Idiomatic Programmer

Learning Keras

Handbook 2: Computer Vision Data Engineering

Andrew Ferlitsch, Google AI

The Idiomatic Programmer - Learning Keras

Handbook 2 - Computer Vision Data Engineering

Part 6 - Computer Vision Data Collection and Assembly
Part 7 - Computer Vision Data Engineering
Part 8 - Computer Vision Data Augmentation
Part 9 - Computer Vision Data Curation

Part 6 - Computer Vision Data Collection and Assembly

As a Googler, one of my duties is to educate software engineers on how to use machine learning. I already had experience creating online tutorials, meetups, conference presentations, and coursework for coding school, but I am always looking for new ways to effectively teach.

Welcome to my latest approach, the idiomatic programmer. My audience are software engineers who are proficient in a non-AI framework, like Angular, React, Django, etc. You know at least the basics of Python. It's okay if you still struggle with what is a compression, what is a generator; you still have some confusion with the weird multi-dimensional array slicing, and this thing about which objects are mutable and non-mutable on the heap. For this tutorial its okay.

You have a desire (or requirement) to become a machine learning engineer. What does that mean? A machine learning engineer (MLE) is an applied engineer. You don't need to know statistics (really you don't!), you don't need to know computational theory. If you fell asleep in your college calculus class on what a derivative is, that's okay, and if somebody asks you to do a dot product between two matrices you'd look them in the eyes and say why?

Your job is to learn the knobs and levers of a framework, and apply your skills and experience to produce solutions for real world problems. That's what I am going to help you with.

Overview

In this part, we will cover best practices for data collection and assembling when building a dataset for training of a deep (convolutional) neural network for computer vision (i.e., image classification).

Data Collection

The capability of a trained model for image classification is dependent on the quantity and distribution of the image data the model was trained with. Here are typical problems you might encounter with your training data:

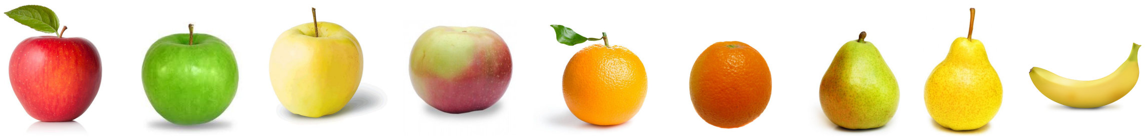
1. Too few images overall.
2. Too few images of a specific label (class).
3. Insufficient variation in perspective (e.g., angle, zoom).
4. Insufficient variation in lighting conditions.

Too Few Images Overall

A dataset may simply be too small for a convolutional neural network to learn:

- The features associated with each label (class).
- The distribution of the features across samples of each label.
- The variance in the features of each label.

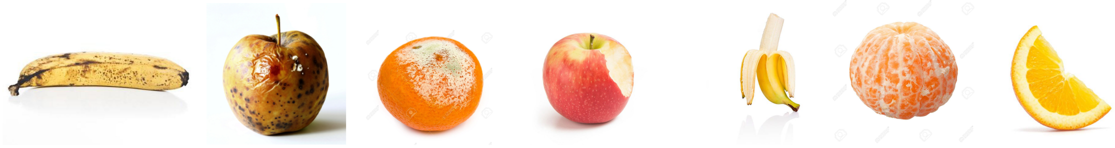
Let's consider a dataset of types of fruits (e.g., apples, oranges, bananas, pears, etc). Apples, oranges and pears have some similarity in shape, while bananas would be very different in shape. Oranges are fairly consistent in color, while apples and bananas can have a range of colors (green and yellow for both, and red for apples). For apples, pears and oranges, we might still have part of the stem, while bananas have no stems.



Samples of fruits of different colors and with and without stems - License

Let's say the above is the dataset. I have apples with stems, but I also have oranges and pears with stems, and there is an apple and an orange without a stem. While the oranges look round, so does the green apple. There is a red apple, but there is also an orange with a red hue. There are simply too few samples to learn the features, the distribution and the variance.

We may want to recognize the fruit when it is spoiled. But spoiling patterns differ. For bananas, apples and pears they turn brown, but with orange you have a white fungus. We may want to recognize the fruit when it is partially eaten, like a bit out of an apple, a sliced orange, or a peeled banana.



Samples of fruits of different spoilage, bitten, peeled and sliced

There are a lot of things to consider, but let's start with the basic common practices when collecting images for a dataset.

1. Each label (class) should have at least 1000 images.

2. Each distinct color for a label should be at least 10% of the images for that label.

   Example: 1000 images of apples with color red, green and yellow. At least 100 should be green apples, another 100 red apples and another 100 yellow apples.

3. Each secondary (not always present) feature for a label should be at least 5% of the images for the label.

   Example: 1000 images of apples. At least 50 are with a stem and 50 without a stem.

4. When there is a progression in state (e.g., age) for a label, the progression should be at least 20% of the images for the label.

   Example: 1000 images of apples. At least 200 are ripe, spoiled, insect damaged, sliced or Partially eaten.

Another common practice that improves overall accuracy is to create one additional label as *not any*, and populate the label with randomly chosen images that are not of any of the labels which the model will be trained to recognize.

Too Few Images of a Specific Class

A label (class) within a dataset may simply be too small, relative to the other labels. In this case, the dataset is referred to as being *unbalanced*.

Here are some common practices for determining when a dataset is considered unbalanced:

1. If the number of images for any label is less than 20% of the mean.

   For example, let's say that for each label in the fruits dataset has an average (mean) of 1000 images --then for any one label (e.g., plum), there should be at least 200 images.

2. If the number of images for any label is less than 10% of the label with the most images.

For example, let’s say that apples have 3000 images and has the most images of all the labels --then for any one label, there should be at least 300 images.

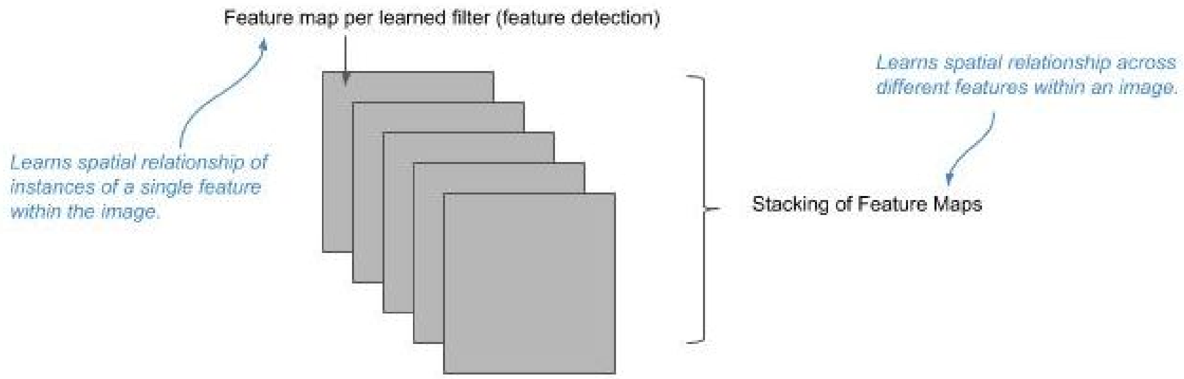
Insufficient Variation in Perspective

If all the images in a label are from the same perspective (angle, distance) then it’s likely during training that the model will overfit. Overfitting means that the weights and biases learned will be tightly fitted to the images for that label(s) in the training data, and while the accuracy will be very high with the training data, but when evaluated against image samples of a different perspective, the accuracy is very low.

A change in perspective will cause a change in spatial distances between features. Consider the example of the two images of an apple below. In the first one, the apple is viewed straight on, while in the second one, the apple is slightly tilted. It’s the same apple, but the spatial relationship between the stem and the body of the apple has changed (i.e., rotation).

In a CNN, the convolutional layers will learn filters to extract individual features into feature maps. Each feature map contains the strength of detection and location in the image of detecting the individual feature, from which the neural network can learn the spatial relationship and variance between instances of the single feature.

Feature maps are then stacked (see picture below). This stacking allows the neural network to learn the spatial relationships and variance between different features.


Feature map per learned filter (feature detection)
Learns spatial relationship across different features within an image.
Learns spatial relationship of instances of a single feature within the image.
Stacking of Feature Maps

The common practice is to have images from a variety of perspectives that closely matches the likely distribution of perspectives in the application that will use the trained model.

Insufficient Variation in Lighting Conditions

Another area to consider is the lighting conditions. Does the lighting conditions of the images reflect the variance in lighting conditions in the application that will use the trained model.

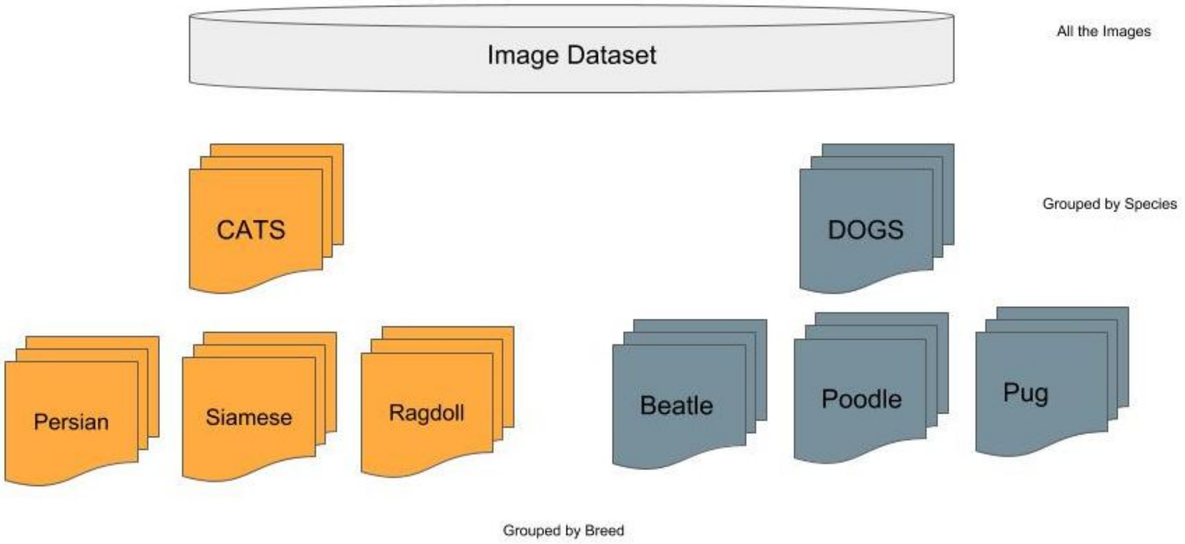
If the application is used only indoors and under a constant well lighted condition, then there may not be a need for variance in lighting conditions. For all other environments the images should have different lighting conditions, such as:

1. Natural Light - Daytime
2. Natural Light - Dusk
3. Indoor Light - Well Lighted
4. Indoor Light - Dimmed

Assembling an Image Dataset

Let's start with the basics, you need photos (images) of what your going to train a neural network to recognize (classify). Second, you need to decide what it is your going to classify, and finally you will need to group the photos by the classification you decided on.

For example, let's say you have lots of photos of cats and dogs. Well, we could simply group the photos into two groups (classes), one for cats and one for dogs. That might not be interesting or what you want to achieve. For example, perhaps you want to classify cats and dogs by their breeds. In this case, you further group your cat images into breeds, and correspondingly your dog images into breeds. But maybe that's not all you want to do. Let's say you're building an application for a veterinary service. Perhaps you may also want to recognize (classify) characteristics like color, black, gray, calico, patchy,... or obvious sign of an alignment, like bite marks.



Maybe you're building an application for antique appraisal business where users can upload photos and get an automated appraisal. Let's say you have thousands of photos of antiques sold over the years, and documentation saying what they are, their condition and price at the time of sale. So perhaps you may want to group your photos by item and further by condition for recognition and then index each photo by an adjusted price in today's dollars that's fed to yet another machine learning algorithm (e.g., linear regression) to make a value estimate.

The next question to answer is, how many images do you need. There are a lot estimates that float around. For example, you might hear a hundred images of each class is enough to get started. Yes, that's enough to get started, but you will get low accuracy in the real world, except for trivial recognition like recognizing a ball from a box. Best practices recommend at least a 1000 images of each class. For example, the well-known MNIST dataset for handwritten digits has 5000 images per class, and it's easy to get 99%+ accuracy. On the other hand, the well-known flowers dataset (tulips, daisy, dandelion, sunflower, rose) averages 850 images per class, and takes very advanced techniques to get high accuracy.

