SAKARYA ÜNİVERSİTESİ

Derin Öğrenmeye Giriş

HAFTA 3

Forward Propagation (ileri yönlü yayılım)

Backpropagation (geri yayılım)

Gradyan İniş

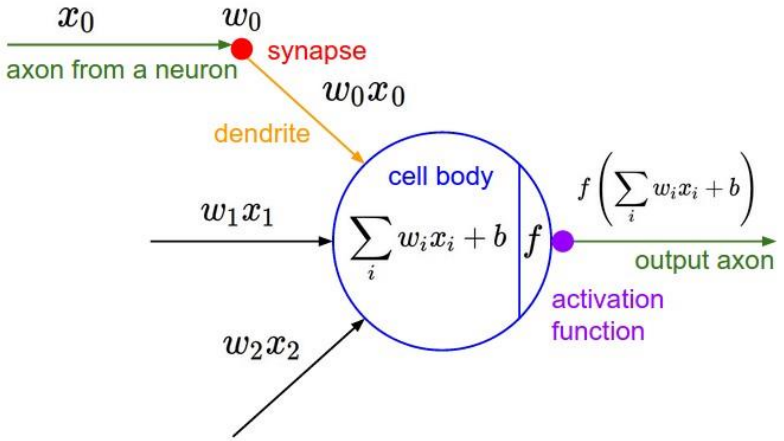
Kayıp Fonksiyonu

Dr. Öğretim Üyesi Burcu ÇARKLI YAVUZ

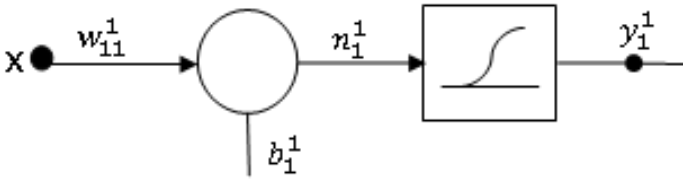
bcarkli@sakarya.edu.tr

Yapay Sinir Ağlarını Hatırlayalım:

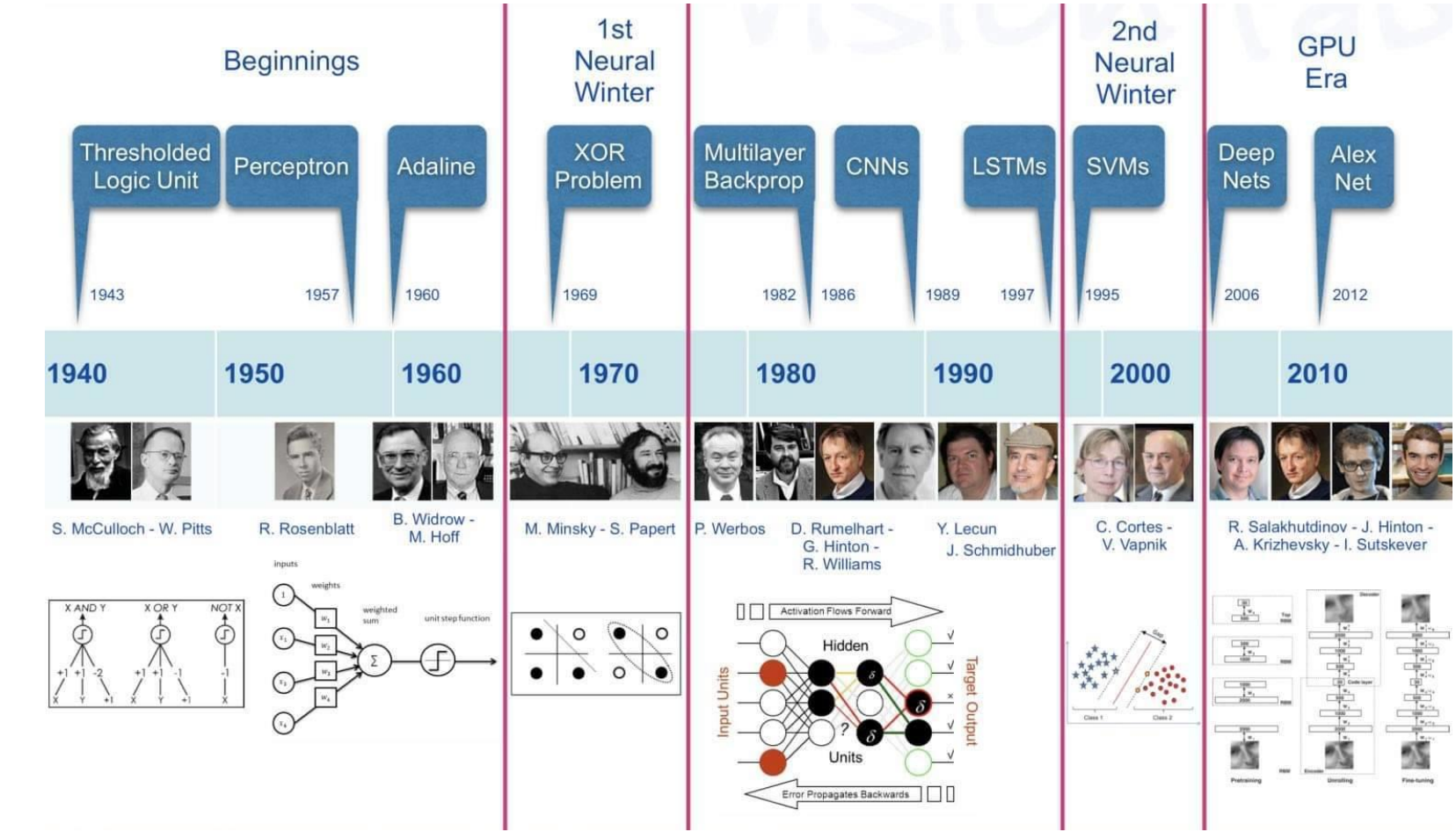
- **Tanım:** Yapay sinir ağları, insan beyninin bilgi işleme yeteneğini taklit etmeyi amaçlayan bilgisayar sistemleridir.
- **Ana Bileşenler:** Nöronlar (yapay hücreler), katmanlar (giriş, gizli, çıkış) ve ağırlıklar (nöronlar arası bağlantı gücü) yapay sinir ağlarının temel yapı taşlarıdır.
- **Nöronlar:** Yapay nöronlar, gerçek nöronların işlevlerini taklit eden ve sinyalleri işleyen matematiksel fonksiyonlardır.
- **Katmanlar:** Giriş, gizli ve çıkış katmanlarından oluşur. Her katman, belirli bir işlemi gerçekleştirmek için nöronlar içerir.
- **Ağırlıklar:** Nöronlar arasındaki bağlantıların gücünü temsil eder. Ağırlıklar, öğrenme sürecinde sürekli olarak güncellenir.



