Image classification from scratch in keras. Beginner friendly, intermediate exciting and expert refreshing.

Rising Odegua
Oct 26, 2018 · 19 min read

source: pixabay.com

Last weekend I was thinking out loud .

> what if I had a really really small data set of images that I captured myself and wanted to teach a computer to recognize or distinguish between some specified categories.

Let's say I have a few thousand images and I want to train a model to automatically detect one class from another. Would I be able to train a deep neural network to successfully classify these images since I have such little data?

Well, after doing some research, I found out that having to train deep neural network with little data was a common situation people encountered in the field of computer vision.

Image source: pixabay.com

> well let's face it, not everyone has access to big data like the Googs or the Faces, and some data are very difficult to obtain.

But I also found out that the solution to this problem was quite easy. So today I'm going to walk you through training a **Convolutional Neural Network** using that little Image data you have to get a really good classifier which will have an accuracy of about 81%.

In this **post**, I introduced a really powerful technique called **transfer learning** that helps us increase our accuracy up to about 95%.

Image source: pixabay.com

Alright you can go out and start gathering your data.

I'll be using an existing data set of dogs and cats available on the Kaggle platform. — -Yes, I'm lazy. I can't get my own data.

kaggle.com

Kaggle is the home of data science and machine learning practitioners worldwide. They host some of the biggest data science competitions and it is a great place for getting open source data as well as learning from winning experts.

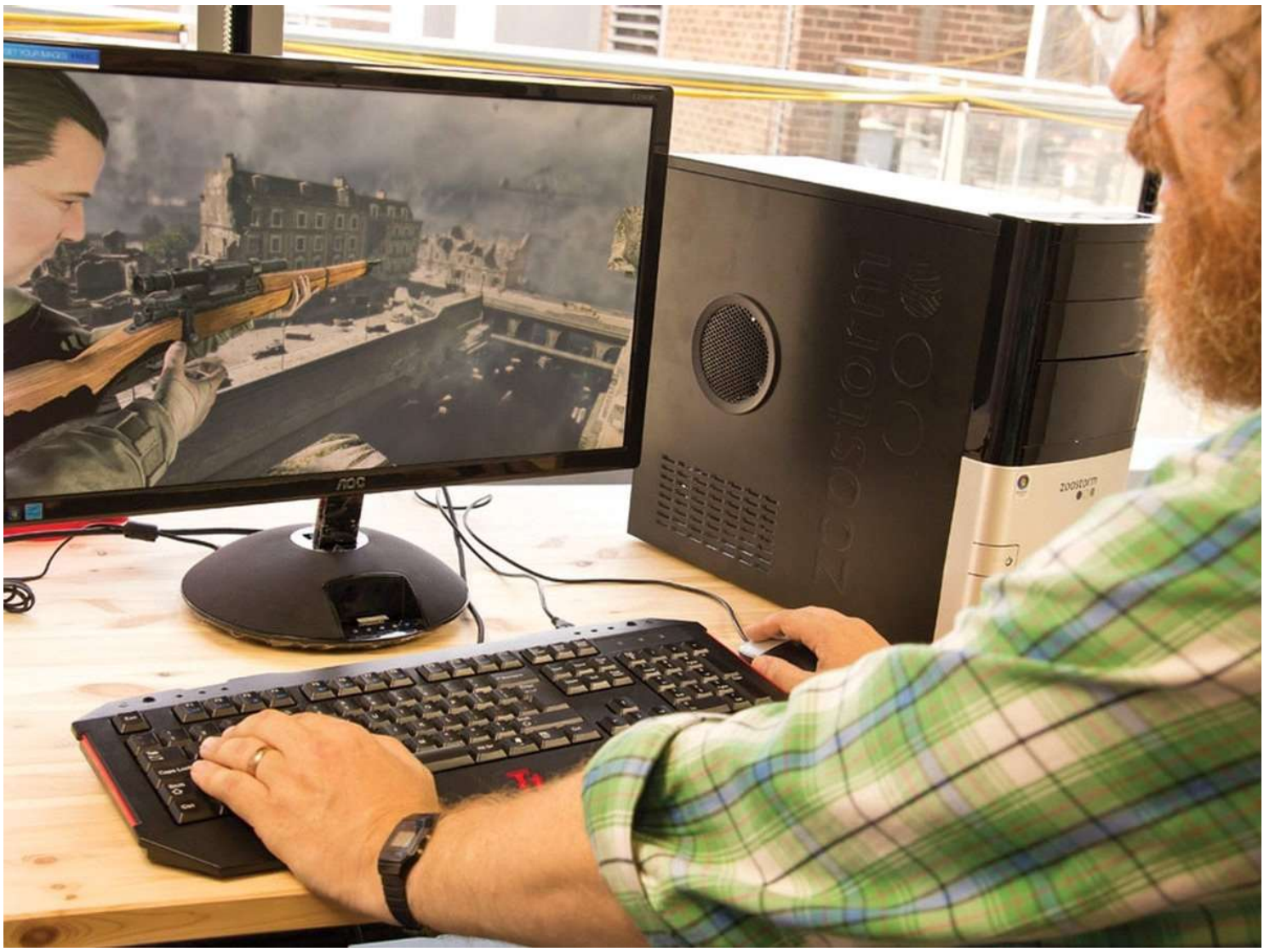
Yes i'm a Zindian. source: zindi.africa

Well, I feel I should say this. If you're from Africa, we have a new platform called **Zindi** which is like **Kaggle** but fine tuned to the Africa society. It contains data sets that are collected from African businesses and organisation. This is a big step in preventing bias in AI, and **Zindi** is a great platform.

So watch out for the Zindians, we're coming.

Okay, back to getting the data. We head over to this page on Kaggle. Here we have two options, we can either download the data and train our model locally on our Laptop or we can use Kaggle kernels which gives us more compute power, access to a GPU, and almost all the libraries pre-installed for machine learning and deep learning.

I don't know about you, but I'd rather go with Kaggle kernels unless I have a GPU.



