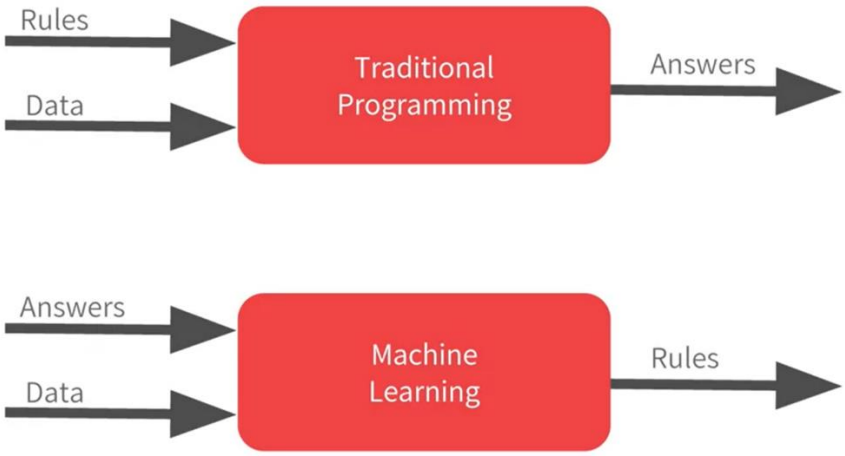
1. A primer in machine learning

Traditional programming aşağıdaki gibidir, kurallar ve data input olarak kullanılır, programlama ile istenen sonuçlar elde edilir. Burada kurlları oluşturan programımızdır.



Machine learning ise farklı bir yol izler, veriyi ve istenilen çıktıları programa veririz, böylece hangi kuralların gerektiğini program bulur.

Yani aslında traditional programming ile datayı answers'a bağlayan fonksiyonu biz elimizde yazıyoruz, şöyleyse şöyle olsun böyleyse bu olsun gibi. Ancak machine learning ile bu fonksiyonu ml algoritması buluyor biz sadece datayı ve ne istediğimizi belirtiyoruz bu amacı minimum cost ile yerine getirebilen algoritma bulunuyor.

2. The ‘Hello World’ of neural networks

Diyelim ki aşağıdaki X ve Y verilerini birbirine map eden bir fonksiyon bulmak istiyoruz, biraz bakarsak bu ilişkinin Y = 2X-1 olduğunu anlayabiliriz.

```
X = -1,  0, 1, 2, 3, 4
Y = -3, -1, 1, 3, 5, 7
```

Bu problemi basit bir Neural Network ile de çözebiliriz, bu basit bir linear regression problemi.

```
model = keras.Sequential([keras.layers.Dense(units=1, input_shape=[1])])
model.compile(optimizer='sgd', loss='mean_squared_error')

xs = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
ys = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0], dtype=float)

model.fit(xs, ys, epochs=500)

print(model.predict([10.0]))
```

Yukarıdaki satırda keras ile tek bir dense yani tek bir layer tanımlanmış, üstelik bu layer’da tek bir unit olacak ve tek bir input kullanılacak.

Sonraki satır ile modelimize bir optimizer ve loss function atadık. Ancak henüz modelimiz eğitilmedi. Tek bir unitten oluşan bir NN.

Sonraki adımda ise modelimize input’u ve output’u veriyoruz, böylece modelimizi her gradient step için o anki fonksiyonunu input’ları sokacak ve output hesaplayacak, hesapladığı output’ları da gerçek output’lar ile kıyaslayıp hipotezinin ne kadar yanıldığını yani cost’u hesaplayacak.

Daha sonra da bu cost’a göre parametrelerini güncelleyecek. Bu işlemi tüm dataset’I 500 kez kullanıncaya kadar devam edecek.

En sonunda umuyoruz ki modelimizi Y = 2X-1 gibi bir denklem elde etmiş olsun. Bunun için de model.predict ile X=10 değeri için Y’yi tahmin ettiriyoruz, sonucun 19 çıkmasını bekleriz.

Ancak sonuç tam olarak 19.0 çıkmaz 19'a çok yakın bir sayı çıkar. Peki bunun sebebi nedir?

Bence bunun sebebi NN'in öğrendiği line'ı iteratif olarak öğrenmesi, ortada analitik bir çözüm yok, tam aradığımız line'a oturması baya zor bir olasılık çünkü küçük küçük adımlarla bu eğim ve bias hesaplanıyor.

