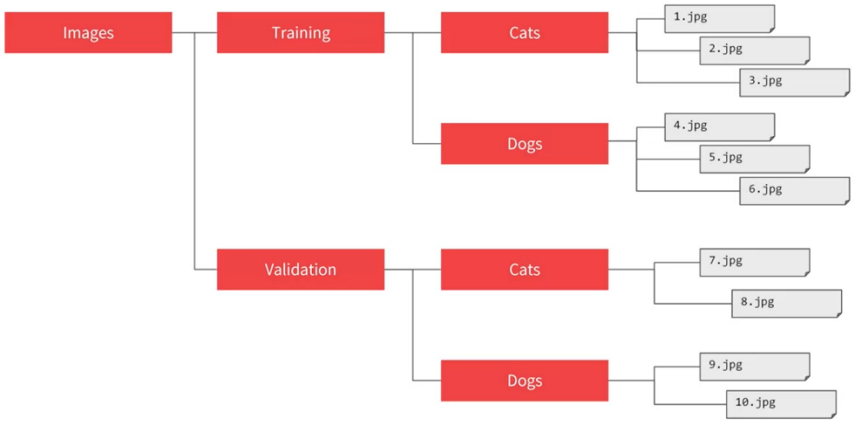
1. Cats vs Dogs Dataset Intro

Bu kısımda kaggle'ın cats vs dogs competition dataset'ini kullanacağız. İlk olarak datayı manipüle ederek training ve validation set'imizi elde edeceğiz bunun yanında bu dataset genel olarak oldukça temiz olsa da bazı problemlerle karşılacağız ve datayı daha temiz bir hale getirmemiz gerekecek.

Verileri aşağıdaki gibi bir dosya sisteminde tutarak, basitçe ImageDataGenerator ile training ve validation set'leri oluşturabileceğimizi biliyoruz:



Bunun için aşağıdaki gibi bir kod bloğu yeterli olcaktır, öncelikle datayı normalize etmek için rescale parametresini 1./255 olarak veriyorum. Ardından flow_from_driectory methodu ile training directory'ı belirtiyorum. Farklı boyutlardaki images'ı 150x150 boyutlarında kullanmak istediğimi, batch size'ımı ve classification mode'umu belirtiyorum.

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

train_generator = train_datagen.flow_from_directory(
        train_dir,
        target_size=(150, 150),
        batch_size=20,
        class_mode='binary')
```

Böylece training setimi hazırlamış oldum.

Validation set'imi hazırlamak içinde aynı mantığı kullanıyorum, sadece farklı bir directory ismi kullanacağım:

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

validation_generator = test_datagen.flow_from_directory(
        validation_dir,
        target_size=(150, 150),
        batch_size=20,
        class_mode='binary')
```

Dataset'imizi hazırladık, şimdi modelimizi kurabiliriz, daha önce yaptığımız gibi convolution ve pooling layer'ları takip eden bir fully connected layer ile, ve en sonda kullanılacak bir sigmoid unit ile classification yapılabilir.

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(16, (3,3), activation='relu',
                           input_shape=(150, 150, 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')
])
```

150x150x3 images ile çalıştığımız için input shape'i belirtiyoruz, ilk convolutional layer için 3x3'lük 16 filtre kullanıyoruz.

Aşağıda da model summary'sini görebiliriz:

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 148, 148, 16)      448
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 74, 74, 16)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 72, 72, 32)        4640
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 36, 36, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 34, 34, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 17, 17, 64)        0
_________________________________________________________________
flatten (Flatten)            (None, 18496)             0
_________________________________________________________________
dense (Dense)                (None, 512)               9470464
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 513
=================================================================
Total params: 9,494,561
Trainable params: 9,494,561
Non-trainable params: 0
```

Şimdi hazırladığımız modeli eğitirken kullanacağımız loss function'ı, optimizer'ı ve takip etmek istediğimiz metrics'i belirtiyoruz, modeli compile ediyoruz.

```
from tensorflow.keras.optimizers import RMSprop

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

Son olarak, modelimizi fit ediyoruz, bunun için aşağıdaki gibi training ve validation set'I veriyoruz, bunun yanında steps_per_epoch parametreleri dataset'in içindeki veri sayısı/batch size ile hesaplanıyor.