Kaynak: http://cs231n.github.io/convolutional-networks/



Yapay Sinir Ağlarının Tarihçesi



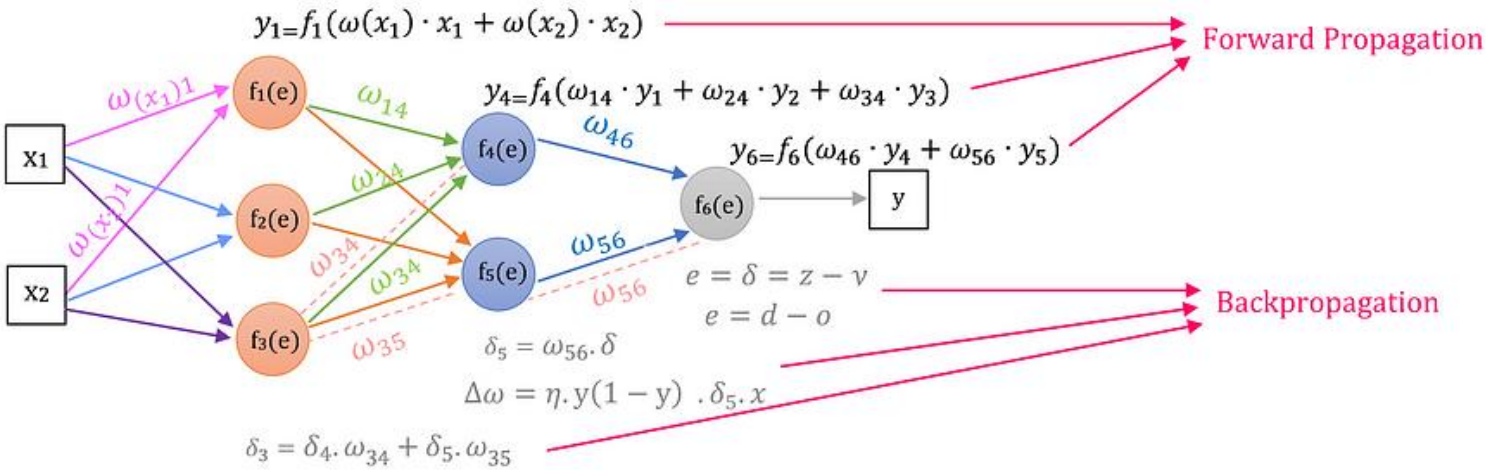
Yapay Sinir Ağlarının Tarihçesi
Önemli Gelişmeler ve Dönüm Noktaları

1.McCulloch ve Pitts (1943): Warren McCulloch ve Walter Pitts, sinir ağlarının mantıksal hesaplamaları nasıl gerçekleştirebileceğini açıklayan bir model yayınladılar. Bu model, yapay sinir ağlarının teorik temellerini oluşturdu.
2.Hebbian Öğrenme (1949): Donald Hebb, "The Organization of Behavior" adlı kitabında, sinir hücrelerinin birbirleriyle nasıl etkileşime girdiğine dair bir teori olan Hebbian öğrenmeyi tanıttı. Bu teori, yapay sinir ağlarının öğrenme algoritmaları için temel oluşturdu.
3.Perceptron (1958): Frank Rosenblatt, perceptron adı verilen ve basit öğrenme yeteneklerine sahip bir yapay sinir ağı modeli geliştirdi. Perceptron, yapay zekâ araştırmalarında önemli bir adım oldu.
4.XOR Problemi ve AI Kışı (1969): Marvin Minsky ve Seymour Papert, "Perceptrons" adlı kitaplarında, tek katmanlı perceptronların XOR gibi bazı basit problemleri çözemediğini gösterdiler. Bu, yapay zekâ araştırmalarında bir durgunluğa neden olan "AI Kışı"nın başlangıcı oldu.

Yapay Sinir Ağlarının Tarihçesi
Önemli Gelişmeler ve Dönüm Noktaları

5.Backpropagation (1986): David Rumelhart, Geoffrey Hinton ve Ronald Williams, çok katmanlı sinir ağlarının eğitimi için geri yayılım (backpropagation) algoritmasını tanıttı. Bu, yapay sinir ağlarının karmaşık problemleri çözebilmesi için bir dönüm noktası oldu.
6.Convolutional Neural Networks (1998): Yann LeCun ve ekibi, el yazısı rakamlarını tanıyan bir convolutional neural network (CNN) olan LeNet-5'i geliştirdi. CNN'ler, görüntü işleme ve bilgisayarlı görüde devrim yarattı.
7.Deep Learning (2006): Geoffrey Hinton ve ekibi, derin öğrenme adını verdikleri çok katmanlı sinir ağlarının etkili bir şekilde eğitilmesi üzerine bir makale yayınladı. Bu, yapay zekâ araştırmalarında yeni bir dönem başlattı.
8.ImageNet Challenge (2012): Alex Krizhevsky, Ilya Sutskever ve Geoffrey Hinton tarafından geliştirilen AlexNet, ImageNet Large Scale Visual Recognition Challenge'da (ILSVRC) rekor bir başarı elde etti. Bu olay, derin öğrenme tekniklerinin popülaritesinde büyük bir artışa yol açtı.

Yapay Sinir Ağlarının Çalışma Mekanizması

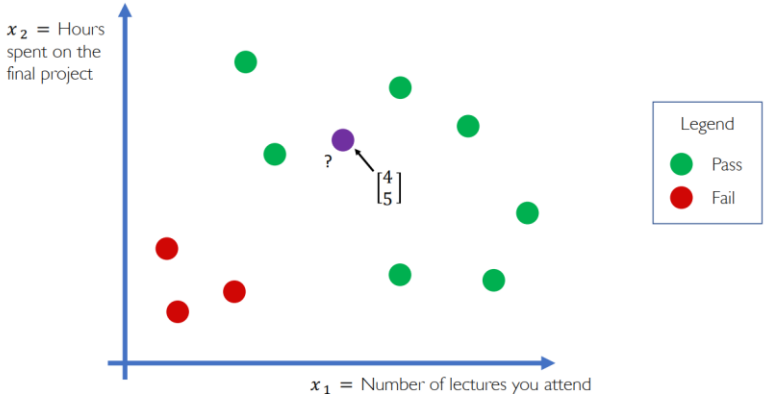


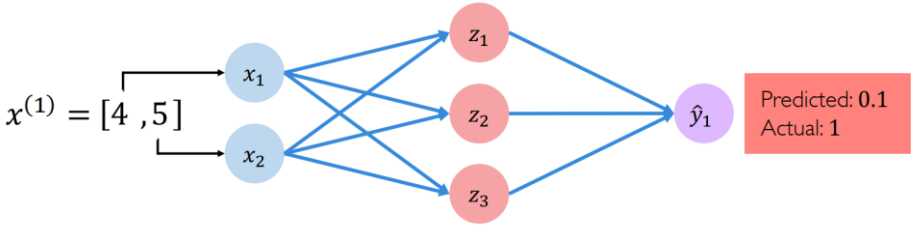
- **İleri yayılım (forward propagation),** sinir ağındaki girdi katmanından (sol) çıktı katmanına (sağ) geçmenin yoludur.
- Çıktı katmanından girdi katmanına sağdan sola yani geriye doğru hareket etme işlemine **geri yayılım(backpropagation)** denir. Geriye yayılım, minimum kayıp fonksiyonuna ulaşmak için ağırlıkları ayarlamak veya düzeltmek için tercih edilen yöntemdir.
- **Hata Fonksiyonu:** Gerçek çıktı ile beklenen çıktı arasındaki farkı ölçer. Ağırlıklar, bu hatayı minimize edecek şekilde ayarlanır.
- **Optimizasyon:** Gradient descent gibi optimizasyon yöntemleri, en düşük hata değerine ulaşmak için ağırlıkları günceller.

https://medium.com/@gulsahseher/yapay-sinir-aglari-artificial-neural-network-ann-af78c7b8f22e

