Jupyter ile Yapay Sinir Ağları

Bölüm 1 - Veri ve Mimari

Koddaki Sembol	Matematiksel Sembol	Tanım	Boyutlar
X	X	Giriş Verisi	(ornekSayisi, girisKatman)
y	y	Hedef Veri	(ornekSayisi, cikisKatman)

Tablo 1: Değişkenler

Herhangi bir x değişkenine veya değişkenlerine bağlı bir skaler y değişkenimizin olduğu bir tablomuz olsun. Diyelim ki sınav öncesi son gece uyuduğumuz saat ve çalıştığımız saat bizim x1 ve x2 değişkenlerimiz olsun. Sınav sonucumuz da buna bağlı bir skaler y olmak üzere;

Uyku	Çalışma	Sonuç
3	5	75
5	1	82
10	2	93

Tablo 2: Giriş, Çıkış Boyut ve Değerleri

Değerlerimizi numpy dizilerine aktaralım. Amacımız, bir sonraki uyku ve çalışma saatlerimizden hareketle, sınav sonucunu tahmin etmek için bir model geliştirmek. Amacımız bir sonraki uyku ve çalışma saatlerimizden hareketle, sınav sonucunu tahmin etmek için bir model geliştirmek. Elimizde bir Gözetimli Regresyon modeli bulunmakta.

Gözetimli dememizin sebebi giriş ve çıkış değerlerini etiketlediğimiz için bunlardan hareketle yeni sonucu üretebiliyor olmamızdan. Regresyon problemi olmasının sebebi ise giriş değerlerini tek bir skaler değişkende özetleyerek sürekli bir eğride ifade edebiliyor oluşumuzdan kaynaklanıyor.

```
%pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
# X = (uyku, calisma), y = sonuc
X = np.array(([3,5], [5,1], [10,2]), dtype=float)
y = np.array(([75], [82], [93]), dtype=float)
```

```
X
```

```
array([[  3.,   5.],
       [  5.,   1.],
       [ 10.,   2.]])
```

```
y
```

```
array([[ 75.],
       [ 82.],
       [ 93.]])
```

Eğer sınav sonucunu 0-100 arasında skaler bir değer olarak almak yerine harf notu olarak bulmaya çalışsaydık, problemimiz bir öbekleme problemi haline gelecekti. Ağımızı eğitmeye başlamadan önce elimizdeki verileri birimleştirmeliyiz ki bir standardımız olsun. Elimizdeki değerlerin hepsi pozitif, ve sınav sonucunun en fazla 100 olabileceğini kabul ederek normalize edelim.

```
X = X/np.amax(X, axis=0)
y = y/100 #Sonuç En fazla 100 olabilir
```

```
X
```

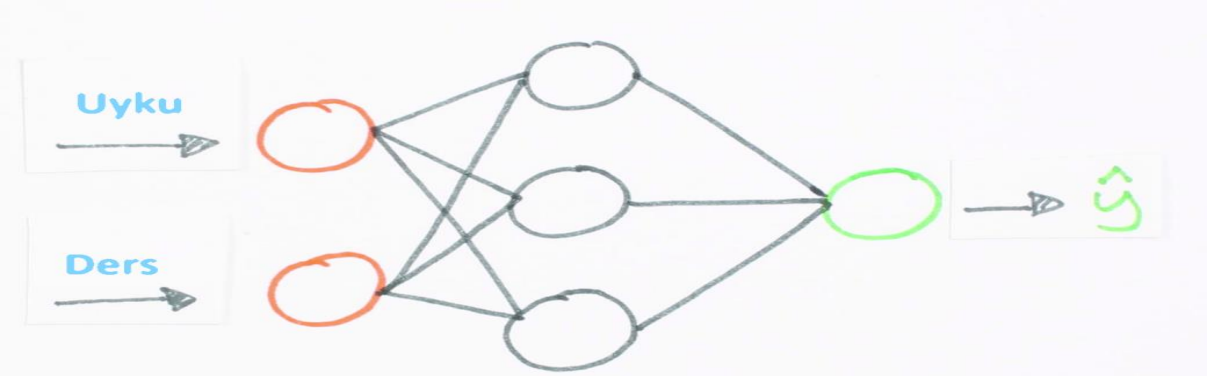
```
array([[ 0.3,  1. ],
       [ 0.5,  0.2],
       [ 1. ,  0.4]])
```

```
y
```

```
array([[ 0.75],
       [ 0.82],
       [ 0.93]])
```

Giriş Değerlerimiz iki boyutlu, sonucumuz y ise bir skaler, y den farklı olmak üzere y^ yhat terimini kullanmamızın sebebi ise yhat'ın bizim olası gördüğümüz, yani tahmin ettiğimiz sonuç olmasından kaynaklanıyor.

Giriş ve sonuç arasındaki katmanlara saklı katmanlar diyoruz, çok sayıda ara katmanın karmaşık problemleri oluşturduğu ağlar için derin inanç ağları diyoruz.



Tablo 3: Giriş, Çıkış ve Ara Katman Sinir Ağları

Bizim problemimizde sadece 3 boyutlu bir ara katman bulunmakta. Yuvarlaklarımız nöronlarımız. Çizgiler de sinapslar. Sinapsların görevi aldıkları giriş değerini spesifik bir ağırlık değeriyle çarparak bir sonraki nörona iletmek. Nöronlar ise bağlı olan tüm sinapslardan gelen girişleri toplamak ve aktivasyon fonksiyonuna sokmaktır.

Bölüm 2: İleri Yayılım

Koddaki Sembol	**Matematiksel Sembol**	**Tanım**	**Boyutlar**
X	X	Giriş Verisi, her satır bir örnek	(numExamples, inputLayerSize)
y	y	Hedef Veri	(numExamples, outputLayerSize)
W1	W(1)	Katman 1 ağırlıklar	(inputLayerSize, hiddenLayerSize)
W2	W(2)	Katman 2 ağırlıklar	(hiddenLayerSize, outputLayerSize)
z2	z(2)	Katman 2 aktivasyon	(numExamples, hiddenLayerSize)
a2	a(2)	Katman 2 aktivite	(numExamples, hiddenLayerSize)
z3	z(3)	Katman 3 aktivasyon	(numExamples, outputLayerSize)

Tablo 4: Değişkenler

Ağ yapımızı bir python sınıfı(class) olarak tanımlayacağız giriş metodumuz session için değişkenleri ve sabitleri ekleyecek. Değişken ve sabitlere tüm sınıfların dışarıdan ulaşılabilmesini istiyorsak, public nesneler halinde çağırmamız gerekli. Bunun için de C#'ta this metoduyla olduğu gibi Python'da da self metoduyla çağırıyoruz. self.deger veya değişken adı diyerek harici sınıflardan da bu değerleri çağırabilmekteyiz.

Ağımızda 2 giriş , 3 saklı katman ve 1 çıkış katmanı bulunuyor. bunlar bizim hiperparametrelerimiz. Hiperparametreler bir sinir ağının yapısını ve davranışını belirleyen sabitlerdir. Ağımızı eğitirken değiştirilemezler.

Öğrenme algoritmamız bir başka ara katmana ihtiyaç duyup duymadığını hesaba katamaz. Bu yüzden eğitimden önce hiperparametrelere bizim karar vermemiz gereklidir. Sinir ağları parametrelerden, sinapslardaki ağırlıklardan öğrenir.