Also consider the source of the images. If they are copyrighted, you may not have the legal right to use them for training a neural network for a commercial application.

If you're not certain on the accuracy of how correctly the images are labeled (e.g., an image of a Persian cat is labeled as a Siamese), you may want to have a second set of eyes go over and correct any mistakes. You can still learn and build a reliable neural network model with some errors if the proportion of incorrectly labeled images is very small. Typically, for statistical purposes, the rule of thumb is less than 4%.

Finally, consider how the images were taken and if they are diverse enough. For example, if all the images are taken with the same background that is generally not going to be a problem. That is, if all the classes have the same background, the background will not contribute to the class recognition, and the neural network will “learn” to filter it out. On the other hand, if all images of one class have the same background, and all the remaining classes have varied background, the neural network may inadvertently learn the background instead of the object for the one class. For example, in a paper published in 2016 by Rubeiro, an example dataset of wolves and huskies was used to train a model, where all the images of wolves were taken during the winter with snow in the background. The paper showed how the neural network inadvertently learned ‘snow’ for classifying a wolf, and when shown a picture of a husky in snow, it would predict the image as a wolf. Below is a checklist when collecting images for a dataset:

[ ] Do I have enough images per class?
[ ] Is my error labeling rate low enough?
[ ] Is my background uniform or diversified across all classes?

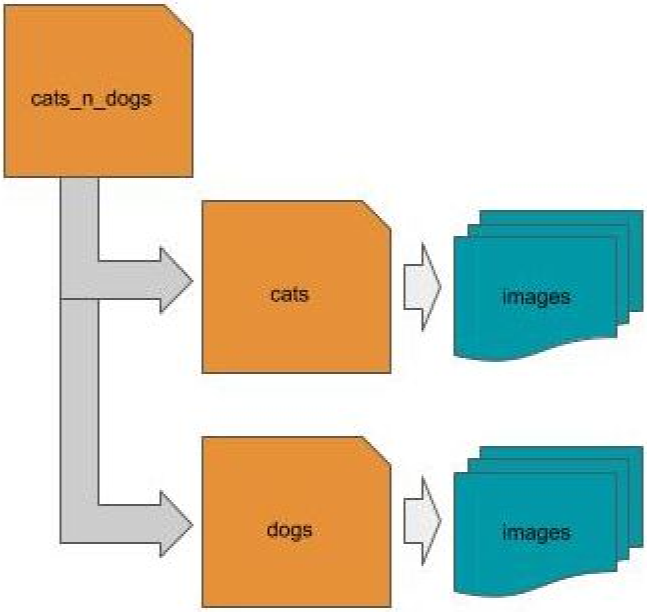
In a later part, we will show some additional image manipulation techniques (image augmentation) that can improve the quality of your image dataset.

Image Dataset Layouts

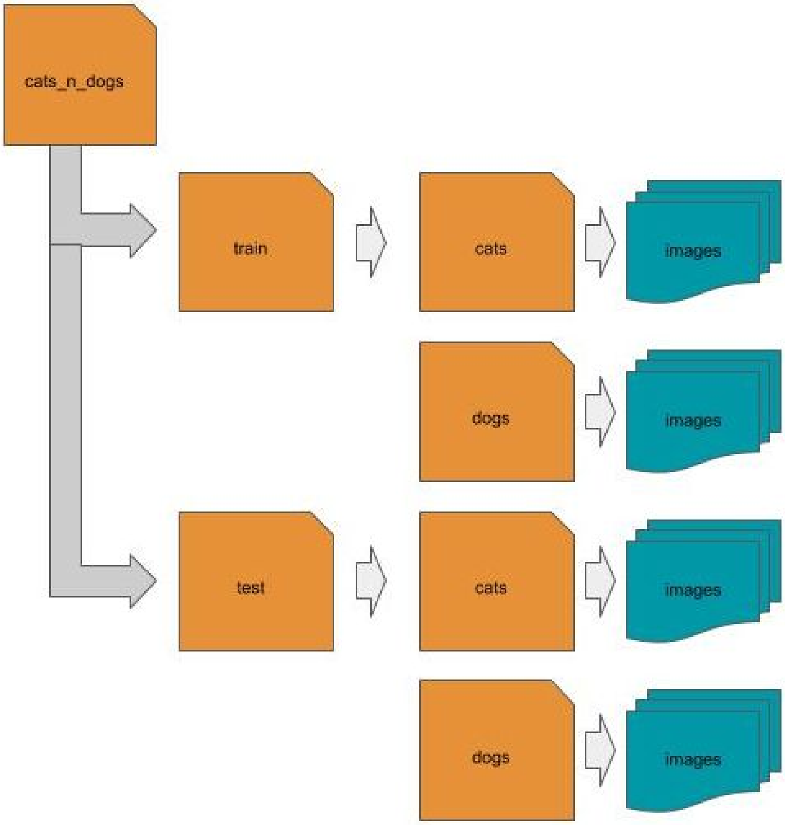
Now that you have your image dataset, the next step is to layout the image dataset so it can be prepared into *machine learning ready data* for training a deep (convolutional) neural network. We will cover several common methods.

Directory Structure

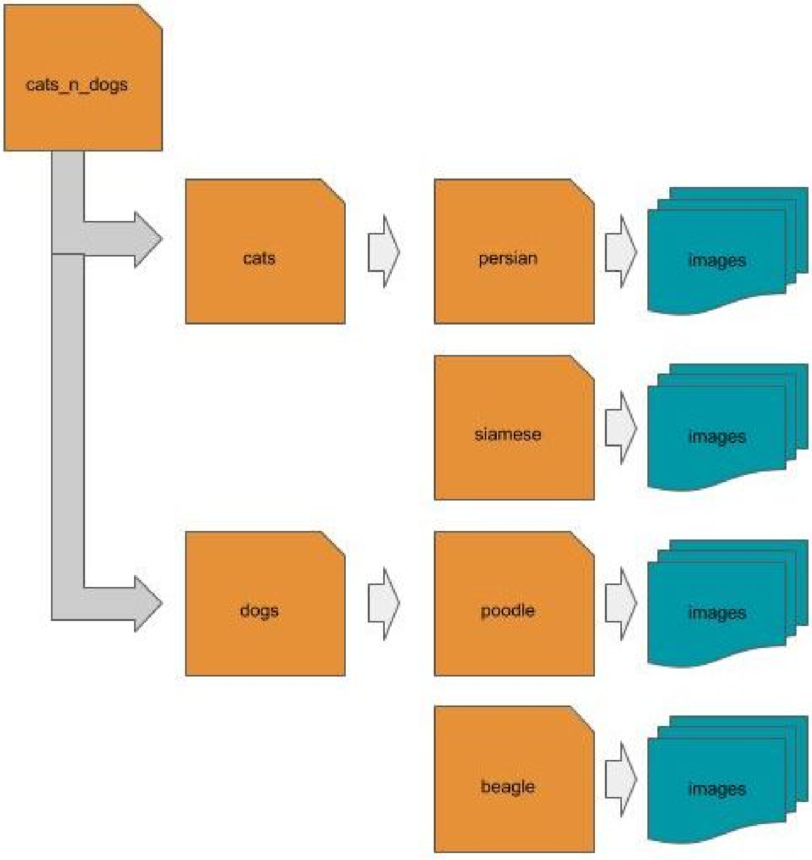
Laying out (placing) the images into a directory folder structure on a local disk is one of the most common layouts. In this layout, the root (parent) folder is a container for the dataset. Below the root level are one or more subdirectories, where each subdirectory corresponds to a class (label), and where each subdirectory contains the images that correspond to that class. Using our cats and dogs example, one would have a parent directory that might be named ‘cats_n_dogs’, with two subdirectories, one named ‘cats’ and the other ‘dogs’. Within each subdirectory would be the corresponding class of images.



Alternatively, if the dataset has been pre-split into training and test data, explained subsequently, then the split images for each class (label) are placed under the corresponding subfolders train and test, which are then underneath (subfolder of) the root folder of the dataset.



When the dataset is hierarchically labeled, each top-level class (label) subfolder is further partitioned into child-subfolders according to the class (label) hierarchy. Using our cats and dogs example, each image is hierarchically labeled by whether its a cat or dog (species) and then by breed.



CSV File

Another common layout is to use a CSV file to identify the location and class (label) of each image. In this case, each row in the CSV file is a separate image, and the CSV file contains at least two columns, one for the location of the image, and the other for the class (label) of the image. The location might be either a local path, a remote location, or the pixel data is embedded as the value of the location.

local path example:

```
label,location
‘cat’, cats_n_dogs/cat/1.jpg
‘dog’,cats_n_dogs/dog/2.jpg
…
```

remote path example:

```
label,location
‘cat’,’http://mysite.com/cats_n_dogs/cat/1.jpg’
‘dog’,’http://mysite.com/cats_n_dogs/dog/2.jpg’
...
```

embedded data example:

```
Label,location
‘cat’,[[...],[...],[...]]
‘dog’,[[...], [...], [...]]
```

JSON File

Another common layout is to use a JSON file to identify the location and class (label) of each image. In this case, the JSON file is an array of objects, where each object is a separate image, and each object has at least two keys, one for the location of the image, and the other for the class (label) of the image.The location might be either a local path, a remote location, or the pixel data is embedded as the value of the location.

local path example:

```
[
        {‘label’: ‘cat’, ‘location’: ‘cats_n_dogs/cat/1.jpg’ },
        {‘label’: ‘dog’, ‘location’: ‘cats_n_dogs/dog/2.jpg’}
        …
]
```

HDF5

Another common layout and storage format is to use a HDF5 (hierarchical data format) format file, which are very efficient for storing and accessing large amounts of multi-dimensional data, such as images. The format supports dataset and group objects, as well as attributes (metadata) per object. Typically, the file will consist of a group object per class (label), and each group will consist of one or more dataset objects for the corresponding images. Additional information, such as the label name, are stored as group attributes.

```
Group
        Attribute: {label: cats}
        Dataset: [...]

Group
        Attribute: {label: dogs}
        Dataset
```

The format supports hierarchical storage of groups. If the images are hierarchically labeled, then each group is further partitioned into a hierarchy of subgroups.

```
Group
        Attribute: {label: cats}
        Group:
                Attribute: {label: persian}
                Dataset: [...]
        Group:
                Attribute: {label: siamese}
                Dataset: [...]

Group
        Attribute: {name: dogs}
        Group:
                Attribute: {label: poodle}
                Dataset: [...]
        Group:
                Attribute: {label: beagle}
                Dataset: [...]
```

In-Memory

An image dataset may already be in memory. This is typical for curated datasets provided by ML frameworks (like Keras) for educational purposes. In this case, the image dataset has been prepared into machine learning ready data, and pre-split into train and test, with the label data already one-hot encoded (discussed later in this part).

```
              Image 1             Image 2             Image 3             Image 4             Image 5
X_train = [ [ [...],[...],[...]], [ [...],[...],[...]], [ [...],[...],[...]], [ [...],[...],[...]], [ [...],[...],[...]], … ]
Y_train = [ [... 1 …]          [… 1 …],            [… 1 …],            [... 1 …],          [... 1 …], …          ]
```

Next

In the next part, we will cover the principal behind and best practices for data engineering of an image dataset, in preparation for training.

Part 7 - Computer Vision Data Engineering

Overview

In this part, we will cover data engineering for training a deep (convolutional) neural network for computer vision (i.e., image recognition). Let's start with the basics. You can't just drop images into a neural network and expect it to learn (recognize) what the images are.

There are some preprocessing steps that need to be done with an image dataset, which is referred to as data engineering. These include:

1. Loading (reading) in the images as raw pixel data.
2. Fitting them to the input vector of the neural network which will be used for training.
3. Conversion of the pixel data.
4. Splitting the dataset

In the following sections, we will cover the different ways to perform these steps in **openCV, PIL** and **Keras** library modules.

PIL - Python Image Library

PIL or **Pillow** is Python's image manipulation library. In this section, we use the library to manually prepare an image dataset into machine learning ready data. Let's first cover some basics of PIL.

