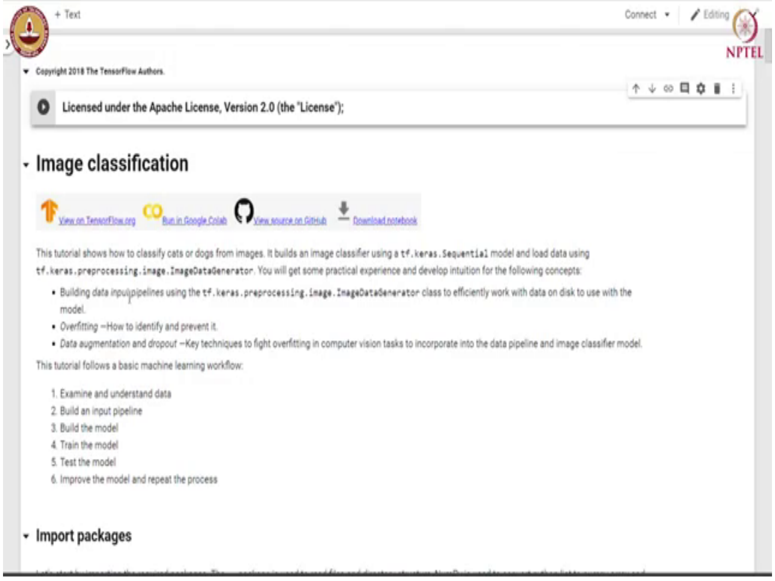
Practical Machine Learning

Dr. Ashish Tendulkar
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture - 24
Image Classification and Visualization

(Refer Slide Time: 00:13)



Copyright 2018 The TensorFlow Authors.

Licensed under the Apache License, Version 2.0 (the "License");

Image classification

This tutorial shows how to classify cats or dogs from images. It builds an image classifier using a `tf.keras.Sequential` model and load data using `tf.keras.preprocessing.image.ImageDataGenerator`. You will get some practical experience and develop intuition for the following concepts:

- Building data input pipelines using the `tf.keras.preprocessing.image.ImageDataGenerator` class to efficiently work with data on disk to use with the model.
- *Overfitting* —How to identify and prevent it.
- *Data augmentation and dropout* —Key techniques to fight overfitting in computer vision tasks to incorporate into the data pipeline and image classifier model.

This tutorial follows a basic machine learning workflow:

1. Examine and understand data
2. Build an input pipeline
3. Build the model
4. Train the model
5. Test the model
6. Improve the model and repeat the process

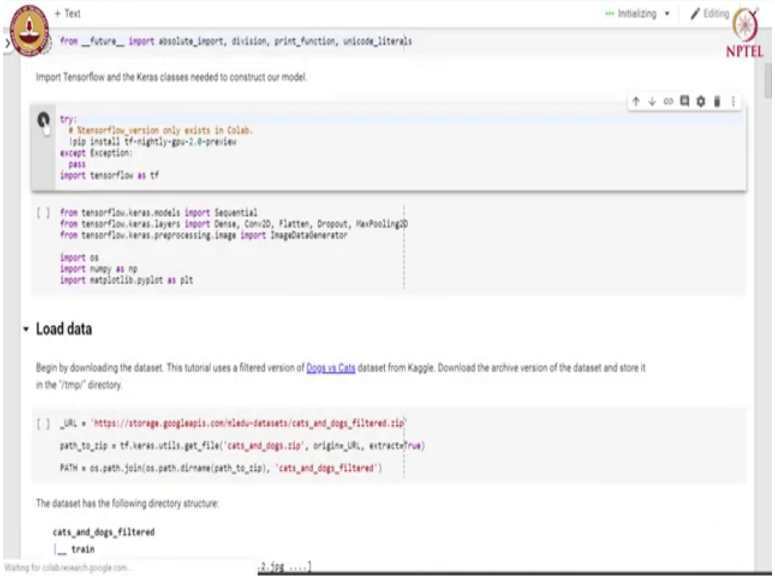
Import packages

In the previous session, we studied CNNs we also learnt how to build CNN models with transfer learning. In this session, we will build Image Classification models from scratch and we will use bunch of strategies that are employed in practice while building image classification model and we will also visualize what the CNN is learning by looking at the activations after each layer.

So, we will follow a basic machine learning work flow where we will examine and understand the data. We will build the input pipeline to bring the data to the training. We will build the model, they train it, test it and then will improve the performance of the model and repeat the process.

We will get some practical experience and develop intuitions for building input pipelines for images using image data generator class. We will also study how to identify over fitting and prevent it and we will also learn key concepts like data augmentation and dropout.

(Refer Slide Time: 01:50)



```
from __future__ import absolute_import, division, print_function, unicode_literals
```

Import Tensorflow and the Keras classes needed to construct our model.

```
try:
  # %tensorflow_version only exists in Colab.
  !pip install tf-nightly-gpu-2.0-preview
except Exception:
  pass
import tensorflow as tf
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Flatten, Dropout, MaxPooling2D
from tensorflow.keras.preprocessing.image import ImageDataGenerator

import os
import numpy as np
import matplotlib.pyplot as plt
```

Load data

Begin by downloading the dataset. This tutorial uses a filtered version of Dogs vs Cats dataset from Kaggle. Download the archive version of the dataset and store it in the "/tmp/" directory.

```
_URL = 'https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtered.zip'

path_to_zip = tf.keras.utils.get_file('cats_and_dogs.zip', origin=_URL, extract=True)

PATH = os.path.join(os.path.dirname(path_to_zip), 'cats_and_dogs_filtered')
```

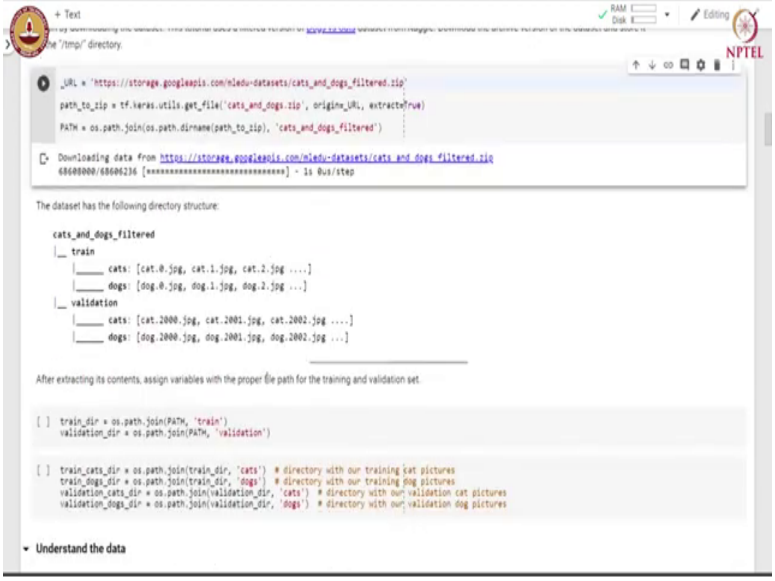
The dataset has the following directory structure:

```
cats_and_dogs_filtered
|__ train
```

Waiting for colab.research.google.com...

Let us install TensorFlow 2.0. Let us import image data generator and other libraries like dense convolution 2D flattened Dropout and MaxPooling 2D from Keras layers and we will also import sequential for building model. We use matplotlib.pyplot for plotting the performance of the model. We use dogs versus cats dataset from Kaggle competition.

(Refer Slide Time: 02:36)



the "/tmp/" directory.

```
_URL = 'https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtered.zip'

path_to_zip = tf.keras.utils.get_file('cats_and_dogs.zip', origin=_URL, extract=True)

PATH = os.path.join(os.path.dirname(path_to_zip), 'cats_and_dogs_filtered')
```

```
Downloading data from https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtered.zip
68608000/68606236 [==============================] - 1s 0us/step
```

The dataset has the following directory structure:

```
cats_and_dogs_filtered
|__ train
    |______ cats: [cat.0.jpg, cat.1.jpg, cat.2.jpg ....]
    |______ dogs: [dog.0.jpg, dog.1.jpg, dog.2.jpg ...]
|__ validation
    |______ cats: [cat.2000.jpg, cat.2001.jpg, cat.2002.jpg ....]
    |______ dogs: [dog.2000.jpg, dog.2001.jpg, dog.2002.jpg ...]
```

After extracting its contents, assign variables with the proper file path for the training and validation set.

```
train_dir = os.path.join(PATH, 'train')
validation_dir = os.path.join(PATH, 'validation')
```

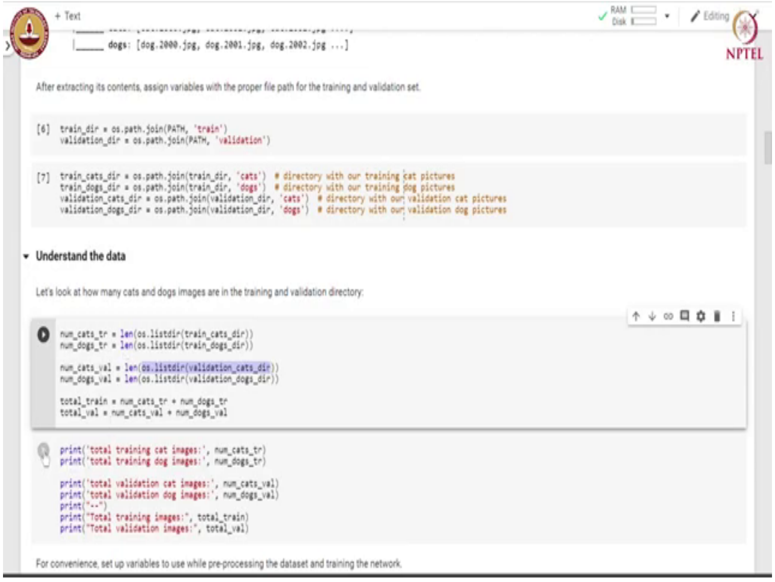
```
train_cats_dir = os.path.join(train_dir, 'cats')  # directory with our training cat pictures
train_dogs_dir = os.path.join(train_dir, 'dogs')  # directory with our training dog pictures
validation_cats_dir = os.path.join(validation_dir, 'cats')  # directory with our validation cat pictures
validation_dogs_dir = os.path.join(validation_dir, 'dogs')  # directory with our validation dog pictures
```

Understand the data

The dataset as the following structure. There is a top level directory called cats and dogs filtered, then we have training and validation dataset. Within training dataset, there are two sub directories cats and dogs, the validation directory structure also follows the same. The validation also has two sub directories cats and dogs. Within cats directory we have images of the cats stored in jpeg format and each file has a name cat.id.jpeg and dog.id.jpeg.

So, first 2000 examples are used as training and the remaining examples are used for validation. After extracting the content, we assign variables with proper file paths for training and validation sets.

(Refer Slide Time: 03:46)



```
    |_____ dogs: [dog.2000.jpg, dog.2001.jpg, dog.2002.jpg ...]
```

After extracting its contents, assign variables with the proper file path for the training and validation set.

```
[6] train_dir = os.path.join(PATH, 'train')
    validation_dir = os.path.join(PATH, 'validation')
```

```
[7] train_cats_dir = os.path.join(train_dir, 'cats')  # directory with our training cat pictures
    train_dogs_dir = os.path.join(train_dir, 'dogs')  # directory with our training dog pictures
    validation_cats_dir = os.path.join(validation_dir, 'cats')  # directory with our validation cat pictures
    validation_dogs_dir = os.path.join(validation_dir, 'dogs')  # directory with our validation dog pictures
```

Understand the data

Let's look at how many cats and dogs images are in the training and validation directory:

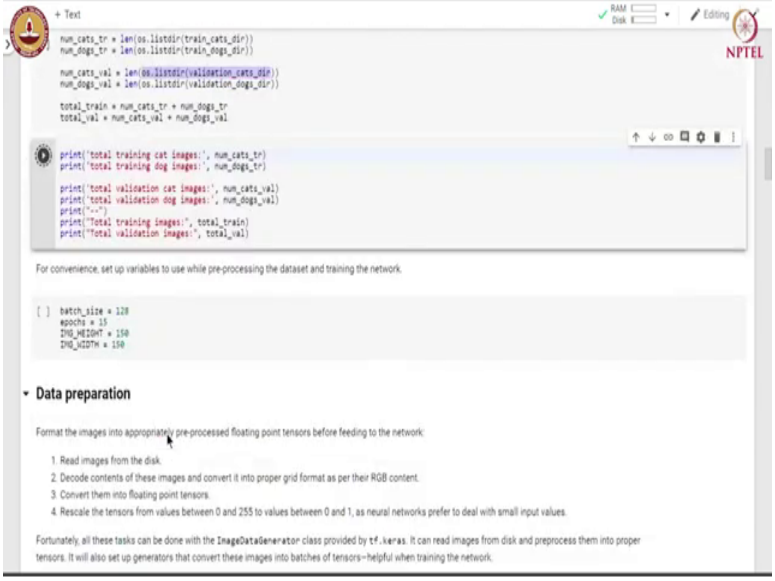
```
num_cats_tr = len(os.listdir(train_cats_dir))
num_dogs_tr = len(os.listdir(train_dogs_dir))

num_cats_val = len(os.listdir(validation_cats_dir))
num_dogs_val = len(os.listdir(validation_dogs_dir))

total_train = num_cats_tr + num_dogs_tr
total_val = num_cats_val + num_dogs_val
```

```
print('total training cat images:', num_cats_tr)
print('total training dog images:', num_dogs_tr)

print('total validation cat images:', num_cats_val)
print('total validation dog images:', num_dogs_val)
print("--")
print("Total training images:", total_train)
print("Total validation images:", total_val)
```

For convenience, set up variables to use while pre-processing the dataset and training the network.

