# Cats Versus Dogs - Image Classification Using CNNs

In this article, we will use **convolutional neural networks** (**CNNs**) to create a classifier that can predict whether a given image contains a cat or a dog.

This project marks the first in a series of projects where we will use neural networks for image recognition and computer vision problems. As we shall see, neural networks have proven to be an extremely effective tool for solving problems in computer vision.

We will cover the following topics:

- Motivation for the problem that we're trying to tackle: image recognition
- Neural networks and deep learning for computer vision
- Understanding convolution and max pooling
- Architecture of CNNs
- Training CNNs in Keras
- Using transfer learning to leverage on a state-of-the art neural network
- Analysis of our results

# Technical requirements

The key Python libraries required for this project are:

- matplotlib 3.0.2
- Keras 2.2.4
- Numpy 1.15.2
- Piexif 1.1.2

The following files are located in the code:

- `main_basic_cnn.py`: This is the main code for the basic CNN
- `main_vgg16.py`: This is the main code for the VGG16 network
- `utils.py`: This file contains auxiliary utility code that will help us in the implementation of our neural network
- `visualize_dataset.py`: This file contains the code for exploratory data analysis and data visualization
- `image_augmentation.py`: This file contains sample code for image augmentation

To run the code for the neural network, simply execute the `main_basic_cnn.py` and `main_vgg16.py` files:

# Computer vision and object recognition

The goal of the engineering discipline of computer vision is to develop software that can interpret pictures. An urban legend states that the origins of computer vision may be traced back to the 1960s, when MIT Professor Marvin Minsky gave a group of undergraduate students a summer project that involved connecting a camera to a computer and having the machine describe what it saw. It was anticipated that the project would be finished in a single summer. It goes without saying that it wasn't finished that summer since computer vision is a very complicated topic that continues to be researched and developed by scientists today.

Computer vision made slow development in its early years. Researchers began by developing algorithms to identify edges, lines, and forms in photos in the 1960s. Over the next few decades, computer vision developed into a number of subfields. Researchers in computer vision focused on a variety of topics, including object identification, computer photometry, signal processing, and image processing.

`

One of the most common uses of computer vision is probably object recognition. For a very long period, researchers had worked on object recognition. Early object recognition researchers encountered the difficulty of teaching computers to detect items due to their dynamic appearance. In order to recognize objects, early computer vision researchers concentrated on template matching, but they frequently encountered challenges because of changes in illumination, angle, and occlusions..

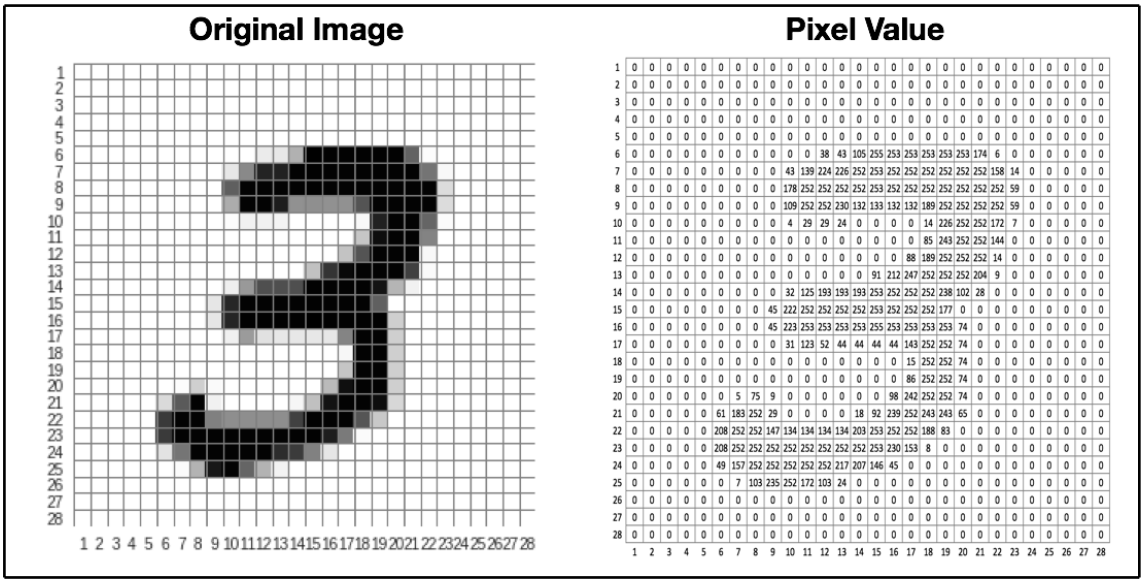
The latest breakthroughs in deep learning and neural networks have driven an exponential growth in the field of object recognition. The **ImageNet Large Scale Visual Recognition Challenge** (ILSVRC) was won in 2012 by Alex Krizhevsky et al. with a large lead over other competitors. The AlexNet architecture, which is a CNN, was the winning concept put out by Alex Krizhevsky and colleagues for object recognition. AlexNet represented a major advancement in object recognition technology. Neural networks have now taken the lead in tasks relating to computer vision and object recognition. You will build a CNN in this project that is akin to AlexNet.

The development of AI as we know it today was aided by the advancement in object recognition. Facebook automatically tags and categorizes pictures of you and your friends using face recognition technology. Facial recognition technology is used by security systems to identify people of interest and intruders. Autonomous vehicles employ object recognition technology to identify road signs, people, and other things. Despite having rather distinct origins, object identification, computer vision, and artificial intelligence are being increasingly seen by society as a single field.

# Digital images as neural network input

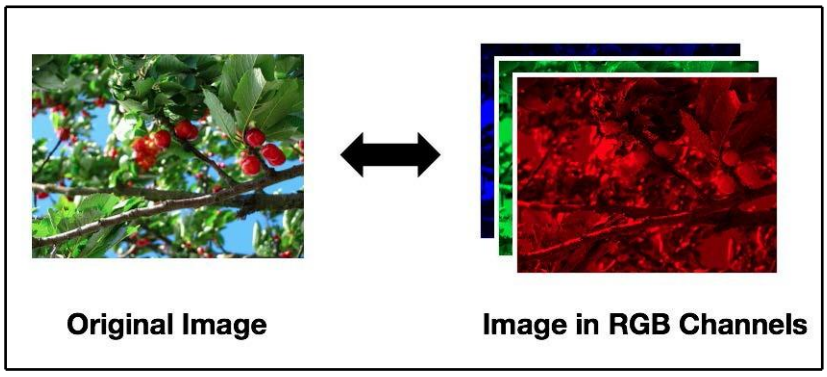
Neural networks require numerical inputs. All digital images are numerical in nature! Consider a 28 x 28 image of a handwritten digit 3, as shown in the following screenshot. Let's assume for now that the image is in grayscale (black and white). If we look at the intensity of each pixel that makes up the image, we can see that certain pixels are totally white, while some pixels are gray and black. In a computer, white pixels are represented with the value 0 and black pixels are represented with a value of 255. Everything else in between white and black (that is, shades of gray) has a value in between 0 and 255. Therefore, digital images are essentially numerical data and neural networks are perfectly able to learn from them:

`




What about color images? Color images are simply images with three channels—red, green, and blue (commonly known as RGB). The pixel values in each channel then represent the red intensity, green intensity, and blue intensity. Another way to think about it is, that for a pure red image, the pixels value will be 255 in the red channel and 0 for the green and blue channels.