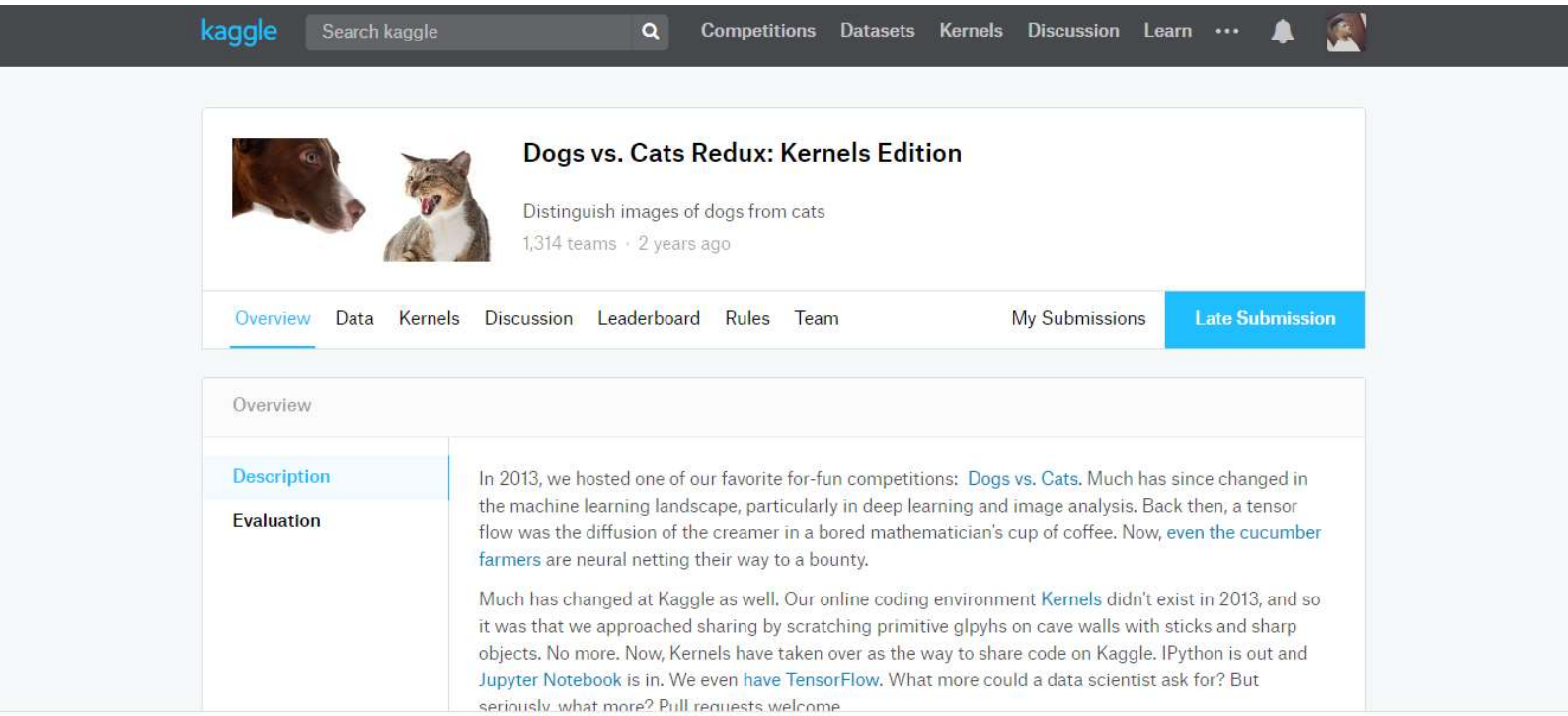
source: pcmag.com

If you can afford a GPU or you're a **gamaic** [*I totally made up that word*] and already have a GPU and you want to train your model on your PC because you

love your PC too much , then you can follow this **tutorial** or this **one** on setting up your own deep learning work station.

For the rest of the lazy-or-cannot-currently-afford-a-GP-us, let's head over to Kaggle and start this engine.

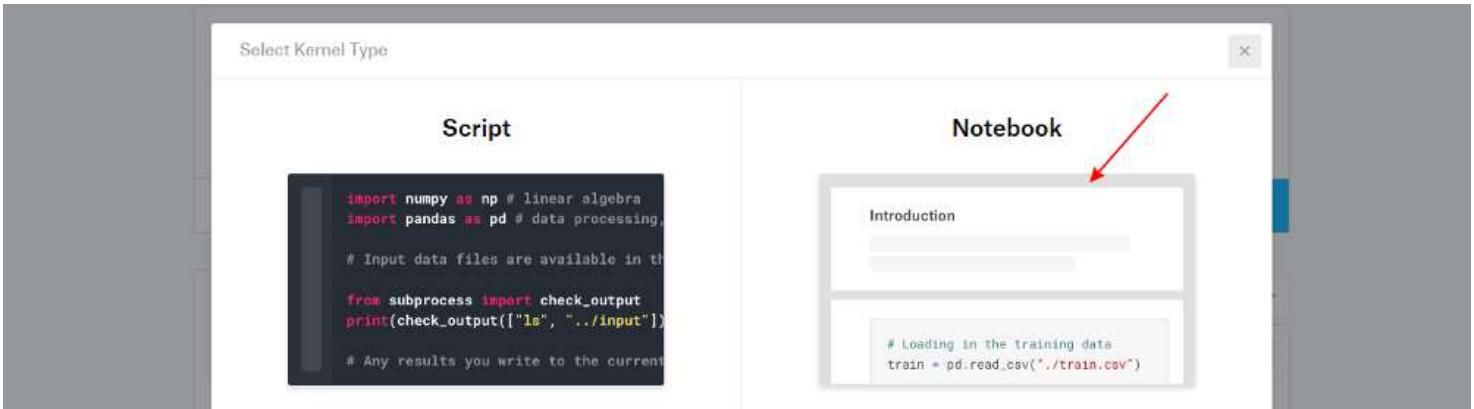
Clicking on the link, we teleport here.



cats vs dogs kernel on kaggle

Note: It is generally a good idea to read the data description when you visit a Kaggle competition page.

Next, click on **Kernels** then the blue **New Kernel** button on the top right corner, this will ask you to **Select Kernel Type.** Pick Notebook so we can do some interactive programming.



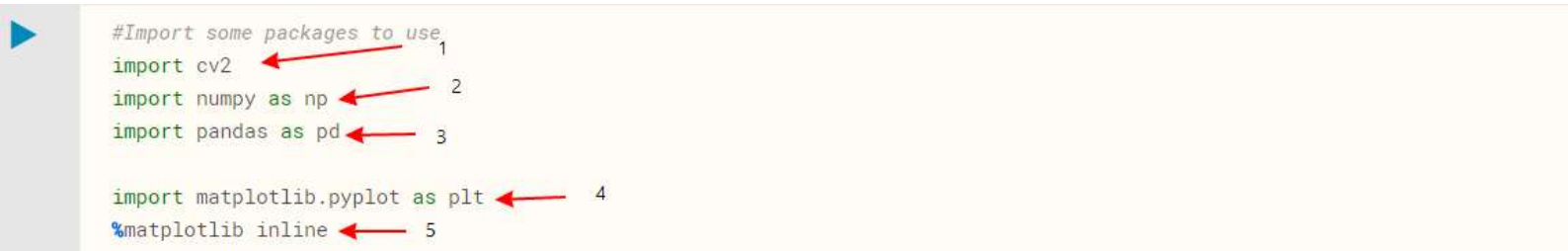
If you don't know what a Notebook is or need to refresh your ***notebooking*** skills, **here** and **here** are great sites with tutorials on Jupyter Notebook.

Clicking on Notebook creates a new private kernel for you and automatically adds the dogs vs cats dataset to your file path — *in the cloud of course*.

Give your Kernel a name and put on your super power (**GPU**) for faster compute time.

Your Notebook opens like this

Now that our kernel is ready, let's import some of the libraries we're going to use. Type the following code and press ***shift Enter*** to run the cell.



Import the following libraries

1. **cv2**
   cv2 also called **OpenCV**, is an image and video processing library available in Python and many other high level programming languages. It is used for all sorts of image and video analysis, like facial recognition and detection, license plate reading, photo editing, advanced robotic vision, optical character recognition, and a whole lot more. In this tutorial, we are going to use it in reading and resizing our Images.

2. **NumPy** is the most popular mathematical library in Python. It makes working and computing large, multi-dimensional arrays and matrices super easy and fast. It has a large collection of high-level mathematical functions to operate on these arrays.

3. **Pandas** pandas is a software library written for the Python programming language for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series.

4. **Matplotlib** is a plotting library for the Python. It can be used for plotting lines, bar-chart, graphs, histograms and even displaying Images.

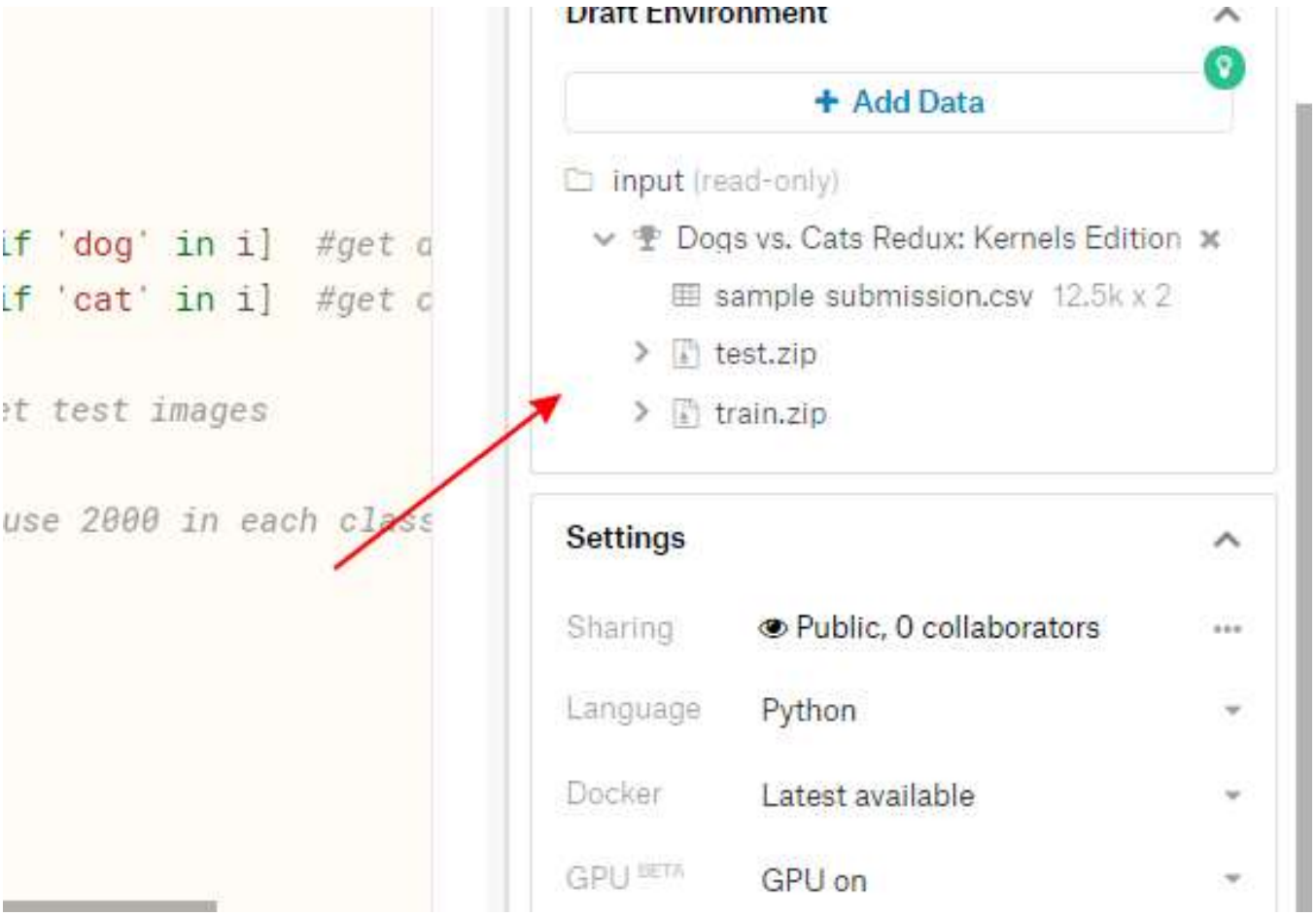
5. **%matplotlib inline** is a command that makes our plots appear in the notebook.