Yapay Sinir Ağlarının Uygulanması

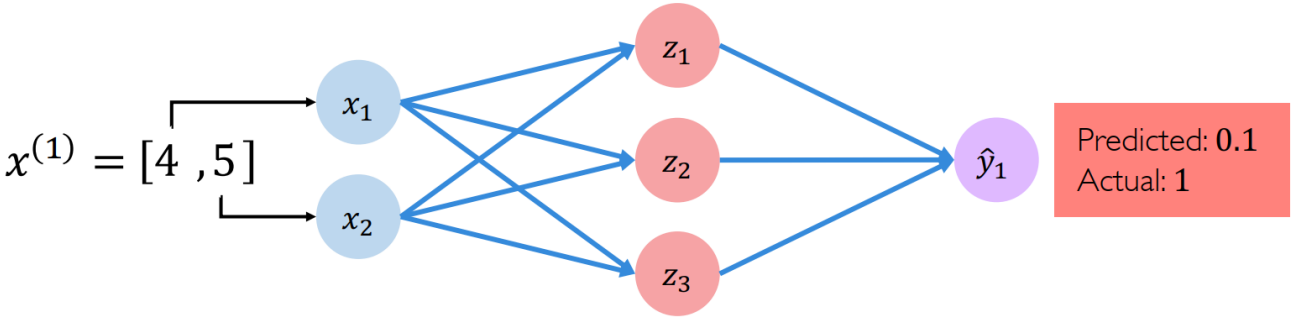
Example Problem: Will I pass this class?





Kaybın ölçülmesi

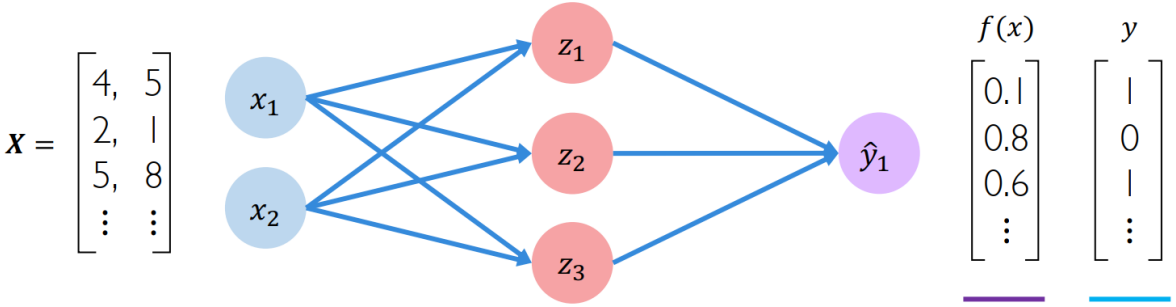
*The **loss** of our network measures the cost incurred from incorrect predictions*



$$\mathcal{L}\left(\underbrace{f\left(x^{(i)}; \boldsymbol{W}\right)}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}}\right)$$

Kaybın ölçülmesi

*The **empirical loss** measures the total loss over our entire dataset*

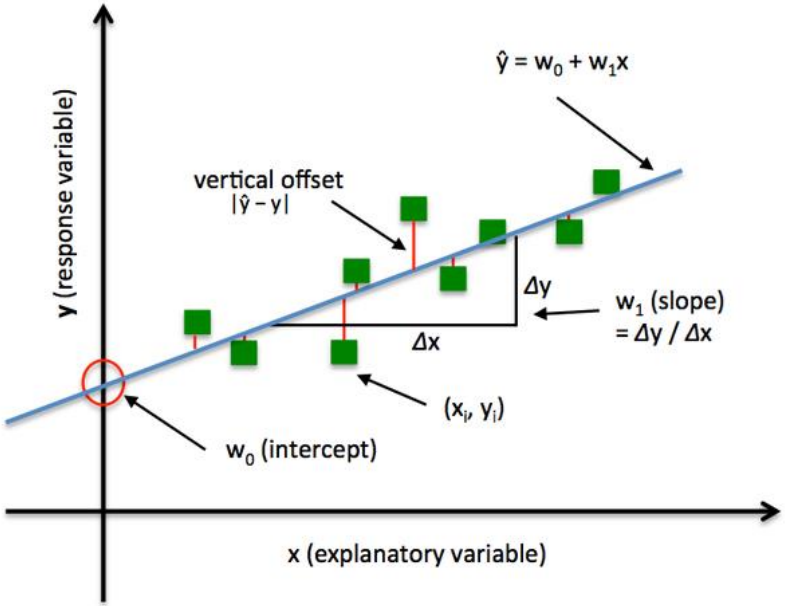


Also known as:
- Objective function
- Cost function
- Empirical Risk

$$\boldsymbol{J(W)} = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

Predicted Actual

Doğrusal regresyon



En Küçük Kareler Doğrusal Regresyonunda hedefimiz, dikey ofsetleri en aza indiren çizgiyi bulmaktır. Ya da başka bir deyişle, en uygun çizgiyi, hedef değişkenimiz (y) ile bizim i'deki tüm örnekler üzerinde öngörülen çıktımız arasındaki **kare hataların (SSE)** veya **ortalama kare hataların (MSE)** toplamını en aza indiren çizgi olarak tanımlarız.

$$y = w_0 x_0 + w_1 x_1 + \ldots + w_m x_m = \sum_{i=1}^{n} = \mathbf{w}^T \mathbf{x}$$

Doğrusal regresyon

- En küçük kareler regresyonunu gerçekleştirmek için aşağıdaki yaklaşımlardan birini kullanarak doğrusal bir regresyon modeli uygulayabiliriz:

$$SSE = \sum_i (\text{target}^{(i)} - \text{output}^{(i)})^2$$

$$MSE = \frac{1}{n} \times SSE$$

- Model parametrelerinin analitik olarak çözülmesi (kapalı form denklemleri)
- Optimizasyon algoritmasının uygulanması (Gradient Descent, Stochastic Gradient Descent, Newton's Method, Simplex Method, vb.)

Gradient Descent (GD)

- Gradient Decent (GD) optimizasyon algoritması kullanılarak, ağırlıklar her periyot (epoch) sonrasında artımlı olarak güncellenir (= eğitim veri setini kullan).
- Karesel hatalarının toplamı (Sum of Squared Error-SSE) olarak maliyet (**loss function, cost**) fonksiyonu J(.), şöyle yazılabilir:

$$J(\mathbf{w}) = \frac{1}{2}\sum_{i}(\text{target}^{(i)} - \text{output}^{(i)})^2$$

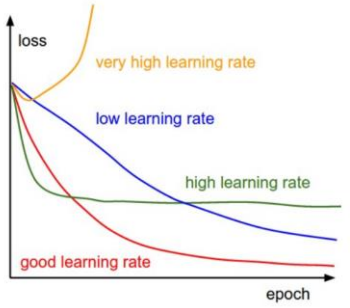
Gradient Descent (GD)

- Ağırlık güncellemesinin büyüklüğü ve yönü, maliyet gradyanının ters yönünde bir adım atılarak hesaplanır:

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j},$$

- **η** (yada α) öğrenme oranıdır (learning rate).
- Ağırlıklar daha sonra her belirlenen aralıkta aşağıdaki güncelleme kuralı ile güncellenir:

$$\mathbf{w} := \mathbf{w} + \Delta \mathbf{w},$$



Kaynak: https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3

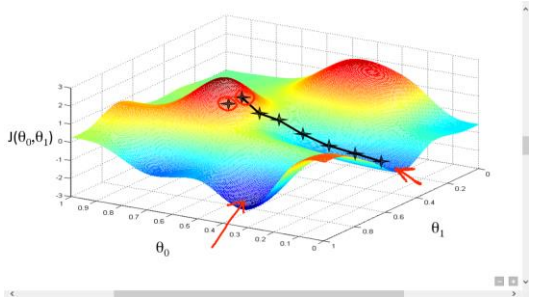
Gradient Descent (GD)

- burada Δw, her bir ağırlık katsayısının w, aşağıdaki gibi hesaplanan ağırlık güncellemelerini içeren bir vektördür;

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j}$$

$$= -\eta \sum_i (\text{target}^{(i)} - \text{output}^{(i)})(-x_j^{(i)})$$

$$= \eta \sum_i (\text{target}^{(i)} - \text{output}^{(i)}) x_j^{(i)}.$$

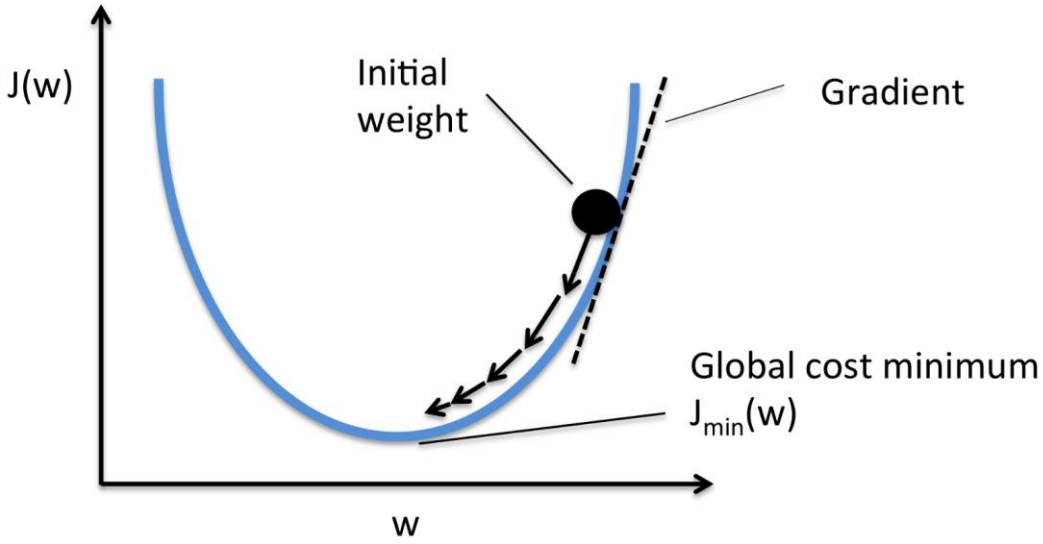


https://hackernoon.com/gradient-descent-aynk-7cbe95a778da

- Temel olarak, GD optimizasyonunu bir dağdan aşağıya (maliyet fonksiyonu) bir vadiye gitmek isteyen (minimum maliyet) bir yürüyüşçü (ağırlık katsayısı) olarak görebiliriz.
- Her adım, eğimin dikliği (eğim-gradient) ve yürüyüşçünün bacak uzunluğu (öğrenme oranı-learning rate) tarafından belirlenir.

Gradient Descent (GD)

- Yalnızca tek bir ağırlık katsayısına sahip bir maliyet fonksiyonu göz önüne alındığında, bu kavramı aşağıdaki gibi gösterebiliriz:



Stochastic Gradient Descent (SGD)

- GD optimizasyonunda, eğitim setindeki tüm veriler için maliyet hesaplanır. Bu nedenle, **yığın (batch) GD** olarak da adlandırılır.
- Çok büyük veri setleri olması durumunda, GD'yi kullanmak hesaplama yükü oldukça artırabilir. Çünkü ağırlıkların bir kez güncellenmesi için tüm eğitim setinin hataya katkısının hesaplaması gerekiyor.
- Bu nedenle, eğitim seti büyüdükçe, algoritma yavaşlar, ağırlıkları uzun sürede günceller ve global minimuma daha yavaş yaklaşır.

Stochastic Gradient Descent (SGD)

- for one or more epochs:
  - for each weight j
    - $w_j := w + \Delta w_j$, where: $\Delta w_j = \eta \sum_i (\text{target}^{(i)} - \text{output}^{(i)}) x_j^{(i)}$

➢GD için ağırlık güncellemeleri yukarıdaki gibi toplanır.

➢**SGD** algoritmasında ise *her eğitim örneğinden sonra ağırlıklar güncellenir*.

- for one or more epochs, or until approx. cost minimum is reached:
  - for training sample i:
    - for each weight j
      - $w_j := w + \Delta w_j$, where: $\Delta w_j = \eta (\text{target}^{(i)} - \text{output}^{(i)}) x_j^{(i)}$

SGD; bazen yinelemeli (iterative) veya çevrimiçi (online) GD olarak da adlandırılır

Stochastic Gradient Descent (SGD)

- Burada "stokastik" terimi, tek bir eğitim örneğine dayalı eğimin "gerçek" maliyet eğiminin "stokastik bir yaklaşımı" olması gerçeğinden kaynaklanmaktadır.
- Stokastik doğasından dolayı, küresel minimum maliyete giden yol GD'deki gibi "doğrudan" değildir, ancak maliyet yüzeyini 2 boyutlu bir alanda görselleştiriyorsak "zig-zag" gidebilir.
- Adaptif bir öğrenme hızı η için, zaman içindeki öğrenme oranını azaltan bir azalma sabiti d seçilir, (t iterasyon numarası):

$$\eta(t+1) := \eta(t)/(1 + t \times d)$$

- Daha hızlı güncellemeler için ağırlık güncellemesine önceki eğimin bir faktörünü ekleyerek momentum öğrenimi:

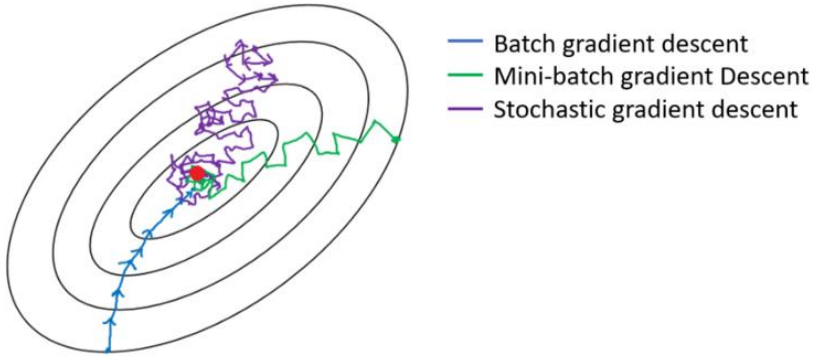
$$\Delta \mathbf{w}_{t+1} := \eta \nabla J(\mathbf{w}_{t+1}) + \alpha \Delta w_t \qquad \mathbf{w} := \mathbf{w} + \Delta \mathbf{w},$$

Mini Batch Gradient Descent (MB-GD)

- Mini Toplu Gradyan İnişi (MB-GD), toplu GD ve SGD arasında bir uzlaşmadır.
- MB-GD'de modeli daha küçük eğitim örnek gruplarına göre güncelliyoruz; 1 örnekten (SGD) veya tüm n eğitim örneğinden (GD) gelen gradyanı hesaplamak yerine, 1 < k < n eğitim örneğinden gelen gradyanı hesaplıyoruz (yaygın bir mini-batch boyutu k=50'dir).
- MB-GD, ağırlıkları daha sık güncellediğimiz için GD'ye göre daha az yinelemede yakınsar; ancak MB-GD, vektörleştirilmiş işlemi kullanmamıza izin verir, bu da genellikle SGD'ye göre hesaplama performansı artışıyla sonuçlanır.