PIL was last updated in 2009 and supports Python 2 only. **Pillow** is a fork of the original PIL repo and continues to be maintained. One can install Pillow with `pip install pillow`.

Reading Images

The first step is to read an image from disk into memory. The image on disk will be in an image format like JPG, PNG, TIF, etc. These formats define how the image is encoded and compressed for storage. An image is read into memory using the PIL `Image.open()` method.

```
from PIL import Image
image = Image.open('myimage.jpg')
```

In practice, you will have a large number of images that need to be read in. Let's assume you want to read in all the images under some subdirectory (e.g., cats). In the code below, we scan (get list of) all the files in the subdirectory, read each one in as an image, and maintain a list of the read-in images as a list:

```
from PIL import Image
import os

def loadImages(subdir):
        images = []

        # get list of all files in subdirectory cats
        files = os.scandir(subdir)
        # read each image in and append the in-memory image to a list
        for file in files:
                images.append(Image.open(file.path))
        return images

loadImages('cats')
```

Note, `os.scandir()` was added in Python 3.5. If you are using Python 2.7 or an earlier version of Python 3, you can obtain a compatible version with `pip install scandir.`

Let’s expand on the above and assume the image dataset is laid out as a directory structure, where each subdirectory is a class (label). In this case, we would want to scan each subdirectory separately, and keep a record of the subdirectory names for the classes.

```
import os

def loadDirectory(parent):
        classes = {}  # list of class to label mappings
        dataset = []  # list of images by class

        # get list of all subdirectories under the parent (root) directory of the
        # dataset
        for subdir in os.scandir(parent):
                # ignore any entry that is not a subdirectory (e.g., license file)
                if not subdir.is_dir():
                        continue

                # maintain mapping of class (subdirectory name) to label (index)
                classes[subdir.name] = len(dataset)

                # maintain list of images by class
                dataset.append(loadImages(subdir.path))

                print("Processed:", subdir.name, "# Images",
                      len(dataset[len(dataset)-1]))
        return dataset, classes

loadDirectory('cats_n_dogs')
```

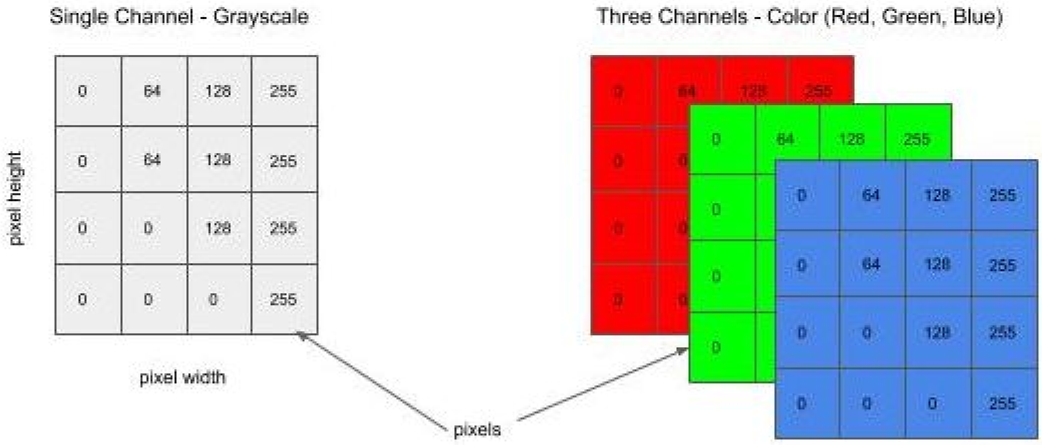
Let’s now try an example where the location of the image is remote (not local) and specified by an URL. In this case, we will need to make an HTTP request for the contents of the resource (image) specified by the URL and then decode the response into a binary byte stream.

```
from PIL import Image
import requests
from io import BytesIO

def remoteImage(url):
        try:
                response = requests.get(url)
                return Image.open(BytesIO(response.content))
        except:
                return None
```

Channel Conversion

Next, we need to set the number of channels to match the input shape of your neural network. The number of channels is the number of color planes in your image. For example, a grayscale image will have one color plane. An RGB color image will have three color planes, one for each of Red, Green and Blue.

In most cases, this is either going to be a single channel (grayscale) or three channels (RGB).



The `Image.open()` method will read in the image according to the number of planes in the image stored on disk. So if its a grayscale image, it will read it in as a single plane, if it's RGB it will read it in as three planes, and if it's RGBA (+alpha channel), it will read it in as 4 planes.

In general when working with RGBA images, the alpha channel can be dropped. It is a mask for setting the transparency of each pixel in the image, and therefore does not contain information that would otherwise contribute to the recognition of the image.

Once the image is read into memory, the next step is to convert the image to the number of channels that match the input shape of your neural network. So if the neural network takes grayscale images (single channel), we want to convert to grayscale; or if the neural network takes RGB images (three channel), we want to convert to RGB. The `convert()` method will do channel conversion. A parameter value of 'L' will convert to single channel (grayscale) and 'RGB' to three channel (RGB color). Below we updated the loadImages() function to include channel conversion:

```
from PIL import Image
import os

def loadImages(subdir, channels):
        images = []

        # get list of all files in subdirectory cats
        files = os.scandir(subdir)
        # read each image in and append the in-memory image to a list
        for file in files:
                   image = Image.open(file.path)
                   # convert to grayscale
                   if channels == 1:
                        image = image.convert('L')
                   # convert to RGB color
                   else:
                        image = image.convert('RGB')
                   images.append(image)
        return images

loadImages('cats', 3)
```

Resizing

Next, we need to resize each image to fit the shape of the input shape of the neural network. For example, a 2D convolutional neural network will take a shape of the form (height, width, channels). We dealt with the channel portion above, so next we need to resize the pixel height and width of each image to match the input shape. For example, if the input shape is (64, 128, 3), then we want to resize the height and width of each image to (64, 128). The `resize()` method will do the resizing.

In most cases, you will be downsizing (i.e., downsampling) each image. For example, 1024 x 768 image will be 3Mb in size. This is far more resolution than a neural network needs (see section on Convolutional Neural Networks). When the image is downsized, some resolution (details) will be lost. To minimize the effect when downsizing, it is a common practice to use the anti-aliasing algorithm in PIL:

```
from PIL import Image
import os

def loadImages(subdir, channels, shape):
        images = []

        # get list of all files in subdirectory cats
        files = os.scandir(subdir)
        # read each image in and append the in-memory image to a list
        for file in files:
                   image = Image.open(file.path)
                   # convert to grayscale
                   if channels == 1:
                        image = image.convert('L')
                   # convert to RGB color
                   else:
                        image = image.convert('RGB')
                   # resize the image to the target input shape
                   images.append(image.resize(shape, Image.ANTIALIAS))
        return images

loadImages('cats', 3, (128, 128))
```

Conversion to Numpy Array

In **Keras** (and other Python ML frameworks), the inputs to neural networks are multi-dimensional **numpy** arrays. Lists in Python are implemented as dynamic arrays and use indirect addressing on the heap. While this provides great flexibility, the performance is very poor for large dynamic arrays. In machine learning, arrays (vectors, matrices, tensors and batches) are very, very large.

The numpy module is a package that implements high-performance large array handling in contiguous memory in C, with a Python linkage.

For our purposes, we are going to want to convert the list of PIL images to a multi-dimensional numpy array. We can do this with the `numpy.asarray()` method. This method can take a single object (e.g., single PIL image) and convert it to an array, or a list of objects and convert them to a multidimensional array. The later will be faster for our purposes. Consider this, assume we have a list of 10000 PIL images. If we converted one at a time, we would have 10000 invocations of mapping the Python linkage for `numpy.asarray()` to its C linkage, and pushing/popping a stack frame. In the later, we have just one linkage mapping and stack frame.

In the code example below, we update the loadImages() function to include the step of converting the list of PIL images to a numpy array.

```
from PIL import Image
import os
import numpy as np

def loadImages(subdir, channels, shape):
        images = []

        # get list of all files in subdirectory cats
        files = os.scandir(subdir)
        # read each image in and append the in-memory image to a list
        for file in files:
                   image = Image.open(file.path)
                   # convert to grayscale
                   if channels == 1:
                        image = image.convert('L')
                   # convert to RGB color
                   else:
                        image = image.convert('RGB')
                # resize the image to the target input shape
                images.append(image.resize(shape, Image.ANTIALIAS))
        # UPDATED STEP
        # convert all the PIL images to numpy arrays in a single invocation
        return np.asarray(images)

loadImages('cats', 3, (128, 128))
```

OpenCV - Open Computer Vision Library

OpenCV is an open source image manipulation library that is implemented in a wide variety of programming languages, including Python. It has been around a long-time and widely supported and updated by the open source community. It has been the conventional work-horse of imaging scientists.

Everything we did in PIL, we will show in this section how to do in OpenCV. My personal preference is to use OpenCV. The underlying implementation of the Python version of OpenCV is written in C/C++, for high performance. I find that with OpenCV I can write imaging algorithms very close to the hardware (bare metal) and get higher performance when compared to PIL. Let's first cover some basics of OpenCV. One can install OpenCV with `pip install opencv-python`.

Reading Images

An image is read into memory using the `cv2.imread()` method. One of the first advantages I find with the `cv2.imread()` method is the output is already in a multi-dimensional numpy data type.

```
import cv2

image = cv2.imread('myimage.jpg')
```

Channel Conversion

Another advantage of OpenCV over PIL is that you can do the channel conversion at the time of reading in the image, instead of a second step. By default, `cv2.imread()` will convert the image to a three channel RGB image. You can specify a second parameter that indicates which channel conversion to use. In the example below, we show doing the channel conversion at the time the image is read in.

```
# read in image as a single channel (grayscale) image
if channel == 1:
        image = cv2.imread('myimage.jpg', cv2.IMREAD_GRAYSCALE)
# read in image as a three channel (color) image
else:
        image = cv2.imread('myimage.jpg', cv2.IMREAD_COLOR)
```

In the example below, we show reading in the image from a remote location (url) and doing the channel conversion at the same time. In this case, we use the method `cv2.imdecode()`:

```
try:
        response = requests.get(url)
        if channel == 1:
               return cv2.imdecode(BytesIO(response.content), cv2.IMREAD_GRAYSCALE)
        else:
               return cv2.imdecode(BytesIO(response.content), cv2.IMREAD_COLOR)
```

```
except:
       return None
```

Resizing

Images are resized using the `cv2.resize()` method. The second parameter is a tuple of the height and width to resize the image to. The optional (keyword) third parameter is the interpolation algorithm to use when resizing. Since in most cases you will be downsampling, the common practice is to use the `cv2.INTER_AREA` algorithm for best results in preserving information and minimizing artifacts when downsampling an image.

```
# resize an image to 128 (height) x 128 (width)
image = cv2.resize(image, (128, 128), interpolation=cv2.INTER_AREA)
```

Let’s now rewrite the loadImages() function using OpenCV.

```
import cv2
import os
import numpy as np

def loadImages(subdir, channels, shape):
        images = []

        # get list of all files in subdirectory cats
        files = os.scandir(subdir)
        # read each image in and append the in-memory image to a list
        for file in files:
                   # convert to grayscale
                   if channels == 1:
                        image = cv2.imread(file.path, cv2.IMREAD_GRAYSCALE)
                   # convert to RGB color
                   else:
                        image = cv2.imread(file.path, cv2.IMREAD_COLOR)
                   # resize the image to the target input shape
                   images.append(cv2.resize(image, shape, cv2.INTER_AREA))
        return np.asarray(images)

loadImages('cats', 3, (128, 128))
```

Normalization and Standardization

At this point, all the image data is in a numpy multidimensional array. As a final step, we want to perform normalization (or standardization) on the pixel data and fit it to a data type size. To explain in detail why we do normalization (or standardization) is beyond the scope of this tutorial. It's sufficient for you to know that the process makes all the images have a similar pixel value distribution, which leads to faster convergence when training a neural network. This means less epochs (or steps), which means less total compute time.

Normalization will squash all the pixel values in a range between 0 and 1, or between -1 and 1, while standardization will squash the pixel values into a distribution with a mean centered at 0 and one standard deviation.

Images taken by a camera are almost universally 8 bits per pixel per channel format, represented as an unsigned 8 bit integer value (e.g., np.uint8). Therefore, we can do the squashing function by dividing each pixel value by 255. So a value 0 would become a 0, a value of 255 would become a 1, and all other values will be in-between.

