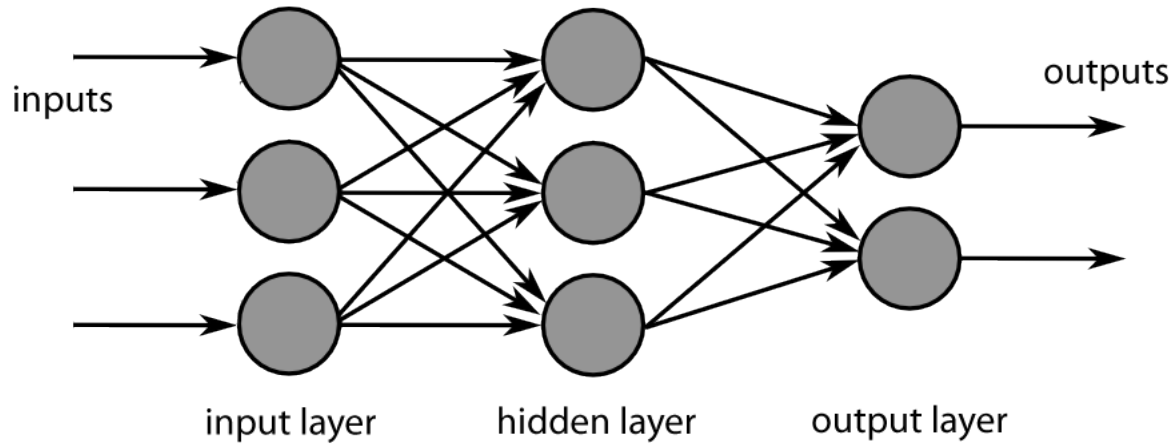
Kemalcan Bora
Samet Çetin

YAPAY SİNİR AĞLARI

Yapay sinir ağları katmanlardan oluşur. Katmanlar ise çeşitli sayıda nöronların bir araya gelmesiyle şekillenir. Yapay sinir ağlarının ilk katmanı input katmanı, son katmanı output katmanı ve bu iki katmanın arasında yer alan tüm katmanlar hidden katman olarak adlandırılır. En temel tipteki yapay sinir ağları feedforward mimariye sahiptir. Bu tip yapay sinir ağlarında, hidden katmanlardaki nöronların kendilerinden bir önce gelen katmana bağlı input bağlantıları ve kendilerinden bir sonra gelen katmana bağlı output bağlantıları vardır.



Yapay sinir ağı modellerinin çalışma prensibi kısaca; input katmanına gönderilen input verilerinin, çeşitli şekillerde konumlandırılabilecek hidden katmanlardan geçerek output katmanına ulaşması ve hedeflenen bilginin output olarak elde edilmesidir. Başlangıçta, hidden katmanlardaki nöronların her birine, rastgele ve otomatik olarak bir weight değeri atanır. Eğitim, input katmanına input verilerinin gönderilmesiyle başlar. Weight değerleri modelin öğrenme süresi boyunca, daha doğru bir output üretebilmek adına sürekli güncellenir.

Weight değerini daha iyi anlayabilmek ve input verileriyle ne gibi bir ilişkisi olduğunu kavrayabilmek için katmandaki bir nöronun işlevini ve modelin öğrenme sürecini nasıl gerçekleştirdiğini incelememiz gerekir.

Her bir nöronun, basitçe, bir baraja benzetebileceğimiz aktivasyon fonksiyonları vardır. Farklı yapıda barajlar olabileceği gibi aktivasyon fonksiyonlarının da farklı çeşitleri vardır. Nöron, kendisine gelen tüm input verilerini kendi weight değeri ile çarpar ve çıkan sonucu aktivasyon fonksiyonuna sokar. Bu aşama, önce farklı kanallardan gelen suyun bir araya getirilmesi ve sonrasında suyun baraja yönlendirilmesi olarak düşünülebilir. Aktivasyon fonksiyonlarının outputları 0 ile 1 arasında ya da -1 ile 1 arasında değişir ve nöron, eğer toplam input değeri yüksekse 1’e yakın, düşükse 0’a ya da -1’e yakın bir değeri output olarak ateşler. Yani su miktarı ve dolayısıyla su seviyesi ne kadar yüksek olursa barajı aşan (ateşlenen) su

miktarı o kadar fazla olur. Ateşlenen bu output değeri bir sonraki katmanlar için input verisi olarak kullanılır ve böylece tüm nöronların dolaylı veya doğrudan birbirine bağlı olduğu bir ağ oluşturulur.

Öğrenme süresince baştan sona tüm inputlar, tekrar tekrar modele sokulur ve böylece weight değerleri güncellenir. Tüm inputların baştan sona sadece bir defa modele sokulmasına epoch döngüsü denir. Genellikle modelin doğru çalışır, son haline gelmesi için birden fazla epoch döngüsüne ihtiyaç duyulur.

Model, öğrenme sürecini backpropagation adı verilen yöntemle gerçekleştirir. Bu yöntem eğitim süresince, model üzerinde gerekli güncellemeleri yapabilmek için bir -loss function- kayıp fonksiyonuna ve optimizer algoritmasına ihtiyaç duyar. Her bir epoch döngüsü sonucu elde edilen outputlar kayıp fonksiyonuna yerleştirilir. Kayıp fonksiyonu modelin verdiği outputların gerçek ve doğru sonuçlara ne kadar yakın veya uzak olduğunu ölçer. Eğer outputlar gerçek sonuçlardan uzak ise kayıp fonksiyonunun döndürdüğü değer, kaybın fazla olduğunu gösterir şekilde, yüksek olur. Eğer outputlar gerçek değerlere yakınsa kayıp fonksiyonunun döndürdüğü değer küçük olur. Daha sonra model çeşitli optimizer algoritmalarından seçili olanı kullanarak son katmanlardan ilk katmanlara (geriye) doğru kayıp fonksiyonunun değerini minimize edecek şekilde nöronların weight değerlerini sırayla günceller. Bu ileriye ve geriye doğru ilerleyen öğrenme yöntemine backpropagation adı verilir. İşlem toplam epoch döngü sayısı boyunca devam eder ve sonlandığında, model eğitimi de sonuçlanmış olur.

CORE KATMANLAR

Keras kütüphanesindeki core katman yapıları kullanılarak temel feedforward modelleri inşa edilebilir ve katman nöronları konfigüre edilebilir.

Dense

Dense, yapay sinir ağı modelinin temel katman yapısını ifade eder. Ardışık katmanların tüm nöronları birbiriyle bağlantılıdır (fully-connected). Dense yapısı `add()` fonksiyonuyla birlikte modelde katman inşa etmek için kullanılır. Katmandaki nöron sayısını (**units**), nöronlarda kullanılacak aktivasyon fonksiyonlarını (**activation**), modele girecek input verilerinin boyutlarını (**input_dim** veya **input_shape**) tanımlamamıza olanak sağlar. Input_shape veya input_dim yalnızca input katmanında kullanılır, takip eden katmanlarda input şekli otomatik olarak algılanır.