Verileri ağımızda forward metodu ile tek seferde çoklu girdi değerleri matrisleri olarak taşırız. Bu bize Matlab ya da numpy gibi araçlar kullanırken performans optimizasyonu sağlar. X matrisindeki her giriş değeri, kendine karşılık gelen bir ağırlıkla çarpılıp, diğer nöronlardan gelen değerlerle toplanmalıdır.

Giriş verilerimizin oluşturduğu matrisin 3'e 2'lik bir matris olmasının sebebi 3 adet 2 boyutlu veriye sahip olmamızdan kaynaklanıyor. Buna bağlı olarak çıkış değeri olan y, ya da tahmin değerimiz olan yhat, 3 adet 2 boyutlu veriden tek boyutlu skeler bir sonuç ürettiği için 3'e 1'lik bir matristir.

$$z^{(2)} = XW^{(1)}$$

X matrisindeki her giriş değeri, kendine karşılık gelen bir W ağırlıkla çarpılıp, diğer nöronlardan gelen değerlerle toplanmalıdır. z(2) ikinci katmanın aktvitesidir. Her katmanda z değerleri, o katmana air saklı nöronların ağırlıklarla çarpılıp toplanmasıdır.

Bu yüzden z her örnek için bir satırı temsil eder. Her saklı değer için birer sütun ve her örnek için bir satır bize 3'e 3 lük bir matris verecektir.

Sinapsların görevi aldıkları giriş değerini spesifik bir ağırlık değeriyle çarparak bir sonraki nörona iletmek, Nöronlar ise bağlı olan tüm sinapslardan gelen girişleri toplamak ve aktivasyon fonksiyonuna sokmaktır demiştik.

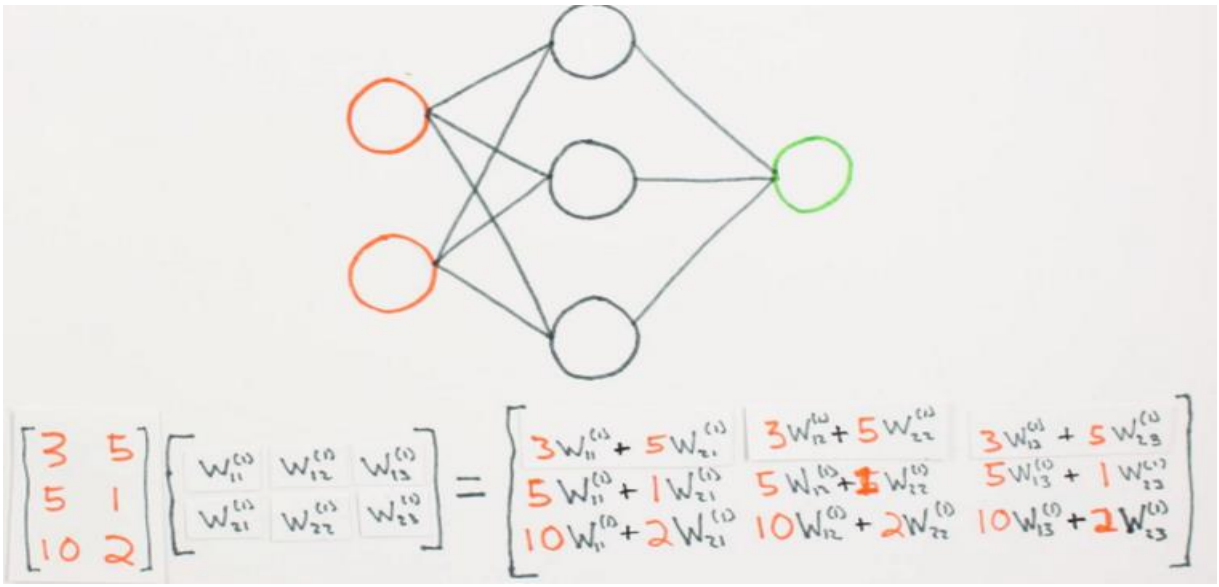
Aktivasyon fonksiyonları ile kompleks, nonlineer fonksiyonlar oluşturulabilir. Biz modelimizde sigmoid aktivasyon fonksiyonu kullanacağız.

Sigmoid aktivasyonu sürekli ve türevlenebilen bir fonksiyondur. Doğrusal olmadığından ysa'da oldukça sık kullanılır. Girdi değerlerinin her biri için 0 ile 1 arasında bir değer üretir.

İkinci katman z değeri elimizde olduğuna göre sigmoid aktivasyon fonksiyonumuzu uygulayabiliriz.

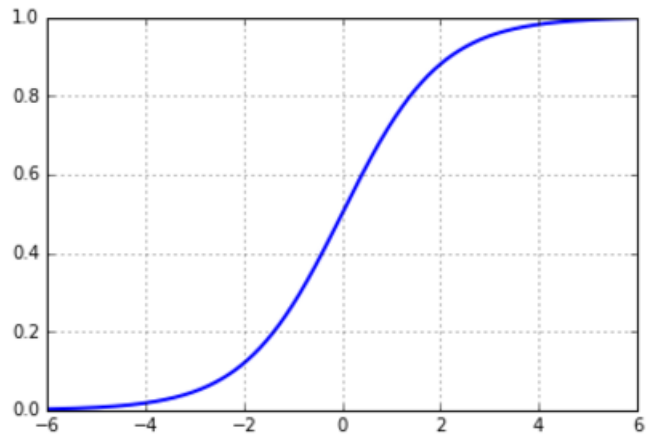
```python
class Neural_Network(object):
    #Sinir Ağları adında, nesne alan bir sınıf oluşturduk,
    #Giriş değerlerini atayacak bir initial metod tanımlayalım.
    def __init__(self):
        #Hiperparametreleri tanımlayalım.
        self.inputLayerSize = 2
        #2 Adet Giriş Ağırlığımız var, 2 giriş parametresi olduğunu belirtiyoruz.
        self.outputLayerSize = 1
        #1 Çıkış yhat skaler değerimiz için 1 parametre alacağını belirtiyoruz.
        self.hiddenLayerSize = 3
        #3 Ara Katmanımız bulunmakta.
    def forward(self, X):
        #Giriş değerlerinin Ağımızda ilerlemesi için forward metodunu kullanırız.
        self.z2 = np.dot(X, self.W1)
        #İkinci katman için W1 ağırlıklarını dot product olarak üretiyoruz
        self.a2 = self.sigmoid(self.z2)
        #İkinci katman için Aktivasyon değerini hesaplıyoruz
        self.z3 = np.dot(self.a2, self.W2)
        #Üçüncü katman için W2 ağırlıklarına ikinci katmandaki aktivasyon
        #değerini dot product olarak ürettiriyoruz
        yHat = self.sigmoid(self.z3)
        #yHat bizim tahmin sonucumuz.
        return yHat
```

Burada numpy sayesinde gönderdiğimiz değişken skaler, vektör ya da matris olabilir, numpy aktivasyon fonksiyonunu her element için tek tek uygular, sonucu ise aynı boyutta döndürür.