```
history = model.fit_generator(
      train_generator,
      steps_per_epoch=100,
      epochs=15,
      validation_data=validation_generator,
      validation_steps=50,
      verbose=2)
```

Bu methodla oluşturduğumuz cat-dog classifier'ı colab dosyaları arasında bulabilirsin.

2. Data Augmentation: A Technique to Avoid Overfitting

Image Augmentation is a very simple, but very powerful tool to help you avoid overfitting your data.

Temelde, eğitim yapılırken alınan batch'ler üzerinde bazı işlemler yapılarak data augmentation elde ediliyor, bu sayede aslında her epoch'da sanki farklı bir dataset üzerinde çalışmışız gibi oluyor, o yüzden overfitting'den kurtulmaya yarıyor.

Zaten bunu önceden de biliyoruz.

Diyelim ki ayakkabı tanıyan bir model geliştiriyoruz, aşağıdaki fotoğrafları input olarak verdik,

Sonrasında model aşağıdaki gibi bir ayakkabı fotoğrafını büyük ihtimalle tanıyacaktır:

Buna karşın aşağıdaki gibi bir ayakkabı fotoğrafı için model büyük ihtimalle yanlış tahmin yapacaktır, çünkü ayakkabıyı bilmeyen biri için, aşağıdaki ayakkabının yukarıdakilerle bir alakası yok.

İşte biliyoruz ki bu durum overfitting oluyor.

Elimizde limited data var ise overfitting ihtimali çok daha fazla oluyor. Yani bu doğal bir şey çünkü limited bir dataset ile, modelin daha sonra göreceği tüm ayakkabıları cover etmek imkansızdır, mükemmel bir model için infinite dataset lazım. Bu yüzden en güzeli kendini tekrar etmeyen olabildiğince geniş bir spektrumdan alınmış ve tüm ayakkabıları iyi yansıtan bir sample dataset ile çalışmak.

İşte Augmentation ile yapacağımız şey, küçük limited datasetimizi alıp, manipüle ederek, daha geniş bir aralığı cover eden daha büyük bir dataset haline getirmek, böylece modelim elimizdeki dataya overfit etmesini engellemiş oluyoruz.

Şimdi bu data augmentation işini nasıl kodladığımıza bakalım, daha önce gördüğümüz ImageDataGenerator ile augmentation işini yapabiliriz. Aslına bakarsan, daha önce zaten küçük bir image augmentation yaptık, bu da rescale işlemi idi, bu işlem ile, eğitim sırasında alınan her image'ı normalize etmiştik.

Şimdi ise daha detaylı augmentation işlemlerine bakalım, aşağıda bir çok farklı ayar görülüyor:

```
# Updated to do image augmentation
train_datagen = ImageDataGenerator(
      rescale=1./255,
      rotation_range=40,
      width_shift_range=0.2,
      height_shift_range=0.2,
      shear_range=0.2,
      zoom_range=0.2,
      horizontal_flip=True,
      fill_mode='nearest')
```

Rotation range'i 0-180 derece arasında ayaralayabilir. Shifting ile image'I frame içerisinde kaydırabiliriz, çoğu image subject centered olduğu için subject centered duruma overfit etmeyelim diye shifting'I kullanabiliriz. 0.2 demek image'ı %20 random olarak shift et demek.

Shearing ile, aslında 3D rotation yapmış gibi oluyoruz, böylece farklı açılardan çekimlere karşı yeni datasetler oluşturabiliyoruz:

Örneğin modelimiz sadece soldaki gibi insanlarla eğitildiyse sağdaki insanı tanımaması muhtemel, çünkü insan farklı bir açıdan fotoğraflanmış.

Ancak shearing uygularsak, training datamız aşağıdaki hale gelir ve aslında istediğimize gayet benzer bir poz elde etmiş oluruz.

Benzer şekilde, zooming ile farklı uzaklıktaki insanları tanımaya yardımcı olabiliriz, yani hep uzaktan çekilmiş insan fotoğrafları kullanmak, modelin 2 bacağa overfit etmesine neden olabilir bu yüzden de yakın çekim bir insanı modelimiz tanımayabilir, bu amaçla zoom ayarı da önemli.

Ayrıca bazı fotoğrafları aynalamak da modelimiz için yeni datalar üretmek anlamına gelir.

Son olarak fill mode = true ile yukarıdaki operasyonlar sırasında kaybolan pixelleri doldurmaya yarar, nearest seçeneği ile kaybolan pixelleri komşu pixelleri ile doldururuz.

Şimdi yukarıdaki augmentation'I cats vs dogs'a uygularsak, göreceğiz ki normalde 0.99 ve 0.70 civarı olan training accuracy ve validation accuracy birbirine yaklaşacak ve 0.86 ve 0.81 civarı değerler alacaktır, yani augmentation sayesinde training set'i daha geniş tuttuk ve validation seti ve hopefully real life datayı daha iyi yansıtan bir training set elde etmiş olduk.

Buna karşın augmentation'ın ters teptiği şöyle bir durum da olabilir, diyelim ki hem training hem de validation set subject centered, uzak çekim insan fotoğraflarından oluşuyor, eğer ki ben bu dataya augmentation uygularsam, training set aslında real dataya daha yakın bir hal alacak buna karşın validation set aslında real datayı temsil etmeyecek çünkü hep uzak çekim ve subject centered images'dan oluşacak. Bu yüzden augmented data ile eğitilen model validation set üzerinde iyi performans veremeyecek.

Yani augmentation'ı sadece training için değil validation veya testing için de uygulamak isteriz ki sonuçlarımız tutarlı olsun.

3. Transfer Learning

Daha önceden de bildiğimiz gibi, eğitilmiş bir modeli parametreli olarak alıp, kendi datasetimiz üzerinde tekrar biraz eğiterek, çok başarılı modeller elde edebiliriz.

Temelde bizim sınırlı datamızla eğitilen model, problem için ayırt edici olan tüm featureları öğrenemiyor, çünkü bunun için yeterli veriyi modele sağlayamıyoruz, ancak milyonlarca image üzerinde pre-trained edilmiş bir modeli parametreleri ile alırsak bu model halihazırda bir çok ayırt edici feature'ı detect edebilecek şekilde eğitilmiş oluyor, bu model üzerinde kısa bir kalibrasyon benzeri eğitimle, modeli kendi problemimize dair detaylı feature'ları extract edebilecek şekilde eğitmemiz mümkün hale geliyor.

Genelde transfer learning ile alınan modelin tüm convolutional layers'ı lock edilir ve kalan dense layers tekrar eğitilir, ancak her zaman böyle olacak diye bir koşul yok, bazen sonlara yakın convolutional layers bizim uygulamamıza zarar verecek kadar complex feature'ları extract ediyor olabilir, bu durumda daha dar bir alanı lock ederiz ve sona yakın convolutional layers ile dense layers'ı tekrar train ederek, önceden eğitilmiş modeli kendi uygulamamız için hazır hale getiririz.

Biz bu kısımda örnek olarak IMAGENET üzerinde 1.4 million images üzerinde 1000 class için eğitilmiş bir inception model'i kullanacağız.

Şimdi kodlama kısmına bakalım, transfer learning için keras layers API kullanılacak. Bu API sayesinde pretrained modelin tekrar eğitmek istediğimiz layer'ları seçebileceğiz.

```
import os

from tensorflow.keras import layers
from tensorflow.keras import Model
```

Kullanılacak pretrained inception modelinin parametreleri aşağıdaki linkte tutuluyor.

```
https://storage.googleapis.com/mledu-datasets/
 inception_v3_weights_tf_dim_ordering_tf_kernels
```

Bu linkten inception için parametreleri indirdikten sonra yapacağımız şey, keras ile sıfırdan bir inception model oluşturmak, bu süreçte aşağıdaki gibi input shape'i de belirtiriz.

```
from tensorflow.keras.applications.inception_v3 import InceptionV3

local_weights_file = '/tmp/inception_v3_weights_tf_dim_ordering_tf_kernels_notop.h5'

pre_trained_model = InceptionV3(input_shape = (150, 150, 3),
                                include_top = False,
                                weights = None)

pre_trained_model.load_weights(local_weights_file)
```

Ancak built-in weights'i kullanmaması için kerası uyarırız ve daha sonra load_weights komutu ile indirdiğimiz weightleri oluşturulan modele inject ederiz.

Include_top = false satırı ile inceptionv3 modelinin üstünde bulunan fully connected layer'ı ignore ederiz yani sanıyorum, modelin fully connected kısmını a.k.a kafa kısmını kesiyoruz, sadece convolutional layers kısmını alıyoruz, sonuçta eğitimden önce bu modele kendi problememize göre bir fully connected kafa eklememiz gerekecek.

Şimdide modelimin layers'ı üzerinde iterate edip layerları lock edebilirim. Böylece artık lock edilen layerlar trainable olmayacaktır.

```
for layer in pre_trained_model.layers:
  layer.trainable = False
```

Bu hali ile modelin summary'sine bakarsak çok büyük olduğunu göreceğiz.

Summary ile modele bakarsak her layer'ın bir ismi olduğunu göreceğiz, son layerlar 3x3 maplarden oluşuyor biz ise sonlara yakın olan mixed7 ismindeki layer'ın 7x7 maplerden oluşan outputunu alacağız.

