MCNA | Pitonca

PYTORCH İLE YAPAY SİNİR AĞLARININ TEMELLERİ

HALUK TANRIKULU

www.pitonca.com

Yazar tarafından verilen eğitimler:

Yapay Öğrenme ve Veri Analizi
Python ile Veri Analizine Giriş
Keras İle Derin Öğrenmeye Giriş
Makine Öğrenmesine Giriş

Python Programlama Dili
Python ile Programlama (Giriş, Orta, İleri)
Python ile Nesne Yönelimli Programlama
Python ile Soket Programlama
Matematikçiler için Python Programlama
Network Mühendisleri için Python
Python ile Database Programlama
Python ile DSP'ye Giriş
Python ile Yapay Zeka Uygulamaları
Python ile Fourier Dönüşümleri
Python | Flask ile Programlama
Python | Django ile Programlama

HTML5, JavaScript, CSS ile Web Programlama
HTML5 ile Web Tasarımı
HTML5, JavaScript ve CSS ile Mobile Programlama

PHP ve MySql İle Programlama
PHP'ye Giriş
PHP ve MySql ile Web Tasarımı

Lua Programlama
Lua ile Mobil Oyun Geliştirme
Mikrokontroller için Lua Programlama

Ağ Teknolojileri
MCNA Network | Yönlendirme | Ağ Anahtarlama
Huawei HCNA | HCNP
Cisco CCNA | CCNP
Network Fundamentals
Comptia Network+ | Cloud+

Siber Güvenlik
Comptia Security+
Ethical Hacking Eğitimleri (CEH)
Python ile Güvenlik Uygulamaları Geliştirme
Sistem Güvenliğine Giriş

İşletim Sistemleri
Linux (Mint, RedHat, Fedora, Kali, Debian, Ubuntu, SuSE)
Windows Servers

Derin Öğrenme Seti

PyTorch ile Yapay Sinir Ağlarının Temelleri

Ver. 0.8 – 2022

Haluk Tanrıkulu

www.haluktanrikulu.com
tanrikul@metu.edu.tr

MCNA | Pitonca

info@pitonca.com | 312 256 16 16
www.mcnatech.com | www.pitonca.com

Ankara 2022

22 21 20 19 18 17 16 15 14 13 12 11

Bıdıklarıma,

İçindekiler

Bölüm 1 - Giriş

Bölüm 2 - İleri Beslemeli Sinir Ağları

Bölüm 3 - Evrişimli Sinir Ağları (ESA)

Bölüm 4 - Tekrarlayan Sinir Ağları (TSA)

MCNA ve Pitonca Education’da Haluk Tanrıkulu tarafından “Deep Learning with Python” dersinde kullanılmak üzere hazırlanmıştır.

Derin Öğrenme Seti:

Aşağıdaki üç kitaptan oluşmaktadır.

Bölüm 1

Giriş

GİRİŞ

Bu bölümde, sinir ağlarının nasıl çalıştığını, bu kadar çok farklı çözüm türüne nasıl uygulanabileceğini ve PyTorch'un nasıl kullanılabileceğini göstereceğiz.

Son birkaç yılda popüler hale gelen birkaç yazılım kitaplığı ve araç takımı vardır. Python, makine öğrenimini içeren projelerin çoğu için en popüler seçimdir ve derin öğrenme için PyTorch, yakın geçmişte popülerliği artan rakip araçlardan biri.

Çok basit bir tanımla sinir ağları, derin öğrenme algoritmalarının kalbinde yer alan birbirine bağlı hesaplama düğümleridir. Sinir ağlarının en temel öğesi, çok temel vektör aritmetik işlemlerini gerçekleştiren algılayıcı olarak adlandırılır. Algılayıcılar, daha fazla hesaplama için birbirlerinin sonuçlarına bağlı olmak üzere bir araya getirilebilir ve böylece bilgi işlem birimleri katmanlarında düzenlenebilir. Bu tür ağlara sinir ağları denir.

Tasarımın basitliği, sinir ağlarının ana güç ve popülerlik kaynağıdır. Ancak ağ büyüdükçe hesaplamalar artacak, sinir ağının programlaması ve ilgili yeni çerçevelerin devreye girmesi ile karmaşıklık artacaktır.