Tablo 5: Matris çarpımı ile giriş vektörlerine ağırlıkların dağıtılması

```
testInput = np.arange(-6,6,0.01)
plot(testInput, sigmoid(testInput), linewidth= 2)
grid(1)
#Örnek sigmoid fonksiyonu
```




```
sigmoid(1)
```

```
0.7310585786300049
```

```
sigmoid(np.array([-1,0,1]))
```

```
array([ 0.26894142,  0.5       ,  0.73105858])
```

```
sigmoid(np.random.randn(3,3))
```

```
array([[ 0.41564745,  0.47579016,  0.60821143],
       [ 0.5578299 ,  0.6960842 ,  0.51123605],
       [ 0.78966134,  0.6266845 ,  0.42944555]])
```

İleri yayılım için elimizdeki ikinci formül ile ikinci katmandaki aktivasyon değeri a(2) değerini hesaplamak için f fonksiyonunu kullanıyoruz, a(2) değeri z(2) gibi 3'e 3'lük bir matris olacaktır.

$$a^{(2)} = f(z^{(2)})$$

İleri yayılımı tamamlamak için bir katmanı daha hesaplamamız gerekiyor. İkinci katmandaki ağırlıklarımıza bir kere daha aktivasyon fonksiyonunu uyguladıktan sonra yHat değerine ulaşacağız.

$$z^{(3)} = a^{(2)} W^{(2)}$$

$$\hat{y} = f(z^{(3)})$$

W2 ağırlıkları her sinaps için bir satır olmak üzere 3'e 1'lik bir matris olacaktır. Aktivasyon fonksiyonu uygulandıktan sonra elimizde z(3) üçüncü katman değeri olacaktır. Z(3)'ün her bir örnek için birer değer olmak üzere 3 aktivite değeri olacaktır. z(3)'e aktivasyon fonksiyonunu uyguladıktan sonra yHat değerini, yani sonucu bulmuş olacağız.

```
class Neural_Network(object):
    def __init__(self):
        #Hiperparametrelerimiz
        self.inputLayerSize = 2
        self.outputLayerSize = 1
        self.hiddenLayerSize = 3

        #Ağırlıklarımız
        self.W1 = np.random.randn(self.inputLayerSize, self.hiddenLayerSize)
        self.W2 = np.random.randn(self.hiddenLayerSize, self.outputLayerSize)

    def forward(self, X):
        #İleri yayılım metodumuz
        self.z2 = np.dot(X, self.W1)
        self.a2 = self.sigmoid(self.z2)
        self.z3 = np.dot(self.a2, self.W2)
        yHat = self.sigmoid(self.z3)
        return yHat

    def sigmoid(self, z):
        #Sigmoid fonksiyonumuz
        return 1/(1+np.exp(-z))
```

Şu anda, çalıştığımız saat ve uyuduğumuz saat değerlerini rastgele üreten ve sonucu hesaplayan bir sınıfa sahibiz. Fakat tahmin değerlerimiz ağımız henüz eğitilmediği için hazır değil.

Kısım 3: Gradyen Azalım

```
yHat
#Tahmin değerlerimiz
```

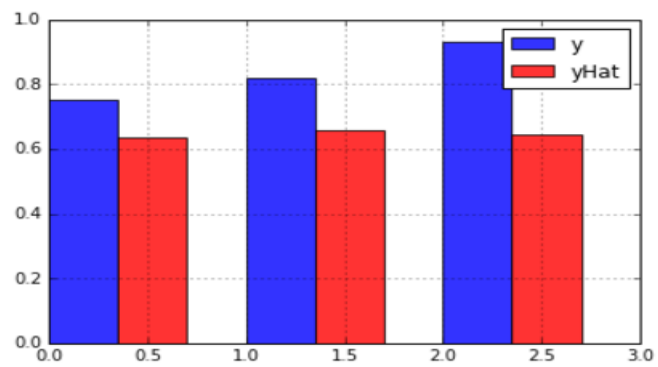
```
array([[ 0.63514767],
       [ 0.65568389],
       [ 0.644363  ]])
```

```
y
#Gerçek y değerlerimiz
```

```
array([[ 0.75],
       [ 0.82],
       [ 0.93]])
```

```
#yHat ile y arasındaki farkı bar chart ile gösterelim
bar([0,1,2], y, width = 0.35, alpha=0.8)
bar([0.35,1.35,2.35],yHat, width = 0.35, color='r', alpha=0.8)
grid(1)
legend(['y', 'yHat'])
#legend bir matplotlib fonksiyonudur. Geçici bir instance içinde değerlerimizi,
#legend handler'ı içinde argüman olarak tutmamızı sağlar.
```

```
<matplotlib.legend.Legend at 0x790969c278>
```



Tahminlerimiz oldukça kötü fakat, oldukça niteliği bize özünde, sonuçlarımızın gerçek sonuçlardan ne kadar kötü olduğunu anlatmaz. Bunun için hesabımızın sonunda bulduğumuz yHat değerinin gerçek sonuçtan ne kadar kötü olduğunu, maliyet fonksiyonu ile belirleriz. Maliyet fonksiyonu bize modelimizin ne kadar maliyetli olduğunu gösterir.

Bütünsel maliyet hesabı için her hata farkının karesini alır ve ikiye böleriz. Buna gradyen karesel maliyet fonsiyonu denir. Elimizde nicel bir maliyet sonucu olduğuna göre, artık görevimiz bu maliyeti minimize etmektir. Sinir ağlarında, bir ağı eğitmekten bahsettiklerinde, aslında hata fonksiyonunu minimize ettiklerini anlatırlar.

$$J = \sum \frac{1}{2}(y - \hat{y})^2$$

Maliyet fonksiyonumuz sadece iki niteliğe bağlı, sinapslardaki ağırlıklarımıza ve örnek sonuçlarımıza. Örnek sonuçlarını değiştiremeyeceğimiz için, maliyeti düşürmek için ağırlıkları manipüle etmemiz gerekmekte.

Elimizde olan 9 ağırlık değeri için öyle değerler vardır ki, bizim maliyet fonksiyonumuzun sonucunu minimum çıkarabilir. Bunun için yöneylem ve optimizasyonda gördüğümüz gibi, yapmamız gereken ağırlık değerlerini değiştirerek denemektir.

Burada karşımıza Curse of Dimensionality denen problem çıkmaktadır. Bu problemi görebilmek için aşağıdaki örnek üzerinden ilerleyelim.

```
import time

weightsToTry = np.linspace(-5,5,1000)
costs = np.zeros(1000)

startTime = time.clock()
for i in range(1000):
    NN.W1[0,0] = weightsToTry[i]
    yHat = NN.forward(X)
    costs[i] = 0.5*sum((y-yHat)**2)

endTime = time.clock()
```

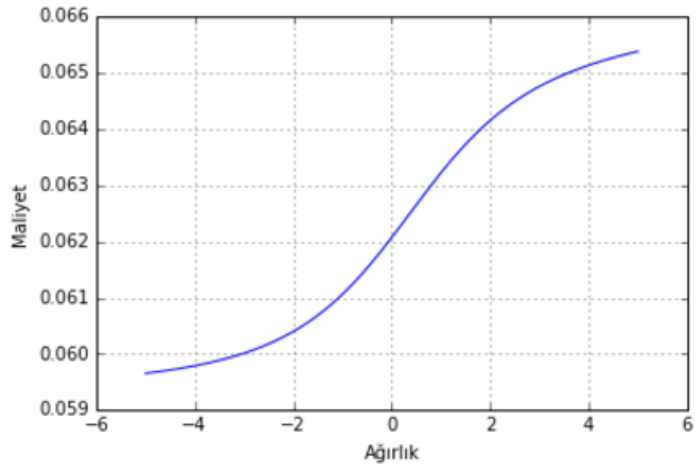
```
timeElapsed = endTime-startTime
timeElapsed
```

```
0.033715922940245946
```

1000 farklı değeri test etmek yaklaşık 0.04 saniye sürdü. Geniş bir varyasyonda değerlerimizi denediğimiz için bize en uygun gelen, yani hata maliyet fonksiyonunun sonucunu en düşük veren değeri seçebiliriz.

```
plot(weightsToTry, costs)
grid(1)
ylabel('Maliyet')
xlabel('Ağırlık')
```

```
<matplotlib.text.Text at 0x7909754b70>
```