6. **os** is an inbuilt python package for accessing your computer and file system. It can be used to display content in directories, create new folders and even delete folders.

7. **random** will help us create random numbers which will be used when we split or shuffle our data set.

8. **gc** short for garbage collector is an important tool for manually cleaning and deleting unnecessary variables. We'll actively use this on Kaggle kernels because the free memory allocated to us may get full since we are working on Image datasets.

After importing the necessary libraries, we read our Images in to memory. The data is stored as Zip files in our kernel.



Our data

We can see three files:

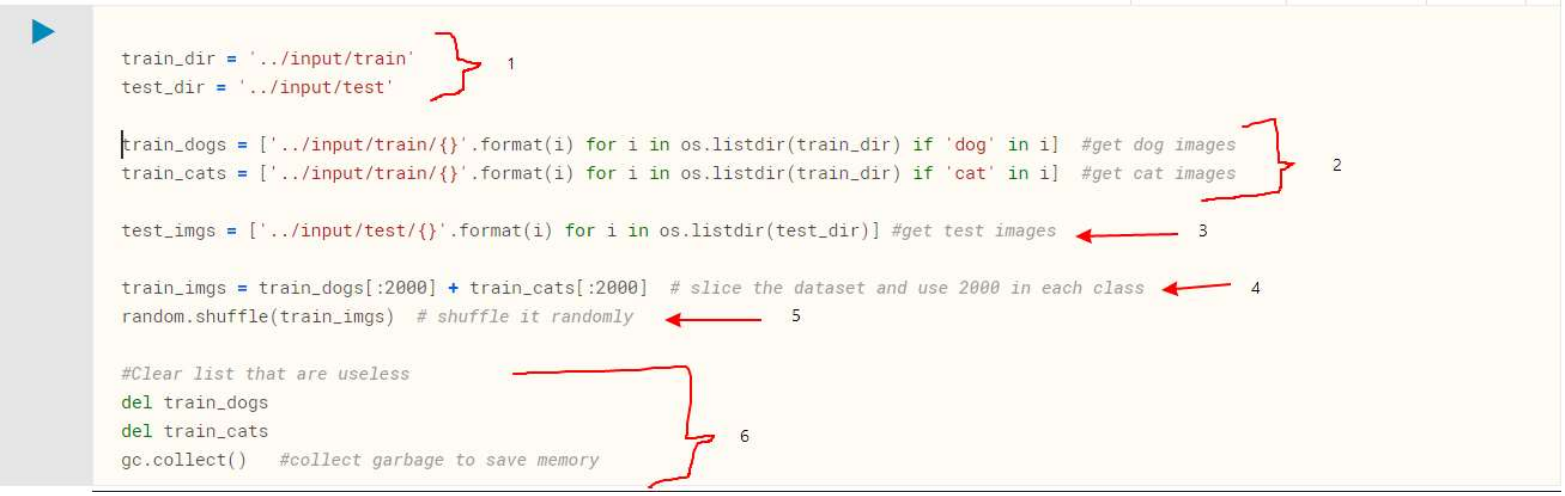
1. sample submission.csv
   This is a csv (comma seperated value) file that is used for making submission after training your model and testing it on the test files given to you. Since this competition is over, we cannot make submissions, so we'll ignore this file.

2. test.zip
   This file contains the Images we're going to test our models on after training, to know if our model has learnt to differentiate dogs from cats.

3. train.zip
   This is the food for our model. It contains the data we're going to use to teach our model what a dog or a cat looks like.

Now to access our training Images, we're going to use the **os** package which we imported earlier.

> Note: All data file paths on kaggle start with the root directory ../input. for example the file path to the train data in this kernel will be ../input/train



import our Images

1. Here we create a file path to our train and test data

2. Here we create two variables ***train_dogs*** and ***train_cats***. One for all dog images and the other for cat images. We write a list comprehension that uses the command ***os.listdir()*** to get all the images in the train data zip file and retrieve all images with *dog* in their name.
   ** We do the same for cat images.

3. We get our test images too.

4. The train data set contains a total of 25,000 images, but since we are experimenting working with a small data set and we obviously have access to little computational power, we're going to extract only 2000 images from both classes.
   ** 2000 dog images and 2000 cat images, making a training data set of 4000 images.
   So we grab the first 2000 images from the ***train_dogs*** and ***train_cats***, then concatenate them into one train set called ***train_imgs***.

5. **VERY IMPORTANT!**
   we randomly shuffle the ***train_imgs*** .

6. Here we do some cleaning. As you may have noticed, we now have ***train_imgs*** meaning ***train_dogs*** and ***train_cats*** variables are useless and taking unnecessary space. If we don't delete them, we might run out of memory when we start training our model.

Okay, lets view some images in our **train_imgs**.