PyTorch kullanımı genellikle basit olduğundan en popüler Python araçlardan biridir. PyTorch aynı zamanda bazı durumlarda diğer popüler derin öğrenme kitaplıklarından daha hızlıdır.

PyTorch Kurulumu

PyTorch'u kurmanın en çok tercih edilen yollarından biri Anaconda dağıtımının paket yöneticisi aracı olan conda'yı kullanmaktır. Anaconda

kurulu değilse, sisteminize uygun paketi Anaconda web sitesinden (***https://www.anaconda.com/products/distribution***) indirebilirsiniz. Anaconda, güçlü ortam yönetimi ve paket yönetimi yetenekleri sağlayan popüler bir Python dağıtımıdır.

Anaconda'yı kurduktan sonra, şu komutu kullanabilirsiniz:

```
conda install pytorch torchvision torchaudio cudatoolkit=10.2 -c pytorch
```

Bu, conda'ya PyTorch'u ve yüksek performanslı gpu tabanlı oluşturmak için bir ortam sağlayan cudatoolkit dahil olmak üzere gerekli kitaplıkları kurmasını söyleyecektir.

Alternatif olarak, Python'nın paket yöneticisi pip'i kullanarak kurulum yapabilirsiniz.

```
pip3 install torch torchvision torchaudio
```

Kurulum için daha fazla alternatif yol keşfetmek için, sisteminize ve gereksinimlerinize bağlı olarak kurulumunuzu yapılandırmanız için size daha fazla seçenek sunan PyTorch'un Başlarken sayfasına (***https://pytorch.org/get-started/locally/***) başvurabilirsiniz.

Kurulumdan sonra, PyTorch kurulumunuzun beklendiği gibi çalıştığını doğrulamak için basit bir test gerçekleştirebilirsiniz.

Torcu içe aktarmak ve PyTorch sürümünü doğrulamak için Jupyter Notebook veya Python Shell'i açın ve aşağıdaki satırdakileri yazın.

```
import torch
torch.__version__
Out: '1.9.1'
```

PyTorch Temelleri

PyTorch'ta tensör, herhangi bir genel çok boyutlu diziye atıfta bulunan bir terimdir. Pratikte, tensör, bir vektör uzayıyla ilgili cebirsel nesne kümeleri arasında çok doğrusal bir ilişki kurar. PyTorch'ta tensör, girdileri, çıktıları ve ayrıca model parametrelerini kodlayan birincil veri yapısıdır.

Tensörleri oluşturmak

NumPy'deki Ndarray'lere benzer. Bir Python listesinden veya çok boyutlu listelerin listesinden bir tensör oluşturabilirsiniz. Ayrıca mevcut bir NumPy dizisinden bir tensör oluşturabilirsiniz.

```
mylist = [1,2,3,4]
```

```
mytensor = torch.tensor(mylist)
mytensor
Out: tensor([1, 2, 3, 4])
```

Tahmin edebileceğiniz gibi, bir NumPy dizisinden bir tensör de oluşturabilirsiniz.

```
import numpy as np
myarr = np.array([[1,2],[3,4]])
mytensor_2 = torch.from_numpy(myarr)
mytensor_2

Out: tensor([[1, 2],
[3, 4]], dtype=torch.int32)
```

NumPy dizisinden bir tensör oluşturduğunuzda, bu yeni bir bellek konumuna kopyalanmaz - ancak hem dizi hem de tensör aynı bellek konumunu paylaşır. Tensörde herhangi bir değişiklik yaparsanız, oluşturulduğu orijinal diziye yansıtılacaktır.

```
mytensor_2[1,1]=5
myarr
Out: tensor([[1, 2],
[3, 5]], dtype=torch.int32)
```

Tersine, aynı verileri paylaşan bir NumPy dizi nesnesi döndürmek için **mytensor_2.numpy()** öğesini de kullanabilirsiniz.

NumPy Ndarray'ler gibi PyTorch tensörleri de homojendir; yani, tensördeki tüm öğeler aynı veri türüne sahiptir. NumPy'nin dizi oluşturma yöntemlerine benzer başka tensör oluşturma yöntemleri de vardır.

