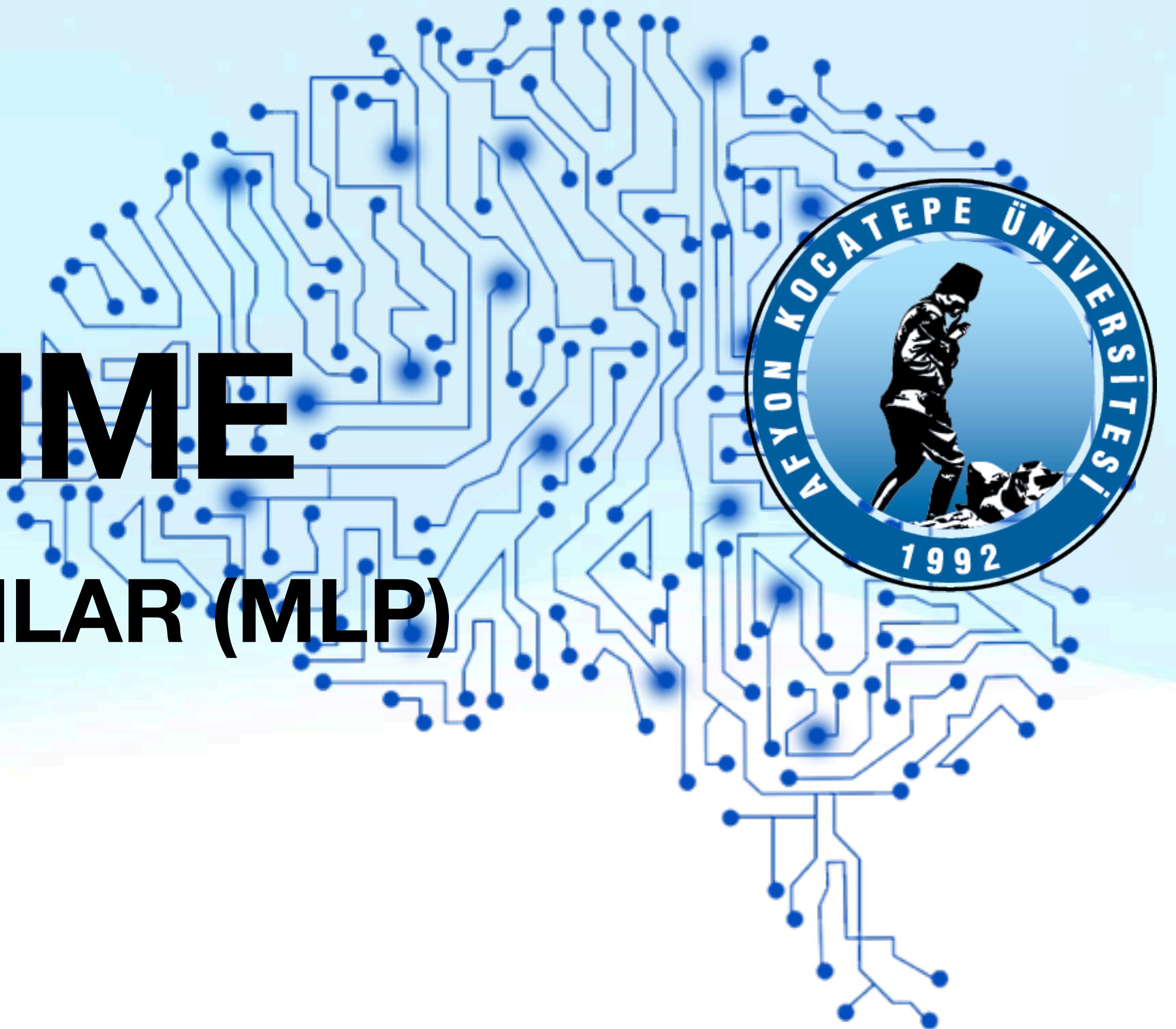


DERİN ÖĞRENME

ÇOK KATMANLI ALGILAYICILAR (MLP)



HAZIRLAYANLAR:

ZEYNEP SENA KOÇ - 212923057
BERKE BÜYÜKKÖPRÜ - 212923060
ECESU TOPÇU - 212923021
EFE AVCI - 212923043

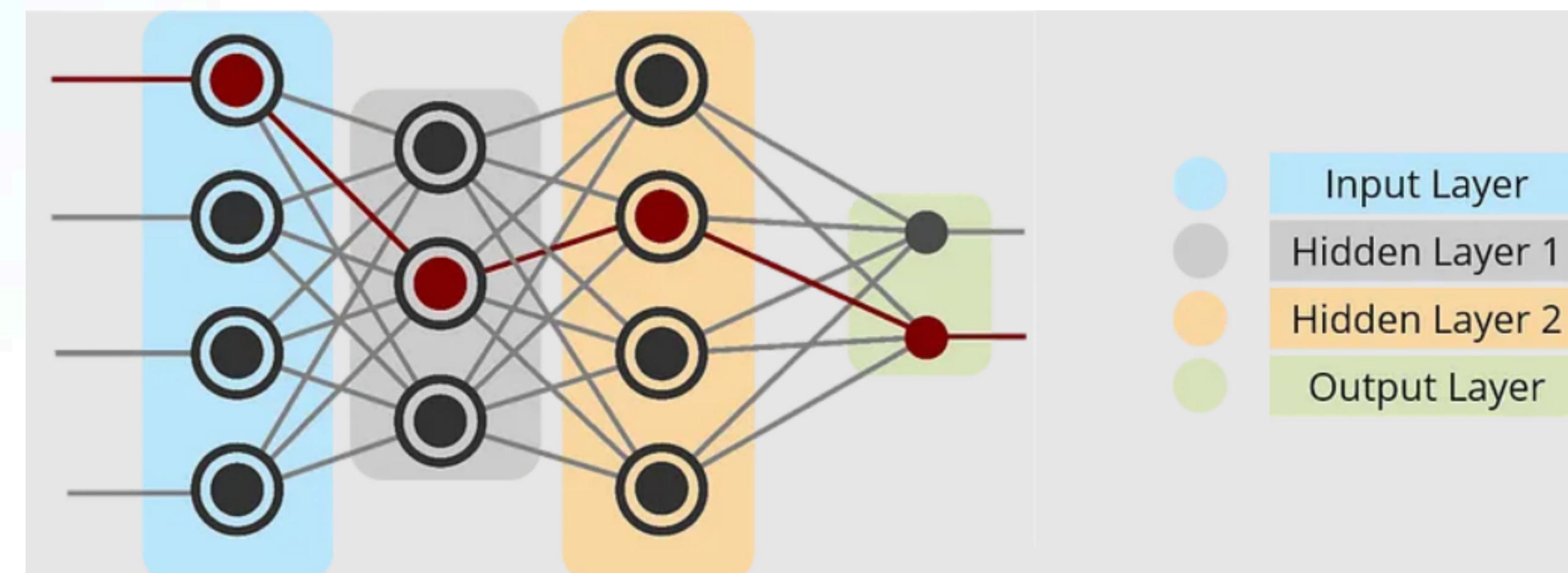
AKADEMİSYEN:

DR. ÖĞR. ÜYESİ KEREM GENCER

ÇOK KATMANLI ALGILAYICILAR (MLP)

MLP Nedir?

- MLP, yani Çok Katmanlı Algılayıcı (**Multilayer 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.



ÇOK KATMANLI ALGILAYICILAR (MLP)

MLP'nin Temel Özellikleri

Katmanlı Yapı:

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

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

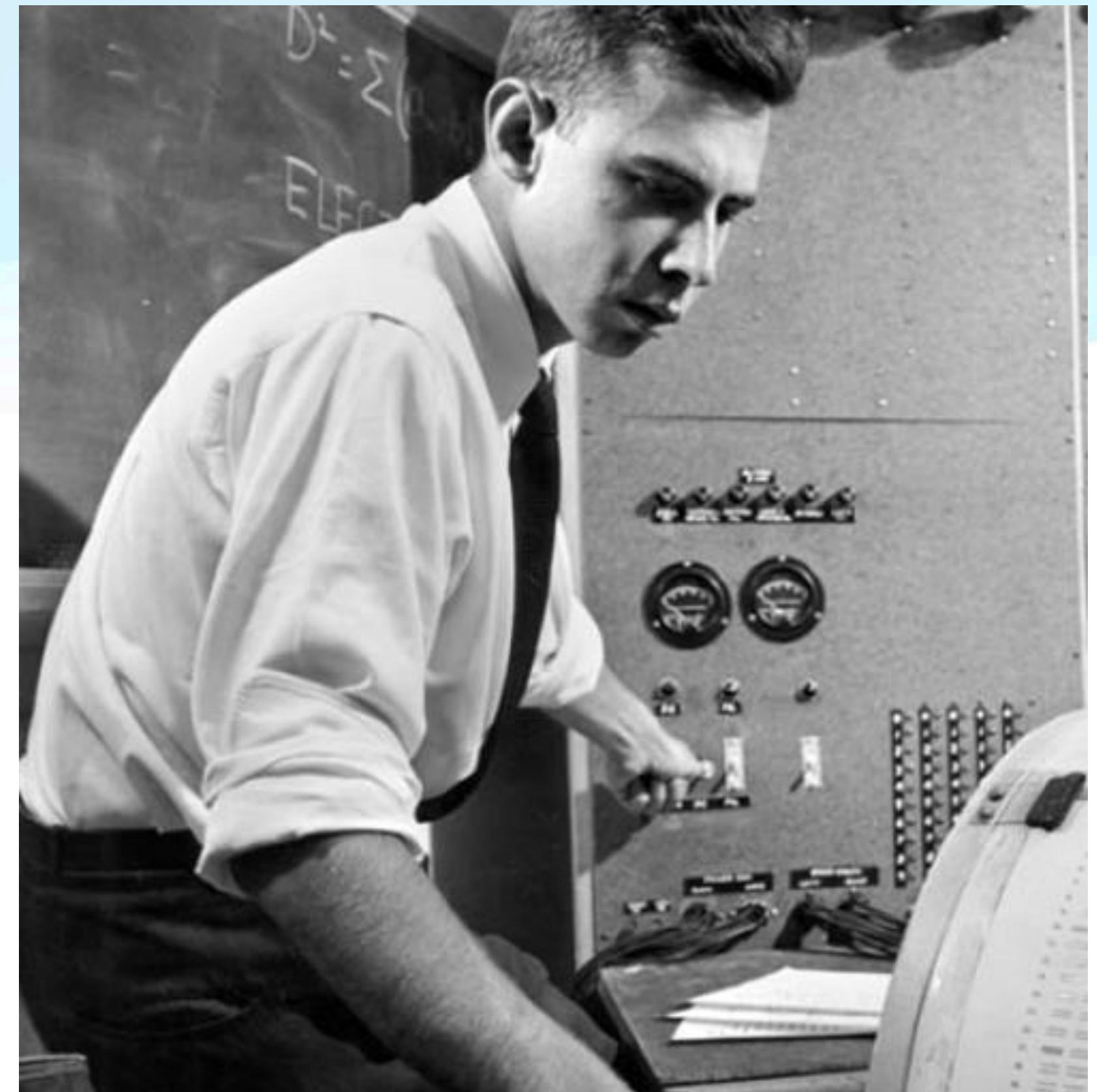
Aktivasyon Fonksiyonları: Her nöronun çıktısı, genellikle bir aktivasyon fonksiyonu (**sigmoid, ReLU...**) tarafından işlenir. Bu fonksiyonlar, nöronların lineer olmayan karmaşık örüntülerini öğrenmesini sağlar.

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

ÇOK KATMANLI ALGILAYICILAR (MLP)

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. Algılayıcı, en basit sinir ağı modellerinden biridir ve insan beyninin bilgi işlemini taklit ederek makine öğrenmesine temel oluşturan bir yapıdır.
- Bu modelin gelişimi, arka yayılım (**backpropagation**) algoritmasının 1986'da **Rumelhart, Hinton** ve **Williams** tarafından tanıtılmasıyla ivme kazanmıştır.
- Arka yayılım (**backpropagation**) algoritması, yapay sinir ağlarında hataları azaltmak için kullanılan bir öğrenme algoritmasıdır. Amaç, ağın çıktısı ile hedef değer arasındaki hatayı minimize edecek şekilde ağıdaki ağırlıkları güncellemektir.
- Çok katmanlı sinir ağlarının eğitimi için temel bir algoritma olup, ağırlıkların optimizasyonunu sağlar ve öğrenme sürecini mümkün hale getirir.



ÇOK KATMANLI ALGILAYICILAR (MLP)

Kullanım Alanları, Avantajları ve Dezavantajları

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 teşhisleri, 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 uyum sağlayabilme.
- Derin Öğrenme:** Çok katmanlı yapısı sayesinde karmaşık örüntüler ve ilişkileri öğrenebilme.

Dezavantajları

- Aşırı Uyma (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.

ÇOK KATMANLI ALGILAYICILAR (MLP)

Doğrusal Modeller ve Sınırlamaları

- Doğrusal modeller, girdiler ve çıktılar arasında sadece düz, doğrusal ilişkiler kurar. Bu, bir girdi değiştiğinde her zaman aynı oranda ve doğrultuda çıktı değişeceği anlamına gelir.
- **Örneğin**, bir kişinin gelirini ele alalım. Gelir artarsa, kredi ödeme olasılığının da doğrusal olarak artacağını düşünebiliriz. Ancak çok düşük gelirlerde gelir artışı kredi ödeme olasılığını büyük oranda etkilerken, zaten yüksek gelirlerdeki küçük artışlar daha az fark yaratır. Yani gerçek hayatı bir çok ilişki doğrusal değildir.
- Benzer şekilde, doğrusal modeller “monotonik” bir ilişkiyi varsayar, yani bir özellik arttığında çıktı hep artmalı ya da azalmalı. Bu, bazı durumlarda işe yarar. **Örneğin**, sıcaklık arttığında ateşten kaynaklı riskin artması gibi. Ancak, bazı özelliklerdeki değişimler monotonik olmayabilir, yani ilişki her zaman tek yönlü olmayıabilir.

ÇOK KATMANLI ALGILAYICILAR (MLP)

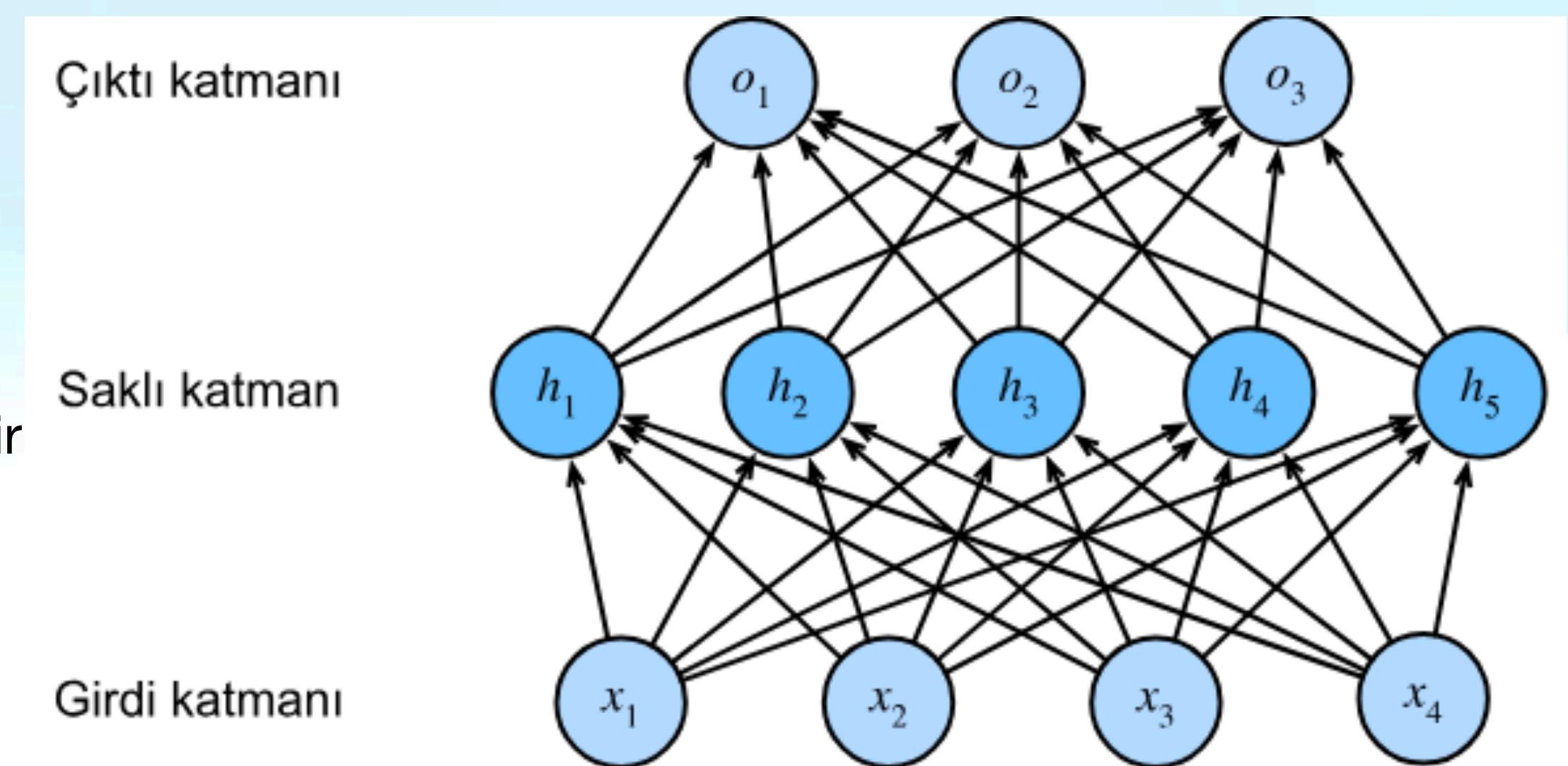
Doğrusal Modeller ve Sınırlamaları

- Doğrusal olmayan ilişkiler oldukça yaygındır. **Örneğin**, insan vücudu 37°C 'nin altına düştüğünde soğuk sebebiyle hastalık riski artar, 37°C 'nin üstünde ise ateş sebebiyle artar. Bu gibi karmaşık durumlarda, doğrusal bir model, girdiler ve çıktı arasındaki ilişkiyi tam olarak yansıtamaz.
- Sıcaklıkörneğinde olduğu gibi, 37°C 'den uzaklığın hastalık riski ile ilişkili olduğunu söylemek daha doğru olabilir. Bu gibi durumlarda doğrusal modeller işe yaramaz. Bu noktada devreye gizli katmanlar girer.
- **Gizli katmanlar**, bir sinir ağı içinde, girdi ile çıktı arasındaki katmanlardır ve karmaşık, doğrusal olmayan ilişkileri modellememize olanak tanır. Gizli katmanlar sayesinde model, verinin derin yapısını keşfeder ve öznitelikler arasındaki etkileşimleri öğrenir.
- Bu katmanlar, doğrusal olmayan ilişkileri anlamaya ve modellemeye yardımcı olur. Böylece sinir ağı, doğrusal modellerin tek başına çözmekte zorlanacağı problemleri çözebilir.

ÇOK KATMANLI ALGILAYICILAR (MLP)

Gizli Katmanları Birleştirme

- Gizli katmanları, girdi ve çıktı katmanları arasına yerleştirilen ek katmanlar olarak düşünebiliriz. Bu katmanlar, veriyi daha karmaşık bir yapıya dönüştürerek, doğrusal olmayan ilişkileri modellememize yardımcı olur.
- Birden fazla tam bağlı (**fully connected**) katmanı üst üste ekleyerek gizli katmanlar içeren bir yapıyı oluşturabiliriz.
- Bu yapıya, **Çok Katmanlı Algılayıcı (MLP)** denir. MLP'ler, girdi katmanını, bir veya birden fazla gizli katmanı ve çıktı katmanını içerir. MLP'nin çalışma prensibi şudur:
 - Girdi katmanındaki her bir nöron, bir sonraki katmandaki tüm nöronlara bağlıdır. Yani her girdi, her gizli nöronu etkiler.
 - Gizli katmandaki her bir nöron, bir sonraki katmandaki (örneğin çıktı katmanındaki) her nörona bağlıdır.
- Bu bağlar sayesinde, her nöron önceki katmandan gelen tüm bilgiyi kullanarak kendi çıktısını hesaplar ve bu bilgi akışı nihai olarak çıktı katmanına ulaşır. Gizli katman veriyi işler ve onu doğrusal olmayan bir şekilde dönüştürür. Sonrasında, çıktı katmanı veriyi alır ve nihai sonuçları oluşturur. Böylece, MLP sadece doğrusal değil, doğrusal olmayan ilişkileri de modelleyebilir.



ÇOK KATMANLI ALGILAYICILAR (MLP)

Evrensel Yaklaşımalar

- MLP'ler, girdilerin her birinin değerine bağlı olan gizli nöronları aracılığıyla girdilerimiz arasındaki karmaşık etkileşimleri yakalayabilir. **Örneğin**, bir çift girdi üzerinde temel mantık işlemleri gibi rastgele hesaplamalar için kolayca gizli düğümler tasarlatabiliriz. Dahası, etkinleştirme fonksiyonunun belli seçimleri için MLP'lerin evrensel yaklaşımçılar olduğu yaygın olarak bilinmektedir.
- Yeterli düğüm (**muhtemelen saçma bir şekilde çok**) ve doğru ağırlık kümesi verilen tek gizli katmanlı bir ağla bile, aslında o işlevi öğrenmek zor olan kısım olsa da herhangi bir işlevi modelleyebiliriz. Sinir ağınızı biraz C programlama dili gibi düşünebilirsiniz. Dil, diğer herhangi bir modern dil gibi, herhangi bir hesaplanabilir programı ifade etme yeteneğine sahiptir. Ama aslında sizin şartnamenizi karşılayan bir program bulmak zor kısımdır.
- Dahası, tek gizli katmanlı bir ağın herhangi bir işlevi öğrenebilmesi, tüm problemlerinizi tek gizli katmanlı ağlarla çözmeye çalışmanız gerektiği anlamına gelmez. Aslında, daha derin (**daha geniş kıyasla**) ağları kullanarak birçok işlevi çok daha öz bir şekilde tahmin edebiliriz.

ÇOK KATMANLI ALGILAYICILAR (MLP)

Etkinleştirme Fonksiyonları

- Etkinleştirme fonksiyonları, bir nöronun aktif olup olmayacağı belirlemek için öncelikle girdi sinyallerinin ağırlıklı toplamını hesaplar ve ardından bu toplamı bir ek girdi ile toplar.
- Bu fonksiyonlar, girdi sinyallerini çıktılara dönüştürmek için kullanılır ve genellikle doğrusal olmayan bir yapı ekleyerek daha karmaşık ilişkileri öğrenmeyi sağlar.

