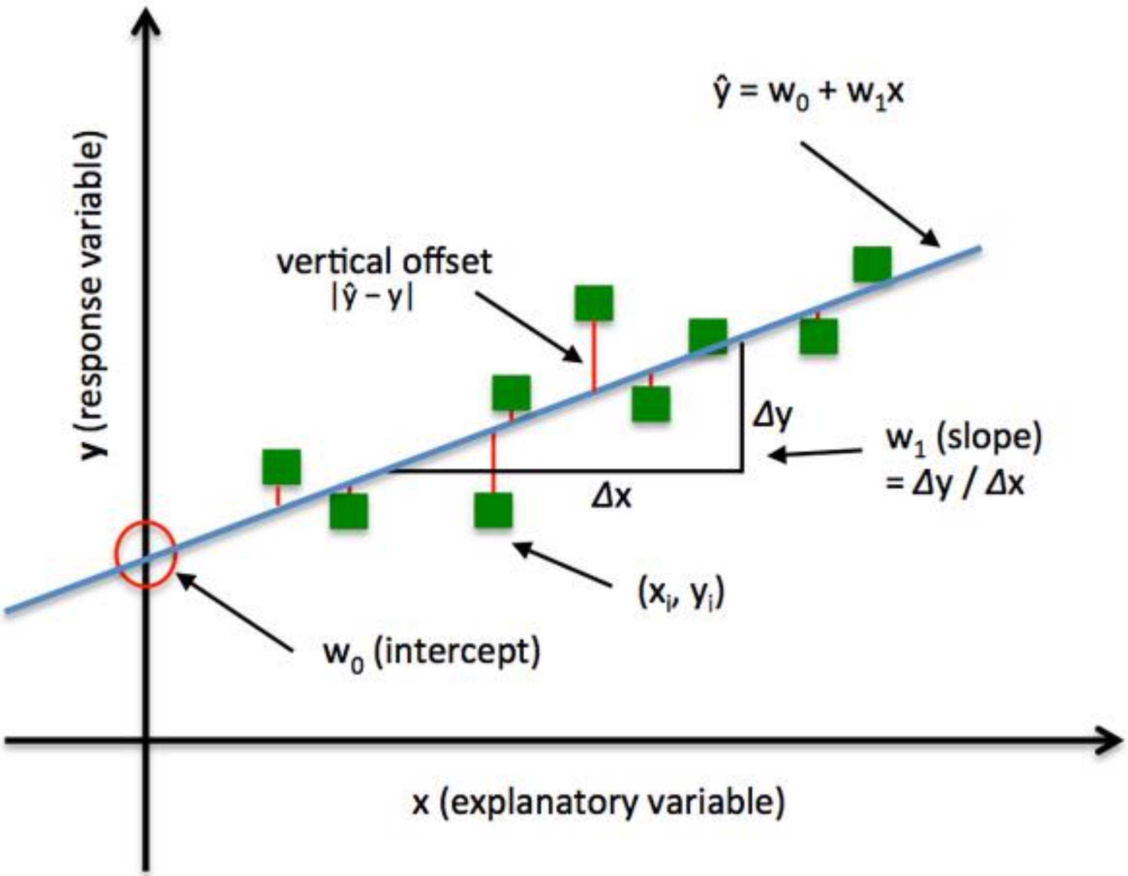
- Gradient Descent (GD)
- Geri yayılım (Back Propagation)
- Kayıp (loss) fonksiyonları

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. boyut veri kümesi

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

Doğrusal regresyon

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

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

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

- 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:

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

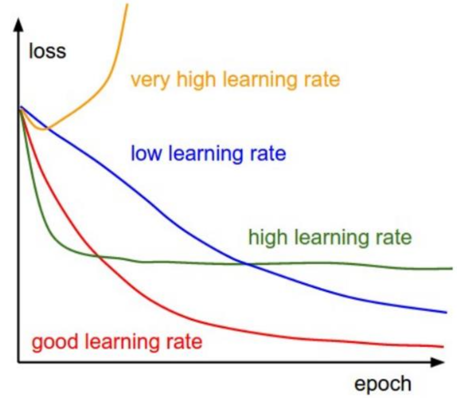
Gradient Descent (GD)