Aslında burada sadece bir W1 için hesap yapıyoruz. İki ağırlık için deneme yapıyor olsak 1000x1000'den bir milyon farklı değeri denemek zorunda kalacağız. Hızlı bir bilgisayar için bile bu iş biraz zaman alıyor. Aşağıdaki örneğimizde iki ağırlığın minimum hata maliyet fonksiyon değeri için hesaplanmasını görüyoruz.

```
weightsToTry = np.linspace(-5,5,1000)
costs = np.zeros((1000, 1000))

startTime = time.clock()
for i in range(1000):
    for j in range(1000):
        NN.W1[0,0] = weightsToTry[i]
        NN.W1[0,1] = weightsToTry[j]
        yHat = NN.forward(X)
        costs[i, j] = 0.5*sum((y-yHat)**2)

endTime = time.clock()
```

```
timeElapsed = endTime-startTime
timeElapsed
```

```
30.322623532681973
```

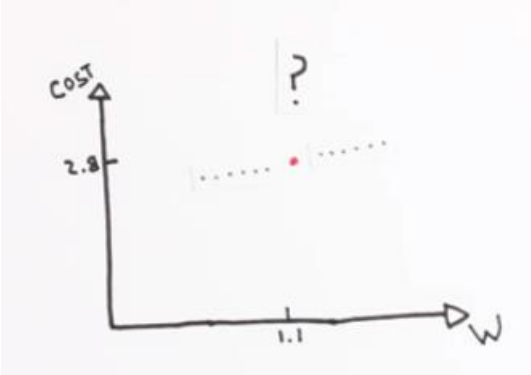
Biz ağırlıkları ekledikçe hesaplama süresi oldukça hızlı bir biçimde artıyor. Bu denemede hesaplama süresi yaklaşık 30 saniye sürdü. Fakat ağırlık değerleri arttıkça sonucu hesaplamak için çok daha uzun süre gerekeceğini ölçebiliriz.

```
0.04*(1000**(9-1))/(3600*24*365)
```

```
1268391679350583.5
```

Basit bir sinir ağında bile 9 ağırlık için hesap yaptığımızda, hesaplama süresi ömrümüzden uzun süreceğini görebiliyoruz. Kaba kuvvet optimizasyonu bu noktada etkili olmayacaktır.

1 boyutlu hesaplamamıza dönelim, spesifik bir w ağırlığı için maliyet fonksiyonumuzun ne sonuca varacağını hesaplayalım. w=1.1 değeri için Maliyet j=2.8 çıkmakta.



Tablo 6: Zamana bağlı maliyet fonksiyonunun değişim eğilimi

w değerini arttırdığımızda mı yoksa azalttığımızda mı daha küçük bir j değeri elde ederiz? Eğer bunu öngörebilirsek, doğrudan ağırlık değerini azaltarak ya da arttırarak sonuca daha çabuk varabiliriz. Bu yönteme numerik tahmin denmekte. Fakat bizim problemimiz için daha iyi bir tahmin yolu bulunmakta.

$$z^{(2)} = XW^{(1)}$$

$$a^{(2)} = f(z^{(2)})$$

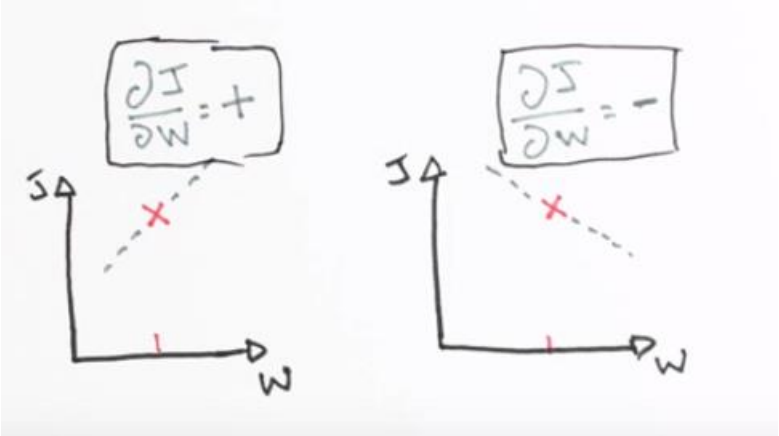
$$z^{(3)} = a^{(2)} W^{(2)}$$

$$\hat{y} = f(z^{(3)})$$

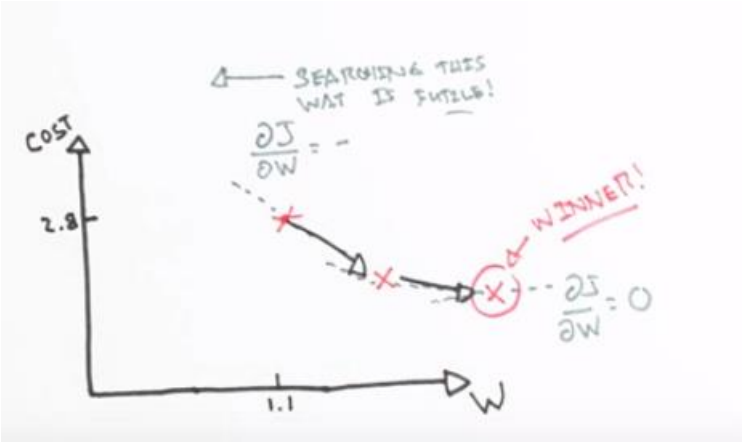
$$J = \sum \frac{1}{2}(y - \hat{y})^2$$

Elimizdeki 5 denklemi aslında tek bir denklem olarak düşünebiliriz.

$$J = \sum \frac{1}{2}(y - f(f(XW^{(1)})W^{(2)}))^2$$



Tablo 7: Maliyet Fonksiyonunun ağırlıklara diferansiyelinin yönelimi



Tablo 8: Gradyen Azalım

Bu metoda gradyen azalım denir. Tek bir boyutta etkileyici görünmese de, boyutlar arttığında bize değişimin hangi yönde olması gerektiğine dair hızlı çıkarımlar yapmamıza olanak verir. Karşımıza çıkacak olan sorun ise, herhangi bir yerel minimuma ulaştığımızda, en düşük hata maliyetini bulduğumuzu sanabiliriz. Bu da bizi global minimumdan uzaklaştırır.

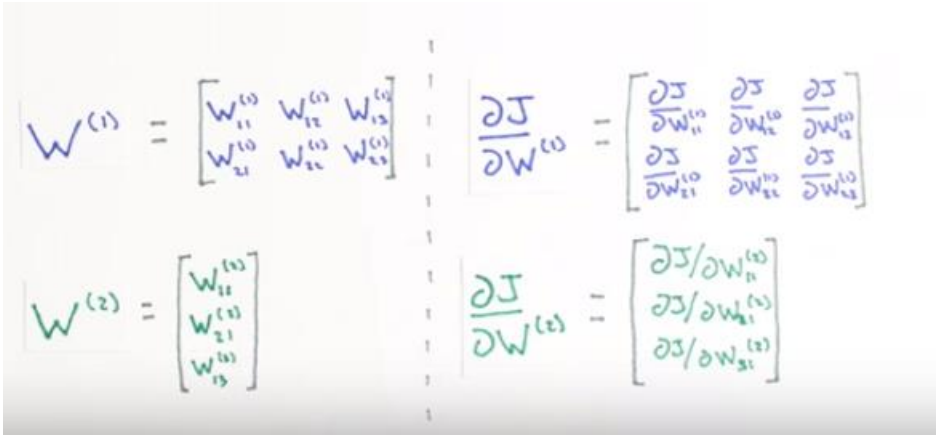
Bu sonuç da elde ettiğimiz hata maliyet grafiğimizin konveks olmaması demektir. y=x^2 gibi fonksiyonlar tek global minimum değerine sahip fonksiyonlardır. Daha üst boyutlu konveks fonksiyonlar da bulunmaktadır.

