Introduction

Derin öğrenme dünyasında yoğun katmanlardan çok daha fazlası var. Bir modele ekleyebileceğiniz düzinelerce katman türü var. (Bir örnek için Keras belgelerine göz atmayı deneyin!) Bazıları yoğun katmanlar gibidir ve nöronlar arasındaki bağlantıları tanımlarken, diğerleri ön işleme veya başka tür dönüşümler gerçekleştirebilir.

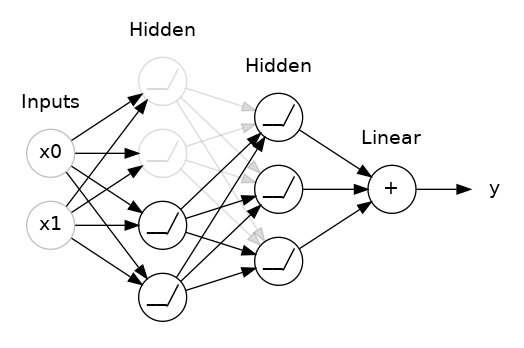
Bu derste, kendileri nöron içermeyen, ancak bazen bir modele çeşitli şekillerde fayda sağlayabilecek bazı işlevler ekleyen iki tür özel katmanı öğreneceğiz. Her ikisi de modern mimarilerde yaygın olarak kullanılır.

# Dropout[¶](https://www.kaggle.com/code/ryanholbrook/dropout-and-batch-normalization" \l "Dropout" \t "_self)

Bunlardan ilki, aşırı uyumu düzeltmeye yardımcı olabilecek "dropout katmanı"dır.

Son derste, aşırı uyumun, ağın eğitim verilerindeki sahte örüntüleri öğrenmesinden kaynaklandığından bahsetmiştik. Bu sahte örüntüleri tanımak için bir ağ genellikle çok belirli ağırlık kombinasyonlarına, bir tür "ağırlıklar komplosuna" güvenir. Çok spesifik oldukları için kırılgan olma eğilimindedirler: birini kaldırırsanız, komplo dağılır.

**Dropout**'un arkasındaki fikir budur. Bu komploları bozmak için, eğitimin her adımında bir katmanın girdi birimlerinin bir kısmını rastgele **devre dışı bırakırız**. Bu da ağın eğitim verilerindeki bu sahte örüntüleri öğrenmesini çok daha zor hale getirir. Bunun yerine, ağırlık örüntüleri daha sağlam olma eğiliminde olan geniş, genel örüntüleri araması gerekir.



Burada iki gizli katman arasına %50 dropout eklenmiştir.

Dropout'u bir tür ağ **topluluğu (ensemble)** oluşturmak olarak da düşünebilirsiniz. Tahminler artık tek bir büyük ağ tarafından değil, daha küçük ağlardan oluşan bir komite tarafından yapılacaktır. Komitedeki bireyler farklı türde hatalar yapma eğilimindedir, ancak aynı zamanda doğru da olurlar; bu da komiteyi bir bütün olarak herhangi bir bireyden daha iyi hale getirir. (Eğer karar ağaçlarının bir topluluğu olarak random forest'lara aşinaysanız, fikir aynıdır.)

## **Adding Dropout[¶](https://www.kaggle.com/code/ryanholbrook/dropout-and-batch-normalization" \l "Adding-Dropout" \t "_self)**

Keras'ta, bırakma oranı argümanı, giriş birimlerinin yüzde kaçının kapatılacağını tanımlar. Bırakma katmanını, bırakmanın uygulanmasını istediğiniz katmanın hemen önüne yerleştirin: (bırakma dropout)

keras.Sequential([

# ...

layers.Dropout(rate=0.3), # apply 30% dropout to the next layer

layers.Dense(16),

# ...

])

# Batch Normalization

Ele alacağımız bir sonraki özel katman, yavaş veya kararsız olan eğitimi düzeltmeye yardımcı olabilecek "toplu normalleştirme" (veya "batchnorm") işlemini gerçekleştirir.

Sinir ağlarında, verilerinizin tamamını ortak bir ölçekte tutmak genellikle iyi bir fikirdir; bunu scikit-learn'ün **StandardScaler** veya **MinMaxScaler** gibi araçlarıyla yapabilirsiniz. Bunun nedeni, SGD'nin ağ ağırlıklarını verilerin ürettiği aktivasyonun büyüklüğü ile orantılı olarak kaydırmasıdır. Çok farklı boyutlarda aktivasyon üretme eğiliminde olan özellikler, kararsız eğitim davranışlarına yol açabilir.