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

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

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

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

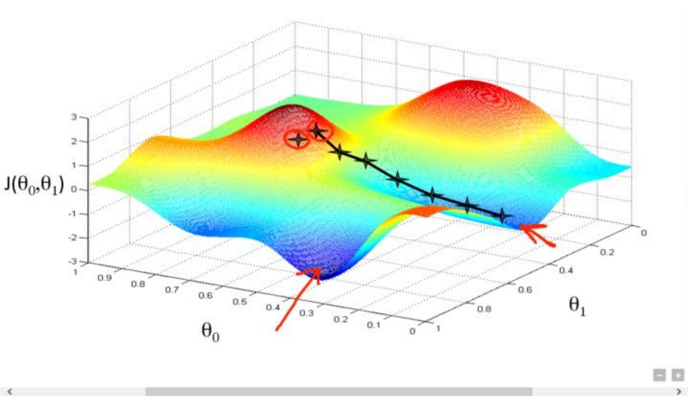


Kaynak: https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3

Gradient Descent (GD)

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

$$
\begin{aligned}
\Delta w_j &= -\eta \frac{\partial J}{\partial w_j} \\
&= -\eta \sum_i (\text{target}^{(i)} - \text{output}^{(i)})(-x_j^{(i)}) \\
&= \eta \sum_i (\text{target}^{(i)} - \text{output}^{(i)}) x_j^{(i)}.
\end{aligned}
$$

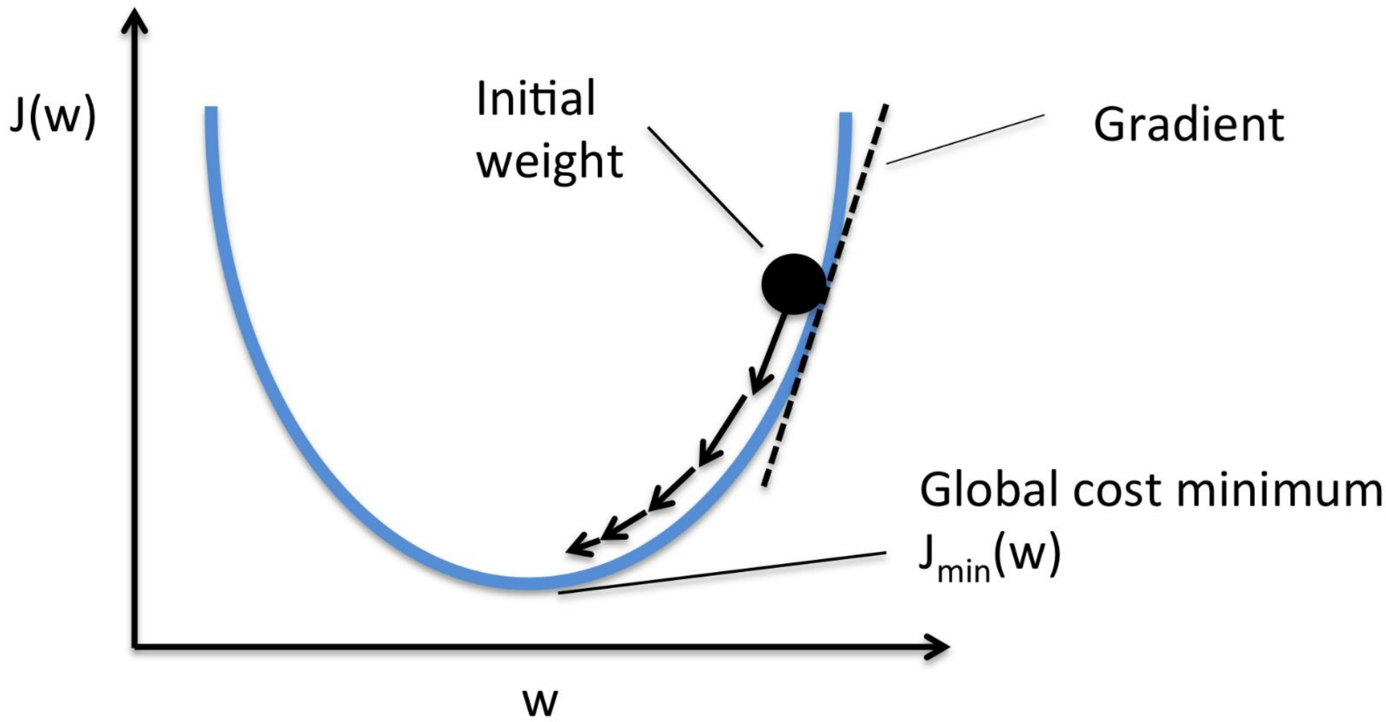


https://hackernoon.com/gradient-descent-aynk-7cbe95a778da

- Temel olarak, GD optimizasyonunu bir dağdan aşağıya (maliyet fonksiyonu) bir vadiye gitmek isteyen (minimum maliyet) bir yürüyüşçü (ağırlık katsayısı) olarak görebiliriz.
- Her adım eğimin (eğim) ve eğimin dikliği ve uzunluğu (öğrenme hızı) ile belirlenir.