Veri kullanımı:
•GD: Tüm veri seti
•SGD: Her seferinde bir örnek
•Mini-batch GD: Veri setinin küçük alt kümeleri
Hesaplama hızı: SGD > Mini-batch GD > GD
Gradyan tahmini doğruluğu: GD > Mini-batch GD > SGD
Yakınsama kararlılığı: GD > Mini-batch GD > SGD
Yerel minimumlardan kaçınma: SGD > Mini-batch GD > GD

https://sebastianraschka.com/faq/docs/closed-form-vs-gd.html



GD, SGD, MBGD avantajları ve dezavantajları

Gradient Descent (GD): Avantajları:

- Tüm veri setini kullanarak daha doğru gradyan hesaplaması yapar.
- Genellikle daha kararlı ve pürüzsüz bir yakınsama sağlar.

Dezavantajları:

- Büyük veri setleri için hesaplama açısından pahalıdır.
- Her iterasyonda tüm veri setini işlediği için yavaştır.
- Yerel minimumlara takılma riski vardır.

Stochastic Gradient Descent (SGD): Avantajları:

- Her iterasyonda sadece bir örnek kullanıldığı için çok hızlıdır.
- Büyük veri setleri için uygundur.
- Yerel minimumlardan kaçma potansiyeli daha yüksektir.

Dezavantajları:

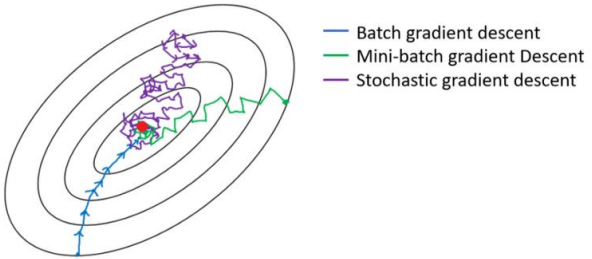
- Gradyan tahmini gürültülüdür, bu da yakınsamayı dalgalı hale getirebilir.
- Optimum çözüme ulaşmak için daha fazla iterasyon gerektirebilir.
- Öğrenme oranının dikkatli ayarlanması gerekir.

Mini-Batch Gradient Descent: Avantajları:

- GD ve SGD arasında bir denge sağlar.
- SGD'den daha kararlı yakınsama, GD'den daha hızlı hesaplama sunar.
- Paralel işleme için uygundur.

Dezavantajları:

- Batch boyutunun doğru seçilmesi gerekir.
- SGD kadar hızlı değildir, GD kadar kararlı da değildir.



Gradient Descent Algoritmalar

Batch Gradient Descent

```
for i in range(num_epochs):
    grad = compute_gradient(dataset, params)
    params = params – learning_rate * grad
```

Stochastic Gradient Descent

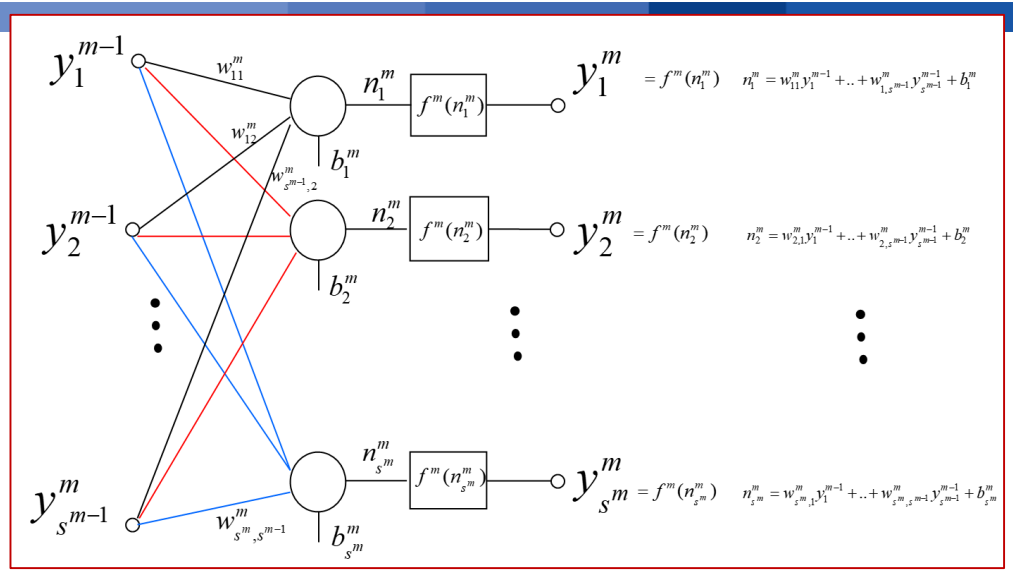
```
for i in range(num_epochs):
    np.random.shuffle(dataset)
    for example in dataset:
        grad = compute_gradient(example, params)
        params = params – learning_rate * grad
```

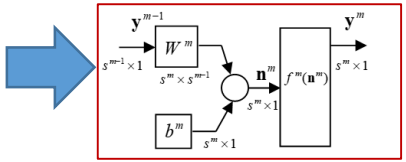
Mini-batch Gradient Descent

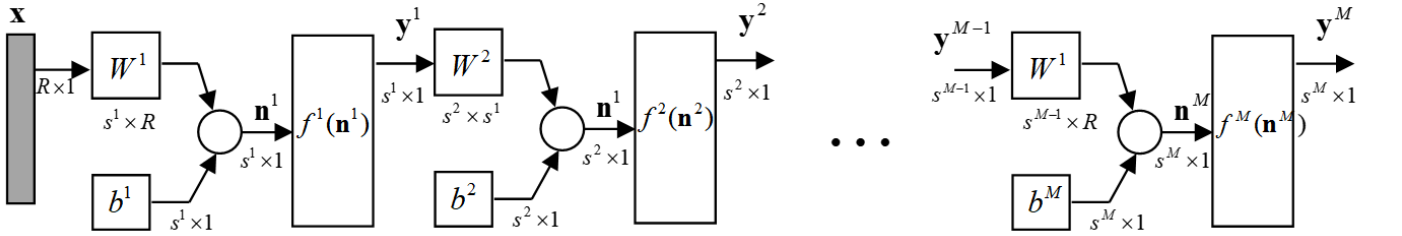
```
for i in range(num_epochs):
    np.random.shuffle(dataset)
    for batch in radom_minibatches(data, batch_size=32):
        grad = compute_gradient(batch, params)
        params = params – learning_rate * grad
```

Kaynak: https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3

Yoğun bağlantılı katman (Dense Layer)







Yapar Sinir Ağı

- m. katmandaki, i. çıkış:

$$y_i^m = f^m(w_{i,1}^m y_1^{m-1} + w_{i,2}^m y_2^{m-1} + \ldots + w_{i,s^{m-1}}^m y_1^{m-1} + b_1^m) \quad i = 1,2,..,s^m$$

