Yapay Zeka
Eğitim Kitapçığı

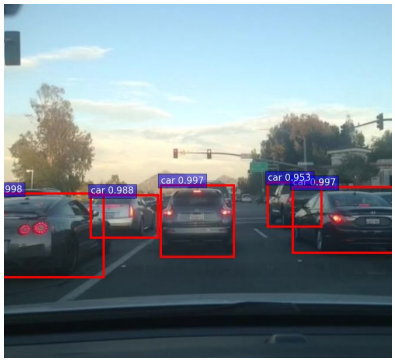
(18 Ağustos 2019 tarihinde Güncellenmiştir)

Yapay Zeka Nedir?

Yapay zeka, normal şartlarda insan zekası gerektiren görevleri yerine getirebilen bilgisayar sistemlerinin geliştirilmesini konu alan bilgisayar bilimleri dalıdır. Yapay zekanın gelişimi bilgisayar biliminin de babası kabul edilen Alan Turing'in "Makineler Düşünebilir Mi?" makalesiyle başlamıştır. Yapay zeka ismi ise ünlü Dartmouth konferansında yine yapay zeka alanının kurucu babalarından kabul edilen John McCarthy tarafından 1955 yılında verilmiştir. O zamandan beri üzerinde çalışılan bir alan olsa da özellikle bilgisayarların işlem gücünün artmasıyla işe yarar hale gelmiş, 1997'de satranç dünya şampiyonunu yenen Deep Blue ile de dünya çapında popülerleşmeye başlamıştır. Yapay Zeka Araştırmacıları bugüne kadar birçok başarıya imza atmıştır. Yapay zeka birçok alanda insan zekasını geçmiş ve ilerlemeye devam etmektedir. Yapay zeka birçok alanda kullanılabilir. SAC Race için ise özellikle görüntü verisi üzerindeki kullanımları önemlidir.


YZ'nin yendiği ilk şampiyon.

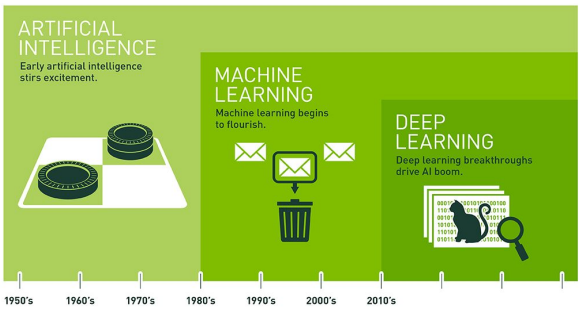

YZ, en zor oyun GO'da kazanıyor.

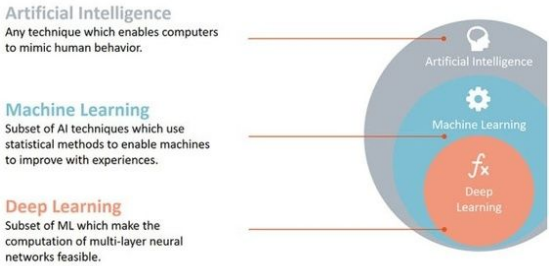


YZ, araçları tespit ediyor.

Makine Öğrenmesi ve Derin Öğrenme

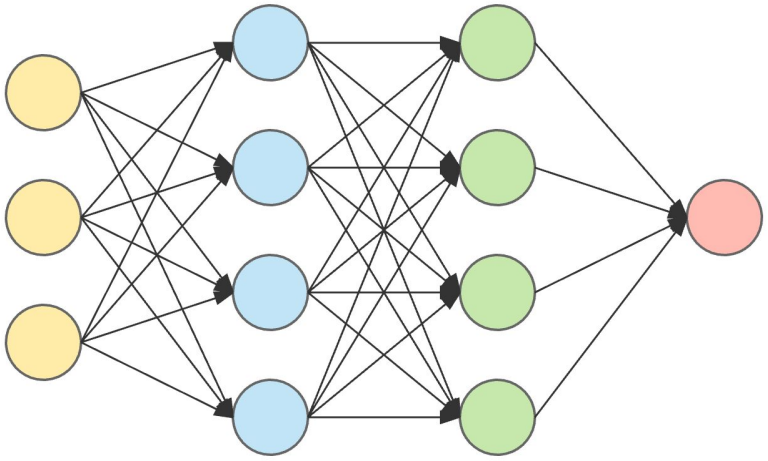
Makine öğrenmesi (Machine Learning), yapay zekanın bir alt dalıdır. Makine öğrenmesi araştırmaları problemi çözecek algoritmalar üretmek yerine bilgisayarların veriyi kullanarak problemi nasıl çözeceğini anlamasını sağlayan sistemler üretir. Bu şekilde normalde kullandığımız algoritmalar ile çözmemiz çok uzun sürecek veya imkansız olacak problemlere bir çözüm oluşturur. Örneğin araç tespiti yapmak istediğimizi düşünelim. Bir algoritma aracın fotoğrafı ile kameradan gelen veriyi piksel piksel karşılaştırabilir ancak bu algoritma neredeyse kullanılamaz çünkü kameradan gelen resim aracı farklı açılardan, uzaklıklardan görebilir veya farklı marka model bir araç görebilir. Bu durumda algoritma işe yaramaz. Yukarıdaki araç tespiti fotoğrafına bakarsanız tek bir yapay zeka ile yoldaki farklı birçok araç tespit edilebilmektedir. Makine öğrenmesinin genel prensipi olarak verideki karakteristik özellikleri, tekrarlamaları bulmaya çalışır. Örneğin bir araçtaki karakteristik özellikler farlar, tekerlekler vb. olabilir. Ancak birçok makine öğrenmesi algoritması yalnızca basit veri setlerinde doğru sonuçlar verebilir. Sadece az çeşitli sayısal verinin olduğu bir veri setinde (örneğin basketbolcunun koşma hızı, boyu ve kaç yıldır basketbol oynadığından takıma alınmalı/alınmamalı diye sınıflandırma yapılan bir kullanım) çok iyi çalışabiliyor iken araç tespiti için yeterince öğrenecek kapasiteye sahip olmayabilir. Bu durum derin öğrenmenin ortaya çıkmasına neden olmuştur.





Derin Öğrenme ve Yapay Sinir Ağları

Derin öğrenme oluşturulurken ilham kaynağı doğal zekanın da gerçekleştiği yer olan beyin olmuştur. Beyin, nöron ismi verilen sinir hücrelerinden oluşmuş bir organdır. Nöronlar kendinden önceki birçok nörondan sinyal alır ve kendinden sonraki nöronlara sinyal verirler. Bu nöronlardan binlercesi, belki milyonlarcası birbirine bağlanarak doğal zekayı oluştururlar. Yapay sinir ağları ise bu doğal yapıya benzer bir sistem üretmek amacıyla ortaya çıkmıştır. Yapay sinir ağları, beyin gibi binlerce nörondan oluşur. Bu nöronlar beyindekiler gibi kimyasal değil matematiksel işlemler yaparak sonraki nöronlara gerekli veriyi gönderirler.