Gradient Descent (GD)

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



Stachastic Gradient Descent (SGD)

- GD optimizasyonunda, eğitim setindeki tüm veile 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 dayanan degradenin "gerçek" maliyet gradyanının "stokastik bir yaklaşım" olması gerçeğinden gelir.
- Stokastik yapısı nedeniyle, küresel minimum maliyete giden yol BGD'de olduğu gibi "doğrudan" değildir, "zig-zag" olabilir.
- Adaptif bir öğrenme hızı *η* için, zaman içindeki öğrenme oranını azaltan bir azalma sabiti *d* seçilir, (t iterasyon numarası):

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

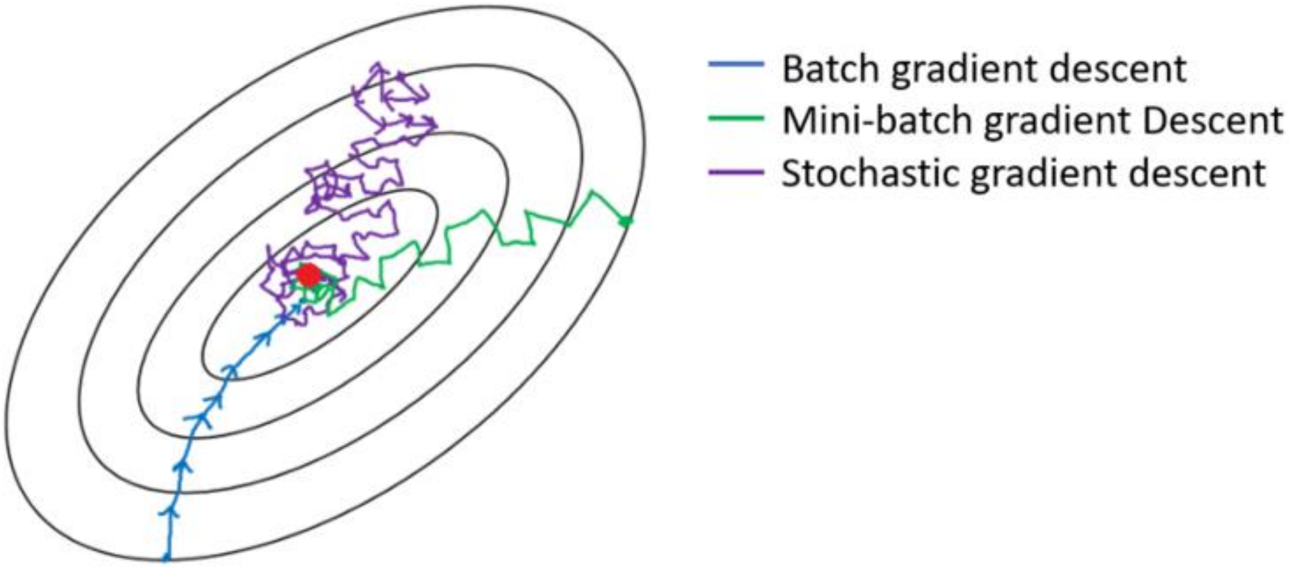
- Momentum öğrenme, daha hızlı güncellemeler için ağırlık güncellemesine önceki gradyanın bir çarpan (decay) ile ekleyerek sağlanır.

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

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

Mini Batch Gradient Descent (MB-GD)

- MB-GD yaklaşımı BGD ve SGD arasında bir yol izler.
- MB-GD'de model daha küçük eğitim örnekleri grubuna göre güncellenir.
- Gradyanı bir örnekten (SGD) veya tüm n eğitim setinden numunelerinden (BGD) hesaplamak yerine eğitim setini belli parçalara bölerek hesaplar.
- MB-GD, GD'den daha az yinelemeyle bir araya gelir; çünkü ağırlıkları daha sık güncelleriz; bununla birlikte MB-GD, genellikle SGD'den hesaplanan performans artışıyla sonuçlanan vektörelleştirilmiş işlemi kullanalım.