Input_dim, veri kümesinin kaç sütundan oluştuğunu, yani her bir veri örneğinin kaç farklı özelliği olduğunu gösterir. Örneğin elimizde ölçümlerini daha önceden yaptığımız plastik toplar olduğunu düşünelim. Topların ölçümünü yaptığımız özellikleri hacim, çap ve yüzey alanı olsun. Bu verileri sakladığımız dosyanın her bir satırında farklı bir top yer alır. Sütunlar ise 3’e ayrılmıştır ve her biri farklı bir top özelliğini temsil eder. Bu verileri yapay sinir ağına sokmadan önce, ağın input

katmanında input_dim'i belirtmek gerekir ve bu durumda input_dim 3 olarak seçilmelidir.

```
model = Sequential()
# Her bir top verisinin üç farklı özelliği için input_dim 3 seçilir.
model.add(Dense(32, input_dim=3))

# Sonraki katmanlar için tekrar input şekli tanımlamak gerekmez.
model.add(Dense(16))
```

Eğer işlem yaparken bir takım hafıza kısıtlamalarına takılacak olunursa, batch yapısı kullanılarak bunun üstesinden gelinebilir. Batch yapısı kullanılarak eldeki veri kümesi, alt kümelere (batchlere) ayrılır ve işlemler parça parça yapılarak model güncellenir. Böyle durumlarda ilk katmana tüm input verileri değil belirlediğimiz batch boyutuna göre değişen sayıda input verisi girer. Ancak yine de, batch size'ı tüm veri sayımızı yansıtacak şekilde ayarlayarak kullanabiliriz. Elimizde 500 plastik top verisi olduğunu kabul edersek, batch_size'ı 500 olacak şekilde ayarlamak sıkıntı yaratmayacaktır. Ancak hafıza kısıtlamasından kaçınmak isteniyorsa batch_size, örnek olarak, 32 seçilebilir. Batch_size'ın 32 olması epoch döngüsü sayısını değiştirmez. Sadece bir epoch döngüsü için 32'lik parçalara bölünmüş batchlerin 500 sayısını tamamlayacak şekilde sırayla modele sokulması beklenir. 1 epoch döngüsü, ancak bu işlem tamamlandığında gerçekleşmiş sayılır.

```
model = Sequential()
# Elimizdeki toplam top verisi sayısı 500 olduğu için batch_size 500 seçilir.
model.add(Dense(32, input_dim=3, batch_size=500))

# Hafıza kısıtlamalarından dolayı batch_size 32 olarak ayarlanabilirdi..
model.add(Dense(32, input_dim=3, batch_size=32))

# Sonraki katmanlar için tekrar input şekli tanımlamak gerekmez.
model.add(Dense(16))
```

Input_dim'in yanında bir de input_shape parametresi vardır. Yine input_dim'de olduğu gibi input_shape'te de toplam veri sayısı yerine, eldeki her bir verinin boyutları tanımlanır. Ancak dense katman yapısında azami 2 boyutlu bir input verisinden bahsedilebilir. Diğer bazı katman yapılarında boyut sayısı daha fazla olabilir.

```
model = Sequential()

# Aşağıdaki input_shape kullanımı input_dim=3 kullanımıyla tamamen aynıdır.
# input_shape=(3,) yerine input_shape=(3,None) da kullanılabilir.
model.add(Dense(32, input_shape=(3,)))

# Sonraki katmanlar için tekrar input şekli tanımlamak gerekmez.
model.add(Dense(16))
```

Kemalcan Bora
Samet Çetin

Açıklayıcı olmak adına MNIST veri kümesini ele alalım. MNIST veri kümesindeki veriler on rakamdan birininin el yazısı karakterini içeren görüntü dosyalarıdır. Her bir görüntü 28x28 boyutundadır. Kısacası veri kümesindeki her bir veri (görüntü) 784 (28x28) piksel içermektedir. Dosyanın her bir satırındaki verilerimiz için rakamın ne olduğunu belirten etiket ile birlikte 785 adet sütun vardır.

```
model = Sequential()

# Aşağıdaki input_shape kullanımı input_dim=785 kullanımıyla tamamen aynıdır.
model.add(Dense(128, input_shape=(785,)))
model.add(Dense(1))
```

Activation

Nöronlarda kullanacağımız aktivasyon fonksiyonlarını modeli oluştururken tanımlayabiliriz. Aktivasyon fonksiyonlarını Dense yapısı içerisinde tanımlayabileceğimiz gibi, ayrı bir biçimde Activation yapısını kullanarak da tanımlayabiliriz ve `add()` fonksiyonunun yardımıyla modele ekleyebiliriz. Aktivasyon fonksiyonlarının seçimi modelin öğrenmesini etkili bir biçimde farklılaştırır. Doğru sonuçlara varmak adına, model katmanlarında farklı aktivasyon fonksiyonlarının denenmesi faydalı olabilir. Aktivasyon fonksiyonları katmanlara özel tanımlanır ve eğer istenirse her bir katman için farklı fonksiyon tanımlanabilir. Fonksiyonu tanımlanmış belli bir katmanın tüm nöronlarının aktivasyon fonksiyonları aynı olur.

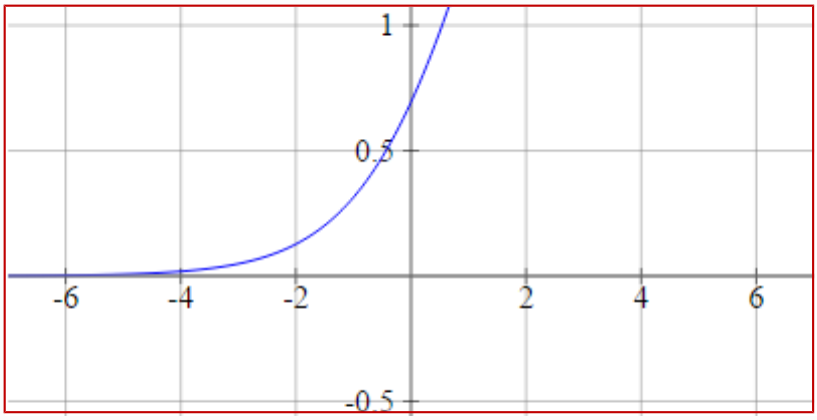
```
model = Sequential()
model.add(Dense(32, input_shape=(784,)))
model.add(Activation('relu'))
# Yukarıda oluşturulan Sequential model ile aşağıdaki arasında işleyiş bakımından
# hiçbir fark yok
model = Sequential()
model.add(Dense(32, input_shape=(784,), activation='relu'))
```

Yapay sinir ağları bölümünde açıklandığı gibi aktivasyon fonksiyonlarının yapısı baraj yapılarına benzerlik gösterir. Bir nöronda toplanan tüm input değerleri birleştirilip, nöronun aktivasyon fonksiyonuna gönderilir. Aktivasyon fonksiyonu input miktarına göre bir sonraki katmana aktarılacak olan output miktarını belirler. Output değerleri, yapay sinir ağlarının çalışma mantığı dolayısıyla ya 0 ile 1 arasında ya da -1 ile 1 arasında değerler alır. Aktivasyon fonksiyonlarının çeşitli türleri vardır. Bunlardan kısaca bahsedelim.