Then we construct paths for training and validation directories for cats and dogs. Let us look at how many cats and dog images are there in training and validation directory. We use os.listdir command to list the content of the directory and take the length of this directory listing to calculate the number of cats and dogs in training and validation.

(Refer Slide Time: 04:21)



```
num_cats_tr = len(os.listdir(train_cats_dir))
num_dogs_tr = len(os.listdir(train_dogs_dir))

num_cats_val = len(os.listdir(validation_cats_dir))
num_dogs_val = len(os.listdir(validation_dogs_dir))

total_train = num_cats_tr + num_dogs_tr
total_val = num_cats_val + num_dogs_val
```

```
print('total training cat images:', num_cats_tr)
print('total training dog images:', num_dogs_tr)

print('total validation cat images:', num_cats_val)
print('total validation dog images:', num_dogs_val)
print("--")
print("Total training images:", total_train)
print("Total validation images:", total_val)
```

For convenience, set up variables to use while pre-processing the dataset and training the network.

```
[ ] batch_size = 128
    epochs = 15
    IMG_HEIGHT = 150
    IMG_WIDTH = 150
```

Data preparation

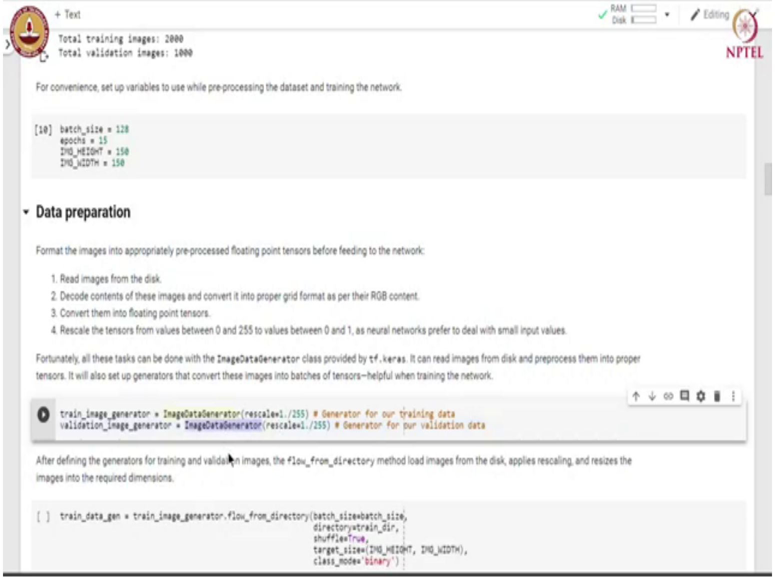
Format the images into appropriately pre-processed floating point tensors before feeding to the network:

1. Read images from the disk.
2. Decode contents of these images and convert it into proper grid format as per their RGB content.
3. Convert them into floating point tensors.
4. Rescale the tensors from values between 0 and 255 to values between 0 and 1, as neural networks prefer to deal with small input values.

Fortunately, all these tasks can be done with the `ImageDataGenerator` class provided by `tf.keras`. It can read images from disk and preprocess them into proper tensors. It will also set up generators that convert these images into batches of tensors—helpful when training the network.

You can see that we are using 2000 images for training and 1000 images total for validation. 1000 cat images are used for training and 500 cat images are used for validation. The same proportion of images are used for training and validation from dog class. So, let us setup some variables like batch size, epochs and the height and width of the image.

(Refer Slide Time: 05:09)



```
Total training images: 2000
Total validation images: 1000
```

For convenience, set up variables to use while pre-processing the dataset and training the network.

```
[10] batch_size = 128
     epochs = 15
     IMG_HEIGHT = 150
     IMG_WIDTH = 150
```

Data preparation

Format the images into appropriately pre-processed floating point tensors before feeding to the network:

1. Read images from the disk.
2. Decode contents of these images and convert it into proper grid format as per their RGB content.
3. Convert them into floating point tensors.
4. Rescale the tensors from values between 0 and 255 to values between 0 and 1, as neural networks prefer to deal with small input values.

Fortunately, all these tasks can be done with the `ImageDataGenerator` class provided by `tf.keras`. It can read images from disk and preprocess them into proper tensors. It will also set up generators that convert these images into batches of tensors—helpful when training the network.

```
train_image_generator = ImageDataGenerator(rescale=1./255) # Generator for our training data
validation_image_generator = ImageDataGenerator(rescale=1./255) # Generator for our validation data
```

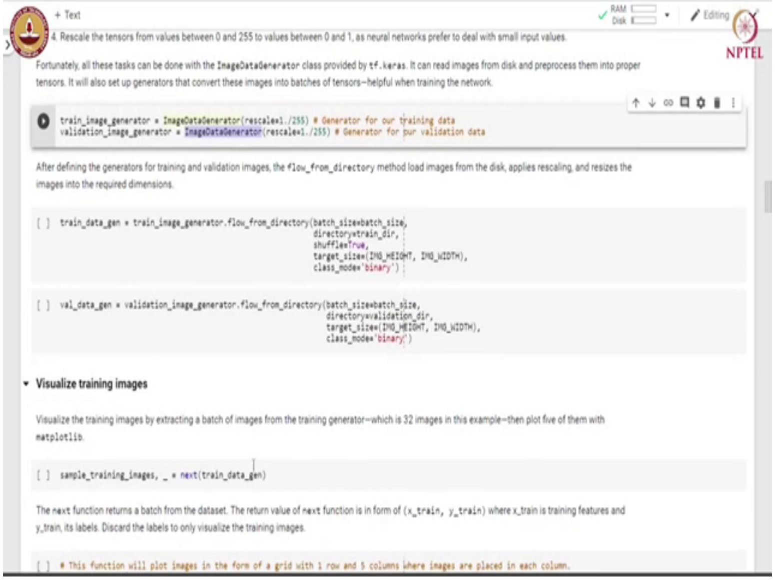
After defining the generators for training and validation images, the `flow_from_directory` method load images from the disk, applies rescaling, and resizes the images into the required dimensions.

```
[ ] train_data_gen = train_image_generator.flow_from_directory(batch_size=batch_size,
                                                           directory=train_dir,
                                                           shuffle=True,
                                                           target_size=(IMG_HEIGHT, IMG_WIDTH),
                                                           class_mode='binary')
```

Let us prepare the data for training. So, we perform the following steps we will first read the images from the disk, then we will decode the content of this images and convert them into proper grid format as per their RGB content. Then we convert them into floating point tensors and then we rescale this tensors from values between 0 and 255 to values between 0 and 1.

So, all this task are done by image data generator class provided by tf.keras. It can read images from the disc and preprocess them into proper tensors. It will also setup generators that convert this images into batches of tensors which is very helpful during the training. So, we setup image data generator for training and validation set.

(Refer Slide Time: 06:22)



4. Rescale the tensors from values between 0 and 255 to values between 0 and 1, as neural networks prefer to deal with small input values.

Fortunately, all these tasks can be done with the `ImageDataGenerator` class provided by `tf.keras`. It can read images from disk and preprocess them into proper tensors. It will also set up generators that convert these images into batches of tensors—helpful when training the network.

```
train_image_generator = ImageDataGenerator(rescale=1./255) # Generator for our training data
validation_image_generator = ImageDataGenerator(rescale=1./255) # Generator for our validation data
```

After defining the generators for training and validation images, the `flow_from_directory` method load images from the disk, applies rescaling, and resizes the images into the required dimensions.

```
[ ] train_data_gen = train_image_generator.flow_from_directory(batch_size=batch_size,
                                                           directory=train_dir,
                                                           shuffle=True,
                                                           target_size=(IMG_HEIGHT, IMG_WIDTH),
                                                           class_mode='binary')
```

```
[ ] val_data_gen = validation_image_generator.flow_from_directory(batch_size=batch_size,
                                                              directory=validation_dir,
                                                              target_size=(IMG_HEIGHT, IMG_WIDTH),
                                                              class_mode='binary')
```

Visualize training images

Visualize the training images by extracting a batch of images from the training generator—which is 32 images in this example—then plot five of them with `matplotlib`.

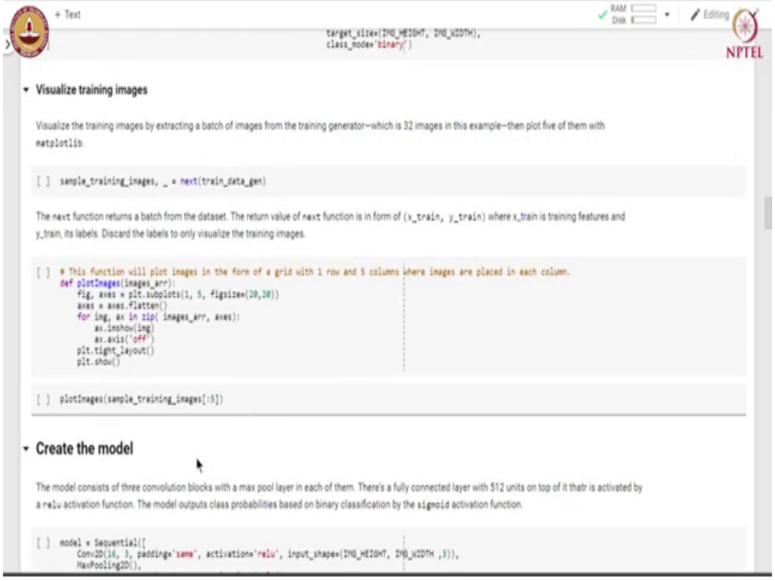
```
[ ] sample_training_images, _ = next(train_data_gen)
```

The `next` function returns a batch from the dataset. The return value of `next` function is in form of `(x_train, y_train)` where x_train is training features and y_train, its labels. Discard the labels to only visualize the training images.

```
[ ] # This function will plot images in the form of a grid with 1 row and 5 columns where images are placed in each column.
```

After defining these generators, we use flow from directory method to load images from the disc, apply rescaling and resizing of the image into required dimension. So, here the target image size is 150 x 150 and we want to shuffle the training data. We do not shuffle the data in the validation set. We also specify the batch size and the directories were data is stored.