ÇOK KATMANLI ALGILAYICILAR (MLP)

ReLU İşlevi

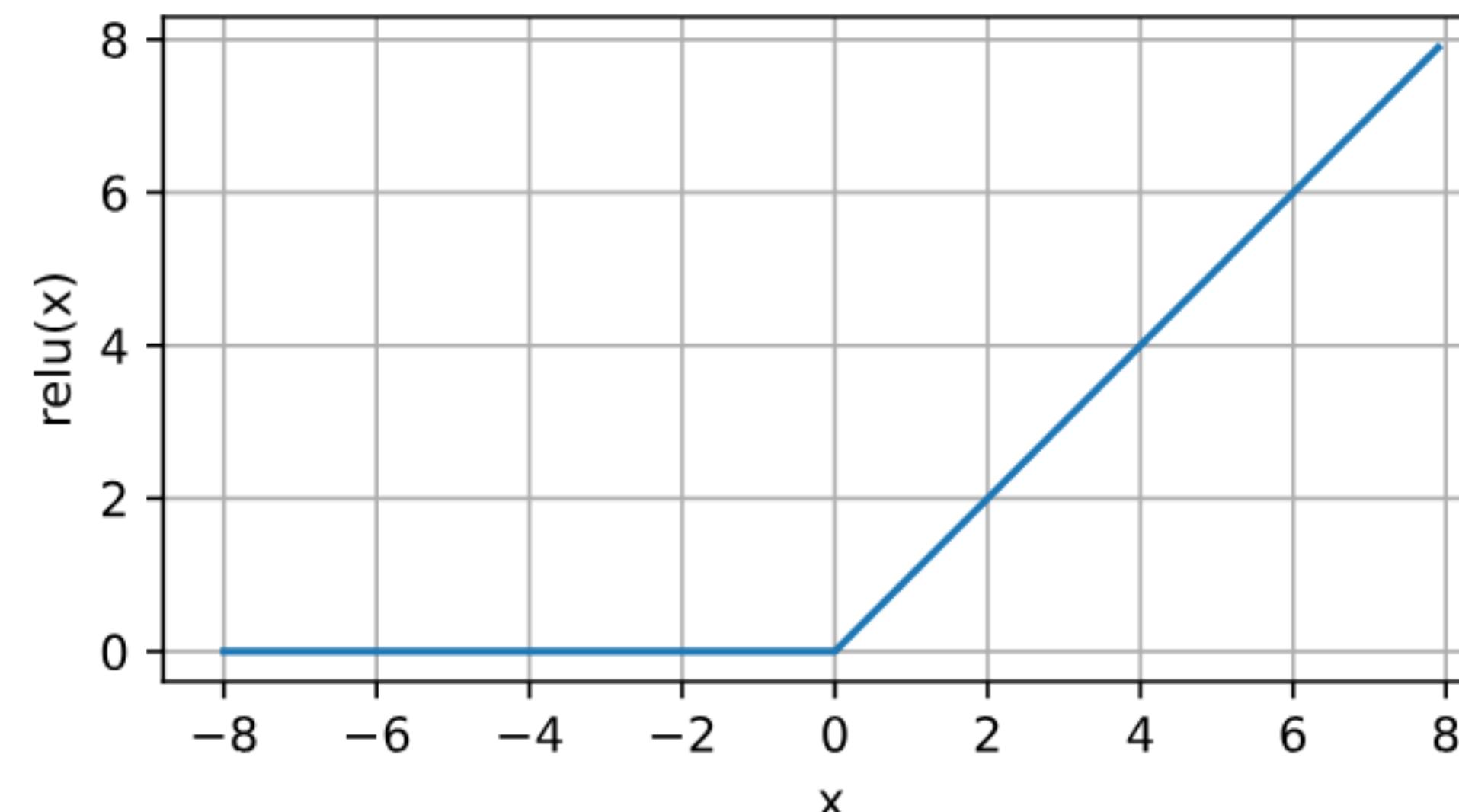
- Hem uygulamanın basitliği hem de çeşitli tahminci görevlerdeki iyi performansı nedeniyle en popüler seçenek, düzeltilmiş doğrusal birim yani ReLU' dur. ReLU, çok basit doğrusal olmayan bir dönüşüm sağlar. “ x ” ögesi verildiğinde, işlev o ögenin ve 0'ın maksimum değeri olarak tanımlanır. Kısaca, girdi negatifse 0, pozitifse kendi değerini alır.
- **ReLU**, sadece pozitif değerlerle ilgilendiği ve bu sayede eşit dağılmış bir veri setinde yarı yarıya daha az işlem yaptığı için diğer aktivasyon fonksiyonlarına göre daha hızlıdır. Bu yüzden sıkça kullanılır.

$$\text{ReLU}(x) = \max(x, 0).$$

ÇOK KATMANLI ALGILAYICILAR (MLP)

ReLU İşlevi

```
%matplotlib inline # Jupyter Notebook'ta grafikleri hücre altında göster.  
import tensorflow as tf # TensorFlow kütüphanesini içe aktar.  
from d2l import tensorflow as d2l # "d2l" kitabıın TensorFlow ile ilgili fonksiyonlarını içe aktar.  
  
x = tf.Variable(tf.range(-8.0, 8.0, 0.1), dtype=tf.float32) # bir dizi oluştur.  
y = tf.nn.relu(x) # x değişkenine ReLU aktivasyon fonksiyonunu uygular.  
# Negatif değerler 0 olur, pozitif değerler olduğu gibi kalır.  
d2l.plot(x.numpy(), y.numpy(), 'x', 'relu(x)', figsize=(5, 2.5)) # x ve y değerlerini kullanarak bir grafik çizer;
```

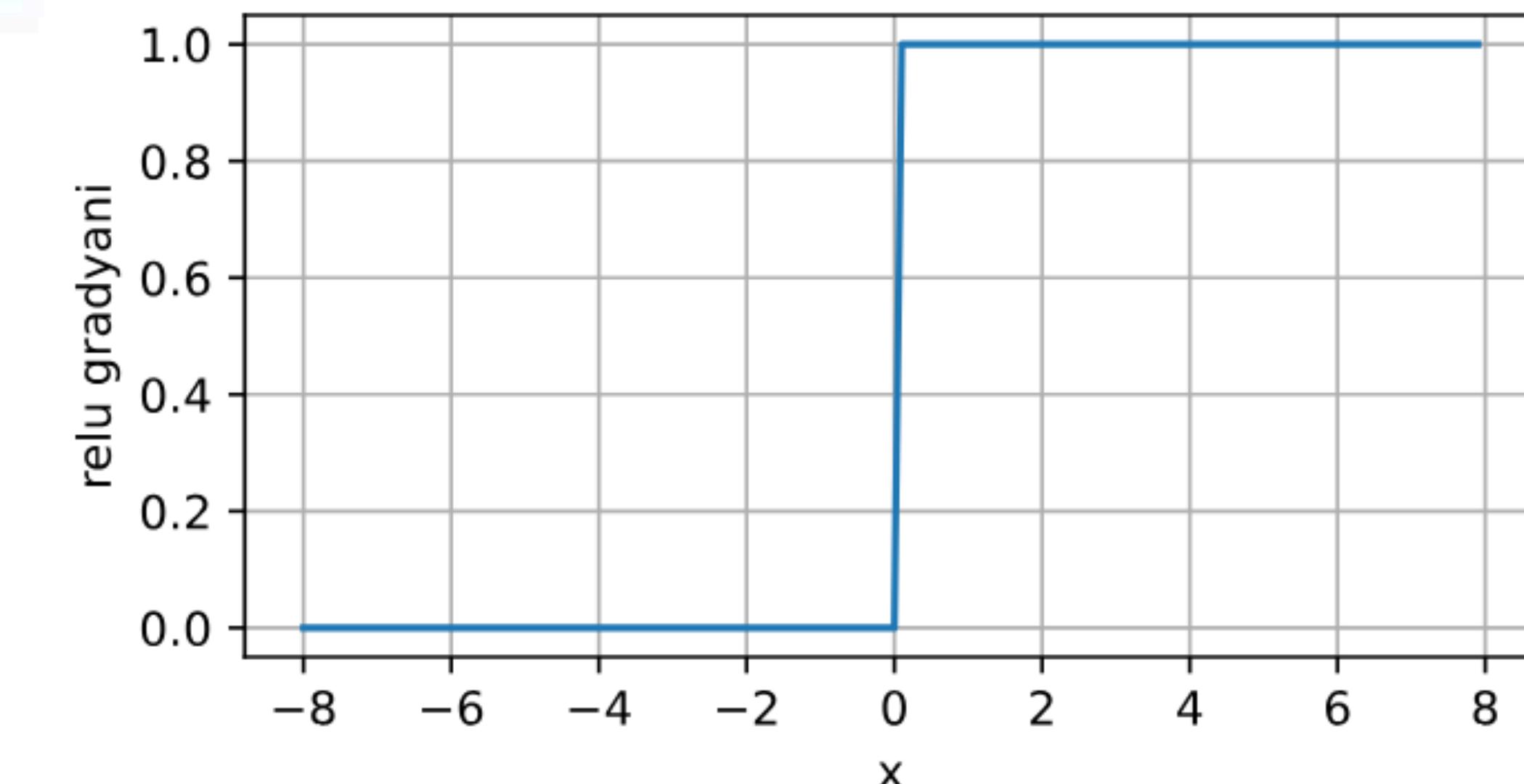


ÇOK KATMANLI ALGILAYICILAR (MLP)

ReLU İşlevi

- Girdi negatif olduğunda, ReLU fonksiyonunun türevi 0'dır ve girdi pozitif olduğunda ReLU fonksiyonunun türevi 1'dir.

```
with tf.GradientTape() as t: # Gradyan hesaplamak için bir ortam oluştur
    y = tf.nn.relu(x) # x değişkenine ReLU aktivasyon fonksiyonu uygulanır.
d2l.plot(x.numpy(), t.gradient(y, x).numpy(), 'x', 'relu gradyANI', figsize=(5, 2.5))
# ReLU'nun gradyanını x'e göre hesapla
```



- Girdi tam olarak 0'a eşit değer aldığında ReLU fonksiyonunu türevlenemez. Bu durumlarda, sol taraftaki türev varsayılan olarak kullanılır ve girdi 0 olduğunda türevin 0 olduğunu varsayılar.

ÇOK KATMANLI ALGILAYICILAR (MLP)

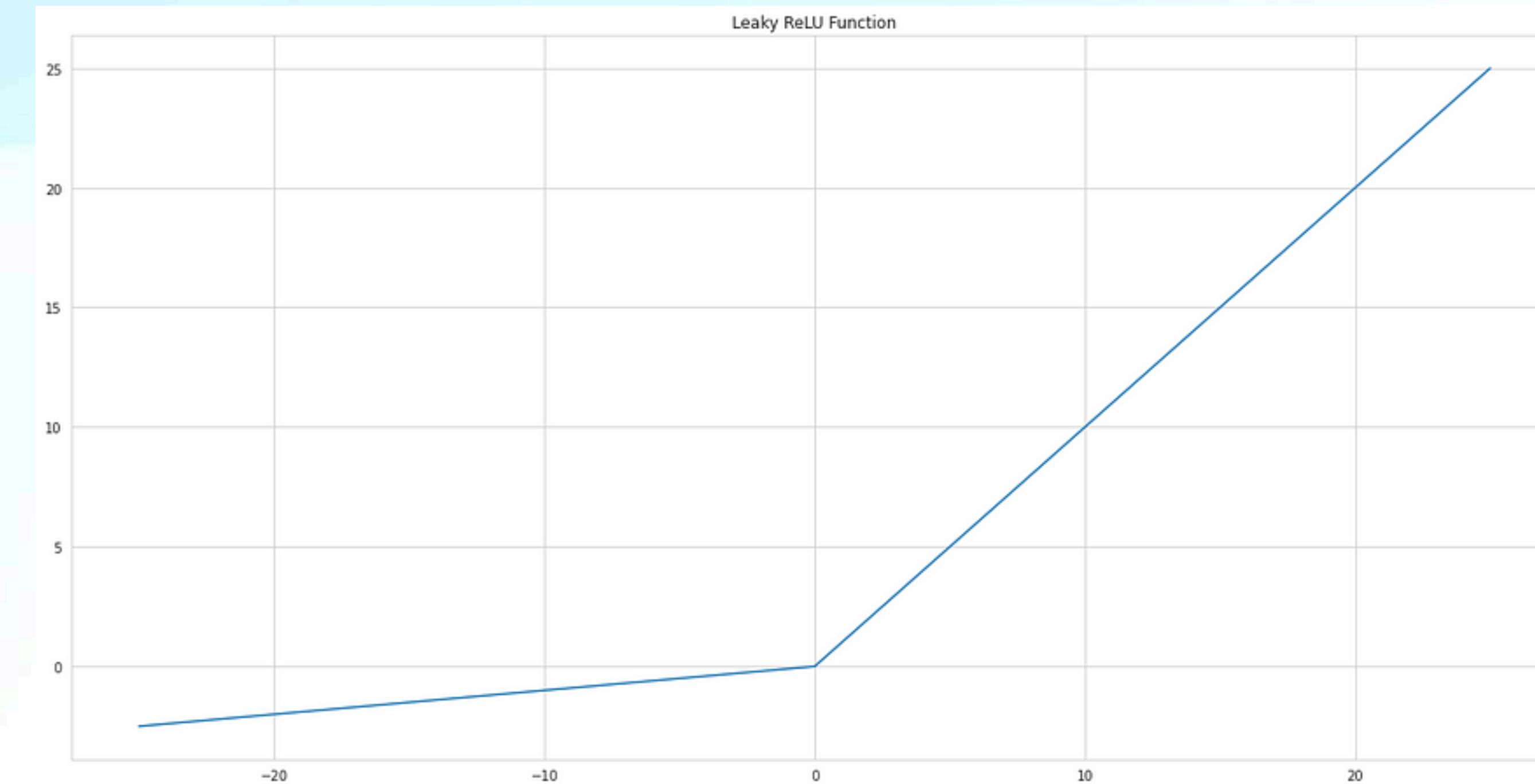
ReLU İşlevi

- Türev hesaplamak için o noktanın limiti olmalı ve 3 şartı sağlamalıdır. **Sol tarafının limitinin olması, sağ tarafının limitinin olması ve sağ ile sol limitinin birbirine eşit olması gerekmektedir.** Bu çizimde, 0 noktasının solundaki türev 0, sağındaki türev ise 1'dir. Bu da türevin 3. kuralı ihlal eder. Bu yüzden 0 noktasında türev hesaplanamaz.
- Burada ReLU kullanılmasının nedeni , türevlerinin özellikle makul davranıştır: Ya kaybolurlar ya da sadece argümanın geçmesine izin verirler. Bu, optimizasyonun daha iyi davranışmasını sağlar ve sinir ağlarının önceki sürümlerinin belalısı bilindik gradyanların kaybolması (**Vanishing Gradient**) sorununu hafifletir.
- Vanishing Gradient Problemi, kısaca Aktivasyon fonksiyonlarının verileri belirli bir aralığa sıkıştırması ve bu sıkıştırılan aralığın küçük olması hatta türevinin daha da küçük olmasıdır. Dolayısıyla belirli bir süre sonra dikkate alınmayacak kadar küçük optimizasyon parametrelerine ulaşması ve modelin tam olarak öğrenememesi sorunudur.
- Negatif değerlere 0 vermesi bazen bizim avantajımızayken bazen bir sorun oluşturur. Mesela metin üzerinde duygusal analizi yapılrken -1 etiketini karşılığının negatif olduğunu düşünelim. Bu durumda negatif etiketi silinmiş olacaktır. Bu sorunları aşmak için ReLU'nun birkaç varyasyonunu kullanabiliriz.

ÇOK KATMANLI ALGILAYICILAR (MLP)

Leaky ReLU

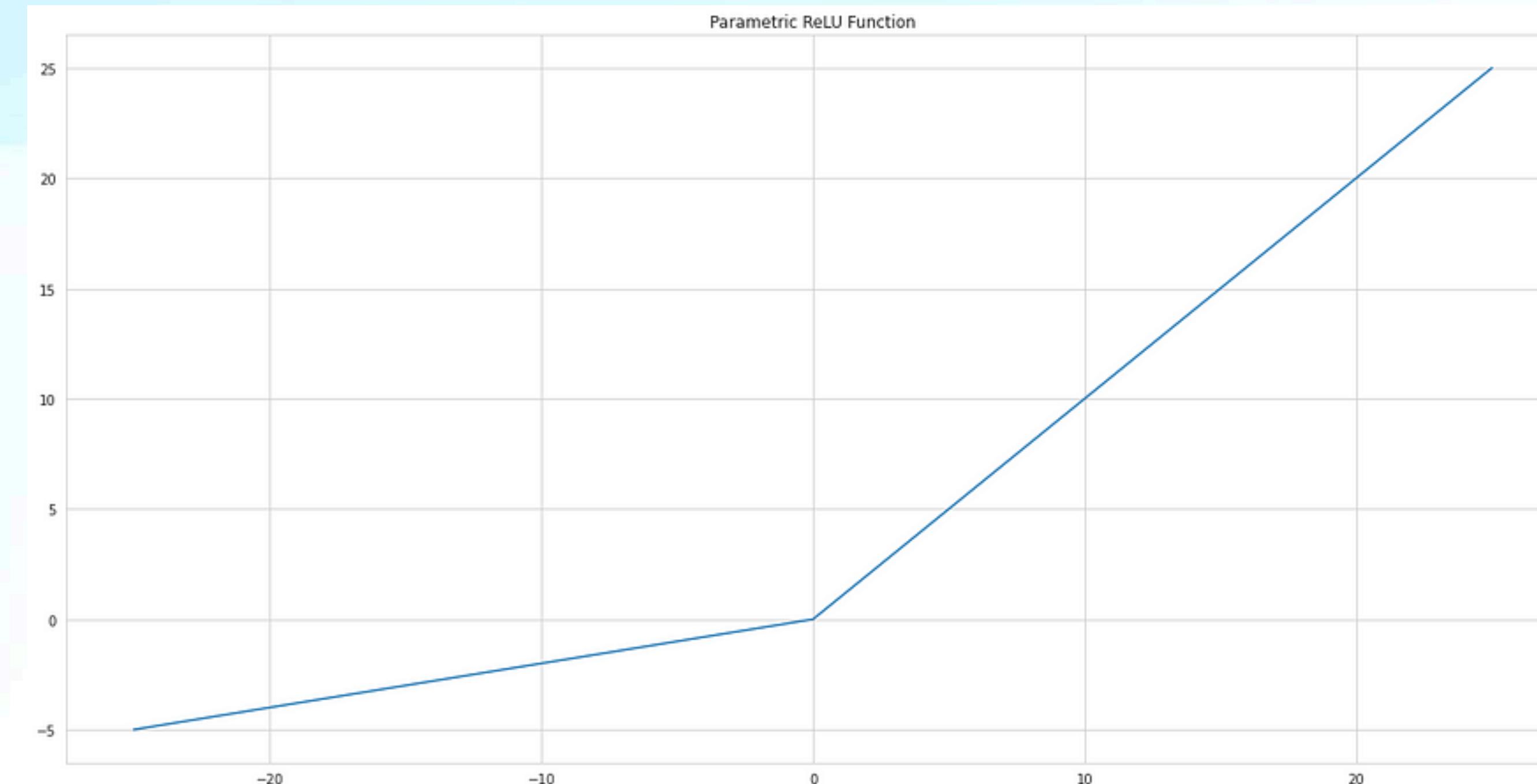
- ReLU'daki gibi pozitif ve 0 değerine dokunmaz ama negatif değerleri 0.01 ile çarparak konumlandırır. Kısmen negatifliği korumaya çalışır.



ÇOK KATMANLI ALGILAYICILAR (MLP)

Parametric ReLU

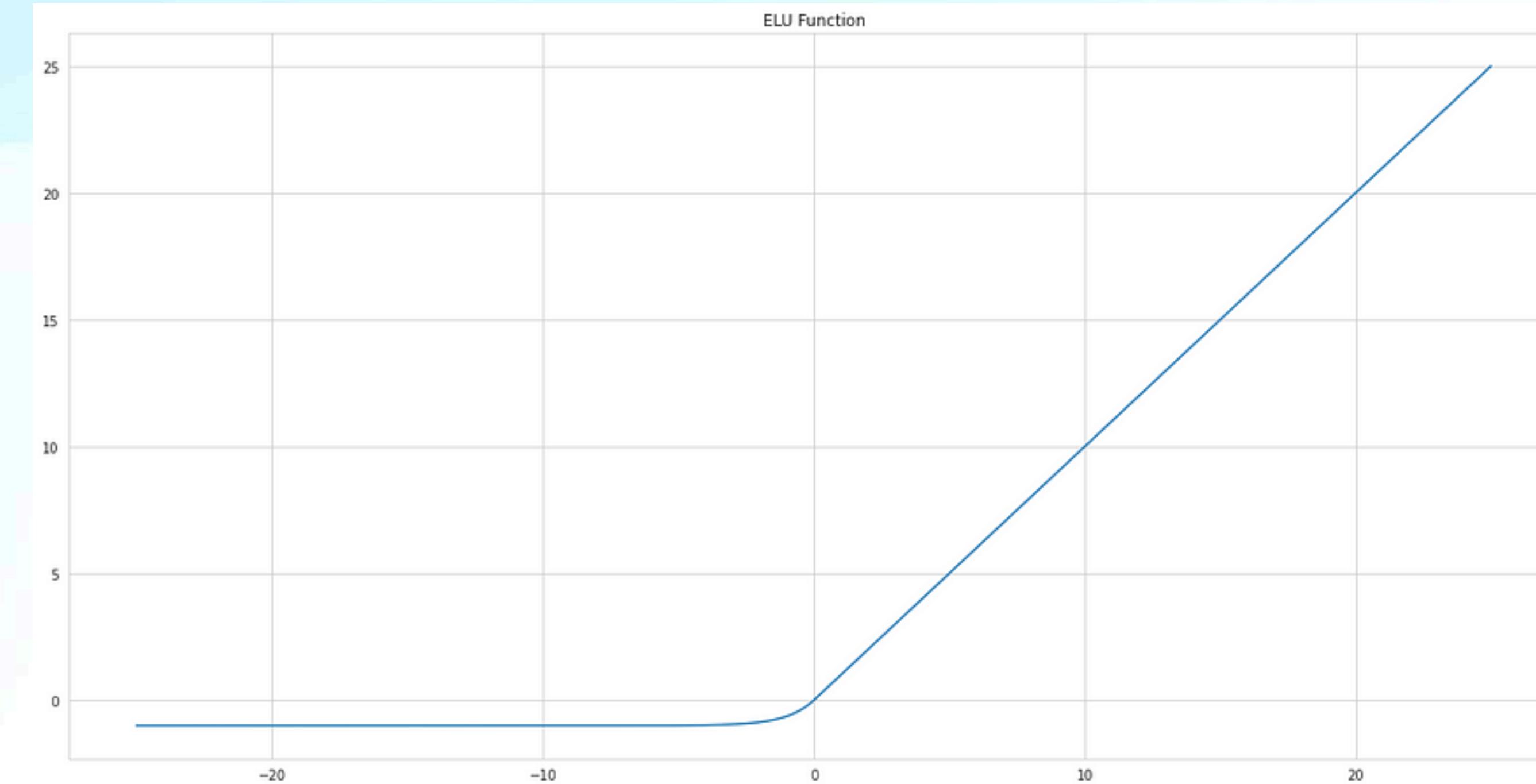
- Dışarıdan α parametresi alır, negatif değerleri onunla çarparak ReLU'da bahsetmiş olduğumuz negatif değerlerin göz ardı edilmesi sorununu çözer.



ÇOK KATMANLI ALGILAYICILAR (MLP)

ELU

- Exponential Linear Units (**ELU**) ortalama aktivasyon değerini sıfıra yaklaşımaya çalışır, bu da öğrenmeyi hızlandırır.



ÇOK KATMANLI ALGILAYICILAR (MLP)

Sigmoid İşlevi

- Sigmoid işlevi, değerleri reel sayılar(R) alanında bulunan girdileri (0, 1) aralığındaki çıktılara dönüştürür.