softmax

Kemalcan Bora
Samet Çetin

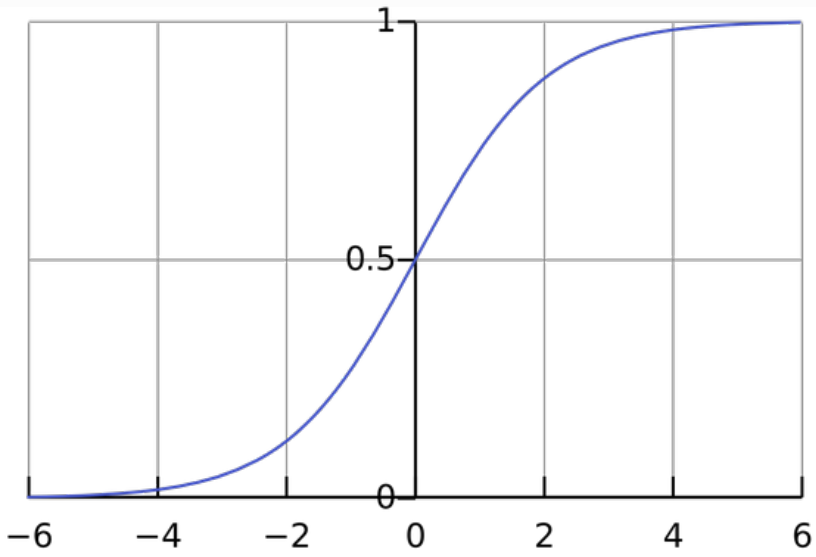
softmax aktivasyon fonksiyonun grafiği aşağıdaki gibidir. Yatay eksen fonksiyona giren input değerini, dikey eksen ise output değerini ifade eder. Input değeri ne kadar yüksek olursa nöronun ateşleyeceği output değeri 1'e o kadar yakın olur. Input değerinin az olması durumunda ise output değeri 0'a yakın olur.



softplus

softplus aktivasyon fonksiyonun grafiği aşağıdaki gibidir. Yatay eksen fonksiyona giren input değerini, dikey eksen ise output değerini ifade eder.

relu

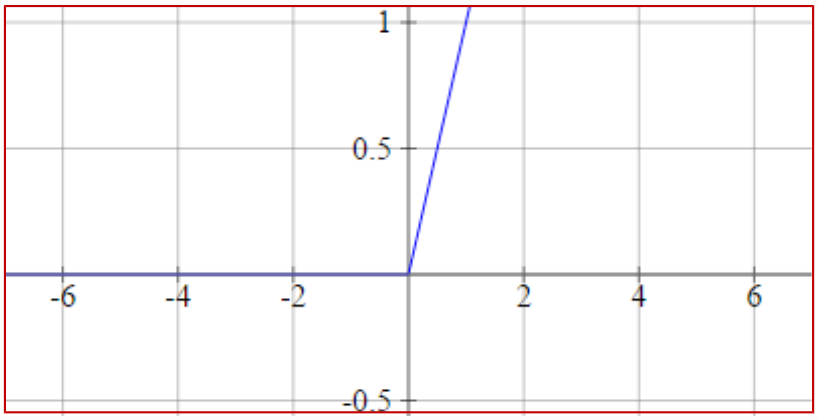


Kemalcan Bora
Samet Çetin

relu aktivasyon fonksiyonun grafiği aşağıdaki gibidir. Yatay eksen fonksiyona giren input değerini, dikey eksen ise output değerini ifade eder.

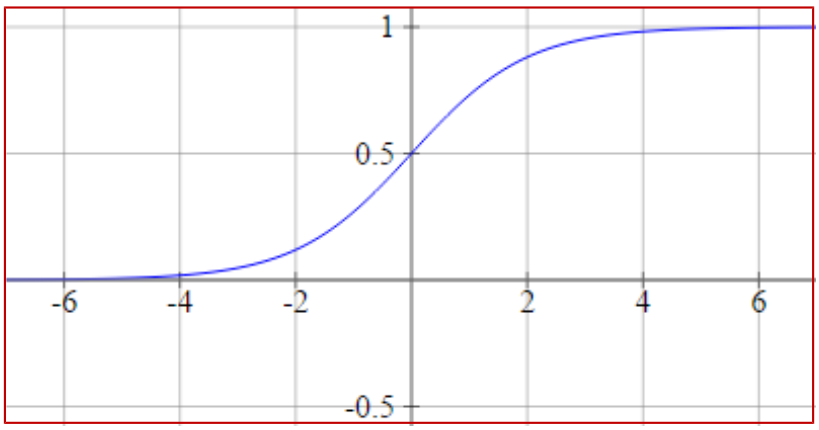
sigmoid

sigmoid aktivasyon fonksiyonun grafiği aşağıdaki gibidir. Yatay eksen fonksiyona giren input değerini, dikey eksen ise output değerini ifade eder.



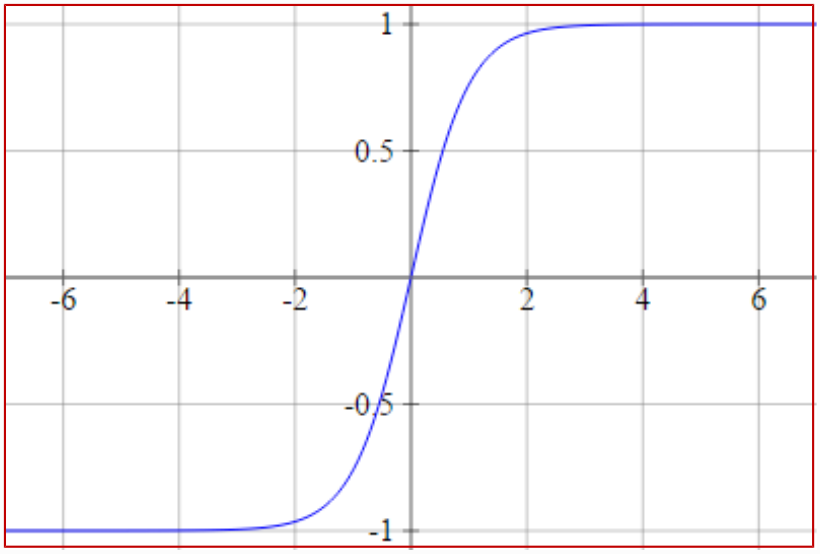
tanh

tanh aktivasyon fonksiyonun grafiği aşağıdaki gibidir. Yatay eksen fonksiyona giren input değerini, dikey eksen ise output değerini ifade eder.

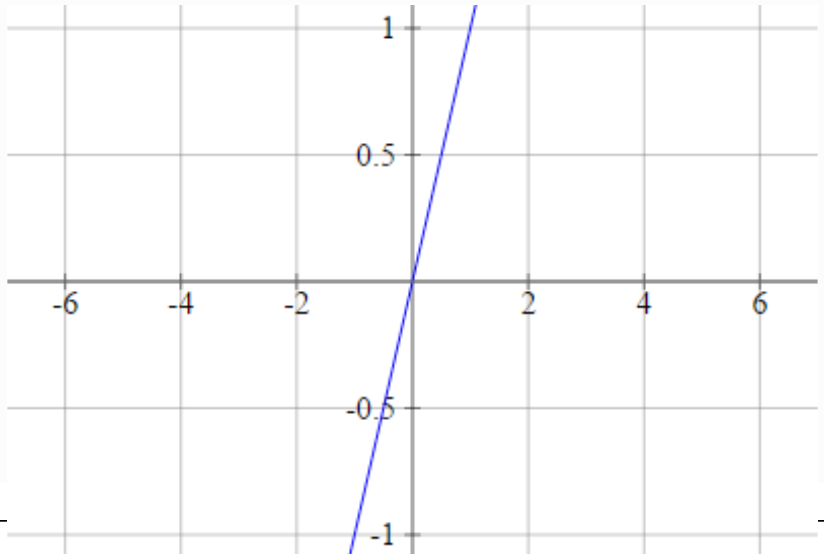


Kemalcan Bora
Samet Çetin

linear



linear aktivasyon fonksiyonun grafiği aşağıdaki gibidir. Yatay eksen fonksiyona giren input değerini, dikey eksen ise output değerini ifade eder.