(Refer Slide Time: 07:02)



```
        target_size=(IMG_HEIGHT, IMG_WIDTH),
        class_mode='binary')
```

Visualize training images

Visualize the training images by extracting a batch of images from the training generator—which is 32 images in this example—then plot five of them with matplotlib.

```
[ ] sample_training_images, _ = next(train_data_gen)
```

The next function returns a batch from the dataset. The return value of next function is in form of (x_train, y_train) where x_train is training features and y_train, its labels. Discard the labels to only visualize the training images.

```
[ ] # This function will plot images in the form of a grid with 1 row and 5 columns where images are placed in each column.
    def plotImages(images_arr):
        fig, axes = plt.subplots(1, 5, figsize=(20,20))
        axes = axes.flatten()
        for img, ax in zip( images_arr, axes):
            ax.imshow(img)
            ax.axis('off')
        plt.tight_layout()
        plt.show()
```

```
[ ] plotImages(sample_training_images[:5])
```

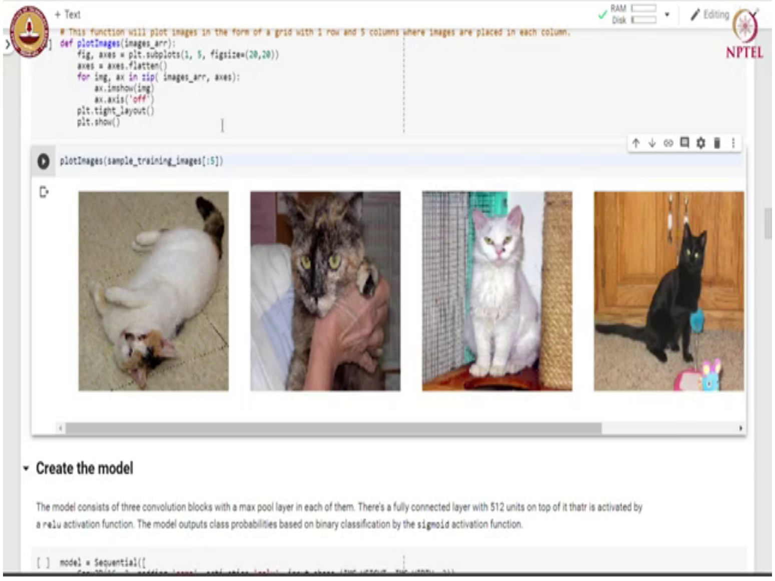
Create the model

The model consists of three convolution blocks with a max pool layer in each of them. There's a fully connected layer with 512 units on top of it thatr is activated by a relu activation function. The model outputs class probabilities based on binary classification by the sigmoid activation function.

```
[ ] model = Sequential([
        Conv2D(16, 3, padding='same', activation='relu', input_shape=(IMG_HEIGHT, IMG_WIDTH ,3)),
        MaxPooling2D(),
```

Let us visualize the training images by extracting a batch of images from the image generator. Let us plot five of these images.

(Refer Slide Time: 07:28)



```
    # This function will plot images in the form of a grid with 1 row and 5 columns where images are placed in each column.
    def plotImages(images_arr):
        fig, axes = plt.subplots(1, 5, figsize=(20,20))
        axes = axes.flatten()
        for img, ax in zip( images_arr, axes):
            ax.imshow(img)
            ax.axis('off')
        plt.tight_layout()
        plt.show()
```

```
plotImages(sample_training_images[:5])
```

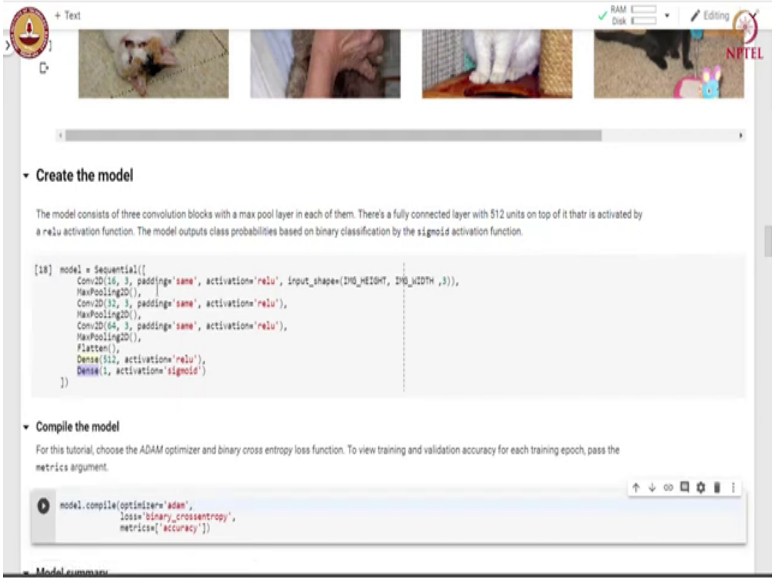
Create the model

The model consists of three convolution blocks with a max pool layer in each of them. There's a fully connected layer with 512 units on top of it thatr is activated by a relu activation function. The model outputs class probabilities based on binary classification by the sigmoid activation function.

```
[ ] model = Sequential([
```

We use matplotlib for plotting these images.

(Refer Slide Time: 07:45)



Create the model

The model consists of three convolution blocks with a max pool layer in each of them. There's a fully connected layer with 512 units on top of it thatr is activated by a `relu` activation function. The model outputs class probabilities based on binary classification by the `sigmoid` activation function.

```
[18] model = Sequential([
         Conv2D(16, 3, padding='same', activation='relu', input_shape=(IMG_HEIGHT, IMG_WIDTH ,3)),
         MaxPooling2D(),
         Conv2D(32, 3, padding='same', activation='relu'),
         MaxPooling2D(),
         Conv2D(64, 3, padding='same', activation='relu'),
         MaxPooling2D(),
         Flatten(),
         Dense(512, activation='relu'),
         Dense(1, activation='sigmoid')
     ])
```

Compile the model

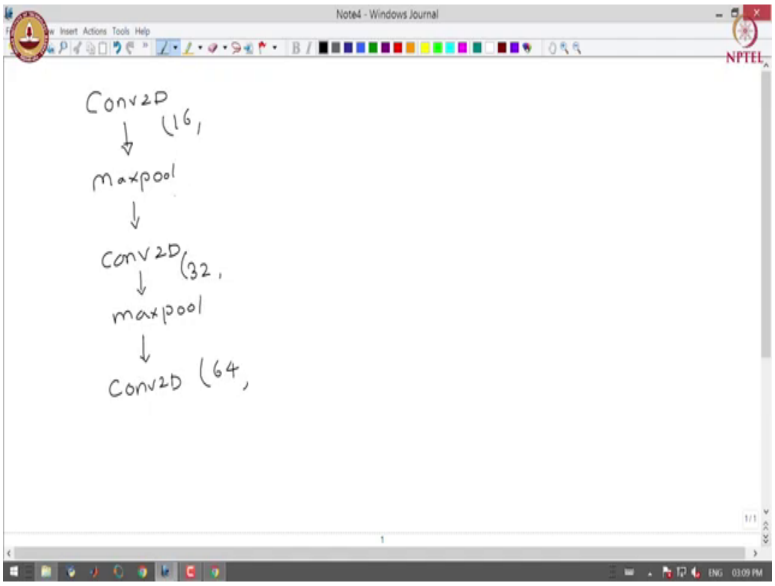
For this tutorial, choose the *ADAM* optimizer and *binary cross entropy* loss function. To view training and validation accuracy for each training epoch, pass the `metrics` argument.

```
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

Model summary

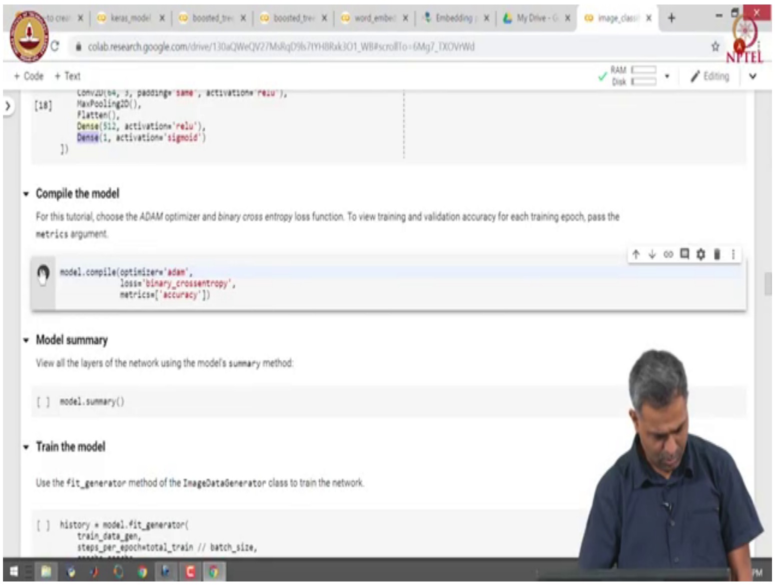
Let us create a model a CNN model for classifying cats and dogs. So, we use three convolution blocks with a MaxPool layer in each one of them. Then we use a fully connected layer with 512 units on top of that with Relu activation. And the model outputs class probabilities based on binary classification by sigmoid activation function in the output layer. Let us look at the structure of the model.

(Refer Slide Time: 08:47)



So, we use a Conv2D followed by MaxPool, then another convolution layer.

(Refer Slide Time: 09:37)



```
    Conv2D(64, 3, padding='same', activation='relu'),
[18]    MaxPooling2D(),
        Flatten(),
        Dense(512, activation='relu'),
        Dense(1, activation='sigmoid')
    ])
```

▾ **Compile the model**

For this tutorial, choose the *ADAM* optimizer and *binary cross entropy* loss function. To view training and validation accuracy for each training epoch, pass the `metrics` argument.

```
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

▾ **Model summary**

View all the layers of the network using the model's `summary` method:

```
[ ] model.summary()
```

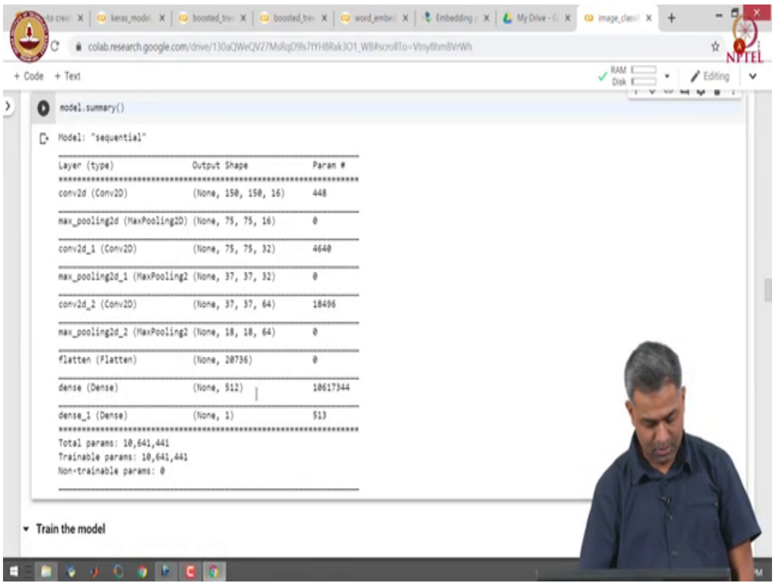
▾ **Train the model**

Use the `fit_generator` method of the `ImageDataGenerator` class to train the network.

```
[ ] history = model.fit_generator(
        train_data_gen,
        steps_per_epoch=total_train // batch_size,
```

Reduce *binary_crossentropy* as a loss with *Adam* as an optimizer and *accuracy* as a metric to track.

(Refer Slide Time: 09:52)

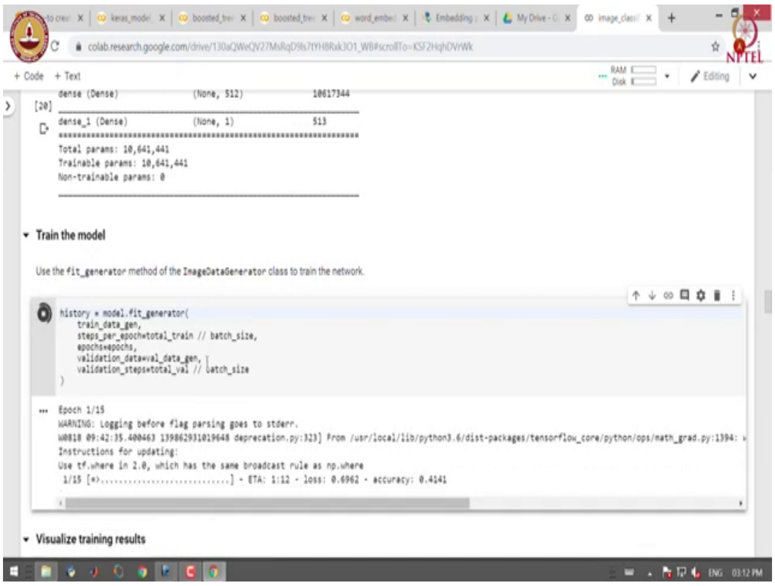


```
model.summary()

Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 150, 150, 16)      448
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 75, 75, 16)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 75, 75, 32)        4640
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 37, 37, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 37, 37, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 18, 18, 64)        0
_________________________________________________________________
flatten (Flatten)            (None, 20736)             0
_________________________________________________________________
dense (Dense)                (None, 512)               10617344
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 513
=================================================================
Total params: 10,641,441
Trainable params: 10,641,441
Non-trainable params: 0
_________________________________________________________________
```

▾ **Train the model**

So, you can see that we have modelled with more than 10 million parameters.