Şimdi, verileri ağa girmeden önce normalleştirmek iyi bir fikir ise, ağın içinde normalleştirmek de daha iyi olabilir! Aslında, bunu yapabilen özel bir katman türümüz var: **toplu normalleştirme katmanı (batch normalization layer)**. Bir toplu normalleştirme katmanı, gelen her bir batch'e bakar, önce batch'i kendi ortalaması ve standart sapmasıyla normalleştirir ve ardından verileri, iki adet eğitilebilir yeniden ölçekleme parametresiyle yeni bir ölçeğe koyar. Batchnorm, girdilerinin bir tür koordineli yeniden ölçeklendirmesini gerçekleştirir.

Çoğu zaman, batchnorm optimizasyon sürecine yardımcı olmak için eklenir (bazen tahmin performansına da yardımcı olabilir). Batchnorm içeren modeller, eğitimi tamamlamak için daha az epoch'a ihtiyaç duyma eğilimindedir. Dahası, batchnorm, eğitimin "tıkanmasına" neden olabilecek çeşitli sorunları da düzeltebilir. Özellikle eğitim sırasında sorun yaşıyorsanız, modellerinize toplu normalleştirme eklemeyi düşünün.

## **Adding Batch Normalization[¶](https://www.kaggle.com/code/ryanholbrook/dropout-and-batch-normalization" \l "Adding-Batch-Normalization" \t "_self)**

Toplu normalizasyonun ağdaki hemen hemen her noktada kullanılabileceği anlaşılıyor. Bunu bir katmandan sonra kullanabilirsiniz...

layers.Dense(16, activation='relu'),

layers.BatchNormalization(),

... veya bir katman ile onun aktivasyon fonksiyonu arasında:

layers.Dense(16),

layers.BatchNormalization(),

layers.Activation('relu'),

Ve eğer bunu ağınızın ilk katmanı olarak eklerseniz, Sci-Kit Learn'ün StandardScaler'ı gibi bir şeyin yerine geçen bir tür uyarlanabilir ön işlemci görevi görebilir.

# Example - Using Dropout and Batch Normalization[¶](https://www.kaggle.com/code/ryanholbrook/dropout-and-batch-normalization" \l "Example---Using-Dropout-and-Batch-Normalization" \t "_self)

Red Wine modelini geliştirmeye devam edelim. Şimdi kapasiteyi daha da artıracağız, ancak aşırı uyumu kontrol etmek için dropout ve optimizasyonu hızlandırmak için toplu normalleştirme ekleyeceğiz. Bu sefer, toplu normalleştirmenin eğitimi nasıl dengeleyebileceğini göstermek için verileri standartlaştırmayı da bırakacağız.

*# Setup plotting*

import matplotlib.pyplot as plt

plt.style.use('seaborn-whitegrid')

*# Set Matplotlib defaults*

plt.rc('figure', autolayout=True)

plt.rc('axes', labelweight='bold', labelsize='large',

titleweight='bold', titlesize=18, titlepad=10)

import pandas as pd

red\_wine = pd.read\_csv('../input/dl-course-data/red-wine.csv')

*# Create training and validation splits*

df\_train = red\_wine.sample(frac=0.7, random\_state=0)

df\_valid = red\_wine.drop(df\_train.index)

*# Split features and target*

X\_train = df\_train.drop('quality', axis=1)

X\_valid = df\_valid.drop('quality', axis=1)

y\_train = df\_train['quality']

y\_valid = df\_valid['quality']

Dropout eklerken Yoğun katmanlarınızdaki birim sayısını artırmanız gerekebilir.

from tensorflow import keras

from tensorflow.keras import layers

model = keras.Sequential([

layers.Dense(1024, activation='relu', input\_shape=[11]),

layers.Dropout(0.3),

layers.BatchNormalization(),

layers.Dense(1024, activation='relu'),

layers.Dropout(0.3),

layers.BatchNormalization(),

layers.Dense(1024, activation='relu'),

layers.Dropout(0.3),

layers.BatchNormalization(),

layers.Dense(1),

])

Bu sefer eğitimi nasıl kurduğumuz konusunda değiştireceğimiz hiçbir şey yok.

model.compile(

optimizer='adam',

loss='mae',

)

history = model.fit(

X\_train, y\_train,

validation\_data=(X\_valid, y\_valid),

batch\_size=256,

epochs=100,

verbose=0,

)

*# Show the learning curves*

history\_df = pd.DataFrame(history.history)

history\_df.loc[:, ['loss', 'val\_loss']].plot();

Verilerinizi eğitim için kullanmadan önce standartlaştırırsanız genellikle daha iyi performans elde edersiniz. Ancak ham verileri kullanabilmemiz, toplu normalizasyonun daha zorlu veri kümelerinde ne kadar etkili olabileceğini gösteriyor.