kombinasyonu bulma garantisi verir.
- **Dezavantajı:** Hiperparametre sayısı ve denenecek değerler arttıkça, denenecek kombinasyon sayısı geometrik olarak artar. Bu durum, işlem süresini çok uzatabilir.

2. Random Search (Rastgele Arama)

Grid Search'e göre daha verimli bir yöntemdir. Tüm kombinasyonları denemek yerine, belirlenen aralıklardan **rastgele hiperparametre kombinasyonları** seçer ve dener.

- **Avantajı:** Çoğu durumda, daha kısa sürede Grid Search'e yakın veya ondan daha iyi sonuçlar bulabilir. Özellikle hangi hiperparametrenin daha önemli olduğunu bilmediğimiz durumlarda çok etkilidir.
- **Dezavantajı:** En iyi kombinasyonu bulma garantisi yoktur.

3. Bayesian Optimization (Bayesçi Optimizasyon)

Bu, en gelişmiş ve en verimli yöntemlerden biridir. Random Search'in aksine, Bayesçi Optimizasyon denediği her kombinasyonun sonuçlarını **öğrenir** ve bu bilgiyi bir sonraki deneme için daha akıllıca bir seçim yapmakta kullanır. Geçmiş performans verilerini kullanarak, hangi hiperparametre kombinasyonlarının iyi sonuç verme olasılığının daha yüksek olduğuna dair bir model oluşturur.

- **Avantajı:** Diğer yöntemlere göre çok daha az deneme ile en iyi sonuçlara ulaşabilir. Zaman ve kaynak açısından en verimli yöntemdir.
- **Dezavantajı:** Uygulaması diğer yöntemlere göre daha karmaşıktır.

Dropout ve Hiperparametre Optimizasyonu

Dropout, bir hiperparametredir. Bizim belirlediğimiz bir ayardır ve en iyi değeri bulmak için optimize edilmesi gerekir.

- **Optimizasyon Süreci:** Hiperparametre optimizasyonu yaparken, genellikle denenecek **dropout oranları** listesi belirleriz. Örneğin, `[0.1, 0.2, 0.3, 0.4]` gibi bir liste oluşturabiliriz.
- **Nasıl Çalışır:** Optimizasyon algoritması (Grid Search, Random Search vb.) bu oranları tek tek veya rastgele dener. Her denemede, model farklı bir dropout oranıyla eğitilir ve doğrulama (validation) verisindeki performansı ölçülür.
- **Sonuç:** En iyi doğrulama performansını veren dropout oranı, model için en uygun değer olarak seçilir. Örneğin, %20'lik bir dropout oranı aşırı öğrenmeyi en iyi şekilde engelliyorsa, bu değer nihai modelimizde kullanılır.

Erken Durdurma ve Hiperparametre Optimizasyonu

Erken durdurma ise bir hiperparametre olmaktan ziyade, hiperparametre optimizasyonunu **kolaylaştıran** bir araçtır. Özellikle modelin **kaç epok eğitilmesi gerektiğini** belirleme konusunda çok yardımcı olur.

- **Patience (Sabır) Ayarı:** Erken durdurma içinde `patience` (sabır) adı verilen bir hiperparametre bulunur. Bu ayar, doğrulama kaybında (validation loss) iyileşme olmadığı halde eğitimin kaç epok daha devam edeceğini belirtir. Bu değeri de optimize edebilirsin.
- **Eğitimi Kısaltma:** Erken durdurma, deneme yanılma sürecini hızlandırır. Hiperparametre optimizasyonu sırasında her bir kombinasyonu 100 epok yerine, sadece en iyi performansa ulaştığı noktaya kadar eğitir. Bu, optimizasyonun tamamlanma süresini ciddi şekilde kısaltır.
- **Daha İyi Kararlar:** Erken durdurma sayesinde, hiperparametre optimizasyon algoritması, bir modelin gerçek potansiyelini daha doğru değerlendirebilir. Çünkü algoritma, modelin aşırı öğrenmeye başlamadan önceki en iyi performansını görür.

Özet

Dropout, bir hiperparametredir ve değeri optimize edilir. **Erken durdurma** ise hem kendi içinde ayarlanabilen (patience) bir hiperparametreye sahiptir hem de hiperparametre optimizasyonunun daha verimli ve doğru sonuçlar vermesini sağlayan bir **mekanizmadır**.

Yani, ikisi de modelin genelleme yeteneğini artırmak için kritik öneme sahiptir ve hiperparametre optimizasyonunun ayrılmaz bir parçasıdır. Biri "ne kadar" düzenleme yapılacağını (dropout oranı), diğeri ise "ne zaman" eğitimin duracağını (erken durdurma) belirler.