The following depicts a color image, and the separation of the color image into its RGB channels. Notice how a color image is stacked in a three-dimensional manner. In contrast, a grayscale image has only two dimensions:




`

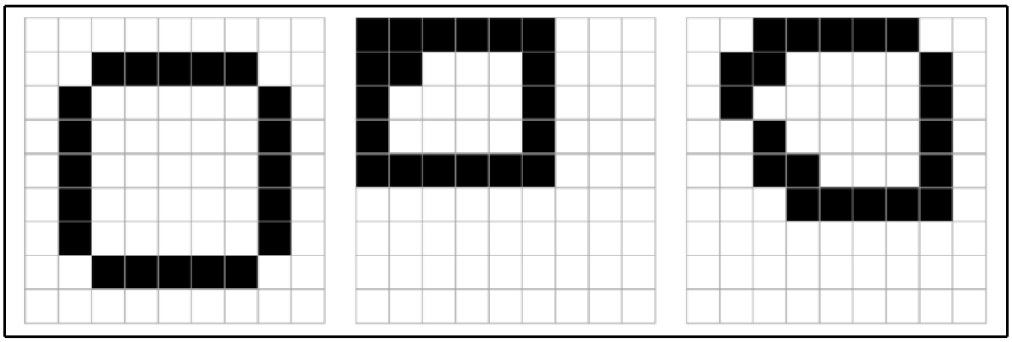
# Building blocks of CNNs

Image classification faces challenges due to the dynamic appearance of objects, such as the various breeds of cats and dogs. Humans can easily differentiate between cats and dogs by identifying key features like size, ears, and snout. Convolutional neural networks can be trained to mimic this approach by looking for these features in images, enabling accurate classification.

# Filtering and convolution

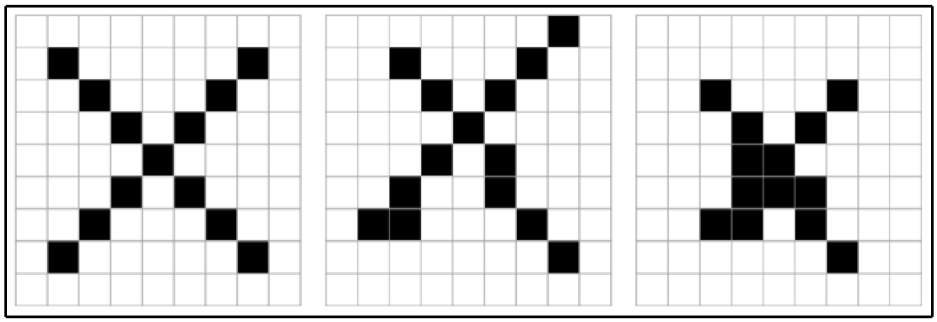
Before diving into convolution, it's essential to grasp the concept of filtering. Imagine we have a 9x9 image and want to classify it as an X or an O. The diagram below displays sample input images, with a well-drawn O on the left and poorly drawn Os on the right

:

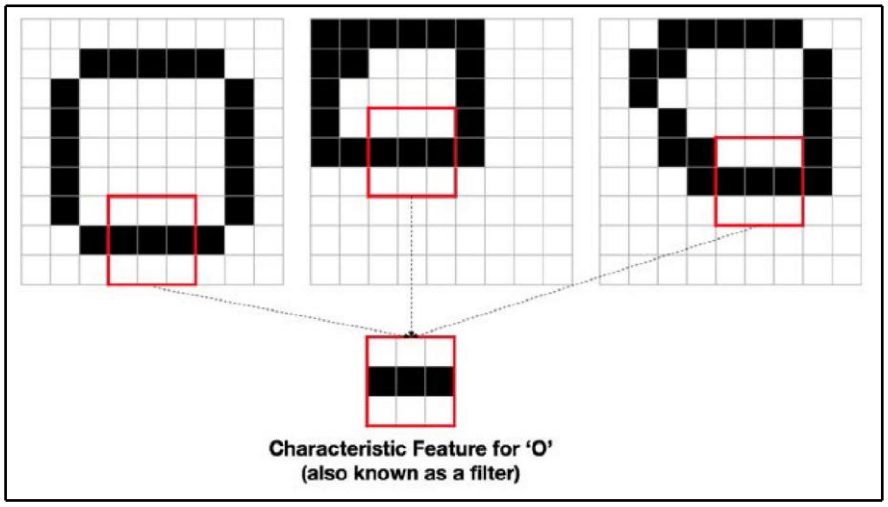


`

A perfectly drawn X is shown in the leftmost box, while the other two boxes show badly drawn Xs:

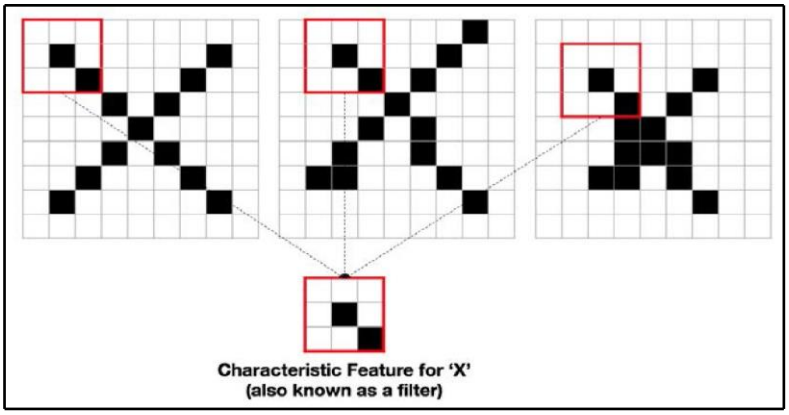


In real-world scenarios, we may encounter imperfectly drawn figures, but humans can easily distinguish between Os and Xs even in poorly drawn cases. This is because Os typically have flat horizontal edges, while Xs have diagonal lines.This distinguishing feature makes it easy for us to differentiate between the two shapes.