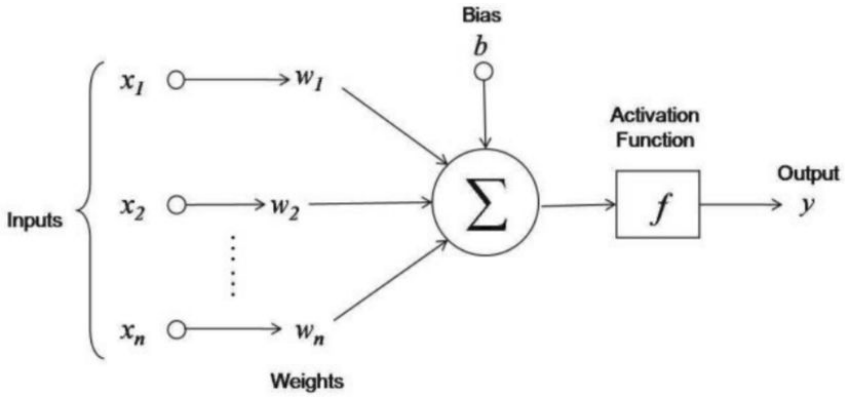
input layer hidden layer 1 hidden layer 2 output layer

Daire => Nöron Ok => Veri aktarımı

Yapay sinir ağları yukarıdaki şekil gibi gösterilebilir. Bir sinir ağı veri girişi yapan bir katman, bir çıkış katmanı ve istediğiniz sayıda gizli (hidden) katmandan oluşur. Bir katmandaki her nöron (daire şekilleri ile gösterilmiş) bir önceki ve bir sonraki katmanlardaki nöronlara (genelde tümüne ancak değişebilir) bağlıdır.

Peki bir yapay nöron nasıl çalışır?

Bir nöron söylediğimiz gibi matematik işlemleri yapan ve öğrenebilen bir fonksiyondur. Öğrenme kısmına sonra geçeceğiz ancak şimdilik eğitimi tamamlanmış bir sinir ağındaki nöronların nasıl çalıştığına bakalım.



Yukarıdaki görselde bir nöronun çalışma şekli görselleştirilmiştir. Bu nörona önceki nöronlardan gelen sayılar x olarak ifade ediliyor. Önceki katmandan gelen bu verilerin hepsi bir ağırlık ile çarpılıyor. Daha sonra çıkan sonuçların hepsi toplanıyor. Bias ismini verdiğimiz o nöron için sürekli aynı olan bir sayı ekleniyor. Daha sonra aktivasyon fonksiyonu dediğimiz çıkan sayıyı diğer nöronlardan çıkanlar ile benzer aralıkta tutmak için kullanılan bir fonksiyona giriyor. Aktivasyon fonksiyonu sinir ağının öğrenebilmesi için önemli bir parça. Bir nöronu çalıştıran formül ise şu şekilde:

$$f\left(b+\sum_{i=1}^{n} x_i w_i\right)$$

Bir örnek yapalım. Bu nöron bir basketbolcunun boy (m), ortalama hız (m/s), yaş (yıl) değerlerini alarak çalışıyor olsun.

Veri	Değer	Ağırlık	Sonuç
Boy (m)	1.91	1.2	2.292
Hız (m/s)	2.4	-0.3	-0.72
Yaş	24	0.1	2.4

Sonuçların toplamı 3.972 ediyor. Sonuca sigmoid (ileride açıklanacak) aktivasyon fonksiyonu uygulanırsa sonuç ~0.981 oluyor. Bu ne anlama geliyor? Bilmiyoruz. Sinir ağlarında bir nöronun ne işe yaradığını bilmek çoğunlukla imkansız olsa da birleşince anlamlı bir bütün elde ediliyor.

Aktivasyon Fonksiyonları

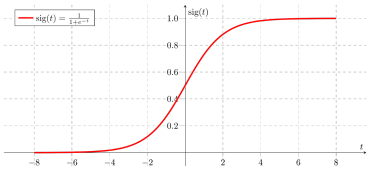
Aktivasyon fonksiyonu olmadan bir nöron lineer (doğrusal) bir fonksiyondur. Lineer fonksiyonların bizim için kötü yanı ise kompleksite bakımından sınırlı olmaları ve bu nedenle kompleks fonksiyon haritalarını veriden öğrenmelerinin zor olmasıdır. Aktivasyon fonksiyonları olmayan bir sinir ağının gücü sınırlıdır ve çoğunlukla iyi sonuç vermez. İstediğimiz ise lineer fonksiyondan daha karmaşık şekilde öğrenebilmesidir. Bu sayede resim, ses vb. kompleks veri çeşitlerinden öğrenebilir.

Genellikle kullanılan birkaç aktivasyon fonksiyonu bulunmaktadır. Bunlardan hangisinin daha iyi sonuç vereceği sinir ağı ve veri ile ilgili birçok duruma bağlıdır. Bu fonksiyonlardan iyi bilinen üç tanesi Sigmoid, Tanh ve ReLU olarak bilinmektedir.

Sigmoid fonksiyonu:

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

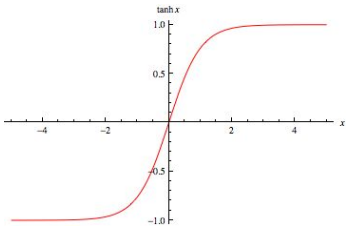
Sigmoid fonksiyonunun grafiği:



Tanh fonksiyonu:

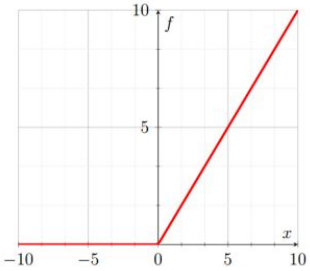
$$\tanh x = \frac{\sinh x}{\cosh x} = \frac{e^x - e^{-x}}{e^x + e^{-x}},$$

Tanh fonksiyonunun grafiği:



ReLU fonksiyonu:

`ReLU(x) = max(0,x)`



Daha fazlası için:
towardsdatascience.com/activation-functions-and-its-types-which-is-better-a9a5310cc8f

Yapay Sinir Ağı Eğitimi (Training)

Yeni bir yapay sinir ağı ürettiğinizde ağırlıkların hepsi rastgele olarak belirlenir. Sinir ağının öğrenmesi olayı da bu ağırlıkların daha doğru sonuçlar verecek şekilde düzenlenmesidir. Farklı öğrenme yöntemleri olan derin öğrenme modelleri farklı şekillerde eğitim yapabilmektedirler. Bu öğrenme yöntemlerinden en çok bilinenleri gözetimli (supervised) öğrenme, gözetimsiz (unsupervised) öğrenme ve pekiştirmeli (reinforcement) öğrenmedir. Biz bu bölümde gözetimli öğrenme üzerine çalışacağız.

Gözetimli öğrenme için elimizde etiketlenmiş bir veri seti olması gerekmektedir. Etiketlenmiş olması, bilinen bir girdiler dizisi için bilinen bir çıktı olması demektir. Yani daha önceden olmuş benzer durumlar ve sonuçları verilerek, yeni durum verildiğinde çıktının ne olacağı tahmin edilmeye çalışılmaktadır. Örnek bir veri seti olarak bir öğrencinin türkçe, matematik, fen bilgisi sınav notlarından lise geçiş sınavı puanını tahmin etmeye çalışalım. Veri setimiz aşağıdaki gibi olsun. Bu veri seti bir önceki yıl sınavlara giren öğrencilerin sonuçları.

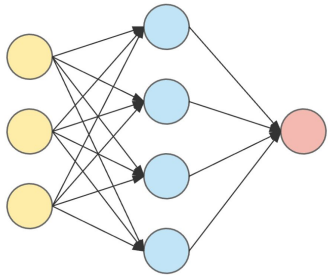
Ders Notları	**Sınav Puanı**
85, 95, 95	440
90, 75, 80	410
95, 95, 100	490
75, 55, 80	370

* Ders notları 100, Sınav puanı 500 üzerinden değerlendiriliyor.