The numpy library has overridden the division operator between two numpy arrays to perform a very fast array-wide matrix division. So we can do this operation as a single step.

```
# Normalization between 0 and 1 of 8 bits per pixel per channel
images = images / 255.0
```

In some image datasets, the images may be in a different bits per pixel representation. For example, 16 bits per pixel per channel for high color images, which have a dynamic color range between 0 .. 65535 vs 0 .. 255 for 8 bits per pixel. When normalizing between 0 and 1, we need to divide by the maximum value of the pixel range.

```
if images.itemsize == 1:
        images = images / 255.0
# Normalization between 0 and 1 of 16 bits per pixel per channel
elif images.itemsize == 2:
        images = images / 65535.0
```

When normalizing, some prefer to normalize between -1 and 1, to center the pixel values around 0.

```
if images.itemsize == 1:
        images = images / 127.5 - 1
elif images.itemsize == 2:
        images = images / 32767.5 - 1
```

Another method of squashing pixel values into a similar distribution is called standardization. In this method, a mean and a standard deviation are calculated across all the pixel values. Then the mean is centered at zero and each pixel divided by a standard deviation of 1. This takes more computation than normalization.

If your images are a fairly basic and fairly consistent distribution of pixel values, then a normalization is sufficient. For example, in the MNIST dataset, all the digits are in a narrow grayscale range and all the backgrounds are the same. For this case, normalization is sufficient. When your working though with color images, the distribution of pixel values can be very diverse across images. In general, normalization is used with grayscale images and standardization with color images.

To prevent a division by zero in the case where the standard deviation is zero, it is a common practice to add a tiny amount, referred to as an epsilon, so the denominator will never be zero.

```
# standardization across all pixels/channels
e = 1e-7 # epison, add a tiny amount to standard deviation to prevent divide by 0.
images = (images - np.mean(images)) / (np.std(images) + e)
```

The above calculates the mean and standard deviation across all pixels and channels. Other practices will calculate a mean and standard deviation per channel. In the case of a 3 channel RGB image, this is done as:

```
# standardization by per channel mean
e = 1e-7 # epison, add a tiny amount to standard deviation to prevent divide by 0.
images = (images - np.mean(images, axis=(0,1))) / (np.std(images, axis=(0,1)) + e)
```

Another practice is to calculate the mean and standard deviation on a per pixel location. In the case of a 3 channel RGB image, this is done as:

```
# standardization by mean image
e = 1e-7 # epison, add a tiny amount to standard deviation to prevent divide by 0.
images = (images - np.mean(images, axis=(2))) / (np.std(images, axis=(2)) + e)
```

The last step is setting the data type of the normalized (or standardized) pixel data. At this point the pixel data values are floating point numbers, and generally very small numbers (e.g., between 0.0 and 1.0). By default, numpy will represent floating point values as double precision floats (float64). That's 8 bytes per float, which is 8 bytes per pixel per channel. Let's assume the input shape to the neural network is (128, 128, 3). That will require 50K pixels per image, and at 8 bytes per pixel per channel, that's 400K bytes per image. At 10,000 images, that will require 4GB of memory to hold them all at once in memory.

Neural networks work well with lower float precision. Single precision float (float32) is the common practice, which takes ½ the space and ¼ the number of floating point instructions per cycle. If your neural network is not exposed to vanishing gradients (see batch normalization), you may be able to reduce the size to half precision float (float16) if supported by the hardware, which is 2 bytes per float, reducing the size another ½ and ¼ the number of floating point instructions per cycle.

The data type of the numpy array is set using the `astype()` method. Since the (smaller) data type will also affect the amount of time to do matrix operations, it is best to do the data type conversion prior to the normalization (standardization) step, to maximize the speed of the matrix operations.

```
# set the data type
images = images.astype(np.float32)
# now do the standardization
images = (images - np.mean(images)) - np.std(images)
```

Keras Image Preprocessing

Keras provides image preprocessing functions for both in-memory (i.e., numpy arrays) and on-disk images with the class `ImageDataGenerator`. This class combines data preprocessing, data feeding and data augmentation in a single step. I find whether you do your data preprocessing, as well as augmentation and feeding, outside of Keras functions or all-in-one is a personal preference. For those wanting the fine level control of the data for functionality and the bare-metal programming for speed, I recommend the 'manual' approach of using openCV or PIL.For those seeking an easy encapsulation of the steps, I would recommend the methods builtin to Keras.

The `ImageDataGenerator` class creates a generator for feeding batches into a neural network during training --which I describe in more detail in later sections. For our purpose here, I will only cover the preprocessing functions. The subset of parameters in `ImageDataGenerator` for preprocessing are:

- rescale / dtype
- featurewise_center / featurewise_std_normalization
- samplewise_center / samplewise_std_normalization
- zca_whitening / zca_epsilon

We will not cover ZCA whitening, since its uncommon to use this normalization method for images in a convolutional neural network.

The image preprocessing using `ImageDataGenerator` is a two-step process:

1. Specify normalization / standardization in the instantiation of the `ImageDataGenerator` object.
2. The images are presumed to either:
   a. be in memory as numpy tensors and already resized (`flow()` method), or
   b. are resized when feeding from disk using the `flow_from_directory()` method.

Normalization and Standardization

In the code example below, we instantiate the `ImageDataGenerator` to do a normalization of the pixel data by squashing the values between 0 and 1, and then perform the normalization by invoking the `fit()` method of the training data (x_train).

```
from keras.preprocessing.image import ImageDataGenerator

# specify normalizing the pixel data between 0 and 1 (for 8 bits per pixel/per
# channel)
datagen = ImageDataGenerator(rescale=1/255.0)

# normalize the pixels pixels in the training data
datagen.fit(x_train)
```

Alternately, we can normalize the pixel data by centering it around the mean (i.e., per channel mean) by setting the `featurewise_center` to `True`.

```
# specify normalizing the pixel data by centering them around a mean of 0
datagen = ImageDataGenerator(featurewise_center=True)
```

Additionally, we can specify to do standardization of both centering the pixel data around the mean (i.e., per channel) and squashing the values to +/- one standard deviation by setting `featurewise_std_normalization` to `True`

```
# specify normalizing the pixel data by centering them around a mean of 0 and
# squashing the values within +/- one standard deviation
datagen = ImageDataGenerator(featurewise_std_normalization=True)
```

The parameters `samplewise_center` and `samplewise_std_normalization` will center on the mean and squash between +/- one standard deviation on a per sample or mini-batch basis. At one time, some practitioners recommended this, to smooth out fluctuations in the gradient during the loss function (discussed later), but with the addition of batch normalization this is unnecessary --which is supported by Keras layer method `BatchNormalization()`

Datatype

The data type of the pixel data, after optional normalization or standardization, by default with be a double precision float (FLOAT64). The parameter `dtype` when specified, will cast the pixel data to the corresponding data type, which is specified as a numpy data type (e.g., np.float32 for FLOAT32).

Resizing

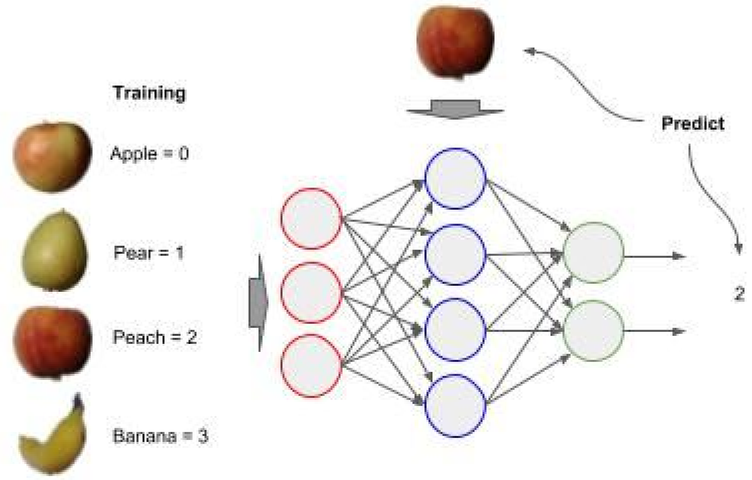
When feeding data during training to a Keras model, the data is either feed from in-memory or on-disk. When feeding in-memory using the `flow()` method, you must resize the images prior to feeding the neural network, such as the methods we described using PIL or OpenCV.

When you feed from on-disk using the `flow_from_directory()`, your set the parameter `target_size` to the height x width (HxW) of the input vector to the neural network.

```
# Create generator for feeding neural network from on-disk, and
# resize images to 128 x 128 as being feed
feeder = datagen.flow_from_directory('root_of_dataset', target_size=(128, 128))

# Train the neural network by calling the fit_generator() method with the above
feeder
model.fit_generator(feeder, steps_per_epoch=steps, epochs=epochs)
```

Label Encoding

Next, we need to keep track of which label (class) goes with each image. For each class (e.g., apple, pear, peach, banana, etc - Fruits360) we assign a unique integer value, starting with 0 to represent the class (remember everything in a neural network is a number - even the labels). When we train a neural network, the label will be the integer value we assigned to the class (y) and when predicting the neural network will output the integer value of the class it predicts (ŷ , aka. y-hat).

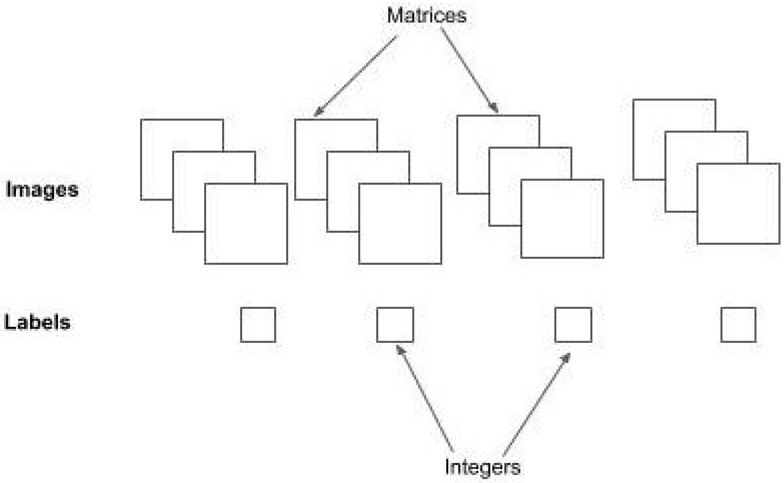


For the next step, we build a list of labels that correspond to the list of images. In our loadDirectory() example, the list dataset has an entry for each class of images processed, and the length of the entry is the number of images. In the code below we build a parallel list for the corresponding labels, with the same number of entries, and same length per entry as the dataset (processed images). For example, if the third entry (index 2) is a peach, and we have 100 images, then in the labels list at the third entry, we want to add a 100 element array, each with the corresponding label value of 2.

```
dataset, classes = loadDirectory('fruits')

# construct a corresponding labels list
for ix, images in enumerate(dataset):
        # The labels have the same indices as the images (that's the label)
        label = np.asarray([ix] * len(images)).astype(np.uint16)
        labels.append(label)
```

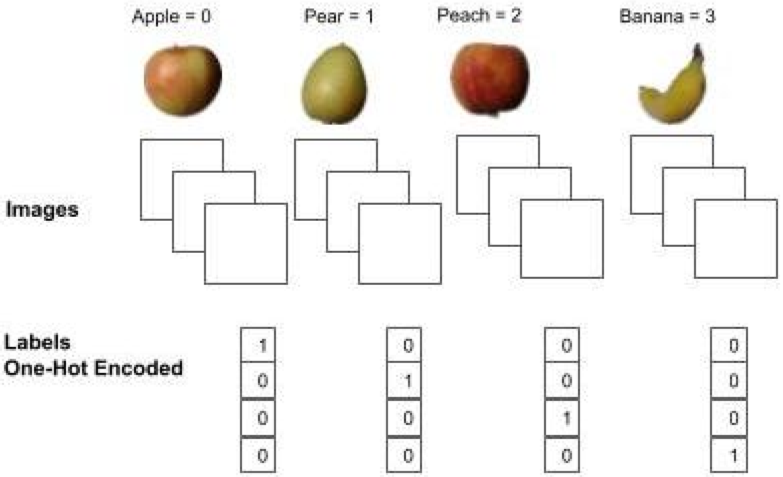
Note, in the above, we set the data type of the labels to `np.uint16`, which is a 16-bit integer which can hold up to 65,535, to conserve space. If you have more than 65,535 classes, then you will want to set the data type as `np.uint32`.