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

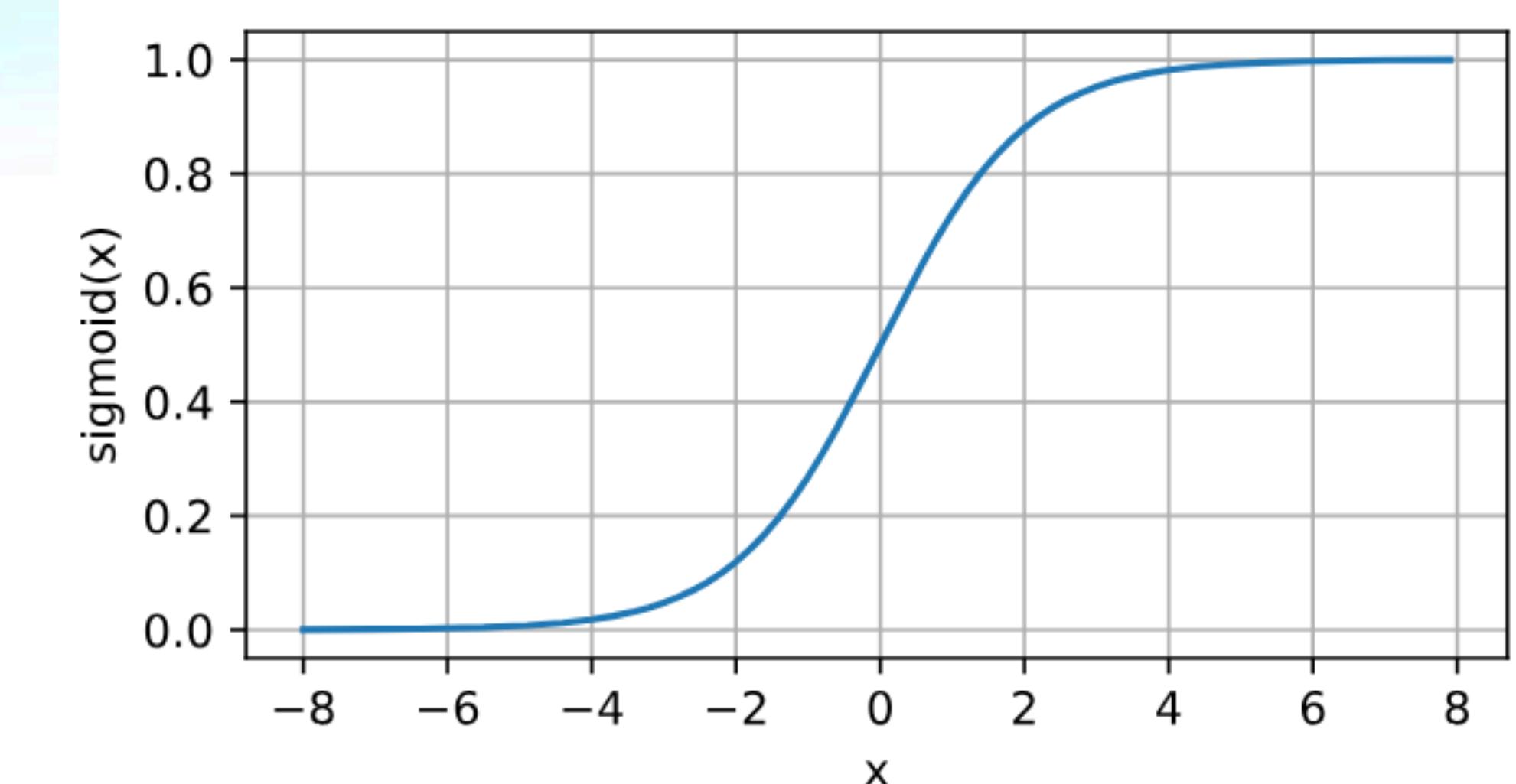
- Bu nedenle, sigmoid genellikle sıkıştırma işlevi olarak adlandırılır. (-sonsuz, sonsuz) aralığındaki herhangi bir girdiyi (0, 1) aralığındaki bir değere sıkıştırır.

ÇOK KATMANLI ALGILAYICILAR (MLP)

Sigmoid İşlevi

- **Sigmoidler**, çıktıları ikili sınıflandırma problemleri için olasılıklar olarak yorumlamak istediğimizde, çıktı birimlerinde etkinleştirme fonksiyonları olarak hala yaygın olarak kullanılmaktadır.

```
y = tf.nn.sigmoid(x) # x'teki her değere sigmoid uygula ve sonucu y'ye ata  
d2l.plot(x.numpy(), y.numpy(), 'x', 'sigmoid(x)', figsize=(5, 2.5)) # x ve sigmoid(x) grafiğini çiz.
```



- Bununla birlikte, çoğu kullanımda gizli katmanlarda sigmoid çoğunlukla daha basit ve daha kolay eğitilebilir ReLU ile değiştirilmiştir.

ÇOK KATMANLI ALGILAYICILAR (MLP)

Sigmoid İşlevi

- Sigmoid fonksiyonunun türevi aşağıdaki denklemle bulunur.

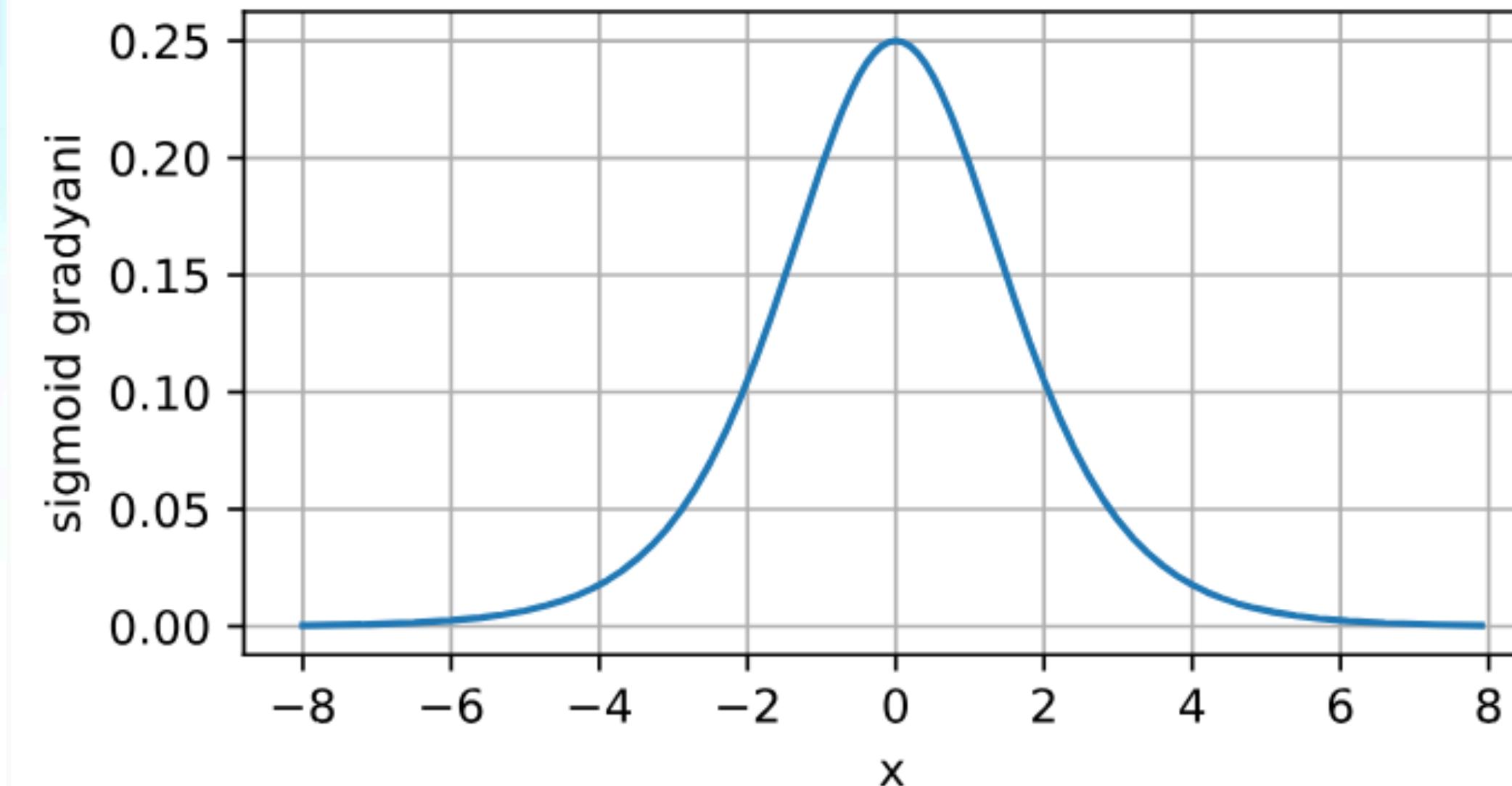
$$\frac{d}{dx} \text{sigmoid}(x) = \frac{\exp(-x)}{(1 + \exp(-x))^2} = \text{sigmoid}(x)(1 - \text{sigmoid}(x)).$$

- Sigmoid fonksiyonunun türevi, sinir ağlarını gradyan tabanlı yöntemlerle eğitmek için gereklidir.
- Sigmoid türevlerinin küçük olması "kaybolan gradyan" problemine yol açabilir ve öğrenmeyi zorlaştırabilir.

ÇOK KATMANLI ALGILAYICILAR (MLP)

Sigmoid İşlevi

```
with tf.GradientTape() as t: # Gradyan hesaplama için GradientTape başlatılır.  
    y = tf.nn.sigmoid(x)      # x'in sigmoid aktivasyon fonksiyonundan geçirildiği y çıktısı hesaplanır.  
d2l.plot(x.numpy(), t.gradient(y, x).numpy(), 'x', 'sigmoid gradyani', figsize=(5, 2.5))  
    # x'e göre y'nin türevi hesaplanır ve bu türev grafikte gösterilir.
```



- Girdi 0 olduğunda, sigmoid fonksiyonunun türevinin maksimum 0.25'e ulaşır. Girdi her iki yönde de 0'dan uzaklaştıkça, türev 0'a yaklaşır.

ÇOK KATMANLI ALGILAYICILAR (MLP)

Tanh İşlevi

- Sigmoid işlevi gibi, **tanh (hiperbolik tanjant)** işlevi de girdilerini sıkıştırarak -1 ile 1 aralığındaki öğelere dönüştürür.

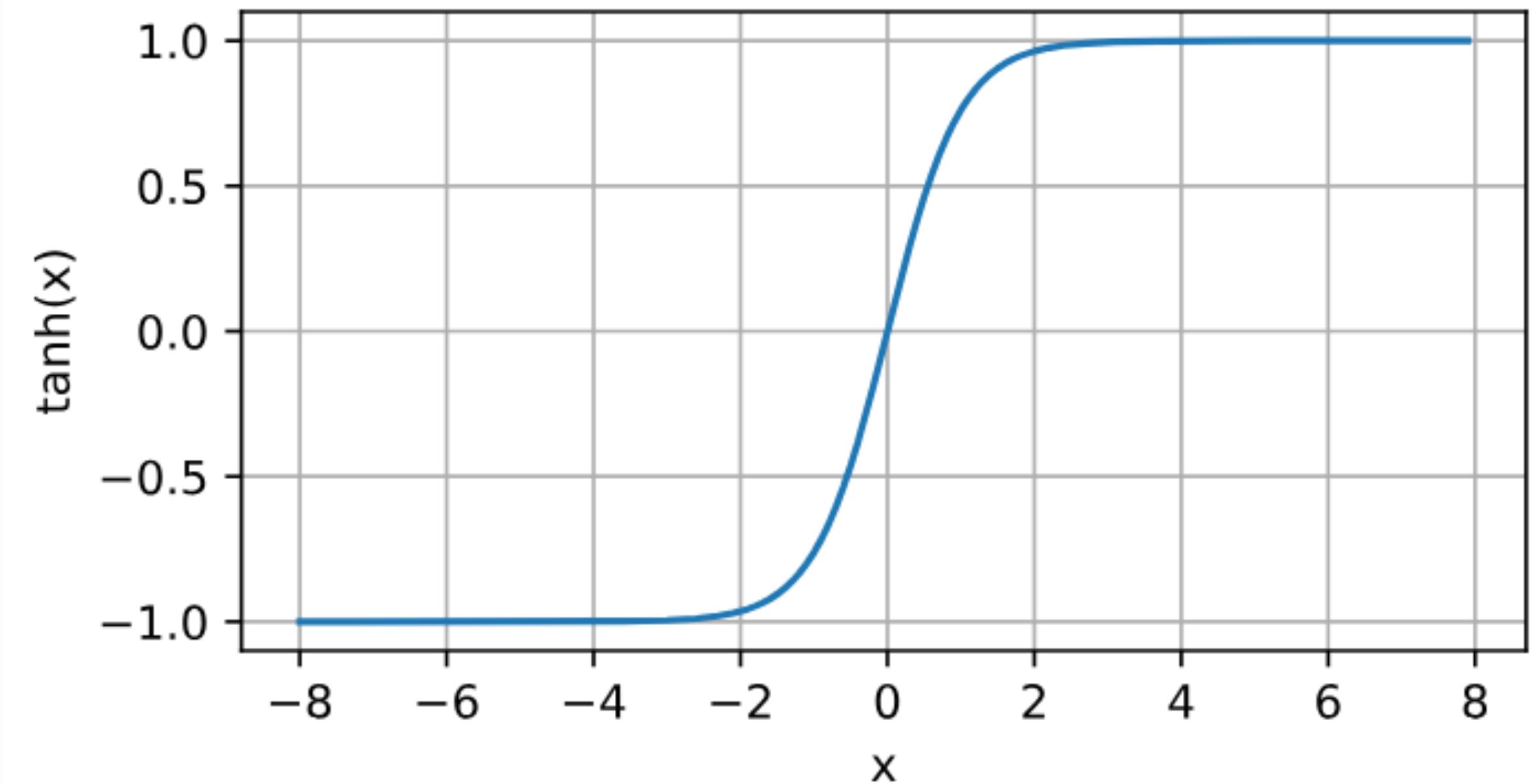
$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}.$$

ÇOK KATMANLI ALGILAYICILAR (MLP)

Tanh İşlevi

- Tanh fonksiyonununda, girdi 0'a yaklaşıkça tanh fonksiyonunun doğrusal bir dönüşümü yaklaşır.
- Fonksiyonun şekli sigmoid fonksiyonuna benzer olmasına rağmen, tanh fonksiyonu koordinat sisteminin orijinine göre nokta simetrisi sergiler.
- Sigmoid fonksiyonuna göre avantajı ise türevinin daha dik olması yani daha çok değer alabilmesidir.
- Bu daha hızlı öğrenme ve sınıflama işlemi için daha geniş aralığa sahip olmasından dolayı daha verimli olacağı anlamına gelmektedir.

```
y = tf.nn.tanh(x) # Girdi değerlerini tanh fonksiyonuya (-1, 1) aralığına dönüştür.  
d2l.plot(x.numpy(), y.numpy(), 'x', 'tanh(x)', figsize=(5, 2.5))  
# Girdi ve tanh ile dönüştürülmüş çıktı değerlerini grafikte göster.
```



ÇOK KATMANLI ALGILAYICILAR (MLP)

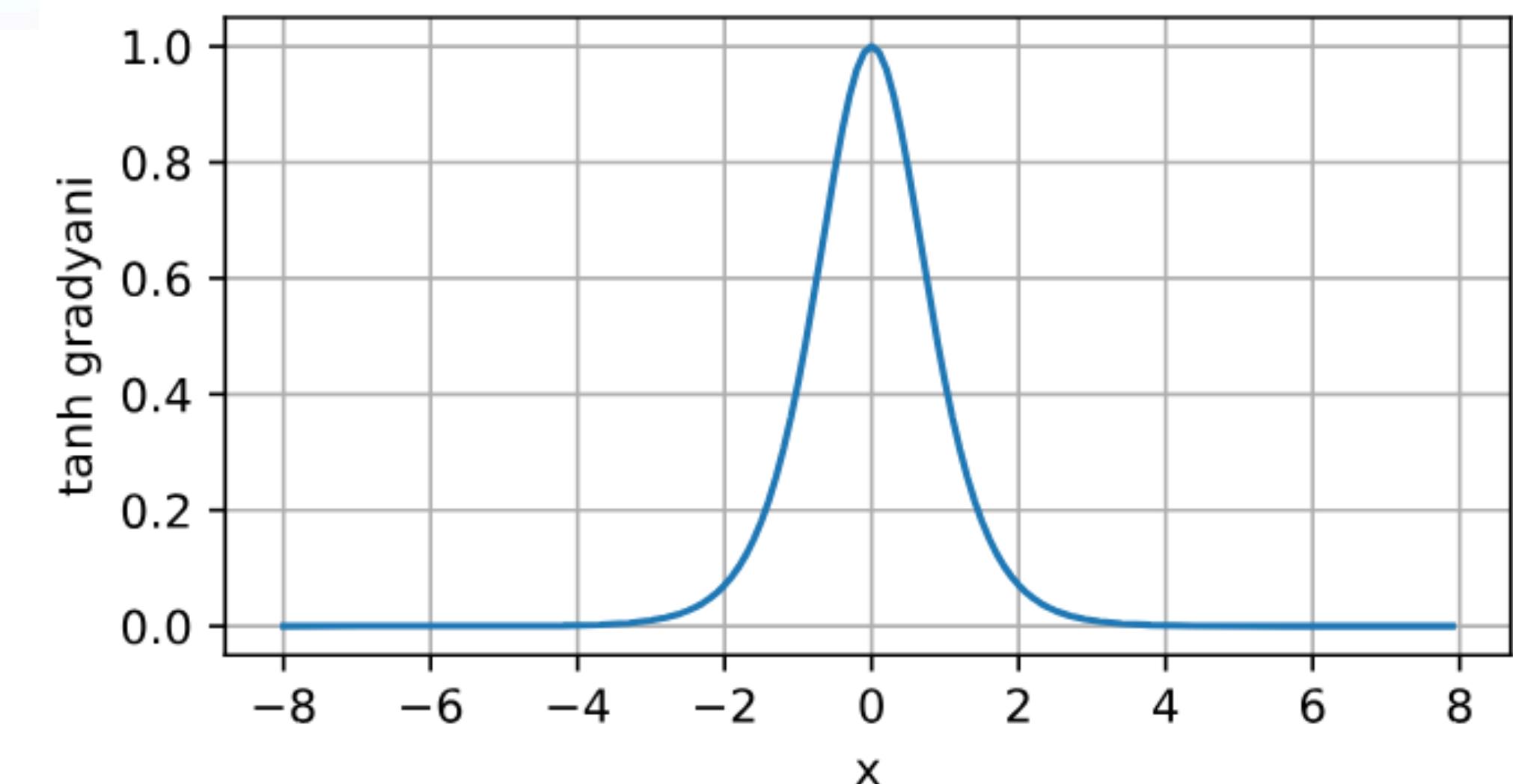
Tanh İşlevi

- Tanh fonksiyonunun uçlarında da **vanishing gradient** problemi vardır.

- Tanh fonksiyonunun türevi ise şu formülle bulunur:

$$\frac{d}{dx} \tanh(x) = 1 - \tanh^2(x).$$

```
with tf.GradientTape() as t: # GradientTape, gradyanları hesaplamak için kullanılır
    y = tf.nn.tanh(x) # x değerlerine tanh fonksiyonunu uygular ve y değerlerini elde eder
d2l.plot(x.numpy(), t.gradient(y, x).numpy(), 'x', 'tanh gradyani', figsize=(5, 2.5))
# x ve tanh fonksiyonunun gradyanlarını grafik olarak gösterir
```



ÇOK KATMANLI ALGILAYICILAR (MLP)

Çok Katmanlı Algılayıcıların Sıfırdan Uygulanması

```
import tensorflow as tf # TensorFlow kütüphanesini içe aktarır.  
from d2l import tensorflow as d2l # d2l kütüphanesinden TensorFlow fonksiyonlarını içe aktarır.  
  
batch_size = 256 # Eğitim ve test verileri için bir batch boyutu belirler.  
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)  
# Fashion MNIST veri setini yükler ve verileri belirtilen batch boyutuna göre böler.  
# 'train_iter' eğitim verileri ve 'test_iter' test verileri için iterator (yineleyici) döner.
```

- **Fashion-MNIST** veri kümesi, 10 farklı sınıf içerir ve her bir resim, gri tonlamalı piksel değerlerinden oluşan bir izgaradan meydana gelir.
- Bu durumda, piksellerin uzamsal yapısını göz ardı ederek, bu veriyi basit bir sınıflandırma problemi olarak düşünebiliriz. Her resim, toplamda 784 girdi özniteliği (28x28 piksel) ve 10 sınıfı sahiptir.

ÇOK KATMANLI ALGILAYICILAR (MLP)

Çok Katmanlı Algılayıcıların Sıfırdan Uygulanması

```
# Girdi, çıktı ve gizli katman boyutlarını tanımla.  
num_inputs, num_outputs, num_hiddens = 784, 10, 256  
  
# İlk katmanın ağırlık matrisini rastgele normal dağılımdan oluştur.  
W1 = tf.Variable(tf.random.normal(  
    shape=(num_inputs, num_hiddens), mean=0, stddev=0.01))  
# İlk katmanın ekleme vektörünü sıfırlardan oluştur.  
b1 = tf.Variable(tf.zeros(num_hiddens))  
  
# İkinci katmanın ağırlık matrisini rastgele normal dağılımdan oluştur.  
W2 = tf.Variable(tf.random.normal(  
    shape=(num_hiddens, num_outputs), mean=0, stddev=0.01))  
# İkinci katmanın ekleme vektörünü rastgele normal dağılımdan oluştur.  
b2 = tf.Variable(tf.random.normal([num_outputs], stddev=.01))  
  
# Tüm parametreleri bir liste içinde birleştir.  
params = [W1, b1, W2, b2]
```

ÇOK KATMANLI ALGILAYICILAR (MLP)

Model

- Her şeyin nasıl çalıştığını bildiğimizden emin olmak için, yerleşik relu işlevini doğrudan çağrırmak yerine ReLU aktivasyonunu kendimiz uygulayacağız.

```
def relu(X): # ReLU aktivasyon fonksiyonu tanımla.  
    return tf.math.maximum(X, 0) # Giriş değeri X ile 0 arasındaki maksimum değeri döner.  
    # X negatifse 0 döner; X pozitifse X'in kendisini döner.
```

- Uzamsal yapıyı göz ardı ettiğimiz için, her iki boyutlu imgeyi **num_inputs** uzunluğuna sahip düz bir vektör halinde (**reshape**) yeniden şekillendiriyoruz. Son olarak, modelimizi sadece birkaç satır kodla uyguluyoruz.

```
def net(X):  
    X = tf.reshape(X, (-1, num_inputs)) # Giriş verisini (X) belirtilen şekle (num_inputs) dönüştür.  
    H = relu(tf.matmul(X, W1) + b1)  
    # Giriş verisi ile, ağırlık matrisini çarp ve bias ekle, ReLU aktivasyon fonksiyonu uygula.  
    return tf.matmul(H, W2) + b2 # Gizli katmandan çıkış katmanına çarp ve bias ekleyerek sonucu döndür.
```

ÇOK KATMANLI ALGILAYICILAR (MLP)

Kayıp İşlevi

- Sayısal kararlılığı sağlamak, hesaplamaların doğru bir şekilde yapılabilmesi için gereklidir. Softmax işlevini sıfırdan uygularken, sayısal kararlılığı artırmak için genellikle yüksek seviyeli API'ler kullanılır.
- **Softmax İşlevi, bir sınıflandırma probleminde modelin çıktısını olasılıklara dönüştürmek için kullanılır.**
- **Çapraz entropi kaybı** ise bu olasılıkların gerçek etiketlerle ne kadar iyi eşleştiğini ölçmek için kullanılır. Yüksek seviyeli API'ler, bu iki işlemi bir arada yaparak hesaplamaları daha kolay ve güvenli hale getirir.