Biz de yapay sinir ağımızdan yeni öğrencilerin sonuçlarını tahmin etmeyi öğrenmesini isteyeceğiz. Ama öncelikle bu ders notları verildiğinde doğru sınav puanını bulması gerekiyor. Aslında yapay zeka eğitimi bu ağırlıkları optimizasyon yöntemleri kullanarak doğru puanı elde edecek şekilde değiştirmek anlamına geliyor.

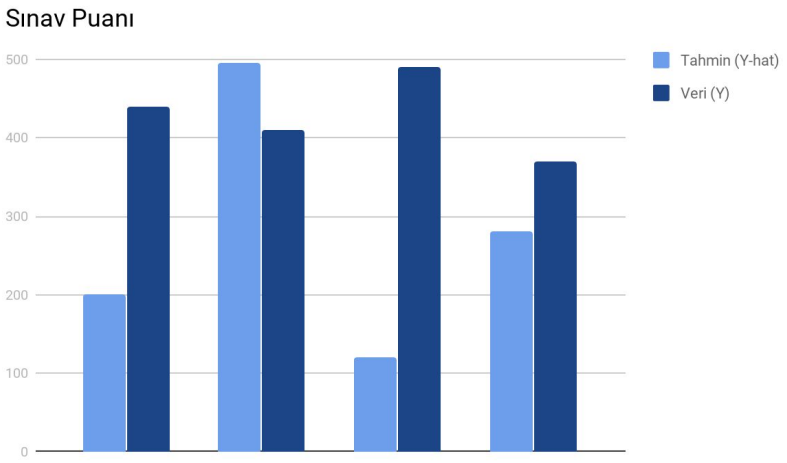
Öncelikle kullanacağımız sinir ağını oluşturalım. Bu sinir ağının girdi katmanında ders notlarını alması gerekiyor. Bu nedenle ilk katmanda 3 node (nöron) olacak. Çıktı olarak da tek bir sayı vermesi gerekiyor. Bu örnekte çözdüğümüz türdeki problemlere regression problemi diyoruz. Bunun nedeni sonucun etiketlerden farklı olabilecek bir sayı olacak olması. Eğer etiketlerden birini seçmesini isteseydik bu bir sınıflandırma (classification) problemi olurdu. Örneğin öğrencinin hangi liseye gideceğini tahmin etmek gibi.

Girdi ve çıktı arasındaki gizli katmanlarla ise istediğimiz gibi oynayabiliriz ancak şimdilik basit bir sinir ağı olması için tek bir katmanda 4 node olacak. Görselleştirelim:

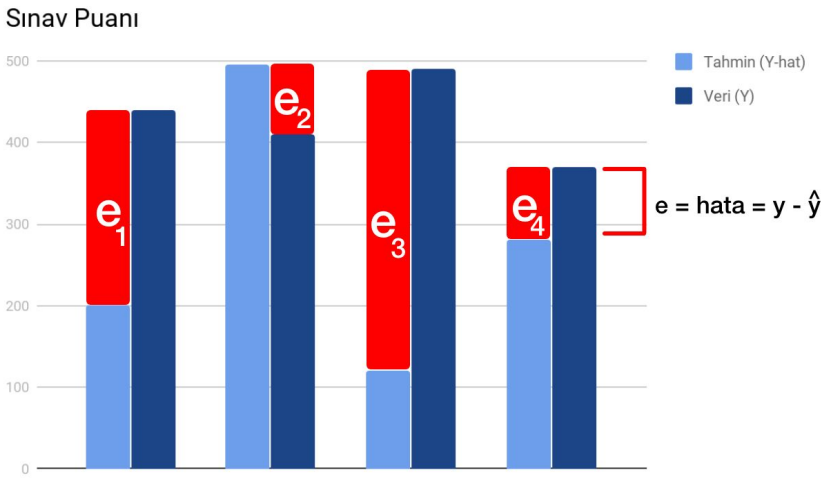


Peki bu sinir ağında kaç adet ağırlık var? Gizli katmandaki her nörona üç veri geliyor. Bunların her biri bir ağırlık ile çarpılacak. Bu katmanda 4 nöron olduğuna göre bu katmanda 4 x 3 = 12 ağırlık var. Çıktı katmanında ise tek bir nöron var ve 4 veri geliyor. Bu katmanda da 1 x 4 = 4 ağırlık var. Toplam olarak da 16 ağırlık bulunuyor.

Bu ağırlıklar eğitim sırasında değiştirilecek ancak şimdilik rastgele olarak belirleniyor. Bu rastgele belirlenmiş ağırlıkları değiştirmek istiyoruz ancak neye göre değiştireceğiz? Buradaki amacımız sinir ağının yukarıdaki tablodan aldığı veriye göre yaptığı tahmin ile tablodaki çıktının birbirine yakınlaşması. Bu nedenle ilk önce sinir ağının tahmini ($\hat{y}$ ile gösteriyoruz.) ile veri setindeki etiketin (y ile gösteriyoruz.) arasındaki hatayı bulmamız gerekiyor. Bunun için kullanılan fonksiyona cost function (maliyet fonksiyonu) deniyor ve bu fonksiyonun amacı modelimizin ne kadar hatalı olduğunu hesaplamak. Bunun için ilk önce modele şuanki rastgele ağırlıklarla tahmin yaptırıyoruz. Sonuçlar tabiki yanlış çıkıyor.



Nu çıkan sonuçlardaki hataları bir şekilde birleştirip tek bir maliyet/hata ortaya çıkaracak fonksiyon cost function oluyor. İlk önce hataları hesaplıyoruz. Bir tahmindeki hata direkt olarak asıl veri - tahmin olarak hesaplanabilir.



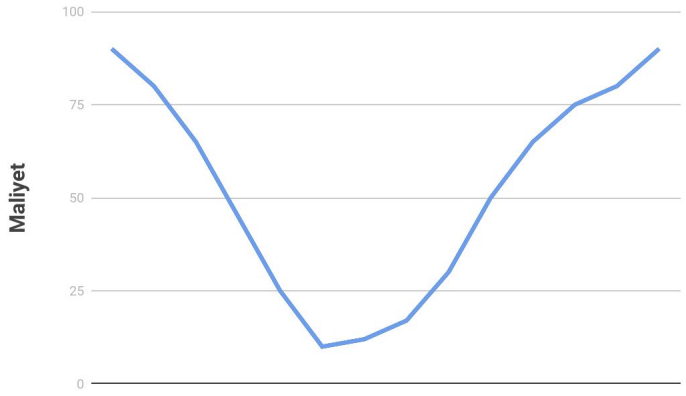
Bu hata değerlerini kullanarak da bir cost fonksiyonu oluşturabiliriz. Kullanabileceğiniz birçok geliştirilmiş cost fonksiyonu bulunmakta ancak biz en basitlerinden birini kullanacağız.

$$cost = j = \sum \frac{1}{2}e^2$$

$$e = (y - \hat{y})$$

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

Şimdi bir maliyetimiz (j) olduğuna göre görevimiz bu maliyeti azaltmak. Bir sinir ağı eğitmek aslında maliyeti azaltmak ile aynı anlama geliyor. Maliyeti oluşturan iki ana şey var. Veri ve ağırlıklar. Veriyi kontrol edemediğimize göre ağırlıkları maliyeti en az hale getirecek şekilde değiştirmek istiyoruz. Bir ağırlığı her oynattığımızda maliyet artıyor ya da azalıyor. Örnek olarak bir ağırlığı değiştirdiğimizde maliyete ne olduğuna bakalım ve grafiğini çizelim.