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

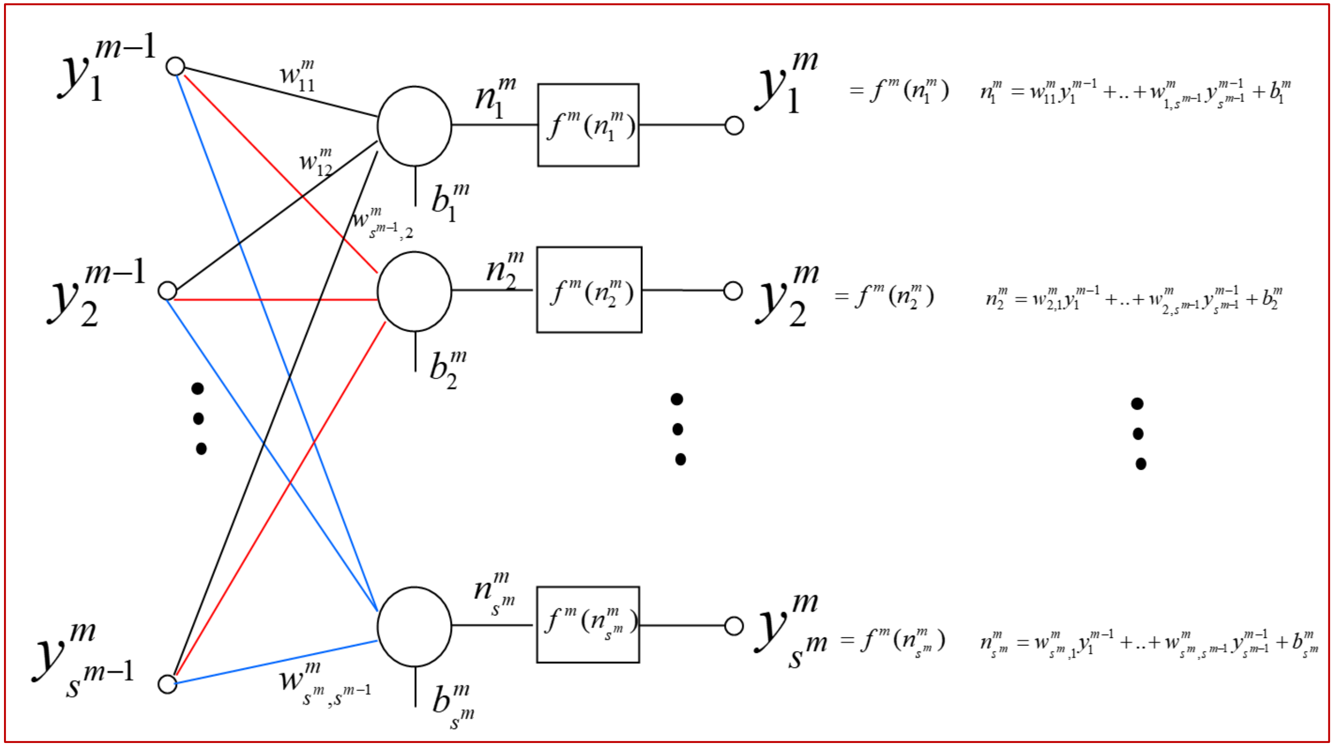
```
for i in range(num_epochs):
    np.random.shuffle(dataset)
    for example in dataset:
        grad = compute_gradient(example, params)
        params = params - learning_rate * grad
```

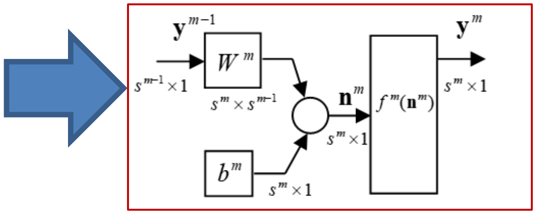
Mini-batch Gradient Descent

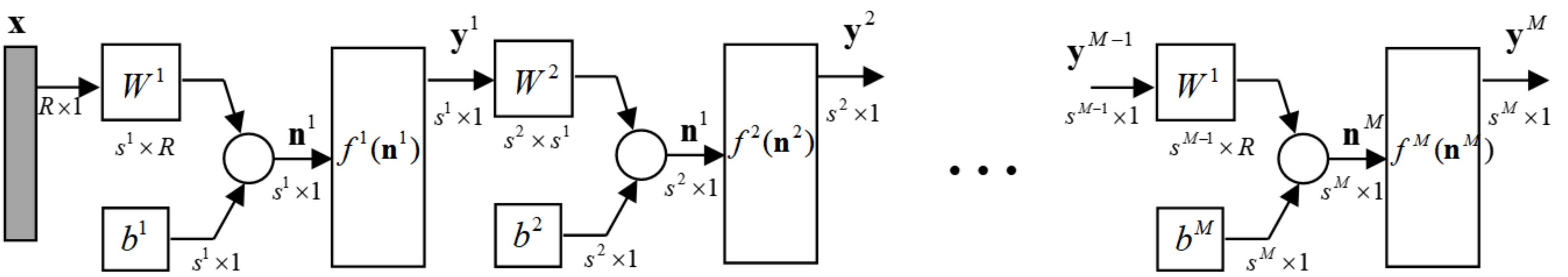
```
for i in range(num_epochs):
    np.random.shuffle(dataset)
    for batch in radom_minibatches(data, batch_size=32):
        grad = compute_gradient(batch, params)
        params = params - learning_rate * grad
```