(Refer Slide Time: 10:25)



```
dense (Dense)                (None, 512)               10617344
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 513
=================================================================
Total params: 10,641,441
Trainable params: 10,641,441
Non-trainable params: 0
_________________________________________________________________
```

Train the model

Use the `fit_generator` method of the `ImageDataGenerator` class to train the network.

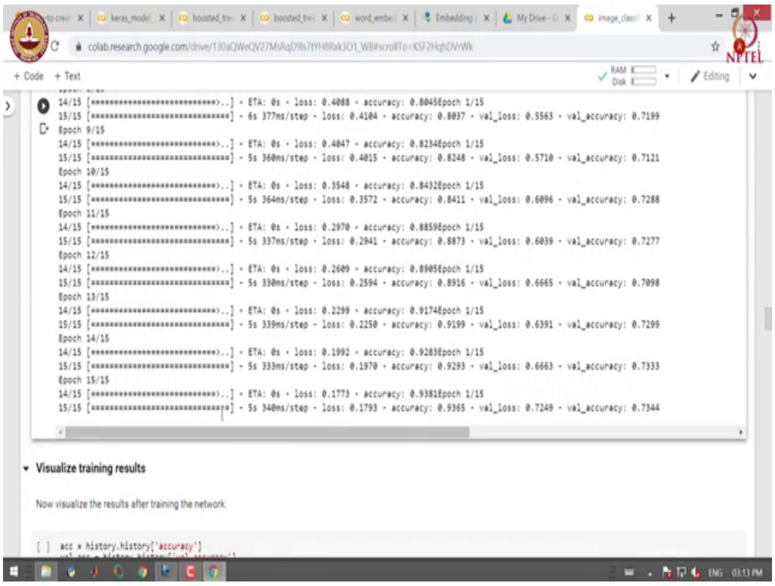
```
history = model.fit_generator(
    train_data_gen,
    steps_per_epoch=total_train // batch_size,
    epochs=epochs,
    validation_data=val_data_gen,
    validation_steps=total_val // batch_size
)
```

```
Epoch 1/15
WARNING: Logging before flag parsing goes to stderr.
W0818 09:42:35.400463 139862931019648 deprecation.py:323] From /usr/local/lib/python3.6/dist-packages/tensorflow_core/python/ops/math_grad.py:1394: w
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where
 1/15 [=>............................] - ETA: 1:12 - loss: 0.6962 - accuracy: 0.4141
```

Visualize training results

Let us train the model for 15 epochs.

(Refer Slide Time: 10:35)



```
14/15 [===========================>..] - ETA: 0s - loss: 0.4088 - accuracy: 0.8045Epoch 1/15
15/15 [==============================] - 6s 377ms/step - loss: 0.4104 - accuracy: 0.8037 - val_loss: 0.5563 - val_accuracy: 0.7199
Epoch 9/15
14/15 [===========================>..] - ETA: 0s - loss: 0.4047 - accuracy: 0.8234Epoch 1/15
15/15 [==============================] - 5s 360ms/step - loss: 0.4015 - accuracy: 0.8248 - val_loss: 0.5710 - val_accuracy: 0.7121
Epoch 10/15
14/15 [===========================>..] - ETA: 0s - loss: 0.3548 - accuracy: 0.8432Epoch 1/15
15/15 [==============================] - 5s 364ms/step - loss: 0.3572 - accuracy: 0.8411 - val_loss: 0.6096 - val_accuracy: 0.7288
Epoch 11/15
14/15 [===========================>..] - ETA: 0s - loss: 0.2970 - accuracy: 0.8859Epoch 1/15
15/15 [==============================] - 5s 337ms/step - loss: 0.2941 - accuracy: 0.8873 - val_loss: 0.6039 - val_accuracy: 0.7277
Epoch 12/15
14/15 [===========================>..] - ETA: 0s - loss: 0.2609 - accuracy: 0.8905Epoch 1/15
15/15 [==============================] - 5s 330ms/step - loss: 0.2594 - accuracy: 0.8916 - val_loss: 0.6665 - val_accuracy: 0.7098
Epoch 13/15
14/15 [===========================>..] - ETA: 0s - loss: 0.2299 - accuracy: 0.9174Epoch 1/15
15/15 [==============================] - 5s 339ms/step - loss: 0.2250 - accuracy: 0.9199 - val_loss: 0.6391 - val_accuracy: 0.7299
Epoch 14/15
14/15 [===========================>..] - ETA: 0s - loss: 0.1992 - accuracy: 0.9283Epoch 1/15
15/15 [==============================] - 5s 333ms/step - loss: 0.1970 - accuracy: 0.9293 - val_loss: 0.6663 - val_accuracy: 0.7333
Epoch 15/15
14/15 [===========================>..] - ETA: 0s - loss: 0.1773 - accuracy: 0.9381Epoch 1/15
15/15 [==============================] - 5s 340ms/step - loss: 0.1793 - accuracy: 0.9365 - val_loss: 0.7249 - val_accuracy: 0.7344
```

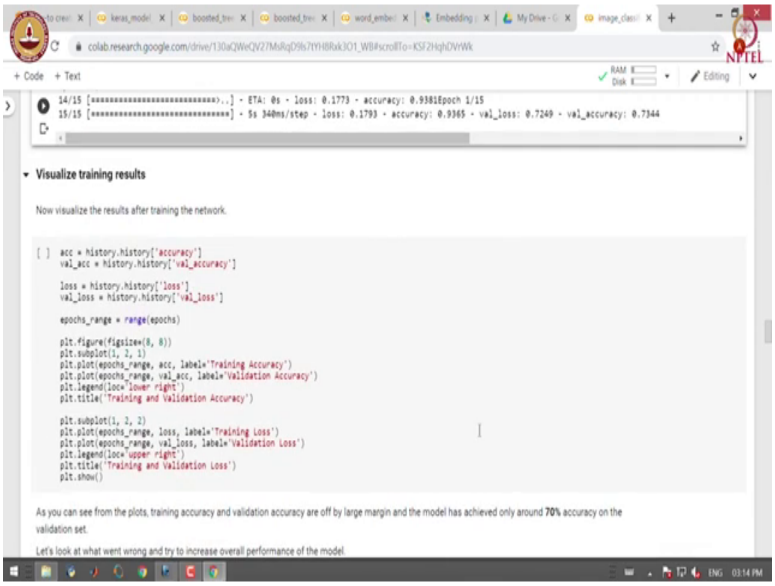
Visualize training results

Now visualize the results after training the network.

```
acc = history.history['accuracy']
```

So after 15 iteration we got accuracy closed to 94 % on the training set and 73 % on the validation set.

(Refer Slide Time: 10:48)



```
14/15 [===========================>..] - ETA: 0s - loss: 0.1773 - accuracy: 0.9381Epoch 1/15
15/15 [==============================] - 5s 340ms/step - loss: 0.1793 - accuracy: 0.9365 - val_loss: 0.7249 - val_accuracy: 0.7344
```

Visualize training results

Now visualize the results after training the network.

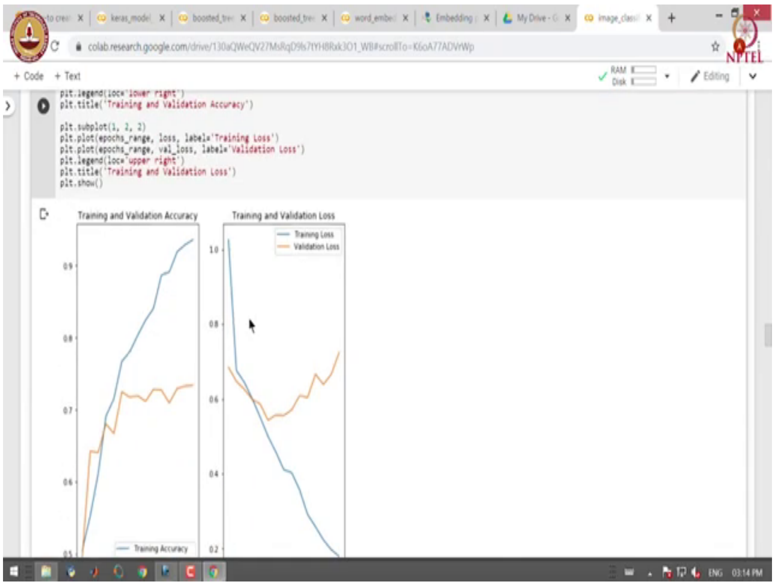
```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(epochs)

plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```

As you can see from the plots, training accuracy and validation accuracy are off by large margin and the model has achieved only around **70%** accuracy on the validation set.

Let's look at what went wrong and try to increase overall performance of the model.

Let us visualize the training and validation accuracy across epochs.

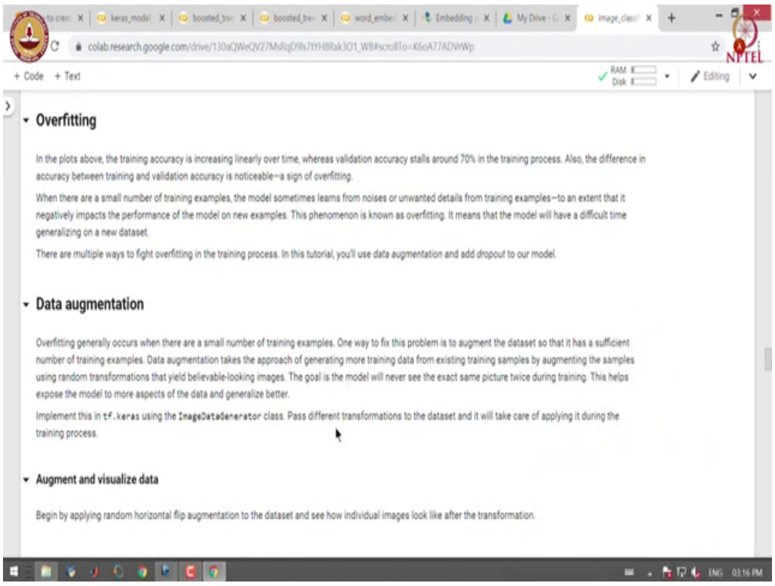
(Refer Slide Time: 10:56)



```
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```

You can see that as we trained for more epochs, the training the training accuracy kept raising whereas, the validation accuracy plateaued after some time. We also see similar trend in the training and validation loss where you found that training loss was constantly decreasing as you train for more epochs, but validation loss initially declined, but then grows steadily after certain epochs after about 50 epochs.

(Refer Slide Time: 11:37)



colab.research.google.com/drive/130aQWeQV27MsRqD9ls7tYH8Rxk3O1_WB#scrollTo=K6oA77ADVrWp
+ Code
+ Text
RAM
Disk
Editing

Overfitting

In the plots above, the training accuracy is increasing linearly over time, whereas validation accuracy stalls around 70% in the training process. Also, the difference in accuracy between training and validation accuracy is noticeable—a sign of overfitting.

When there are a small number of training examples, the model sometimes learns from noises or unwanted details from training examples—to an extent that it negatively impacts the performance of the model on new examples. This phenomenon is known as overfitting. It means that the model will have a difficult time generalizing on a new dataset.

There are multiple ways to fight overfitting in the training process. In this tutorial, you'll use *data augmentation* and add *dropout* to our model.

Data augmentation

Overfitting generally occurs when there are a small number of training examples. One way to fix this problem is to augment the dataset so that it has a sufficient number of training examples. Data augmentation takes the approach of generating more training data from existing training samples by augmenting the samples using random transformations that yield believable-looking images. The goal is the model will never see the exact same picture twice during training. This helps expose the model to more aspects of the data and generalize better.

Implement this in `tf.keras` using the `ImageDataGenerator` class. Pass different transformations to the dataset and it will take care of applying it during the training process.