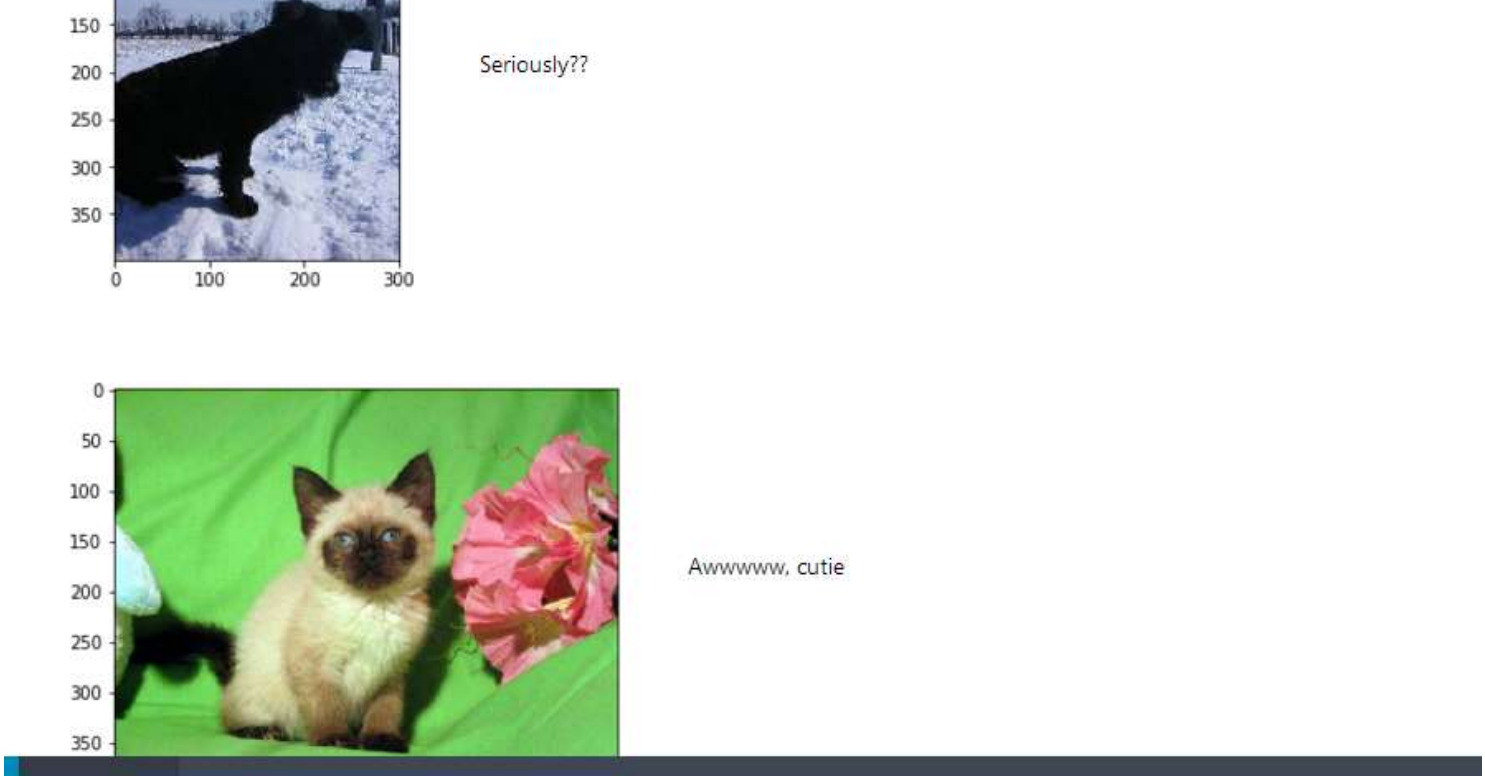
```
[41]: import matplotlib.image as mpimg                    1
      for ima in train_imgs[0:3]:
          img=mpimg.imread(ima)
          imgplot = plt.imshow(img)                       2
          plt.show()
```

lets view some cute dogs

1. Import an image plotting module from matplotlib

2. Run a for loop to plot the first three images in ***train_imgs***

Remember its a random list, but luckily when I ran the code the first three images was made up to two dogs and one cat and notice that they have different dimensions.



The last two images of plot

In the next code block we're going resize the images using the **cv2** module. First let's declare the new dimensions we want to use. Here I'm using 150 by 150 for height and width and 3 channels.

```
#Lets declare our image dimensions
#we are using coloured images.
nrows = 150
ncolumns = 150
channels = 3  #change to 1 if you want to use grayscale image
```

declare some important variables

A colored Image is made up of 3 channels, i.e 3 arrays of red, green and blue pixel values. We could use 1 channel which would read our images in gray-scale format (black and white).

Now, lets write a little function that helps also read and then resize our images to the height and width stated above.

```
#A function to read and process the images to an acceptable format for our model
def read_and_process_image(list_of_images):
    """
    Returns two arrays:
        X is an array of resized images
```

```
    y is an array of labels
    """
    X = [] # images
    y = [] # labels

    for image in list_of_images:
        X.append(cv2.resize(cv2.imread(image, cv2.IMREAD_COLOR), (nrows,ncolumns), interpolation=cv2.INTER_CUBIC))  #Read the image
        #get the labels
        if 'dog' in image:
            y.append(1)
        elif 'cat' in image:
            y.append(0)

    return X, y
```

1

2

3

helper function

1. Create a new variable ***X*** which will hold the new training set and ***y*** which will hold our training labels. (1 if the image is a dog and 0 if it is a cat)
2. we read our images one after the other and resize them with the ***cv2*** commands.
3. We append 1 to ***y*** if the image is a dog and 0 if it is a cat.

Now, let's call our function and process the images.

```
X, y = read_and_process_image(train_imgs)
```

X is now an array of image pixel values and y is a list of labels. Let's preview the first image in X.

```
X[0]
```

```
array([[[243, 243, 243],
        [246, 246, 246],
        [245, 245, 245],
        ...,
        [249, 249, 249],
        [249, 249, 249],
        [251, 251, 251]],

       [[247, 247, 247],
        [247, 247, 247],
        [249, 249, 249],
        ...,
        [249, 249, 249],
        [249, 249, 249],
        [251, 251, 251]],

       [[249, 249, 249],
```

Yea, that's what your computer calls a dog

And that my friend is a dog image. Or we can say what our computer calls a dog. Wait! how did I know its a dog? well, lets look at the corresponding value in the label list ***y***.

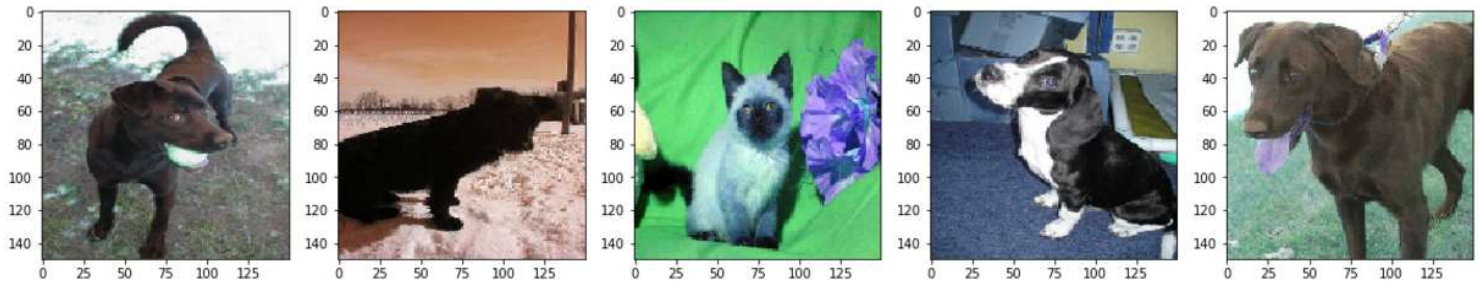
```
43]: y

array([1, 1, 0, ..., 1, 1, 0])
```

remember 1 for dog and 0 for cat

Well, remember we said lets 1 and 0 represent dogs and cats respectively.

Still don't believe me. Then, let's plot the first 5 arrays of X. We can't plot the images in X with the **mpimg** module of ***matplotlib.image*** above because these are now arrays of pixels not raw ***jpg*** files. So we should use the ***imshow***() command.

```
[34]: plt.figure(figsize=(20,10))
columns = 5
for i in range(columns):
    plt.subplot(5 / columns + 1, columns, i + 1)
    plt.imshow(X[i])
```



plot the first five images

Now that we're confident that our training set contains the appropriate images of dogs and cats, let's look at our labels. Remember we have a total of 4000 images (2000 dogs and 2000 cats), therefore our label list **y** should contain 2000 of 1s and 2000 of 0s. Let's plot this and confirm.

```
import seaborn as sns
del train_imgs
gc.collect()

#Convert list to numpy array
X = np.array(X)
y = np.array(y)

#Lets plot the label to be sure we just have two class
sns.countplot(y)
plt.title('Labels for Cats and Dogs')
```

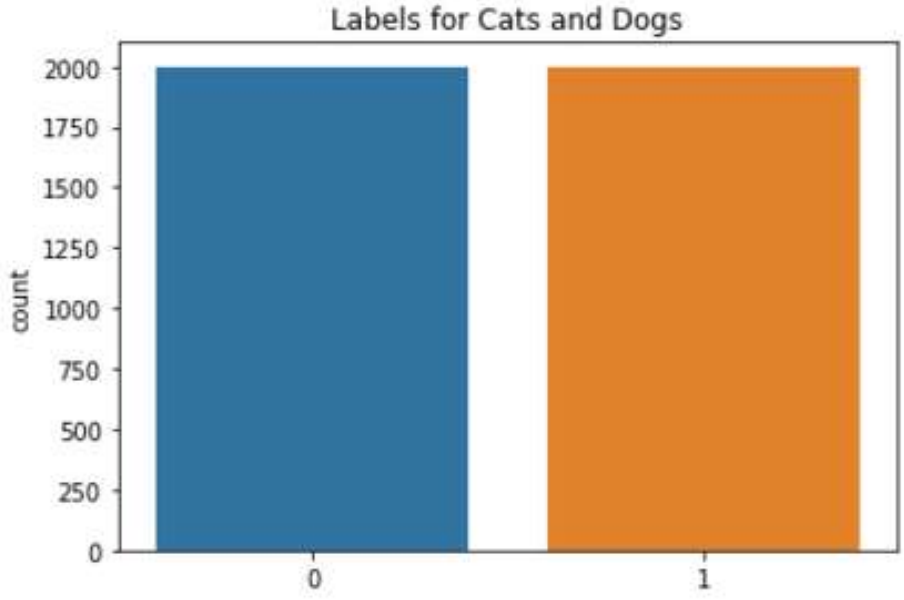
1. We import ***seaborn*** package, another plotting package built on top of ***matplotlib*** that gives very beautiful plots.