Ancak elbette eğer daha fazla data kullanırsak ve daha fazla iterasyon ile eğitimi yaparsak sonucun 19.0'a daha da yakın bir sayı çıkma ihtimalini artırırız.

3. Introduction to Computer Vision

Bu kısımda computer vision'a giriş yapmak için fashion_mnist dataset'I kullanılacak. Bu dataset yaklaşık 70k labeled fashion items verisinden oluşuyor.

```
fashion_mnist = keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
```

Fashion mnist'de yaklaşım 60k image train set için kalan 10k image'da test set için ayrılıyor.

Burada labeling integer sayılarla oluyor, elbette her bir integer label aslında kıyafet/ayakkabı/aksesuar gibi bir fashion item'e denk geliyor.

Dataset images'da küçültümüş, böylece eğitim daha kolay olacak, sanırım 24x24 px boyutlarında grayscale fotoğraflar kullanılıyor.

4. Computer Vision Neural Network

Geçen kısımda kullanılan model tek bir layer'dan ve tek bir unitten oluşuyordu, computer vision için burada kullanılacak yapı ise 3 layerdan oluşuyor.

```
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(10, activation=tf.nn.softmax)
])
```

Aslında ilk layer tam anlamıyla bir NN layer'ı değil, 28x28'lik inputu flatten etmek için kullanılan bir layer. Yani aslında bunu input layer gibi düşünebiliriz.

İkinci layer 128 unitli bir hidden layer ayrıca relu aktivasyonu da tanımlanmış.

Son layer ise 10 unitli outpur layer'ı ayrıca softmax activation'ı tanmımlanmış yani son layer'da bir classification yapılacak.

```
model.compile(optimizer = tf.optimizers.Adam(),
              loss = 'sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(training_images, training_labels, epochs=5)
```

5. Using Callbacks to Control Training

Training loop callbacks'i destekler. Böylece her epoch sonunda callback fonksiyonunu çağırarak, metrics'i check edebilirim eğer istenilen metriclere ulaştıysam training'i durdurabilirim.

Aşağıda fashion images'ı classify etmek için geçen kısımda kullanılan kod yer alıyor.

```
mnist = tf.keras.datasets.fashion_mnist
(training_images, training_labels), (test_images, test_labels) = mnist.load_data()
training_images=training_images/255.0
test_images=test_images/255.0
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(512, activation=tf.nn.relu),
  tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy')
model.fit(training_images, training_labels, epochs=5)
```

Özellikle model.fit() fonkisyonuna dikkat edelim.

Şimdi bir callback class'ı tanımlayalım:

```
class myCallback(tf.keras.callbacks.Callback):
  def on_epoch_end(self, epoch, logs={}):
    if(logs.get('loss')<0.4):
      print("\nLoss is low so cancelling training!")
      self.model.stop_training = True
```

Bu classback içerisinde tanımlanan on_epoch_end() methodu her epoch tamamlandığında callback tarafından çağıralacak!

Ayrıca bu model çağırılırken içine logs objesi parametre olarak girer ki bu parametrede training'in current state'i ile ilgili birçok bilgi tutulur!

Örneğin bu logs objesi üzerinden modelin o anki epoch'taki loss değerine yukarıdaki gibi ulaşabilirim ve bu değere bağlı bir koşul yazarım. Eğer loss'um 0.4'ün altına düşerse eğitim durdurulsun dedim.

Şimdi bu tanımlanan callback class'ını modelimiz için kullanabilmek için ana eğitim kodunda aşağıdaki gibi iki değişiklik yapıyoruz:

```
callbacks = myCallback()
mnist = tf.keras.datasets.fashion_mnist
(training_images, training_labels), (test_images, test_labels) = mnist.load_data()
training_images=training_images/255.0
test_images=test_images/255.0
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(512, activation=tf.nn.relu),
  tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy')
model.fit(training_images, training_labels, epochs=5, callbacks=[callbacks])
```

Öncelikle tanımlanan class'tan bir object instantiate ediyoruz, daha sonra da model.fit() içerisinde bu object'I parametre olarak veriyoruz, böylece her epoch sonunda on_epoch_end methodumuz çalışacak ve koşulumuz sağlanırsa eğitimi durduracaktır!

6. Implementing Convolutional Layers

Hemen yukarıda fashion mnist dataset'I için oluşturduğumuz modelin yapısını görüyoruz, bir flatten ve 2 dense layer'dan oluşuyor.