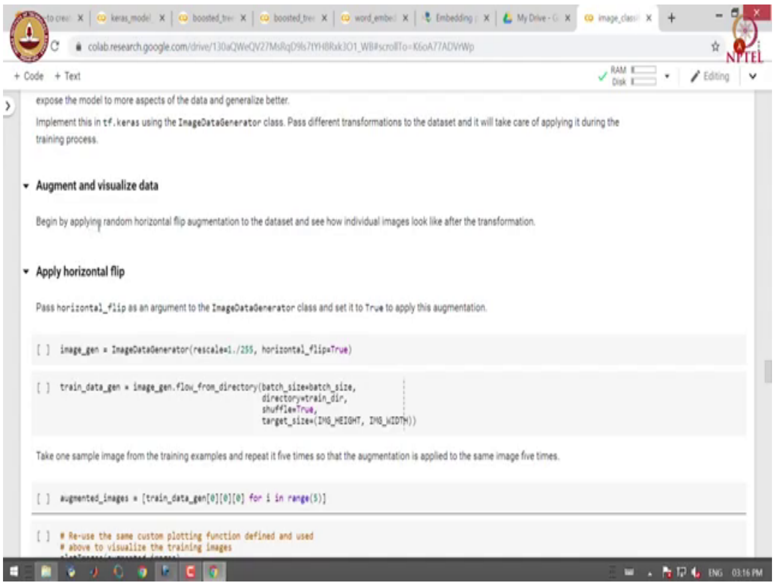
Augment and visualize data

Begin by applying random horizontal flip augmentation to the dataset and see how individual images look like after the transformation.

So, this points to the fact that the model is over fitting. So, you have to explore strategies to increase the performance of the model by reducing the over-fitting. So, we will use data augmentation and dropout as two strategies for fighting over-fitting problem. In data augmentation, we will take the existing images and perform certain transformation on them to gather more data.

So, what kind of transformations we do? We can rotate, translate, change the scale of the image to create more and more examples of the image such that model is less likely to over-fit with more data.

(Refer Slide Time: 12:39)



expose the model to more aspects of the data and generalize better.

Implement this in tf.keras using the ImageDataGenerator class. Pass different transformations to the dataset and it will take care of applying it during the training process.

Augment and visualize data

Begin by applying random horizontal flip augmentation to the dataset and see how individual images look like after the transformation.

Apply horizontal flip

Pass horizontal_flip as an argument to the ImageDataGenerator class and set it to True to apply this augmentation.

```
[ ] image_gen = ImageDataGenerator(rescale=1./255, horizontal_flip=True)
```

```
[ ] train_data_gen = image_gen.flow_from_directory(batch_size=batch_size,
                                                   directory=train_dir,
                                                   shuffle=True,
                                                   target_size=(IMG_HEIGHT, IMG_WIDTH))
```

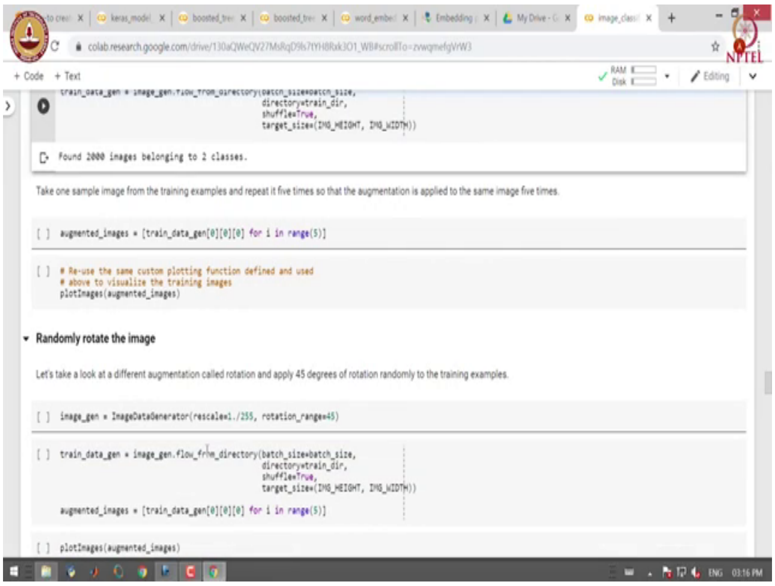
Take one sample image from the training examples and repeat it five times so that the augmentation is applied to the same image five times.

```
[ ] augmented_images = [train_data_gen[0][0][0] for i in range(5)]
```

```
[ ] # Re-use the same custom plotting function defined and used
    # above to visualize the training images
```

So, let us try to augment the data and visualize it. We can perform random horizontal flip augmentation to the dataset. We can use ImageDataGenerator with horizontal flip = *True* and let us generate the data.

(Refer Slide Time: 13:09)



```
    train_data_gen = image_gen.flow_from_directory(batch_size=batch_size,
                                                   directory=train_dir,
                                                   shuffle=True,
                                                   target_size=(IMG_HEIGHT, IMG_WIDTH))

    Found 2000 images belonging to 2 classes.
```

Take one sample image from the training examples and repeat it five times so that the augmentation is applied to the same image five times.

```
[ ] augmented_images = [train_data_gen[0][0][0] for i in range(5)]
```

```
[ ] # Re-use the same custom plotting function defined and used
    # above to visualize the training images
    plotImages(augmented_images)
```

Randomly rotate the image

Let's take a look at a different augmentation called rotation and apply 45 degrees of rotation randomly to the training examples.

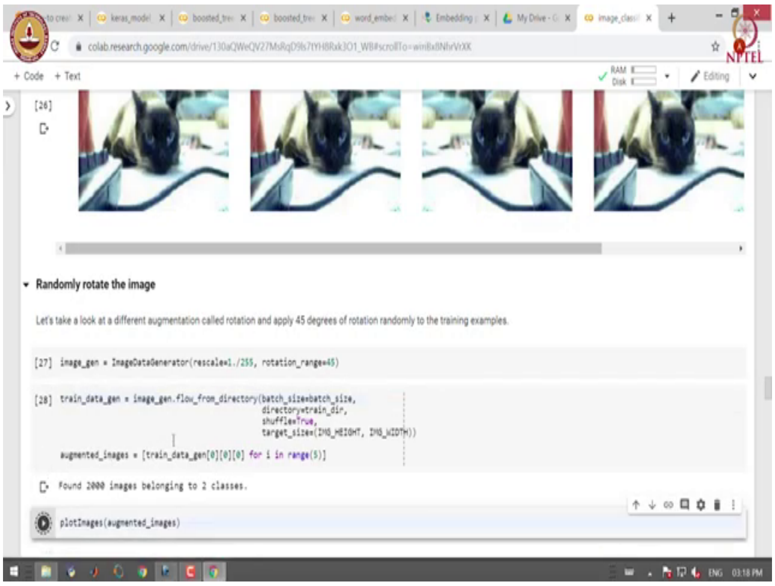
```
[ ] image_gen = ImageDataGenerator(rescale=1./255, rotation_range=45)
```

```
[ ] train_data_gen = image_gen.flow_from_directory(batch_size=batch_size,
                                                   directory=train_dir,
                                                   shuffle=True,
                                                   target_size=(IMG_HEIGHT, IMG_WIDTH))

    augmented_images = [train_data_gen[0][0][0] for i in range(5)]
```

```
[ ] plotImages(augmented_images)
```

Let us take one example and see how the data got augmented with horizontal flipping.

(Refer Slide Time: 13:22)



Randomly rotate the image

Let's take a look at a different augmentation called rotation and apply 45 degrees of rotation randomly to the training examples.

```
[27] image_gen = ImageDataGenerator(rescale=1./255, rotation_range=45)
```

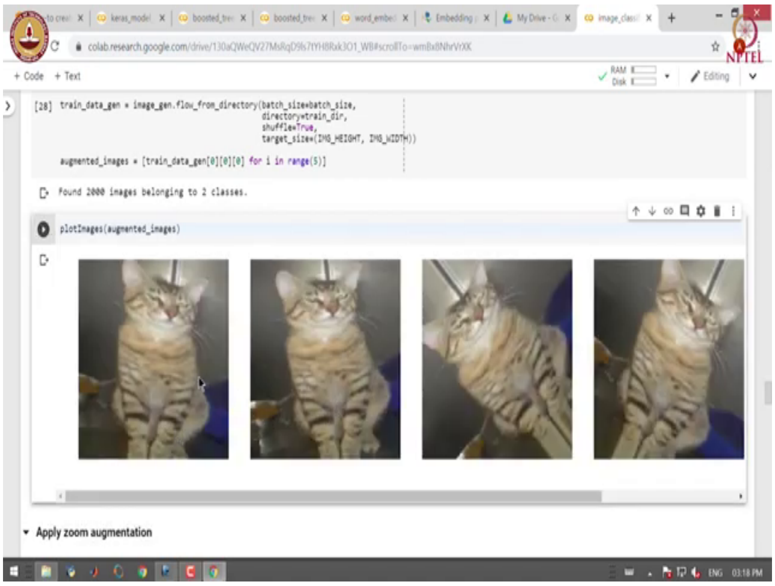
```
[28] train_data_gen = image_gen.flow_from_directory(batch_size=batch_size,
                                                   directory=train_dir,
                                                   shuffle=True,
                                                   target_size=(IMG_HEIGHT, IMG_WIDTH))

     augmented_images = [train_data_gen[0][0][0] for i in range(5)]
```

```
Found 2000 images belonging to 2 classes.
```

```
plotImages(augmented_images)
```

So, after horizontal flipping, we got more examples that were generated from a single example. We can also do a different kind of augmentation by rotation. Let us apply 45 degree rotation randomly to the training examples.

(Refer Slide Time: 14:17)



```
[28] train_data_gen = image_gen.flow_from_directory(batch_size=batch_size,
                                                   directory=train_dir,
                                                   shuffle=True,
                                                   target_size=(IMG_HEIGHT, IMG_WIDTH))

     augmented_images = [train_data_gen[0][0][0] for i in range(5)]
```

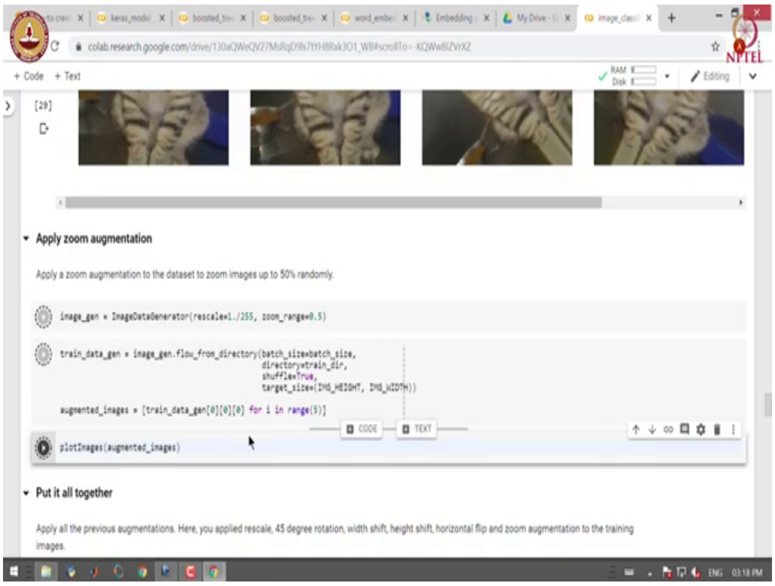
```
Found 2000 images belonging to 2 classes.
```

```
plotImages(augmented_images)
```

Apply zoom augmentation

Now, you can see that the same picture of the cat was rotated in different orientations with different angles and by applying this particular data augmentation strategy, we created more pictures from a single picture.