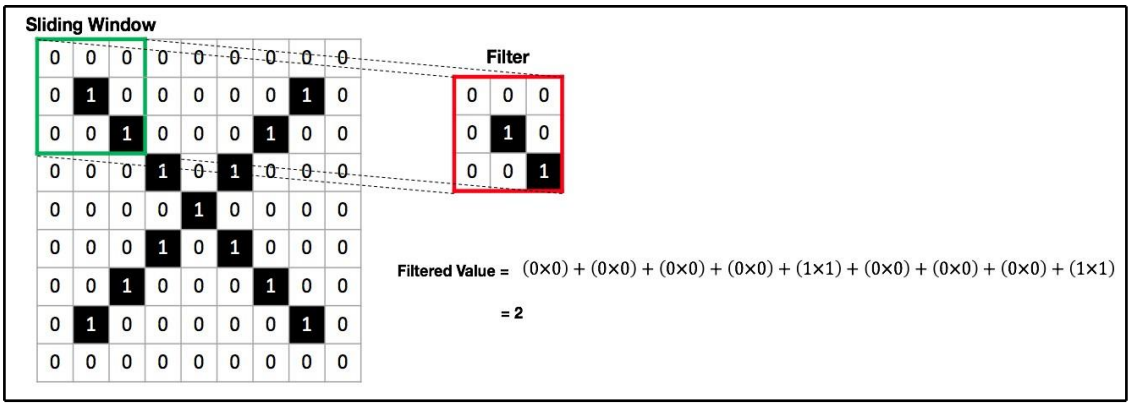



`

And the following diagram depicts one such characteristic feature for Xs:

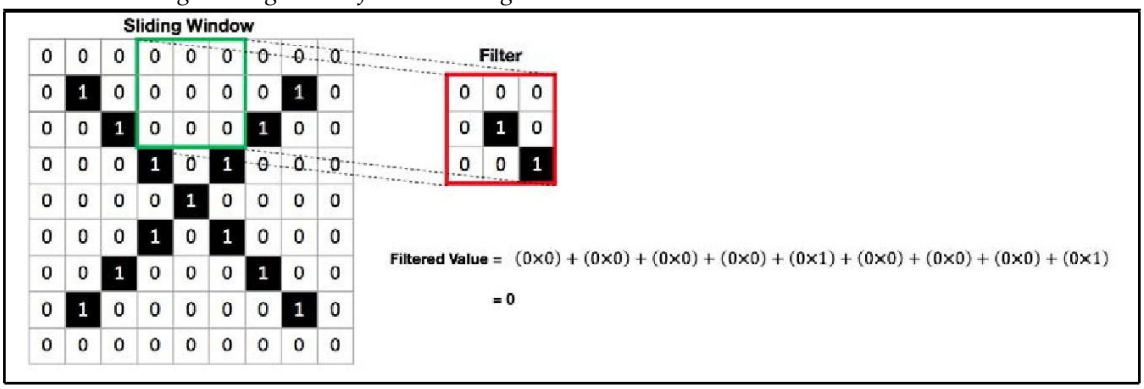



In this scenario, a 3x3 filter is used as a characteristic feature to identify specific patterns in an image. By sliding the filter across the image and performing element-wise multiplication, we can detect the presence of the feature. For example, if the filter matches a horizontal edge (representing the letter O), the image likely contains an O. This process involves a brute force search through each pixel to find matches. The filtering operation starts at the top left corner, where a match yields an output of 2. The diagram below illustrates this operation, assuming simplified pixel intensity values of 0 or 1.




Next, we slide the window toward the right to cover the next 3 x 3 section in the image. The following diagram shows the filtering operation on the next 3 x 3 section:

`




Convolution is the process of sliding a window through an image to calculate filtered values, performed by the convolutional layer in a neural network. This helps identify characteristic features in images, enabling intelligent object recognition. In training a neural network, the filters are automatically learned, unlike in the manual filter creation example. Just like tuning weights in a fully connected layer, the weights in a convolutional layer are adjusted during training to improve recognition accuracy..

Finally, take notice that a convolutional layer has two primary hyperparameters:

Number of filters: There was just one filter used in the example above.

To identify more distinguishing traits, we can apply more filters.

Filter size: A 3 bx 3 filter size was utilized in the example above.

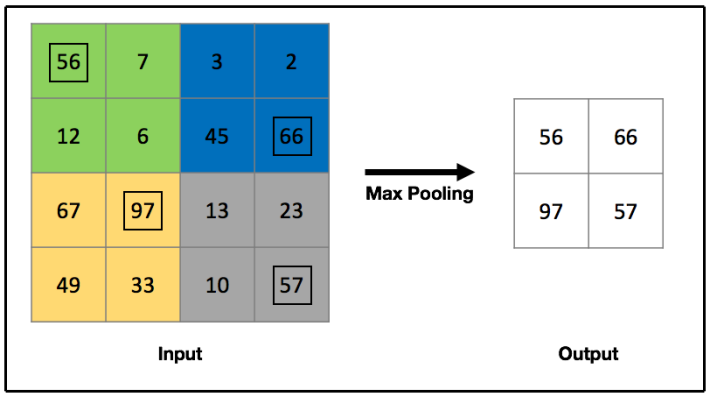
To depict more prominent distinctive traits, we might adjust the filter size.

When we build our neural network later in the chapter, we will go into more depth about these hyperparameters.

`

# Max pooling

The max pooling layer merely examines each input subset that is sent to it and eliminates all values other than the largest one in each subset. Let's see what this implies by looking at an example. Assume that we are using a 2 x 2 max pooling layer and that our input to the max pooling layer is a 4 x 4 tensor (a tensor is just an n-dimensional array, such as those output by a convolutional layer).




As we can see from the preceding diagram, **Max Pooling** simply looks at each 2 x 2 region of the input, and discards all but the maximum value in that region (boxed up in the preceding diagram). This effectively halves the height and width of the original input, reducing the number of parameters before passing it to the next layer.

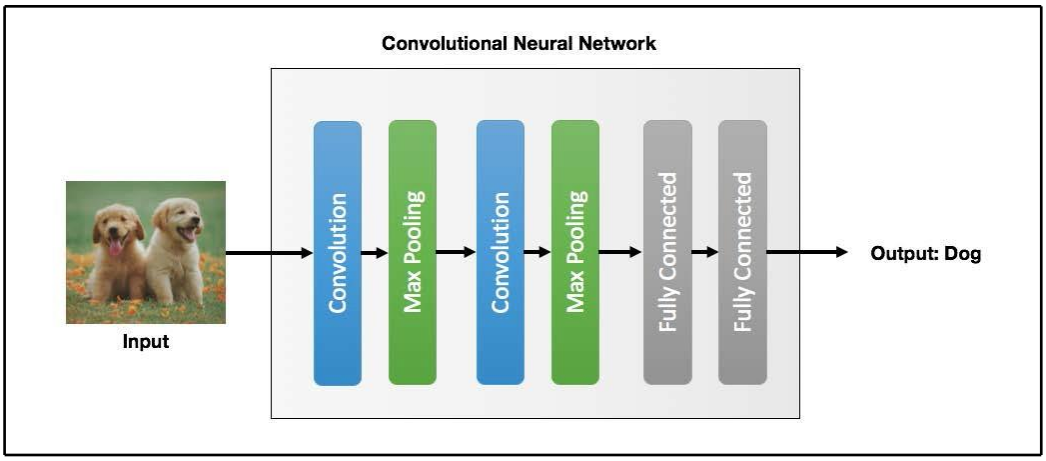
`

# Basic architecture of CNNs

We have seen the basic building blocks of CNNs in the previous section. Now, we'll put these building blocks together and see what a complete CNN looks like.