Next, we want both the list of processed images (machine learning ready data) and list of labels to each be a single long contiguous array. We use the `np.concatenate()` method to concatenate each element sequentially in a list into a single contiguous multidimensional numpy array.

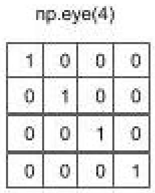


```
# Convert list of Images and Labels to a contiguous numpy multidimensional array
dataset = np.concatenate(dataset)
labels  = np.concatenate(labels)
```

The final preparation step for the labels is to one-hot encode them to match the output vector of the neural network. The neural network will have an output node per class. For example, if the classes are apple, peach, pear and banana, the output layer will have four nodes, where each node corresponds to one of the classes. For each label (e.g., 2), a one value is placed at the index location in the output vector that corresponds to the node for that label, and the remaining entries have the value 0.



The first code example below demonstrates how to 'manually' do a one hot encoding in a single shot. The function one_hot() takes as input the 1D array of labels (Y) and outputs a 2D one-hot encoded version (return Y). The second parameter (C) is the number of classes (output nodes), which will correspond to the length of each one-hot encoded label. The `np.eye()` method creates a zero value 2D matrix with an equal number of rows and columns (square) specified by the parameter C, and then places a one value along the diagonal.



np.eye(4)

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

The label value (e.g., 2) at each entry in the 1D labels vector is used as an index into the np.eye() generated matrix for the corresponding replacement one-hot encoded label.

```
def one_hot(Y, C):
        """ Convert Vector to one-hot encoding """
        Y = np.eye(C)[Y.reshape(-1)]
        return Y

# one-hot encode the labels
labels = one_hot(labels, len(classes))
```

Alternatively, the Keras framework has a builtin method `to_categorical()` to automatically do one-hot encoding of labels.

```
from keras.utils import to_categorical

# one-hot encode the labels
labels = to_categorical(labels, len(classes))
```

Splitting into Train and Test

When training a neural network for computer vision, one splits the processed image dataset into training data and test data, also referred to as the holdout set. That is, a portion of the processed image dataset is set aside to test the accuracy of the trained model, and is not used as part of the training. Generally only a small percentage should be set aside, and the larger the image dataset, the smaller that percentage can be. Below is a general rule of thumb of the percentage to set aside for test data, based on the number of images in the dataset:

< 1000	20%
<10000	10%
>10000	5%

Prior to splitting the image dataset, we will want to randomly shuffle it. Why? Let’s presume that when building the dataset that all the images were sequenced by class. For example, all the apples come first, followed by pears, followed by peaches, and then by bananas. If we split this as 75% train and 25% test without shuffling, all the apples, pears and peaches will be in the training set and the bananas in the test set. The model won't learn to recognize bananas, since they are not in the training set, and the test set will attempt to test for a class that the model hasn't been trained on. Useless!

If we randomly shuffle the processed image dataset, we can assume a fairly balanced distribution of the classes between the training and test data. The example code below shows a 'manual' method to shuffle the processed image dataset prior to splitting. We start by creating an array of indices for each index in the dataset. That is, if the dataset has 100 items, then we generate a 100 item array with the values 0 .. 99, where every element in the indices array corresponds to an entry in the dataset (and corresponding labels). We use Python's `random.shuffle()` method to randomly shuffle the entries in our indices array.

Once shuffled, we construct a shuffled version of the dataset and labels data in the corresponding lists X and Y. We iterate through the shuffled indices and use the indices item values to construct the shuffled versions, and then finally convert the lists back into numpy multidimensional arrays.

```
import random
import numpy as np

# Shuffle the Data (Image Data and Labels)
indices = [_ for _ in range(len(dataset))]
random.shuffle(indices)

# Let's reassemble of list of images and corresponding labels according to our
shuffle order in 'indices'
X =[]
Y =[]
for _ in indices:
    X.append(dataset[_])
    Y.append(labels[_])

X = np.asarray(X)
Y = np.asarray(Y)
```

Alternately, we can shuffle the dataset (images) and labels in place without using an indirect index if we use a seed. Random number generators are actually pseudo random number generators, in that they use a mathematical formula to generate a pseudo random number sequence. A seed initializes the formula; whereby the same seed will always generate the same random number sequence. To shuffle both the dataset (images) and labels separately and maintain the same corresponding random sequence between the two, we use the same seed before each shuffle.

```
import random

# set a seed for the random number distribution before shuffling the data (images)
random.seed(101)
random.shuffle(dataset)

# set the same seed before shuffling the corresponding labels to get the same
# random number distribution
random.seed(101)
random.shuffle(labels)
```

In the code below, we calculate a pivot point in the shuffled dataset, where the training percentage will be before the pivot point and the test percentage after the pivot point. We then use the pivot point to create (array slicing) the corresponding train and test dataset and labels.

```
# Split the preprocessed image data (and labels) into train and test data
pivot = int(len(indices) * (1 - SPLIT))
X_train = X[:pivot]
Y_train = Y[:pivot]
X_test  = X[pivot:]
Y_test  = Y[pivot:]
```

As an alternative, one can automatically shuffle and split a processed image dataset and labels using the **scikit-learn** function `train_test_split()`.

```
from sklearn.model_selection import train_test_split

# Split the processed image dataset into training and test data
X_train, X_test, Y_train, Y_test = train_test_split(dataset, labels,
                                                    test_size=0.20, shuffle=True)
```

As an alternative if the dataset is to be fed from on-disk (vs. in-memory), one can automatically shuffle and split the dataset and labels using Kera's `ImageDataGenerator` method `flow_from_directory()`. In the code below, we:

1. When instantiating `ImageDataGenerator,`
    a. Set the parameter `rescale` to normalize the images before being fed to the model during training.
    b. Set the parameter `validation_split` to split 10% of the training data for validation.
2. In the method `flow_from_directory(),`

   a. Set the parameter target_size to resize the images to the input shape of the model.
   b. Set the parameter batch_size to set the mini-batch size for feeding the model during training.
   c. Set the parameter shuffle to True to randomly shuffle the training data on each epoch.
3. Invoke the method fit_generator() to train the model for 50 epochs.

```
# Create data generator where images will be normalized between 0 and 1, and
# split the dataset between training and validation (10%).
datagen = ImageDataGenerator(rescale=1/255.0, validation_split=0.1)

# Create a generator to feed the model during training
generator = datagen.flow_from_directory(dataset_directory,
                                        target_size=(128, 128),
                                        batch_size=32,
                                        shuffle=True)
# train the model
model.fit_generator(generator, epochs=50)
```

Next

In the next part, we will cover the principal behind and best practices for data augmentation to improve the accuracy of a trained model.

Part 8 - Computer Vision Data Augmentation

Overview

In this part, we will cover best practices for image augmentation when building a dataset for improving the training of a deep (convolutional) neural network for computer vision (i.e., image classification).

Image Augmentation

Image augmentation is the process of generating new images from existing images to improve the balance and/or variance in the training data. It is typically used to address:

1. Label(s) which have too few images.
2. Insufficient variance in [view] perspective.
3. Insufficient variance in lighting.

Image augmentation uses imaging algorithms which will modify an existing image without changing what the image is. That is, when a person views the modified image they will still see the same thing.

You should take image augmentation with some caution. As you increase the amount of image augmentation, the model will become more and more generalized. At some point the model may over generalized and become underfitted. In this situation, you will have a high rate of correctly classifying an image of label A is label A (True Positive - TP), but you will see a dramatic increase of the model classifying images of non-label A as label A (False Positive - FP). This is referred to as precision (TP / (TP + FP)). So when using a high level of image augmentation, you need to pay attention to drops in precision (i.e., increase in false positives) as well as paying attention to accuracy.

$$\text{Accuracy} = \frac{TP}{No.\ of\ Samples} \qquad \text{Precision} = \frac{TP}{TP + FP}$$

Perspective Change

In general practice, the augmentation method that contributes the most to increasing the accuracy of a neural network are changes in the perspective. The methods, also referred to as transformations, are:

1. Flip
2. Rotation
3. Zoom

4. Shift

Flip

Flip transforms an image by flipping an image either on the vertical or horizontal axis. Since the image data is represented as a stack of 2D matrices (i.e., one per channel), A flip can be done efficiently as a matrix transpose function without changes (e.g., interpolation) of the pixel data.

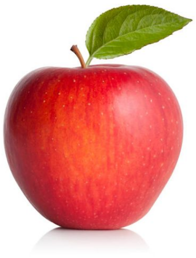
Let's start by showing how to flip an image, and then we will show how to flip an image using the popular imaging libraries in Python.

The code example below demonstrates how to flip an image vertically (mirror) and horizontally using a matrix transpose method in Python's `PIL` imaging library:

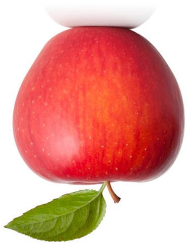
```
from PIL import Image

# read in the image
image = Image.open('apple.jpg')

# display the image in its original perspective
image.show()

# flip the image on the vertical axis (mirror)
flip = image.transpose(Image.FLIP_LEFT_RIGHT)
flip.show()

# flip the image on the horizontal axis (upside down)
flip = image.transpose(Image.FLIP_TOP_BOTTOM)
flip.show()
```

Original - License

Vertical Flip



Horizontal Flip



Alternately, the flips can be done using the PIL class `ImageOps` module, as demonstrated in the code below:

```
from PIL import Image, ImageOps

# read in the image
image = Image.open('apple.jpg')

# flip the image on the vertical axis (mirror)
flip = ImageOps.mirror(image)
flip.show()

# flip the image on the horizontal axis (upside down)
flip = ImageOps.flip(image)
flip.show()
```

The code example below demonstrates how to flip an image vertically (mirror) and horizontally using a matrix transpose method in `OpenCV`:

```
import cv2
from matplotlib import pyplot as plt

# read in the image
image = cv2.imread('apple.jpg')

# display the image in its original perspective
plt.imshow(image)

# flip the image on the vertical axis (mirror)
flip = cv2.flip(image, 1)
plt.imshow(flip)

# flip the image on the horizontal axis (upside down)
flip = cv2.flip(image, 0)
plt.imshow(flip)
```

The code example below demonstrates how to flip an image vertically (mirror) and horizontally using a matrix transpose method in `numpy`:

```
import numpy as np
import cv2
from matplotlib import pyplot as plt

# read in the image
image = cv2.imread('apple.jpg')
```

```
# flip the image on the vertical axis (mirror)
flip = np.flip(image, 1)
plt.imshow(flip)

# flip the image on the horizontal axis (upside down)
flip = np.flip(image, 0)
plt.imshow(flip)
```

Rotate 90 (right), 180 (upside down), 270 (left) degrees

In addition to flips, a matrix transpose operation can be used to rotate an image 90 degrees (left), 180 degrees, and 270 degrees (right). Like a flip, the operation is very efficient, and does not require interpolation of pixels and does not have a side effect of clipping.

The code example below demonstrates how to rotate an image 90, 180 and 270 degrees using a matrix transpose method in Python's `PIL` imaging library:

```
from PIL import Image

# read in the image
image = Image.open('apple.jpg')

# rotate the image 90 degrees
rotate = image.transpose(Image.ROTATE_90)
rotate.show()

# rotate the image 180 degrees
rotate = image.transpose(Image.ROTATE_180)
rotate.show()

# rotate the image 270 degrees
rotate = image.transpose(Image.ROTATE_270)
rotate.show()
```

90

180

270

OpenCV does not have a transpose method for 90 or 270 degrees, you can do a 180 by using the flip method with value of -1. All other rotations using OpenCV are demonstrated later using imutils module.

```
import cv2
from matplotlib import pyplot as plt

# read in the image
image = cv2.imread('apple.jpg')

# rotate the image 180 degrees
rotate = cv2.flip(image, -1)
plt.imshow(rotate)
```

The code example below demonstrates how to rotate an image 90, 180 and 270 degrees using the numpy method rot90(), where the first parameter is the image to rotate 90 degrees and the second parameter (k) is the number of times to perform the rotation:

```
import numpy as np
import cv2
from matplotlib import pyplot as plt

# read in the image
image = cv2.imread('apple.jpg')

# rotate the image 90 degrees
rotate = np.rot90(image, 1)
plt.imshow(rotate)

# rotate the image 180 degrees
rotate = np.rot90(image, 2)
plt.imshow(rotate)

# rotate the image 270 degrees
rotate = np.rot90(image, 3)
plt.imshow(rotate)
```

Note, when flipping the image 90 or 270 degrees, you are changing the orientation of the image, which is not a problem if the height and width of the image are the same. If not, the height and width will be transposed in the rotated image and will not match the input vector of the neural network. In this case, you should use the imutils module discussed below to resize the image.