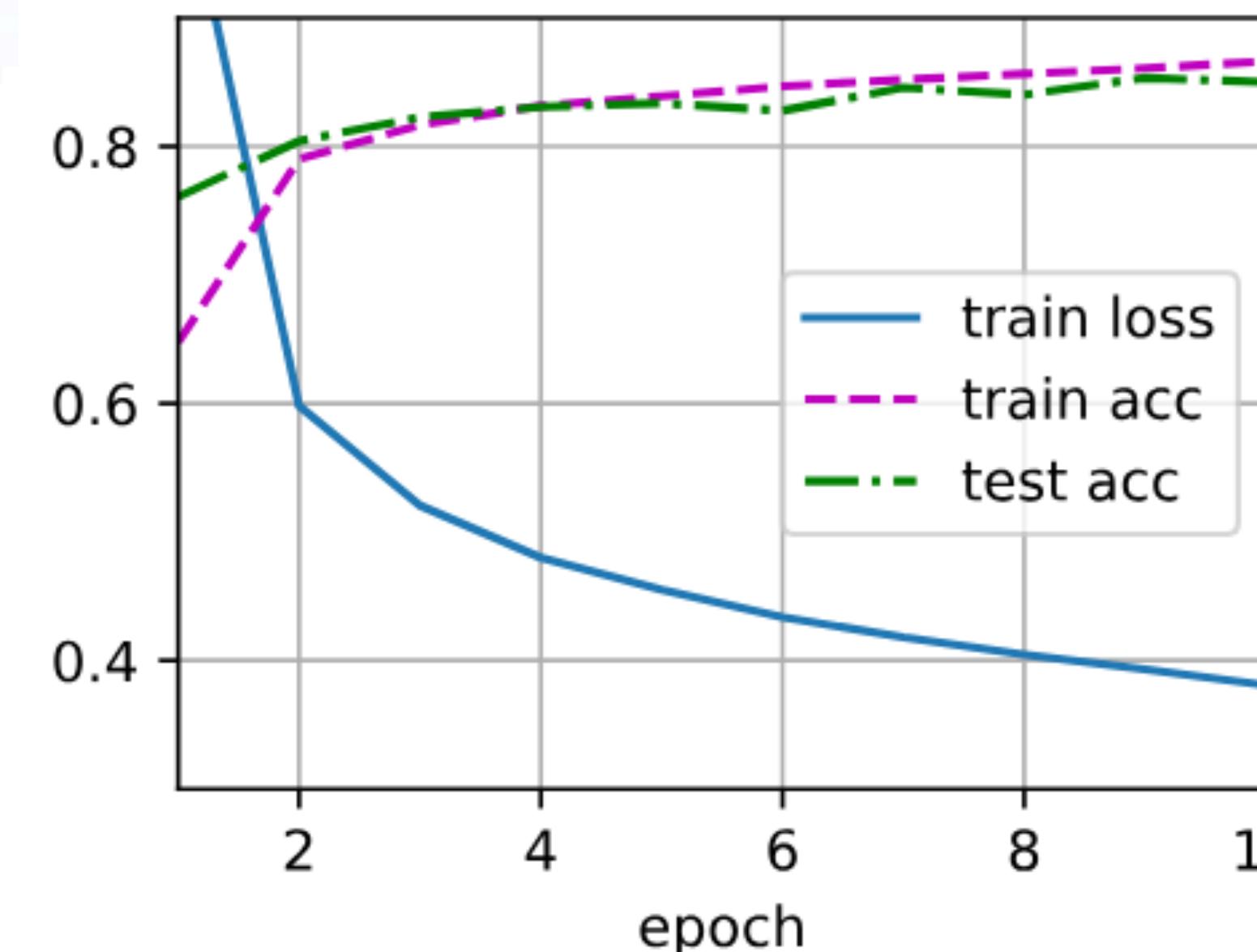
```
def loss(y_hat, y): # Kayıp fonksiyonu tanımlanır.  
    return tf.losses.sparse_categorical_crossentropy(  
        # Gerçek etiketler ile tahminler arasındaki kaybı hesaplanır.  
        y, y_hat, from_logits=True)  
    # 'from_logits=True' ile tahminlerin ham değerler (logits) olduğunu belirttilir.
```

ÇOK KATMANLI ALGILAYICILAR (MLP)

Eğitim

- MLP'ler için eğitim döngüsü softmax bağlanmasıyla tamamen aynıdır. “**d2l**” paketini kullanarak, **train_ch3** fonksiyonunu çağıyoruz. Dönem sayısını 10 ve öğrenme oranını 0,1 olarak ayarlıyoruz.

```
num_epochs, lr = 10, 0.1 # 10 eğitim döngüsü ve 0.1 öğrenme oranı ayarlıyoruz.  
updater = d2l.Updater([W1, W2, b1, b2], lr) # Ağırlık ve bias güncelleleyici oluşturuyoruz.  
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, updater) # Modeli eğitiyoruz.
```



ÇOK KATMANLI ALGILAYICILAR (MLP)

Eğitim

- Son olarak öğrenilen modeli değerlendirmek için onu test verisine uyguluyoruz.

```
d2l.predict_ch3(net, test_iter)
# Eğitimden geçmiş MLP modelinin test verileri üzerinde tahmin yapmasını sağlar.
```

ÇOK KATMANLI ALGILAYICILAR (MLP)

İleri Yayılım, Geriye Yayılım ve Hesaplamalı Grafikler

- Derin öğrenme, insan beyninin çalışma biçimini taklit eden yapay sinir ağlarına dayanır. Bu ağlar, büyük veri setlerini analiz ederek karmaşık problemleri çözmemize olanak tanır. Ancak, bir sinir ağının eğitilmesi, çeşitli matematiksel işlemler ve optimizasyon yöntemleri gerektirir.
- Burada devreye ileri yayılım (**forward propagation**), geri yayılım (**backward propagation**) ve hesaplamalı grafikler (**computational graphs**) girer.

ÇOK KATMANLI ALGILAYICILAR (MLP)

İleri Yayılım

- İleri yayılım**, girdi katmanından çıktı katmanına sırayla bir sinir ağı için ara değişkenlerin (**çıktılar dahil**) hesaplanması ve depolanması anlamına gelir.

•**Giriş Katmanı**

- Giriş Verileri**: Giriş katmanında, modelin alacağı verilere karar verilir. Her bir girdi, bir nöron tarafından temsil edilir.

•**Ağırlık ve Bias**

- Ağırlıklar (Weights)**: Her nöron, kendisine bağlı diğer nöronlardan gelen girdilere ağırlıklar ile katkıda bulunur. Ağırlıklar, modelin öğrenme sürecinde ayarlanan parametrelerdir.

- Bias (Önyargı)**: Her nöronun çıkışına eklenen bir sabit terimdir. Bias, modelin daha esnek hale gelmesini sağlar.

•**Aktivasyon Fonksiyonu**

- Her nöron, gelen toplam girdiyi bir aktivasyon fonksiyonu kullanarak işler. Bu fonksiyon, nöronun çıkışını belirler. Yaygın olarak kullanılan aktivasyon fonksiyonları arasında:

- Sigmoid**

- ReLU** (Rectified Linear Unit)

- Tanh** gibi fonksiyonlar yer alır.

ÇOK KATMANLI ALGILAYICILAR (MLP)

İleri Yayılım Hesaplamaları

• İleri Yayma Hesaplamaları

İleri yayma, genellikle aşağıdaki adımlarla gerçekleştirilir:

- **Girdilerin Ağırlıkla Çarpılması:** Her nöron için, gelen girdiler ile ağırlıkların çarpımı yapılır.

Formül:

$$z = w_1x_1 + w_2x_2 + \dots + w_nx_n + b$$

- Burada **z**, nöronun toplam girdisidir; **w**, ağırlıkları; **x**, girdileri; **b**, bias terimini temsil eder.
- **Aktivasyon Fonksiyonu Uygulama:** Hesaplanan toplam girdiye aktivasyon fonksiyonu uygulanır:

$$a = f(z)$$

- Burada **f**, aktivasyon fonksiyonunu temsil eder ve **a** nöronun çıkışıdır.

ÇOK KATMANLI ALGILAYICILAR (MLP)

İleri Yayılım Hesaplamaları

- **Son Katman Çıkışı**

- Bu işlem, tüm katmanlar boyunca tekrarlanır. Son katmanda, ağıın çıktısı elde edilir.

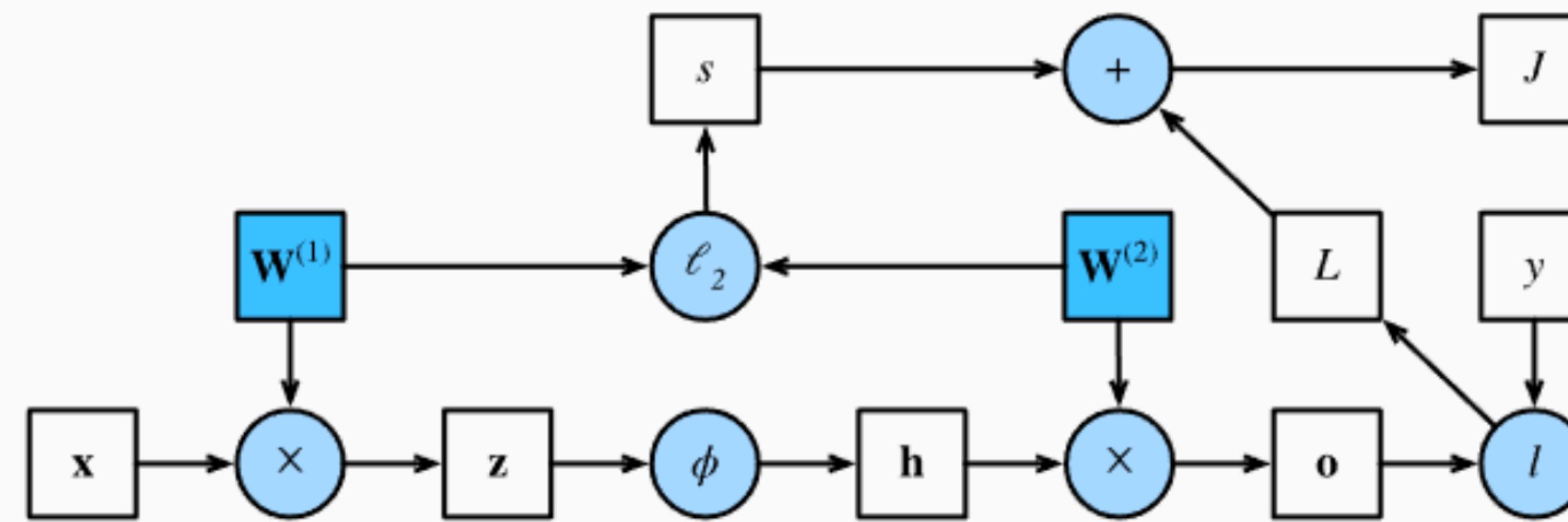
- **Cıktı**

- Sonuç olarak, MLP modelinin çıktısı, girdi verilerine dayanarak hesaplanan bir değer kümesidir. Bu çıkış, genellikle sınıflandırma veya regresyon problemleri için kullanılır.

ÇOK KATMANLI ALGILAYICILAR (MLP)

İleri Yayılım'ın Hesaplama Grafiği

- Hesaplama çizgeleri çizmek hesaplamadaki operatörlerin ve değişkenlerin bağımlılıklarını görselleştirmemize yardımcı olur.



- **Kareler** değişkenleri ve **daireler** işlemleri temsil eder. **Sol alt köşe** girdiyi, **sağ üst köşesi** çıktıyi belirtir. **Okların yönleri** veri akışını gösterir.

ÇOK KATMANLI ALGILAYICILAR (MLP)

Geri Yayılım

- Geri Yayılım, **Çok Katmanlı Perceptron** (Yapay Sinir Ağları) eğitimi için denetimli bir öğrenme algoritmasıdır.
- Geri Yayılım (Backpropagation) algoritması, **delta kuralı veya gradyan iniş** olarak adlandırılan bir teknik kullanarak ağırlık alanındaki hata fonksiyonunun **minimum değerini** arar.
- Hata fonksiyonunu en aza indiren ağırlıkların öğrenme problemine bir çözüm olduğu düşünülmektedir.

ÇOK KATMANLI ALGILAYICILAR (MLP)

Neden Geri Yayılım (Backpropagation) Algoritmasına İhtiyacımız Var?

- Bir yapay sinir ağı tasarlarken, başlangıçta, ağırlıkları bazı rastgele değerler veya bu gerçek için herhangi bir değişken ile başlatırız. Dolayısıyla, seçmiş olduğumuz ağırlık değerleri doğru olmalı ya da modelimize en uygunu olmalıdır.
- Başlangıçta bazı ağırlık değerleri seçtik, ancak model çıktımız gerçek çıktımdan çok farklı, yani hata değeri çok büyük.
- Temel olarak, yapmamız gereken şey, bir şekilde parametrelerin (ağırlıklar) değiştirilmesinin modelini açıklamak zorundayız; Onu başka bir şekilde ifade edelim, modelimizi eğitmemiz gerekiyor.

ÇOK KATMANLI ALGILAYICILAR (MLP)

Geri Yayılm Adımları

- **Hatayı hesaplayın** – Model çıktısı gerçek çıkıştan ne kadar uzakta?
- **Minimum hata mı?** – Hatanın minimize olup olmadığını kontrol edin.
- **Parametreleri güncelle** – Hata büyükse o zaman parametreleri (ağırlık ve önyargıları) güncelleyin. Bundan sonra tekrar hatayı kontrol edin. Hata minimuma ulaşıcaya kadar işlemi tekrarlayın.
- **Model bir tahminde bulunmaya hazır** – Hata minimuma ulaştığında, bazı girdileri modelinize besleyebilir ve çıktıyı üretecektir.
- Bir modelin ileri besleme aşamasında, ağı ileri doğru takip ederek değişkenler hesaplanır ve çıktı elde edilir. Geriye yayılım ise bu işlemlerin ters yönde yapılmasıdır; çıktıının, kayıp fonksiyonuna ve daha sonra model parametrelerine göre türevleri hesaplanır. Böylece modelin parametrelerini güncelleyebilmek için gradyanlar elde edilir. Bu işlem, ağın her katmanında tekrarlanır ve ileri besleme sırasında elde edilen ara değerler, tekrar hesaplamaya gerek kalmadan geri yayılımda kullanılır.
- Backpropagation'ın amacı, kayıp fonksiyonuna göre model parametrelerinin gradyanlarını bulmaktır. İleri besleme işlemi tamamlandıktan sonra kayıp fonksiyonu hesaplanır ve ardından geriye doğru giderek her katmanın parametrelerine göre gradyanlar bulunur. Bu gradyanlar, zincir kuralı kullanılarak bir sonraki katmanın gradyanına bağlanır.

ÇOK KATMANLI ALGILAYICILAR (MLP)

Geri Yayılım (Backpropagation) Algoritması:

ağ ağırlıklarını başlat (genellikle küçük rastgele değerler)

yap

forEach eğitim örneği

tahmini = nöral-net-çıktı (ağ, eski) // ileri geçiş

gerçek = öğretmen-çıktı (ex)

çıkıcı birimlerindeki hata (tahmin - gerçek)

compute $\Delta w_{\{h\}}$ gizli katmandan çıktı katmanına kadar tüm ağırlıklar için // geriye doğru geçiş

compute $\Delta w_{\{i\}}$ giriş katmanından gizli katmana kadar tüm ağırlıklar için // geriye doğru geçiş devam etti

ağ ağırlıklarını / hata katmanı tarafından değiştirilmemiş giriş katmanını güncelle

tüm örnekler doğru şekilde sınıflandırılana veya başka bir durma kriteri memnun olana kadar ağı döndür

ÇOK KATMANLI ALGILAYICILAR (MLP)

Sayısal Kararlılık ve Başlatma

- Şu ana kadar uyguladığımız her model, parametrelerimizin belirli bir dağılıma göre başlatılmasını gerektiriyordu.
- Ancak, başlatma şemasının seçimi sinir ağı öğreniminde önemli bir rol oynar ve sayısal kararlılığın korunmasında da esaslı bir rol oynayabilir. Dahası, bu seçimler aktivasyon fonksiyonunun seçimi ile ilginç şekillerde bağlanabilir.
- Hangi doğrusal olmayan aktivasyon fonksiyonunu seçtiğimiz ve parametrelerimizi nasıl başlatmaya karar verdigimiz, optimizasyon algoritmasının hızlı bir şekilde yakınsamasında çok önemli bir rol oynayabilir.
- Bu konulara dikkat edilmemesi, **gradyanların patlamasına ya da kaybolmasına** yol açabilir. Şimdi bu konuları daha ayrıntılı bir şekilde ele alacağız.

ÇOK KATMANLI ALGILAYICILAR (MLP)

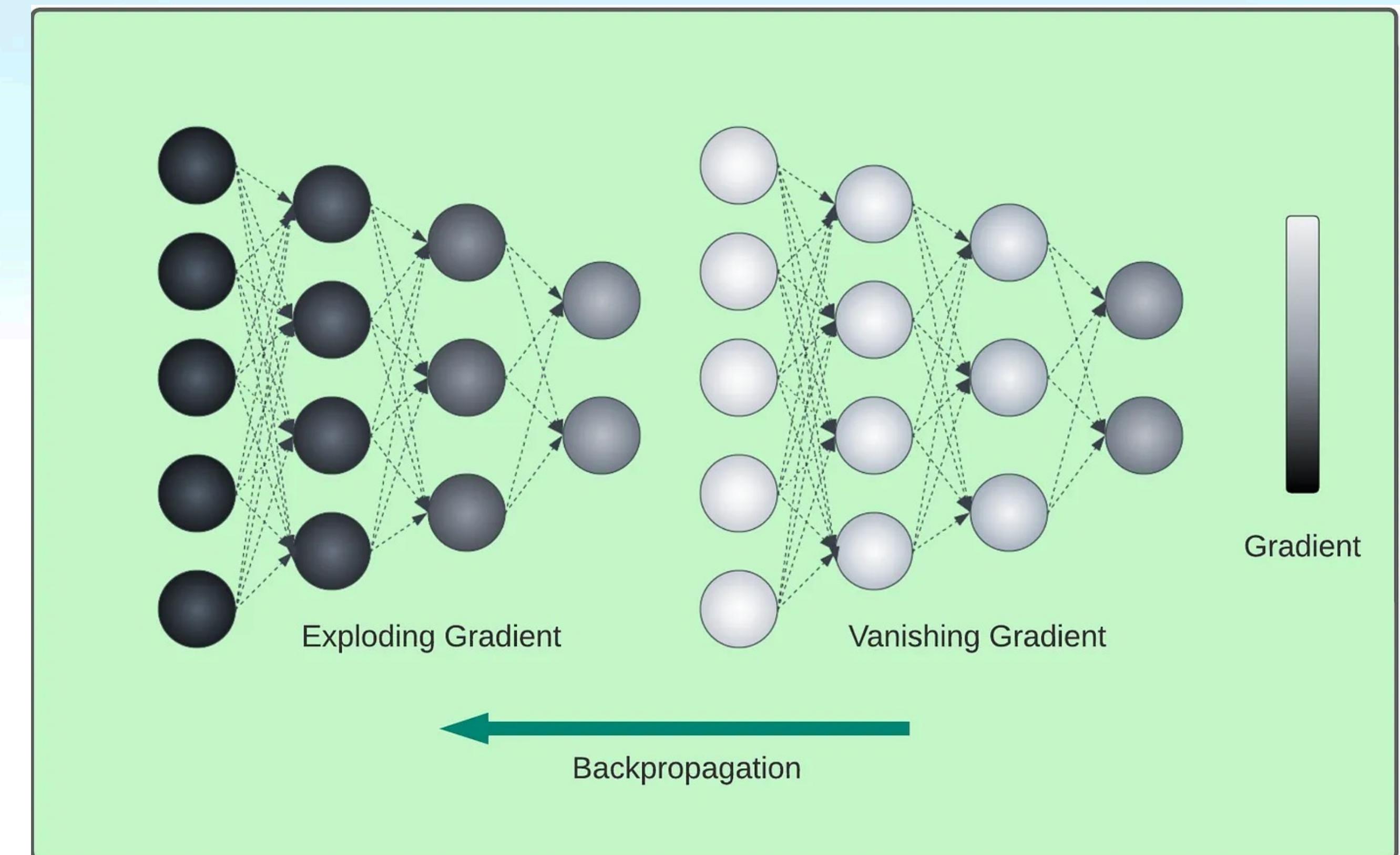
Kaybolan ve Patlayan Gradyanlar

- Gradyan kararsızlığı, derin sinir ağlarında öğrenmeyi zorlaştıran bir sorundur. Ağırlık matrislerinin çarpımları sonucunda gradyanlar çok büyük veya çok küçük olabilir.
- Bu durum, modelin parametre güncellelemelerini ya aşırı büyük yaparak modeli bozabilir (**patlayan gradyanlar**) ya da çok küçük yaparak öğrenmeyi imkânsız hale getirebilir (**kaybolan gradyanlar**).
- Bu yüzden, başlatma şemaları ve aktivasyon fonksiyonları gibi yöntemlerle bu problemler önlenmeye çalışılır.

ÇOK KATMANLI ALGILAYICILAR (MLP)

Kaybolan ve Patlayan Gradyan Problemleri

- **Exploding Gradient (Patlayan Gradyan):** Geri yayılım sırasında gradyanlar aşırı büyük değerlere ulaşır, bu da modelin öğrenmesini zorlaştırır ve istikrarsız hale getirir.
- **Vanishing Gradient (Kaybolan Gradyan):** Gradyanlar çok küçük hale gelir ve ağırlıklar etkili bir şekilde güncellenmez, bu da modelin öğrenmemesine yol açar.



ÇOK KATMANLI ALGILAYICILAR (MLP)

Kaybolan Gradyanlar

- **Vanishing gradyanlar**, geri yayılım sırasında gradyanların son derece küçük hale gelmesiyle oluşur. Bu durum, özellikle derin ağlarda öğrenmeyi yavaşlatır veya durdurur. Doygun aktivasyon fonksiyonları, bu sorunun başlıca nedenlerindendir.

- **Nedenler:**

Doygun Aktivasyon Fonksiyonları: Sigmoid gibi fonksiyonlar, giriş değerleri çok büyük veya çok küçük olduğunda gradyanların sıfıra yaklaşmasına neden olabilir.

Ağırlık Başlatma: Yanlış başlatma yöntemleri, erken doygunluğa yol açabilir.

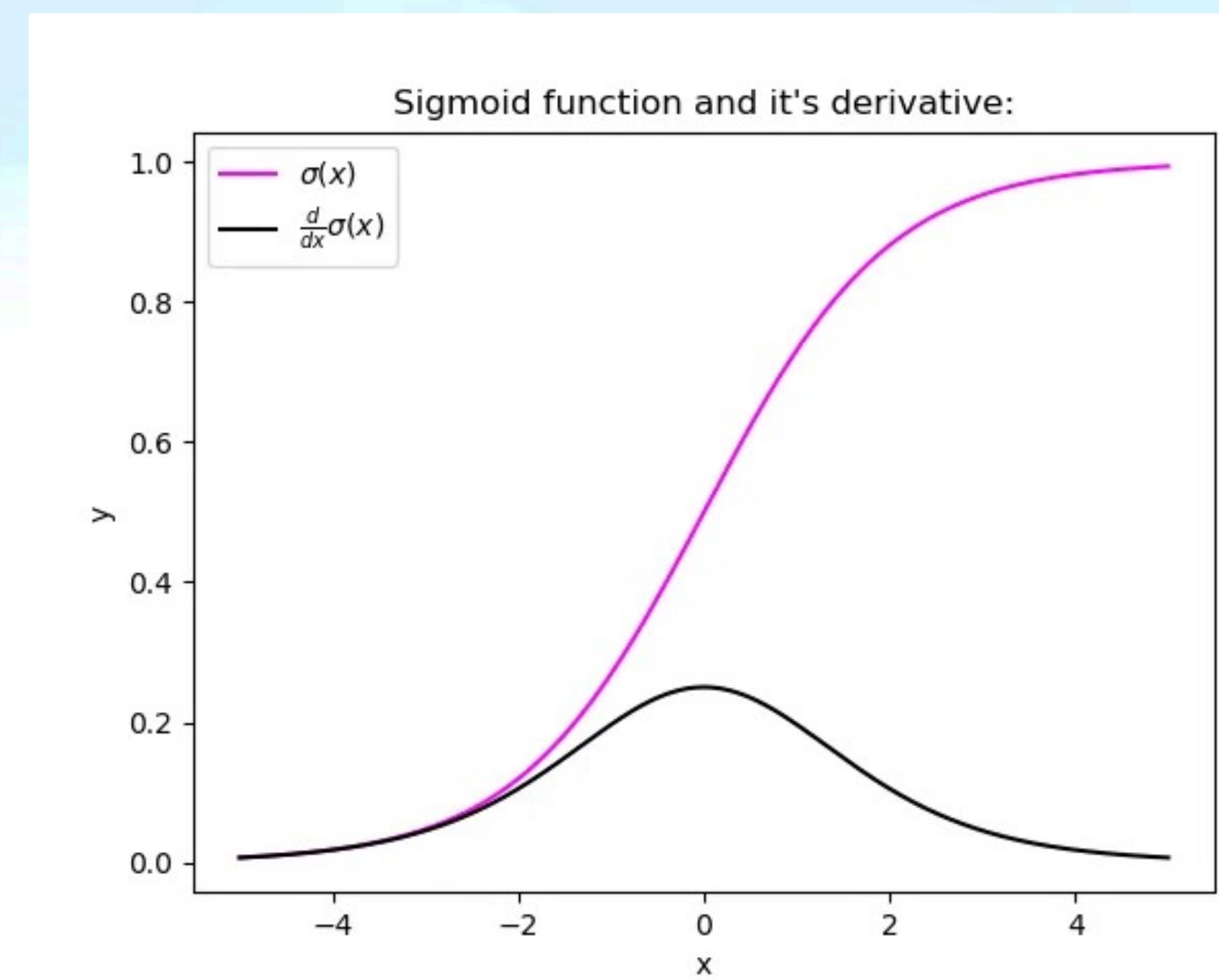
Derin Yapılar: Derin ağlar, uzun yollar üzerinden gradyanların zayıflamasına neden olur.