Elimizdeki verileri nasıl kullandığımıza göre maliyet fonksiyonumuzun konveks olup olmamasının önemi kalmadan hala iyi sonuçlar elde edebiliriz. Bunun için tek bir seferde hepsini denemek yerine her bir örneğimiz için tek tek maliyet fonksiyonlarını hesaplamamız gerekir. Bu yönteme Stokastik Gradyen Azalım denir.

Bölüm 4 – Geri Yayılım

Sembol	Matematiksel İfade	Tanım	Boyutlar
X	X	Giriş Verileri	(numExamples, inputLayerSize)
y	y	Hedef Veri	(numExamples, outputLayerSize)
W1	$W(1)$	Katman 1 ağırlıkları	(inputLayerSize, hiddenLayerSize)
W2	$W(2)$	Katman 2 ağırlıkları	(hiddenLayerSize, outputLayerSize)
z2	$z(2)$	Katman 2 aktivasyonu	(numExamples, hiddenLayerSize)
a2	$a(2)$	Katman 2 aktivitesi	(numExamples, hiddenLayerSize)
z3	$z(3)$	Katman 3 aktivasyonu	(numExamples, outputLayerSize)
J	J	Maliyet	(1, outputLayerSize)
dJdz3	$\partial J \partial z(3) = \delta(3)$	Maliyetin $z(3)$ 'e göre kısmi türevi	(numExamples,outputLayerSize)
dJdW2	$\partial J \partial W(2)$	Maliyetin $W(2)$ e göre kısmi türevi	(hiddenLayerSize, outputLayerSize)
dz3dz2	$\partial z(3) \partial z(2)$	$z(3)$ ün $z(2)$ e göre kısmi türevi	(numExamples, hiddenLayerSize)
dJdW1	$\partial J \partial W(1)$	Maliyetin $W(1)$ e göre kısmi türevi	(inputLayerSize, hiddenLayerSize)
delta2	$\delta(2)$	Geriyayılım Hata 2	(numExamples,hiddenLayerSize)
delta3	$\delta(3)$	Geriyayılım Hata 1	(numExamples,outputLayerSize)

Tablo 9: Değişkenler



Tablo 10: Ağırlık matrislerinin kısmi maliyet diferansiyelleri

Ağırlıklarımız W1 ve W2 olmak üzere iki matrise ayrılmış durumda. Ağırlıklara göre maliyet fonksiyonlarının kısmi türevlerini dJdW1 ve dJdW2 için ayrı ayrı hesaplamamız gerekiyor. Ağırlık değerlerimiz kadar gradyen değerlerimiz de olduğunda, dJdW1 ve dJdW2 , W1 ve W2 aynı boyutta matrisler olacaktır.

$$\frac{\partial J}{\partial W^{(2)}} = \frac{\partial \sum \frac{1}{2}(y-\hat{y})^2}{\partial W^{(2)}}$$

Diferansiyel toplamları, ayrı ayrı diferansiyellerin toplamına eşit olacağından.

$$\frac{\partial J}{\partial W^{(2)}} = \sum \frac{\partial \frac{1}{2}(y-\hat{y})^2}{\partial W^{(2)}}$$

Zincir kuralını uygulayarak,

$$\frac{\partial J}{\partial W^{(2)}} = -(y-\hat{y})\frac{\partial \hat{y}}{\partial W^{(2)}}$$

yHat bizim z3 aktivasyon fonksiyonumuzun sonucu olduğundan, dyHat/dW2 için zincir kuralını uygulayarak dyHat/dz3 çarpı dz3/dW2 olarak ifade edebiliriz.

$$\frac{\partial J}{\partial W^{(2)}} = -(y-\hat{y})\frac{\partial \hat{y}}{\partial z^{(3)}}\frac{\partial z^{(3)}}{\partial W^{(2)}}$$

z3 aktivasyon fonksiyonu için yHat'ın değişim miktarını bulmak için sigmoid fonksiyonumuzun da z'ye göre diferansiyelini almamız gerekir.

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

$$f'(z) = \frac{e^{-z}}{(1+e^{-z})^2}$$

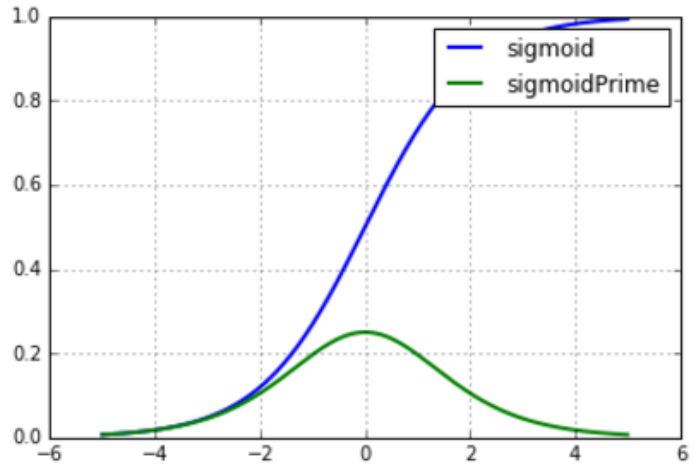
Sigmoid fonksiyonumuzun en keskin olduğu noktada, türevimiz en büyük değeri alacağından, z aktivasyon fonksiyonumuz sıfıra yaklaşır.

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

```
def sigmoidPrime(z):
    #Sigmoid fonksiyonunun türevi
    return np.exp(-z)/((1+np.exp(-z))**2)
```

```
testValues = np.arange(-5,5,0.01)
plot(testValues, sigmoid(testValues), linewidth=2)
plot(testValues, sigmoidPrime(testValues), linewidth=2)
grid(1)
legend(['sigmoid', 'sigmoidPrime'])
```

```
<matplotlib.legend.Legend at 0xb5ea0053c8>
```



Şimdi dyHat/dz3'ü f(z(3))'ün türevi ile değiştirebiliriz.

$$\frac{\partial z^{(3)}}{\partial W^{(2)}} = -(y - \hat{y}) f'(z^{(3)}) \frac{\partial z^{(3)}}{\partial W^{(2)}}$$

Son olarak dz3dW2, z'deki değişimi ifade etmektedir, z(3) = a(2).W(2) değerlerinin toplamı olduğundan, W(2) ile z(3) arasında, a'nın eğim olduğu bir ilişki vardır.