CNNs are almost always stacked together in a block of convolution and pooling pattern. The activation function used for the convolution layer is usually ReLU.

The following diagram shows the first few layers in a typical CNN, made up of a series of convolution and pooling layers:




The final layers in a CNN will always be **Fully Connected** layers (dense layers) with a sigmoid or softmax activation function. Note that the sigmoid activation function is used for binary classification problems, whereas the softmax activation function is used for multiclass classification problems.

In CNNs, the early layers learn and extract the characteristic features of the data they are trying to predict. For example, we have seen how a convolutional layer learns the characteristic spatial features of Os and Xs. The convolutional layers then pass this information on to the fully connected layers, which then learn how to make accurate predictions, just like in an MLP.

`

CNNs have progressed and improved exponentially in the past few years. In fact, recent CNNs can outperform humans at certain image recognition tasks. The recurring theme in recent years is to use innovative techniques to improve model performance, while preserving the model complexity. Clearly, the speed of the neural network is just as important as the accuracy.

# The cats and dogs dataset

Now that we understand the theory behind CNNs, let's dive into data exploration. The cats and dogs dataset is provided by Microsoft.

Let's plot the images to better understand the kind of data we're working with. To do that, we can simply run the following code:

```
from matplotlib import pyplot as plt
import os
import random

# Get list of file names
_, _, cat_images = next(os.walk('Dataset/PetImages/Cat'))

# Prepare a 3x3 plot (total of 9 images)
fig, ax = plt.subplots(3,3, figsize=(20,10))

# Randomly select and plot an image
for idx, img in enumerate(random.sample(cat_images, 9)):
    img_read = plt.imread('Dataset/PetImages/Cat/'+img)
    ax[int(idx/3), idx%3].imshow(img_read)
    ax[int(idx/3), idx%3].axis('off')
    ax[int(idx/3), idx%3].set_title('Cat/'+img)
plt.show()
```

`

We'll see the following output:


We can make some observations about our data:

- The images have different dimensions.
- The subjects (cat/dog) are mostly centered in the image.
- The subjects (cat/dog) have different orientations, and they may be occluded in the image. In other words, there's no guarantee that we'll always see the tail of the cat in the image.

Now, let's do the same for the dog images:

```
# Get list of file names
_, _, dog_images = next(os.walk('Dataset/PetImages/Dog'))

# Prepare a 3x3 plot (total of 9 images)
fig, ax = plt.subplots(3,3, figsize=(20,10))

# Randomly select and plot an image
for idx, img in enumerate(random.sample(dog_images, 9)):
    img_read = plt.imread('Dataset/PetImages/Dog/'+img)
    ax[int(idx/3), idx%3].imshow(img_read)
    ax[int(idx/3), idx%3].axis('off')
    ax[int(idx/3), idx%3].set_title('Dog/'+img)
plt.show()
```

`

We'll see the following output:


# Managing image data for Keras

One common problem encountered in neural network projects for image classification is that most computers do not have sufficient RAM to load the entire set of data into memory. Even for relatively modern and powerful computers, it would be far too slow to load the entire set of images into memory and to train a CNN from there.

To alleviate this problem, Keras provides a useful `flow_from_directory` method that takes as an input the path to the images, and generates batches of data as output. The batches of data are loaded into memory, as required before model training. This way, we can train a deep neural network on a huge number of images without worrying about memory issues. Furthermore, the `flow_from_directory` method allows us to perform image preprocessing steps such as resizing and other image augmentation techniques by simply passing an argument. The `flow_from_directory` method would then perform the necessary image preprocessing steps in real time before passing the data for model training.

`

To do all these, there are certain schemas for file and folder management that we must abide by, in order for `flow_from_directory` to work. In particular, we are required to create subdirectories for training and testing data, and within the training and testing subdirectories, we need to further create one subdirectory per class. The following diagram illustrates the required folder structure:

```
/data
    … /train
        … /cat
            … 0.jpg
            … 1.jpg
            … 2.jpg
        … /dog
            … 0.jpg
            … 1.jpg
            … 2.jpg
    … /test
        … /cat
            … 0.jpg
            … 1.jpg
            … 2.jpg
        … /dog
            … 0.jpg
            … 1.jpg
            … 2.jpg
```

The `flow_from_directory` method would then infer the class of the images from the folder structure.

The raw data is provided in a `Cat` and `Dog` folder, without separation of training and testing data. Therefore, we need to split the data into a `Train` and `Test` folder as per the preceding schema. To do that, we need to perform the following steps:

1. Create `/Train/Cat`, `/Train/Dog`, `/Test/Cat`, and `/Test/Dog` folders.
2. Randomly assign 80% of the the images as train images and 20% of the images as test images.
3. Copy those images into the respective folders.

`

We have provided a helper function in `utils.py` to do these steps. We simply need to invoke the function, as follows:

```
from utils import train_test_split

src_folder = 'Dataset/PetImages/'
train_test_split(src_folder)
```

Our images are now placed in the appropriate folders for Keras.

# Image augmentation

Keras provides a handy `ImageDataGenerator` class to help us easily perform image augmentation. Let's create a new instance of the class: `image_generator`

As we can see from the code, there are several arguments that we can provide to the `ImageDataGenerator` class. Each of the arguments control how much of a modification is done to the existing image. We should avoid extreme transformations, as those extremely distorted images do not represent images from the real world and may introduce noise into our model.