Bu, basit bir tensör oluşturmaya bir örnektir.

```
torch.zeros((2,3))
Out: tensor([[0., 0., 0.],
[0., 0., 0.]])
```

,

Bu, tüm değerleri sıfır olarak içeren 3x3 şeklinde bir tensör oluşturacaktır. NumPy'deki benzer bir işlev **np.zeros(3,3)**'tür. 3x3 şeklinde bir dizi döndürür.

Gösterim benzer olsa da, tensörler PyTorch'ta veri temsilinin birincil birimidir. Bir dizi veya kullanıcı tanımlı boyutta rastgele değerler oluşturmak için benzer işlevleri kullanabilirsiniz.

```
torch.ones((2,3))
Out: tensor([[1., 1., 1.],
[1., 1., 1.]])
```

```
torch.rand((2,3))
Out: tensor([[0.0279, 0.5261, 0.9984],
[0.7442, 0.3559, 0.3686]])
```

PyTorch ayrıca başka bir tensörün özelliklerine (şekil gibi) sahip bir tensörü oluşturmak veya başlatmak için bir yöntem içerir.

```
torch.ones_like(mytensor_2)
Out: tensor([[1, 1],
[1, 1]], dtype=torch.int32)
```

Tensör İşlemleri

PyTorch tensörleri, NumPy'nin dizilerine benzer şekilde birkaç işlemi destekler - ancak yetenekleri daha fazladır. Ölçekleyicilerle aritmetik işlemler yayınlanır, yani tensörün tüm öğelerine uygulanır. Uyumlu şekillerin tensörleri arasındaki matris işlemleri benzer bir şekilde uygulanır.

```
myarr = np.array([[1.0,2.0],[3.0,4.0]])
tensor1 = torch.from_numpy(myarr)
tensor1+1
Out: tensor([[2., 3.],
[4., 5.]], dtype=torch.float64)

tensor1/ tensor1
Out: tensor([[1., 1.],
[1., 1.]], dtype=torch.float64)

tensor1.sin()
Out: tensor([[ 0.8415, 0.9093],
[ 0.1411, -0.7568]], dtype=torch.float64)

tensor1.cos()
Out: tensor([[ 0.5403, -0.4161],
[-0.9900, -0.6536]], dtype=torch.float64)

tensor1.sqrt()
Out: tensor([[1.0000, 1.4142],
[1.7321, 2.0000]], dtype=torch.float64)
```

Verileri tanımlamaya yönelik işlevler de benzer şekilde kullanılabilir:

```
mean, median, min_val, max_val = tensor1.mean(), tensor1.median(), tensor1. min(), tensor1.max()

print ("Statistical Quantities: ")
print ("Mean: {}, \nMedian: {}, \nMinimum: {}, \nMaximum: {}".format(mean, median, min_val, max_val))
print ("The 90-quantile is present at {}".format(tensor1.quantile(0.5)))
```

Bu işlemler aşağıdaki çıktıyı verir:

```
Statistical Quantities:
Mean: 2.5,
Median: 2.0,
Minimum: 1.0,
Maximum: 4.0
The 90-quantile is present at 2.5
```

NumPy'ye benzer şekilde PyTorch, tensörlere katılmak için cat, hstack, vstack vb. gibi işlemler de sağlar. İşte örnekler:

```
tensor2 = torch.tensor([[5,6],[7,8]])
torch.cat([tensor1, tensor2], 0)
```

Bu yöntem iki tensörü birleştirir. Birleştirmenin yönü (veya ekseni) ikinci argüman olarak sağlanır. 0, tensörlerin dikey olarak birleştirileceğini, 1 ise yatay olarak birleştirileceğini belirtir.

```
tensor([[1., 2.],
[3., 4.],
[5., 6.],
[7., 8.]], dtype=torch.float64)
```

Diğer benzer işlevler, iki veya daha fazla tensörü yatay veya dikey olarak birleştirmek için de kullanılabilen hstack ve vstack'tir.

```
torch.hstack((tensor1,tensor2))
Out: tensor([[1., 2., 5., 6.],
[3., 4., 7., 8.]], dtype=torch.float64)

torch.vstack((tensor1,tensor2))
Out: tensor([[1., 2.],
[3., 4.],
[5., 6.],
[7., 8.]], dtype=torch.float64)
```

Tensörün şeklini değiştiren bir yeniden şekillendirme işlevi var. Tensörü isteğe bağlı sayıda sütun içeren tek bir satıra dönüştürmek için reshape(1,-1) olarak kullanabiliriz:

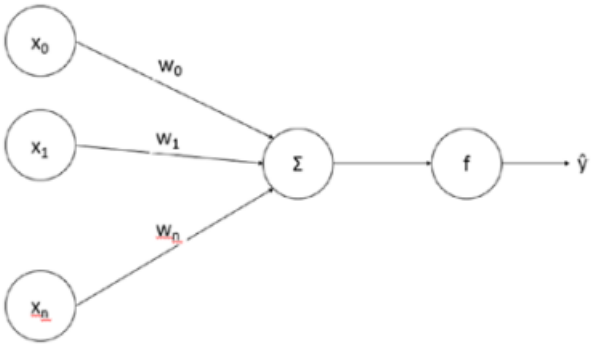
```
torch.reshape(tensor1, (1, -1))
Out: tensor([[1., 2., 3., 4.]], dtype=torch.float64)
```

İleriki bölümlerde ve bölümlerde kullanımlarına göre daha fazla işlemi tartışmaya devam edeceğiz.

Perceptron (Algılayıcılar)

Şekil 2'de gösterilen Perceptron, sinir ağının en basit şeklidir. Genellikle bir veri öğesinin özelliklerini girdi olarak tanımlayan bir veya daha fazla miktar alır, üzerinde basit bir hesaplama yapar ve tek bir çıktı üretir.

Algılayıcının en basit biçimi, tek katmanlı bir algılayıcıdır - yalnızca doğrusal olarak ayrılabilir verileri sınıflandırabilmesine rağmen, anlaşılması kolay, çalıştırılması hızlı ve uygulanması basittir.



Şekil 1. Bileşen hesaplamaları ile basit bir algılayıcı yapısı

Şekil 1'de gösterildiği gibi, perceptron, özelliklerin [x_1, x_2, x_3 … x_n] olarak temsil edildiği ve özelliklerin . [w_1, w_2, w_3, … w_n] şu şekilde temsil edildiği bir ağırlık vektörü ile x girdi vektörüne dayalı olarak sınıf etiketini tahmin etmek için basit bir hesaplama uygular. Genellikle, w_0 olarak girdiye bağlı olarak çıktıyı etkilemeyen ek bir sapma terimi ekleriz. Hesaplamayı kolaylaştırmak için, değeri 1 olarak ayarlanan x0 olarak bir girdi özelliği ekleriz. Böylece, iki vektör, x ve w, basit bir adım fonksiyonuna dayalı nihai çıktıya yol açar:

$$\hat{y} = f(x) = \begin{cases} 1, & x^t w > 0 \\ 0, & otherwise \end{cases}$$

Burada x, n+1 boyutlu girdi vektörüdür, w ağırlık vektörüdür ve öğrenme sürecinin amacı, sonuçları eğitim etiketleriyle karşılaştırarak hesaplanan hatanın en aza indirilmesi için mümkün olan en iyi ağırlıkları öğrenmektir.

Algılayıcıyı eğitmek için önce ağırlıkları rastgele değerlerle başlatıyoruz. Daha önce gösterilen formüle dayalı olarak tahmin edilen çıktıyı bulmak için eğitim veri setini kullanırız. Algoritma henüz doğru ağırlık setini öğrenmediği için sonuçlar beklediğimizden çok uzak olabilir ve bu da bir hataya yol açabilir. Sonraki yinelemelerdeki gürültüyü azaltmak için, mevcut çıktılara dayalı olarak ağırlıklara aşağıdaki güncellemeyi uygularız:

$$w = w + \alpha(y - \hat{y}).x$$

Burada ayrıca ağırlıkların ne kadar ciddi şekilde etkilendiğini kontrol eden bir adım parametresi olan **α** ekledik. Bu işlemi önceden belirlenmiş bir sayıda (veya yakınsamaya kadar) yinelemeli olarak tekrarlıyoruz ve sonuna kadar yeterince iyi bir ağırlık vektörüne ulaşmayı umuyoruz, bu da düşük bir hataya yol açabilir.

Çıktıyı tahmin etmek için, x özelliklerini aynı hesaplamaya eklememiz yeterlidir. Bu hesaplama işlevine **adım işlevi** denir. Çoğu durumda, sigmoid işlevini görmeyi tercih edebilirsiniz, yani:

$$y = \frac{1}{1 + e^{-w.x}}$$

Bunları önce temel Python ve NumPy kullanarak nasıl programlayacağımızı görelim ve daha sonra aynısını yapmak için PyTorch'u kullanacağız.

