Image Classification using Convolutional Neural Networks

Object recognition tasks can be broadly classified into three different types:

- **Image Classification**
- **Object Detection**
- **Instance Segmentation**

In **Image Classification**, the input to the problem is an image and the required output is simply a prediction of the class that the image belongs to. In image classification, the problem is applied on pixels as our input data and specifically, the intensity value of each pixel.

In **Object Detection**, the input to the problem is an image and the required output are bounding boxes surrounding the detected objects. The neural network can no longer assume that there is only one class present in the image, and must assume that the image contains multiple classes. The neural network must then identify the presence of each class in the image, and to draw a bounding box around each of them. The intuitive idea behind their approach is to propose multiple boxes where objects of interest may exist, and then to use a CNN to predict the most likely class inside each bounding box.

Lastly, in **Instance Segmentation**, the input to the problem is an image and the output are pixel groupings that correspond to each class. You can think of instance segmentation as a refinement of object detection.

Digital Image Data

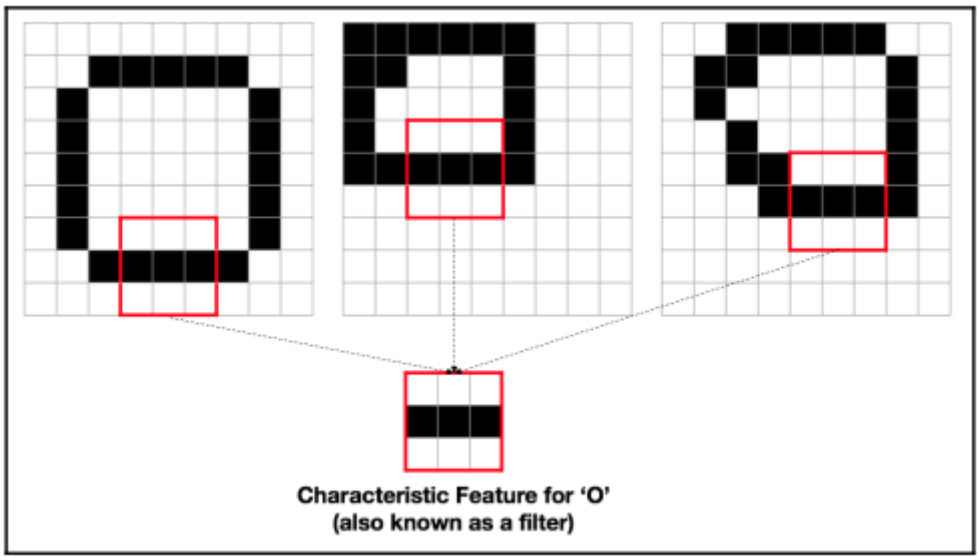
Let's assume that we have a greyscaled image. In a computer, white pixels are represented with the value **0** and black pixels are represented with a value of **255**. Everything else in between white and black(that is shades of gray) has a value in between **0** and **255**.

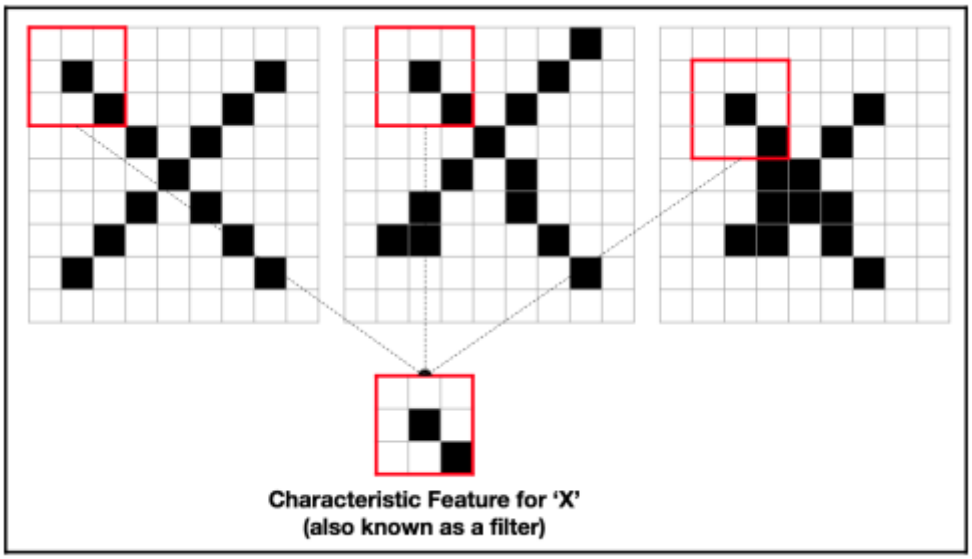
Color images are simply images with three channels - red, green and blue (RGB). The pixel values in each channel then represent the red intensity, the green intensity and blue intensity within **0**-**255** scale.