Şimdi ise bu modele convolutional layers ekleyelim:

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Conv2D(64, (3,3), activation='relu',
                         input_shape=(28, 28, 1)),
  tf.keras.layers.MaxPooling2D(2, 2),
  tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
  tf.keras.layers.MaxPooling2D(2,2),
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(10, activation='softmax')
])
```

Son üç layer'ın aynı kaldığına dikkat et, ancak bundan önce 2 adet 2D convolution ve pooling layers kullanılmış.

İlk convolutional layer input olarak 28x28x1 yapısında greyscale bir image'I alıyor, 3x3 boyutlarında 64 tane filtre uygulanıyor. Ayrıca relu activation kullanılıyor.

Sonuçta convolution derslerinden hatırlarsan padding'e ve stride'a göre 64 tane map elde ediliyor bu (X,X,64) boyutlarındaki aktivasyonlar da pooling'den sonra tekrar sıradaki convolutional layer'a alınıyor. Bu işlemlerin detayını zaten Andrew'in DL Specialization'ında görmüştük istersen aç bak.

Burada sanıyorum padding yok, stride 1 bu yüzden convolution'dan sonra her bir map'in boyu 2 px azalıyor. Ayrıca max pooling ile de maplerin boyutu yarıya düşüyor.

Bunu görebilmek için model.summary() methodu kullanılabilir.

```
model.summary()
```

Bu sayede aşağıdaki gibi modelimizin layer'larını layer için gerekli parametre sayısını ve layer'ın output shape'ini görebiliyoruz.

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_12 (Conv2D)           (None, 26, 26, 64)        640
_________________________________________________________________
max_pooling2d_12 (MaxPooling (None, 13, 13, 64)        0
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 11, 11, 64)        36928
_________________________________________________________________
max_pooling2d_13 (MaxPooling (None, 5, 5, 64)          0
_________________________________________________________________
flatten_5 (Flatten)          (None, 1600)              0
_________________________________________________________________
dense_10 (Dense)             (None, 128)               204928
_________________________________________________________________
dense_11 (Dense)             (None, 10)                1290
=================================================================
```

7. Understanding ImageGenerator

Önceki örneklerde kullanılan dataset 28x28 boyutlarında hazırlanmış, crop edilmiş, tespit edilecek objenin hep merkezde olduğu fotoğraflardan oluşuyordu.

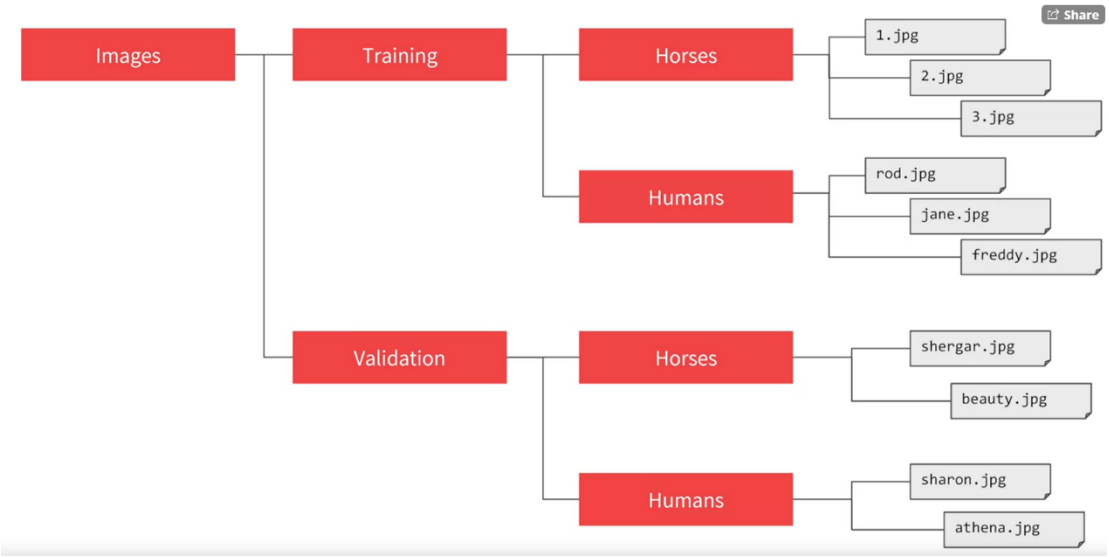
Ancak dataset her zaman böyle temiz olmaz, objeler fotoğrafın farklı kısımlarında yer alıyor olabilir ayrıca her fotoğraf farklı boyutlarda da olabilir, ve fotoğraf boyutları da 28x28'den büyük olabilir. Bu durumda ne yapılacak?