- Herhangi bir katmanın matrisel formda tüm çıkışları:

$$\begin{bmatrix} y_1^m \\ y_2^m \\ \vdots \\ y_{s^m}^m \end{bmatrix} = \mathbf{f}^m \left(\begin{bmatrix} w_{1,1}^m & w_{1,2}^m & \cdots & w_{1,s^{m-1}}^m \\ w_{2,1}^m & w_{2,1}^m & \cdots & w_{2,s^{m-1}}^m \\ \cdots & & & \cdots \\ w_{1,s^m}^m & w_{2,s^m}^m & \cdots & w_{s^m,s^{m-1}}^m \end{bmatrix} \begin{bmatrix} y_1^{m-1} \\ y_2^{m-1} \\ \vdots \\ y_{s^{m-1}}^{m-1} \end{bmatrix} + \begin{bmatrix} b_1^m \\ b_2^m \\ \vdots \\ b_{s^m}^m \end{bmatrix} \right)$$

Matrisel semboller ile

$$\mathbf{y}^m = \mathbf{f}^m(\mathbf{W}^m \mathbf{y}^{m-1} + \mathbf{b}^m)$$

Yapar Sinir Ağı

Toplam karesel hata (SSE)

$$E = \mathbf{e}^T\mathbf{e} = (\mathbf{t}-\mathbf{y})^T(\mathbf{t}-\mathbf{y}) = \sum_{i=1}^{s^M}(t_i - y_i)^2$$

$\mathbf{y} = \mathbf{f}(\mathbf{n}) = \mathbf{f}(\mathbf{W}\mathbf{x}+\mathbf{b})$ (hatanın karesi W ve b ile ilişkilidir)

Gradient Descent

$$w^m_{i,j}(k+1) = w^m_{i,j}(k) - \alpha\frac{\partial E}{\partial w^m_{i,j}}$$

$$b^m_i(k+1) = b^m_i(k) - \alpha\frac{\partial E}{\partial b^m_i}$$

Matrisel değişkenler cinsinden,

$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \alpha\frac{\partial E}{\partial \mathbf{W}^m}$$

$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \alpha\frac{\partial E}{\partial \mathbf{b}^m}$$

((k) mevcut iterasyondaki değeri, (k+1) sonraki iterasyondaki değeri)

Geri yayılım algoritması (Back propogation)

Giriş: $\mathbf{y}^0 = \mathbf{x}$, Çıkış: $\mathbf{y}^M = \mathbf{y}$

m. katmanın çıkış ifadesi:

$$\mathbf{y}^m = \mathbf{f}^m(\mathbf{W}^m\mathbf{y}^{m-1} + \mathbf{b}^m)$$

Geri yayılım:

En son katman için: $\mathbf{d}^M = -2\dot{\mathbf{F}}^M(\mathbf{n}^M)(\mathbf{t} - \mathbf{y})$ m=M

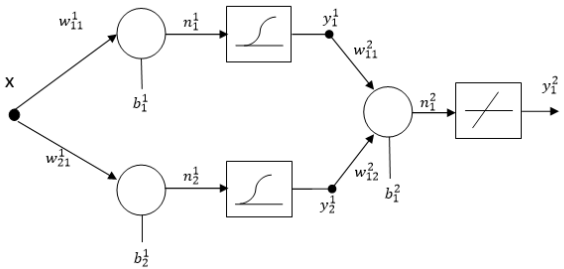
Diğerleri için: $\mathbf{d}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T\ \mathbf{d}^{m+1}$ m=M-1,...,2,1

Ağırlık ve sabitlerin yenilenmesi (Gradient Descent veya SGD, MB-GD,..)

$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \alpha\mathbf{d}^m(\mathbf{y}^{m-1})^T$$

$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \alpha\mathbf{d}^m$$

Örnek: Geri yayılım algoritması



Eğitim seti

P=[-2, -1.5, -1, -0.5, 0, 0.5, **1**, 1.5, 2]

T=[0, 0.075, 0.292, 0.617, 1.0, 1.382, **1.707**, 1.923, 2]

$$X = 1 \rightarrow y = y_1^2 = ?, \quad (T=1.707, \ \alpha = 0.1)$$

Başlangıç değerleri:

$$W^1 = \begin{bmatrix} w_{11}^1 \\ w_{21}^1 \end{bmatrix} = \begin{bmatrix} -0.3 \\ 0.5 \end{bmatrix} \qquad b^1 = \begin{bmatrix} b_1^1 \\ b_2^1 \end{bmatrix} = \begin{bmatrix} 0.2 \\ 0.1 \end{bmatrix}$$

$$W^2 = [w_{11}^2 \quad w_{12}^2] = [0.5 \quad 0.6] \qquad b^2 = [b_1^2] = [-0.1]$$

Başlangıç değerleri random atanır

Örnek: Geri yayılım algoritması

1. katman çıkışları

$$\begin{bmatrix} y_1^1 \\ y_2^1 \end{bmatrix} = \text{sigmoid}\left(\begin{bmatrix} n_1^1 \\ n_2^1 \end{bmatrix}\right) = \text{sigmoid}\left(\begin{bmatrix} -0.3 \\ 0.5 \end{bmatrix}[1] + \begin{bmatrix} 0.2 \\ 0.1 \end{bmatrix}\right) = \begin{bmatrix} 0.4750 \\ 0.6456 \end{bmatrix}$$

2. katman çıkışları

$$y_1^2 = \text{linear}(n_1^2) = \text{linear}\left([0.5 \quad 0.6]\begin{bmatrix} 0.4750 \\ 0.6456 \end{bmatrix} + [-0.1]\right) = 0.5249$$

Oluşan hata

İstenen sonuç: $[1] \rightarrow [1.707]$,

Hesaplanan sonuç: $[1] \rightarrow [0.5249]$,

Hata miktarı: **e= t-y**=1.1820

Örnek: Geri yayılım algoritması

Hatanın geri yayılımı: $d^2 \rightarrow d^1$

2. katman (çıkış katmanı) için geri yayılım

$$\mathbf{d}^2 = -2(\mathbf{t}-\mathbf{y})\dot{\mathbf{F}}^2(\mathbf{n}^2)$$

$$\dot{\mathbf{F}}^2(\mathbf{n}^2) = \left[\dot{f}^2\right] = 1$$

$$d^2 = -2\ (1.1820)1 = -2.3641$$

1. katman (gizli katman) için geri yayılım

$$\mathbf{d^1} = \dot{F}^1(n^1)(W^2)^T\mathbf{d^2}$$

$$\dot{F}^1(n^1) = \begin{bmatrix} \dot{f}^1 & 0 \\ 0 & \dot{f}^1 \end{bmatrix} = \begin{bmatrix} (1-y_1^1)y_1^1 & 0 \\ 0 & (1-y_2^1)y_2^1 \end{bmatrix} = \begin{bmatrix} 0.2493 & 0 \\ 0 & 0.2287 \end{bmatrix}$$

$$\mathbf{d^1} = \begin{bmatrix} 0.2493 & 0 \\ 0 & 0.2287 \end{bmatrix}\begin{bmatrix} 0.5 \\ 0.6 \end{bmatrix}(-2.3641) = \begin{bmatrix} -0.2947 \\ -0.3245 \end{bmatrix}$$