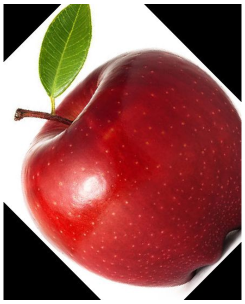
Rotation

A rotation transforms an image by rotating the image within -180 and 180 degrees. Generally, the degree of rotation is randomly selected. You may also want to limit the range of rotation to match the environment the model will be deployed in. Below are some common practices:

1. If the images will be dead-on, use -15 to 15 degree range.
2. If the images may be on an incline, use -30 to 30 degree range.
3. For small objects, like packages, money, use the full range of -180 to 180.

Another issue with rotation, is that if you rotate an image within the same size boundaries, other than 90, 180, or 270, a portion of the edge of the image will end up outside the boundary (i.e., clipped).

Below is an example of using PIL method `rotate()` to rotate the image of the apple 45 degrees. You can see part of the bottom of the apple and the stem are clipped.



The correct way to handle a rotation is to rotate it within a larger bounding area, such that none of the image is clipped, and then resize the rotated image back to the original size. For this purpose, I recommend using the `imutils` module (Adrian Rosebrock), which consists of a collection of convenience methods for `openCV`

```
import cv2, imutils
from matplotlib import pyplot as plt

# read in the image
image = cv2.imread('apple.jpg')

# remember the original Height and Width
shape = (image.shape[0], image.shape[1])

# rotate the image
rotate = imutils.rotate_bound(image, 45)
```

```
# resize the image back to its original shape
rotate = cv2.resize(rotate, shape, interpolation=cv2.INTER_AREA)
plt.imshow(rotate)
```

Zoom

Zoom transforms an image by zooming in from the center of the image, which is done with a resize and crop operation. First you find the center of the image, calculate the crop bounding box around the center, and then you crop the image.

When enlarging an image using `Image.resize()` the `Image.BICUBIC` interpolation generally provides the best results.

The code example below demonstrates how to zoom an image using Python's `PIL` imaging library:

```
from PIL import Image
image = Image.open('apple.jpg')

zoom = 2 # zoom by factor of 2
# remember the original height, width of the image
height, width = image.size

# resize (scale) the image proportional to the zoom
image = image.resize( (int(height*zoom), int(width*zoom)), Image.BICUBIC)

# find the center of the scaled image
center = (image.size[0]//2, image.size[1]//2)

# calculate the crop upper left corner
crop = (int(center[0]//zoom), int(center[1]//zoom))

# calculate the crop bounding box
box = ( crop[0], crop[1], (center[0] + crop[0]), (center[1] + crop[1]) )

image = image.crop( box )
image.show()
```

Below is the apple image zoomed by a factor of two:

The code example below demonstrates how to zoom in an image using OpenCV imaging library.

When enlarging an image using cv2.resize() interpolation cv2.INTER_CUBIC generally provides the best results. The interpolation cv2.INTER_LINEAR is faster and provides nearly comparable results. The interpolation cv2.INTER_AREA is generally used when reducing an image.

```
import cv2
from matplotlib import pyplot as plt

zoom = 2 # zoom by a factor of 2

# remember the original height, width of the image
height, width = image.shape[:2]

# find the center of the scaled image
center = (image.shape[0]//2, image.shape[1]//2)
z_height = int(height // zoom)
z_width  = int(width  // zoom)

# slice (cutout) the zoomed image by forming a crop bounding box
image = image[(center[0] - z_height//2):(center[0] + z_height//2), center[1] -
              z_width//2:(center[1] + z_width//2)]

# resize (enlarge) the cropped image back to the original size.
image = cv2.resize(image, (width, height), interpolation=cv2.INTER_CUBIC)

plt.imshow(image)
```

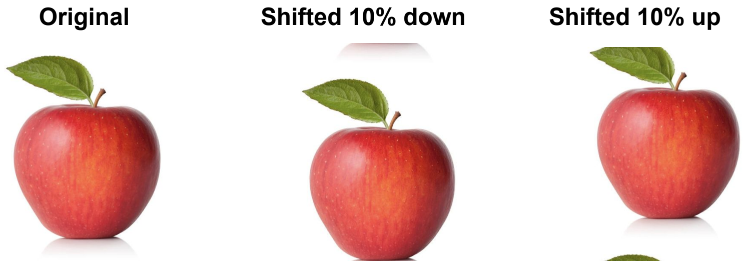
Shift

A shift will shift the pixel data in the image +/- in the vertical (height) or horizontal (width) axis. This will change the location in the image of the object being classified.

The code below demonstrates shifting an image +/- 10% vertically and horizontally using the numpy np.roll() method.

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

# read in the image
image = cv2.imread('apple.jpg')

# get the height and width of the image
height = image.shape[0]
Width  = image.shape[1]

# shift the image down by 10%
roll = np.roll(image, height // 10, axis=0)
plt.imshow(roll)

# shift the image up by 10%
roll = np.roll(image, -(height // 10), axis=0)
plt.imshow(roll)

# shift the image right by 10%
roll = np.roll(image, width // 10, axis=1)
plt.imshow(roll)

# shift the image left by 10%
roll = np.roll(image, -(width // 10), axis=1)
plt.imshow(roll)
```

Below are the apple image shifted down 10% and up 10%:



Original

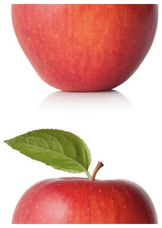
Shifted 10% down

Shifted 10% up

A shift is very efficient in that it is implemented as a roll operation of the matrix, where the rows (height) or columns (width) are shifted. As such, the pixels that are shifted off the end are added to the beginning. If the shift is too large, the image can become fractured into two pieces with each piece opposing each other.

Below is an example of where the apple was shifted by 50% vertically, leaving it fractured.

A fractured image



To avoid fracture, It is a general practice to limit the shift of the image to no more than 20%.

Lighting

There are a number of techniques for changing lighting. For example:

- Apply a uniform multiplier on each pixel.
- Apply specific multiplier per channel on each pixel.
- Apply specific multiplier per range of pixel values on each pixel.

The code below demonstrates using the `OpenCV` method `convertScaleAbs()`. This method applies a uniform multiplier across the pixels and a delta added to each value, specified by the parameters `alpha` and `beta`, respectively:

```
import cv2
from matplotlib import pyplot as plt

image = cv2.imread('apple.jpg')

contrast   = 0.5
brightness = 40

# scale the pixel values and then convert back to uint8
image = cv2.convertScaleAbs(image, alpha=contrast, beta=brightness)

plt.imshow(image)
```

Below is the apple image after the above code is applied:

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

image = cv2.imread('apple.jpg')

# multiplier: Blue* 1.7, Green * 0.2, Red * 0.1
image = image * np.array([1.7, 0.2, 0.1])
image = image.astype(np.uint8)

plt.imshow(image)
```

Below is the apple image after applying the above code:

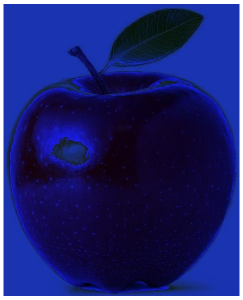


Image Augmentation in Keras

The Keras image preprocessing module supports a wide variety of image augmentation with the class `ImageDataGenerator`. The `ImageDataGenerator` class creates a generator for generating batches of augmented images. The class initializer takes as input zero or more parameters for specifying the type of augmentation. Below are a few of the parameters, which we will cover in this section:

- horizontal_flip=True
- vertical_flip=True
- rotation_range=degrees
- zoom_range=(lower, upper)
- width_shift_range=percent

- height_shift_range=percent
- brightness_range=(lower, upper)

Flip

In the code example below, we do the following:

1. Read in a single image of an apple.
2. Create batch of one image (the apple).
3. Instantiate an `ImageDataGenerator` object.
4. Initialize the `ImageDataGenerator` with our augmentation options (in this case: horizontal and vertical flip).
5. Use the `flow()` method of the `ImageDataGenerator` method to create a batch generator.
6. Iterate through the generator 6 times, each time returning a batch of 1 image in x.
    a. The generator will randomly select an augmentation (including no augmentation) per iteration.
    b. After transformation (augmentation), the pixel values will be 32-bit float.
    c. Change the data type of the pixels back to 8-bit integer, for displaying using matplotlib.

```
from keras.preprocessing.image import ImageDataGenerator
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Let's make a batch of 1 image (apple)
image = cv2.imread('apple.jpg')
batch = np.asarray([image])

# Create a data generator for augmenting the data
datagen = ImageDataGenerator(horizontal_flip=True, vertical_flip=True)

# Let's run the generator, where every image is a random augmentation
step=0
for x in datagen.flow(batch, batch_size=1):
        step += 1
        if step > 6: break
        plt.figure()
        # the augmentation operation will change the pixel data to float
        # change it back to uint8 for displaying the image
        plt.imshow(x[0].astype(np.uint8))
```

Rotation

In the code below, we use the `rotation_range` parameter to set random rotations between -60 and 60 degrees. Note, that rotate operation does not perform a bounds check and resize (like `imutils.rotate_bound()`, so part of the image may end up being clipped.

```
from keras.preprocessing.image import ImageDataGenerator
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Let's make a batch of 1 image (apple)
image = cv2.imread('apple.jpg')
batch = np.asarray([image])

# Create a data generator for augmenting the data
datagen = ImageDataGenerator(rotation_range=60)

# Let's run the generator, where every image is a random augmentation
step=0
for x in datagen.flow(batch, batch_size=1):
        step += 1
        if step > 6: break
        plt.figure()
        # the augmentation operation will change the pixel data to float
        # change it back to uint8 for displaying the image
        plt.imshow(x[0].astype(np.uint8))
```

Zoom

In the code below, we use the `zoom_range` parameter to set random values between 0.5 (zoom out) and 2 (zoom in). Note that the value can be specified either as a tuple or list of two elements.

```
from keras.preprocessing.image import ImageDataGenerator
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Let's make a batch of 1 image (apple)
image = cv2.imread('apple.jpg')
batch = np.asarray([image])

# Create a data generator for augmenting the data
```

```
datagen = ImageDataGenerator(zoom_range=(0.5, 2))

# Let's run the generator, where every image is a random augmentation
step=0
for x in datagen.flow(batch, batch_size=1):
        step += 1
        if step > 6: break
        plt.figure()
        # the augmentation operation will change the pixel data to float
        # change it back to uint8 for displaying the image
        plt.imshow(x[0].astype(np.uint8))
```

Shift

In the code below, we use the `width_shift_range` and `height_shift_range` to set random values between 0 and 20% in shift horizontally or vertically:

```
from keras.preprocessing.image import ImageDataGenerator
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Let's make a batch of 1 image (apple)
image = cv2.imread('apple.jpg')
batch = np.asarray([image])

# Create a data generator for augmenting the data
datagen = ImageDataGenerator(width_shift_range=0.2, height_shift_range=0.2)

# Let's run the generator, where every image is a random augmentation
step=0
for x in datagen.flow(batch, batch_size=1):
        step += 1
        if step > 6: break
        plt.figure()
        # the augmentation operation will change the pixel data to float
        # change it back to uint8 for displaying the image
        plt.imshow(x[0].astype(np.uint8))
```

Brightness

In the code below, we use the `brightness_range` parameter to set random values between 0.5 darker) and 2 (brighter). Note that the value can be specified either as a tuple or list of two elements.

```
from keras.preprocessing.image import ImageDataGenerator
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Let's make a batch of 1 image (apple)
image = cv2.imread('apple.jpg')
batch = np.asarray([image])

# Create a data generator for augmenting the data
datagen = ImageDataGenerator(brightness_range=(0.5, 2))

# Let's run the generator, where every image is a random augmentation
step=0
for x in datagen.flow(batch, batch_size=1):
        step += 1
        if step > 6: break
        plt.figure()
        # the augmentation operation will change the pixel data to float
        # change it back to uint8 for displaying the image
        plt.imshow(x[0].astype(np.uint8))
```

As a final note, transformations like brightness that add a fixed amount to the pixel value are done after normalization or standardization. If done before, then normalization would squash the values into the same original range, undoing the transformation.