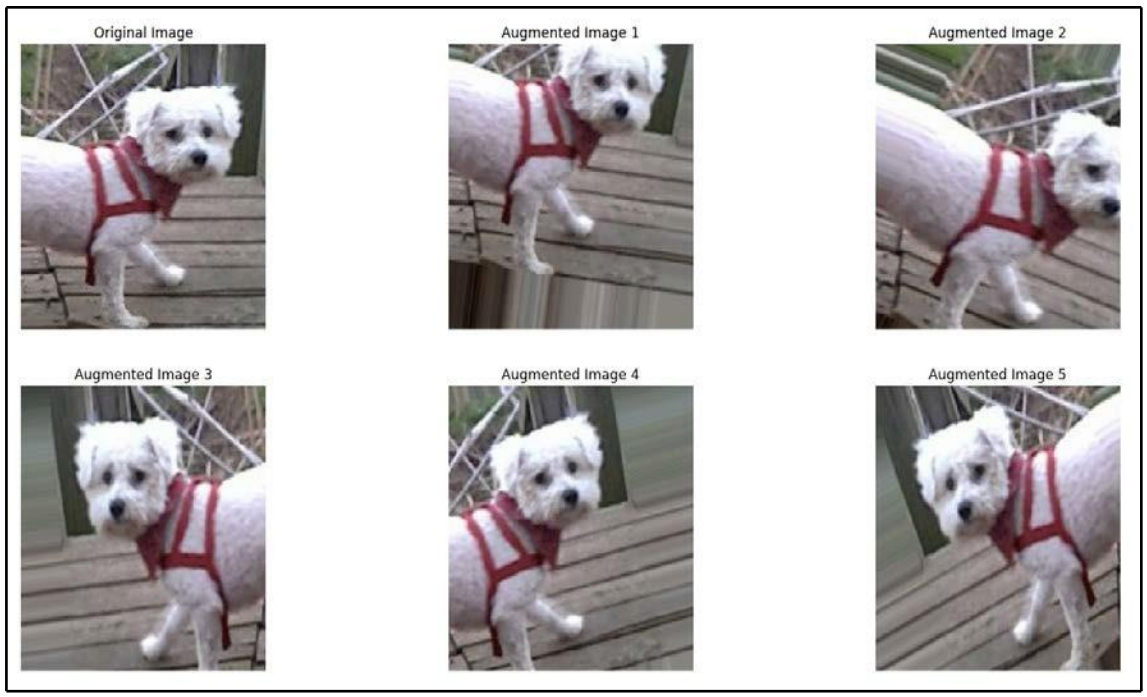
Next, let's use it to augment a randomly selected image from the `/Train/Dog/` folder. Then, we can plot it to compare the augmented images with the original image. We can do this by running the following code:

```
fig, ax = plt.subplots(2,3, figsize=(20,10))
all_images = []

_, _, dog_images = next(os.walk('Dataset/PetImages/Train/Dog/'))
random_img = random.sample(dog_images, 1)[0]
random_img = plt.imread('Dataset/PetImages/Train/Dog/'+random_img)
all_images.append(random_img)

random_img = random_img.reshape((1,) + random_img.shape)
sample_augmented_images = image_generator.flow(random_img)

for _ in range(5):
    augmented_imgs = sample_augmented_images.next()
    for img in augmented_imgs:
        all_images.append(img.astype('uint8'))

for idx, img in enumerate(all_images):
    ax[int(idx/3), idx%3].imshow(img)
    ax[int(idx/3), idx%3].axis('off')
```

`

```
        if idx == 0:
            ax[int(idx/3), idx%3].set_title('Original Image')
        else:
            ax[int(idx/3), idx%3].set_title('Augmented Image {}'.format(idx))

    plt.show()
```

We'll see the following output:




As we can see, each augmented image is randomly shifted or rotated by a certain amount as controlled by the arguments passed into the `ImageDataGenerator` class. These augmented images will provide supplemental training data for our CNN, increasing the robustness of our model.

# Model building

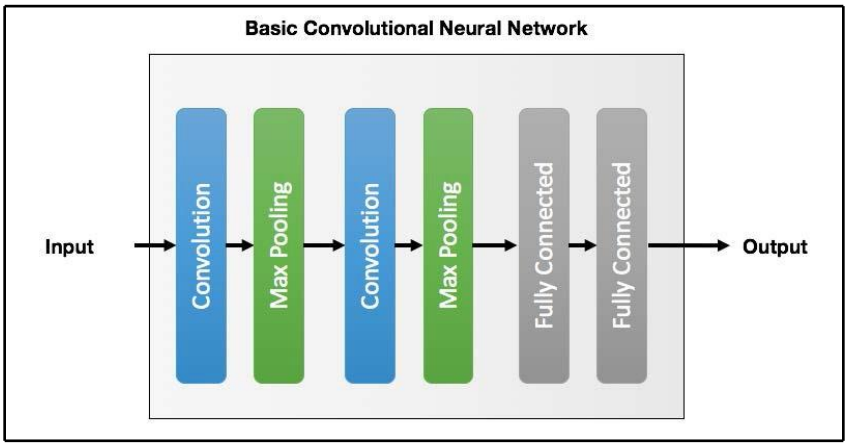
We're finally ready to start building our CNN in Keras. In this section, we'll take two different approaches to model building. First, we'll start by building a relatively simple CNN consisting of a few layers. We'll take a look at the performance of the simple model, and discuss its pros and cons. Next, we'll use a model that was considered state-of-the art

`

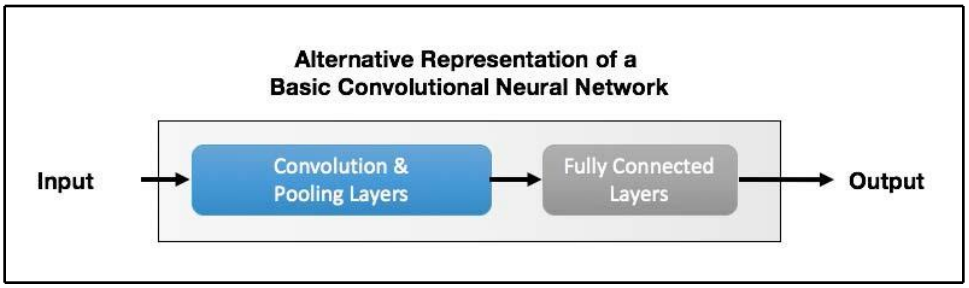
just a few years ago—the VGG16 model. We'll see how we can leverage on the pre-trained weights to adapt the VGG16 model for cats versus dogs image classification.