Ağırlık 1

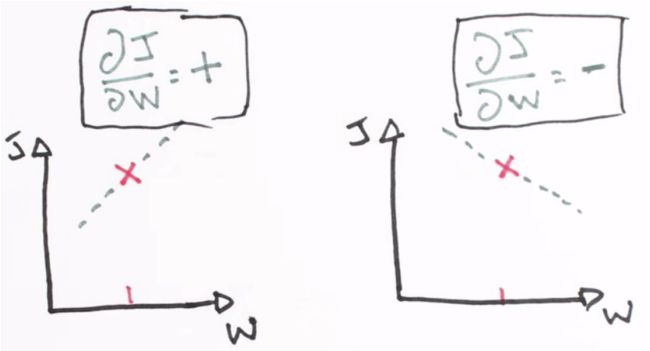
Maliyeti azaltmak için ağırlık 1'i yukarıdaki grafikteki en dip noktaya getirmek gerekiyor. Bu noktaya global minimum diyoruz. Ancak bunu nasıl yapacağız? Her işlemde ancak verdiğimiz ağırlığa karşı gelen maliyet değerini bulabiliyoruz. Örneğin ağırlık 3 ise maliyet 4.5'dur. 2 ise maliyet 1.2'dir şeklinde. Bir çok farklı değeri deneyip en küçüğünü alabilir miyiz? Ne yazık ki bilgisayarımızın işlem gücü bu işe yetmeyecektir. Bu ağırlığa 100 farklı değer verip test ettiğimizi düşünelim. Bilgisayarlarımız güçlü makinalar sonuçta. Çok hızlı sonuç alabiliriz. Ancak burada "Curse of dimensionality" (Boyutluluk laneti) ismi verilmiş bir problem ile karşılaşıyoruz. Bir ağırlık ile çok kolay olabilir ancak iki ağırlığı düzenlemek istiyorsak? Bu sefer 100^2 yani 10000 işlem yapmamız gerekiyor. Biz en başta 16 ağırlık ile çalışacağımızı söylemiştik. Bu da 100^{16} işlem gerektiriyor. (10^{32}, 100 Nonilyon deniyormuş. İngilizce nonillion) Kullandığımız sinir ağı bir derin sinir ağı bile sayılmayacak kadar küçük. Şimdi katmanlarındaki nöron sayıları şöyle olan bir sinir ağı düşünelim: [256, 256, 128, 64, 10]. Bu nispeten küçük bir derin sinir ağı. Bu sinir ağı için bu yöntemle kaç hesaplama yapmak gerektiğini hesaplamaya bile gerek yok.

O zaman ne yapacağız? Durduğumuz yerden hangi tarafın aşağı doğru gittiğini bilebilir miyiz? O zaman işimiz çok daha kolay olurdu. Evet bilebiliriz ve bu konuda bize yardımcı olacak çok sevdiğimiz bir konsept de var: Türev.

Aslında aradığımız şey, maliyetin (j) ağırlığa (w) göre değişim oranı, yani türevi. Biz her seferinde bir ağırlık için kullandığımızdan dolayı kısmi türev (Partial derivative).

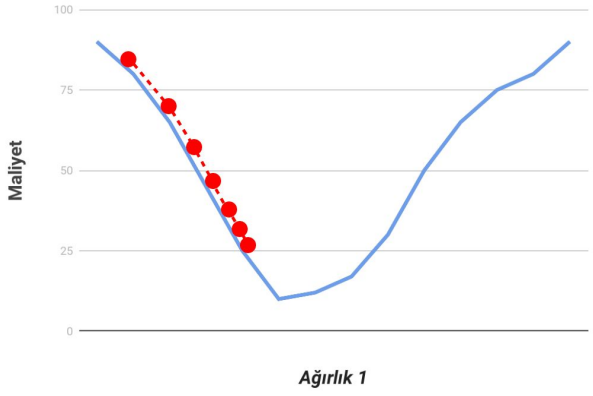
$$\frac{\partial j}{\partial w}$$

Bu konsepti kullanarak hangi yöne doğru ilerlemek gerektiğini bulabiliriz. Eğer sonuç pozitif ise sola doğru aşağı gidiyor, negatif ise sağa doğru aşağı gidiyordur.

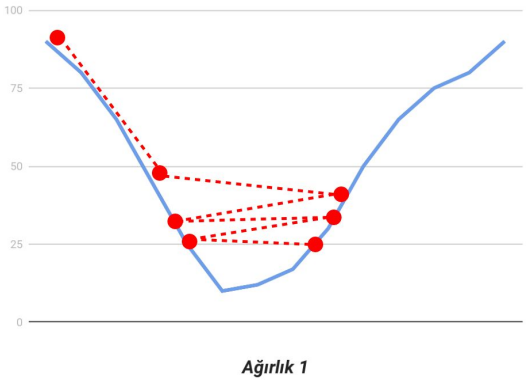


Bu sayede çok daha hızlı bir şekilde öğrenme gerçekleştirilebilmektedir. Bu sayede hangi yönde ağırlığı değiştireceğimizi bildiğimize göre o yönde biraz ilerleyip hangi yöne gitmek gerektiğine tekrar bakabiliriz. Bu sefer bu yönde biraz ilerleyip (genellikle bir öncekinden daha az) tekrar bakarız. Bu işleme “gradient descent” ismi verilmektedir. Tüm katmanlara ve ağırlıklara bu işlemi uygulamamızı sağlayan backpropagation isimli bir algoritma kullanılmaktadır. Bu algoritma gradient descent’i her katmandaki tüm nöronlara önceki katmandaki maliyete göre uygular.

Her ilerleyişte ne kadar ilerleyeceğimizi “learning rate” ismini verdiğimiz bir değişkeni değiştirerek kontrol ederiz. Learning rate çok az ise öğrenme çok uzun sürer ve en alta ulaşamayız. Learning rate çok fazla ise en altı kaçırıp diğer tarafa geçeriz ve yine öğrenme çok uzun sürer, en alta ulaşamayız. Learning rate seçmek için ise belirli bir yöntem yoktur. Şuan için deneme yanılma işinize yarayacaktır.



Çok küçük learning rate



Çok büyük learning rate

Derin Öğrenme Kütüphaneleri

Derin öğrenmenin matematik alt yapısını bilmek geliştirebilmek için çok önemli olsa da her projede en baştan fonksiyonları ve sinir ağı matematiğini yazmamız gerekmiyor. Tekerleği baştan icat etmeye gerek yok. Bu nedenle derin öğrenme için geliştirilmiş, yapay sinir ağlarının çok kullandığı işlemler için optimize edilmiş kütüphaneler bulunmaktadır. Alanda çok fazla büyük kurumun bulunması da onlarca farklı kütüphane oluşmasına neden oldu. Bu kütüphanelerin neredeyse hepsi açık kaynaklı olarak sunuluyor ve GitHub'da büyük topluluklara sahipler. Bu kütüphanelerden bazıları Google'un Tensorflow, Microsoft'un Cognitive Toolkit (CNTK), Facebook'un PyTorch, Berkeley'nin Caffe, Université de Montréal'in Theano framework/kütüphaneleridir. Bunlar ayrıca GitHub toplulukları tarafından çok iyi desteklenen platformlardır. Örneğin Tensorflow GitHub'da 130 bin kişi tarafından takip edilmektedir. Biz de kullanımı kolay olması, herkes tarafından her yazılım ortamında kullanılabilmesi, dökümantasyonun ve kullanıldığı örneklerin internetteki fazlalığı nedeniyle tensorflow ile çalışmaya karar verdik.