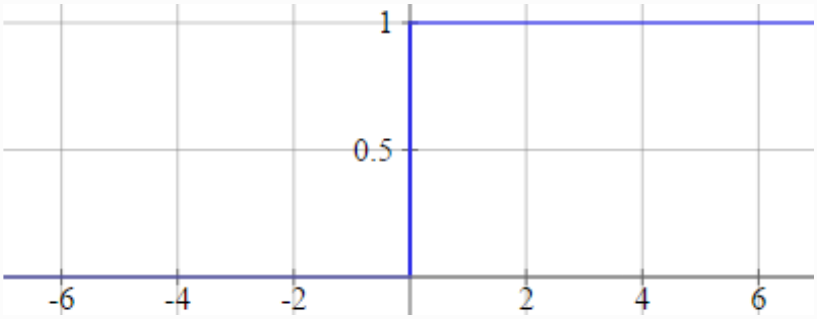

```
# Çeşitli akticasyon fonksiyonlarının modelde kullanım örnekleri
model = Sequential()
model.add(Dense(32, input_dim=5))
model.add(Activation(softmax))
model.add(Dense(16))
model.add(Activation('tanh'))
```

Kemalcan Bora
Samet Çetin

```
model.add(Dense(1, activation='sigmoid'))
```

step

step aktivasyon fonksiyonun grafiği aşağıdaki gibidir. Yatay eksen fonksiyona giren input değerini, dikey eksen ise output değerini ifade eder.



Dropout

Dropout yapısı modelin overfitting yapmasını yani ezberlemesini önlemek için kullanılır. Overfitting, modelin eğitim veri kümesiyle gereğinden uzun süre eğitilmesi sonucu ortaya çıkan, istenmeyen bir durumdur. Model, eğitim veri kümesini ezberler ve bu verilerle çok doğru outputlar verir fakat, daha önceden modelin karşılaşmadığı test verileri input olarak kullanıldığında, model eğitim veri kümesindeki başarılı sonuçları tekrar edemez. Dropout kullanımı eğitim esnasında modelin yapısını değiştirerek, nöron değerlerinin farklı şekillerde güncellenmesini sağlar.

Dropout her bir epoch öncesi model katmanlarındaki nöronların rastgele seçilen belli bir kısmını input ve output bağlantılarıyla birlikte siler. Epoch bitiminde silinen nöronlar geri gelir ve bir sonraki epoch öncesi rastgele seçim tekrar yapılır ve işlem model öğrenmesi bitene kadar tekrarlanır. Bunun sonucunda nöronlar değişen model yapısına uyum sağlayacak şekilde eğitilirler ve modelin ezberlemesinin mümkün olduğunca önüne geçilmiş olur.

```
# Dropout yapısı kullanılmadan oluşturulmuş bir model.
model = Sequential()
model.add(Dense(32, input_dim=5))
model.add(Activation('relu'))

# Aşağıda kullanılan Dropout yapısı katmandaki nöronların 0.2'sini (5'te 1'ini) # her bir epoch başlangıcında rastgele seçer ve epoch boyunca erişilemez hale # getirir.
model.add(Dropout(0.2))
model.add(Dense(16, input_dim=5))
```

Kemalcan Bora
Samet Çetin

```
model.add(Dropout(0.2))
model.add(Dense(1))
model.add(Activation('sigmoid'))
```

RECURRENT KATMANLAR

Keras kütüphanesindeki recurrent katman yapıları kullanılarak recurrent modeller inşa edilebilir. Recurrent yapay sinir ağı modelleri, katmanlardaki nöronların yalnızca kendilerinden önce ve sonra gelen katmanlarla ilişkili değil, aynı zamanda bulundukları katmandaki diğer nöronlarla da bağlantılarının olduğu model yapılarıdırlar. Recurrent modeller o anda modele giriş yapan input verilerinin yanında geçmiş zaman verilerinden de yararlanarak eğitilirler. Zaman serileri gibi input verilerinin sıralamalarının da önemli olduğu, outputların zaman gibi değişkenlere göre bazı dönemler farklı eğilimler sergiledikleri veri kümelerinde kullanılırlar. Böyle veri kümelerinde, inputların yanı sıra input dizilerinde de öğrenilebilir bilgiler saklıdır. Feedforward yapay sinir ağlarının aksine recurrent ağlar bu diziler ve sıralamalardan da yararlanabilecek şekilde tasarlanmıştır. Ses tanıma, çeviri, görüntü yakalama gibi alanlarda da recurrent katmanlar başarılı bir biçimde kullanılmaktadır.

Keras kütüphanesinde 3 farklı recurrent katman yapısı vardır. Bunlar **LSTM**, **GRU** ve **SimpleRNN**'dir. Recurrent katmanlar core katmanlarına çok benzer yapıda kurulurlar.

```
# Sequential modelin ilk katmanı
model = Sequential()
model.add(LSTM(32, input_shape=(10, 64)))
# Bir Recurrent katman yapısı olan LSTM katmanı modele yerleştirildi.

# Takip eden katmanlar için input şeklini belirtmek gereksiz.
model.add(LSTM(16))

# Eğer modelimizde birbirini takip eden recurrent katmanlar kullanılacaksa
# bir sonraki katman için return_sequences=True ifadesinin kullanılması gerekir. # Ancak
unutulmamalıdır ki, input shape'in sadece ilk katmanda belirtilmesi # yeterlidir.
model = Sequential()
model.add(LSTM(64, input_dim=64, input_length=10, return_sequences=True))
model.add(LSTM(32, return_sequences=True))
model.add(LSTM(10))
```

LSTM (Long-Short Term Memory)

Kemalcan Bora
Samet Çetin

Recurrent yapay sinir ağı modellerini eğitmek feedforward modellerle kıyaslandığında çok daha zordur. Bunun nedeni vanishing/exploiding gradient problemidir. LSTM yapıları bu problemin üstesinden gelmek amacıyla geliştirilmiştir ve günümüzde recurrent modellerde yaygın bir biçimde kullanılmaktadır.

Bir yapay sinir ağı modelinin ilk katmanı veri kümesindeki basit ve kolay bağıntıları tespit etmeye yarar. İkinci katman ilk katmandan gelen verileri kullanarak daha karmaşık ilişkileri tespit eder ve sonrasında gelen katmanlar da aynı şekilde, giderek daha da karmaşıklaşan ilişkileri algılayıp, modelin bu ilişkileri çözümleyebilecek şekilde eğitilmesini sağlarlar. Sonuç olarak katman sayısı arttıkça işe yarar ve doğru bir output elde etmek olası hale gelir. Fakat dikkat edilmelidir ki, gereğinden fazla katman kullanımı, modeli olması gerekenden daha kompleks hale getirir ve basit problemlerin çözümü bile kolayca yapılamayacak hale gelir. Katman sayısının artışı, modelin her zaman daha doğru olacağı anlamına gelmez. Bu yüzden katman sayısını kontrolsüzce artırmak faydalı bir uygulama pratiği değildir. Yine de karmaşık problemlerde katman sayısının fazla olması gerekir. Ancak simpleRNN katman yapılarıyla kurulacak olan böyle modellerde beklenen sonuç elde edilemez. Modelin erken katmanlarının öğrenmesi yavaş, geç katmanlarının öğrenmesi ise hızlı olur. (Vanishing Gradient) Erken katmanların tam olarak öğrenemediği basit bağıntıların geç katmanlar tarafından hızlıca öğrenilmesi modelin istenmeyen sonuçlar vermesine yol açar. Bazen bu durumun tam tersi olur ve modelin erken katmanları hızlı, geç katmanları ise yavaş öğrenir. (Exploiding Gradient) Aynı şekilde bu durum da modelin yanlış sonuçlar vermesine neden olur. LSTM katman yapıları işte bu problemi aşmak ve recurrent modellerin doğru oluşturulabilmesini sağlamak amacıyla geliştirilmişlerdir.