$$z^{(3)} = a^{(2)} W^{(2)}$$

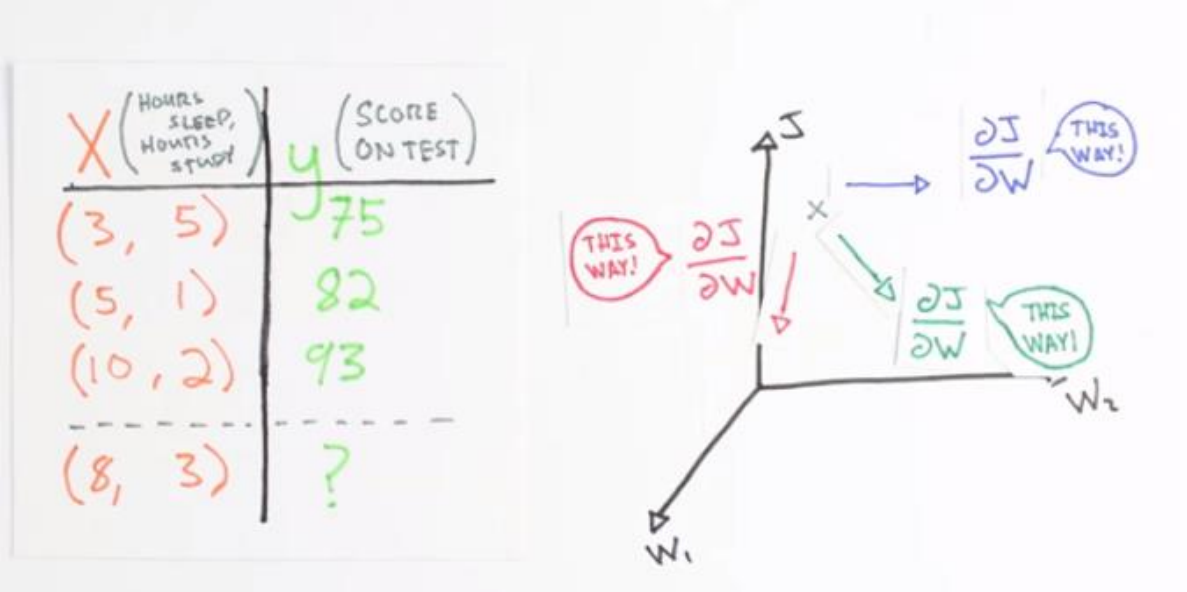
Aslında burada matematiksel olarak yaptığımız hesap, her ağırlıktaki hatayı bir adım geriye yaymaya çalışmaktır. Hatayı geriye yayarken, en büyük hataya sahip olan ağırlık değeri, gradyen azalım uyguladığımızda en çok değişecek olan değerlerdir.

Denklemin ilk kısmı için y-yHat çıkış değerimiz ile aynı boyutta, yani 3'e 1'lik bir matristir. F(z(3))'ün türevi de aynı olup 3'e 1'lik bir matristir, o yüzden ilk işlemimiz skaler çarpımdır. Sonuç olarak elde edeceğimiz matris de 3'e 1'lik geri yayılım hatası olan delta 3 değeridir.

$$\frac{\partial J}{\partial W^{(2)}} = (a^{(2)})^T \delta^{(3)}$$

$$\delta^{(3)} = -(y - \hat{y}) f'(z^{(3)})$$

Her bir örneğimiz bize gradyen azalımın hangi yönde olması gerektiğini ayrı ayrı söylemektedir.



Tablo 11: Gradyen azalım yönünü belirlemek

```
def costFunctionPrime(self, X, y):
    #Verilen X ve y için W ve W2'ye göre türev hesapla:
    self.yHat = self.forward(X)

    delta3 = np.multiply(-(y-self.yHat), self.sigmoidPrime(self.z3))
    dJdW2 = np.dot(self.a2.T, delta3)
```

Eğer daha derin ağlar elde etmek istiyorsak, burada hesapladığımız gibi, metotları birbirine eklememiz gerekmektedir.

```
# Whole Class with additions:
class Neural_Network(object):
    def __init__(self):
        #Define Hyperparameters
        self.inputLayerSize = 2
        self.outputLayerSize = 1
        self.hiddenLayerSize = 3

        #Weights (parameters)
        self.W1 = np.random.randn(self.inputLayerSize,self.hiddenLayerSize)
        self.W2 = np.random.randn(self.hiddenLayerSize,self.outputLayerSize)

    def forward(self, X):
        #Propogate inputs though network
        self.z2 = np.dot(X, self.W1)
        self.a2 = self.sigmoid(self.z2)
        self.z3 = np.dot(self.a2, self.W2)
        yHat = self.sigmoid(self.z3)
        return yHat

    def sigmoid(self, z):
        #Apply sigmoid activation function to scalar, vector, or matrix
        return 1/(1+np.exp(-z))

    def sigmoidPrime(self,z):
        #Gradient of sigmoid
        return np.exp(-z)/((1+np.exp(-z))**2)

    def costFunction(self, X, y):
        #Compute cost for given X,y, use weights already stored in class.
        self.yHat = self.forward(X)
        J = 0.5*sum((y-self.yHat)**2)
        return J

    def costFunctionPrime(self, X, y):
        #Compute derivative with respect to W and W2 for a given X and y:
        self.yHat = self.forward(X)

        delta3 = np.multiply(-(y-self.yHat), self.sigmoidPrime(self.z3))
        dJdW2 = np.dot(self.a2.T, delta3)

        delta2 = np.dot(delta3, self.W2.T)*self.sigmoidPrime(self.z2)
        dJdW1 = np.dot(X.T, delta2)

        return dJdW1, dJdW2
```

Şimdi dJ/dW değerini hesaplayarak, optimizasyon uzayında hangi yöne doğru gitmemiz gerektiğini hesaplayabiliriz.

```
NN = Neural_Network()
```

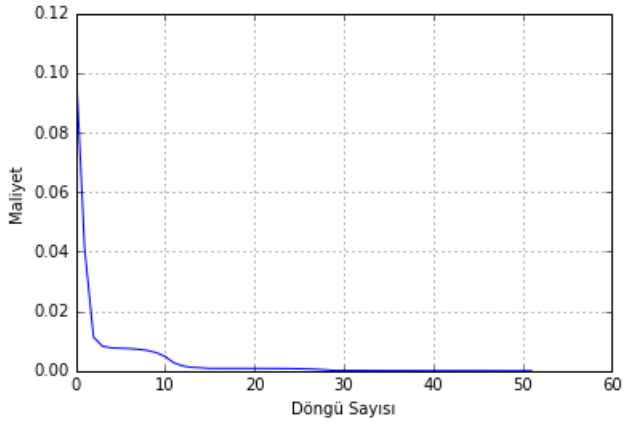
```
T = trainer(NN)
```

```
T.train(X,y)
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 52
         Function evaluations: 56
         Gradient evaluations: 56
```

```
plot(T.J)
grid(1)
xlabel('Döngü Sayısı')
ylabel('Maliyet')
```

```
<matplotlib.text.Text at 0xbd08140e10>
```




```
dJdW2
```

```
array([[-0.2126647 ],
       [-0.15672235],
       [-0.17063581]])
```

```
scalar = 3
NN.W1 = NN.W1 + scalar*dJdW1
NN.W2 = NN.W2 + scalar*dJdW2
cost2 = NN.costFunction(X,y)
```

```
print (cost1, cost2)
```

```
0.350478940245 0.665296667532
```

```
dJdW1, dJdW2 = NN.costFunctionPrime(X,y)
NN.W1 = NN.W1 - scalar*dJdW1
NN.W2 = NN.W2 - scalar*dJdW2
cost3 = NN.costFunction(X, y)
```

```
print (cost2, cost3)
```

```
0.585069978632 0.292698670705
```

Bölüm 5 – Eğitim

Klasik bilgi işleme yöntemlerinin çoğu programlama yoluyla hesaplamaya dayanmaktadır. Yani bir problemin çözümü için probleme yönelik bir algoritma geliştirilmelidir. Bunun yanında bu yöntemler tam tanımlı olmayan problemleri çözemez. Yapay sinir ağları ise belirli bir probleme göre programlanmadığı halde o problemi çözmeyi öğrenebilir.