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

Yoığun bağlantılı katman (Dense Layer)







Yapar Sinir Ağı

- m. katmandaki, i. çıkış:

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

- 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)

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

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

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

Gradient Descent

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

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)

kısmi türevler hesaplanırken zincir kuralına göre iki parça halinde yazılır:

$$\frac{\partial E}{\partial w_{i,j}^{m}} = \boxed{\frac{\partial E}{\partial n_{i}^{m}}} \frac{\partial n_{i}^{m}}{\partial w_{i,j}^{m}}$$

$$\frac{\partial n_{i}^{m}}{\partial w_{i,j}^{m}} = \frac{\partial}{\partial w_{i,j}^{m}}\left(w_{i,1}^{m} y_{1}^{m-1} + .. + w_{i,j}^{m} y_{j}^{m-1} + \ldots + w_{i,s^{m-1}}^{m} y_{s^{m-1}}^{m-1} + b_{i}^{m}\right)$$

$$= \frac{\partial}{\partial w_{i,j}^{m}}\left(w_{i,j}^{m} y_{j}^{m-1}\right) = y_{j}^{m-1}$$

$$\frac{\partial E}{\partial w_{i,j}^{m}} = \frac{\partial E}{\partial n_{i}^{m}} \frac{\partial n_{i}^{m}}{\partial w_{i,j}^{m}} \rightarrow \frac{\partial E}{\partial w_{i,j}^{m}} = d_{i}^{m} y_{j}^{m-1}$$ ve

$$\frac{\partial E}{\partial b_{i}^{m}} = \frac{\partial E}{\partial n_{i}^{m}} \frac{\partial n_{i}^{m}}{\partial b_{i}^{m}} \rightarrow \frac{\partial E}{\partial b_{i}^{m}} = d_{i}^{m}$$ şeklinde sadeleşir.

$$w_{i,j}^{m}(k+1) = w_{i,j}^{m}(k) - \alpha d_{i}^{m} y_{j}^{m-1}$$

$$b_{i}^{m}(k+1) = b_{i}^{m}(k) - \alpha d_{i}^{m}$$

Geri yayılım algoritması (Back propogation)

$$d_i^m = \frac{\partial E}{\partial n_i^m}$$, kısmi türevinin belirlenmesi

$$d_i^m = \frac{\partial E}{\partial n_i^m} = \frac{\partial n_i^{m+1}}{\partial n_j^m} \frac{\partial E}{\partial n_i^{m+1}}$$

$$\mathbf{d}^m = \frac{\partial E}{\partial \mathbf{n}^m} = \left(\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m} \right)^T \frac{\partial E}{\partial \mathbf{n}^{m+1}}$$

$$\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m} = \begin{bmatrix} \frac{\partial n_1^{m+1}}{\partial n_1^m} & \frac{\partial n_1^{m+1}}{\partial n_2^m} & \cdots & \frac{\partial n_1^{m+1}}{\partial n_{s^m}^m} \\ \frac{\partial n_2^{m+1}}{\partial n_1^m} & \frac{\partial n_2^{m+1}}{\partial n_2^m} & \cdots & \frac{\partial n_2^{m+1}}{\partial n_{s^m}^m} \\ \cdots & & & \cdots \\ \frac{\partial n_{s^{m+1}}^{m+1}}{\partial n_1^m} & \frac{\partial n_{s^{m+1}}^{m+1}}{\partial n_2^m} & \cdots & \frac{\partial n_{s^{m+1}}^{m+1}}{\partial n_{s^m}^m} \end{bmatrix}_{s^{m+1} x s^m}$$

$$\mathbf{d}^m = \left(\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m} \right)^T \mathbf{d}^{m+1}$$

$$\mathbf{d}^{m+1} = \frac{\partial E}{\partial \mathbf{n}^{m+1}}$$

Geri yayılım algoritması (Back propogation)

$$\frac{\partial n_i^{m+1}}{\partial n_j^m} = \frac{\partial}{\partial n_j^m}\left(w_{i,1}^{m+1} y_1^m(n_1^m) + .. + w_{i,s^m}^{m+1} y_{s^m}^m(n_{s^m}^m) + b_i^{m+1}\right)$$

$$= w_{i,j}^{m+1} \frac{\partial y_j^m}{\partial n_j^m} = w_{i,j}^{m+1} \dot{f}(n_j^m) \qquad (y_j^m = f(n_j^m))$$