Örnek: Geri yayılım algoritması

Hata düzeltme

1. Katmanın yeni ağırlıkları

$$W^1 = W^1 - \alpha d^1 (y^0_{\square})^T$$

$$W^1 = \begin{bmatrix} -0.3 \\ 0.5 \end{bmatrix} - 0.1 \times \begin{bmatrix} -0.2947 \\ -0.3245 \end{bmatrix} (1)^T = \begin{bmatrix} -0.2705 \\ 0.5324 \end{bmatrix}$$

$$b^1 = b^1 - \alpha d^1 = \begin{bmatrix} 0.2 \\ 0.1 \end{bmatrix} - 0.1 \begin{bmatrix} -0.2947 \\ -0.3245 \end{bmatrix} = \begin{bmatrix} 0.2294 \\ 0.1324 \end{bmatrix}$$

2. Katmanın yeni ağırlıkları

$$W^2 = W^2 - \alpha d^2 (y^1_{\square})^T$$

$$W^2 = [0.5 \quad 0.6] - 0.1 \times 0.2957 \times \begin{bmatrix} 0.4750 \\ 0.6456 \end{bmatrix}^T = [0.6123 \quad 0.7526]$$

$$b^2 = b^2 - \alpha d^2 = [-0.1] - 0.1(-2.3641) = 0.1364$$

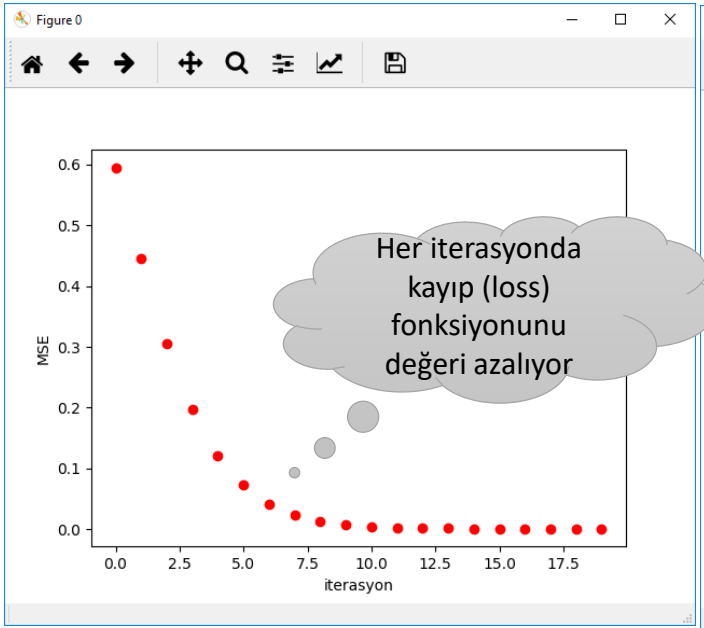
Örnek: Geri yayılım algoritması

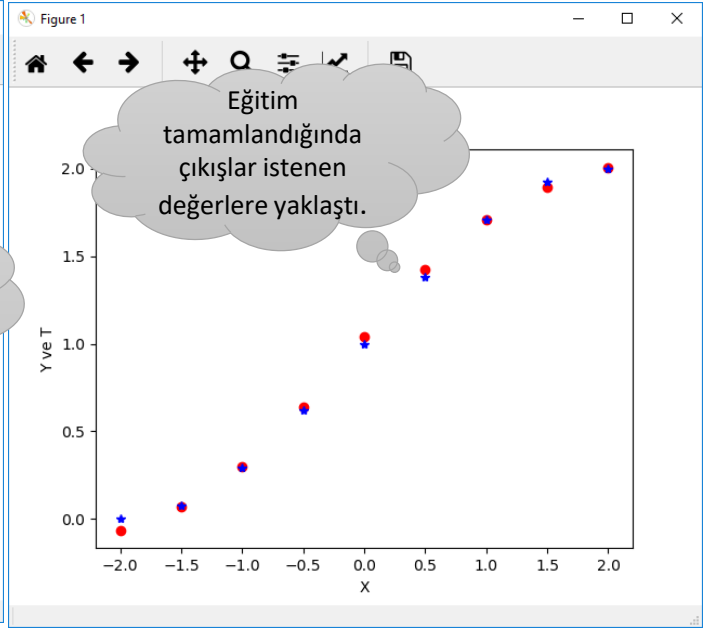
```
import numpy as np
import matplotlib.pyplot as grafik
from matplotlib.pyplot import plot
import nnw

# SGD (online training)
#VERI SETI ---------------------------------------------
#X=np.array([-2,-1.5,-1,-0.5, 0,0.5,1,1.5,2],dtype='f')
X=np.linspace(-2,2,9)
#T=np.array([0, 0.075,0.292,0.617,1.0, 1.382,1.707,1.923,2])
T=1+np.sin(X*np.pi/4)
W1=np.random.rand(2,1)
b1=np.random.rand(2,1)
W2=np.random.rand(1,2)
b2=np.random.rand(1)

alfa=0.1#öğrenme oranı(learning rate)
epoch=20

hataMSE=np.empty(epoch)
```

Örnek: Geri yayılım algoritması

```
for k in range(epoch):#Eğitim setinin kaç tur dolaşılacağını belirler

    for i in range(X.size):
        #print(i)
        #1. katman
        y1=nnw.sigmoid( W1*X[i]+b1)
        #2. katman
        y2=np.matmul(W2,y1)+b2 #W2*y1, linear: f(n)=n
        #hata
        e=T[i]-y2

        #GERI YAYILIMF2=[1];
        F2=1
        d2=-2*F2*e

        F1=np.array([[ (1-y1[0])*y1[0] , 0],
                     [0 , (1-y1[1])*y1[1]] ])

        d1= np.matmul(F1.astype(float), W2.reshape(2,1))*d2

        # 2. Katmandaki parametreler
        W2=W2-alfa*d2*y1.reshape(1,2) #y1'
        b2=b2-alfa*d2
        #1. Katmandaki parametreler
        W1=W1-alfa*d1*X[i] #X(i)'
        b1=b1-alfa*d1
```

Örnek: Geri yayılım algoritması






```
#Doğruluk testi
hata=0
for i in range(len(X)):
    #1. katman
    Y1=nnw.sigmoid( W1*X[i]+b1)
    #2. katman
    Y2=np.matmul(W2,Y1)+b2 #linear
    hata= hata+(T[i]-Y2)**2

MSE=hata/len(X)
print("MSE=",MSE)
hataMSE[k]=MSE
```

Kayıp (Loss) fonksiyonları

- Kayıp fonksiyonları geri yayılım algoritmasındaki hata miktarını hesaplamak için kullanılır. Dolayısıyla ağın eğitim sonucunu etkiler.
- Probleme uygun seçilmesi gerekir.
- Sık kullanılan bazı loss foksiyonları:
  - MSE
  - MAE
  - binary cross entropy
  - categorical cross entropy

MSE loss ve MAE loss

- MSE (Mean Square Error):Hesaplanan çıkışlarla beklenen çıkışların farklarının karelerinin toplamının eleman sayısına bölümü ile hesaplanır.

$$MSE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}$$

- MAE (Mean Absolute Error):Hesaplanan çıkışlarla beklenen çıkışların farklarının mutlak değerlerinin toplamının eleman sayısına bölümü ile hesaplanır.

$$MAE = \frac{\sum_{i=1}^{n} \mid y_i - \hat{y}_i \mid}{n}$$

Kayıp (Loss) Fonksiyonları

➢Mean Absolute Percentage Error (MAPE)

$$\mathrm{M} = \frac{100\%}{n} \sum_{t=1}^{n} \left| \frac{A_t - F_t}{A_t} \right|,$$