Python'da Perceptron

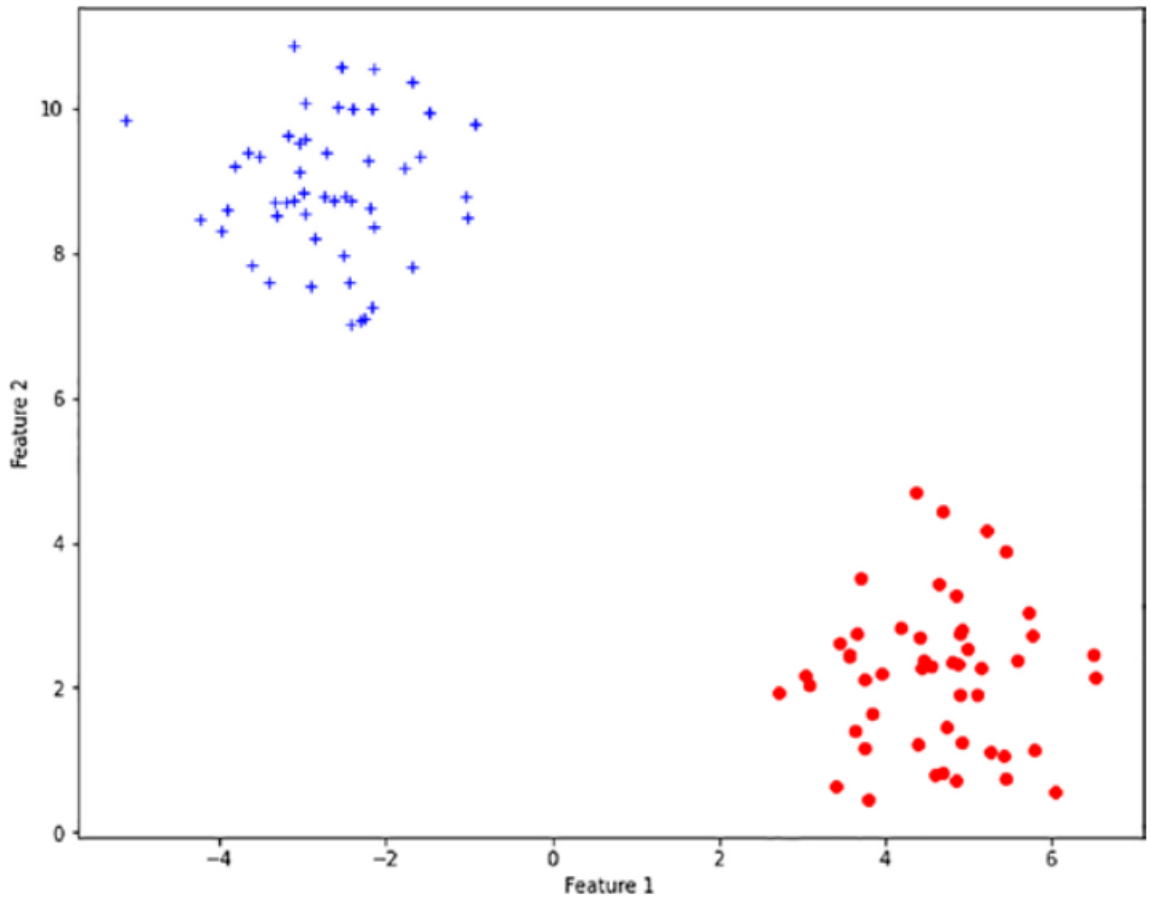
İlk önce Scikit-learn'in veri kümesi modülünü kullanarak basit bir ayrılabilir veri kümesi oluşturacağız. Daha önce kullandığımız diğer veri kümelerini kullanabilirsiniz.

```
from sklearn import datasets
import matplotlib.pyplot as plt
X, y = datasets.make_blobs(n_samples=100,n_features=2, centers=2,
                                        random_ state=42, shuffle=1)
```

Bu satırlar, iki ana bloğa bölünmüş iki özelliğe sahip 100 veri satırı oluşturacaktır. Algılayıcıyı oluşturmadan önce onu görselleştirelim.

```
fig = plt.figure(figsize=(10,8))
plt.plot(X[:, 0][y == 0], X[:, 1][y == 0], 'b+')
plt.plot(X[:, 0][y == 1], X[:, 1][y == 1], 'ro')
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
```

Veri noktaları, Şekil 2'de gösterildiği gibi açıkça görülebilir ve sınıflandırma sınırını basit tutmak için kasıtlı olarak bu tür rastgele oluşturulmuş veri setini seçtik.