Tensorflow Kurulumu

Tensorflow'un birçok yazılım dili için API'ı vardır. Bunlardan C++ ve python resmi olarak destekledikleri dillerdir. Biz de derin öğrenme için çok uygun olduğu ve kullanımı kolay olduğu gerekçesiyle python dilini kullanacağız. Python için tensorflow'u kurmak için terminalde şu komutlar kullanılabilir.

CPU versiyonunu kurmak için:

```
pip install tensorflow
```

GPU versiyonu için (Nvidia GPU ile kullanım için):

```
pip install tensorflow-gpu
```

Yeni çıkacak Tensorflow 2.0'ın beta versiyonu için:

```
pip install tensorflow==2.0.0-beta1
```

veya

```
pip install tensorflow-gpu==2.0.0-beta1
```

Daha fazla bilgi için www.tensorflow.org/install

Tensorflow Python'da Kullanmaya Başlamak

Tensorflow'u python'da kullanmaya başlamak için gereken tensorflow'u import etmektir. Bu işlem şu şekilde yapılmaktadır:

```
import tensorflow as tf
```

Burada tensorflow'un ismini tf olarak değiştirmenin nedeni yalnızca daha kısa olmasıdır. Daha sonra birçok kez yazmamız gerekeceği için iki harf ile kullanmak daha rahattır.

Tensorflow ve Keras

Keras, diğer derin öğrenme kütüphanelerinin üzerine kurulmuş ve yüksek seviyeli işlemleri çok daha kolay yapmanızı sağlayan bir kütüphanedir. Keras; Tensroflow, CNTK, Theano gibi birçok kütüphaneyle birlikte kullanılabilmektedir. Keras'ın önemini fark eden tensorflow ekibi de kendi kütüphanesinin high-level API'ını Keras ile değiştirmiş ve keras'ın neredeyse tamamını tensorflow'un içinde kendilerine göre daha iyi optimize edilmiş biçimde kullanıma sunmuşlardır. Tensorflow'u kurduktan sonra keras'ın tüm fonksiyonlarına tf.keras şeklinde erişebilirsiniz. Keras, yapay sinir ağı (model olarak isimlendirilmektedir) oluşturma işlemini çok basit hale getirmiş ve derin öğrenmenin herkes tarafından kullanılabilmesini sağlamıştır. Bugün keras sayesinde 20 satır kod ile bir derin öğrenme modeli oluşturulabilmektedir. Tensorflow, keras haricinde bir de low-level api'a sahiptir ancak bu api genel olarak derin öğrenmenin en alt seviye matrix çarpımlarını bile teker teker yazma ihtiyacı olan projeler için kullanılmak üzere yazılmıştır. Bugün tüm derin öğrenme işlerinin %95'inden fazlası keras ile yapılabilmektedir.

Tensorflow ve Keras Kullanarak İlk Modelimizi Oluşturalım

```
import tensorflow as tf
import numpy

# Bu örnek için kullanacağımız veriyi yükler.
# Fashion MNIST seti 28x28 piksel kıyafet resimlerinden oluşur.
# Bu veri setinde 10 sınıf bulunur.
fashion_mnist = tf.keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) =
fashion_mnist.load_data()

# Normalizasyon: Daha kolay eğitim için veriyi 0 ile 1 arasına indirger.
train_images =  train_images / 255.0
test_images = test_images / 255.0
```

```
# Keras Sequential API'i ile yeni model oluşturur.
# Bu API ile modeli add fonksiyonu kullanarak katman katman oluşturabiliriz.
model = tf.keras.Sequential()

# Modele yeni katmanlar (layer) eklemek bu kadar kolay
# Flatten (28x28) şeklindeki veriyi tek bir diziye dönüştürür.
model.add(tf.keras.layers.Flatten())
# Dense layerları bildiğimiz nöronlardan oluşan katmanlardır.
# 64, 32 ve 10 kaç nöron olacağını belirtir.
# Burada bir sınıflandırma problemi çözüldüğü için çıktı katmanında 10 sınıf
için 10 nöron bulunur.
# activation, yukarıda da bahsettiğimiz aktivasyon fonksiyonlarıdır.
model.add(tf.keras.layers.Dense(64, activation="relu"))
model.add(tf.keras.layers.Dense(32, activation="relu"))
# Softmax tüm veriyi 0 ile 1 arasında olasılık değerlerine dönüştürür.
# En yüksek olasılık olan sinir ağının tahminidir.
model.add(tf.keras.layers.Dense(10, activation="softmax"))

# Model eğitimi için gereken ayarlamaları yapan fonksiyon.
# Loss fonksiyonu, öğrendiğimiz cost fonksiyonudur.
# Optimizer, gradient descent gibi ağırlıkların en iyi olduğu noktaları bulmaya
çalışan sistem.
model.compile(optimizer="adam", loss="sparse_categorical_crossentropy",
metrics=["accuracy"])
# Modeli resimleri ve etiketlerini kullanarak eğiten fonksiyon.
# Epoch, verinin üzerinden kaç defa tekrar geçileceğidir.
model.fit(train_images, train_labels, epochs=10)
# Sonucu test etmek için kullanılan fonksiyon.
test_loss, accuracy = model.evaluate(test_images, test_labels)
print("Evaluation Accuracy:", accuracy)
```

Gördüğünüz gibi çok kısa bir kod ile resimleri sınıflandıran bir derin öğrenme modeli yaptık. Bu modelde yalnızca dense katmanlarını kullandık. Dense katmanları kitapçığın önceki kısımlarında gördüğümüz nöronlardan oluşan sinir ağını oluşturmak için kullanılmaktadır. Ancak bir modeli oluşturabilen tek katman türü bu değildir. Yukarıdaki örnekte Dense haricinde flatten() kullandık. Flatten iki veya daha fazla boyutlu bir diziyi (örneğin bir resim) tek boyutlu bir diziye dönüştürür. Bunlar dışında da farklı veri türlerini işlemede çok işe yarayan farklı katman türleri üretilmiştir. Bundan sonraki bölümlerde bir derin öğrenme projesinde veriyi içeri aktarmaktan, eğitilmiş modeli kullanmak için dışarı aktarmaya ve sonrasında tahmin için kullanmaya kadar derin öğrenme iş sürecinin tamamı sırayla işlenecektir.

Eğitim Verisini İçeri Aktarma

Bildiğimiz bir şey de eğitim için çok fazla veriye ihtiyacımız olduğudur. Peki hazırladığımız bir veri setini nasıl kullanırız? Öncelikle veri setleri birçok farklı şekilde hazırlanabilir. Örneğin bir sınıflandırma probleminde resimler kendi sınıflarının adını taşıyan klasörlere koyulabilir. Başka bir yöntem bir klasörde tüm resimleri saklamak ve resimlerin etiketlerini ikinci bir klasörde resimle aynı isimde bir txt dosyasında saklamak olabilir. Örneğin 00216.png için 00216.txt'de "cat" yazması gibi. Başka bir yöntem ise resimlerin bir klasörde saklanması ile etiketlerin tek bir txt veya csv dosyasında saklanmasıdır. Hangisini kullanacağınız size kalmış olsa daha sonra kullandığınız formattan keras'ın sizden istediği formata dönüştürmeniz gerekmektedir.

Peki keras nasıl bir veri istiyor? Üstteki örnekten de gördüğünüz gibi keras'ta kullandığımız veri train_images ve train_labels şeklinde iki diziye ayrılmış durumdadır. Bu dizilerden train_images'ın ilk element'indeki resmin etiketi train_labels'ın ilk elementi olacak şekilde sıralı bir saklama işlemi yapılır.