# Building a simple CNN

In an earlier section, we showed how the fundamental building blocks of a CNN consist of a series of convolutional and pooling layers. In this section, we're going to build a basic CNN consisting of this repeating pattern, as shown in the following diagram:




This basic CNN consists of two repeated blocks of **Convolution** and **Max Pooling**, following by two **Fully Connected** layers. As discussed in a previous section, the convolution and max pooling layers are responsible for learning the spatial characteristics of the classes (for example, identifying the ears of cats), whereas the **Fully Connected** layers learn to make predictions using these spatial characteristics. We can thus represent the architecture of our basic CNN in another manner (we shall see why it is useful to visualize our neural network in this manner in the next subsection):




`

Building a CNN is similar to building an MLP or a feedforward neural network, as we've done in the previous chapters. We'll start off by declaring a new `Sequential` model instance:

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

model = Sequential()
```

Before we add any convolutional layers, it is useful to think about the hyperparameters that we are going to use. For a CNN, there are several hyperparameters:

- **Convolutional layer filter size**: Most modern CNNs use a small filter size of `3` x `3`.
- **Number of filters**: Let's use a filter number of `32`. This is a good balance between speed and performance.
- **Input size**: As we've seen in an earlier section, the input images have different sizes, with their width and height approximately 150 px. Let's use an input size of `32` x `32` pixels. This compresses the original image, which can result in some information loss, but helps to speed up the training of our neural network.
- **Max pooling size**: A common max pooling size is `2` x `2`. This will halve the input layer dimensions.
- **Batch size**: This corresponds to the number of training samples to use in each mini batch during gradient descent. A large batch size results in more accurate training but longer training time and memory usage. Let's use a batch size of `16`.
- **Steps per epoch**: This is the number of iterations in each training epoch. Typically, this is equal to the number of training samples divided by the batch size.
- **Epochs**: The number of epochs to train our data. Note that, in neural networks, the number of epochs refers to the number of times the model sees each training sample during training. Multiple epochs are usually needed, as gradient descent is an iterative optimization method. Let's train our model for `10` epochs. This means that each training sample will be passed to the the model 10 times during training.

`

Let's declare variables for these hyperparameters so that they are constant throughout our code:

```
FILTER_SIZE = 3
NUM_FILTERS = 32
INPUT_SIZE  = 32
MAXPOOL_SIZE = 2
BATCH_SIZE = 16
STEPS_PER_EPOCH = 20000//BATCH_SIZE
EPOCHS = 10
```

We can now add the first convolutional layer, with `32` filters, each of size (`3` x `3`):

```
model.add(Conv2D(NUM_FILTERS, (FILTER_SIZE, FILTER_SIZE),
                 input_shape = (INPUT_SIZE, INPUT_SIZE, 3),
                 activation = 'relu'))
```

Next, we add a max pooling layer:

```
model.add(MaxPooling2D(pool_size = (MAXPOOL_SIZE, MAXPOOL_SIZE)))
```

This is the basic convolution-pooling pattern of our CNN. Let's repeat this once more according to our model architecture:

```
model.add(Conv2D(NUM_FILTERS, (FILTER_SIZE, FILTER_SIZE),
                 input_shape = (INPUT_SIZE, INPUT_SIZE, 3),
                 activation = 'relu'))

model.add(MaxPooling2D(pool_size = (MAXPOOL_SIZE, MAXPOOL_SIZE)))
```

We are now done with the convolution and pooling layers. Before we move on to the fully connected layers, we need to flatten its input. `Flatten` is a function in Keras that transforms a multidimensional vector into a single dimensional vector. For example, if the vector is of shape (5,5,3) before passing to `Flatten`, the output vector will be of shape (75) after passing to `Flatten`.

To add a `Flatten` layer, we simply run the following code:

```
model.add(Flatten())
```

We can now add a fully connected layer with `128` nodes:

```
model.add(Dense(units = 128, activation = 'relu'))
```

`

Before we add our last fully connected layer, it is a good practice to add a dropout layer. The dropout layer randomly sets a certain fraction of its input to 0. This helps to reduce overfitting, by ensuring that the model does not place too much emphasis on certain weights:

```
# Set 50% of the weights to 0
model.add(Dropout(0.5))
```

We add one last fully connected layer to our model:

```
model.add(Dense(units = 1, activation = 'sigmoid'))
```

We'll compile our model using the `adam` optimizer. The `adam` optimizer is a generalization of the **stochastic gradient descent** (**SGD**) algorithm and it is widely used to train CNNs. The loss function is `binary_crossentropy` since we're doing a binary classification:

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

We're now ready to train our CNN. Notice that we have not loaded any of the data into memory. We'll use the `ImageDataGenerator` and `flow_from_directory` method to train our model in real time, which loads batches of the dataset into memory only as required:

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

training_set = training_data_generator. \
                   flow_from_directory('Dataset/PetImages/Train/',
                                       target_size=(INPUT_SIZE,INPUT_SIZE),
                                                    batch_size=BATCH_SIZE,
                                                    class_mode='binary')

model.fit_generator(training_set, steps_per_epoch = STEPS_PER_EPOCH,
                    epochs=EPOCHS, verbose=1)
```

`

This will start the training and once it is complete, you will see the following output:

```
Epoch 1/10
1250/1250 [==============================] - 79s 63ms/step - loss: 0.6347 - acc: 0.6247
Epoch 2/10
1250/1250 [==============================] - 85s 68ms/step - loss: 0.5540 - acc: 0.7175
Epoch 3/10
1250/1250 [==============================] - 81s 65ms/step - loss: 0.5066 - acc: 0.7511
Epoch 4/10
1250/1250 [==============================] - 87s 69ms/step - loss: 0.4778 - acc: 0.7696
Epoch 5/10
1250/1250 [==============================] - 80s 64ms/step - loss: 0.4478 - acc: 0.7858
Epoch 6/10
1250/1250 [==============================] - 85s 68ms/step - loss: 0.4247 - acc: 0.8054
Epoch 7/10
1250/1250 [==============================] - 81s 65ms/step - loss: 0.4007 - acc: 0.8141
Epoch 8/10
1250/1250 [==============================] - 82s 65ms/step - loss: 0.3835 - acc: 0.8241
Epoch 9/10
1250/1250 [==============================] - 85s 68ms/step - loss: 0.3635 - acc: 0.8371
Epoch 10/10
1250/1250 [==============================] - 81s 65ms/step - loss: 0.3395 - acc: 0.8486
```

We can clearly see that the loss decreases while the accuracy increases with each epoch.

Now that our model is trained, let's evaluate it on the testing set. We'll create a new `ImageDataGenerator` and call `flow_from_directory` on the images in the `test` folder:

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

test_set = testing_data_generator. \
               flow_from_directory('Dataset/PetImages/Test/',
                                   target_size=(INPUT_SIZE,INPUT_SIZE),
                                   batch_size=BATCH_SIZE,
                                   class_mode = 'binary')
score = model.evaluate_generator(test_set, steps=len(test_set))
for idx, metric in enumerate(model.metrics_names):
    print("{}: {}".format(metric, score[idx]))
```

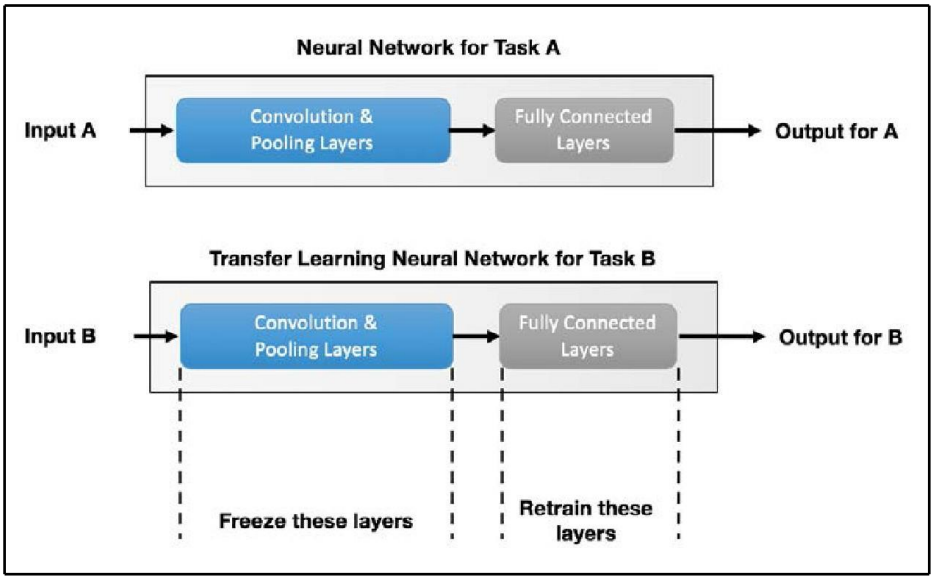
We obtained an accuracy of 80%! That's pretty impressive considering that we only used a basic CNN. This shows the power of CNNs; we obtained an accuracy close to human performance from just a few lines of code.

`

# Leveraging on pre-trained models using transfer learning

We can obtain better performance by leveraging on transfer learning.

Transfer learning is a technique in machine learning where a model trained for a certain task is modified to make predictions for another task. For example, we may use a model trained to classify cars to classify trucks instead, since they are similar. In the context of CNN, transfer learning involves freezing the convolution-pooling layers, and only retraining the final fully connected layers. The following diagram illustrates this process:




How does transfer learning work? Intuitively, the purpose of the convolution and pooling layers is to learn the spatial characteristics of the classes. We can therefore reuse these layers since the spatial characteristics are similar in both tasks. We just need to retrain the final fully connected layers to re-purpose the neural network to make predictions for the new class. Naturally, a crucial requirement for transfer learning is that tasks A and B must be similar to one another.

`

In this section, we're going to re-purpose the VGG16 model to make predictions on images of cats and dogs. The VGG16 model was originally developed for the ILSVRC, which required the model to make a 1,000 class multiclass classification. Among the 1,000 classes are specific breeds of cats and dogs. In other words, VGG16 knows how to recognize specific breeds of cats and dogs, and not just cats and dogs in general. It is therefore a viable approach to use transfer learning using the VGG16 model for our cats and dogs image classification problem.

The VGG16 model and its trained weights are provided directly in Keras. Let's create a new `VGG16` model, as shown in the following code:

```
from keras.applications.vgg16 import VGG16

INPUT_SIZE = 128 # Change this to 48 if the code takes too long to run
vgg16 = VGG16(include_top=False, weights='imagenet',
              input_shape=(INPUT_SIZE,INPUT_SIZE,3))
```

Note that we used `include_top=False` when we created a new VGG16 model. This argument tells Keras not to import the fully connected layers at the end of the VGG16 network.

We're now going to freeze the rest of the layers in the VGG16 model, since we're not going to retrain them from scratch. We can freeze the layers by running the following code snippet:

```
for layer in vgg16.layers:
    layer.trainable = False
```

Next, we're going to add a fully connected layer with `1` node right at the end of the neural network. The syntax to do this is slightly different, since the VGG16 model is not a Keras `Sequential` model that we're used to. In any case, we can add the layers by running the following code:

```
from keras.models import Model

input_ = vgg16.input
output_ = vgg16(input_)
last_layer = Flatten(name='flatten')(output_)
last_layer = Dense(1, activation='sigmoid')(last_layer)
model = Model(input=input_, output=last_layer)
```

`

This is just a manual way of adding layers in Keras, which the `.add()` function in `Sequential` model has simplified for us so far. The rest of the code is similar to what we have seen in the previous section. We declare a training data generator, and we train the model (only the newly added layers) by calling `flow_from_directory()`. Since we only need to train the final layer, we'll just train the model for `3` epochs:

Mã có thể mất khoảng một giờ để chạy mà không cần GPU. Để tăng tốc quá trình đào tạo, em có thể giảm tham số INPUT_SIZE nhưng điều này có thể ảnh hưởng đến độ chính xác của mô hình.

```
# Define hyperparameters
BATCH_SIZE = 16
STEPS_PER_EPOCH = 200
EPOCHS = 3

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

training_data_generator = ImageDataGenerator(rescale = 1./255)
testing_data_generator = ImageDataGenerator(rescale = 1./255)

training_set = training_data_generator. \
                   flow_from_directory('Dataset/PetImages/Train/',
                                       target_size=(INPUT_SIZE,INPUT_SIZE),
                                       batch_size = BATCH_SIZE,
                                       class_mode = 'binary')

test_set = testing_data_generator. \
               flow_from_directory('Dataset/PetImages/Test/',
                                   target_size=(INPUT_SIZE,INPUT_SIZE),
                                   batch_size = BATCH_SIZE,
                                   class_mode = 'binary')

model.fit_generator(training_set, steps_per_epoch = STEPS_PER_EPOCH,
                    epochs = EPOCHS, verbose=1)
```

`

We'll get the following output:

```
Epoch 1/3
200/200 [==============================] - 381s 2s/step - loss: 0.3808 - acc: 0.8253
Epoch 2/3
200/200 [==============================] - 418s 2s/step - loss: 0.2903 - acc: 0.8731
Epoch 3/3
200/200 [==============================] - 404s 2s/step - loss: 0.2941 - acc: 0.8754
```

The training accuracy doesn't look much different to the basic CNN in the previous section. This is expected, since both neural networks do really well in the training set. However, the testing accuracy is ultimately the metric which we will use to evaluate the performance of our model. Let's see how well it does on the testing set:

```
score = model.evaluate_generator(test_set, len(test_set))

for idx, metric in enumerate(model.metrics_names):
    print("{}: {}".format(metric, score[idx]))
```

What is the output?

# Results analysis

Let's take a deeper look into our results. In particular, we would like to know what kind of images our CNN does well in, and what kind of images it gets wrong.

Recall that the output of the sigmoid activation function in the last layer of our CNN is a list of values between 0 and 1 (one value/prediction per image). If the output value is < `0.5`, then the prediction is class 0 (that is, cat) and if the output value is >= `0.5`, then the prediction is class 1 (that is, dog). Therefore, an output value close to `0.5` means that the model isn't so sure, while an output value very close to `0.0` or `1.0` means that the model is very sure about its predictions.

Let's run through the images in the testing set one by one, using our model to make predictions on the class of the image, and classify the images according to three categories:

- **Strongly right predictions**: The model predicted these images correctly, and the output value is > `0.8` or < `0.2`
- **Strongly wrong predictions**: The model predicted these images wrongly, and the output value is > `0.8` or < `0.2`
- **Weakly wrong predictions**: The model predicted these images wrongly, and the output value is between `0.4` and `0.6`

`

The following code snippet will do this for us:

```
# Generate test set for data visualization
test_set = testing_data_generator. \
               flow_from_directory('Dataset/PetImages/Test/',
                                   target_size = (INPUT_SIZE,INPUT_SIZE),
                                   batch_size = 1,
                                   class_mode = 'binary')

strongly_wrong_idx = []
strongly_right_idx = []
weakly_wrong_idx = []

for i in range(test_set.  len  ()):
    img = test_set.  getitem  (i)[0]
    pred_prob = model.predict(img)[0][0]
    pred_label = int(pred_prob > 0.5)
    actual_label = int(test_set.  getitem  (i)[1][0])
    if pred_label != actual_label and (pred_prob > 0.8 or
        pred_prob < 0.2): strongly_wrong_idx.append(i)
    elif pred_label != actual_label and (pred_prob > 0.4 and
        pred_prob < 0.6): weakly_wrong_idx.append(i)
    elif pred_label == actual_label and (pred_prob > 0.8 or
        pred_prob < 0.2): strongly_right_idx.append(i)
    # stop once we have enough images to plot
    if (len(strongly_wrong_idx)>=9 and len(strongly_right_idx)>=9
        and len(weakly_wrong_idx)>=9): break
```

Let's visualize the images from these three groups by randomly selecting 9 of the images in each group, and plot them on a $3 \times 3$ grid. The following helper function allows us to do that:

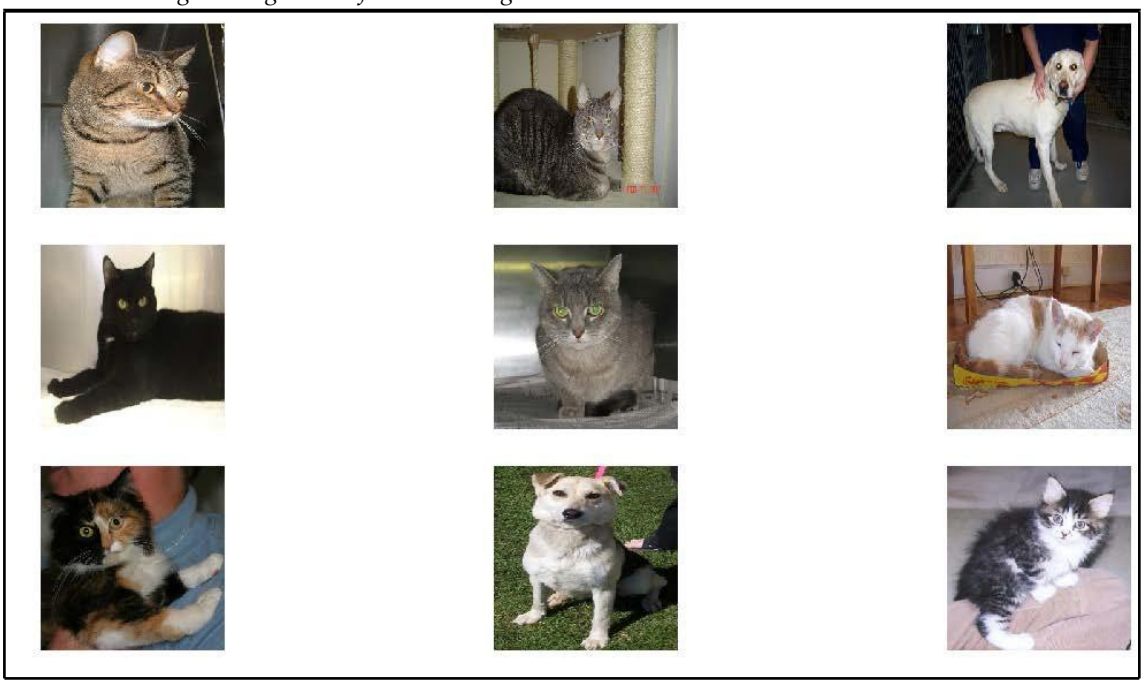
```
from matplotlib import pyplot as plt
import random

def plot_on_grid(test_set, idx_to_plot, img_size=INPUT_SIZE):

    fig, ax = plt.subplots(3,3, figsize=(20,10))
    for i, idx in enumerate(random.sample(idx_to_plot,9)):
        img = test_set.  getitem  (idx)[0].reshape(img_size, img_size ,3)
        ax[int(i/3), i%3].imshow(img)
        ax[int(i/3), i%3].axis('off')
```

We can now plot 9 randomly selected images from the strongly right predictions group:

```
plot_on_grid(test_set, strongly_right_idx)
```

We'll see the following output:

`



Selected images that have strong predictions, and are correct

No surprises there! These are almost classical images of cats and dogs. Notice that the pointy ears of cats and the dark eyes of dogs can all be seen in the preceding images. These characteristic features allow our CNN to easily identify them.

Let's now take a look at the strongly wrong predictions group:

```
plot_on_grid(test_set, strongly_wrong_idx)
```

`

We'll get the following output:

Selected images that have strong predictions, but are wrong

We notice a few commonalities among these strongly wrong predictions. The first thing we notice is that certain dogs do resemble cats with their pointy ears. Perhaps our neural network placed too much emphasis on the pointy ears and classified these dogs as cats. Another thing we notice is that some of the subjects were not facing the camera, making it really difficult to identify them. No wonder our neural network got them wrong.

Finally, let's take a look at the weakly wrong predictions group:

```
plot_on_grid(test_set, weakly_wrong_idx)
```

`

We'll get the following output:

Selected images that have weak predictions, and are wrong

These images are ones that our model is on the fence with. Perhaps there is an equal number of characteristics to suggest that the object could be a dog or a cat. This is perhaps the most obvious with the images in the first row, where the puppies in the first row have a small frame like a cat, which could have confused the neural network.

## Câu hỏi:

1. Bằng cách sử dụng mô hình VGG16 đã tinh chỉnh, bạn đã đạt được độ chính xác là bao nhiêu?
2. Hãy kiểm tra lại và cho biết những ảnh như thế nào đã gây khó khăn cho mô hình phân loại của bạn?
3. Vai trò của lớp tích chập (convolutional layer) và lớp gộp (pooling layer) là gì?
4. Vai trò của lớp kết nối đầy đủ (fully connected layer) là gì?
5. Transfer learning là gì và nó hữu ích như thế nào?

`