Next

In the next part, we will cover the principal behind and best practices for curating a dataset.

Part 9 - Data Curation

Overview - You Only Get What You Trained For

As a data scientist and educator, I get a lot of questions from software engineers on how to improve the accuracy of a model. The five basic answers to increase the performance of the model are:

- Increase training time.
- Increase the depth of the mode.
- Add regularization.
- Expand dataset with data augmentation.
- Increase hyperparameter tuning.

While the above may or may not improve accuracy, the limitation ultimately is in the dataset used to train the model. That's what I am going to cover here --the why?

We need to go back to some basic statistics, you likely studied in high school or college, to understand the why. First, the term "model" was not created by AI, or by machine learning, or otherwise a new thing. The term "model" originates from statistics. As a software engineer, you're used to coding an algorithm that generally has a one-to-one relationship between the input and output --we typically refer to this as the inputs have a linear relationship to the output; or in otherwise, the output is deterministic.

The field of statistics deals with algorithms that are not deterministic. These algorithms are called models --which model a behavior to make an output (outcome) prediction over a probability distribution. That sure sounds like statistics, right! In this part, we will cover distributions, particularly population, sub-population and sampling distributions --in how they affect the training of a model.

Before we jump into this, let's discuss one more thing, the origination of the term 'machine learning'. In statistics, these models may start initially simple, like a linear or logistic regression, but quickly can become complex when modeling a complex behavior, such as forecasting the weather. The term machine learning was coined by Arthur Samuel in 1959 at IBM as a researcher in artificial intelligence. The term became widely recognized by the UCI machine learning repository. That's a well-known repository of datasets used by researchers and academics.

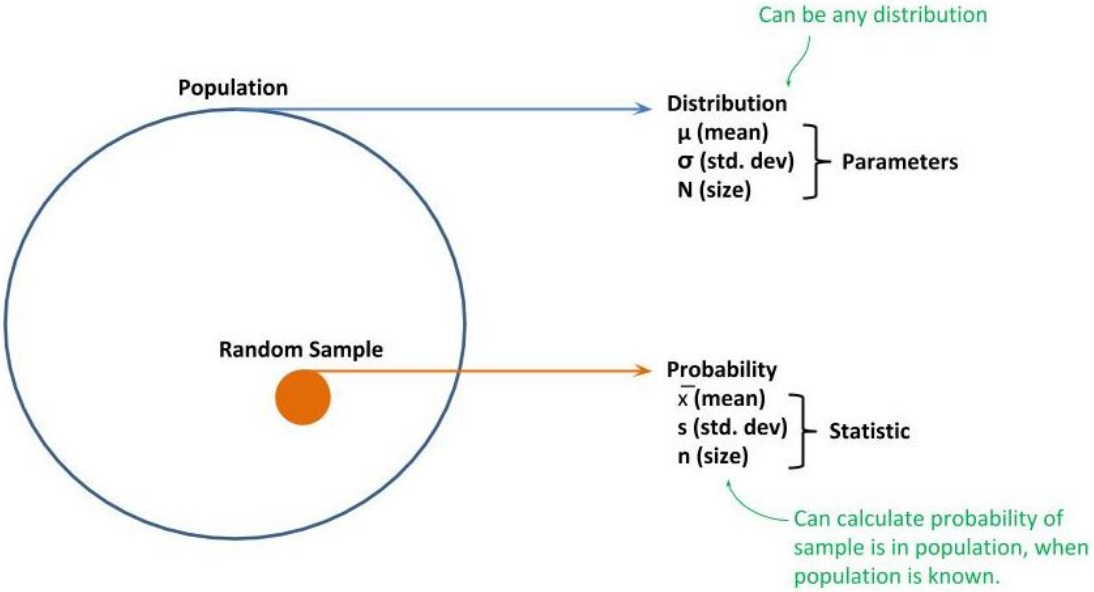
The advent of deep learning using neural networks to develop models is from the field of artificial intelligence. In recent years, these two separate (sub) fields have fused together and we collectively call both now machine learning. But whether you are doing what I refer to as classical machine learning (statistics) or deep learning, the limitation in what you can model (learn) is with the dataset.

For this purpose I will use the MNIST dataset. While normally I consider this dataset outdated, the MNIST dataset is small enough that I can demonstrate all these concepts with it and leave you with code snippets which you can reproduce and see with your own eyes why (and how) the data is the limitation.

Population Distribution

This is important and likely the primary source when you make a model and it turns out to not perform as you expected “in the wild” (i.e., production). If you were to build a model to predict the shoe size of an adult male in the United States, then the population distribution of this model would be *all* adult males in the United States, and their different shoe sizes and corresponding features (e.g., height, ethnicity, etc) would be the distribution of shoe sizes in the population. That’s what we want to model and predict.

The problem of course is that you would not have data for all adult males in the United States. Instead, you will have some subset of data; whereby, one takes batches of the data at random, which we call a random sample, to determine a distribution within the batch.

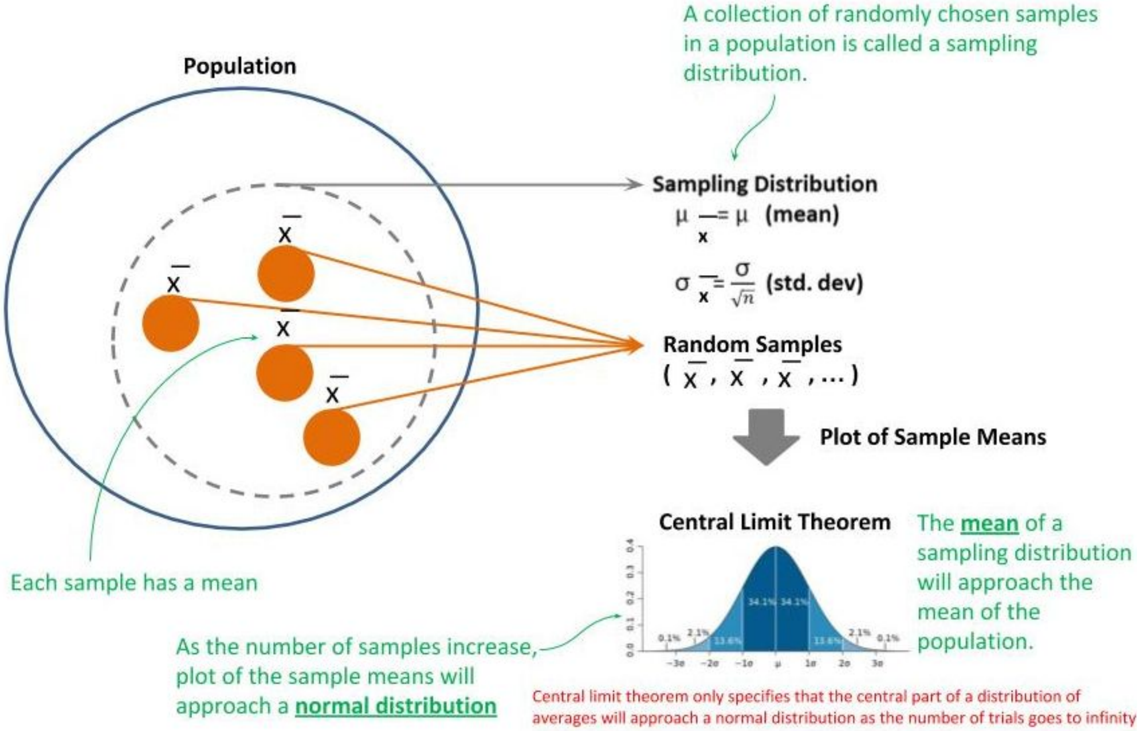


Population vs Sample

Sampling Distribution

The goal is to have enough random samples of the population; that collectively the distributions within these samples can be used to predict the distribution within the population as a whole;

which is referred to as a sampling distribution. The keyword here is “predict”, meaning we are determining a probabilistic distribution vs. a deterministic distribution.

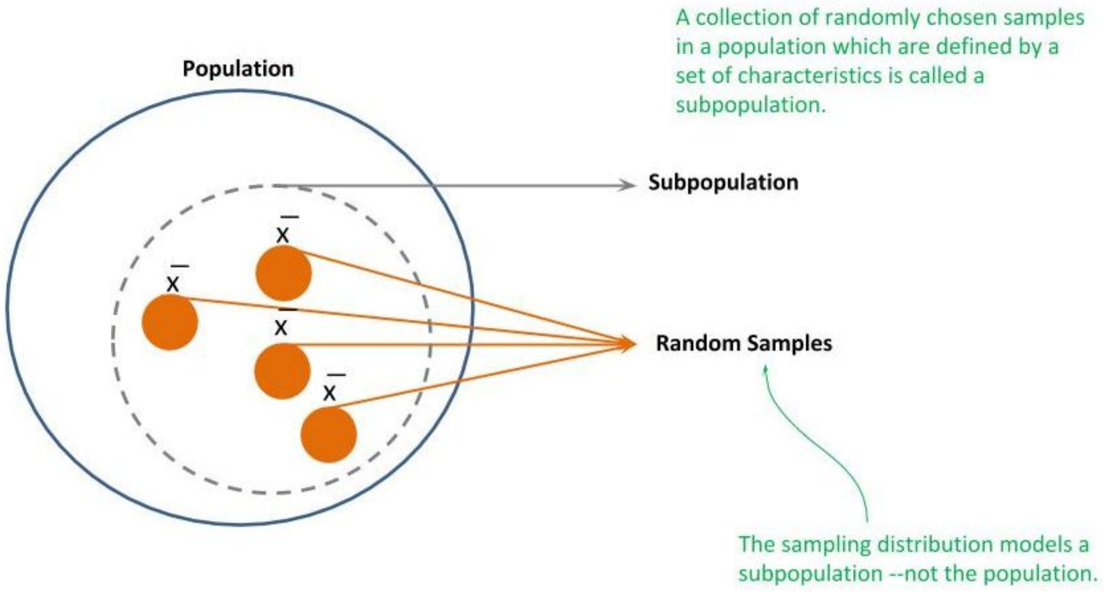

A collection of randomly chosen samples in a population is called a sampling distribution.
Population
Sampling Distribution
$\mu_{\bar{x}} = \mu$ (mean)
$\sigma_{\bar{x}} = \frac{\sigma}{\sqrt{n}}$ (std. dev)
$\bar{x}$
$\bar{x}$
$\bar{x}$
$\bar{x}$
Random Samples
$(\bar{x}, \bar{x}, \bar{x}, \dots)$
Plot of Sample Means
Central Limit Theorem
The mean of a sampling distribution will approach the mean of the population.
Each sample has a mean
As the number of samples increase, plot of the sample means will approach a normal distribution
34.1%
34.1%
13.6%
13.6%
2.1%
2.1%
0.1%
0.1%
-3σ
-2σ
-1σ
μ
1σ
2σ
3σ
Central limit theorem only specifies that the central part of a distribution of averages will approach a normal distribution as the number of trials goes to infinity

Sampling Distribution

Subpopulation Distribution

What you need to understand, is that regardless of how large and comprehensive your dataset is, it is likely a sampling distribution of a subpopulation and not the population. A subpopulation is a subset of a population that is defined by a set of characteristics, which would not have the same probability distribution of the population. As in our earlier adult male shoe example, let’s assume our samples are all from a chain of stores that specialize in selling sports shoes to professional athletes. While with sufficient samples, we can develop a sampling distribution which is representative (predictive) of the subpopulation, it is unlikely to be representative of the population.

This is not the same as a bias, if our intent is to model the subpopulation instead of the population. A bias is when batches of random samples, no matter how many we draw from, the corresponding sampling distribution will not be representative of the population or subpopulation we are modeling.



A collection of randomly chosen samples in a population which are defined by a set of characteristics is called a subpopulation.