Genel anlamda öğrenme, sinir ağının, giriş uyarıcılarını kullanarak kendini istenen sonuçları vermek üzere ayarlamasıdır. İşlem elemanı ve ağ yapısı tasarlandıktan sonra, YSA'nın öğrenme işlemi başlatılabilir. Öğrenme hemen hemen bütün yapay sinir Artanağlarının temelidir.

Öğrenme, ağdaki nöronların değiştirilmesi ile değil, nöronlar arasındaki bağlantı ağırlıklarının değiştirilmesi ile sağlanır. Tek bir nöronun çıkışının nasıl belirlendiği göz önüne alındığında; nöronun aktivasyon fonksiyonunun sabit olması koşuluyla çıkışını, yalnızca giden işaretin ve nörona giriş bağlantı ağırlıklarının belirlediği bilinmektedir. Nöronun gelen işarete doğru cevap vermesinde ve performansının artırılmasında en önemli elemanlar bağlantı ağırlıklarıdır.

Öğrenme yöntemleri başlıca iki başlıkta ele alınabilir:

Danışmanlı Öğrenme: YSA'da gerçek çıkış istenen çıkış ile kıyaslanır. Rasgele değişen ağırlıklar ağ tarafından öyle ayarlanır ki, bir sonraki döngüde gerçek çıkış ile istenen çıkış arasındaki fark azalsın. Öğrenme yöntemi, bütün işleme elemanlarının anlık hatalarının en aza indirmeye çalışır. Bu hata azaltma işlemi, kabul edilebilir doğruluğa ulaşana kadar ağırlıklar devamlı olarak derlenir.

Danışmansız Öğrenme: Danışmansız öğrenmede sistemin doğru çıkış hakkında bilgisi yoktur ve girişlere göre kendi kendisini örnekler. Danışmansız olarak eğitilebilen ağlar istenen ya da hedef çıkış olmadan giriş bilgilerinin özelliklerine göre ağırlık değerlerini ayarlar. Danışmansız öğrenmeye, Hebbian öğrenme, Grossberg öğrenme, Kohonen'in özörgütlemeli harita ağı örnek olarak verilebilir.

Sonraki sayfada adım adım öğrenme adımları için Python kodlarına devam edeceğiz.

```
from scipy import optimize
```

```
class trainer(object):
    def __init__(self, N):
        #Ağ için sınıf içi yerel bir değişken ata
        self.N = N

    def callbackF(self, params):
        self.N.setParams(params)
        self.J.append(self.N.costFunction(self.X, self.y))

    def costFunctionWrapper(self, params, X, y):
        self.N.setParams(params)
        cost = self.N.costFunction(X, y)
        grad = self.N.computeGradients(X,y)

        return cost, grad

    def train(self, X, y):
        #Callback fonksiyonu için yerel bir değişken ata
        self.X = X
        self.y = y

        #Maliyetleri yerleştirmek için boş bir dizi oluştur
        self.J = []

        params0 = self.N.getParams()
        #Parametreleri çağır
        #Eğitim için maksimum döngü sayısını belirle
        options = {'maxiter': 200, 'disp' : True}
        _res = optimize.minimize(self.costFunctionWrapper, params0, jac=True, method='BFGS', \
                                 args=(X, y), options=options, callback=self.callbackF)

        self.N.setParams(_res.x)
        self.optimizationResults = _res
```

```
NN = Neural_Network()
```

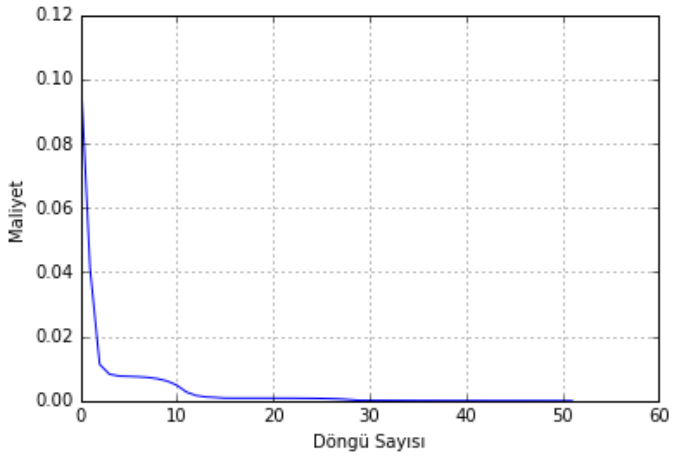
```
T = trainer(NN)
```

```
T.train(X,y)
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 52
         Function evaluations: 56
         Gradient evaluations: 56
```

```
plot(T.J)
grid(1)
xlabel('Döngü Sayısı')
ylabel('Maliyet')
```

```
<matplotlib.text.Text at 0xbd08140e10>
```




```
NN.costFunctionPrime(X,y)
```

```
(array([[  4.41116193e-07,  -6.69181296e-07,  -1.44317206e-06],
        [  1.21336982e-06,  -1.76243188e-06,  -3.90451115e-06]]),
 array([[  2.70220482e-06],
        [  2.10902384e-06],
        [  7.87454011e-07]]))
```

```
NN.forward(X)
```

```
array([[ 0.75001281],
       [ 0.82000216],
       [ 0.93000337]])
```

```
y
```

```
array([[ 0.75],
       [ 0.82],
       [ 0.93]])
```

Bulduğumuz sonuçların, gerçek değerlere çok yakın olduğunu fark ediyoruz. Bunun sebebi, modelimizi, giriş değerlerine göre eğitirken, aşırı eğitme, overfitting'e maruz bırakmamış olmamızdan kaynaklanıyor. Modelin overfitting'e maruz kalması, veri seti üzerinde ezberleme yaptığını gösterir.