- **Çözüm Yöntemleri**

ReLU Aktivasyon Fonksiyonu: ReLU (Rectified Linear Unit), pozitif girişler için gradyanların sıfırdan büyük kalmasını sağlar. Bu, gradyanların kaybolmasını önler.

Ağırlık Başlatma: He başlatma gibi uygun teknikler, gradyanların düzgün bir şekilde yayılmasını destekler.

Batch Normalization: Bu yöntem, katman girişlerini normalize ederek aktivasyonları ve gradyanları stabilize eder.



Bu görselde, sigmoid fonksiyonu ve türevi gösterilmiştir. Bu küçük türevler, geri yayılım algoritmasında vanishing gradient problemine yol açabilir.

ÇOK KATMANLI ALGILAYICILAR (MLP)

Patlayan Gradyanlar

- **Patlayan gradyanlar**, gradyanların aşırı büyük hale gelmesiyle ortaya çıkar. Bu durum, eğitimde dengesizlik ve sayısal taşmalara yol açar.

Çözüm Yöntemleri:

- **Gradyan Kırpma (Gradient Clipping)**: Gradyanların belirli bir eşigi aşması durumunda, bu gradyanları o eşeğe ölçeklendirme yöntemidir. Bu, eğitimi stabilize eder ve yakınsamayı sağlar.

Görselleştirme ve Değerlendirme:

- Eğitim sırasında gradyan normlarının görselleştirilmesi önemlidir. Eğitim geçmişini (doğruluk ve kayıp) gradyan normlarıyla birlikte gösteren grafikler, sorunların ne zaman ortaya çıktığını anlamaya yardımcı olabilir.

ÇOK KATMANLI ALGILAYICILAR (MLP)

Patlayan Gradyanlar

- Bu kod parçası, 4x4 boyutunda rastgele matrislerle yapılan çarpma işlemlerini içerir. Her döngüde yeni bir rastgele matris oluşturulup mevcut matrisle çarpıldığında, matrisin değerleri hızla büyüyebilir. **Bu durum, patlayan gradyan sorununa benzer şekilde gradyanların büyümesine neden olabilir.**
- Özellikle, matris çarpımındaki her işlem, derin öğrenme modelinin katmanları arasındaki bilgi akışını etkileyerek gradyanların büyük değerlere ulaşmasına yol açabilir. **Sonuç olarak, çok büyük gradyanlar, modelin öğrenme sürecini dengesiz hale getirebilir ve kayıp fonksiyonunu optimize etmesini zorlaştırabilir.**
- Bu sebeple, bu kod örneği, **patlayan gradyan olasılığını artırın matris çarpımlarını göstermektedir.**

```
# TensorFlow kütüphanesini içe aktar
import tensorflow as tf

# 4x4 boyutunda rastgele bir normal dağılım matrisini oluştur
M = tf.random.normal((4, 4))
print('A single matrix \n', M.numpy()) # İlk rastgele matrisi yazdır

# 100 kez döngü başlat
for i in range(100):
    # Mevcut matrisi yeni rastgele bir 4x4 matrisle çarp
    M = tf.matmul(M, tf.random.normal((4, 4)))

# 100 çarpma işleminden sonra elde edilen matrisi yazdır
print('After multiplying 100 matrices\n', M.numpy())
```

ÇOK KATMANLI ALGILAYICILAR (MLP)

Simetrinin Kırılması

- Neural network tasarıımında, simetri, bir ağın yapı taşları (parametreleri) arasında eşitlik olduğu durumları ifade eder. **Örneğin**, iki gizli birim aynı girişleri alıyorsa ve ağırlıkları aynı şekilde başlatılmışsa, bu birimler eşit sonuçlar üretecektir.

- **Problemin Açıklaması:**

Bir Çok Katmanlı Algılayıcı (MLP) modelini düşünelim. Bu modelde, gizli katmanda iki birimimiz olduğunu varsayıyalım. Eğer bu iki birimin ağırlıkları aynı sabit değerle başlatılırsa, her iki birim de aynı girişleri alır ve aynı aktivasyonları üreterek aynı çıkış birimine ileteceklerdir. Geri yayılım (backpropagation) sırasında, bu durum tüm gradyanların (türevlerin) aynı değeri almamasına yol açar. Sonuç olarak, parametre güncellemlerini asla simetriyi kıramaz ve gizli katman tek bir birim gibi davranır.

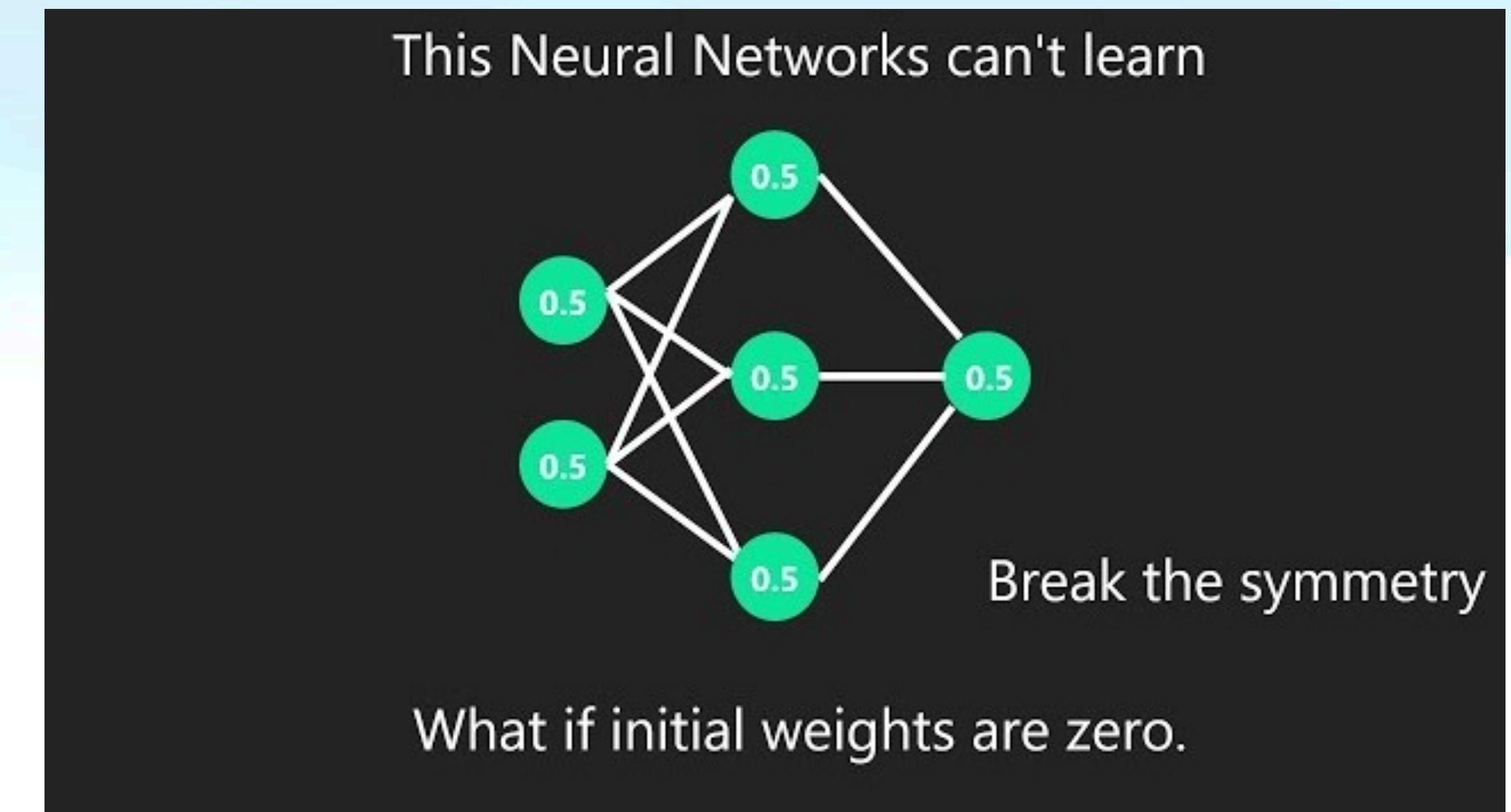
- **Çözüm Yöntemi:**

Mini-batch stokastik gradyan inişi (SGD) simetriyi kıramaz, ancak **dropout** düzenlemesi gibi teknikler bu durumu aşabilir. Dropout, eğitim sırasında bazı birimleri rastgele kapatır, bu da her güncellemede ağın farklı bir mimaridemasına olanak tanır.

ÇOK KATMANLI ALGILAYICILAR (MLP)

Simetrinin Kırılması

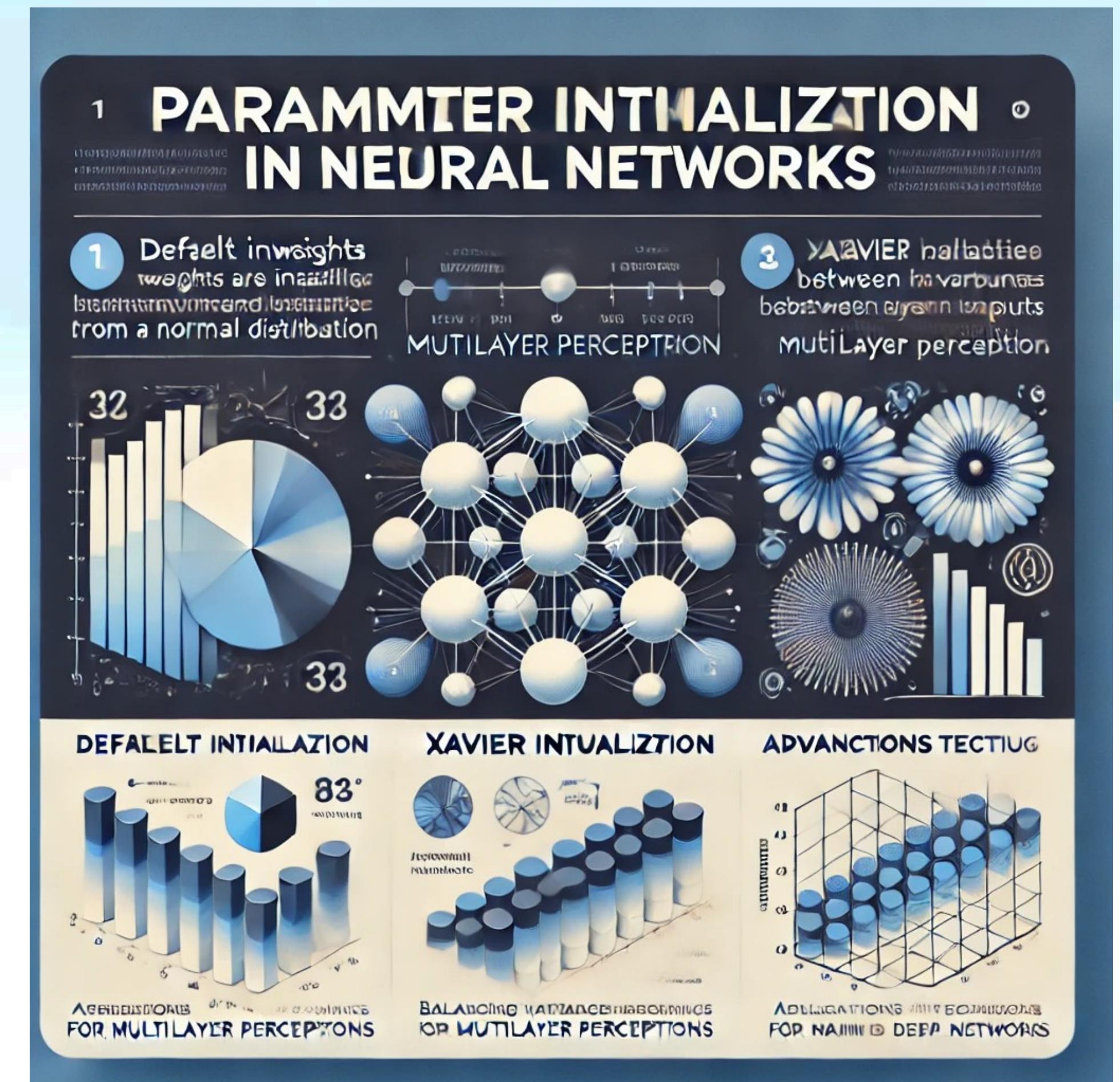
- Bir yapay sinir ağının simetrisini temsil eden bir grafiktir. Grafik, her biri 0.5 değerine sahip olan sabit ağırlıklarla birbirine bağlı üç katman içeren bir çok katmanlı algılayıcı (**MLP**) modelini göstermektedir.
- **Simetrik Ağırlıklar:** Tüm bağlantıların ağırlıkları 0.5 olarak ayarlanmıştır. Bu, her nöronun çıkışının diğer nöronların çıkışına ile aynı olduğu anlamına gelir. Böyle bir durumda, ağaın öğrenmesi için gereken çeşitliliği sağlamak zordur.
- **Öğrenememe Sorunu:** Simetrik ağırlıklar, her nöronun aynı şekilde çalışmasına neden olur, bu da modelin öğrenmesini engeller. Eğer tüm nöronlar aynı şekilde güncellenirse, hiçbir farklı bilgi öğrenemez ve bu durum öğrenme sürecini durdurur.
- **Simetri Kırılması:** Görsel, simetrinin kırılmasının önemini vurgulamakta. Eğer ilk ağırlıklar sıfır olursa, ağaın ilerleyen eğitim süreçlerinde farklı değerler öğrenmesi imkansız hale gelir. Bu nedenle, ağırlıkların rastgele başlatılması gerektiği ifade edilmektedir.



ÇOK KATMANLI ALGILAYICILAR (MLP)

Sinir Ağlarında Parametre Başlatma

- Sinir ağlarının etkili bir şekilde öğrenebilmesi için ağırlıkların ve bias'ların uygun bir şekilde başlatılması büyük önem taşır.
- Başarılı bir başlatma yöntemi, öğrenme sürecindeki dengesizlikleri azaltarak modelin performansını artırır.



ÇOK KATMANLI ALGILAYICILAR (MLP)

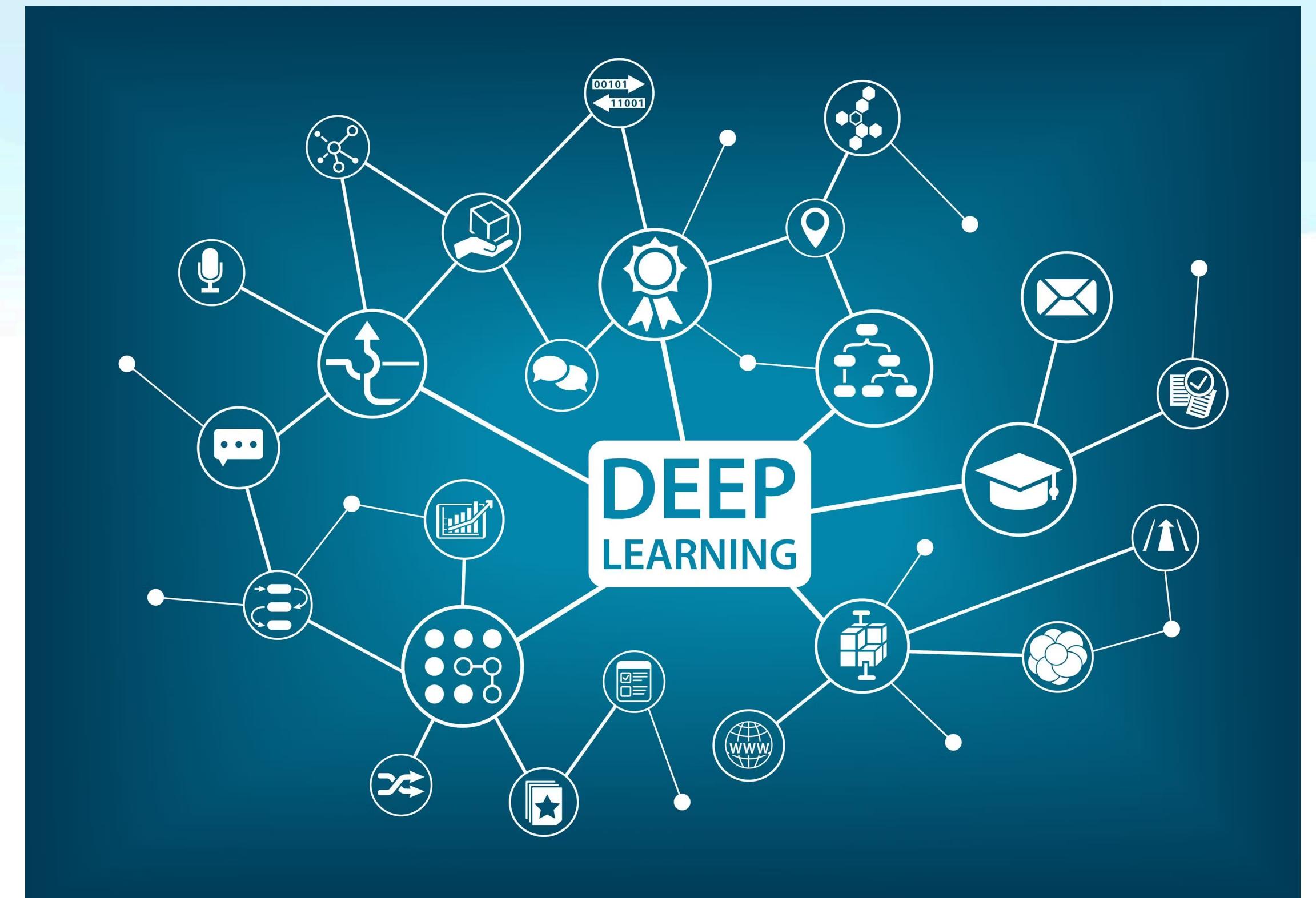
Sinir Ağlarında Parametre Başlatma

- **Varsayılan Başlatma:** Çoğu derin öğrenme çerçevesi, ağırlıkları normal dağılımdan rastgele başlatır. Bu, varsayılan olarak kullanılan bir yöntemdir ve genellikle orta boyutlu problemler için yeterli sonuçlar verir.
- **Xavier Başlatma:** Xavier başlatma, özellikle çok katmanlı algılayıcı (MLP) gibi yapılar için geliştirilmiştir. Ağırlıkların başlatılması sırasında, çıkış ve giriş arasındaki varyans dengesinin sağlanması hedeflenir. Bu yöntem, sıfır ortalama ve belirli bir varyansa sahip ağırlıkların kullanılmasını önerir. Ağırlıkların uygun bir dağılımdan seçilmesi, ağıın daha derin katmanlarında gradyanların patlamasını veya kaybolmasını engeller.
- **Uygulamalar ve İleri Düzey Başlatma Yöntemleri:** Modern derin öğrenme araştırmaları, daha karmaşık başlatma yöntemlerine yönelmiştir. Örneğin, bazı yöntemler, bağlı (paylaşılan) parametreler ve ardışık modeller için özelleştirilmiştir. Xiaove diğerleri (2018) gibi araştırmalar, 10,000 katmanlı sinir ağlarını eğitmek için yenilikçi başlatma teknikleri geliştirmiştir.

ÇOK KATMANLI ALGILAYICILAR (MLP)

Derin Öğrenmede Genelleme Nedir?

- Derin öğrenmede '**Genelleme**' terimi, modelin, modeli oluşturmak için kullanılanla aynı dağılımdan alınan, daha önce görülmemiş yeni verilere uyum sağlama ve uygun şekilde tepki verme yeteneğini ifade eder.
- Başka bir deyişle genelleme, bir modelin bir eğitim seti üzerinde eğitildikten sonra yeni verileri ne kadar iyi sindirebildiğini ve doğru tahminler yapabildiğini inceler.
- Bir modelin ne kadar iyi genelleme yapabildiği başarısının anahtarıdır. Bir modeli eğitim verileri üzerinde çok iyi eğitseniz, genelleme yapamayacaktır. Bu gibi durumlarda, yeni veriler verildiğinde hatalı tahminler yapacaktır. Bu, eğitim veri seti için doğru tahminler yapabilse bile modeli etkisiz hale getirecektir. Bu durum **aşırı uyum** olarak bilinir. Bunun tersi de **yetersiz uyumdur** ve bir modeli yetersiz verilerle eğittiğinizde gerçekleşir.
- Yetersiz uyum durumunda, model eğitim verileriyle bile doğru tahminler yapamayacaktır. Bu da modeli aşırı uyum kadar işe yaramaz hale getirecektir.



ÇOK KATMANLI ALGILAYICILAR (MLP)

Aşırı Öğrenme (Overfitting) ve Düzenleme (Regularization): Derin Öğrenmede Yeniden Düşünme

• Aşırı Öğrenme (Overfitting)

• **Overfitting**, modelin eğitim verisindeki hataları ve gürültüyü öğrenmesiyle oluşan bir problemdir. Bu durum, modelin eğitim verisinde çok iyi sonuçlar elde etmesine rağmen yeni veriler üzerinde zayıf performans göstermesine yol açar. Eğitim verileri, modelin gördüğü örneklerdenoluştugu için model bu verilere aşırı uyum sağlar. Ancak, bu öğrenme süreci genellenebilir olmadığı için model gerçek dünyada kararsız ve hatalı sonuçlar verebilir. Bu, modelin yeni ve daha önce karşılaşmadığı verilerle baş edemeyeceği anlamına gelir.

• Düzenleme (Regularization)

• **Regularization**, modelin aşırı öğrenmeyi önlemesi ve daha iyi genelleme yapabilmesi için kullanılan tekniklerin genel adıdır. Bu bölümde birkaç düzenleme yöntemine değinilir

• **Ağırlık Bozunumu (Weight Decay)**: Modelin aşırı büyük ağırlık değerlerini engelleyen bir yöntemdir. Büyük ağırlıklar, modelin veriye aşırı uyum sağlamasına neden olabilir. Ağırlık bozunumu, her bir ağırlığa bir ceza uygulayarak modelin daha dengeli ve genel öğrenme gerçekleştirmesini sağlar.

• **Dropout**: Bu teknik, her eğitim adımında modelin belirli nöronlarını rastgele devre dışı bırakır. Bu şekilde, modelin aşırı bağımlı olduğu nöronlar engellenir ve model, farklı özelliklerden bağımsız olarak öğrenebilir. Bu da genelleme yeteneğini artırır.

• **Erken Durdurma (Early Stopping)**: Modelin doğrulama seti üzerindeki performansı izlendiğinde, bu performans kötüleşmeye başladığında eğitim süreci durdurulur. Böylece model, eğitim verisine aşırı uyum sağlamadan önce eğitimi bırakır ve daha iyi genelleme yapabilir.