Filtering and Convolution

In order to look for specific features within an entire image, we will use **Convolution**. Before understanding convolution, we have to understand filtering.

Suppose we have a 9x9 image as our input and we need to classify the image as an X or an O. In either case, we cannot expect the figures to be drawn perfectly. The characteristic features in the images that allows us to differentiate them easily are that the Os tend to have flat horizontal edges, while Xs tend to have diagonal lines.





The characteristic feature, also known as the filter, is of size 3x3. The presence of the characteristic feature in an image give us a big hint on the class of the image.

In order to search for the presence of the characteristic feature in an image, we can simply do a brute force search by taking the 3x3 filter, before sliding it through every single pixel in the image to look for a match. The mathematical function performed by the filter, known as **filtering**, is the element-wise multiplication of the sliding window with the filter.

The process of sliding the window through the entire image and calculating the filtered value is known as **convolution**. The layer in the neural network that performs convolution is known as the convolutional layer. Essentially, convolution provides us with a map to the areas where the characteristic feature is found in each image.

Note that, when we train a neural network, it will automatically learn the most appropriate filter to use. Also, the weights of a convolutional layer will be tuned during training. Lastly, there are two main hyperparameters in a convolutional layer:

- **Number Of Filters**: We can increase the number of filters to find multiple characteristic features.
- **Filter Size**: We can tune the filter size to represent larger characteristic features.