Matrisel formda:

$$\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m} = \begin{bmatrix} w_{1,1}^{m+1}\dot{f}(n_1^m) & w_{1,2}^{m+1}\dot{f}(n_2^m) & \cdots & w_{1,s^m}^{m+1}\dot{f}(n_{s^m}^m) \\ w_{2,1}^{m+1}\dot{f}(n_1^m) & w_{2,2}^{m+1}\dot{f}(n_2^m) & \cdots & w_{2,s^m}^{m+1}\dot{f}(n_{s^m}^m) \\ \cdots & & & \cdots \\ w_{s^{m+1},1}^{m+1}\dot{f}(n_1^m) & w_{s^{m+1},2}^{m+1}\dot{f}(n_2^m) & \cdots & w_{s^{m+1},s^m}^{m+1}\dot{f}(n_{s^m}^m) \end{bmatrix}$$

Geri yayılım algoritması (Back propogation)

Matrisel değişkenler cinsinden:

$$\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^{m}} = \dot{\mathbf{F}}^{m}(\mathbf{n}^{m})(\mathbf{W}^{m+1})^{T} \qquad \dot{\mathbf{F}}^{m}(\mathbf{n}^{m}) = \begin{bmatrix} \dot{f}(n_1^m) & 0 & \cdots & 0 \\ 0 & \dot{f}(n_2^m) & \cdots & 0 \\ \cdots & & & \cdots \\ 0 & 0 & \cdots & \dot{f}(n_{s^m}^m) \end{bmatrix}$$

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

m=1,2,...,M (M katmanlı ağ)

$$\mathbf{d}^{M} \rightarrow \mathbf{d}^{M-1} \rightarrow ,\ldots, \rightarrow \mathbf{d}^{2} \rightarrow \mathbf{d}^{1}$$

Çıkış katmanı için d, hatanın karesinin türevinden belirlenir

Geri yayılım algoritması (Back propogation)

Hesaplamaya son katmandan başlayıp geriye doğru gidilir (backpropogation)

Bundan dolayı öncelikle başlangıç noktasının ($\mathbf{d}^M$) belirlenmesi gerekir.

$$d_i^M = \frac{\partial E}{\partial n_i^M} = \frac{\partial}{\partial n_i^M}(t_i - y_i)^2 = -2(t_i - y_i)\frac{\partial y_i}{\partial n_i^M}$$

$$d_i^M = -2(t_i - y_i)\dot{f}^M(n_i^M), \quad \text{i=1, ..., } s^m$$

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

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

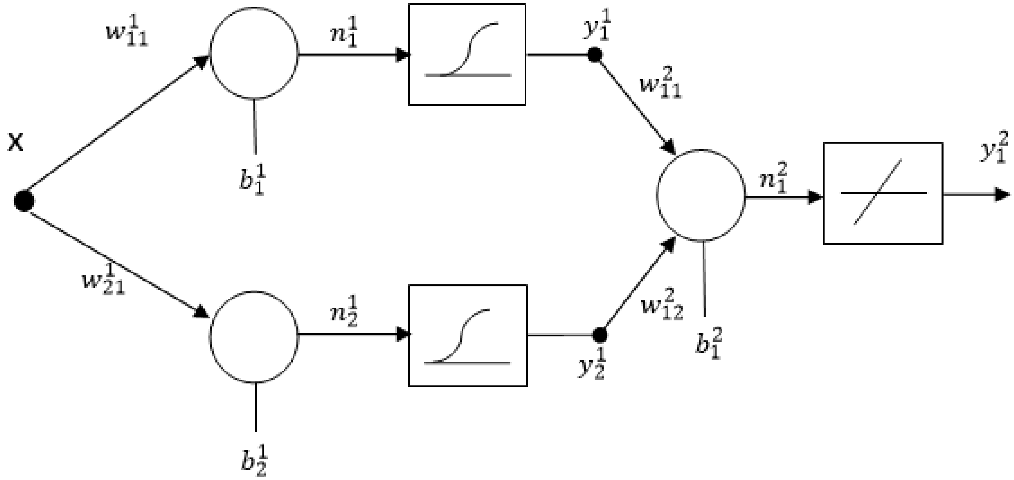
Diğerleri için: $\mathbf{d}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T \mathbf{d}^{m+1}$ m=M-1,...,2,1

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]

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

Başlangıç değerleri:

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

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

Bağlangıç 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

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

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

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

1. katman (gizli katman) için geri yayılım

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

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

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

Örnek: Geri yayılım algoritması

Hata düzeltme

1. Katmanın yeni ağırlıkları

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

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

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

2. Katmanın yeni ağırlıkları

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

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

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

Örnek: Geri yayılım algoritması

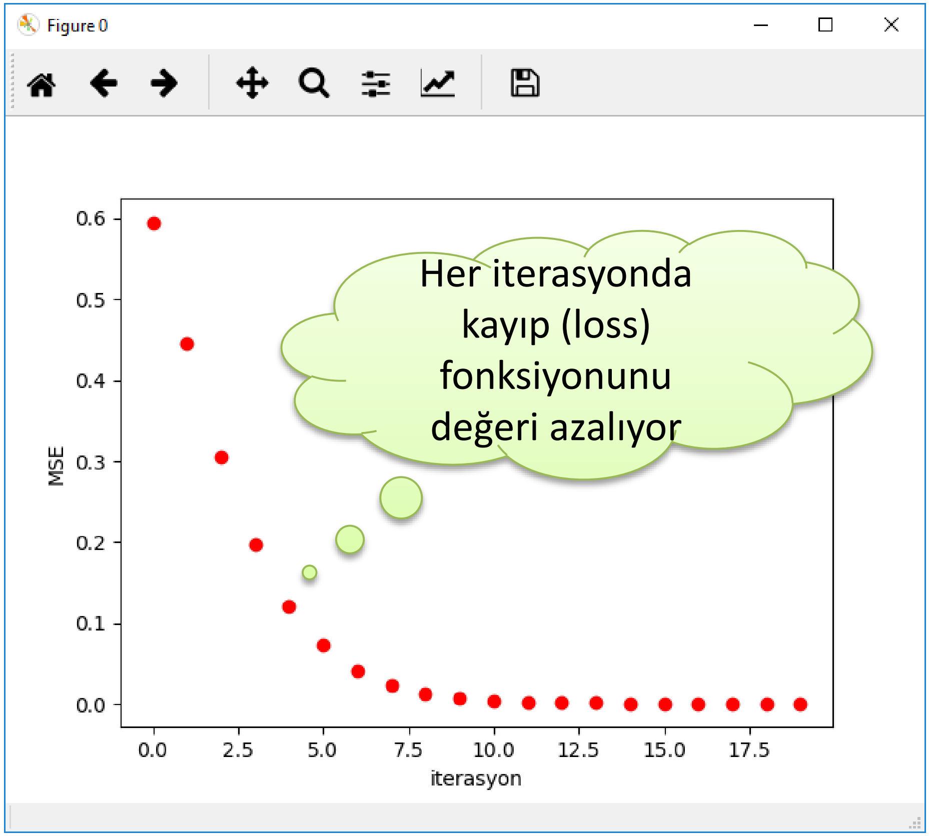
```
import numpy as np
import matplotlib.pyplot as grafik
from matplotlib.pyplot import plot
import nnw

# 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)
```