ÇOK KATMANLI ALGILAYICILAR (MLP)

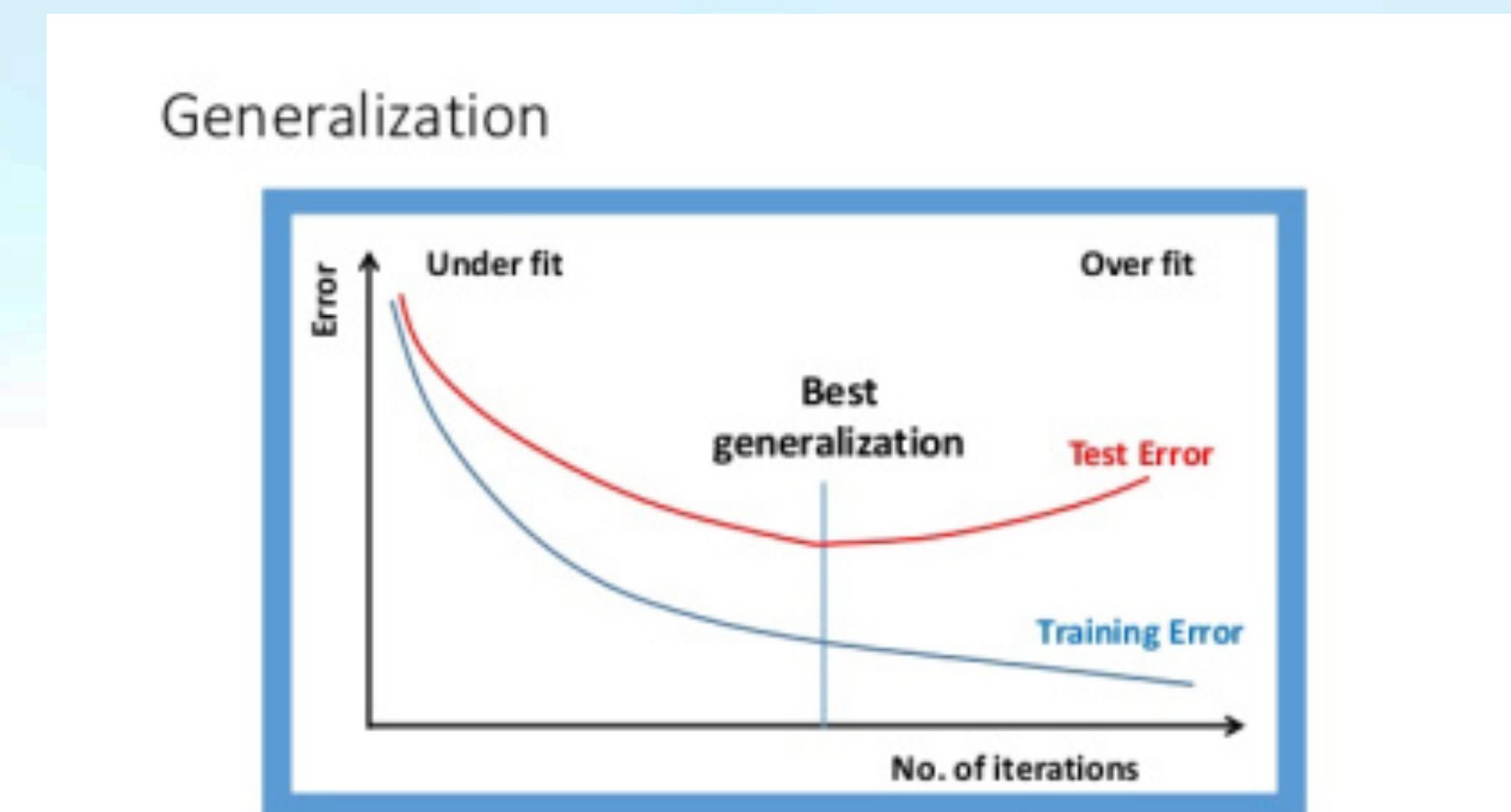
Overfitting ve Regularizasyon'un Derin Öğrenmedeki Zorlukları

- Wolpert ve Macready'nin “**No Free Lunch**” Teoremi'ne göre, her öğrenme algoritması belirli veri dağılımlarında daha iyi, bazı dağılımlarda ise daha kötü sonuçlar verir. Bu nedenle, bir modelin eğitimi sırasında, modelin eğitim verisine uyum sağlama ve genelleme hatasını tahmin etme aşamaları gerçekleşir. **Genelleme farkı**, eğitim verisinde elde edilen performans ile test verisi üzerindeki performans arasındaki faktır ve büyük bir fark olduğunda aşırı öğrenme meydana gelir.
- Geleneksel yaklaşımda, bu farkı azaltmak için modelin karmaşıklığını düşürmek veya parametrelerine kısıtlamalar getirmek gerekebilir. Ancak **derin öğrenme**, bu klasik resme farklı bir boyut kazandırır. Derin öğrenme modelleri, milyonlarca veri örneğine mükemmel uyum sağlayabilen yapılar içerir. Bu durum klasik anlamda aşırı karmaşık olarak değerlendirilebilir, ancak derin öğrenmede bazen daha karmaşık modellerin genelleme hatasını azalttığı gözlemlenir.
- Örneğin, modelin katman veya düğüm sayısını artırarak daha **ifade gücü yüksek** modeller oluşturulabilir ve bu sayede genelleme hatası düşürülebilir. Bu durumu açıklayan bir diğer önemli kavram, “**double-descent**” fenomenidir. Bu fenomen, modelin karmaşıklığı arttıkça genelleme hatasının önce artıp, daha sonra yeniden azalmasıdır.

ÇOK KATMANLI ALGILAYICILAR (MLP)

Derin Öğrenmede Genelleme Sorunları

- Derin öğrenme modellerinin neden genelleme yapabildiğini açıklamak için klasik öğrenme teorisi yetersiz kalabilir. **Neural network'ler**, rastgele etiketlenmiş büyük veri kümelerine bile mükemmel uyum sağlayabilir.
- Geleneksel olarak kullanılan **VC boyutu** veya **Rademacher karmaşıklığı** gibi ölçütler, derin öğrenmenin genelleme yeteneğini açıklamakta zayıf kalmaktadır.
- Bu yüzden derin öğrenmede overfitting'i önlemek için klasik düzenleme yöntemlerinin yanı sıra, modelin ifade gücünü artırarak bile genelleme hatasının azaltılabileceği düşünülmektedir. Bu, derin öğrenmenin klasik yaklaşımlardan farklılaştiği önemli bir noktadır.



ÇOK KATMANLI ALGILAYICILAR (MLP)

Parametrik ve Parametrik Olmayan Modellerin Farklı Perspektifi

•Parametrik Modeller:

•Sabit bir parametre setiyle verileri öğrenen modellerdir. Bu modellerin kapasitesi sabit olduğu için, eğitim verisinin boyutuna veya karmaşıklığına bağlı olarak yetersiz öğrenme veya aşırı öğrenme yapma riski taşırlar. Parametrik modellerde, modelin kapasitesi önceden belirlenir ve bu nedenle eğitim verisinin karmaşıklığı arttıkça modelin genelleme yeteneği sınırlanabilir.

•Nonparametrik Modeller:

•Modelin kapasitesini verinin karmaşıklığına göre ayarlayabilen ve parametre sayısını sabit tutmayan esnek modellerdir. Bu modeller, verinin daha fazla detayını yakalayabilmek için ek parametreler ekleyebilir, bu da onları daha uyumlu ve genelleme yeteneği yüksek hale getirir. Bunun en basit örneklerinden biri, **k-en yakın komşu algoritmasıdır** (k-NN). Bu algoritma, eğitim aşamasında veriyi hafızaya alır ve yeni bir tahmin yapılırken en yakın komşularını bularak karar verir. Eğitim hatası sıfır olabilir, ancak bu algoritma genelleme yapma yeteneğine de sahiptir.

•k-NN algoritmasında en önemli unsur, kullanılan **mesafe metriğidir**. Bu metrik, veriler arasındaki uzaklıkları hesaplar ve farklı metrikler, modelin farklı sonuçlar üretmesine neden olabilir. Bu durum, modelin öğrenme sürecinde varsayımlara dayanarak farklı desenleri algılamasına neden olur. Dolayısıyla, mesafe metriği seçimi, modelin **inductive biasını** yansıtır.

ÇOK KATMANLI ALGILAYICILAR (MLP)

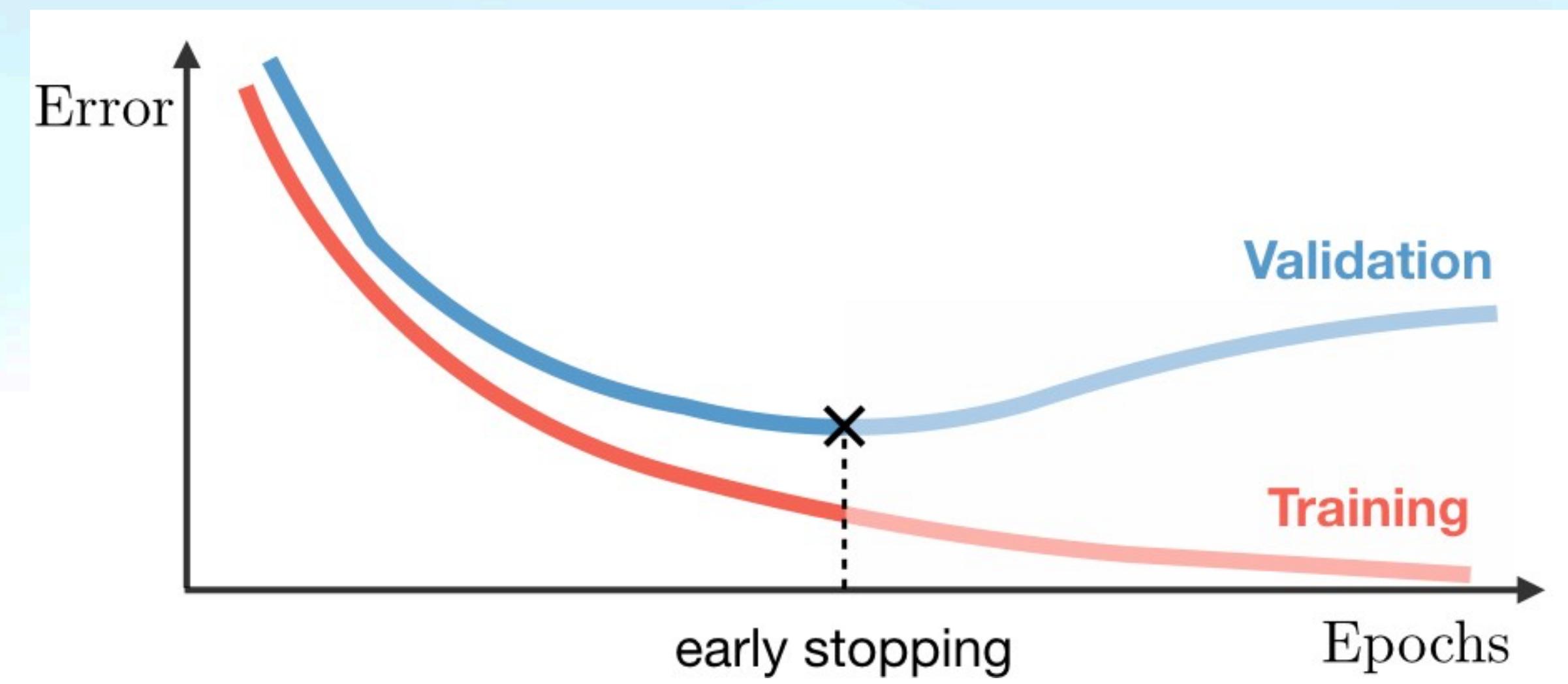
Derin Öğrenme ve Parametrik Olmayan Modeller Arasındaki Bağlantı

- Aşırı **parametrik** sinir ağlarının (neural networks) bazı durumlarda **parametrik olmayan** modeller gibi davranabilir. Büyük sinir ağları, eğitilen veriyi mükemmel bir şekilde öğrenebilir ve bu süreç, tipki parametrik olmayan yöntemler gibi karmaşıklığı artırabilir.
- Son zamanlardaki teorik araştırmalar, büyük sinir ağları ile parametrik olmayan yöntemler, özellikle **kernel (çekirdek) yöntemleri** arasında güçlü bağlantılar keşfetmiştir. Örneğin, **Jacot ve arkadaşları (2018)**, sonsuz genişlikte çok katmanlı algılayıcıların (MLP'ler), belirli bir çekirdek fonksiyonuyla **neural tangent kernel** adı verilen bir yapıya denk geldiğini göstermiştir.
- Bu keşif, modern derin öğrenme modellerinin davranışını anlamak için parametrik olmayan modellemenin faydalı olabileceğini işaret etmektedir.

ÇOK KATMANLI ALGILAYICILAR (MLP)

Erken Durma Nedir?

- **Erken durdurma**, modelin doğrulama seti üzerindeki performansını sürekli olarak izleyerek, bu performans belirli bir noktadan sonra kötüleşmeye başladığında eğitim sürecini durdurma yöntemidir.
- Modelin eğitim verisi üzerindeki başarısı yüksek olsa bile, doğrulama setindeki hata oranı artıyorsa bu, modelin aşırı öğrenme yapmaya başladığını gösterir.
- Bu durumda, erken durdurma teknigi devreye girer ve modelin daha fazla eğitim almasını engelleyerek genel performansını artırır.



Eğitim verisi ile test verisi hataları arasındaki fark açılığı belli bir seviyeye geldiğinde eğitimi durdurur. Bazı algoritmalar bunu sürekli kontrol ederek otomatik olarak yapar.

ÇOK KATMANLI ALGILAYICILAR (MLP)

Erken Durma: Uygulama

- **Eğitim ve Doğrulama Setlerinin Ayrılması:** Modelin eğitim verisi ve doğrulama verisi olarak iki ayrı set oluşturulur. Model, eğitim seti üzerinde eğitilirken, doğrulama seti üzerinde de performansı değerlendirilir.
- **Doğrulama Performansının İzlenmesi:** Her eğitim döngüsünden sonra, modelin doğrulama setindeki kaybı (**loss**) veya doğruluk (**accuracy**) gibi metrikler kaydedilir.
- **Kötüleşme Tespiti:** Eğer doğrulama performansı, belirli bir sayıda ardışık eğitim döngüsünde düşmeye başlarsa, erken durdurma algoritması devreye girer ve eğitim durdurulur.

ÇOK KATMANLI ALGILAYICILAR (MLP)

Erken Durma: Avantajları

- **Aşırı Öğrenmenin Önlenmesi:** Modelin eğitim verisine aşırı uyum sağlamasını engelleyerek, daha iyi bir genelleme yeteneği kazandırır. Bu, modelin yeni verilerle başa çıkabilme yeteneğini artırır.
- **Zaman Tasarrufu:** Eğitim sürecinin gereksiz yere uzamasını önler. Model, gerekli eğitim süresinden daha az bir süre ile eğitilebilir, bu da daha az hesaplama kaynağı gerektirir.

ÇOK KATMANLI ALGILAYICILAR (MLP)

Derin Ağlar için Klasik Regularization Yöntemleri

- Daha önce bahsettiğimiz gibi, modellerin aşırı öğrenmesini önlemek için kullanılan klasik düzenleme yöntemleri derin öğrenmede de önemini koruyor. Özellikle, **weight decay (ağırlık bozunuşu)** yöntemi, büyük ağırlıkları cezalandırarak modelin karmaşıklığını sınırlamaya yönelik bir düzenleme tekniğidir.
- Bu yöntem, kullanılan norm türüne göre **ridge düzenlemesi (L2 cezası)** veya **lasso düzenlemesi (L1 cezası)** olarak adlandırılır. Bu düzenleyiciler, modelin aşırı uyum sağlamasını (overfitting) önlemek amacıyla ağırlıkların değerlerini sınırları.

ÇOK KATMANLI ALGILAYICILAR (MLP)

Derin Ağlar için Klasik Regularization Yöntemleri: Weight Decay (Ağırlık Bozunuşu)

- Bu yöntem, modelin öğrenme sürecinde, ağırlıkların büyümeyi engellemek için kayıp fonksiyonuna ek bir terim ekler. Weight decay, kayıp fonksiyonuna ağırlıkların büyüklüğüne bağlı bir ceza ekleyerek, modelin parametrelerinin gereksiz yere büyümeyi önler. Bu sayede model daha basit ve daha genellenebilir hale gelir.
- **Weight decay**, genellikle **L2** düzenlemeye ile aynı anlama gelir, çünkü L2 düzenlemesi de ağırlıkların karelerinin toplamını ceza olarak ekler. Ancak, weight decay terimi, öğrenme sürecinin her adımında, ağırlıkların küçülmesine neden olarak modelin aşırı karmaşık hale gelmesini önler.

ÇOK KATMANLI ALGILAYICILAR (MLP)

Derin Ağlar için Klasik Regularization Yöntemleri: L1 Düzenleme (Lasso Regularization)
L2 Düzenleme (Ridge Regularization)

• L1 Düzenleme (Lasso Regularization)

- Bu yöntemde, modelin kayıp fonksiyonuna ağırlıkların mutlak değerlerinin toplamı eklenir. L1 düzenleme, bazı ağırlıkları sıfıra yaklaştırarak modelin **sparsiteli** hale gelmesini sağlar, yani bazı parametrelerin sıfırlanmasına neden olur. Bu, daha basit modeller oluşturarak aşırı öğrenmeyi önleyebilir.

• L2 Düzenleme (Ridge Regularization)

- L2 düzenlemesinde ise ağırlıkların karelerinin toplamı kayıp fonksiyonuna eklenir. Bu yöntem, tüm ağırlıkları küçültme eğilimindedir ve aşırı büyük parametrelerin model üzerinde fazla etkili olmasını engeller. L2 düzenleme, modelin daha dengeli ve daha küçük parametrelerle çalışmasını sağlar.

ÇOK KATMANLI ALGILAYICILAR (MLP)

Derin Ağlar için Klasik Regularization

- Derin öğrenme modellerinde **weight decay** hâlâ sıkça kullanılsa da, araştırmalar göstermiştir ki bu yöntem tek başına sinir ağlarının veriyi tam olarak öğrenmesini (interpolasyon) önlemek için yeterli olmayabilir.
- Bu yüzden, **erken durdurma** gibi ek yöntemlerle birlikte kullanıldığında daha etkili olur. Erken durdurma olmadığında, bu yöntemler genelleme performansını artırabilir, ancak bunun nedeni modelin kapasitesini sınırlandırmaktan çok, veri setindeki kalıplara daha uygun induktif önyargılar sunmaları olabilir.
- Ek olarak, derin öğrenme araştırmacıları, klasik düzenleme tekniklerine dayalı yeni yöntemler geliştirmiştir. Örneğin, modelin girdi verilerine gürültü ekleme gibi teknikler klasik yöntemlerden türemiştir.
- Bir sonraki bölümde bahsedeceğimiz ünlü **dropout** teknigi de bu bağlamda önemli bir yer edinmiştir ve derin öğrenmenin temel düzenleme araçlarından biri haline gelmiştir.

ÇOK KATMANLI ALGILAYICILAR (MLP)

Dropout'a Genel Bakış

- Bir modelin iyi bir tahmin performansı göstermesini beklediğimizde, bu modelin **görülmemiş veri** üzerinde de başarılı olmasını isteriz. Klasik genelleme teorisi, eğitim ve test performansı arasındaki farkı kapatmanın en iyi yolunun **basit bir model** hedeflemek olduğunu söyler. Basitlik, az sayıda boyut kullanmakla sağlanabilir. Bunu, doğrusal modellerin **monomial temel fonksiyonları** ile inceledik. Ek olarak, **weight decay (L2 düzenlemesi)** yönteminde olduğu gibi, parametrelerin normunun tersinin de basitliğin iyi bir ölçüsü olduğunu gördük.
- Başka bir basitlik anlayışı ise **düzgünlük** (smoothness) fikridir. Yani, bir fonksiyonun girdilerdeki küçük değişikliklere karşı duyarlı olmaması gereklidir. Örneğin, görüntüleri sınıflandırırken, piksellere eklenen rastgele gürültünün sonuçları ölçüde değiştirmemesi beklenir.
- **Bishop (1995)** bu fikri, girdilere gürültü ekleyerek eğitmenin, Tikhonov düzenlemesine denk olduğunu kanıtlayarak matematiksel olarak formalize etti. Bu çalışma, bir fonksiyonun düzgün olması gereksinimi ile girdilerdeki bozulmalara karşı dayanıklı olmasının gerekliliği arasında net bir bağlantı kurdu.
- Daha sonra **Srivastava ve arkadaşları (2014)**, Bishop'ın bu fikrini ağıın **uç katmanlarına** nasıl uygulayacaklarına dair dahiyane bir yöntem geliştirdiler. Bu fikir, sinir ağı eğitimi sırasında **her bir iç katmana gürültü eklemeyi** içerir ve bu yöntem **dropout** olarak adlandırılmıştır. Dropout, eğitim sırasında bazı nöronları rastgele devre dışı bırakmayı ifade eder. Her yinelemede, her bir katmandaki düğümlerin bir kısmını sıfırlayarak (devre dışı bırakarak) sonraki katmanları hesaplar.

ÇOK KATMANLI ALGILAYICILAR (MLP)

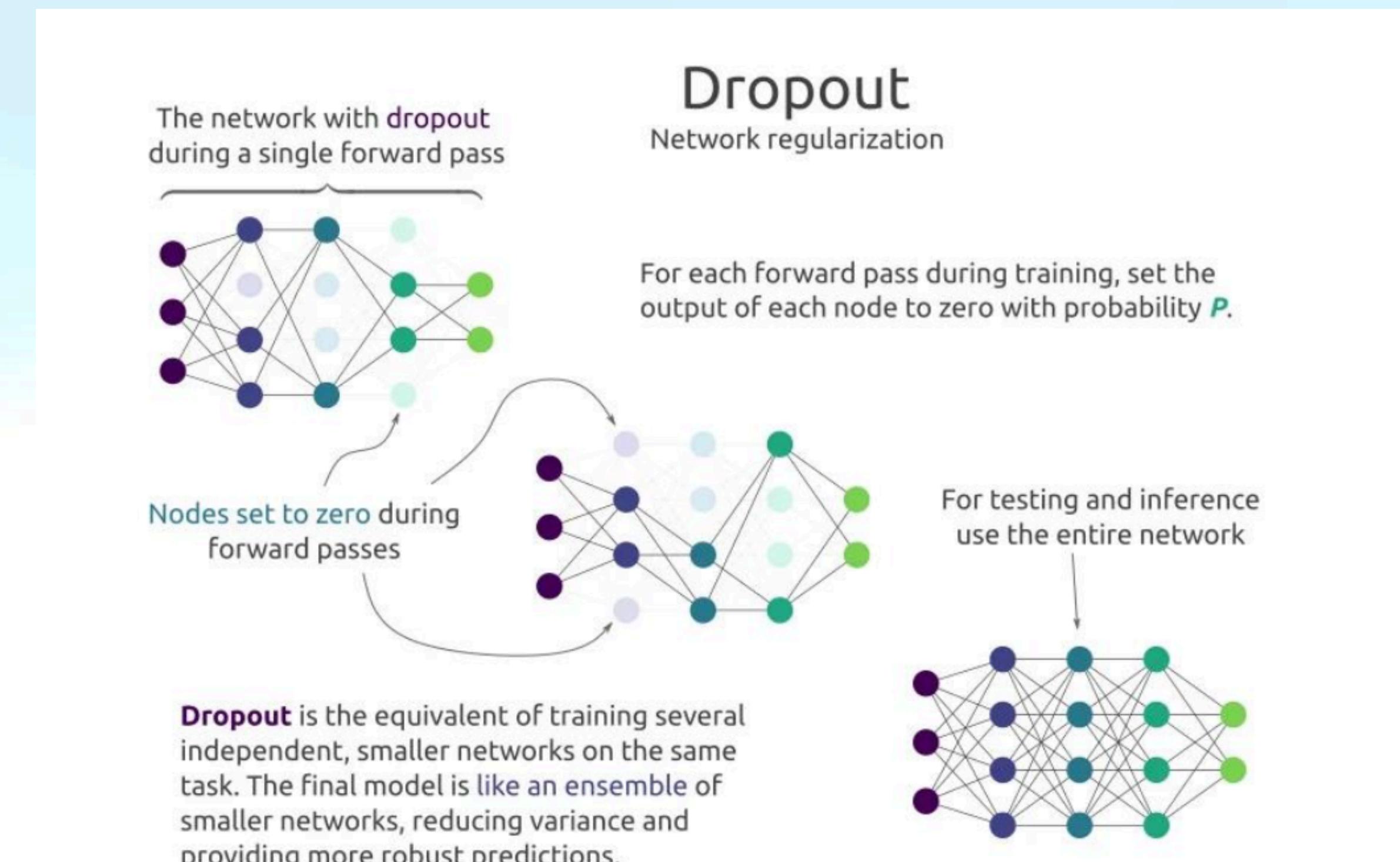
Dropout'a Genel Bakış

- Dropout terimi, her eğitim adımda bazı nöronları “düşürmemiz” gerektiğini ifade eder. Yazarlar, aşırı öğrenmenin sinir ağlarındaki katmanların önceki katmanların belirli aktivasyon kalıplarına çok fazla güvenmesinden kaynaklandığını savunmuşlardır ve bu durumu **ko-adaptasyon** olarak adlandırmışlardır.
- Dropout'un, bu ko-adaptasyonu kırrarak sinir ağlarının daha genellenebilir hale gelmesini sağladığı öne sürülmüştür. Dropout teknigi, sinir ağlarının eğitiminde kalıcı bir yöntem haline gelmiştir ve çoğu derin öğrenme kütüphanesinde uygulanmaktadır.
- Dropout'un temel zorluğu, gürültüyü nasıl enjekte edeceğimizdir. Bu işlemi tarafsız bir şekilde yapmak önemlidir. Yani, her bir katmanın beklenen değeri—diğer katmanlar sabitken—gürültü olmadan alacağı degere eşit olmalıdır.
- **Bishop**, lineer bir modele Gaussian gürültüsü eklemiştir. Her eğitim yinelemesinde, sıfır ortalamalı bir dağılımdan örneklenen gürültü girişe eklenir. Sonuç olarak, pertürbe edilmiş (**Perturbasyon teorisi, tam olarak çözümlenemeyen bir problemin, bu probleme bağlı başka bir problemden yola çıkılarak yaklaşık bir çözüm elde etmek için matematiksel metodlar içeren teoridir.**) bir nokta elde edilir ve bu noktaların beklenen değeri giriş verisine eşittir.