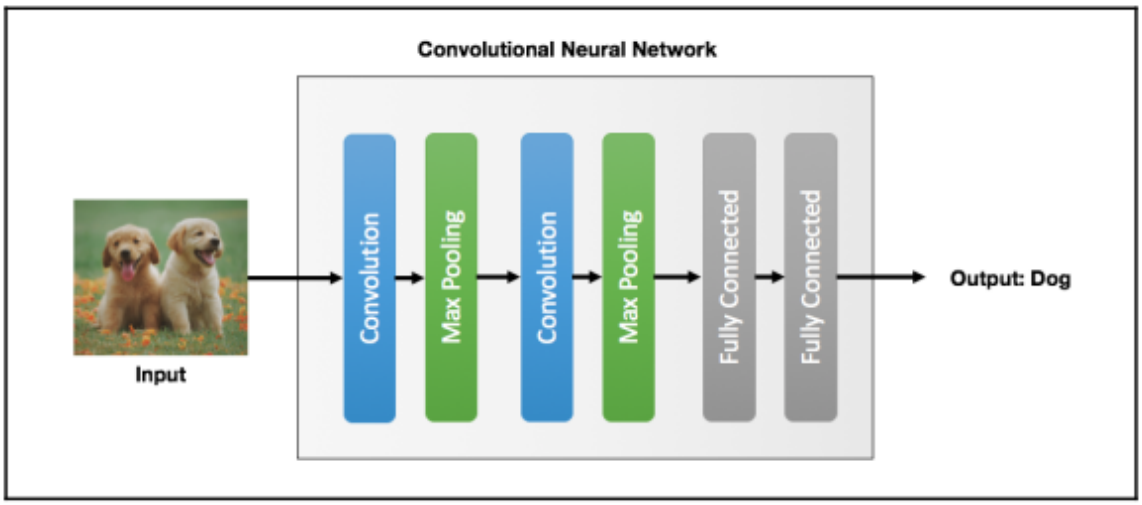
Max Pooling

In CNNs, it is common to place a max pooling layer immediately after a convolution layer. The object of **max pooling layer** is to reduce the number of weights after each convolution layer, thereby reducing model complexity and avoid overfitting.

The max pooling layer does this simply by looking at each subset of the input passed to it, and throwing out all but the maximum value in the subset.

Basic architecture of CNNs

The following diagram shows a typical CNN :



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

$$ReLU(x) = max(0, x)$$

The final layers in a CNN will always be **Fully Connected** layers with a ***sigmoid*** or ***softmax*** activation function.

$$Softmax(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

$$Sigmoid(x) = \frac{1}{1 + e^{-x}}$$

Note that the sigmoid activation function is used for binary classification, whereas the softmax activation function is used for multiple class classification problems.

The early layers of a CNN are responsible for identifying the characteristic spatial features, and the fully connected layers at the end are responsible for making predictions. Instead of creating features, we are simply providing all the data to the CNN as it is and then, the CNN automatically learns the best characteristic features to differentiate the classes.

Cat and Dog Dataset

By plotting various cat and dogs images, we can make some observations about the 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 is no guarantee that we will always see the tail of the subject in the image.

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. 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.

So, we need to separate **Cat** and **Dog** folder into *Train* and *Test* folder.

Image Augmentation

Image augmentation is the creation of additional training data by making minor alterations to images in certain ways in order to create new images. For example, we can do the following:

- Image rotation
- Image translation
- Horizontal flip
- Zooming into the image

The motivation for image augmentation is that CNN require a huge amount of training data before they can generalize well.

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.

Building a simple CNN

The basic CNN consists of two repeated blocks of **Convolution** and **Max Pooling**, following by two **Fully Connected** layers. The convolution and max pooling layers are responsible for learning the spatial characteristics of the classes (i.e identifying the ears of cats), whereas the fully connected layers learn to make predictions using these spatial characteristics.

Before adding any convolutional layers, there are several hyperparameters that we need to consider:

- **Convolutional layer filter size**: Most CNNs use a small filter size of 3 x 3.
- **Number of filters**: We will use a filter number of 32, that is a good balance between speed and performance.
- **Input size**: We will 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 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. So, we will 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 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. We will train our model for 10 epochs. This means that each training sample will be passed to the model 10 times during training.

After we are done with the convolution and pooling layers, we need to flatten our input for fully connected layers. `Flatten` is a function that transforms a multidimensional vector into a single dimensional vector. For example, if the vector is of shape (x,y,z) before passing to `Flatten`, the output vector will be of shape (*xyz*) after passing to it.

Before add our last fully connected layer, it is good 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 to much emphasis on certain weights.

Lastly, our final fully connected layer should have one unit with `sigmoid` activation since we are doing binary classification.

We will compile our model using the `adam` optimizer that is a generalization of the **stochastic gradient descent (SGD)** algorithm with `binary_crossentropy` as loss function.

The model is shown below:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 126, 126, 32)      896
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 63, 63, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 61, 61, 32)        9248
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 30, 30, 32)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 28800)             0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               3686528
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 129
=================================================================
Total params: 3,696,801
Trainable params: 3,696,801
Non-trainable params: 0
```

The testing accuracy of our model is showed below:

```
loss: 0.46369498391151426
acc: 0.7906
```

We obtained an accuracy of nearly 80% with the basic CNN.

Leveraging on pre-trained models using 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. In the context of CNN, transfer learning involves freezing the convolution-pooling layers and only retraining the final fully connected layers.

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.

Result analysis

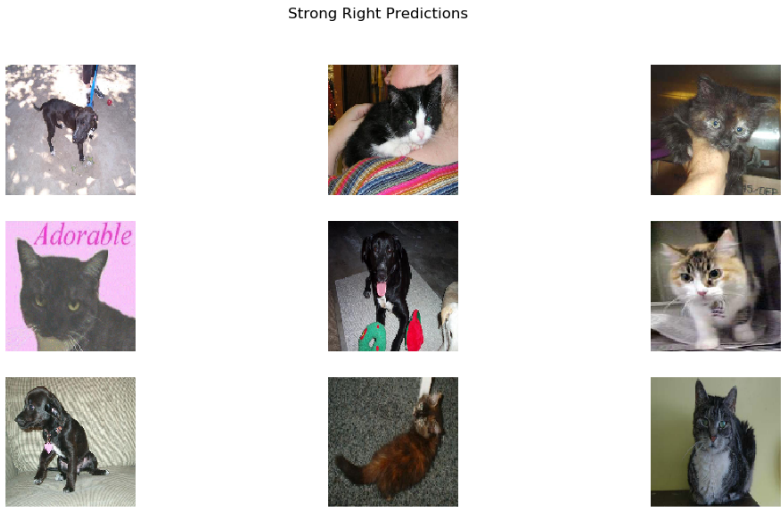
The output of the **`sigmoid`** activation function in the last layer of our CNN is a list of values between 0 and 1. If the output value is < 0.5, the the prediction is class 0 (**Cat** class) and if the output value is >= 0.5, then the prediction is class 1 (**Dog** class). Therefore, an output value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

We will classify the test images into 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
- **Weekly wrong predictions**: The model predicted those images wrongly and the output value is between 0.4 and 0.6

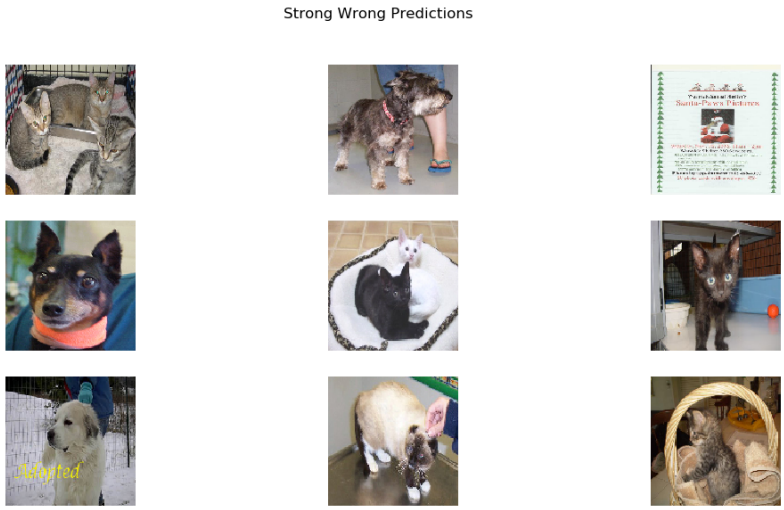
After we identify images from both classes into those 3 categories, we receive the following plots.

Strong Right Predictions



We can see clearly that there are almost classical images of cats and dogs. Notice that the pointy ears of cats, the dark eyes and the bigger size of dogs can be seen in the preceding images.

Strong Wrong Predictions



We notice a few similarities among these predictions. The first thing to notice is that certain dogs resemble cats with their pointy ears. Perhaps our network put an emphasis on those traits and classified dogs as cats. Another thing we notice is that a lot of subjects are not facing the camera and there are more than one subjects in the same image, making them difficult to identify.

Weakly Wrong Predictions

Weakly Wrong Predictions

We deduce that there is an equal number of characteristics to suggest that the object could be a dog or a cat. This is perhaps most obvious at 5th image that displays a puppy with green eyes that has small frame and eyes similar to a cat.