2. Remember we are neat people and we don't wait for the garbage man to do the cleaning for us. So we delete the ***train_imgs***, since it has already been converted to an array and saved in ***X***.

3. X and y are currently of type list (list of python array), we will convert these to ***numpy*** array so we can feed it into our model.

4. Plot a colorful diagram to confirm the number of classes in our y label variable

```
Text(0.5,1,'Labels for Cats and Dogs')
```



bar count of labels

Great! we have 2000 classes of both dogs and cats. Let's move on.

Next let's check the shape of our data. Always check and confirm the shapes of your data, it is super important.

```
print("Shape of train images is:", X.shape)
print("Shape of labels is:", y.shape)
```

```
Shape of train images is: (4000, 150, 150, 3)
Shape of labels is: (4000,)
```

shape of our data

We can see that our image is a tensor of rank 4, or we could say a 4 dimensional array with dimensions 4000 x 150 x 150 x 3 which correspond to the batch size, height, width and channels respectively.

> *The shape of our image array is important for the* ***keras*** *model we're going to build. The model takes as input an array of (height, width,channels)*

Now that our data is ready (X,y) we could start training, but first we have to do something that is very important, which is to split our data into train and validation set. This is one of the most important things to do before you start training your model.

For splitting, we're going to use a handy function from a popular machine learning package in python called ***sklearn***.

```
[11]:
#Lets split the data into train and test set
from sklearn.model_selection import train_test_split                    1
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.20, random_state=2)   2

print("Shape of train images is:", X_train.shape)
print("Shape of validation images is:", X_val.shape)                    3
print("Shape of labels is:", y_train.shape)
print("Shape of labels is:", y_val.shape)
```

```
Shape of train images is: (3200, 150, 150, 3)
Shape of validation images is: (800, 150, 150, 3)
Shape of labels is: (3200,)
Shape of labels is: (800,)
```

1. Imports ***train_test_split*** from ***sklearn***
2. We tell the function we want 20% of the data to be assigned to the validation set and the other 80% to the train set.
3. Here we print the shape of the new train and validation set

Next, we're going to declare some important variables that will be used when training our model.

```
#clear memory
del X
del y
gc.collect()

#get the length of the train and validation data
ntrain = len(X_train)
nval = len(X_val)

#We will use a batch size of 32. Note: batch size should be a factor of 2.***4,8,16,32,64...***
batch_size = 32
```

1
2

1. Yes, we're still cleaning.
2. get the length of the train and validation set.

. . .

Wheeeeeew….Now its time to create our model.

We are going to use a Convolutional Neural Network (convnet) to train our model. Convets are currently the standard when it comes to Computer Vision problems. They have always outperform other types of neural network in any image problem.

New to convnets? At the end of this post, there are good links to sites where you can learn all about them.

In creating our model we're going to use **KERAS**.

According to wikipedia…

> Keras is an open source neural network library written in Python. It is capable of running on top of TensorFlow, Microsoft Cognitive Toolkit, or Theano. Designed to enable fast experimentation with deep neural networks, it focuses on being user-friendly, modular, and extensible.

First let's import the neccessary keras modules we are going to use

1. Here we import keras ***layers module*** which contains different types of layers used in deep learning such as:
   ** Convolutional layer (Mostly used in computer vision)
   ** Pooling layer (also used in computer vision)
   ** Recurrent layer (Mostly used in sequential and time series modelling)
   ** Embedding layers (Mostly used in Natural Language processing)
   ** Normalization layers
   ** and many more

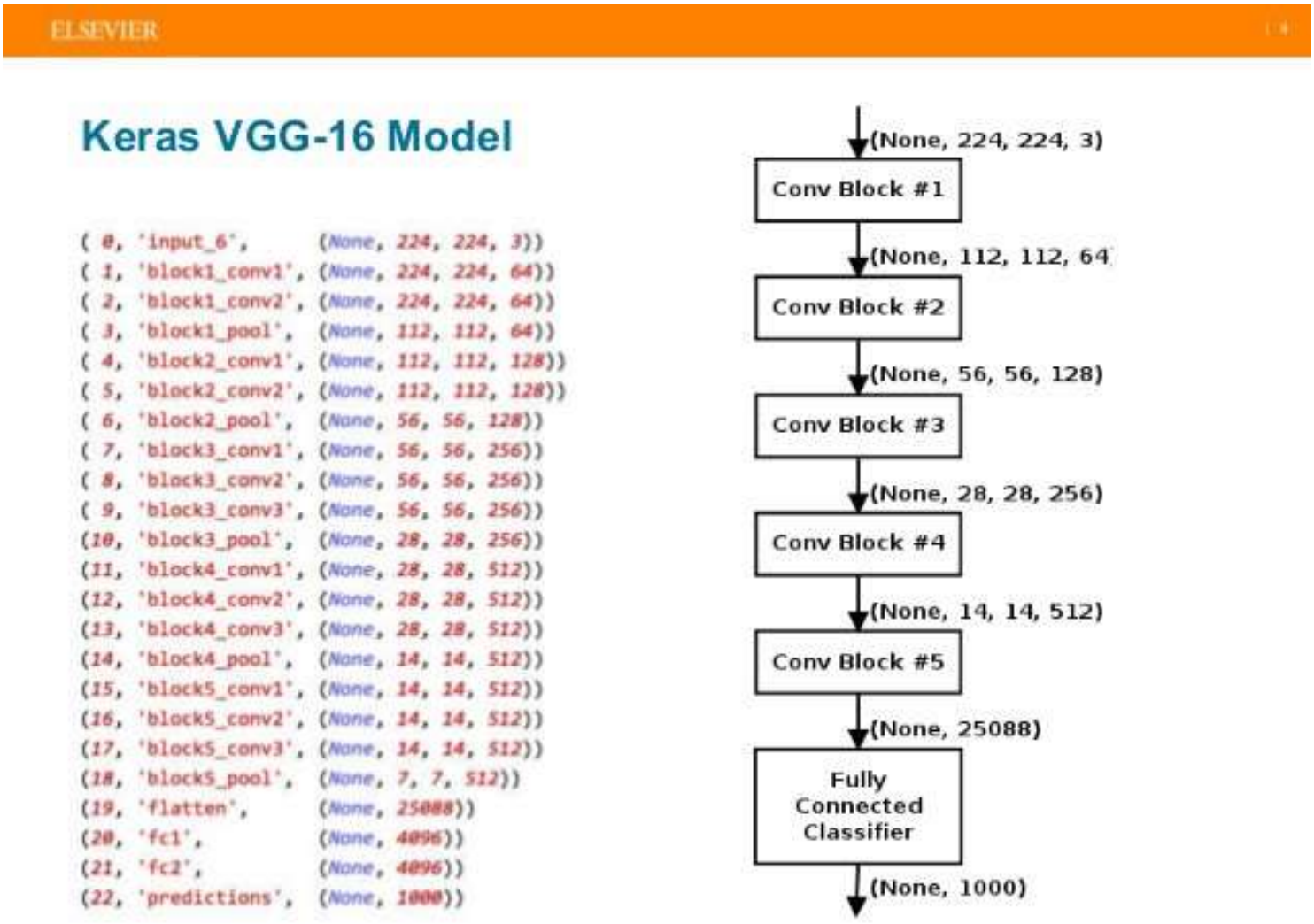
2. Here we import keras ***models*** which contains two types:
   **Sequential model which we'll be using in this tutorial and

**The model with the Functional API

3. Here we import keras ***optimizer, a module*** that contains different types of back propagation algorithm for training our model. Some of these optimizers are:
   **sgd (stochastic gradient descent)
   **rmsprop (root mean square propagation)
   **Adams
   **Adagrad
   **Adadelta

4. Here we import one of the most important function (**ImageDataGenerator**) used when working with a small data set. More on this below.