Şekil 2. make_blobs kullanılarak oluşturulan basit veri kümesi

X ve y'miz var. Sıfırdan fit() ve predict() yöntemlerini oluşturalım. Ağırlıkların, geçerli adımda her bir veri noktasının tahmin edilen değerine göre güncellendiğini biliyorsunuz. $w+\alpha(y-\hat{y}).x$ formülünde. x ve y'yi zaten biliyoruz; α, adımları ayarlamak için yapılandırabileceğiniz bir hiperparametredir ve w, bu süreçte öğreneceğimiz ağırlıklar kümesidir.

Veri kümemiz iki özellik içerdiğinden, ağırlık vektöründe üç (sapma terimi için 2 + 1) ağırlığa ihtiyacımız var. Önce tahmin fonksiyonunu uygulayalım.

```
def predict(X, weight):
        return np.where(np.dot(X, weight) > 0.0, 1, 0)
```

Bu, ilk önce girdi verileri ile ağırlık arasındaki çarpımı hesaplayarak ve ardından elde edilen ürünün 0'dan büyük olup olmadığını karşılaştırmak için durumu uygulayarak $\hat{y}$ formülünü uygular. Bu fonksiyon, doğru değeri bulmamıza yardımcı olmanın yanı sıra tahmin fonksiyonu olarak da işlev görebilir.

Ağırlıkları güncellemek için Veri kümesinde hala iki sütun olmasına rağmen, üç ağırlık olduğunu unutmayın. Sapmayı telafi etmek için bir sütun

ekleyeceğiz, böylece 100x3'lük bir şekle yol açacağız. Ağırlıkları rastgele değerler alarak başlatacağız.

```
X = np.concatenate( (np.ones((X.shape[0],1)), X), axis=1)
weight = np.random.random(X.shape[1])
```

Başlatmadan sonra, önceden belirlenmiş sayıda yineleme (veya dönem) kadar yinelemeli bir süreç çalıştıracağız ve her yinelemede her noktayı işleyeceğiz ve ağırlıkları güncelleyeceğiz. fit() yöntemi şimdi aşağıdaki gibi görünmelidir:

```
def fit(X, y, niter=100, alpha=0.1):
        X = np.concatenate( (np.ones((X.shape[0],1)), X), axis=1)
        weight = np.random.random(X.shape[1])
        for i in range(niter):
                err = 0
                for xi, target in zip(X, y):
                        weight += alpha * (target - predict(xi, weight)) * xi
        return weight
```

Kodu, ağırlıkları dahili olarak depolayabilecek bir sınıf olarak yapılandırmadık - ancak tahmin yöntemine sağlanabilecek şekilde geri döndürmek zorundayız. Ağırlıkları öğrenmek için şimdi arayabiliriz.

```
w = fit(X,y)
w
Out: array([ 0.21313539, 0.96752865, -0.84990543])
```

W, sapmayı temsil eden üç değere ve her özellik için karşılık gelen katsayıya sahip bir ağırlık vektörüdür. Çıktıyı tahmin etmek için predict() yöntemini kullanabiliriz. Algılayıcımızın onları nasıl etiketlediğini karşılaştırmak için X'ten bazı rastgele öğeler seçelim:

```
random_elements = np.random.choice(X.shape[0], size=5, replace=False)
X_test = X[random_elements, :]
```

X_test şimdi veri kümesinden rastgele beş satır içerecek. Tahmin yöntemini çağırmadan önce, bunlarla birlikte ek bir sütun eklememiz gerekecek.

```
X_test = np.concatenate( (np.ones((X_test.shape[0],1)), X_test), axis=1)
```

Şimdi tahmin yöntemini çağıralım ve sonuçları gerçek değerlerle karşılaştıralım.

```
print (predict(X_test, w))
print (y[random_elements])
Out:
[0 0 1 0 0]
[0 0 1 0 0]
```