Gerçek hedef 10 iken tahminimin 12 olduğunu varsayalım, bu tahmin için MAPE

$$|(10-12)/10| = 0.2.$$

Kaynak: https://www.machinecurve.com/index.php/2019/10/04/about-loss-and-loss-functions/

Kayıp (Loss) Fonksiyonları

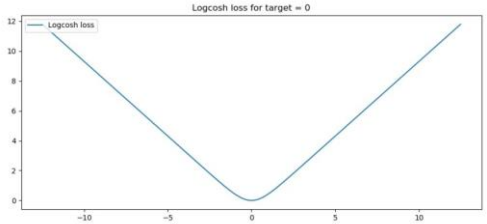
Root Mean Squared Error (L2 Loss)

$$\mathrm{RMSD}(\hat{\theta}) = \sqrt{\mathrm{MSE}(\hat{\theta})}$$

Logcosh

Log-cosh is the logarithm of the hyperbolic cosine of the prediction error.” (Grover, 2019).

$$Logcosh(t) = \sum_{p \in P} \log\left(\cosh(p - t)\right)$$

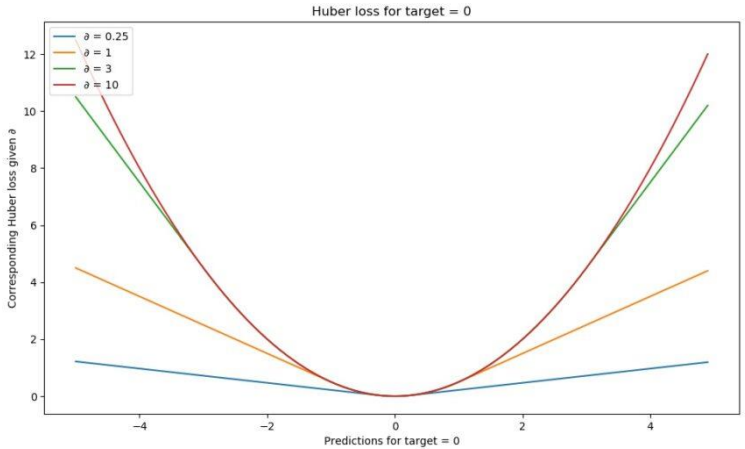


Kaynak: https://www.machinecurve.com/index.php/2019/10/04/about-loss-and-loss-functions/

Kayıp (Loss) Fonksiyonları

- **Huber loss**

$$Huber\ loss(t,p) = \begin{cases} \frac{1}{2}(t-p)^2, & when\ |t-p| \leq\ \partial \\ \partial|t-p| - \frac{\partial^2}{2}, & otherwise \end{cases}$$



Kaynak: https://www.machinecurve.com/index.php/2019/10/04/about-loss-and-loss-functions/

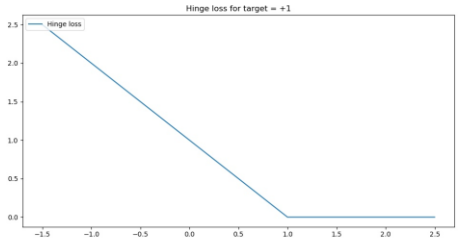
Kayıp (Loss) Fonksiyonları

- Sınıflandırma için kullanılır
- The **hinge loss** is defined as follows (Wikipedia, 2011):

$$\ell(y) = \max(0, 1 - t \cdot y)$$

It simply takes the maximum of either 0 or the computation 1–t×y, where t is the machine learning output value (being between -1 and +1) and y is the true target (-1 or +1).

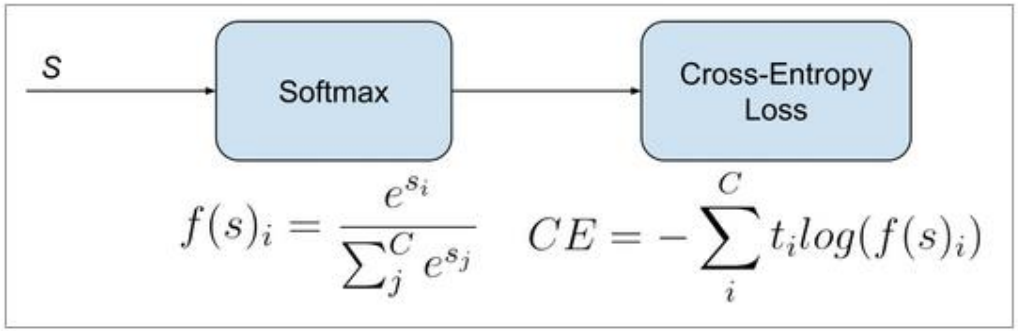
When the target equals the prediction, the computation t×y is always one: 1×1=−1×−1=1). Essentially, because then 1–t×y=1–1=0, the max function takes the maximum max(0,0), which of course is 0.



Kaynak: https://www.machinecurve.com/index.php/2019/10/04/about-loss-and-loss-functions/

Cross Entropy Loss

- **Categorical cross entropy**
- Çok sınıflı sınıflandırma (multiclass classification) ile kullanılır.



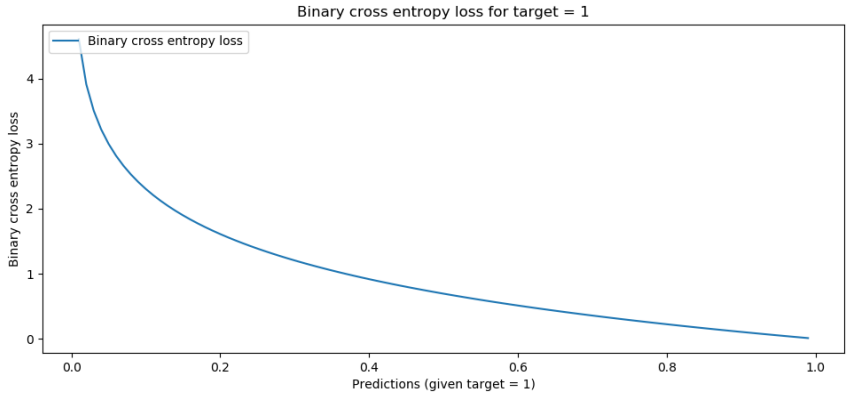
- 0: $[1, 0, 0]$
- 1: $[0, 1, 0]$
- 2: $[0, 0, 1]$

Kaynak: https://gombru.github.io/2018/05/23/cross_entropy_loss/

Kayıp (Loss) Fonksiyonları

- **Binary crossentropy**
- İkili sınıflandırma için kullanılır

$$BCE\ (t,p) = -(t * \log(p) + (1-t) * \log\ (1-p))$$



Keras loss functions

- **keras.losses**
- mean_squared_error
- mean_absolute_error
- mean_absolute_percentage_error
- mean_squared_logarithmic_error
- squared_hinge
- hinge
- categorical_hinge
- logcosh
- huber_loss
- categorical_crossentropy
- sparse_categorical_crossentropy
- binary_crossentropy
- kullback_leibler_divergence
- poisson
- cosine_proximity

Kaynak: https://keras.io/losses/

Kaynaklar

- Prof. Dr. Devrim Akgün, Deep Learning ders notları
- https://miuul.com/blog/yapay-sinir-aglarinda-ogrenme-sureci