LSTM katman yapısında, katmanlardaki nöronların kendi hafızaları bulunmaktadır. Bu hafızalarda - zaman serilerinden örnek verilirse - geçmiş zaman verileri depolanır ve model geliştirilirken bu verilerden de yararlanılır. Hafızalarda hangi verilerin depolanacağı eğitim sürecinde şekillenir. Böylece oldukça eski inputların veya input dizilimlerinin bile yeni inputlarla etkileşime girebileceği ve output üzerinde etkilerinin olabileceği bir ağ yapısı oluşturulmuş olur.

```
# Sequential modelin ilk katmanı
model = Sequential()
model.add(LSTM(32, input_shape=(10, 64)))
# LSTM katmanları modele tıpkı bir core katman ekler gibi eklenebilirler.
model.add(LSTM(16))
```

LSTM katmanları recurrent yapıda oldukları için daha önceki verilerden de yararlanabilecek şekilde tasarlanmışlardır. Bu yüzden, yüksek verim alabilmek için, karşılaşılan problemin recurrent bir çözüm modeline uygun olması gerekir. Aynı şekilde veri kümesinin de modele sokulmadan önce düzenlenmesi şarttır. Zaman serileri için konuşursak, output verisini elde etmek için, belirlenen bir anda ne kadar geçmiş zaman verisinden

yararlanılacağının saptanması ve input verilerinin bu duruma uygun şekillendirilmesi gereklidir. Kaç geçmiş zaman verisinden yararlanılacağını belirten değişkene timesteps denir.

LSTM katmanlarında input verilerinin boyutlarını çeşitli şekillerde tanımlayabiliriz. Örneğin elimizde 30 adet günlük sıcaklık verisi olduğunu ve bunları kullanarak önümüzdeki günler için sıcaklık tahmini yapmaya çalıştığımızı düşünelim. (Forecasting) Öncelikle kaç tane geçmiş sıcaklık verisinden yararlanacağımızı yani timesteps'in kaç olacağını belirlememiz gerekir. Bu örnekte timesteps'i 3 olarak seçtiğimizi kabul edelim. Batch_size, yani eğitime hazır toplam veri sayısı, 30, timesteps 3 ve input boyutumuz (sıcaklık değeri) 1 olur.

```
model = Sequential()
model.add(LSTM(30, batch_input_shape=(30, 3, 1)))
model.add(Dense(1))
```

Yukarıdaki model, batch_size'ı hiç kullanmadan, aşağıda verilen şekillerdeki gibi de tanımlanabilir.

```
model = Sequential()
model.add(LSTM(30, input_shape=(3, 1)))
model.add(Dense(1))
```

```
model = Sequential()
model.add(LSTM(30, input_length=3, input_dim=1)))
# Timesteps'i belirtmek için input_length, input boyutunu belirtmek için input_dim #
kullanıldı.
model.add(Dense(1))
```

COMPILATION

Katmanları problemin sonucuna uygun olacak şekilde inşa edilen modeli, eğitmeye başlamadan önce, öğrenme sürecinde kullanılacak parametrelerin belirlenmesi gereklidir. Bu işlem `compile()` fonksiyonu kullanılarak yapılır. `compile()` fonksiyonu üç farklı parametreden yararlanır. Bunlar loss function (kayıp fonksiyonu), optimizer ve metric listesidir. Kayıp fonksiyonu ve optimizer bir modelin compile edilmesi için olmazsa olmazlardandır. Metric listesi opsiyonel olarak kullanılabilir.

Losses

Kayıp fonksiyonu, yapay sinir ağları başlığının altında bahsedildiği gibi, modelin her bir epoch döngüsü sonunda outputları yerleştirdiği ve sonucun ne kadar hatalı ya da kayıbın ne kadar fazla olduğunu gösteren fonksiyondur. Kayıp fonksiyonunun sonucuna göre modeldeki nöronların weight değerleri düzenlenir.

Kemalcan Bora
Samet Çetin

```
model = Sequential()
model.add(LSTM(10, input_shape=(10, 2)))
model.add(Activation('relu'))
model.add(Dense(1))
# Modelin eğitimi esnasında kayıp fonksiyonu olarak mean squared error
# optimizer olarak ise sgd (stochastic gradient descent) kullanıldı.
model.compile(loss='mean_squared_error', optimizer='sgd')
```

```
model = Sequential()
model.add(Dense(24, input_dim=2))
model.add(Activation('sigmoid'))
model.add(Dense(1))
# Modelin eğitimi esnasında kayıp fonksiyonu olarak mean absolute error
# optimizer olarak ise adam kullanıldı.
model.compile(loss='mean_absolute_error', optimizer='adam')
```

Keras kütüphanesinde doğrudan ulaşılabilecek bazı hazır kayıp fonksiyonları vardır. Bu fonksiyonlar doğru output değerleri ile tahmin edilmiş output değerleri arasındaki farkı (hatayı) hesaplarlar. Bu fonksiyonlardan kısaca bahsedelim.

mean_squared_error

Doğru ve tahmin edilmiş değerler arasındaki farkların karelerini alır ve bunları toplar. Daha sonra bu toplamın ortalamasını sonuç olarak verir.

```
mean_squared_error(y_true, y_pred)
# Parametre olarak kullanılan y_true doğru output değerlerini, y_pred ise tahmin
# edilmiş output değerlerini ifade eder. Fonksiyon bu iki değer arasındaki hatayı # bulur.
```

mean_absolute_error

Doğru ve tahmin edilmiş değerler arasındaki farkların pozitif değerlerini yani uzaklıklarını alır ve bunları toplar. Daha sonra bu toplamın ortalamasını sonuç olarak verir.

```
mean_absolute_error(y_true, y_pred)
# Parametre olarak kullanılan y_true doğru output değerlerini, y_pred ise tahmin
# edilmiş output değerlerini ifade eder. Fonksiyon bu iki değer arasındaki hatayı # bulur.
```

mean_absolute_percentage_error

Doğru ve tahmin edilmiş değerler arasındaki farkların pozitif değerlerini yani uzaklıklarını alır ve ardından bu değerleri doğru değerlere bölerek toplar. Daha sonra bu toplamın ortalamasını alır ve 100 ile çarparak yüzdelik hata biçimine çevirir.

Kemalcan Bora
Samet Çetin

```
mean_absolute_percentage_error(y_true, y_pred)
# Parametre olarak kullanılan y_true doğru output değerlerini, y_pred ise tahmin
# edilmiş output değerlerini ifade eder. Fonksiyon bu iki değer arasındaki hatayı # bulur.
```

mean_squared_logarithmic_error

Doğru ve tahmin edilmiş değerlerin logaritmalarının farkının alır ve ardından bu değerleri toplar. Daha sonra bu toplamın ortalamasının karekökünü sonuç olarak verir.

```
mean_squared_logarithmic_error(y_true, y_pred)
# Parametre olarak kullanılan y_true doğru output değerlerini, y_pred ise tahmin
# edilmiş output değerlerini ifade eder. Fonksiyon bu iki değer arasındaki hatayı # bulur.
```