ÇOK KATMANLI ALGILAYICILAR (MLP)

Dropout Yöntemi

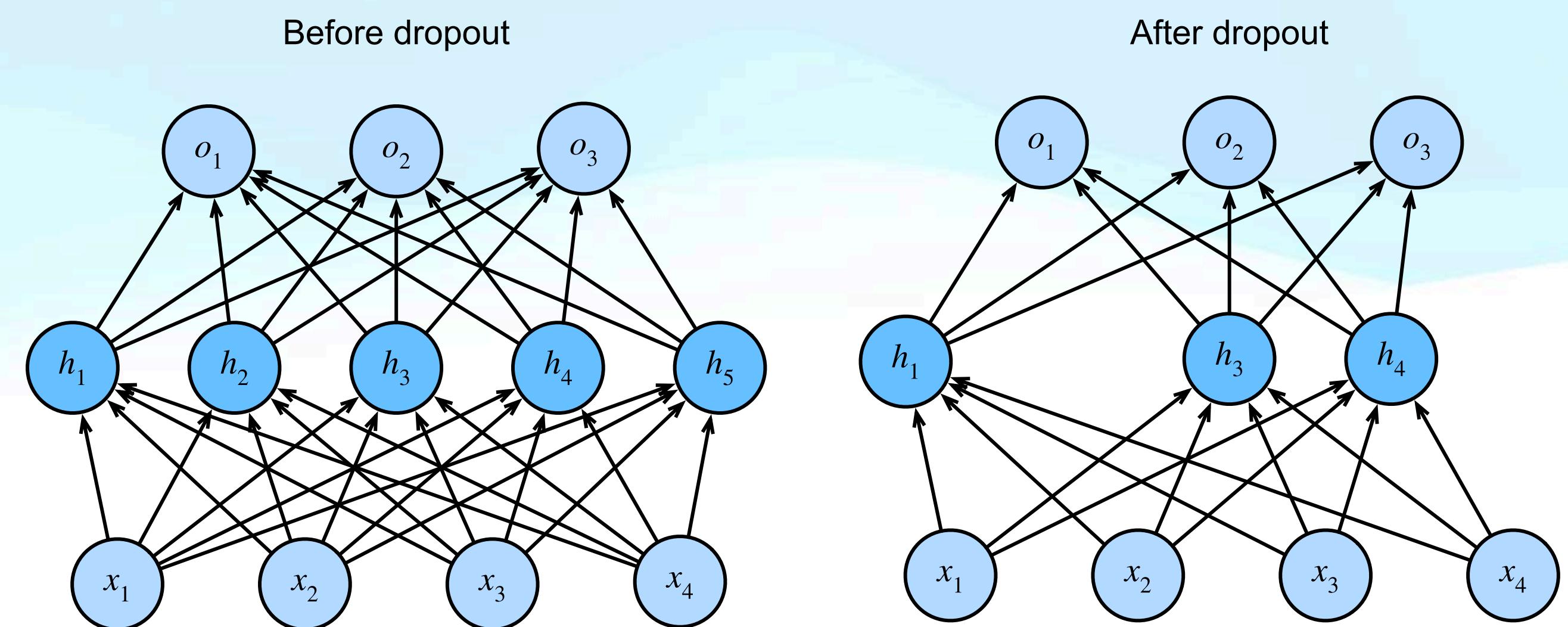
- Bu resim, **dropout (düşürme)** yönteminin nasıl çalıştığını açıklamaktadır.
- **Dropout Süreci:** Eğitim sırasında, her bir ileri geçişte her bir düğümün çıktısı, belirli bir olasılıkla (p) sıfıra ayarlanır. Bu, bazı düğümlerin rastgele sıfır yapıldığı anlamına gelir.
- **Ağ Yapısı:** Düğümlerin sıfıra ayarlanması, ağıın daha küçük bağımsız ağlar gibi çalışmasını sağlar. Bu, modelin daha sağlam hale gelmesine ve varyansı azaltarak daha sağlam tahminler yapılmasına yardımcı olur.
- **Test ve Inferans:** Eğitim sırasında dropout uygulandıktan sonra, test aşamasında tüm ağ kullanılır. Bu, daha önce eğitimde oluşturulan küçük ağların birleşimi gibi işlev görür.
- Sonuç olarak, **dropout, overfitting'i azaltmak için etkili bir yöntemdir.**



ÇOK KATMANLI ALGILAYICILAR (MLP)

Uygulamada Dropout

- Bu yapıda, bir gizli katmana dropout uyguladığımızda, her gizli birim belirli bir olasılıkla sıfırlanır. Bunun sonucunda, model artık orijinal nöronların sadece bir alt kümesiyle çalışır.
- Şekilde, h_1 ve h_2 nöronları çıkarıldığında, çıktılarının hesaplanması artık h_1 ve h_2 nöronlarına bağlı değildir ve geri yayılım sırasında bu nöronların gradyanları da sıfırlanır. Bu durum, çıktı katmanının tek bir nörona aşırı bağımlı olmasını engeller.
- Genellikle dropout'u **test zamanı** devre dışı bırakırız. Eğitim tamamlandıktan sonra yeni bir örneğe karşı model uygulanırken, nöronların hiçbirini düşürmeyez ve normalizasyon yapmamıza gerek kalmaz.
- Ancak bazı araştırmacılar, test zamanı sırasında **dropout'u** kullanarak sinir ağlarının tahminlerindeki belirsizliği tahmin etmek için bir kestirim yöntemi olarak kullanırlar.
- Eğer dropout'lu birçok farklı çıktı arasında tahminler tutarlıysa, ağın daha güvenilir olduğu söylenebilir.



ÇOK KATMANLI ALGILAYICILAR (MLP)

Dropout: Sıfırdan Uygulama

- Dropout, bir nöral ağın eğitim sürecinde bazı nöronların rastgele sıfırlanmasıdır. Bu, her bir nöronun bağımsız olarak öğrenmesini sağlayarak ortak adaptasyonu azaltır ve aşırı uyum (**overfitting**) sorununu çözer.
- Aşağıdaki kod, **dropout_layer** fonksiyonu ile dropout işlemini gerçekleştirmektedir:

```
def dropout_layer(X, dropout):
    assert 0 <= dropout <= 1
    if dropout == 1:
        return tf.zeros_like(X) # Tüm nöronları sıfırla
    mask = tf.random.uniform(shape=tf.shape(X), minval=0, maxval=1) < 1 - dropout # Rastgele maske oluştur
    return tf.cast(mask, dtype=tf.float32) * X / (1.0 - dropout) # Kalan nöronları ölçeklendir
```

ÇOK KATMANLI ALGILAYICILAR (MLP)

Dropout: Sıfırdan Uygulama

- Aşağıdaki kod örneği, **dropout_layer** fonksiyonunu kullanarak çeşitli dropout oranlarıyla giriş verisi X üzerinden dropout işlemi uygular:

```
X = tf.reshape(tf.range(16, dtype=tf.float32), (2, 8))
print('dropout_p = 0:', dropout_layer(X, 0)) # Dropout yok, tüm nöronlar aktif
print('dropout_p = 0.5:', dropout_layer(X, 0.5)) # %50 olasılıkla nöronlar sıfırlanır
print('dropout_p = 1:', dropout_layer(X, 1)) # Tüm nöronlar sıfır
```

- Kodun gösterdiği etkiler şunlardır:

- Dropout p = 0:** Tüm nöronlar aktif olduğu için, model girdi verilerini doğrudan çıkışa aktarır. Bu durumda dropout uygulanmaz ve modelin genelleme yeteneği sınırlı olabilir.
- Dropout p = 0.5:** Yüzde 50 olasılıkla nöronlar sıfırlanır. Bu, modelin daha fazla çeşitlilikle öğrenmesini sağlar ve aşırı uyumu azaltır. Ancak bazı bilgiler kaybolur.
- Dropout p = 1:** Tüm nöronlar sıfırlanır, bu da modelin çıkışının sıfır olduğu anlamına gelir. Model işlevsiz hale gelir. Bu etkiler, dropout'un modelin genel performansını ve genelleme yeteneğini nasıl etkilediğini gösterir.

ÇOK KATMANLI ALGILAYICILAR (MLP)

Dropout: Modelin Tanımlanması

- Bu kod, sıfırdan bir **Multi-Layer Perceptron (MLP)** modelini oluşturuyor ve eğitim sırasında Dropout uygulayarak aşırı öğrenmeyi önlemeyi amaçlıyor. **Modelin yapısı** şu şekildedir:

- **Giriş katmanı:** Veri alınıp işlenir ve gizli katmanlara aktarılır.
- **İki gizli katman:** Her katman, belirli bir Dropout oranıyla eğitilir.
- **Cıkış katmanı:** Modelin çıktısı hesaplanır.
- Dropout, yalnızca eğitim sırasında aktif olup, test sırasında uygulanmaz. Dropout oranları her katman için ayrı ayrı belirlenmiştir.

```
# Dropout ile sıfırdan bir MLP (Çok Katmanlı Algılayıcı) sınıfı tanımlanıyor
class DropoutMLPScratch(d2l.Classifier):

    # Modelin yapısı başlatılıyor
    def __init__(self, num_outputs, num_hiddens_1, num_hiddens_2,
                 dropout_1, dropout_2, lr):
        # Üst sınıf olan d2l.Classifier'in init fonksiyonu çağrılıyor
        super().__init__()

        # Hiperparametreler (katman sayıları, dropout oranları vb.) kaydediliyor
        self.save_hyperparameters()

        # İlk gizli katman: 'num_hiddens_1' kadar nöron içerir ve ReLU aktivasyonu kullanılır
        self.lin1 = tf.keras.layers.Dense(num_hiddens_1, activation='relu')

        # İkinci gizli katman: 'num_hiddens_2' kadar nöron içerir ve ReLU aktivasyonu kullanılır
        self.lin2 = tf.keras.layers.Dense(num_hiddens_2, activation='relu')

        # Çıkış katmanı: 'num_outputs' kadar nöron içerir, genellikle sınıflandırma için kullanılır
        self.lin3 = tf.keras.layers.Dense(num_outputs)

    # Modelin ileri yönlü geçiş fonksiyonu (veriler katmanlardan geçirilir)
    def forward(self, X):
        # Giriş verisi (X), yeniden şekillendirilip ilk gizli katmana (lin1) veriliyor
        H1 = self.lin1(tf.reshape(X, (X.shape[0], -1)))

        # Eğer eğitim aşamasındaysak, ilk gizli katmana dropout_1 oranında Dropout uygulanıyor
        if self.training:
            H1 = dropout_layer(H1, self.dropout_1)

        # İkinci gizli katmana veri gönderiliyor
        H2 = self.lin2(H1)

        # Eğer eğitim aşamasındaysak, ikinci gizli katmana dropout_2 oranında Dropout uygulanıyor
        if self.training:
            H2 = dropout_layer(H2, self.dropout_2)

        # Çıkış katmanı: H2'den gelen veriler üçüncü (çıkış) katmana aktarılıyor ve sonuç döndürülüyor
        return self.lin3(H2)
```

ÇOK KATMANLI ALGILAYICILAR (MLP)

Dropout: Eğitim

Bu kod, bir MLP modelinin hiperparametrelerini tanımlayıp, **FashionMNIST** veri kümesi üzerinde eğitilmesini sağlar. Dropout ile eğitim yapılacak olan modelin eğitimi bir eğitim döngüsü (trainer) aracılığıyla gerçekleştirilir.

Bu kodda:

- **Hiperparametreler** tanımlanır (katman büyüklükleri, Dropout oranları, öğrenme oranı).
- **MLP modeli** oluşturulur, ve her katman Dropout teknigi ile desteklenir.
- **FashionMNIST** veri seti yüklenir ve mini partilere bölünür.
- **Eğitim döngüsü (trainer)** tanımlanır ve modelin 10 epoch boyunca eğitilmesi sağlanır.

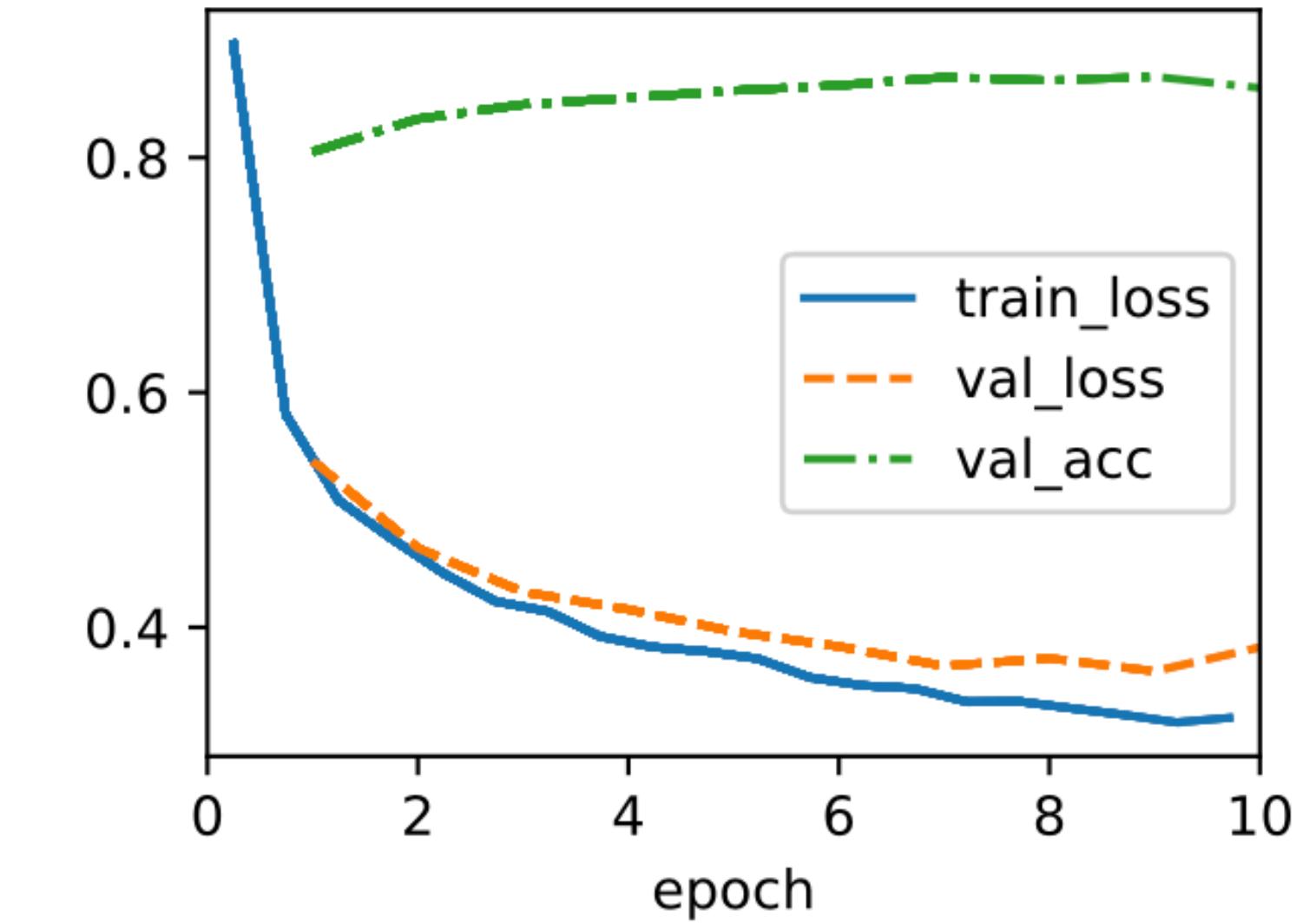
```
# Hiperparametreler bir sözlük (dictionary) içinde tanımlanıyor
hparams = {
    'num_outputs': 10,                      # Çıkış katmanındaki sınıf sayısı (FashionMNIST için 10 sınıf)
    'num_hiddens_1': 256,                   # İlk gizli katmandaki nöron sayısı
    'num_hiddens_2': 256,                   # İkinci gizli katmandaki nöron sayısı
    'dropout_1': 0.5,                      # İlk gizli katman için Dropout oranı (%50 nöron devre dışı bırakılacak)
    'dropout_2': 0.5,                      # İkinci gizli katman için Dropout oranı (%50 nöron devre dışı bırakılacak)
    'lr': 0.1                               # Öğrenme oranı (learning rate), modelin parametrelerinin ne kadar hızlı güncelleneceğini belirler
}

# Hiperparametreler kullanılarak DropoutMLPScratch model sınıfından bir model örneği oluşturuluyor
model = DropoutMLPScratch(**hparams)

# FashionMNIST veri seti kullanılıyor ve veri kümesi 256 boyutunda mini partilere (batch) ayrılıyor
data = d2l.FashionMNIST(batch_size=256)

# Eğitici (trainer) oluşturuluyor. Maksimum 10 dönem (epoch) boyunca eğitilecek
trainer = d2l.Trainer(max_epochs=10)

# Model, tanımlanan veri kümesiyle eğitici yardımıyla eğitiliyor
trainer.fit(model, data)
```



ÇOK KATMANLI ALGILAYICILAR (MLP)

Dropout: Kısa Uygulama

- Bir sonraki sayfadaki kod, **Dropout** katmanları kullanarak bir **Çok Katmanlı Algılayıcı (MLP)** modeli oluşturur. Dropout katmanları, modelin eğitim sırasında belirli bir orandaki nöronları rastgele devre dışı bırakarak aşırı öğrenmeyi (**overfitting**) önler.
- **Keras'ın Sequential API'si** ile her tam bağlı katmandan sonra bir Dropout katmanı eklenmiştir. Model, eğitim sırasında Dropout katmanlarını kullanarak veri kayıplarını azaltmaya çalışırken, test sırasında bu katmanlar devre dışı bırakılır ve veriler doğrudan aktarılır.
- Eğitim süreci için tanımlanan **trainer.fit(model, data)** komutu, modelin tanımlanan hiperparametreler ve **FashionMNIST** veri kümesi üzerinde eğitilmesini sağlar.

ÇOK KATMANLI ALGILAYICILAR (MLP)

Dropout: Kısa Uygulama

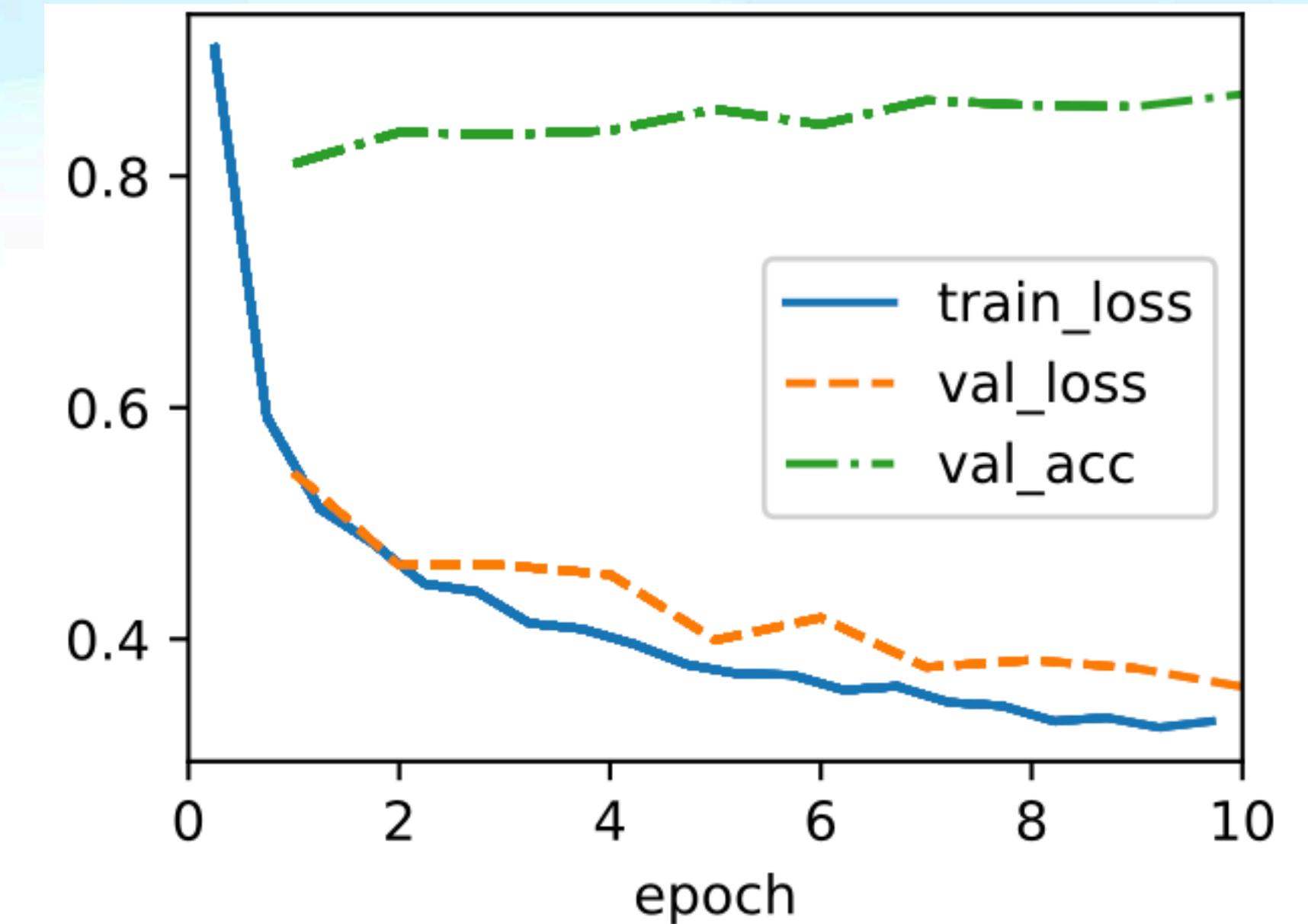
```
# Dropout ile bir MLP (Çok Katmanlı Algılayıcı) sınıfı tanımlanıyor
class DropoutMLP(d2l.Classifier):

    # Modelin yapısı başlatılıyor
    def __init__(self, num_outputs, num_hiddens_1, num_hiddens_2,
                 dropout_1, dropout_2, lr):
        # Üst sınıf olan d2l.Classifier'in init fonksiyonu çağrılıyor
        super().__init__()

        # Hiperparametreler (katman sayıları, dropout oranları vb.) kaydediliyor
        self.save_hyperparameters()

        # Sequential model tanımlanıyor, bu modelde katmanlar sırayla çalışır
        self.net = tf.keras.models.Sequential([
            # Giriş verisi tek boyutlu hale getiriliyor
            tf.keras.layers.Flatten(),
            # İlk gizli katman: 'num_hiddens_1' kadar nöron içerir ve ReLU aktivasyonu kullanılır
            tf.keras.layers.Dense(num_hiddens_1, activation=tf.nn.relu),
            # İlk gizli katmandan sonra Dropout katmanı ekleniyor, dropout_1 oranında nöronlar devre dışı bırakılacak
            tf.keras.layers.Dropout(dropout_1),
            # İkinci gizli katman: 'num_hiddens_2' kadar nöron içerir ve ReLU aktivasyonu kullanılır
            tf.keras.layers.Dense(num_hiddens_2, activation=tf.nn.relu),
            # İkinci gizli katmandan sonra Dropout katmanı ekleniyor, dropout_2 oranında nöronlar devre dışı bırakılacak
            tf.keras.layers.Dropout(dropout_2),
            # Çıkış katmanı: 'num_outputs' kadar nöron içerir, genellikle sınıflandırma için kullanılır
            tf.keras.layers.Dense(num_outputs)
        ])

    # Modelin oluşturulması ve eğitilmesi
    # Hiperparametreler tanımlanıyor
    model = DropoutMLP(**hparams)
```