Ayrıca fashion dataset'inde tek satırla training ve validation set'ini ayırabiliyorduk, labelları hazır olarak geliyordu. Gerçekte durum bu olmayacak.

Bu kısımda ImageGenerator API'si ile bu problemlere bir çözüm bulunacak.

Aslında ImageGenerator API'si klasörleme ile ayrılmış bir dataset'i, otomatik olarak label eder ve istediğimiz boyutlara getirir.

Diyelim ki aşağıdaki gibi bir directory yapımız var yani Images klasörü içerisinde training ve validation klasörleri var onların içi de göründüğü gibi.



ImageGenerator ile bu klasörleme yapısında çok pratik bir şekilde istediğimiz boyutlarda otomatik olarak label edilmiş bir dataset elde edebiliriz.

```
from tensorflow.keras.preprocessing.image
import ImageDataGenerator
```

Yukarıdaki gibi class'I import ettikten sonra, bir ImageGenerator objesi instantiate edebiliriz, datayı normalize etmek için obje içine aşağıdaki gibi bir rescale pass edebiliriz.

```
train_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        train_dir,
        target_size=(300, 300),
        batch_size=128,
        class_mode='binary')
```

Daha sonra flow_from_directory methodu ile belirtilen train directory'sinden fotoğrafları istediğimiz size'da istediğimiz batch_size de load etmek için kullanabiliriz.

Burada belirtilecek directory Training klasörü olmalı, böylece bu klasör içindeki sub-directories isimleri labeling için kullanılacak.

Burada target_size'ın source file'I etkilemediğine dikkat et sadece image dataset olarak yüklenirken boyutu ayarlanıyor bu yüzden orjinal dataset'i bozmadan experimenting yapabiliriz.

Son olarak 2 farklı class olduğunu class_mode ile belirtiyoruz. Binary olmayan durumlar için ne yapılacağını ileride göreceğiz.

Sonuçta yukarıdaki şekilde train datası yaratılmış olacak aynı mantıkla sadece directory ismini değiştirerek validation set'i de yaratabiliriz.

```
test_datagen = ImageDataGenerator(rescale=1./255)

validation_generator = test_datagen.flow_from_directory(
        validation_dir,
        target_size=(300, 300),
        batch_size=32,
        class_mode='binary')
```

Burada yapılanların pure python olduğunu unutma, tensorflow'la falan alakası yok baya dosyalar alınıyor, boyutu değiştiriliyor saklanıyor falan gibi şeyler.

8. Defining a ConvNet to use complex images

Horses vs Humans classification için kullanılacak ConvNet fashion mnist için kullanılandan biraz daha farklı olacak ancak temel mantık aynı.

Gördüğümüz gibi yine conv ve pooling layers'I izleyen flatten ve dense layers'lardan oluşan bir sequential söz konusu.

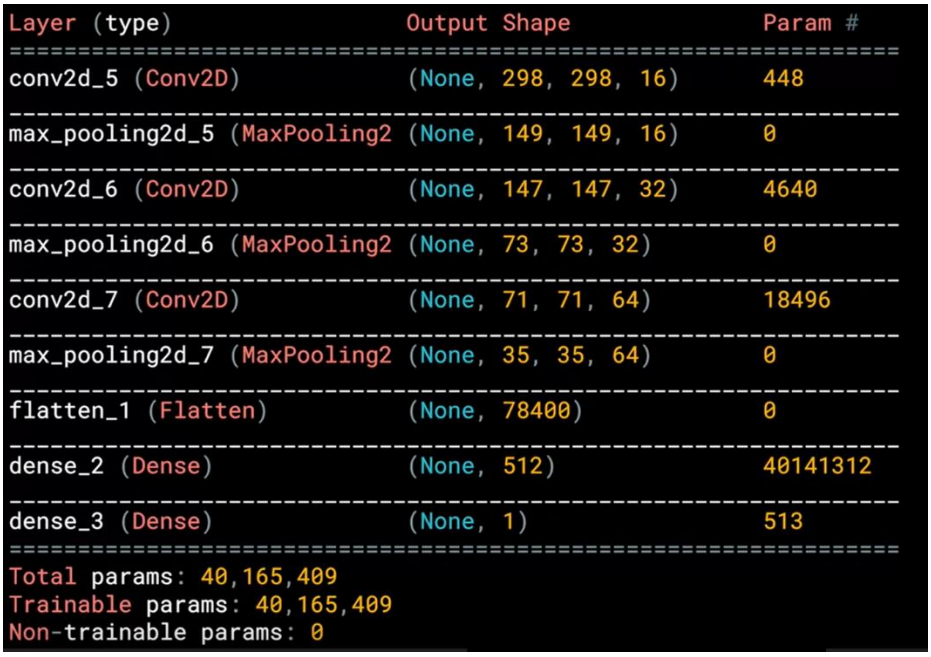
Öncekinde 2 convnet olmasına rağmen burada 3 convnet kullanılmış, bu higher complexity ve higher image size'ı reflect ediyor.