(Refer Slide Time: 14:45)



Apply zoom augmentation

Apply a zoom augmentation to the dataset to zoom images up to 50% randomly.

```
image_gen = ImageDataGenerator(rescale=1./255, zoom_range=0.5)
```

```
train_data_gen = image_gen.flow_from_directory(batch_size=batch_size,
                                               directory=train_dir,
                                               shuffle=True,
                                               target_size=(IMG_HEIGHT, IMG_WIDTH))

augmented_images = [train_data_gen[0][0][0] for i in range(5)]
```

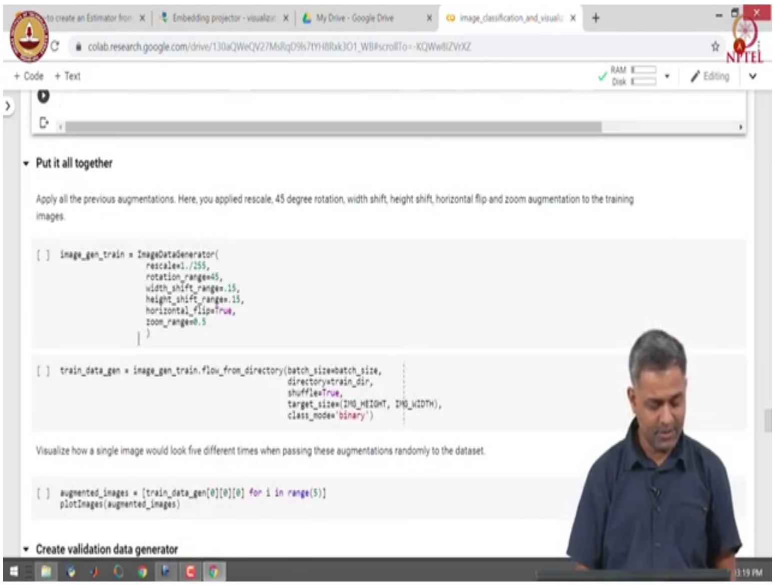
```
plotImages(augmented_images)
```

Put it all together

Apply all the previous augmentations. Here, you applied rescale, 45 degree rotation, width shift, height shift, horizontal flip and zoom augmentation to the training images.

In addition to that we can also apply zoom augmentation by specifying the zoom range. Let us see how zoom augmentation look like.

(Refer Slide Time: 14:59)



Put it all together

Apply all the previous augmentations. Here, you applied rescale, 45 degree rotation, width shift, height shift, horizontal flip and zoom augmentation to the training images.

```
image_gen_train = ImageDataGenerator(
                    rescale=1./255,
                    rotation_range=45,
                    width_shift_range=.15,
                    height_shift_range=.15,
                    horizontal_flip=True,
                    zoom_range=0.5
                    )
```

```
train_data_gen = image_gen_train.flow_from_directory(batch_size=batch_size,
                                                     directory=train_dir,
                                                     shuffle=True,
                                                     target_size=(IMG_HEIGHT, IMG_WIDTH),
                                                     class_mode='binary')
```

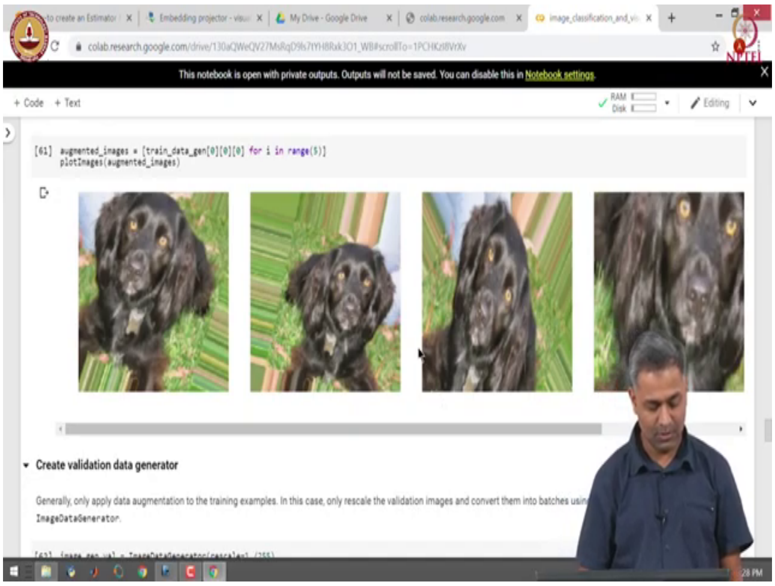
Visualize how a single image would look five different times when passing these augmentations randomly to the dataset.

```
augmented_images = [train_data_gen[0][0][0] for i in range(5)]
plotImages(augmented_images)
```

Create validation data generator

Let us put together all this augmentations strategies. So, we use ImageDataGenerator and apply rescaling, then rotation of 45 degrees, width shift, height shift and horizontal flip and the zoom augmentation to the training images.

(Refer Slide Time: 15:50)



This notebook is open with private outputs. Outputs will not be saved. You can disable this in Notebook settings.

```
[61] augmented_images = [train_data_gen[0][0][0] for i in range(5)]
     plotImages(augmented_images)
```

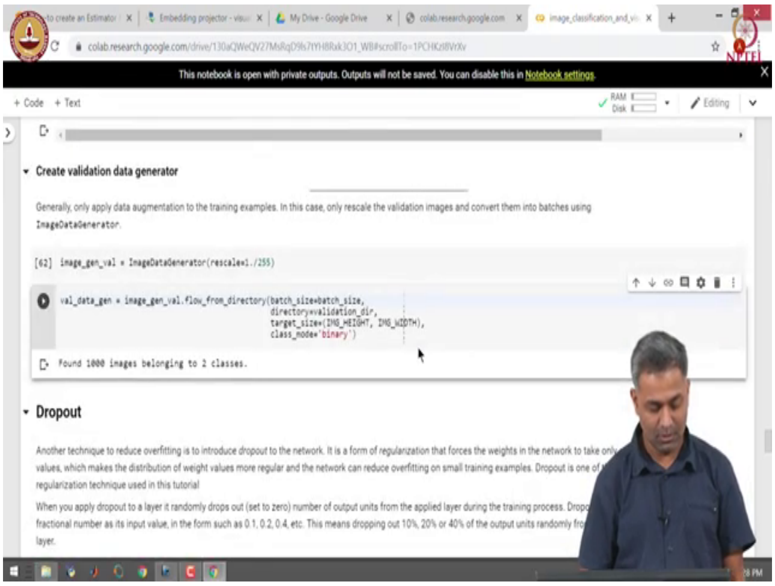
Create validation data generator

Generally, only apply data augmentation to the training examples. In this case, only rescale the validation images and convert them into batches usin

ImageDataGenerator.

So we apply all this augmentation techniques to the trainings set and obtain more data by augmenting the original images. These are some of the augmented pictures of a dog where we have the original picture and these are the pictures that were obtained by applying various data augmentations strategies.

(Refer Slide Time: 16:14)



This notebook is open with private outputs. Outputs will not be saved. You can disable this in Notebook settings.

Create validation data generator

Generally, only apply data augmentation to the training examples. In this case, only rescale the validation images and convert them into batches using ImageDataGenerator.

```
[62] image_gen_val = ImageDataGenerator(rescale=1./255)

val_data_gen = image_gen_val.flow_from_directory(batch_size=batch_size,
                                                  directory=validation_dir,
                                                  target_size=(IMG_HEIGHT, IMG_WIDTH),
                                                  class_mode='binary')

Found 1000 images belonging to 2 classes.
```

Dropout

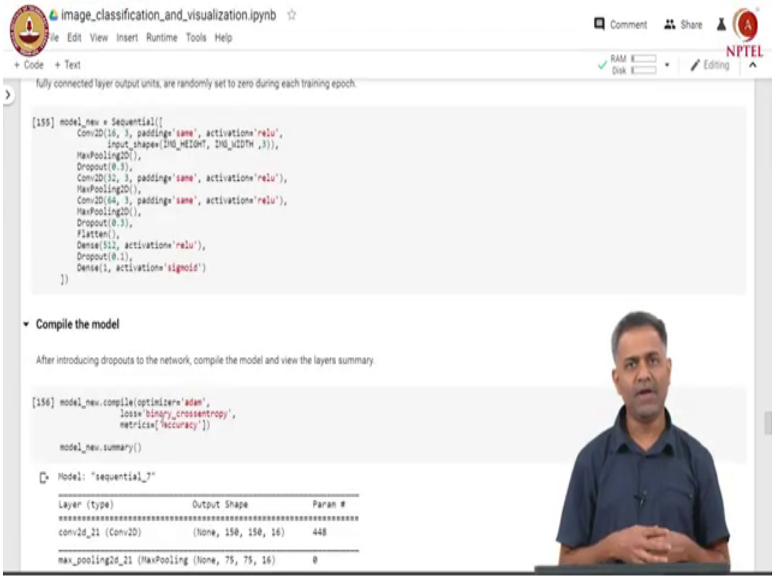
Another technique to reduce overfitting is to introduce dropout to the network. It is a form of regularization that forces the weights in the network to take only values, which makes the distribution of weight values more regular and the network can reduce overfitting on small training examples. Dropout is one of regularization technique used in this tutorial

When you apply dropout to a layer it randomly drops out (set to zero) number of output units from the applied layer during the training process. Drop fractional number as its input value, in the form such as 0.1, 0.2, 0.4, etc. This means dropping out 10%, 20% or 40% of the output units randomly fro layer.

We apply the data augmentation only to the training examples, we do not apply data augmentation on the validation examples except rescaling them. And we also convert the validation examples into batches using ImageDataGenerator.

Another technique to reduce over fitting is dropout.

(Refer Slide Time: 16:44)



image_classification_and_visualization.ipynb
File Edit View Insert Runtime Tools Help
Comment
Share
NPTEL
+ Code + Text
RAM
Disk
Editing

fully connected layer output units, are randomly set to zero during each training epoch.

```
[155] model_new = Sequential([
        Conv2D(16, 3, padding='same', activation='relu',
               input_shape=(IMG_HEIGHT, IMG_WIDTH ,3)),
        MaxPooling2D(),
        Dropout(0.3),
        Conv2D(32, 3, padding='same', activation='relu'),
        MaxPooling2D(),
        Conv2D(64, 3, padding='same', activation='relu'),
        MaxPooling2D(),
        Dropout(0.3),
        Flatten(),
        Dense(512, activation='relu'),
        Dropout(0.1),
        Dense(1, activation='sigmoid')
    ])
```

Compile the model

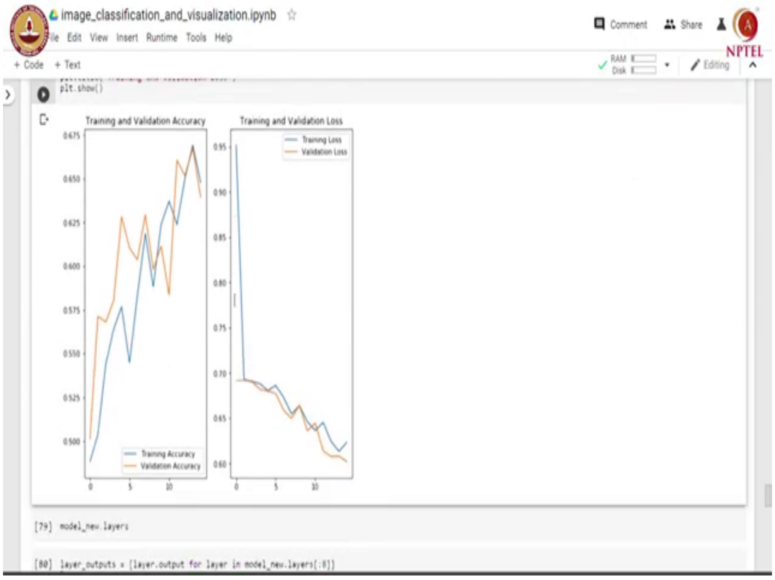
After introducing dropouts to the network, compile the model and view the layers summary.

```
[156] model_new.compile(optimizer='adam',
                  loss='binary_crossentropy',
                  metrics=['accuracy'])

      model_new.summary()
```

```
Model: "sequential_7"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_21 (Conv2D)           (None, 150, 150, 16)      448
_________________________________________________________________
max_pooling2d_21 (MaxPooling (None, 75, 75, 16)        0
```

We had dropout to the model after augmenting the data through various augmentation schemes. We had dropout of 0.3 each after couple of max pooling layers and Dropout of 0.1 after the fully connected layer. The effect of the dropout is that the randomly 30 % of max-pooling nodes and 10 percent of fully connected nodes are set to 0 during each epoch. We compile the model and train it and we can see that at the end of end of 15 epoch. We have a training accuracy of 64 % and validation accuracy of 63 %.