```
last_layer = pre_trained_model.get_layer('mixed7')

last_output = last_layer.output
```

Inception modelinin mixed7 layer'ının outputunu yukarıda elde ettik, şimdi ise bu mixed7'ın outputunu kendine input olarak alan bir fully connected layer yaratacağız:

```
from tensorflow.keras.optimizers import RMSprop

x = layers.Flatten()(last_output)
x = layers.Dense(1024, activation='relu')(x)
x = layers.Dense  (1, activation='sigmoid')(x)

model = Model( pre_trained_model.input, x)
model.compile(optimizer = RMSprop(lr=0.0001),
              loss = 'binary_crossentropy',
              metrics = ['acc'])
```

Öncelikle, output flatten ediliyor ve fully connected layerımız için bir input'a çevriliyor, daha sonra 1024'lük relu activation'lı dense hidden layer ile, tek bir sigmoid unitli output layer ekleniyor.

Sonuçta anladığım kadarı ile inception layer'I mixed7 layerından kesip, yeerine bir fully connected layer ekledik ve model içine bu combine edilen yeni modeli atadık, bundan sonrası aynı eskisi gibi modeli compile etmek ve train etmek, modelin kafa kısmından öncesi zaten pretrained olduğu için, daha hızlı bir şekilde daha iyi performans elde etmeyi bekleriz.

Yine daha önce olduğu gibi training süresince data augmentation yapmak işimize yarayacaktır:

```
# Add our data-augmentation parameters to ImageDataGenerator
train_datagen = ImageDataGenerator(rescale = 1./255.,
                                   rotation_range = 40,
                                   width_shift_range = 0.2,
                                   height_shift_range = 0.2,
                                   shear_range = 0.2,
                                   zoom_range = 0.2,
                                   horizontal_flip = True)
```

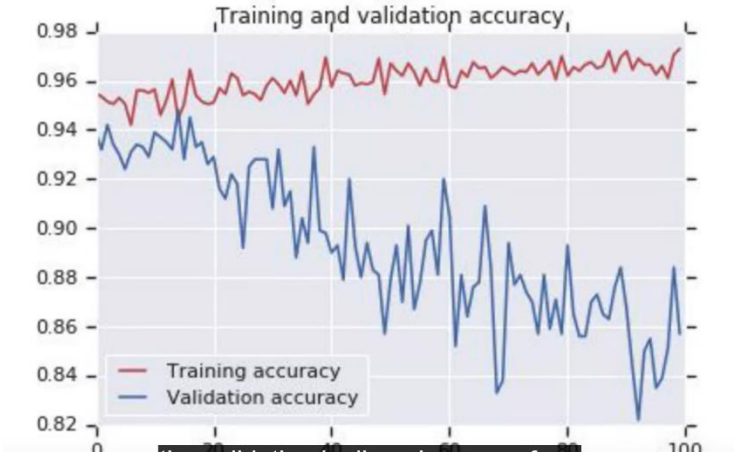
Yine eskisi gibi training datayı oluşturmak için generator'ı kullanırız:

```
train_generator = train_datagen.flow_from_directory(
                    train_dir,
                    batch_size = 20,
                    class_mode = 'binary',
                    target_size = (150, 150))
```

Son olarak yine eskisi gibi eğitimi gerçekleşirebiliriz:

```
history = model.fit_generator(
            train_generator,
            validation_data = validation_generator,
            steps_per_epoch = 100,
            epochs = 100,
            validation_steps = 50,
            verbose = 2)
```

Eğitim sonunda aşağıdaki gibi bir graph ile karşılaştık:



Burada bariz bir overfitting var ama eskisi gibi önce ikisi beraber artan sonra validation accuracy sabit kalan ve training accuracy artan bir graph yerine, önce iyi başlayan sonra zamanla düşen bir bir validation accuracy görüyoruz.

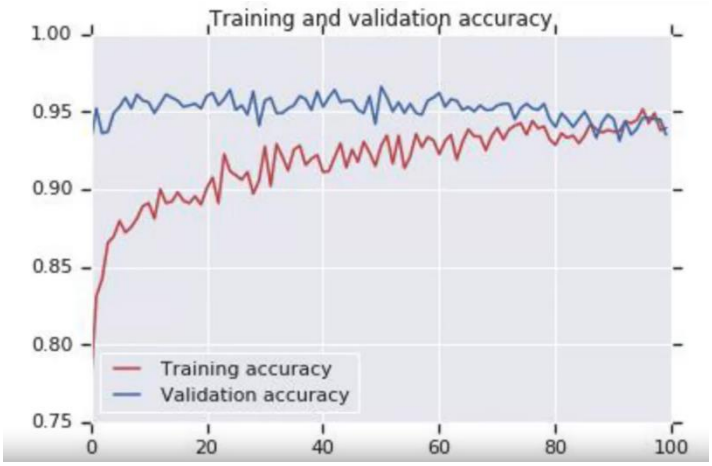
Bu overfitting'den kurtulmak için augmentation'ın yanında regularization da kullanacağız. Aşağıdaki gibi modelimize bir dropout layer ekleyebiliriz.