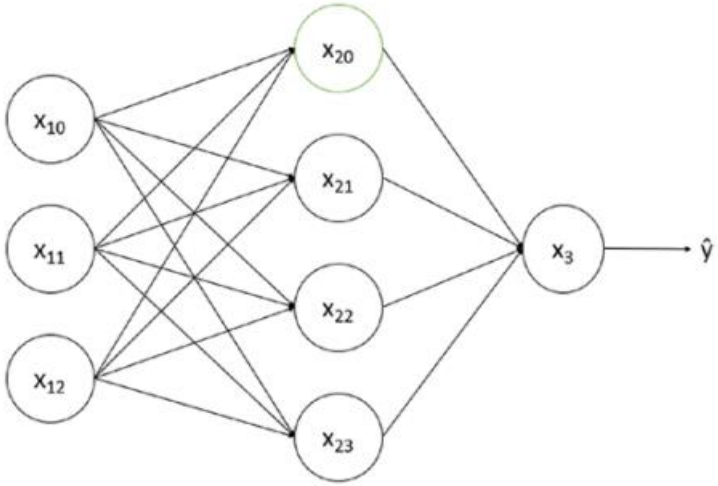
Veri kümesi çok basit olduğu için sonuçlar iyi görünüyor - ancak, karar sınırlarının çok net olmadığı durumlarda basit bir algılayıcının iyi performans göstermediğini unutmayın. Sonraki bölümlerde daha karmaşık bir sinir ağı oluşturmak için bu tür basit hesaplama birimlerinin nasıl birleştirileceğini göreceğiz.

Yapay Sinir Ağları

Basit bir algılayıcı, ağırlıklı özelliklerin toplamını birleştirerek bir fonksiyona geçirmeye çalıştığında, veri kümesinin her bir özelliğinin ne kadar önemli olduğunu tek bir eşik mantık birimi aracılığıyla öğrenir.

Python'da if-else koşulları tarafından uygulanan basit bir adım işlevi veya NumPy kullanılırken *np.where* işlevi kullandık. Bir özellik setinin çıktıyı nasıl ve ne zaman ürettiğini veya nöronu etkinleştirdiğini manipüle etmek için bir sigmoid işlevi veya başka alternatif işlevler kullanabiliriz. Bunun gibi birden fazla aktivasyonu birleştirebilir ve bunları tam bağlantılı bir katman şeklinde bağlayabiliriz.

Bu tür ağlar, basit bir algılayıcı kullanmanın basitliğini aşar ve birden çok tam bağlantılı katman oluşturmanıza izin verir, böylece gizli katmanların oluşturulmasına yol açar, böylece Şekil 3'te gösterildiği gibi çok katmanlı bir algılayıcı oluşturulur. Çıktı, sonraki katmanda yeni bir ağırlık seti tarafından daha fazla manipüle edilen bir sonraki katmana girdi olabilir.



Şekil 3. Tek çıkış birimine sahip basit bir çok katmanlı algılayıcı

Hesaplama birimleri, daha çeşitli derin sinir ağları oluşturmak için farklı bir şekilde düzenlenebilir. Evrişimli sinir ağları (CNN'ler), bir etkinleştirme haritası oluşturmak için filtreyi kaydırarak bir giriş görüntüsüne filtre uygulayan özel bir katman türü kullanır. CNN'ler, birçok bilgisayarla görme

(CV) veya doğal dil işleme (NLP) uygulamalarında oldukça tercih edilen bir seçimdir.

Onlar hakkında daha ayrıntılı olarak ilerleyen bölümler bahsedeceğiz.

Bir başka popüler sinir ağı mimarisi, durum ve bir girdi örneğinin kombinasyonunu kullanarak bir çıktı üretmek için değeri giriş verilerine dayalı olarak korunan bir dahili duruma sahip olan **tekrarlayan sinir ağı** (RNN) olarak adlandırılır. Bu ayrıca dahili durumu güncelleyebilir ve gelecekteki çıktıları etkileyebilir. Bu, NLP uygulamalarında son derece yararlı olan verilerdeki sıralı bilgilerin yorumlanmasına yardımcı olur. RNN'leride ileriki bölümlerde detaylı göreceğiz.