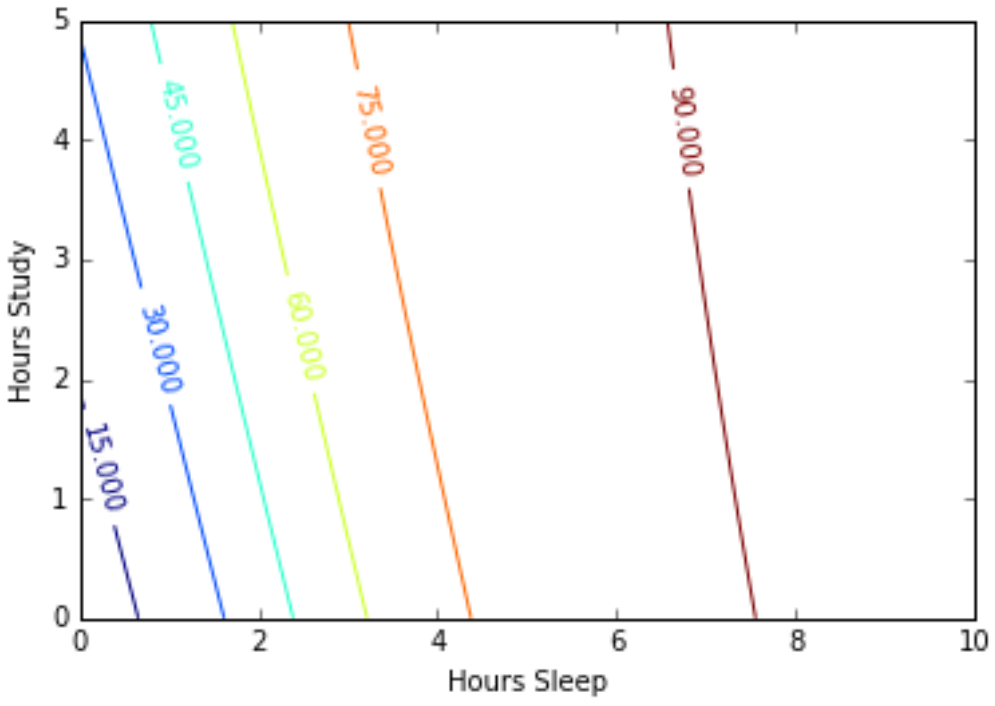
Overfitting makine öğrenmesinde temel sorunlardan biridir. Bu problemi aşmak istiyorsak, ağımızda yer alan tüm ağırlık sayımızın en az on katı kadar elimizde giriş değerleri olmalıdır. Çalışmamızda yer alan modelde 9 adet w ağırlıkları vardır, yani en az 90 darklı giriş satırına sahip olmalıyız. Overfitting'den kaçınmanın bir başka yolu döngü sayısını değiştirmek, hata maliyet fonksiyonlarını önceden belirlenmiş bir değere ulaşıldığında durdurmak, maliyet hata fonksiyonlarının gradyenlerinin önceden belirlenen bir değere ulaşması halinde tahmini durdurmak veya Cross-Validation hatasının en küçük değere ulaşmasını sağlamak örnek verilebilir.

```
#Uyku/Çalışma test kombinasyonları
hoursSleep = linspace(0, 10, 100)
hoursStudy = linspace(0, 5, 100)

#Eğitim verisinde olduğu gibi test verilerini de normalize et
hoursSleepNorm = hoursSleep/10.
hoursStudyNorm = hoursStudy/5.

#Sonucu meshgrid'e ata
a, b  = meshgrid(hoursSleepNorm, hoursStudyNorm)

#Sonucu ravel ile tek bir giriş matrisine ata
allInputs = np.zeros((a.size, 2))
allInputs[:, 0] = a.ravel()
allInputs[:, 1] = b.ravel()
```

```
allOutputs = NN.forward(allInputs)
```

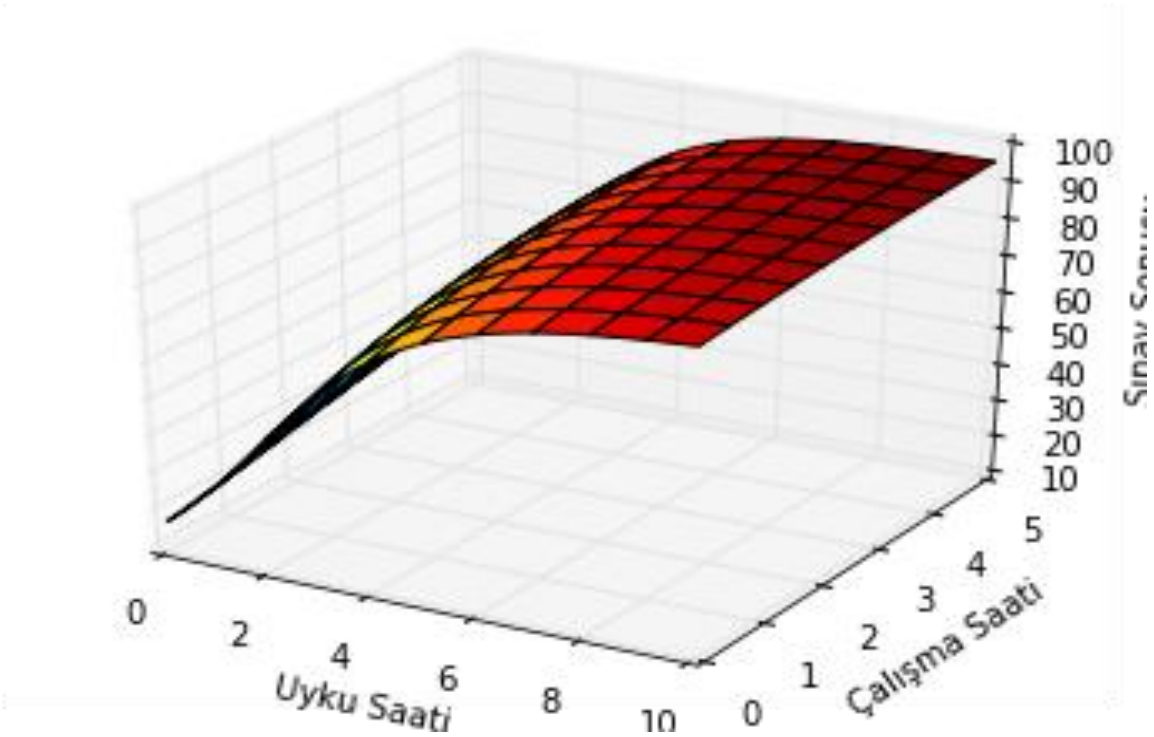
```
#Kontur Çizdir
yy = np.dot(hoursStudy.reshape(100,1), np.ones((1,100)))
xx = np.dot(hoursSleep.reshape(100,1), np.ones((1,100))).T

CS = contour(xx,yy,100*allOutputs.reshape(100, 100))
clabel(CS, inline=1, fontsize=10)
xlabel('Hours Sleep')
ylabel('Hours Study')
```

```
<matplotlib.text.Text at 0xbd09324978>
```



Tablo 12: Çalışma Saati ve Uyku Saati Değişkenlere Bağlı Sonuçların Kontur Grafiği Olarak Gösterimi

```
#3D plot:

from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.gca(projection='3d')

surf = ax.plot_surface(xx, yy, 100*allOutputs.reshape(100, 100), \
                       cmap=cm.jet)

ax.set_xlabel('Uyku Saati')
ax.set_ylabel('Çalışma Saati')
ax.set_zlabel('Sınav Sonucu')
```



Metodumuzun eğitim verilerine dayanarak oluşturduğu tahmin grafiği şekildeki gibi olmakla beraber, metodumuz hala gerçek sonuçlara gerçek sonuçlar üretmekten uzaktır. Çünkü gözlem kümesi sınırlı değişkenleri, farklı değişkenlerden etkileniyor olabilir. Gözardı edilen gürültüler sonuçları etkiliyor olabilir, bu sebeple, her uyku ve çalışma saati giriş değişkenlerine göre aynı sonucu beklemek gerçekçi olmayacaktır.

Buna örnek olarak üniversite sınavına giren öğrencileri gösterebiliriz. Başarıları her ne kadar uzun vadeli ve disiplinli çalışmaya dayalı olsa da sınav ortamında yaşayabilecekleri talihsizlikler, hastalıklar veya çok değişkene bağlı diğer şansa bağlı ihtimaller sebebiyle, modelimizin belli bir gürültüyü hesaba katarak, olasılıksal sonuç vererek tahmin belirtmesi daha gerçekçi sonuçlara ulaşmamızı sağlayacaktır. Modelimizde yer alan gözlem sayılarına ve ağırlıklara dayanarak giriş örneklemimizi genişletmemiz, örnek sayımızı arttırarak modelimizi test ve eğitim verilerini ayırarak uygulamamız, daha doğru sonuçlara ulaşmamızı sağlayacaktır.

Bu araştırmada ve dökümanın hazırlanmasında, Welch Labarotuvar'larının Yapay Sinir Ağları eğitimi referans alınmıştır. Stephen Welch'e e-posta ile ulaşılarak kullanım izni alınmıştır.