ÇOK KATMANLI ALGILAYICILAR (MLP)

Predicting House Prices on Kaggle

- **D2L (Dive into Deep Learning)**, açık kaynak kodlu bir derin öğrenme kitaplığı ve eğitim materyali setidir.
- Derin öğrenmeyi hem teorik hem de uygulamalı olarak öğrenmek isteyenler için tasarlanmıştır.
- Kitaplık, çeşitli derin öğrenme modellerini hızlı bir şekilde geliştirmek ve denemek için kullanılan yüksek seviyeli API'ler sağlar.

```
!pip install d2l==1.0.0
```

```
[ ] %matplotlib inline  
import pandas as pd  
import tensorflow as tf  
from d2l import tensorflow as d2l
```

```
from google.colab import files  
uploaded = files.upload()
```

```
df1 = pd.read_csv('train.csv')
df2 = pd.read_csv('test.csv')
```

```

# İlk beş satırı görüntüleme
print("Train Veri Seti - İlk Beş Satır:")
print(df1.head())

print("\nTest Veri Seti - İlk Beş Satır:")
print(df2.head())

Train Veri Seti - İlk Beş Satır:
   Id MSSubClass MSZoning LotFrontage LotArea Street Alley LotShape \
0   1          60       RL      65.0    8450    Pave   NaN     Reg
1   2          20       RL      80.0    9600    Pave   NaN     Reg
2   3          60       RL      68.0   11250    Pave   NaN    IR1
3   4          70       RL      60.0    9550    Pave   NaN    IR1
4   5          60       RL      84.0   14260    Pave   NaN    IR1

   LandContour Utilities ... PoolArea PoolQC Fence MiscFeature MiscVal MoSold \
0        Lvl    AllPub   ...      0     NaN     NaN           NaN     0      2
1        Lvl    AllPub   ...      0     NaN     NaN           NaN     0      5
2        Lvl    AllPub   ...      0     NaN     NaN           NaN     0      9
3        Lvl    AllPub   ...      0     NaN     NaN           NaN     0      2
4        Lvl    AllPub   ...      0     NaN     NaN           NaN     0     12

   YrSold SaleType SaleCondition SalePrice
0  2008       WD      Normal  208500
1  2007       WD      Normal  181500
2  2008       WD      Normal  223500
3  2006       WD  Abnorml  140000
4  2008       WD      Normal  250000

```

```
print(f"Train Veri Setindeki toplam veri sayısı: {len(df1)}")
```

```
print(f"Test Veri Setindeki toplam veri sayısı: {len(df2)}")
```

Train Veri Setindeki toplam veri sayısı: 1460

Test Veri Setindeki toplam veri sayısı: 1459

• Bu kodda, `@d2l.add_to_class(KaggleHouse)` decorator'ü ile `KaggleHouse` sınıfına bir `preprocess` (ön işleme) fonksiyonu eklenmiştir. İşlevsel olarak bu fonksiyon, eğitim ve doğrulama veri kümelerini ön işleme tabi tutar. Veri setinde tahmin edilmesi gereken hedef değişkenin adını label olarak belirliyoruz. **Burada tahmin edeceğimiz değer "Satış Fiyatı" (`SalePrice`).Id ve SalePrice sütunlarının tahmin modeline dahil edilmek istenmemesinin temel nedenleri şunlardır:**

• Id Sütunu:

- Id, her bir satırdaki veriyi benzersiz şekilde tanımlayan bir kimlik numarasıdır ve modelin öğrenmesi gereken bir bilgi değildir.

- **Tahminle bir ilişkisi yoktur:** Id sadece bir kimlik numarası olduğu için evin fiyatını veya diğer öznitelikleri açıklayan bir anlam taşımaz. Dolayısıyla, model için bu bilgi gereksizdir.

- **Modeli yanıltabilir:** Id gibi anlamsız veya rastgele bir sütunu modele dahil edersek, model yanlış ilişkiler öğrenebilir. Modelin sadece anlamlı özniteliklerle çalışması gereklidir.

• SalePrice Sütunu:

- SalePrice, tahmin etmeye çalıştığımız hedef değişkendir (etiket).

- **Modelin öğreneceği şeydir:** SalePrice, bağımsız değişkenlerden (özelliklerden) tahmin edilmek istenen sonuç olduğu için, modelin hedef değişkeni olarak ayrı tutulur. Eğer bu değişkeni modelin girişine dahil edersek, model öğrenme sırasında "etiket" bilgisini zaten bildiğinden yanlış öğrenir. Bu da overfitting (aşırı öğrenme) gibi sorunlara yol açar.

- **features veri setindeki sayısal sütunlar tespit ediliyor.** Sonra bu sütunlar ortalaması sıfır ve standart sapması 1 olacak şekilde normalleştiriliyor.

- **Kategorik değişkenler one-hot encoding yöntemine çevriliyor.** Bu, her kategorik değişkeni binary (0/1) değerlerle temsil eder.

- **Eğitim veri seti ve doğrulama veri seti birbirinden ayrılıyor.** İlk önce birleştirilen veri setinin başındaki kısmı `self.train` (eğitim seti) olarak alınıyor ve etiket sütunu (`SalePrice`) geri ekleniyor. Kalan kısmı ise `self.val` (doğrulama seti) olarak ayrılıyor.

```
@d2l.add_to_class(KaggleHouse)
def preprocess(self):
    # Etiket sütunu
    label = 'SalePrice'

    # Id ve etiket sütunlarını kaldırıyoruz
    features = pd.concat(
        (self.raw_train.drop(columns=['Id', label]),
         self.raw_val.drop(columns=['Id'])))

    # Sayısal sütunları normalleştiriyoruz
    numeric_features = features.dtypes[features.dtypes != 'object'].index
    features[numeric_features] = features[numeric_features].apply(
        lambda x: (x - x.mean()) / (x.std()))

    # Eksik sayısal değerleri 0 ile dolduruyoruz
    features[numeric_features] = features[numeric_features].fillna(0)

    # Kategorik değişkenleri one-hot encoding'e çeviriyoruz
    features = pd.get_dummies(features, dummy_na=True)

    # İşlenmiş verileri kaydediyoruz
    self.train = features[:self.raw_train.shape[0]].copy()
    self.train[label] = self.raw_train[label]
    self.val = features[self.raw_train.shape[0]:].copy()
```

```

# EvFiyatiTahminci sınıfı tanımlanıyor.
class EvFiyatiTahminci:
    # Sınıfin yapıcı fonksiyonu, train_data'yı alarak başlatılıyor.
    def __init__(self, train_data):
        self.train = train_data # Eğitim verisi sınıf içinde kullanılmak üzere atanıyor.

    # Eğitim ve test verisi için dataloader hazırlayan fonksiyon.
    def get_dataloader(self, data, train=True):
        if train:
            # Eğitim verisi için özellikler (X) ve hedef değişken (Y) hazırlanıyor.
            label = 'SalePrice' # Bu satırda hedef değişken (SalePrice) etiketleniyor.
            get_tensor = lambda x: tf.constant(x.values.astype(float), dtype=tf.float32) # Verileri tensörlerde dönüştürme işlevi.
            tensors = (
                get_tensor(data.drop(columns=[label])), # SalePrice sütunu dışındaki tüm sütunlar özellik olarak alınıyor.
                tf.reshape(tf.math.log(get_tensor(data[label])), (-1, 1)) # SalePrice'in logaritması hedef değişken olarak ayarlanıyor.
            )
        else:
            # Test verisi için yalnızca özellikler alınıyor, çünkü hedef değişken yok.
            get_tensor = lambda x: tf.constant(x.values.astype(float), dtype=tf.float32)
            tensors = (get_tensor(data), None) # Test verisinde hedef olmayacağı için ikinci değer None.

        # Veri yükleyici oluşturuluyor.
        return self.get_tensorloader(tensors, train)

```

- **DataLoader fonksiyonu**, özellikle derin öğrenme ve makine öğrenmesi modelleri için veri yükleme sürecini yönetmek amacıyla kullanılır. Bu terim genellikle PyTorch ve TensorFlow gibi kütüphanelerde sıkça kullanılır. **DataLoader, veriyi bu mini-batch'lere böler ve modelin bu gruplar üzerinde eğitim yapmasını sağlar.**
- SalePrice sütununun logaritmasının hedef değişken olarak kullanılmasının temel sebebi, veri dönüşümüne ihtiyaç duyulmasıdır. Bu dönüşüm genellikle makine öğrenmesi ve özellikle regresyon problemlerinde bazı avantajlar sağlar.

- Bu kod, veriyi **K katmanlı (K-fold) çapraz doğrulama** için hazırlar.

```
# TensorFlow Dataset API kullanılarak veriler yükleniyor.
def get_tensorloader(self, tensors, train):
    X, y = tensors # Özellikler ve hedef değişken alınıyor.
    dataset = tf.data.Dataset.from_tensor_slices((X, y)) if train else tf.data.Dataset.from_tensor_slices(X) # Veri kümesi oluşturuluyor.
    return dataset.shuffle(1000).batch(32) if train else dataset.batch(32) # Eğitim için karıştırma ve batch işlemi yapılıyor.

# K-fold çapraz doğrulama için veriler k parçaya bölünüyor.
def k_fold_data(self, k):
    rets = [] # Sonuçları saklamak için bir liste oluşturuluyor.
    fold_size = len(self.train) // k # Her bir fold'un boyutu hesaplanıyor.
    indices = np.arange(len(self.train)) # Eğitim verisinin indeksleri alınıyor.
    np.random.shuffle(indices) # İndeksler karıştırılıyor.

    # Veriler k parçaya bölünüyor.
    for j in range(k):
        val_indices = indices[j * fold_size: (j + 1) * fold_size] # Bu fold için validasyon verisinin indeksleri alınıyor.
        train_indices = np.delete(indices, np.s_[j * fold_size: (j + 1) * fold_size]) # Eğitim verisi indeksleri.
        val_data = self.train.iloc[val_indices] # Validasyon verisi seçiliyor.
        train_data = self.train.iloc[train_indices] # Eğitim verisi seçiliyor.
        rets.append((train_data, val_data)) # Eğitim ve validasyon verisi sonuçlara ekleniyor.

    return rets # Tüm foldlar döndürülüyor.
```

- **K-fold Çapraz Doğrulama:** Modelin her bir katmanını eğitim ve validasyon verileri üzerinde değerlendirerek, modelin genelleme yeteneğini daha sağlam bir şekilde analiz eder.

```
# K-fold çapraz doğrulama yöntemiyle model eğitimi yapılıyor.
def k_fold(self, model_class, k, lr):
    val_losses = [] # Validasyon kayıplarını saklamak için liste.
    models = [] # Eğitilen modelleri saklamak için liste.
    for i, (train_fold, val_fold) in enumerate(self.k_fold_data(k)): # K-fold verileri döngüye sokuluyor.
        model = model_class(lr) # Model oluşturuluyor.
        train_loader = self.get_dataloader(train_fold, train=True) # Eğitim verisi için dataloader oluşturuluyor.
        val_loader = self.get_dataloader(val_fold, train=True) # Validasyon verisi için dataloader oluşturuluyor.

        # Model eğitim süreci başlatılıyor.
        model.fit(train_loader) # Model, eğitim verisi ile eğitiliyor.
        val_loss = model.evaluate(val_loader) # Validasyon kaybı hesaplanıyor.
        val_losses.append(val_loss) # Validasyon kaybı kaydediliyor.
        models.append(model) # Eğitilen model listeye ekleniyor.

    # K-fold validasyon ortalaması ekrana yazdırılıyor.
    print(f'ortalama validasyon log mse = {np.mean(val_losses)}')
return models # Eğitilen modeller döndürülüyor.
```

- Modelin nasıl yapılandırılacağını, eğitileceğini ve tahminler yapacağını tanımlar.

• Adam Optimizasyonu

- **Adam (Adaptive Moment Estimation)**, Stochastic Gradient Descent (**SGD**) optimizasyon algoritmasının bir çeşididir.
- Öğrenme oranlarını otomatik olarak ayarlayarak, modelin daha hızlı ve etkili bir şekilde öğrenmesini sağlar.
- **MSE Kaybı (Mean Squared Error)**

- **MSE**, tahmin edilen değerler ile gerçek değerler arasındaki farkların karelerinin ortalamasıdır. Regresyon problemlerinde yaygın olarak kullanılan bir **kayıp fonksiyonudur**.

```
# Basit bir model sınıfı tanımlanıyor.
class BasitModel:
    # Model sınıfının yapıcı fonksiyonu.
    def __init__(self, lr):
        self.lr = lr # Öğrenme oranı atanıyor.
        # Modelde bir tam bağlantılı katman tanımlanıyor.
        self.model = tf.keras.Sequential([
            tf.keras.layers.Dense(1) # Tek bir çıkış nöronuna sahip bir katman.
        ])
        # Model derleniyor; Adam optimizasyonu ve MSE kaybı kullanılıyor.
        self.model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=lr), loss='mse')

    # Modelin eğitim fonksiyonu.
    def fit(self, train_loader):
        for X, y in train_loader: # Eğitim veri yükleyicisinden özellikler ve hedef alınarak döngü başlatılıyor.
            self.model.fit(X, y, epochs=10, verbose=0) # Model eğitim verisi ile eğitiliyor.

    # Modelin validasyon verisi üzerinde değerlendirilmesi.
    def evaluate(self, val_loader):
        losses = [] # Kayıpları tutmak için liste.
        for X, y in val_loader: # Validasyon veri yükleyicisinden özellikler ve hedef alınarak döngü başlatılıyor.
            loss = self.model.evaluate(X, y, verbose=0) # Validasyon kaybı hesaplanıyor.
            losses.append(loss) # Kayıp kaydediliyor.
        return np.mean(losses) # Kayıpların ortalaması döndürülüyor.

    # Modelin test verisi üzerinde tahmin yapması.
    def predict(self, test_loader):
        preds = [] # Tahminleri tutmak için liste.
        for X in test_loader: # Test verileri yükleniyor.
            preds.append(self.model.predict(X)) # Model, test verisi üzerinde tahmin yapıyor.
        return np.concatenate(preds) # Tüm tahminler birleştiriliyor ve döndürülüyor.
```

- **k=5 parametresi ile K-fold çapraz doğrulama uygulanıyor.** Burada 5, veri kümесinin 5 katmana bölüneceğini belirtir. Her katman, eğitim ve validasyon setleri için ayrı ayrı kullanılacak ve bu süreç toplam 5 kez tekrarlanacaktır. 5 katman, veri setinin boyutuna bağlı olarak yeterli sayıda eğitim ve validasyon döngüsü sağlar. Daha fazla katman (örneğin, 10) kullanmak, daha fazla eğitim süresi gerektirebilir ve bazı durumlarda **aşırı uyuma yol açabilir**.
- Model tahminleri genellikle logaritmik ölçekte (**örneğin, log(SalePrice)**) elde edilir. Bu nedenle, tahminlerin geri çevrilmesi için üstel (**exponential**) işlem uygulanır. Bu işlem, modelin tahminlerinin logaritmik ölçekten orijinal ölçüye geri dönmesini sağlar.

```
# Eğitimi başlatmak için EvFiyatiTahminci sınıfı kullanılıyor.
ev_tahminci = EvFiyatiTahminci(train_data) # Eğitim verisi kullanılarak sınıf başlatılıyor.
trainer = ev_tahminci.k_fold(BasitModel, k=5, lr=0.01) # K-fold çapraz doğrulama yapılıyor.

# Tahmin yapmak için test verisi yükleyicisi oluşturuluyor.
test_loader = ev_tahminci.get_dataloader(test_data, train=False)

# Tüm eğitilen modellerle test verisi üzerinde tahmin yapılıyor.
preds = [model.predict(test_loader) for model in trainer]

# Logaritmik ölçekteki tahminler üssel olarak geri çevriliyor.
ensemble_preds = tf.reduce_mean(tf.exp(tf.concat(preds, axis=1)), axis=1)

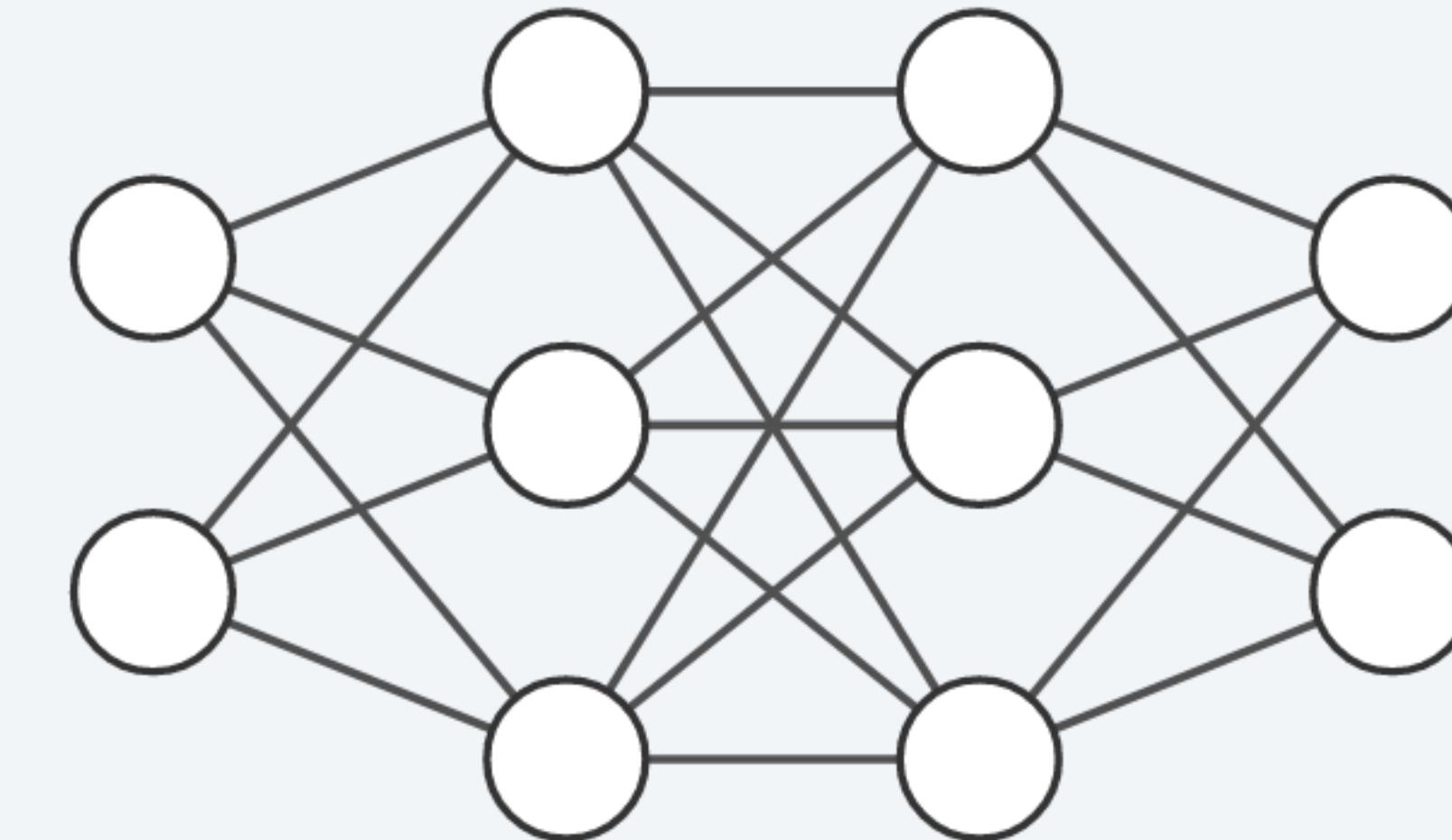
# Sonuçları CSV dosyasına kaydetme işlemi.
submission = pd.DataFrame({
    'SalePrice': ensemble_preds.numpy() # Tahmin sonuçları SalePrice sütununa yazılıyor.
})
submission.to_csv('submission.csv', index=False) # Sonuçlar submission.csv dosyasına kaydediliyor.
```

	SalePrice,Id
1	124735.64,1461
2	203180.66,1462
3	185279.16,1463
4	208724.77,1464
5	217064.66,1465
6	172091.72,1466
7	196622.67,1467
8	158229.02,1468
9	218038.05,1469
10	122390.234,1470
11	171996.97,1471
12	109810.125,1472
13	101917.28,1473
14	150152.03,1474
15	109737.92,1475
16	347100.9,1476
17	225638.2,1477
18	271473.3,1478
19	300697.03,1479
20	511560.94,1480
21	310430.25,1481
22	213157.2,1482
23	163362.84,1483
24	159163.38,1484
25	186363.47,1485
26	194770.05,1486
27	366530.84,1487
28	238719.33,1488
29	194601.78,1489
30	245836.47,1490
31	200373.61,1491
32	101674.22,1492
33	210859.6,1493
34	250281.47,1494
35	285594.53,1495
36	274234.4,1496

KAYNAKÇA

- <https://medium.com/@baristuzemenn/mlp-%C3%A7ok-katman%C4%B1-alg%C4%B1lay%C4%B1c%C4%B1-yapay-sinir-a%C4%9Flar%C4%B1-a8c9c68748f6>
- https://d2l.ai/chapter_multilayer-perceptrons/index.html
- <https://medium.com/deeper-deep-learning-tr/aktivasyon-fonksiyonu-120769040f7b>
- <https://ayyucekizrak.medium.com/derin-%C3%B6%C4%9Frenme-i%C3%A7in-aktivasyon-fonksyonlar%C4%B1n%C4%B1n-kar%C5%9F%C4%B1la%C5%9Ft%C4%B1r%C4%B1mas%C4%B1-cee17fd1d9cd>
- https://github.com/dsgiitr/d2l-pytorch/blob/master/Ch06_Multilayer_Perceptrons/Numerical_Stability_and_Initialization.ipynb
- <https://medium.com/@kushansharma1/vanishing-exploding-gradients-problem-1901bb2db2b2>
- <https://arxiv.org/abs/2409.06402>
- https://www.google.com/url?sa=i&url=https://www.youtube.com/watch?v=CzQHw3fcxQE&psig=AOvVaw3FM6_TG6fjmnF08DEKI79Z&ust=1728669109403000&source=images&cd=vfe&opi=89978449&ved=0CBcQjhxqFwoTCOi1m7iwhIkDFQAAAAAdAAAAABAE
- https://cedar.buffalo.edu/~srihari/CSE676/8.4_ParInitializn.pdf
- <https://magnimindacademy.com/blog/what-is-generalization-in-machine-learning/>
- <https://www.veribilimiokulu.com/overfitting/>
- <https://medium.com/@oran.yasemin/deep-learningde-a%C5%9F%C4%9F%C4%9Frenmeyi%C4%9F-%C4%9F%C4%9Fleme-etcili%C4%9F-y%C4%9Fntemler-ve-uygulamalar-35e53439a80a>
- <https://medium.com/@oran.yasemin/d%C5%BCenlile%C5%9Fitirme-regularization-teknikleri-ile-a%C5%9F%C4%9F%C4%9Frenmeyi%C4%9F-%C4%9F%C4%9Fleme-l1-ve-l2-d%C5%BCenlile%C5%9Fitirme-9afc04e624e3>
- <https://chatgpt.com/>
- <https://www.kaggle.com/>
- <https://www.pycodemates.com/2023/01/multi-layer-perceptron-a-complete-overview.html>

Multi-layer Perceptron



BİZİ DİNLEDİĞİNİZ İÇİN
TEŞEKKÜR EDERİZ