Now lets create our Network architecture. We are going to follow a popular, effective and simple architecture called the **VGGnet**



slide from Elsever

> *Network architecture is just the way we arrange our convolution layers*

We are going to use a small vggnet, but you can see below that our filter size increases as we go down layers.

32 → 64 →128 →512 — and final layer is 1

```
model = models.Sequential()                                                    1
model.add(layers.Conv2D(32, (3, 3), activation='relu',input_shape=(150, 150, 3)))   2
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))                                         3
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())                                                    4
model.add(layers.Dropout(0.5))  #Dropout for regularization                    5
model.add(layers.Dense(512, activation='relu'))                                6
model.add(layers.Dense(1, activation='sigmoid'))  #Sigmoid function at the end because we have just two classes
```

1. Here we create a sequential model. This tells keras to stack all layers sequentially.

2. Here we create the first layer by calling the **.add()** function on the model we created and pass the type of layer we want — a **Conv2D** layer. This first layer is called the **input layer** and has some important parameters we need to set.
   ** **filter size** [**32**]: This is the size of the output dimension (i.e. the number of output filters in the convolution)
   ** **kernel_size [3,3]:** This specifies the height and width of the 2D convolution window.
   ** **activation** [‘**relu**’]: We select an activation function also called non-linearity to be used by our neural network. R***eLU*** (Rectified Linear Unit) is the most common activation function used today, other variations are ***leaky*** R***eLU*** and eLU**.**
   ** **input shape [150,150,3]:** Remember the dimensions we resized our images to? 150 by 150 right? we pass that here including the channel of 3. We do not pass the first dimension of 4000 because that is the batch dimension.

3. Here we add a ***MaxPool2D*** layer. Its function is to reduce the spatial size of the incoming features and therefore helps reduce the amount of parameters and computation in the network, thereby helping to reduce overfitting.

Overfitting happens when our model memorizes the training data. The model will perform excellently at training time but fail at test time.

1. Here we add a **Flatten** layer. A conv2D layers extract and learn spatial features which is then passed to a dense layer after it has been flattened. This is the work of the flatten layer.

2. Here we add a Dropout layer with value 0.5. Dropout randomly drops some layers in a neural networks and then learns with the reduced network. This way, the network learns to be independent and not reliable on a single layer. Bottom-line is that it helps in overfitting.
   0.5 means to randomly drop half of the layers.

3. The last layer has an output size of 1 and a different activation function called ***sigmoid.*** This is because we're trying to detect if an image is a dog or a cat. i.e we want the model to output a probability of how sure an image is a dog and not a cat, that means we want a probability score where higher values means the classifier believes the image is a dog and lower values means it is a cat. The sigmoid is perfect for this because it takes in a set of numbers and returns a probability distribution in the range of 0 to 1.

We can preview the arrangement and parameter size of our convnet by calling the keras function **.summary()** on the model object.

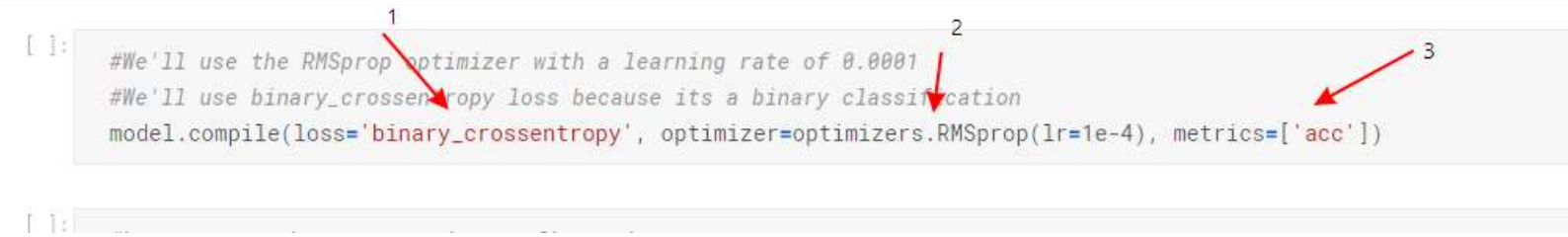
```
[73]: #Lets see our model
      model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 148, 148, 32)      896
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 74, 74, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 72, 72, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 36, 36, 64)        0
```

```
conv2d_3 (Conv2D)            (None, 34, 34, 128)       73856
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 17, 17, 128)       0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 15, 15, 128)       147584
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 7, 7, 128)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 6272)              0
_________________________________________________________________
dropout_1 (Dropout)          (None, 6272)              0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               3211776
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 513
=================================================================
Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
```

We can see the number of parameters we want to train (3 million plus) and the general arrangement of the different layers.

> The next step is to compile our model.



We pass three parameters to the model.compile() command

1. **Loss ['binary_crossentropy'] :** We specify a loss function that our optimizer will minimize. In this case since we're working with a two class problem, we use the ***binary crossentropy loss***.

2. Remember the ***optimizers*** we defined earlier? we'll going to use one of them called the ***rmsprop***. This is not a fixed choice, it is part of a process called ***hyper-parameter tuning*** which may be the difference between a world class model and a naive one.

3. Here we specify which metric we want to use in measuring our model's performance after training. We want to know if our model is doing well or not.
   Since we're doing a classification problem, the accuracy metric **(acc)** is a good choice.
   Note: The metric you use in measuring the performance of your model will depend on the type of problem you're dealing with.

. . .

Finally, before we start training our model we need to perform some Normalization. i.e scale our image pixel values to have a unit standard deviation and a mean of 0.

We'll use an important module in keras called ***ImageDataGenerator*** which perform some important functions when we're feeding Images into our model during training.

But...but what is an ImageDataGenerator?

According to the creator of keras **François Chollet**, Keras ImageDataGenerator() lets us quickly set-up python generators that automatically turn image files into preprocessed tensors that can be fed directly into models during trainng. It performs the following functions for us easily:
1. Decode the JPEG content to RGB grids of pixels.
2. Convert these into floating-point tensors.
3. Rescale the pixel values (between 0 and 255) to the [0, 1] interval (neural networks perform better with normalize data).
4. Helps us easily augment images. (An important feature we'll be using since we're training on a small data set).

Okay let's create our ImageDataGenerator object. We're going to create two generators, one for the training set and the other for our validation set.

```
[15]:
#Lets create the augmentation configuration
#This helps prevent overfitting, since we are using a small dataset
train_datagen = ImageDataGenerator(rescale=1./255,   #Scale the image between 0 and 1
                                   rotation_range=40,
                                   width_shift_range=0.2,
                                   height_shift_range=0.2,
                                   shear_range=0.2,
                                   zoom_range=0.2,
                                   horizontal_flip=True,)

val_datagen = ImageDataGenerator(rescale=1./255)  #We do not augment validation data. we only perform rescale
```

1. We pass the rescale option to the **ImageDataGenerator** object. The ***rescale=1./255*** option is a very **IMPORTANT** parameter. It normalizes the image pixel values to have zero mean and standard deviation of 1. It helps your model to generally learn and update its parameters efficiently.

2. The second set of options are Image augmentation options. They tell the **ImageDataGenerator** to randomly apply some transformation to the Image. This will help to augment our data-set and improve generalization.

3. Here we also create an **ImageDataGenerator** object for our validation set. Note: we don't do data augmentation here. We only perform ***rescale***.

Now that we have the **ImageDataGenerator** objects, lets create python generators from them by passing our train and validation set.

```
#Create the image generators
train_generator = train_datagen.flow(X_train, y_train, batch_size=batch_size)
val_generator = val_datagen.flow(X_val, y_val, batch_size=batch_size)
```

```
[ ]:
#The training part
#We train for 64 epochs with about 100 steps per epoch
history = model.fit_generator(train_generator,
                              steps_per_epoch=ntrain // batch_size,
                              epochs=64,
                              validation_data=val_generator,
                              validation_steps=nval // batch_size)
```

1. We call the ***.flow( )*** method on the data generators we created above passing in the data and label set.
   X_train and **y_train** for training then **X_val** and **y_val** for validation.
   The batch size tells the data generator to only take the specified batch(32 in our case) of Images at a time.