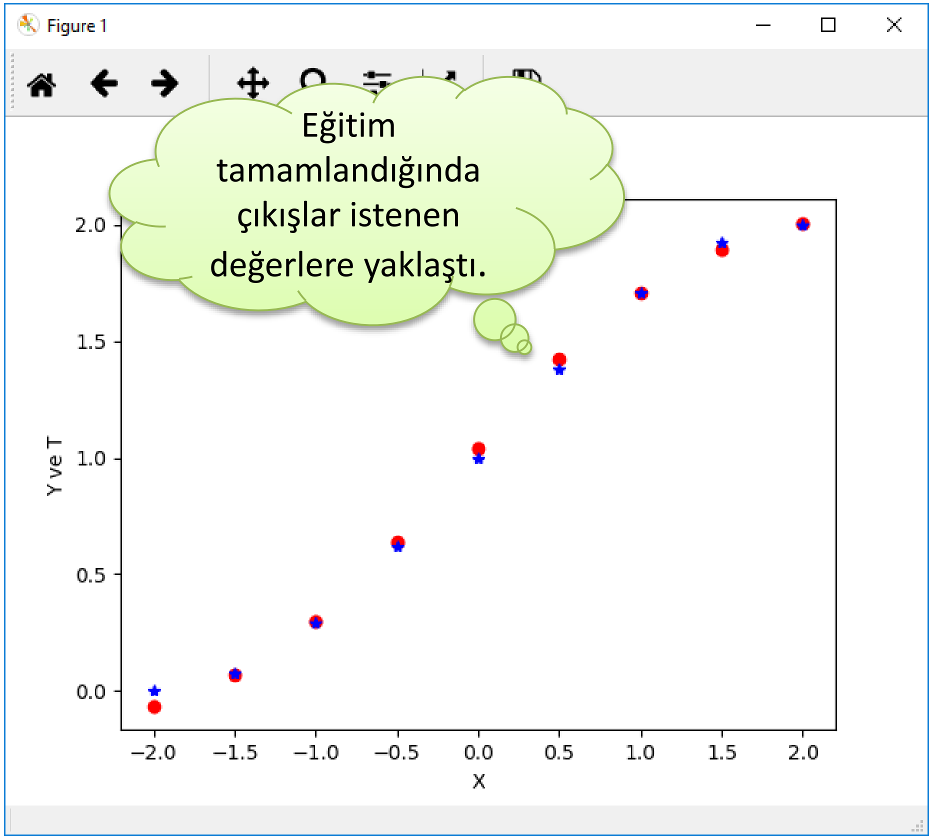
Örnek: Geri yayılım algoritması

```python
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

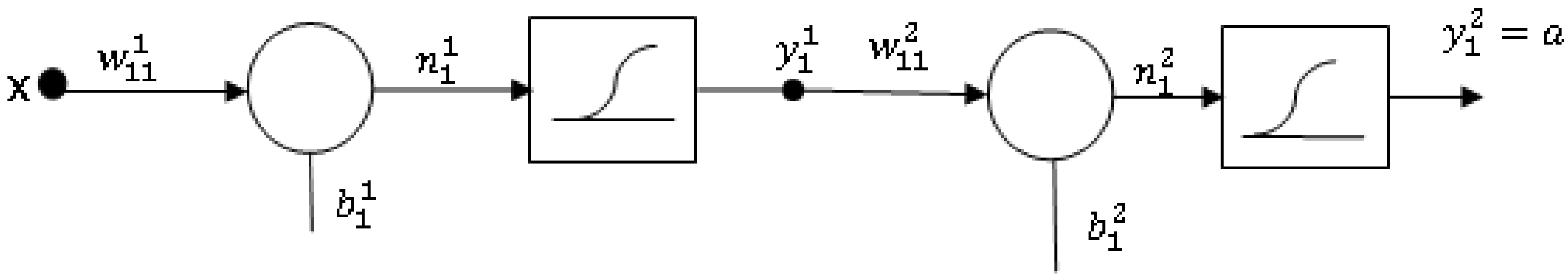
```
#Doğruluk testi
hata=0
for i in range(len(X)):
    #1. katman
    Y1=nnw.sigmoid( W1*X[i]+b1)
    #2. katman
    Y2=np.matmul(W2,Y1)+b2 #Linear
    hata= hata+(T[i]-Y2)**2

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





Örnek: Geri yayılım algoritması

- Aşağıda verilen tek-girişli tek çıkışlı iki katmanlı ağda x=1 için a=2 olması isteniyor. Geri yayılım algoritması 1 iterasyon çalıştırıldığında yeni ağ ağırlıklarını hesaplayınız. Tüm ağırlıkların (sabitler dahil) başlangıç değeri 0.5 ve öğrenme oranı 0.1 alınacaktır.



Kayıp (Loss) fonksiyonları

- Kayıp fonksiyonları geri yayılım algoritmasındaki hata miktarını dolayısıyla ağın eğitimi etkilediği için probleme uygun seçilmesi gerekir.
- Sık kullanılan bazı loss foksiyonları:
  - MSE
  - binary cross entropy
  - categorical cross entropy ve

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.

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

Cross Entropy Loss

- Cross entropy kayıp fonksiyonu sınıflandırma problemlerinde tercih edilir.

$$\text{CE} = -\sum_{i}^{C} t_i log(s_i)$$

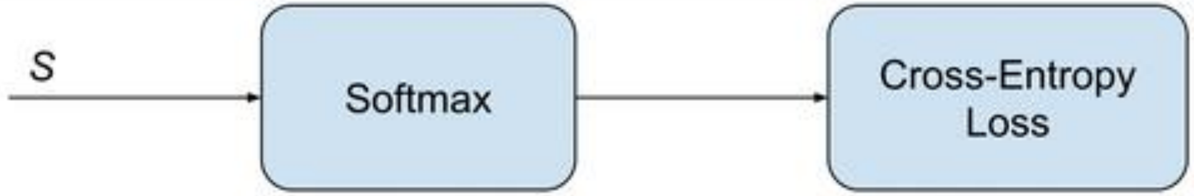
- Binary cross entropy: İki adet sınıf

$$\text{CE} = -\sum_{i=1}^{C=2} t_i \log(s_i)$$

=-t1log(s1)-(1-t1) log(1-s1)

Cross Entropy Loss

- Categorical cross entropy: çok sınıflı sınıflandırma (multiclass classification) ile kullanılır.



$$f(s)_i = \frac{e^{s_i}}{\sum_j^C e^{s_j}} \qquad CE = -\sum_i^C t_i log(f(s)_i)$$

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

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/