```
from tensorflow.keras.optimizers import RMSprop

x = layers.Flatten()(last_output)
x = layers.Dense(1024, activation='relu')(x)
x = layers.Dropout(0.2)(x)
x = layers.Dense  (1, activation='sigmoid')(x)

model = Model( pre_trained_model.input, x)
model.compile(optimizer = RMSprop(lr=0.0001),
              loss = 'binary_crossentropy',
              metrics = ['acc'])
```

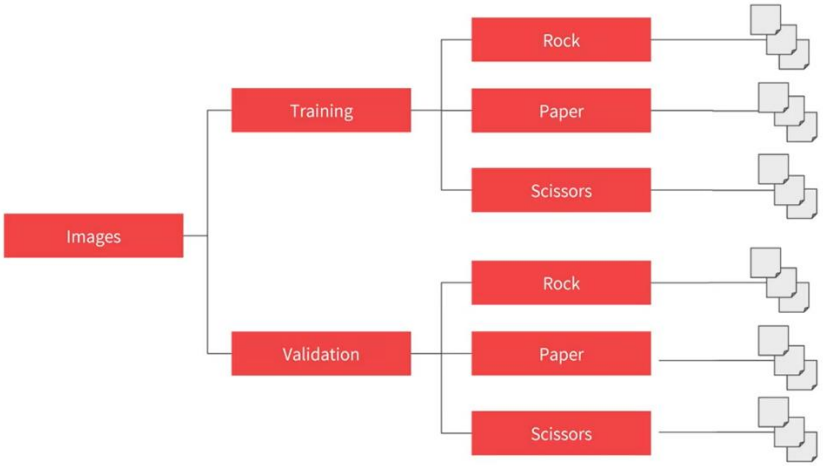
Dropout sonrası accuracy graph aşağıdaki hale geliyor, inanılmaz belirgin bir iyileşme söz konusu.



4. Multiclass Classification

Şimdiye kadar yaptığımız classification problemleri (horse vs human – cat vs dog) binary classification problems idi. Bu kısımda daha fazla class içeren yani multiclass classification problemlerini nasıl çözeceğimize bakacağız.

Örneğin rock paper scissors classifier yapmak istersek, 3 farklı classımız olacak. Biliyoruz ki daha önce labeling için imagedatagenerator'ı kullanmıştık, klasör ismine göre labeling yapılmıştı. E şimdi de aynı mantığı kullanabiliriz, sadece 3 farklı klasör kullanacağız:



Şimdi training directory'den training set'i oluştururken, class_mode'u binary değil categorical olarak değiştireceğiz, çünkü artık multiclass classification yapıyoruz.

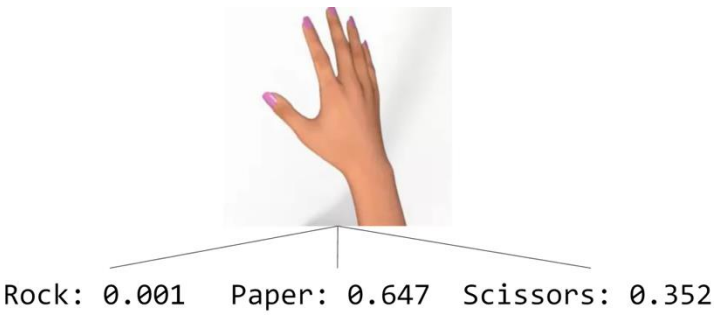
```
train_datagen = ImageDataGenerator(rescale=1./255)

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

Şimdi de modeli oluştururken, son layer'a tek bir sigmoid unit atmak yerine 3 unitli bir softmax layer atıyoruz.

```
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(3, activation='softmax')
])
```

Sonuçta aşağıdaki gibi bir fotoğrafa modelin vereceği output fotoğrafın altındaki gibi olacak, toplamları 1 eden olasılıklar.



Son olarak modeli compile ederken binary crossentropy yerine categorical crossentropy loss function'ını kullanacağız.

```
from tensorflow.keras.optimizers import RMSprop

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