2. Now we train our network by calling **.fit( )** method on the **model** and passing some parameters. The first parameter is the training set *ImageDataGenerator* object [**train_generator**].

3. Here we specify the number of ***steps per epoch***. This tells our model how many images we want to process before making a gradient update to our loss function.
   A total of 3200 images divided by batch size of 32 will give us 100 steps. This means we going to make a total of 100 gradient update to our model in one pass through the entire training set.

4. An **epoch** is a full cycle or pass through the entire training set. In our case, an epoch is reached when we make **100 gradient updates** as specified by our **steps_per_epoch** parameter.
   E**pochs = 64**, means we want to go over our training data 64 times and each time we will make gradient updates 100 times.

5. We pass in our validation data generator.

6. We set the step size here too. I'm going to use the same step size as stated above.

Run the cell to start training....This will take a while. Go disturb someone on twitter...

. . .

Well after just 64 epochs, I got an accuracy of roughly 80%.

```
100/100 [==============================] - 19s 194ms/step - loss: 0.4415 - acc: 0.7922 - val_loss: 0.3984 - val_acc: 0.8050
Epoch 57/64
100/100 [==============================] - 20s 198ms/step - loss: 0.4490 - acc: 0.7869 - val_loss: 0.4476 - val_acc: 0.8025
Epoch 58/64
100/100 [==============================] - 19s 194ms/step - loss: 0.4382 - acc: 0.7944 - val_loss: 0.5588 - val_acc: 0.7200
Epoch 59/64
100/100 [==============================] - 20s 198ms/step - loss: 0.4421 - acc: 0.8016 - val_loss: 0.4373 - val_acc: 0.8163
Epoch 60/64
100/100 [==============================] - 19s 193ms/step - loss: 0.4427 - acc: 0.7919 - val_loss: 0.4172 - val_acc: 0.8213
Epoch 61/64
```

```
100/100 [==============================] - 19s 194ms/step - loss: 0.4194 - acc: 0.8084 - val_loss: 0.4530 - val_acc: 0.8063
Epoch 62/64
100/100 [==============================] - 20s 201ms/step - loss: 0.4362 - acc: 0.7897 - val_loss: 0.4613 - val_acc: 0.7850
Epoch 63/64
100/100 [==============================] - 19s 195ms/step - loss: 0.4228 - acc: 0.7987 - val_loss: 0.4410 - val_acc: 0.8063
Epoch 64/64
100/100 [==============================] - 20s 197ms/step - loss: 0.4260 - acc: 0.8003 - val_loss: 0.4272 - val_acc: 0.8025
```

screenshot of model training

> Not bad for a model we trained from scratch with very little data.

Maybe increasing the number of epochs and playing with some hyperparameters like batch size and the optimizer will help improve this score.

> I 'll leave that to you to explore.

Next, we save our model, with the simple Keras function shown below, this way we can re-use it anytime instead of training again when rerunning our notebook.

```
#Save the model

model.save_weights('model_wieghts.h5')
model.save('model_keras.h5')
```

We'll plot some graphs of the accuracy and the loss in both the train and validation set to see if we can get some insights.

```
#lets plot the train and val curve
#get the details form the history object
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)

#Train and validation accuracy
plt.plot(epochs, acc, 'b', label='Training accurarcy')
plt.plot(epochs, val_acc, 'r', label='Validation accurarcy')
```

1

2

3

```
plt.title('Training and Validation accurarcy')
plt.legend()

plt.figure()
#Train and validation loss
plt.plot(epochs, loss, 'b', label='Training loss')
plt.plot(epochs, val_loss, 'r', label='Validation loss')
plt.title('Training and Validation loss')
plt.legend()

plt.show()
```

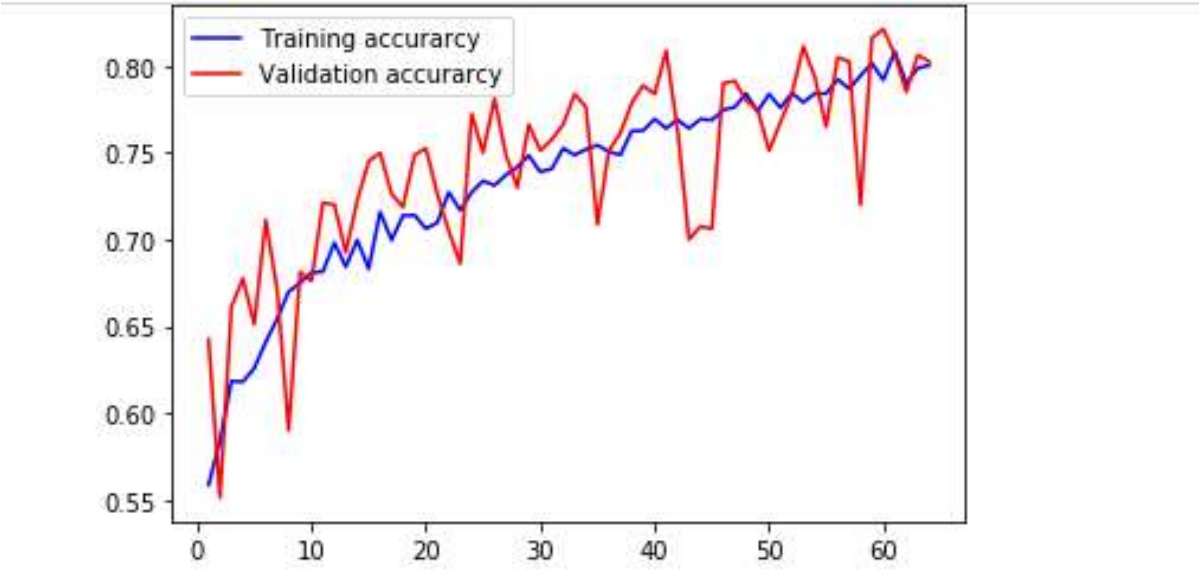
4

1. After training a keras model, it always calculates and saves the metric we specified when we compiled our model in a variable called ***history***. We can extract these values and plot them.
   Note: The history object contains all the updates that happened during training.

2. Here we simply get the size of our ***epoch*** from the number of values in the ***'acc'*** list.

3. Here we plot the accuracy against the epoch size.

4. Here we plot the loss against the epoch size.

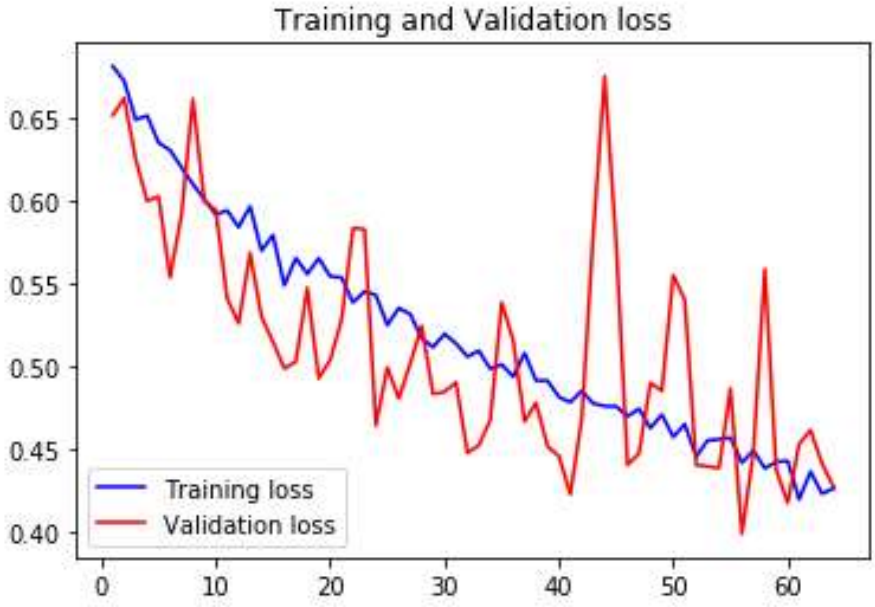


Plot of accuracy against epochs

So what can we take away from this plot?

1. First thing to note is that we're not overfitting as the train and validation accuracy are pretty close and following each other.

2. We can also notice that the accuracy keeps increasing as the epoch increases, giving us the intuition that increasing the epoch size will likely give us a higher accuracy.



Plot of loss against epochs

We're still not overfitting as both train and validation loss are trending down closely just like the accuracy plot above and the loss will likely go lower if we increase the epoch size..