(Refer Slide Time: 17:32)



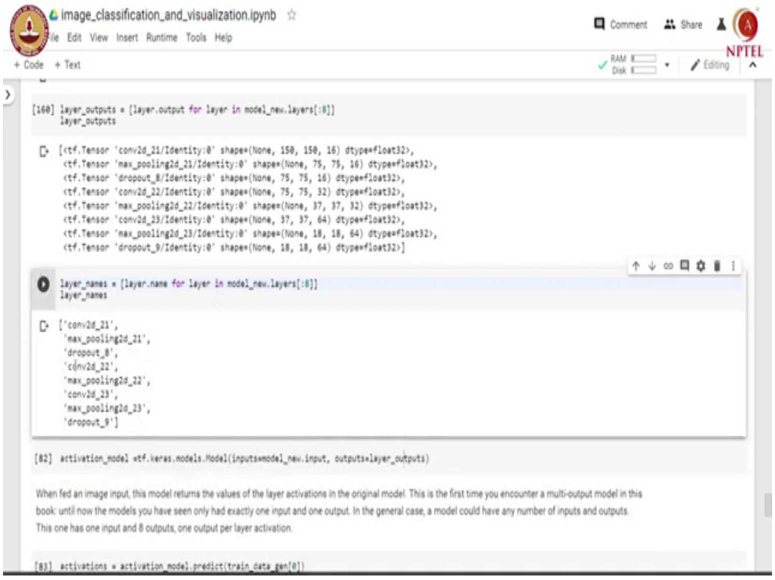
```
plt.show()
```

```
[79] model_new.layers
```

```
[80] layer_outputs = [layer.output for layer in model_new.layers[:8]]
```

And if we plot the losses, you can see that both the training and validation loss is trending together and there is a lesser over fitting than the earlier model.

(Refer Slide Time: 17:50)


image_classification_and_visualization.ipynb
File Edit View Insert Runtime Tools Help
Comment
Share
NPTEL
+ Code
+ Text
RAM
Disk
Editing

```
[160] layer_outputs = [layer.output for layer in model_new.layers[:8]]
      layer_outputs
```

```
[<tf.Tensor 'conv2d_21/Identity:0' shape=(None, 150, 150, 16) dtype=float32>,
 <tf.Tensor 'max_pooling2d_21/Identity:0' shape=(None, 75, 75, 16) dtype=float32>,
 <tf.Tensor 'dropout_8/Identity:0' shape=(None, 75, 75, 16) dtype=float32>,
 <tf.Tensor 'conv2d_22/Identity:0' shape=(None, 75, 75, 32) dtype=float32>,
 <tf.Tensor 'max_pooling2d_22/Identity:0' shape=(None, 37, 37, 32) dtype=float32>,
 <tf.Tensor 'conv2d_23/Identity:0' shape=(None, 37, 37, 64) dtype=float32>,
 <tf.Tensor 'max_pooling2d_23/Identity:0' shape=(None, 18, 18, 64) dtype=float32>,
 <tf.Tensor 'dropout_9/Identity:0' shape=(None, 18, 18, 64) dtype=float32>]
```

```
layer_names = [layer.name for layer in model_new.layers[:8]]
layer_names
```

```
['conv2d_21',
 'max_pooling2d_21',
 'dropout_8',
 'conv2d_22',
 'max_pooling2d_22',
 'conv2d_23',
 'max_pooling2d_23',
 'dropout_9']
```

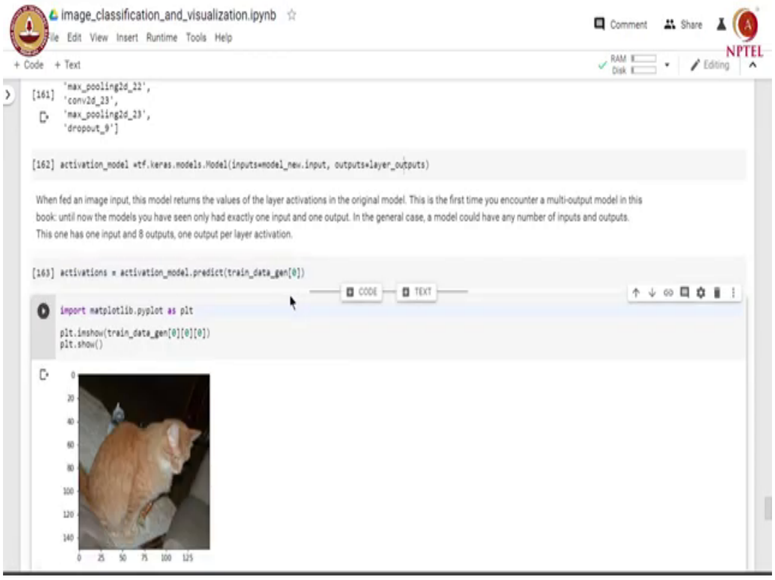
```
[82] activation_model =tf.keras.models.Model(inputs=model_new.input, outputs=layer_outputs)
```

When fed an image input, this model returns the values of the layer activations in the original model. This is the first time you encounter a multi-output model in this book: until now the models you have seen only had exactly one input and one output. In the general case, a model could have any number of inputs and outputs. This one has one input and 8 outputs, one output per layer activation.

```
[83] activations = activation_model.predict(train_data_gen[0])
```

Let us visualize what this particular CNN is learning. So, we will take the output of the model after every layer. So, there are eight different layers convolution followed by pooling, then there is a dropout then, convolution pooling, convolution pooling and dropout.

(Refer Slide Time: 18:27)



image_classification_and_visualization.ipynb
File Edit View Insert Runtime Tools Help
Comment Share
NPTEL
+ Code + Text
RAM
Disk
Editing
CODE
TEXT

```
[161] 'max_pooling2d_22',
      'conv2d_23',
      'max_pooling2d_23',
      'dropout_9']
```

```
[162] activation_model =tf.keras.models.Model(inputs=model_new.input, outputs=layer_outputs)
```

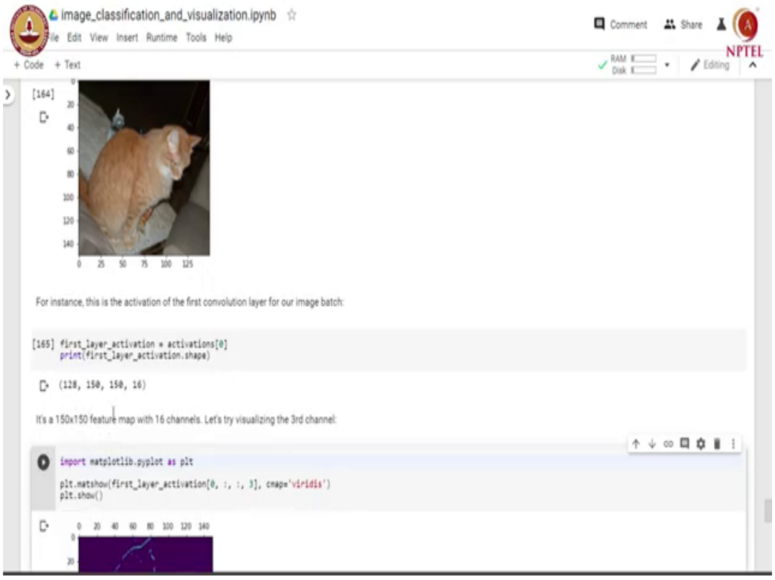
When fed an image input, this model returns the values of the layer activations in the original model. This is the first time you encounter a multi-output model in this book: until now the models you have seen only had exactly one input and one output. In the general case, a model could have any number of inputs and outputs. This one has one input and 8 outputs, one output per layer activation.

```
[163] activations = activation_model.predict(train_data_gen[0])
```

```
import matplotlib.pyplot as plt

plt.imshow(train_data_gen[0][0][0])
plt.show()
```

We get the output after every layer. So, this is different from the models that we have seen so far. This is the first time we are encountering a multi output model until now. We have models that were giving exactly one output at the end of the final layer here, we are gathering output after every layer. Let us look at the predictions coming from the activation model and let us plot those activations.

(Refer Slide Time: 19:07)



For instance, this is the activation of the first convolution layer for our image batch:

```
[165] first_layer_activation = activations[0]
      print(first_layer_activation.shape)
```

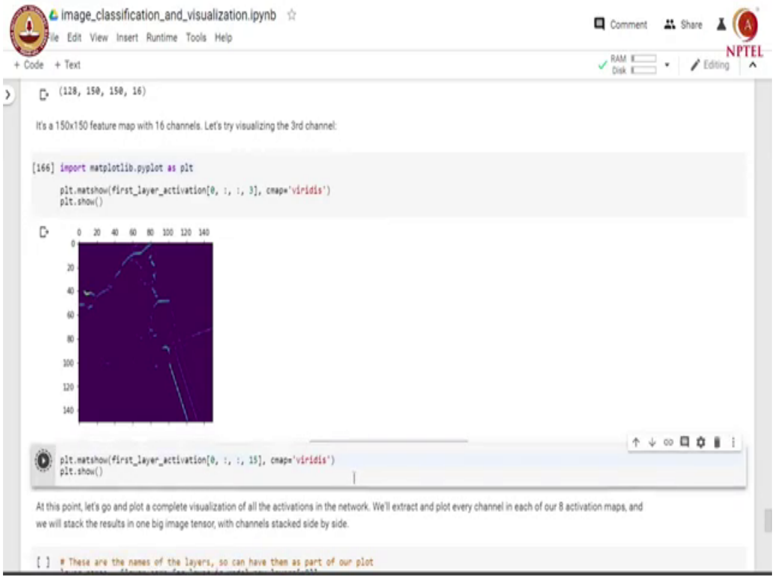
```
(128, 150, 150, 16)
```

It's a 150x150 feature map with 16 channels. Let's try visualizing the 3rd channel:

```
import matplotlib.pyplot as plt

plt.matshow(first_layer_activation[0, :, :, 3], cmap='viridis')
plt.show()
```

So, this is the input this is the first image that is input to us and the first level activation is essentially 150 x 150 feature map with 16 channels.

(Refer Slide Time: 19:23)



image_classification_and_visualization.ipynb
File Edit View Insert Runtime Tools Help
Comment Share
NPTEL
+ Code + Text
RAM Disk
Editing

```
(128, 150, 150, 16)
```

It's a 150x150 feature map with 16 channels. Let's try visualizing the 3rd channel:

```
[166] import matplotlib.pyplot as plt

      plt.matshow(first_layer_activation[0, :, :, 3], cmap='viridis')
      plt.show()
```

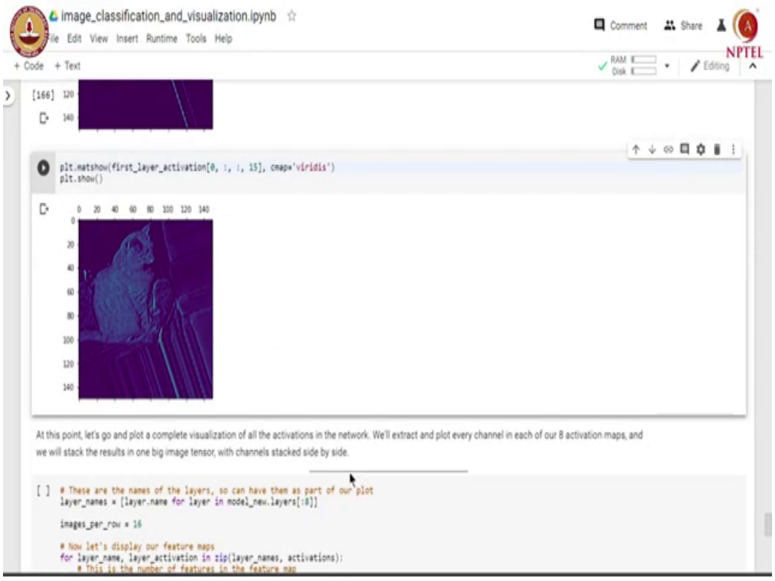
```
plt.matshow(first_layer_activation[0, :, :, 15], cmap='viridis')
plt.show()
```

At this point, let's go and plot a complete visualization of all the activations in the network. We'll extract and plot every channel in each of our 8 activation maps, and we will stack the results in one big image tensor, with channels stacked side by side.

```
[ ] # These are the names of the layers, so can have them as part of our plot
```

So, let us look at the third channel. You can see that the third channel is roughly detecting the edges from the cat picture.