Hem sınıflandırma hem de regresyon verisinin rahatlıkla saklanabildiği ve en az dosya kalabalığı yapan yöntem olduğu için biz genellikle tüm etiketleri bir csv dosyasında saklamayı tercih ediyoruz. Bizim kullandığımız sistemden veriyi keras'a aktarmak da çok basit oluyor. CSV dosyasının her satırını şu şekillerde organize ediyoruz:

Sayısal veri	veri-1, veri-2, … , veri-n, etiket
Resim verisi	resim adresi, etiket

Farklı veri setleri için de csv dosyası çok basit şekillerde düzenlenebilmektedir. Peki csv'den aldığımız veriyi nasıl düzenleyeceğiz. Sayısal veriler için gayet kolay. Tüm veri noktalarını tek bir diziye `data = [[veri-1, … , veri-n], [veri-1, … , veri-n]]` şeklinde sırayla yerleştirebiliriz. Etiketleri de aynı şekilde `labels = [etiket, etiket]` şeklinde koyabiliriz. Resim verisi için ise resim adresini csv'den alıp, opencv ile resmi almak, bu resmi data dizisine koymak, sonrasında csv'den alınan etiketi labels dizisine koymak gerekiyor.

data.csv dosyası bir kişinin resminin adresi ve boyunu yazdığımız bir veri seti için olsun.

```
resim,boy
1.png,168
2.png,179
3.png,191
```

Önemli not: virgülden sonra boşluk bırakmayın.

Bu veriyi kullanmak için yapmamız gerekenler şu şekilde:

```
# Pandas kütüphanesi CSV okumak için kullanılmaktadır.
import pandas as pd
# Görüntü işleme kütüphanelerimizi import ediyoruz.
import numpy as np
import cv2

# Pandas'tan verilerimizi tuttuğumuz csv'yi okuyup işlemesini istiyoruz.
data = pd.read_csv("data.csv")

# Resim adreslerini ve etiketlerini pandastan alıyoruz.
adresler = data["resim"]
etiketler_csv = data["boy"]

# Pandas formatından etiketleri kullanacağımız formata çeviriyoruz.
etiketler = []
for etiket in etiketler_csv:
    etiketler.append(etiket)

# Resimleri opencv kullanarak adreslerinden alıp bir diziye atıyoruz.
resimler = []
for adres in adresler:
    resim = cv2.imread(adres)
    # Resimler farklı boyutlarda ise hepsini aynı yapmanız gerekiyor.
    resim = cv2.resize(resim, (500, 500))
    resimler.append(resim)

# Verileri derin öğrenmede çok kullanılan numpy array formatına
dönüştürüyoruz.
resimler = np.array(resimler)
etiketler = np.array(etiketler)

# Artık hem resimler, hem de etiketlerimiz hazır.
# Kontrol edelim:
print("Resimler boyutu:", resimler.shape)
print("Etiket boyutu:", etiketler.shape)
```

Bu sayede verimizi Keras'ın istediği formata dönüştürerek sisteme yüklemiş olduk. Resmi test etmek istiyorsanız veri setinizin bir kısmını test seti olarak ayırmanızı tavsiye ederim. Öğrettiğiniz resim ile test yapmak gerçekte olduğundan fazla iyi sonuç verebilir.

Model Oluşturmak

Model oluşturmak için Keras'ın birçok API'ı bulunuyor. Bunlar Sequential, Functional ve Model Subclassing API'ları. Bunlardan bizim de yukarıda kullandığımız Sequential, modele yeni katmanları add fonksiyonu ile ekleyebilmemizi sağlıyor. Kullanımı en basit api olmasının yanı sıra Tensorflow ekibi tüm makine öğrenmesi problemlerinin %95'i için uygun olduğunu öne sürüyor. Functional API, tüm katmanların arka arkaya gelmediği daha soyut modeller için kullanılıyor. Örneğin bir katmanın veriyi iki ayrı katmana gönderip sonra bunların sonuçlarının bir şekilde birleştirilmesi gibi. Model Subclassing ise tamamen kişiselleştirilmiş, veri aktarımına kadar çoğu şeyi kendiniz belirleyeceğiniz katmanlar üretmek için tasarlanmış. Diğerlerine de bakmanızı tavsiye ederim ancak biz şimdilik Sequential üzerinden ilerleyeceğiz.

Sequential bir modele başlamak için kullanmanız gereken satır şu şekilde:

```
model = tf.keras.Sequential()
```

Bundan sonra model.add() şeklinde katmanları dizebilirsiniz.

Dense Katmanları

Dense katmanı, kitapçığın ilk bölümünde gördüğümüz şekilde çalışan katmanları içeriyor. Bir dense katmanını aşağıdaki şekilde modelimize ekliyoruz.

```
model.add(tf.keras.layers.Dense(64, activation="relu"))
```

Burada "64" sayısının olduğu alana bu katmanda kaç adet nöron istediğimizi yazıyoruz. Activation kısmına da hangi aktivasyon fonksiyonunu kullanmak istiyorsak adını yazıyoruz. Kullanabileceğiniz fonksiyonlara şu linkten bakabilirsiniz:
www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/activations

Dropout

Dropout, "overfitting" dediğimiz modelin veriyi ezberleyerek o veride çok iyi çalışmasına rağmen yeni veride iyi çalışmamasına neden olan bir durumu engellemek amacıyla üretilmiş bir yöntemdir. Dropout modelin nöronların belirli bir bölümünü (alttaki örnekte %50'sini) eğitim sırasında rastgele olarak kullanamamasını sağlıyor. Bu nedenle sürekli olarak aynı yolları kullanacak şekilde ezberlemesine engel oluyor. Örneğin kedi şeklini öğretmeye çalıştığımızı düşünelim. Aldığı verilerden kedinin kuyruğu olduğunu öğreniyor ve her kuyruk gördüğünde kedi diyecek şekilde öğreniyor. Bu nedenle de kedinin kuyruğunun arkasında kaldığı veya başka bir objeden dolayı görülmediği durumlarda başarısız oluyor. Dropout ise eğitim sırasında kuyruk ile ilgili nöronlardan bir kısmını rastgele iptal ediyor. Kuyrukta sıkıntı olması modeli kedinin kulağı, patileri gibi diğer özelliklerine (feature) yönelmesini de sağlıyor. Daha genel modeller ortaya çıkıyor. Dropout şu şekilde iki katman arasına oradaki bağlantılarda çalışması için ekleniyor:

```
model.add(tf.keras.layers.Dropout(0.5))
```

Düzleştirme Katmanı (Flatten)

Dense katmanları bir boyutlu veri üzerinde çalışmaktadır. Ancak bir problem resimlerin iki boyutlu (siyah-beyaz ise) veya çoğu zaman üç boyutlu (renkli ise) olmasıdır. Peki bu resimleri nasıl dense katmanına uygun hale getireceğiz? Flatten katmanıyla. Flatten çok basit şekilde birden fazla boyutlu dizileri tek boyutlu dizi haline getiriyor. Örneğin bu bir iki boyutlu dizi:

```
[[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]]
```

Ve flatten()'ın yaptığı tek şey buradaki sayıları sıraya dizmek ve bu hale getirmek:

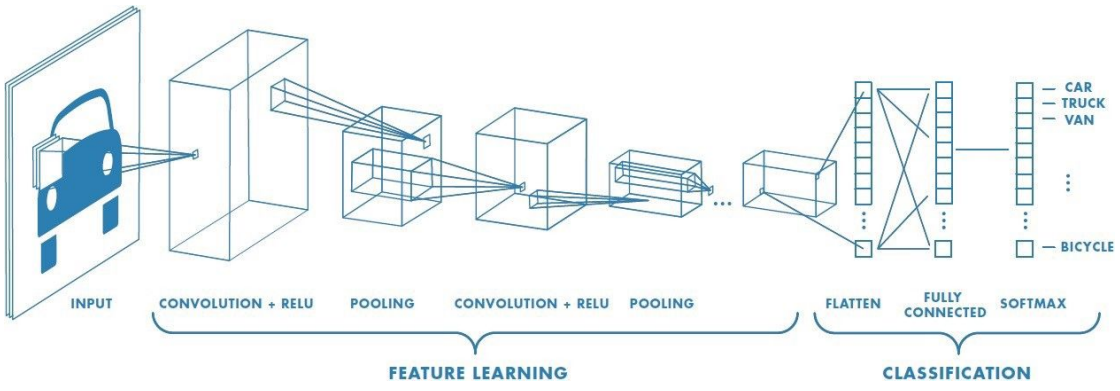
```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Flatten'ı şu şekilde koda ekliyoruz:

```
model.add(tf.keras.layers.Flatten())
```

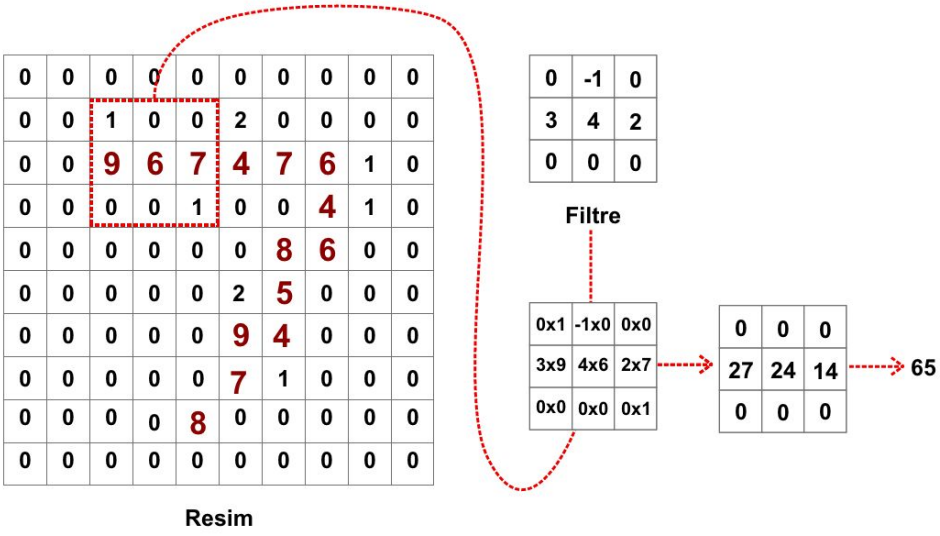
Evrişimli Sinir Ağı Katmanları

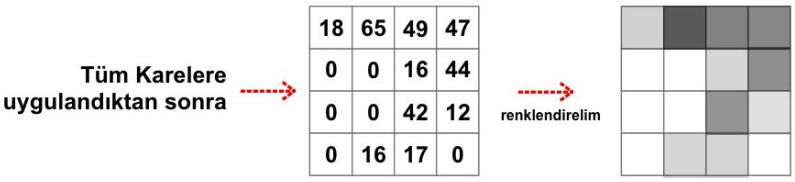
Yapay Zeka ile Görüntü İşleme alanında çığır açan yeniliklerden biri de evrişimli sinir ağları (Convolutional Neural Networks) olmuştur. Dense katmanları için veriyi düzleştirdiğimizi söylemiştik. Ne yazık ki verinin düzleştirilmesi, birbirlerine göre anlamlı lokasyonlarda bulunması sayesinde veriye anlam katan birçok özelliğin kaybedilmesine neden olmaktadır. Bundan kurtulmak için de evrişimli sinir ağları ortaya çıkmıştır. Evrişimli sinir ağları Flatten katmanıyla ikiye bölünmüş şekildedir. İlk kısım resmi direkt 2 boyutlu şekilde kullanarak biraz önce kaybettiğimizi söylediğimiz özellikleri öğrenebilmektedir (Feature Learning). Daha sonra bu çıkarılan özelliklerden oluşan 2 boyutlu son katman Flatten ile düzleştirilir. Bundan sonraki Dense katmanları da bu "feature"ları (özellikleri) alarak sınıflandırma kısmını yaparlar.



Evrişimli sinir ağlarında iki katman türü bulunur. Bunlar convolution (evrişim) ve pooling (havuzlama) katmanlarıdır. Bunlardan convolution resimdeki özellikleri (örneğin aracın farını) öğrenmeye çalışan katmandır. Peki convolution katmanları nasıl çalışır?

Convolutional katmanı filtrelerden oluşur. Bu filtreler de nöronlar gibi öğrenebilmektedirler ancak filtreler resimin üzerinde dolaşan matrixlerdir. Resim değerlerinde belli bir özelliğin olup olmadığını aramaktadırlar. Bir filtre aşağıda gördüğünüz gibi resim üzerinde ilerleyerek resimde denk düşen yerleri kendi katsayılarıyla çarpar. Daha sonra bu çarpımların sonuçlarını toplar. Eğer aradığı özelliğe benzer bir alan varsa bu toplam çok daha yüksek çıkacaktır. Bu sayede aradığı özelliğin nerelerde bulunduğunu gösteren bir harita/resim ortaya çıkmış olur. Bu resim de yeni bir evrişim katmanına konularak özellikleri çıkarılabilir veya Flatten katmanı ile düzleştirilip sınıflandırma yapılabilir.





Bu örnekteki filtre de gördüğünüz gibi 7 harfinin üst kısmını iyi tespit etti. Ancak alt kısmını tespit edemedi. Etmemesi de gerekiyordu zaten. Filtre gördüğünüz gibi düz yatay çizgileri bulmak üzere eğitilmiş. Tabi çok daha fazla filtre kullanabiliyoruz. Yani başka filtre de yedinin alt kısmını tespit edebilir. Sonra diğer rakamlar için başka filtreler daha… Derin öğrenmenin iyi yanı filtreleri de kendi öğrenmesi sayesinde ne kadar büyük olmasına ihtiyaç duyuyorsak birkaç sayıyı değiştirerek o kadar büyük katmanlar elde edebiliyor olmamız. Evrişim katmanı eklemek için kod şu şekildedir:

```
model.add(tf.keras.layers.Conv2D(8, (3,3), (2,2)))
```

8 = Katmanda kaç filtre olacak. (3,3) = filtre boyutu. (2,2) = filtrenin kaç kare ilerleyeceği.

Pooling katmanı ise çok daha basit bir şekilde resmin boyutunu küçültmeye yarar. İki tür pooling vardır: max pooling ve average pooling. Max pooling kapsadığı alandaki en büyük pikseli alır. Average pooling kapsadığı alandaki piksellerin ortalamasını alır. Pooling işlemi genellikle işlem hacmini azaltmak için yapılır.
Modelimize max pooling ekleyelim:

```
model.add(tf.keras.layers.MaxPooling2D((2,2)))
```

Buradaki (2,2) kapsadığı alanı belirtir. Her eksende 2 piksel genişliğinde.
Modelimize average pooling ekleyelim:

```
model.add(tf.keras.layers.AveragePooling2D((2,2)))
```

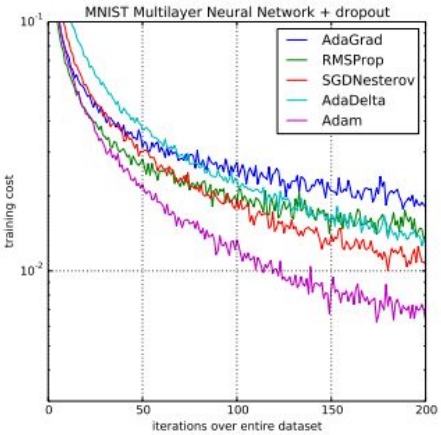
Compile, Optimizer ve Loss

Compile fonksiyonunun amacı modelin eğitim için gereken parametrelerini ayarlayarak eğitilebilir hale getirmektir. Model tahmin yaparken bu işleme ihtiyaç duymaz. Compile fonksiyonunu çalıştırmak için optimizer ve loss/cost fonksiyonunu belirlememiz gerekmektedir. Compile işlemi, bu fonksiyonları sadece adını yazarak modele implement etmemizi sağlamaktadır. Compile işlemi şu şekilde yapılmaktadır:

```
model.compile(optimizer="adam", loss="sparse_categorical_crossentropy",
metrics=["accuracy"])
```

Metrics eğitim sırasında hangi verileri görmek istediğinizdir.

Optimizer'lar, ağırlıkların en iyi sonucu verecek ayarlanması işini yapan, yani maliyet-ağırlık grafiğindeki global minimum (global optimum da denilir) değerlerini bulmaya çalışan algoritmalardır. Sayfa 8'de gördüğümüz gradient descent de bir optimizer olmakla birlikte en iyilerinden biridir. Çok sayıda optimizer da gradient tekniğini kullanır ancak gradient descent'ten daha optimize çalışıyor olabilirler. Hangi optimizer'ı kullanmanın en iyi olduğu modelin yapısına, veriye, problemin türüne ve birçok farklı değişkene bağlıdır. Aşağıda en çok kullanılan bazı optimizerlar ve MNIST veri seti üzerinde bir modeldeki sonuçları verilmiştir.



En iyi sonuç veren ''Adam'' olmuştur. Derin ağlarda Adam genelde iyi performans gösterir.

Keras ile birçok farklı optimizer kullanılabilmektedir. Optimizer'ı compile fonksiyonunun içinde belirlediğimizi söylemiştik. Bunu iki şekilde yapabiliyoruz. Öncelikle eğer parametrelerle çok ilgilenmiyorsak ve Keras ile geldiği gibi kullanacaksak sadece bir string olarak ismini yazabiliyoruz. Bu yöntem keras ile gelen birçok optimizer için geçerli.

```
model.compile(optimizer="adam", loss="...", metrics=["..."])
```

Diğer yöntem ise optimizer'ı keras'ın optimizers modülünden almak.

```
adam = tf.keras.optimizers.Adam(learning_rate=0.01, decay=0.0001)
model.compile(optimizer=adam, loss="...", metrics=["..."])
```

Bu şekilde gördüğünüz gibi optimizer'ın parametrelerini değiştirebiliyoruz. Learning rate'den zaten bahsetmiştik. Decay de learning rate'in her epoch'da biraz azalmasını sağlıyor. Bunlar dışında birçok değiştirebileceğiniz parametre de bulunuyor. Daha fazla bilgi için: keras.io/optimizers/

Loss fonksiyonu, sayfa 6-7'de gördüğümüz cost fonksiyonu ile aynı anlama geliyor. Yalnızca cost tüm veriye uygulanmış hali, loss tek bir veri için olan hali. Farklı problemlerin hata tanımları birbirinden farklı olacağı için çözmeye çalıştığınız probleme uygun bir loss seçmeniz gerekiyor. Biz şimdiye kadar iki problem türü gördük. Regresyon (regression) ve Sınıflandırma (classification).

Regresyon için kullanılan önemli loss fonksiyonlarından bazıları şunlardır:

- Mean Square Error
- Mean Absolute Error
- Mean Bias Error

Sınıflandırma için kullanılan önemli loss fonksiyonlarından bazıları şunlardır:

- Hinge Loss (Binary)
- Cross Entropy Loss (Binary)
- Sparse Categorical Cross Entropy (Çok sınıflı)

Biz genel olarak Mean Square Error ve Sparse Categorical Cross Entropy kullanıyoruz. Her fonksiyonun kendine göre gereksinimleri ve veri sınıflandırma şekilleri oluyor. Mean Square ile tek bir output için regresyon yapacaksanız outputun tek bir nöron olması gerekli. Sparse için her sınıf için output katmanında bir nöron, yani 10 sınıf için 10 nöron olmalı.

Kullanabileceğiniz loss fonksiyonları: keras.io/losses/
Daha fazla bilgi:
https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/

Modeli Eğitmek

Tüm işlemleri yaptınız ve artık eğitmeye hazır bir modeliniz mi var? Haydi o zaman eğitelim. Keras'ta training fonksiyonu çok basit şekilde başlatılabiliyor.

```
model.fit(train_images, train_labels, epochs=10)
```

Burada train_images hazırladığınız veri, train_labels verinizi etiketleri oluyor. Epoch se tüm veri setini eğitime kaç kez koyacağınızı belirttiğiniz parametre. Modelin tüm veriyi bir kez görmesi genelde yeterli olmuyor. Birkaç kez tekrar yapmanız gerekiyor. Epoch sayıs genel olarak onlarla sınırlı kalsa da bazı problemler ve model yapıları için on binin üzerinde epoch gerekebiliyor. Epoch sayısını belirlemenin genel yolu deneme yanılma. Az epoch yeterince öğrenmemesine, çok epoch ezberlemesine neden olabilir. Yapabileceğiniz şey ise 20 epoch gibi ortalama bir sayı seçip ne olduğunu görmek. Yeterince iyi değilse artırıp daha iyi olacak mı görebilirsiniz. Eğer son birkaç epoch sürekli aynı sayılarda dolaştıysa daha fazla öğrenemiyor demektir. Öğrenmemeye başladıktan sonra aynı model yapısı ve veri ile daha fazla epoch fayda etmeyecektir.

Eğitilmiş Modeli Kaydetmek

Eğittiğiniz bir modeli daha sonra kullanım için kaydetmek amacıyla kullanılan birçok yöntem vardır. Biz ise Tensorflow'un kullandığı SavedModel formatını kullanacağız. Keras'ın kendi yöntemleri bulunsa da SavedModel aynı modeli tüm Tensorflow sistemlerinde aynı şekilde kullanmamızı sağlayan, ister web, ister mobil olsun ya da python dışında bir dil kullanıyor olalım fark etmeyen bir model saklama formatı olduğu için çok daha kullanışlı.
Modeli şu şekilde kaydediyoruz:

```
tf.keras.experimental.export_saved_model(model, 'PATH/TO/MODEL')
```

Buradaki 'PATH/TO/MODEL' modeli saklamak istediğiniz lokasyon.

Modeli Tahmin için Kullanmak

Modelimizi kaydettikten sonra herhangi bir python scriptinden çağırıp tahmin yapmasını isteyebiliyoruz. Bunun için kullanacağımız kod şu şekilde:

```
model = tf.keras.experimental.load_from_saved_model('PATH/TO/MODEL')
prediction = model.predict(image)
```

Bir kez load işlemini yaptıktan sonra istediğiniz kadar predict yapabilirsiniz. Sonrasında prediction'ı alıp istediğiniz şekilde kullanabilirsiniz.

Bu Dökümanı Yazan: Cihan Alperen Bosnalı (cihanbosnali@gmail.com)

Okuduğunuz İçin Teşekkürler!