> So there, you've got some intuition. Now try increasing the epoch size and play around with some hyperparameters.

Before I conclude this tutorial we're going to test our model on some Images from the test set.

```
X_test, y_test = read_and_process_image(test_imgs[0:10]) #Y_test in this case will be empty.
x = np.array(X_test)
test_datagen = ImageDataGenerator(rescale=1./255)
```

code to pre-process the test Images

We perform the same pre-processing we did in the train and validation set.

1. We read and convert the first 10 images in our test set to a list of array.
   Note: y_test will be empty because the test set has no label.
2. We convert the list of array to one big numpy array.
3. We create a test **ImageDataGenerator** and perform normalization only.
   Note: We do not augment the test set .

Now we'll create a simple for loop, that iterates over the Images from the generator to make predictions. Then we'll plot the results.

```
[34]:
i = 0
text_labels = []
plt.figure(figsize=(30,20))
for batch in test_datagen.flow(x, batch_size=1):
    pred = model.predict(batch)
    if pred > 0.5:
        text_labels.append('dog')
    else:
        text_labels.append('cat')
    plt.subplot(5 / columns + 1, columns, i + 1)
    plt.title('This is a ' + text_labels[i])
    imgplot = plt.imshow(batch[0])
    i += 1
    if i % 10 == 0:
        break
plt.show()
```

1. Create a list to hold the labels we are going to generate.
2. We set the figure size of the images we're going to plot.

3. Here we make a prediction on that particular image provided by the **ImageDataGenerator** by calling the **.predict( )** method on our trained model.

4. The ***pred*** variable is a probability of how sure the model is that the current image is a dog.
   Since we gave dogs a label of 1, a high probability — at least greater than average 0.5 — means our model is very confident that the image is a Dog, otherwise it is a cat.
   So we simply create an ***if -else*** statement that appends the string '***Dog***' if the probability is greater than 0.5 otherwise it appends '***cat***' to the **text_label.**
   We do this so we can add a title to the image when we plot it.

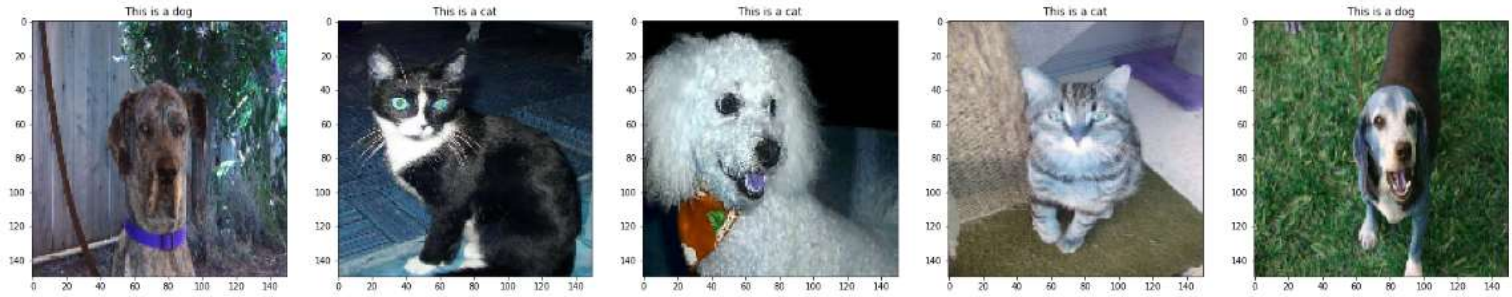
5. Here we add a subplot so we can plot multiple images.

6. Here we add the predicted class as a title to the image plot.

7. We finally plot the image.

. . .

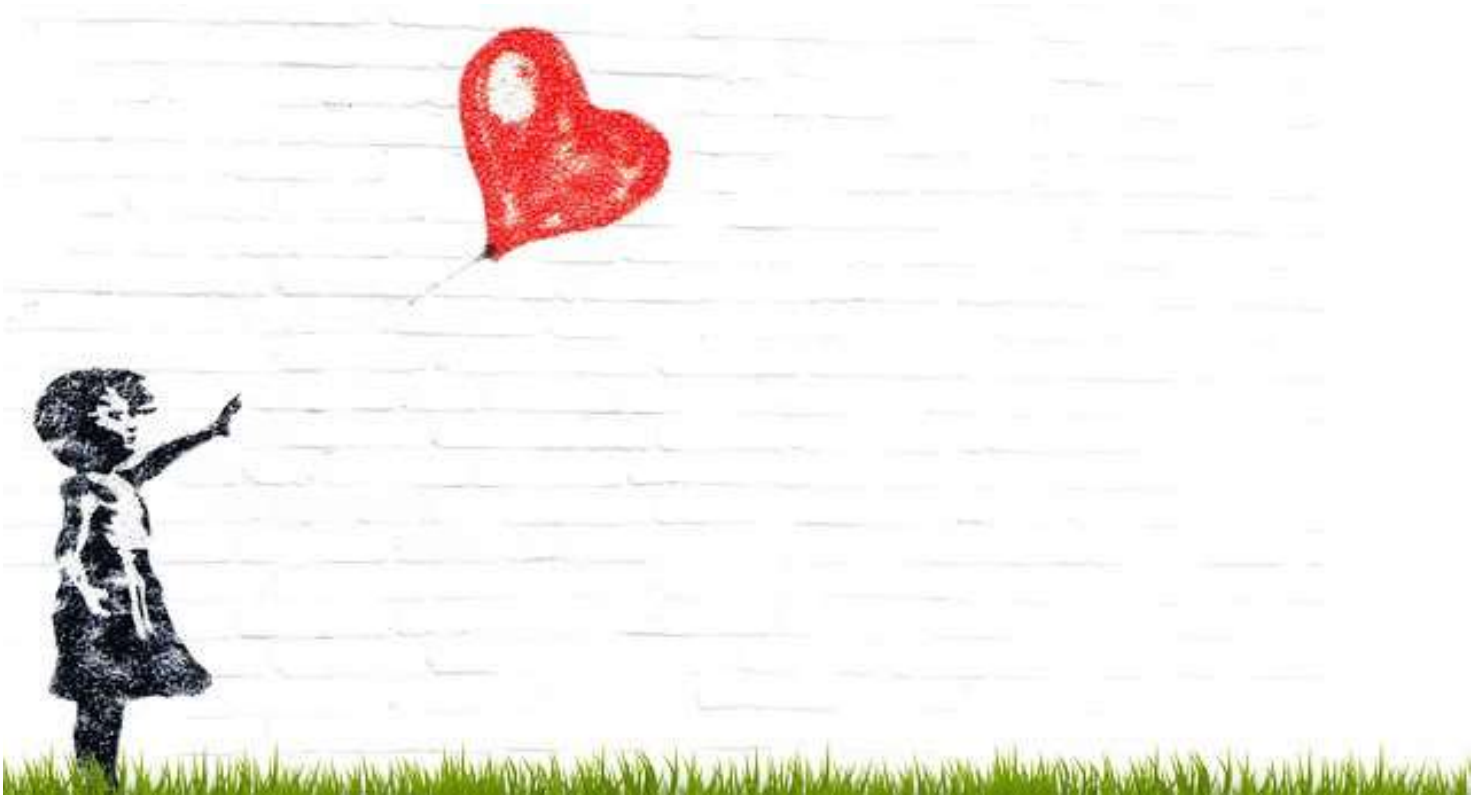
Let's see how our model performed on previously unseen images.



Well....our model got one wrong from five images. I didn't say it was at its best....at least

not yet.

Wheeeew…. it's been a long post, but I guess it was worth it. In this **next tutorial** we improved our model to achieve an accuracy of about 95% by using a ***pre-trained network.*** *A process called* ***Transfer Learning.***



source: pixabay.com

Well, bye for now and happy coding.

Link to this notebooks on Github**.**

Some amazing post and write-ups on CNN and keras.

- CS231n Convolutional Neural Networks for Visual Recognition.
- Deep learning with python by Francois Chollet the creator of Keras.
- Covolution Neural network basics.
- A great medium post on CNN.

- An Intuitive guide to CNN | medium post.
- Keras tutorial.

> Questions, comments and contributions are welcome.

> ***Connect with me on twitter .***
>
> ***Connect with me on Instagram.***

Machine Learning | Keras | Deep Learning | Image Detection | Kaggle