(Refer Slide Time: 19:36)



image_classification_and_visualization.ipynb
File Edit View Insert Runtime Tools Help
Comment Share
NPTEL
+ Code + Text
RAM Disk
Editing
[166] 120
140

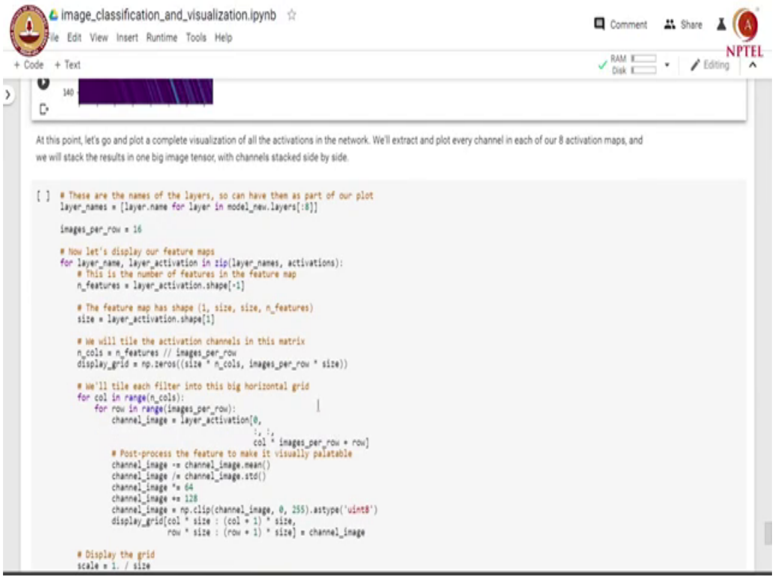
```
plt.matshow(first_layer_activation[0, :, :, 15], cmap='viridis')
plt.show()
```

At this point, let's go and plot a complete visualization of all the activations in the network. We'll extract and plot every channel in each of our 8 activation maps, and we will stack the results in one big image tensor, with channels stacked side by side.

```
[ ] # These are the names of the layers, so can have them as part of our plot
    layer_names = [layer.name for layer in model_new.layers[:8]]

    images_per_row = 16

    # Now let's display our feature maps
    for layer_name, layer_activation in zip(layer_names, activations):
        # This is the number of features in the feature map
```

If you look at the 15th channel, it is identifying probably the overall shape of the image. At this point let us go and plot a complete visualization of all the activations in the network.

(Refer Slide Time: 19:54)

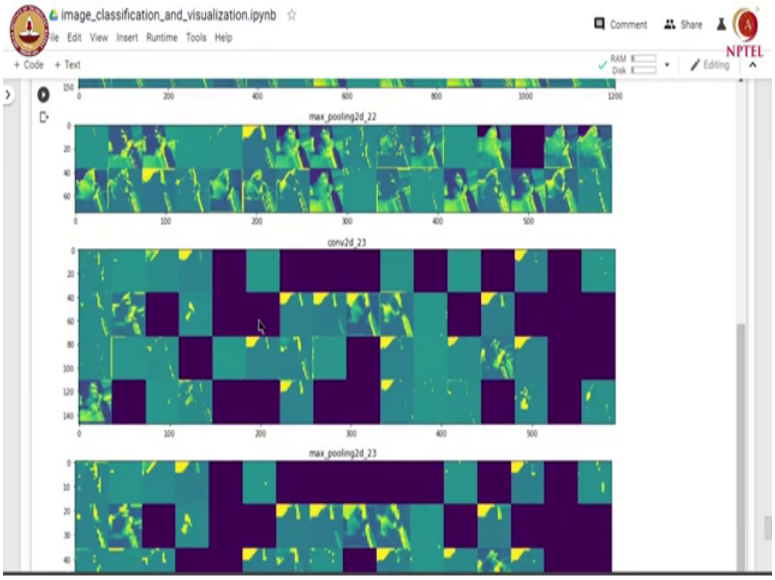

image_classification_and_visualization.ipynb
File Edit View Insert Runtime Tools Help
Comment
Share
+ Code
+ Text
RAM
Disk
Editing
NPTEL
140

At this point, let's go and plot a complete visualization of all the activations in the network. We'll extract and plot every channel in each of our 8 activation maps, and we will stack the results in one big image tensor, with channels stacked side by side.

```
[ ] # These are the names of the layers, so can have them as part of our plot
    layer_names = [layer.name for layer in model_new.layers[:8]]

    images_per_row = 16

    # Now let's display our feature maps
    for layer_name, layer_activation in zip(layer_names, activations):
        # This is the number of features in the feature map
        n_features = layer_activation.shape[-1]

        # The feature map has shape (1, size, size, n_features)
        size = layer_activation.shape[1]

        # We will tile the activation channels in this matrix
        n_cols = n_features // images_per_row
        display_grid = np.zeros((size * n_cols, images_per_row * size))

        # We'll tile each filter into this big horizontal grid
        for col in range(n_cols):
            for row in range(images_per_row):
                channel_image = layer_activation[0,
                                                 :, :,
                                                 col * images_per_row + row]
                # Post-process the feature to make it visually palatable
                channel_image -= channel_image.mean()
                channel_image /= channel_image.std()
                channel_image *= 64
                channel_image += 128
                channel_image = np.clip(channel_image, 0, 255).astype('uint8')
                display_grid[col * size : (col + 1) * size,
                             row * size : (row + 1) * size] = channel_image

        # Display the grid
        scale = 1. / size
```

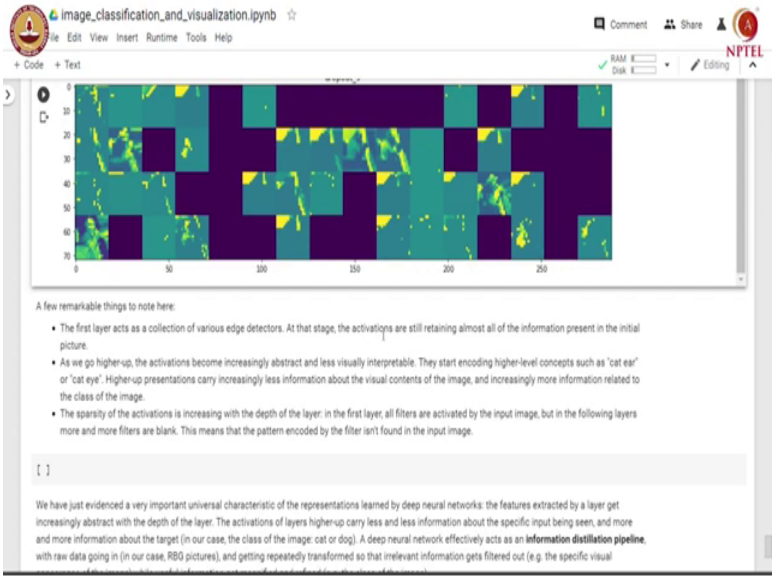
We will extract and plot every channel in each of our 8 activation maps and we will stack the results in one big image tensor with channels stacked side by side.

(Refer Slide Time: 20:06)



So, now you can see that the first convolution layer seems to be detecting edges and as we go deeper and deeper in the network. We are detecting different kind of features from the images. You can see that there are lot of empty spots after the second convolution and max pooling operation.

(Refer Slide Time: 20:38)


image_classification_and_visualization.ipynb
File Edit View Insert Runtime Tools Help
Comment
Share
NPTEL
+ Code
+ Text
RAM
Disk
Editing
0
10
20
30
40
50
60
70
0
50
100
150
200
250

A few remarkable things to note here:

- The first layer acts as a collection of various edge detectors. At that stage, the activations are still retaining almost all of the information present in the initial picture.
- As we go higher-up, the activations become increasingly abstract and less visually interpretable. They start encoding higher-level concepts such as "cat ear" or "cat eye". Higher-up presentations carry increasingly less information about the visual contents of the image, and increasingly more information related to the class of the image.
- The sparsity of the activations is increasing with the depth of the layer: in the first layer, all filters are activated by the input image, but in the following layers more and more filters are blank. This means that the pattern encoded by the filter isn't found in the input image.

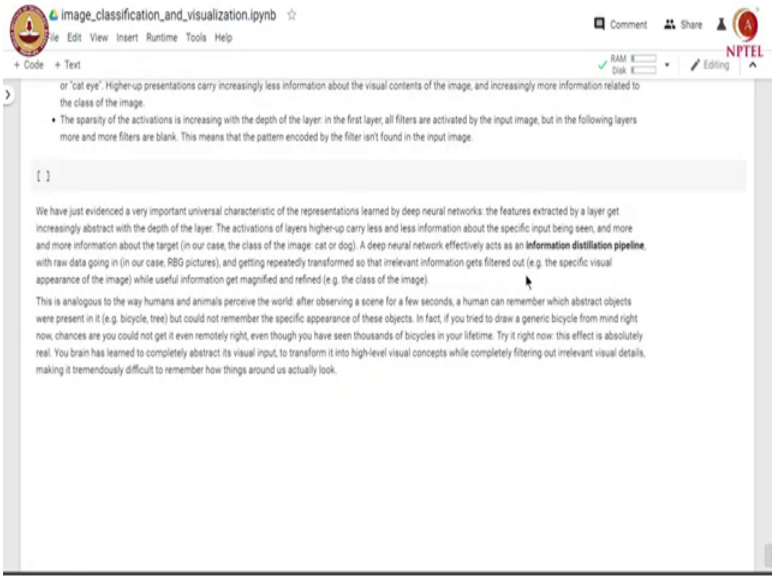
[ ]

We have just evidenced a very important universal characteristic of the representations learned by deep neural networks: the features extracted by a layer get increasingly abstract with the depth of the layer. The activations of layers higher-up carry less and less information about the specific input being seen, and more and more information about the target (in our case, the class of the image: cat or dog). A deep neural network effectively acts as an **information distillation pipeline**, with raw data going in (in our case, RBG pictures), and getting repeatedly transformed so that irrelevant information gets filtered out (e.g. the specific visual

There are few remarkable things to note here. The first layer acts as a collection of various edge detectors. At that stage the activations are still returning almost all the information present in the initial picture. You can see that lot of features from the input are being preserved. As we go higher up the activations become increasingly abstract and less visually interpretable. They start encoding higher level concepts such as cat ear and cat eye higher of representations carry increasingly less information about a visual content of the image and increase in the more information related to the class of the image.

The sparsity of activations is increasing with the depth of the layer. The first layer almost all filters are activated by the input image, but as we go deeper and deeper in the network there are more and more blank filters. This means that patterns encoded by the filter is not found in the input image. So, this is a very nice way of visualizing how convolution convolutional neural network is learning patterns in the image.

(Refer Slide Time: 22:14)


image_classification_and_visualization.ipynb
File Edit View Insert Runtime Tools Help
Comment
Share
+ Code
+ Text
RAM
Disk
Editing
NPTEL

or "cat eye". Higher-up presentations carry increasingly less information about the visual contents of the image, and increasingly more information related to the class of the image.

- The sparsity of the activations is increasing with the depth of the layer: in the first layer, all filters are activated by the input image, but in the following layers more and more filters are blank. This means that the pattern encoded by the filter isn't found in the input image.

[ ]

We have just evidenced a very important universal characteristic of the representations learned by deep neural networks: the features extracted by a layer get increasingly abstract with the depth of the layer. The activations of layers higher-up carry less and less information about the specific input being seen, and more and more information about the target (in our case, the class of the image: cat or dog). A deep neural network effectively acts as an **information distillation pipeline**, with raw data going in (in our case, RBG pictures), and getting repeatedly transformed so that irrelevant information gets filtered out (e.g. the specific visual appearance of the image) while useful information get magnified and refined (e.g. the class of the image).

This is analogous to the way humans and animals perceive the world: after observing a scene for a few seconds, a human can remember which abstract objects were present in it (e.g. bicycle, tree) but could not remember the specific appearance of these objects. In fact, if you tried to draw a generic bicycle from mind right now, chances are you could not get it even remotely right, even though you have seen thousands of bicycles in your lifetime. Try it right now: this effect is absolutely real. You brain has learned to completely abstract its visual input, to transform it into high-level visual concepts while completely filtering out irrelevant visual details, making it tremendously difficult to remember how things around us actually look.

So, we showed a very important universal characteristics of representations learned by deep neural network. The feature the features extracted by layer get increasingly abstract with depth of the layer. The activations of layers higher of carry less and less information about specific input being seen, but more and more information about the target class. A deep neural network effectively acts as an information distillation pipeline with raw data going in and getting repetitively transformed so, that irrelevant information gets filtered out and useful information gets magnified and refined.