Optimizers

Optimizerlar kayıp fonksiyonunu minimize etmek amacıyla kullanılan algoritmalardır. Keras kütüphanesinde çeşitli optimizerlar mevcuttur. Her bir optimizerın kaybı minimize etmek için farklı yöntemleri vardır ve farklı optimizer kullanımı karşılaşılan problemin tipine göre farklı sonuçlar verir. Bu yüzden farklı optimizerları deneyip, performanslarını karşılaştırmak modeli oluştururken faydalı olabilir.

```
model = Sequential()
# 64 nöron ünitesinden oluşan ve 10 özellikli (boyutlu) inputlar kabul eden bir
# katman oluşturuldu.
model.add(Dense(64, input_shape=(10,)))
# Katmanda aktivasyon fonksiyonu olarak tanh kullanıldı.
model.add(Activation('tanh'))
# Modelin eğitimi için kayıp fonksiyonu olarak mean_squared_error, optimizer # olarak
sgd (stochastic gradient descent) seçildi.
model.compile(loss='mean_squared_error', optimizer='sgd')
```

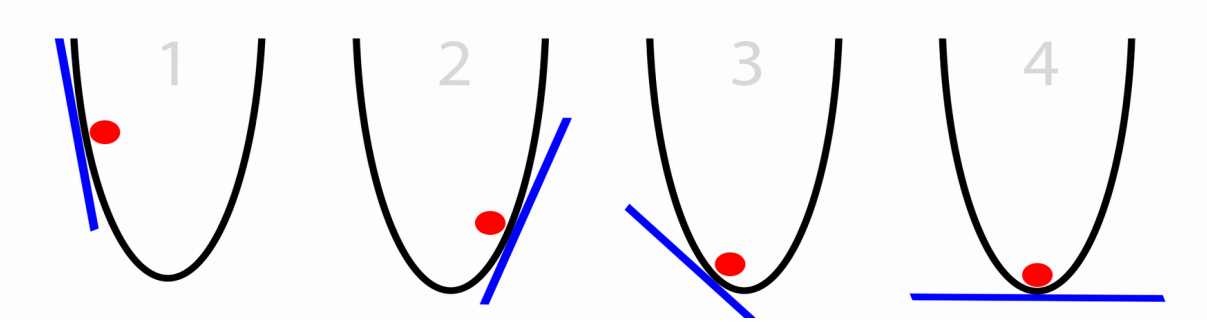
Örneğin çözmeye çalıştığımız problem için bir model oluşturduğumuzu ve kayıp fonksiyonunu belirlediğimizi kabul edelim. Kayıp fonksiyonu 2 boyutlu ve aşağıdaki grafiklerde siyah çizgi ile gösterildiği gibi olsun. Kırmızı noktalar ise modelimizin döngü sonrası hesapladığı kayıp değerleri olsun.

Kemalcan Bora
Samet Çetin

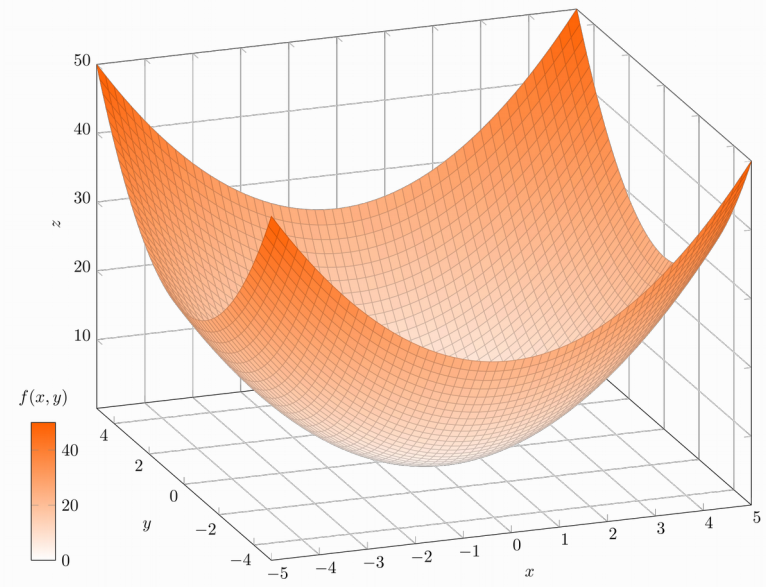
1 numaralı grafikte, modelimizin ilk ve rastgele bir şekilde belirlenmiş nöron weight değerleriyle sahip olduğu kaybın, kayıp fonksiyonu üzerindeki yerini görebiliriz. Modelin eğitimdeki amacı, kayıp fonksiyonunu minimize etmek, yani kırmızı noktayı çukurun merkezine yerleştirmeye çalışmak olacaktır. Bunu bulunduğu noktadaki eğimi hesaplayarak ve daha sonra weight değerlerini güncelleyerek yapar. Grafiklerde görüldüğü gibi epoch döngüleri sonrası, kayıp fonksiyonu minimize hale getirilmiş olur.

Grafiğin 2 boyutlu olması, elimizdeki her bir verinin iki özelliği olduğu anlamına gelir. Yani örnekteki input_dim 2'ye eşittir. Eğer verilerin 3 ayrı özelliği olsaydı, kayıp fonksiyonu da 3 boyutlu bir hale gelirdi ve daha fazla özellik sayısı için kayıp fonksiyonunun boyut sayısı da eşit olacak şekilde artardı. Optimizer algoritmaları da kayıp fonksiyonları üzerinde hareket ederken, tek bir güncellemede tüm özellikleri yani tüm boyutları minimize edecek şekilde yönelirler.

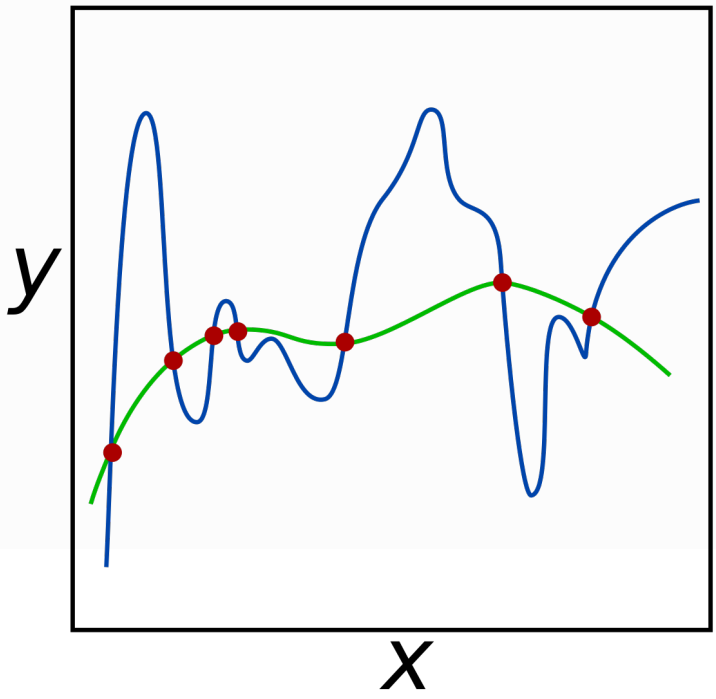
Aşağıdaki grafikte 3 boyutlu bir kayıp fonksiyonu örneği görülebilir. Her bir eksen (x, y, z) farklı veri özelliğini temsil eder. Optimizer algoritmaları yine bu fonksiyonu minimize etmek yani çukurun merkezine inmek amacıyla, nöronların weight değerlerini güncelleyecektir.



Kemalcan Bora
Samet Çetin



Gerçek hayattaki problemler için hesaplanan kayıp fonksiyonları maalesef yukarıdaki grafikler kadar kolay değildir. Genellikle kayıp fonksiyonları aşağıda görülen grafikteki gibi olurlar. Fonksiyon üzerinde global minimum değerinin yanı sıra bir çok yerel minimum değeri vardır. Modelin doğru outputlar üretebilecek şekilde eğitilmesini sağlamak için kayıp fonksiyonu optimizerlar tarafından olabildiğince minimize edilmeye çalışılır ve hedef, global minimum değerine ulaşabilmektir. Ancak genellikle kayıp değerleri yerel minimum değerlerine sıkışıp kalırlar. Bunu önlemek amacıyla çeşitli optimizer algoritmaları geliştirilmiştir ve bu algoritmalar Keras kütüphanesinde de yer almaktadır.