Image size 300x300x3 yani hem daha büyük bir image hem de colored bir image, 3 channellı. Sonuça convolutions ve pooling ile 35x35 boyutuna kadar küçültülüyor.

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(16, (3,3), activation='relu',
                           input_shape=(300, 300, 3)),
    tf.keras.layers.MaxPooling2D(2, 2),
    tf.keras.layers.Conv2D(32, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

Ayrıca output için 2 unitten oluşan bir softmax kullanmak yerine tek bir sigmoid unit kullandığımıza dikkat et, binary classification için böylesi daha verimli.

Model summary ile layerlarımızı ve output shape'leri görebiliriz:



```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_5 (Conv2D)            (None, 298, 298, 16)      448
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 149, 149, 16)      0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 147, 147, 32)      4640
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 73, 73, 32)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 71, 71, 64)        18496
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 35, 35, 64)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 78400)             0
_________________________________________________________________
dense_2 (Dense)              (None, 512)               40141312
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 513
=================================================================
Total params: 40,165,409
Trainable params: 40,165,409
Non-trainable params: 0
```

Sonuçta fully connected layersa 78400'lük bir input besliyoruz.

9 . Training the ConvNet with fit_generator

Geçen bölümde modeli oluşturmuştuk şimdi compile edeceğiz, bildiğimiz gibi bu noktada loss function'I ve optimizer'ı belirtiyorduk.

```
from tensorflow.keras.optimizers import RMSprop

model.compile(loss='binary_crossentropy',
              optimizer=RMSprop(lr=0.001),
              metrics=['acc'])
```

Önceden categorical cross entropy loss functionı kullanmıştık burada binary classification yaptığımz için yukarıdaki gibi binary crossentropy fonksiyonu kullanıldı.

Ayrıca optimizer olarsk Adam yerine RMSprop kullanılıyor, aslında Adam kullanmak sanki daha mantıklı ama learning rate adjust etmek için sanırım burada RMSprop kullanıyor.

Şimdi sırada modeli train etmek var:

```
history = model.fit_generator(
      train_generator,
      steps_per_epoch=8,
      epochs=15,
      validation_data=validation_generator,
      validation_steps=8,
      verbose=2)
```

Burada artık dataset yerine generator kullandığımız için model.fit yerine model.fit_generator kullanılıyor. Bu method içerisinde training ve validation generator'ları belirtiyoruz. Epoch belirtişmiş.

Generator içerisinde batch_size 128 olarak belirtişmişti, training set içerisinde 1024 image var bu da bir epoch için 8 batch anlamına geliyor o yüzden steps_per_epoch = 8 olarak belirtildi ki 1024 image'ın tamamı cover edilsin.

Benzer şekilde validation generator için de validation_steps 8 olarak belirtildi çünkü 256 validation images için batch_size = 32 olarak seçilmişti.

Son olarak verbose = 2 ile training sırasında ekranda ne kadar bilginin display edileceğini belirtiyoruz.

Böylece eğitimi tamamlayabiliriz.

Eğitimden sonra sıradaki adım eğitilen model ile yeni bir image'ı predict etmek, bunun için de aşağıdaki kod kullanılacak.

```
import numpy as np
from google.colab import files
from keras.preprocessing import image

uploaded = files.upload()

for fn in uploaded.keys():

  # predicting images
  path = '/content/' + fn
  img = image.load_img(path, target_size=(300, 300))
  x = image.img_to_array(img)
  x = np.expand_dims(x, axis=0)

  images = np.vstack([x])
  classes = model.predict(images, batch_size=10)
  print(classes[0])
  if classes[0]>0.5:
    print(fn + " is a human")
  else:
    print(fn + " is a horse")
```

Böylece colab'e buton ile bir file'ı yükleyip test edebileceğiz.