Kemalcan Bora
Samet Çetin

Tüm optimizerların bir learning rate parametresi vardır. Learning rate, optimizerların yaptıkları nöron weight güncellemelerinin ne kadar fazla veya hızlı yapılacağını kontrol eder. Kayıp fonksiyonunu minimize eden noktaya yaklaşırken tek seferde atılacak adımın ne kadar büyük ya da küçük olacağını belirler.

Learning rate değeri 0 ile 1 arasında değişir. Eğer rate değeri 0'a yakın bir değer olarak belirlenirse, adım uzunluğu küçük seçilmiş olur ve küçük adımlarla kayıp fonksiyonunun minimum değerine ilerlemek zaman alır. Bu yüzden epoch döngü sayısı yüksek seçilmelidir. Ancak, eğer global minimuma giden yolda bir yerel minimum noktası varsa, fonksiyon küçük adım uzunluğuna sahip olduğundan dolayı bu yerel minimumda sıkışacaktır ve epoch döngü sayısı ne kadar fazla olursa olsun hep yerel minimum noktasına yakın değerler etrafında dolaşacaktır. Eğer rate değeri 1'e yakın bir değer olarak belirlenirse, adım uzunluğu büyük seçilmiş olur ve büyük adımlarla kayıp fonksiyonunun minimum noktasına ilerleme sağlanmış olur. Bu durumda model, yol üzerindeki yerel minimum noktalarından kaçınmış olur ancak bu sefer de global minimum noktasını atlayabilir veya adım uzunluğu nedeniyle bir türlü çukurun merkezine inemeyebilir. Model, minimum noktasının ilerisindeki ve gerisindeki iki nokta arasında epoch döngü sayısı tamamlanıncaya kadar sürekli git-gel yapabilir.

SGD (Stochastic Gradient Descent)

```
keras.optimizers.SGD(lr=0.01, momentum=0.0, decay=0.0, nesterov=False)
```

RMSprop

rmsprop optimizerı recurrent yapay sinir ağı modelleri için iyi bir seçimdir. Learning rate haricindeki diğer parametrelerin değiştirilmemesi tavsiye edilir.

```
keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=1e-08, decay=0.0)
```

adagrad

adagrad optimizerının parametrelerinin değiştirilmemesi tavsiye edilir.

```
keras.optimizers.Adagrad(lr=0.01, epsilon=1e-08, decay=0.0)
```

adadelta

Kemalcan Bora
Samet Çetin

adadelta optimizerının parametrelerinin değiştirilmemesi tavsiye edilir.

```
keras.optimizers.Adadelta(lr=1.0, rho=0.95, epsilon=1e-08, decay=0.0)
```

adam

```
keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-08, decay=0.0)
```

adamax

```
keras.optimizers.Adamax(lr=0.002, beta_1=0.9, beta_2=0.999, epsilon=1e-08, decay=0.0)
```

nadam

nadam optimizerının parametrelerinin değiştirilmemesi tavsiye edilir.

```
keras.optimizers.Nadam(lr=0.002, beta_1=0.9, beta_2=0.999, epsilon=1e-08,
                                                    schedule_decay=0.004)
```

Metrics

Metric yapısı fonksiyonun eğitim esnasındaki performansını takip etmemize yarar. Metric parametresinin ne olacağı compile esnasında belirlenir. Metric listesine yerleştirilen ve modelin performansını gösteren yapıların, eğitim esnasındaki her bir epoch döngüsü sonunda değerleri ekrana yansır. Böylece değişen hata değeri veya doğruluk değeri takip edilebilir.

```
# Kayıp fonksiyonu olarak mean_squared_error, optimizer olarak sgd belirlenmiş.
# Metric listesine mae (mean absolute error) ve acc (accuracy) değerleri # yerleştirilmiş.
model.compile(loss='mean_squared_error', optimizer='sgd', metrics=['mae', 'acc'])
```

Metric yapısı, kayıp fonksiyonuna benzer ve aslında kayıp fonksiyonunun döndürdüğü değerleri ya da o değerlerle hesaplanan farklı değerleri kullanır. Ancak modelin güncellenmesi esnasında kayıp fonksiyonu rol oynar. Metric sadece o andaki ve takip eden anlardaki kaybın gösteriminde rol oynar.

Keras kütüphanesinde farklı metric türleri bulunur. Bu metric türlerinden kısaca bahsedelim.

binary_accuracy

Kemalcan Bora
Samet Çetin

Model outputunun yalnızca iki değer alabildiği durumlarda, eğitim boyunca değişen doğruluğun takip edilebilmesini sağlar.

```
binary_accuracy(y_true, y_pred)
```

categorical_accuracy

Model outputunun ikiden fazla değer alabildiği durumlarda, eğitim boyunca değişen doğruluğun takip edilebilmesini sağlar.

```
categorical_accuracy(y_true, y_pred)
```

sparse_categorical_accuracy

```
sparse_categorical_accuracy(y_true, y_pred)
```

top_k_categorical_accuracy

```
top_k_categorical_accuracy(y_true, y_pred)
```

sparse_top_k_categorical_accuracy

```
sparse_top_k_categorical_accuracy(y_true, y_pred)
```

TRAINING

Modelin katmanları ve eğitim süresince kullanılacak fonksiyonlar belirlendikten sonra, eğitim süreci başlatılabilir. Eğitimi başlatmak için fit() fonksiyonu kullanılır.

```
fit(self, x, y, batch_size=32, epochs=10, verbose=1, callbacks=None,
validation_split=0.0, validation_data=None, shuffle=True, class_weight=None,
sample_weight=None, initial_epoch=0)
```

Kemalcan Bora
Samet Çetin

fit fonksiyonuna temel olarak, eğitmek istediğimiz input verilerimizi (**x**), input verilerimizin doğru ve gerçek outputları olan output verilerimizi (**y**), modelin eğitim boyunca kaç epoch döngüsü yapacağını belirtecek olan epoch döngü sayısını (**epochs**) ve eğitim esnasında sürecin durumuyla ilgili bilgi verecek olan **verbose** parametre değerleri girilir. Eğer veri kümesi alt kümeler olan batchlere ayrıldıysa, **batch_size** parametresini de kullanmak gerekir.

fit fonksiyonu, eğitim süreci boyunca değişen değerleri ve süreci kaydeder. Fonksiyon bir History nesnesi döndürür. Yani fonksiyon bir değişkene eşitlenerek kullanıldığında, daha sonra bu değişken üzerinden eğitim sürecinin farklı zamanlarındaki bilgilere erişilebilir. Hatta bu şekilde eğitim boyunca, kayıp fonksiyonunun döndürdüğü eğitim verilerinin hata değerleri ile modelin daha önceden karşılaşmadığı test verilerinin hata değerleri grafik haline getirilip, modelin eğitimi görselleştirilebilir. Bu grafikler yorumlanarak modelin overfitting yapıp yapmadığı ya da kaç epoch döngüsü sonrası overfitting yapmaya başladığı tespit edilebilir. Böylece model tekrar konfigüre edilip, daha doğru şekilde eğitilmesi sağlanabilir. Eğitim devam ederken test verilerinin hata değerlerinin de hesaplanabilmesi için fit fonksiyonuyla beraber validation_data parametresinin de kullanılması gerekir. Bu parametre validation_data=(x_test, y_test) şeklinde ayarlanarak kullanılır. x_test, test input verilerini ifade ederken, y_test test input verilerinin doğru ve gerçek output karşılıklarını ifade eder.

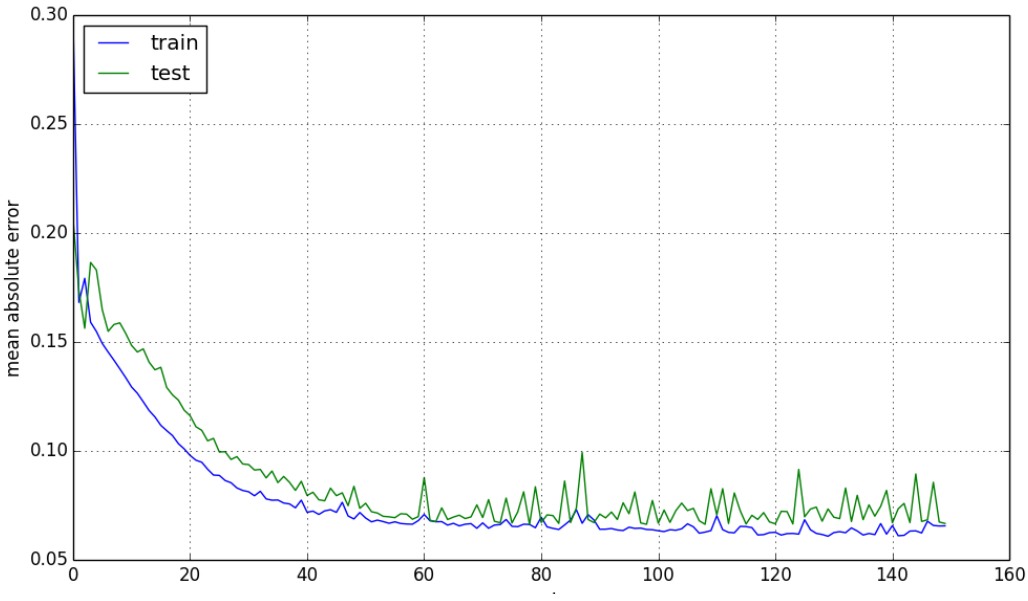
```
model = Sequential()
model.add(LSTM(100, input_shape=(1,3)))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam', metrics=['mae'])
history = model.fit(x_train, y_train, epochs=150, verbose=2,  validation_data=(x_test,
y_test))

# Eğitim veri inputlarının mean_absolute_error'ları çizdirilir.
plt.plot(history.history['mean_absolute_error'])
# Test veri inputlarının mean_absolute_error'ları çizdirilir.
plt.plot(history.history['val_mean_absolute_error'])

# Grafik eksenlerine isimleri verilir.
plt.xlabel('epochs')
plt.ylabel('mean_absolute_error')

plt.legend([train, 'test'], loc='upper_left')
plt.grid()
plt.show()
```

Kemalcan Bora
Samet Çetin

Yukarıda hazırlanan modelin sonucunda elde edilen grafik aşağıda incelenebilir. Yaklaşık ilk 80 epoch sonrası, test verilerinin hata değerlerinde bir azalma olmadığı gözlemlenebilir. Modelin epoch sayısının 150'den 80 civarına doğru çekilmesi, karşılaşılabilecek overfitting problemini önlemek adına faydalı olabilir.

```
Epoch 50/150
0s - loss: 0.0091 - mean_absolute_error: 0.0717 - val_loss: 0.0095 - val_mean_absolute_error: 0.0737
```

Eğer verbose değerini 0'a eşitlersek, model eğitilirken ekrana süreçle ilgili herhangi bir bilgi gelmeyecektir. Ancak verbose değerini 1 yaparsak aşağıdaki görselde görülebileceği gibi eğitimin kaçıncı epoch döngüsünde olduğu, anlık kayıp ve hata değerleri gibi bilgiler ekrana yansıyacaktır.



Kemalcan Bora
Samet Çetin

Eğer verbose değerini, son değer olan 2'ye eşitlersek, eğitimin kaçıncı epoch döngüsünde olduğu, anlık kayıp ve hata değerlerinin yanında bir de işlem çubuğu bilgisi ekrana yansır. Buradan yalnızca bir epoch döngüsünün ne kadarlık kısmının tamamlandığı takip edilebilir.

Verbose değerinin 0'dan farklı olacak şekilde kullanılması, bilgilerin ekrana basılması için de zaman harcanmasını gerektirir. Kısa işlemler için çok fark yaratmayacak bu zaman kaybı, çok uzun sürebilecek kompleks işlemler için fark yaratacak hale gelebilir. Bu yüzden verbose ayarlanırken bunun da hesaba katılması iyi olacaktır. Ancak verbose bilgilerinin, işe yarar bilgiler olduğu da unutulmamalıdır ve

```
Epoch 50/150
205/205 [==============================] - 0s - loss: 0.0091 - mean_absolute_error: 0.0717 - val_los
s: 0.0095 - val_mean_absolute_error: 0.0737
```

bu bilgiler eğitim esnasında yorumlanarak, belki de model inşa edilirken yapılmış basit hataların fark edilmesini sağlar ve zaman kaybetmeden eğitimin tekrar başlatılmasına neden olabilir.

EVALUATION

evaluate() fonksiyonu kullanılarak modele sokulan bir input veri kümesinin, kendi doğru ve gerçek output değerleri ile modelin hesapladığı output değerleri karşılaştırılarak, modelin doğruluğu değerlendirilebilir. evaluate fonksiyonu, eğitim bitişindeki kayıp fonkiyonunun hesapladığı kayıp değerini her zaman döndürür. Bunun yanında eğer varsa metric listesi içerisinde tanımlanmış diğer hata türlerini de döndürür.

```
model = Sequential()
model.add(LSTM(100, input_shape=(1,3)))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam', metrics=['mae'])
history = model.fit(x_train, y_train, epochs=80, verbose=2)

# Modelin daha önceden karşılaşmadığı test verilerini kullanarak modeli
# değerlendirelim.
scores = model.evaluate(x_test, y_test)

print(scores)
# Bu print fonksiyonunun sonucu, eğitimin bitişindeki kayıp değerini ve mae(mean
# absolute error) değerini verir.
#  [0.012582288421690464, 0.085272735357284551]
```

PREDICTION

Kemalcan Bora
Samet Çetin

predict() fonksiyonu kullanılarak herhangi bir input verisinin output değeri tahmin edilebilir. Model bu input verisini alarak önceden eğitilmiş ve son weight değerleri belirlenmiş modele yerleştirir ve output değerini döndürür. Hava durumu, borsa değerleri gibi geleceğe yönelik tahminler (forecasting) bu şekilde yapılmaya çalışılır. Ancak tahminler geleceğe dönelik olmak zorunda değildir. Sistemlerin doğruluğunu test etmek ya da başka amaçlar adına günümüz ve geçmiş zaman için de tahminler yapılabilir.

```
model = Sequential()
model.add(LSTM(100, input_shape=(1,3)))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam', metrics=['mae'])
model.fit(x_train, y_train, epochs=80, verbose=2)

predicted_output = model.predict(x)
```

YAPAY SİNİR AĞ ÇEŞİTLERİ

Kemalcan Bora
Samet Çetin

Farklı kullanımlara amaçlarına veya kullanım alanlarına göre birçok farklı yapay sinir ağı modeli tasarlanmış ve kullanılmaktadır. Bunlardan bir çoğunu, aşağıdaki görselde bulabilirsiniz.