The sampling distribution models a subpopulation --not the population.

Subpopulation Distribution

Curated Datasets

Let’s start demonstrating these concepts with the MNIST dataset. MNIST is a dataset of 70,000 images of handwritten digits, proportionally balanced across each digit. It’s super easy to train a model to get near 100% accuracy on the dataset (and hence why it’s the hello world example of machine learning). But almost all “in the wild” application of the trained model will fail --because the distribution of images in MNIST is a subpopulation.

MNIST is a curated dataset. That is, the data curator selected samples for inclusion whose characteristics meet a definition. In the case of MNIST, each sample is a 28x28 pixel image, with the digit centered in the middle, the digit is white and the background is gray, and there is at least a 4 pixel padding around the digit.

Housekeeping - Setting up the Environment

Let’s first do what I call housekeeping. Below is a code snippet we will use throughout our examples. It includes importing the Keras framework for designing/training models, various Python libraries we will use, and finally the loading of the MNIST dataset that is prebuilt into the Keras framework.

```
# Keras classes we will use to build models
from keras import Sequential, Input
from keras.layers import Flatten, Dense, Activation, ReLU, Conv2D, MaxPooling2D,
Dropout
from keras.utils import to_categorical
```

```
# Keras library for the builtin MNIST dataset
from keras.datasets import mnist

# Other python modules for preparation/manipulating the images
import numpy as np
import random
import cv2

# Get the builtin dataset from Keras
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

The dataset from Keras is in a generic format, so we need to do some initial data preparation to use it for training a deep neural network (DNN) and convolutional neural network (CNN). These include:

- The pixel data (x_train and x_test) are in the original INT8 values (0 .. 255). We will normalize the pixel data to be between 0 and 1 as a FLOAT32.
- The image data are matrices of shape Height x Width (H x W). Keras expects tensors in the shape of Height x Width x Channel. These are grayscale images, so we will reshape the train and test data to (H x W x 1).
- The labels (y_train and y_test) are the original scalar value. We will one-hot encode them into vectors.

We are also going to set aside a copy of the test and training data before it's been prepared (to be discussed shortly).

```
# Let's set aside a copy of the original test data and training data
x_test_copy  = x_test
x_train_copy = x_train

# Let's normalize the training/test data and cast to float32
x_train = (x_train / 255.0).astype(np.float32)
x_test  = (x_test  / 255.0).astype(np.float32)

# Let's reshape for Keras from (H,W) to (H,W,C)
x_train = x_train.reshape(-1, 28, 28, 1)
x_test  = x_test.reshape(-1, 28, 28, 1)

# this will output (60000, 28, 28, 1) and (10000, 28, 28, 1)
print("x_train", x_train.shape, "x_test", x_test.shape)

# We need to convert the labels to one-hot-encoding
y_train = to_categorical(y_train)
```

```
y_test  = to_categorical(y_test)

# this will output (60000, 10) and (10000, 10)
print("y_train", y_train.shape, "y_test", y_test.shape)
```

The Challenge (“In the Wild”)

In addition to randomly choosing test data (holdout set) from this curated dataset, we will create two additional test datasets as examples of what the trained model may see in the wild. We will use these two additional test datasets to demonstrate how the model will fail, ways we might modify the training and dataset to overcome this, and the limitations. The two additional test datasets are:

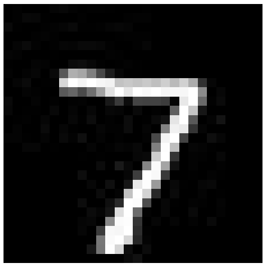
- Inverted - The pixel data is inverted such that the images are now gray digits on white background.
- Shifted - The images are shifted 4 pixels to the right, and thus are not centered anymore. Since there is at least a padding of 4 pixels, none of the digits will be clipped.

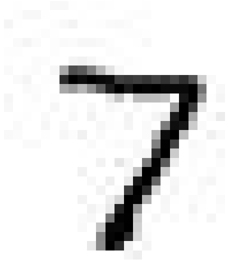
In the code below, we make our two additional test datasets from the copy of the original test dataset.

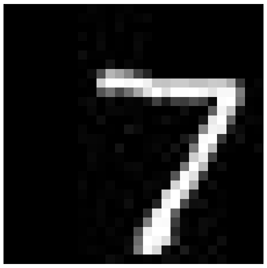
```
# This is the inverted test dataset
x_test_invert = np.invert(x_test_copy)
x_test_invert = (x_test_invert / 255.0).astype(np.float32)

# This is the shifted test dataset
x_test_shift = np.roll(x_test_copy, 4)
x_test_shift = (x_test_shift / 255.0).astype(np.float32)

# Let's reshape for Keras from (H,W) to (H,W,C)
x_test_invert = x_test_invert.reshape(-1, 28, 28, 1)
x_test_shift  = x_test_shift.reshape(-1, 28, 28, 1)
```

Below are an example of a single test image from the original test data, the inverted test data and the shifted test data.

Original **Inverted** **Shifted**







Training as a DNN

MNIST is so easy, we can build a classifier with 97%+ accuracy with a DNN (w/o the need for a CNN). Below is a code example of a function for constructing simple DNNs, consisting of:

- The parameter nodes is a list specifying the number of nodes per layer.
- The input to the DNN are images in the shape 28x28x1
- The input is flattened into a 1D vector of length 784.
- There is an optional dropout (for regularization) after each layer.
- The last dense layer of 10 nodes with a softmax activation function is the classifier.

```
def DNN(nodes, dropout=False):
  model = Sequential()
  model.add(Flatten(input_shape=(28, 28, 1)))
  for n_nodes in nodes:
    model.add(Dense(n_nodes))
    model.add(ReLU())
    if dropout:
      model.add(Dropout(0.5))
      dropout /= 2.0
  model.add(Dense(10))
  model.add(Activation('softmax'))

  # For a multi-class classification problem
  model.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['accuracy'])
  model.summary()
  return model
```

For our first test, we will train the dataset on a single layer (excluding the output layer) of 512 nodes.

```
model = DNN([512])
model.fit(x_train, y_train, epochs=10, batch_size=32, shuffle=True, verbose=2)
score = model.evaluate(x_test, y_test, verbose=1)
print("test", score)
```

The output from `model.summary()`:

```
Layer (type)                 Output Shape              Param #
=================================================================
flatten_1 (Flatten)          (None, 784)               0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               401920
_________________________________________________________________
re_lu_1 (ReLU)               (None, 512)               0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                5130
_________________________________________________________________
activation_1 (Activation)    (None, 10)                0
=================================================================
Total params: 407,050
```

The number of trainable parameters is a measurement of the complexity of our model, which is 408K parameters. We train it for a total of 10 epochs --that’s 10 times we feed the entire training data through the model. Below is the output from the training. The training accuracy quickly reaches 99%+ and our accuracy on the test (holdout) data is nearly 98%.

```
Epoch 1/10
2019-02-08 12:14:59.065963: I tensorflow/core/platform/cpu_feature_guard.cc:141]
Your CPU supports instructions that this TensorFlow binary was not compiled to use:
AVX2 AVX512F FMA
 - 5s - loss: 0.2007 - acc: 0.9409
Epoch 2/10
 - 5s - loss: 0.0897 - acc: 0.9743
Epoch 3/10
 - 5s - loss: 0.0651 - acc: 0.9817
Epoch 4/10
 - 5s - loss: 0.0517 - acc: 0.9853
Epoch 5/10
 - 5s - loss: 0.0419 - acc: 0.9887
Epoch 6/10
 - 5s - loss: 0.0341 - acc: 0.9913
Epoch 7/10
 - 5s - loss: 0.0273 - acc: 0.9928
Epoch 8/10
```

```
 - 5s - loss: 0.0236 - acc: 0.9939
Epoch 9/10
 - 5s - loss: 0.0188 - acc: 0.9953
Epoch 10/10
 - 5s - loss: 0.0163 - acc: 0.9961
10000/10000 [==============================] - 0s 21us/step

test [0.11250439590732676, 0.9791]
```

So far looks good. Let's now try the model on the inverted and shifted test datasets.

```
score = model.evaluate(x_test_invert, y_test, verbose=1)
print("inverted", score)
score = model.evaluate(x_test_shift, y_test, verbose=1)
print("shifted", score)
```

Below is the output. Our accuracy on the inverted is only 2%, and on the shifted it does better but only 41%.

```
inverted [15.660332287597656, 0.0206]
shifted [7.46930496673584, 0.4107]
```

What happened? For the inverted, it looks like our model learned the gray background and the whiteness of the digit as part of the digit recognition. Thus, when we inverted it, it totally failed to classify it.

For the shifted, a dense layer does not preserve spatial relationships between the pixels. Each pixel is a unique feature. Thus, even the shift of a few pixels was enough to dramatically drop the accuracy.

So an approach one might guess to improve the accuracy is to increase the number of nodes in the input layer --more nodes, better learning. Let's repeat the same test with 1024 nodes.

```
model = DNN([1024])
model.fit(x_train, y_train, epochs=10, batch_size=32, shuffle=True, verbose=2)
score = model.evaluate(x_test, y_test, verbose=1)
print("test", score)
```

The output from `model.summary()`:

```
Layer (type)                 Output Shape              Param #
```

```
=================================================================
flatten_2 (Flatten)          (None, 784)               0
_________________________________________________________________
dense_3 (Dense)              (None, 1024)              803840
_________________________________________________________________
re_lu_2 (ReLU)               (None, 1024)              0
_________________________________________________________________
dense_4 (Dense)              (None, 10)                10250
_________________________________________________________________
activation_2 (Activation)    (None, 10)                0
=================================================================
Total params: 814,090
Trainable params: 814,090
```

You can see by doubling the number of nodes on the input layer we double the computational complexity, i.e., the number of trainable parameters. Let’s see if this improves the accuracy on our alternate test data.

Nope, we see a marginal increase on the inverted to about 5%, but it’s so low that’s probably just noise, and the accuracy on the shifted is about the same at 40%. So increasing the number of nodes in the input layer (widening) did not aid in either filtering out (not learning) the background and whiteness of the digits or learning the spatial relationships.

```
inverted [15.157325344848633, 0.0489]
shifted [7.736222146606445, 0.4038]
```

So another approach one might guess is to increase the number of layers. This time let’s make the DNN with two 512 node layers.

```
model = DNN([512, 512])
model.fit(x_train, y_train, epochs=10, batch_size=32, shuffle=True, verbose=2)
score = model.evaluate(x_test, y_test, verbose=1)
print("test", score)
```

The ending output from `model.summary()`:

```
Total params: 669,706
Trainable params: 669,706
```

Let’s see if this improves the accuracy on our alternate test data.

```
inverted [14.464950880432129, 0.1025]
shifted [8.786513813018798, 0.3887]
```

We see another slight increase in the shifted to 10%. But did it really improve. We have 10 classes (digits). If we made random guesses we be right 10% of the time. This is still purely a random outcome --nothing learned here. Looks like adding layers did not aid in learning the spatial relationships either.

Another approach would be to add some regularization to prevent overfitting the model to the training data and be more generalized. We will use the same two layer DNN of 512 nodes per layer, and add a 50% dropout after the first layer and 25% dropout after the second layer. It's a common practice to use a higher dropout at the first layer where the layer is learning coarse features and smaller dropout at subsequent layers which are learning finer features.

```
model = DNN([512, 512], True)
model.fit(x_train, y_train, epochs=10, batch_size=32, shuffle=True, verbose=2)
score = model.evaluate(x_test, y_test, verbose=1)
print("test", score)
```

Let's see if this improves the accuracy on our alternate test data.

```
inverted [15.862942279052735, 0.0144]
shifted [8.341207506561279, 0.3965]
```

Nope, no improvement. Thus widening a layer, adding layers, and regularization did not help in training the model to recognize the digits in our alternate test datasets.

Training as a CNN

Okay, it's time to use a convolutional neural network. With convolutional layers, we should at least learn the spatial relationships. Perhaps the convolutional layers will also filter out the background as well as the whiteness of the digits.

Below is the code for constructing our CNNs:

- The parameter filters is a list specifying the number of filters per convolution.
- The input to the CNN are images in the shape 28x28x1
- A max pooling which reduces the feature map sizes by 75% after each convolution.
- A dropout (regularization) of 25% after each convolution/max pooling layer.
- The last dense layer of 10 nodes with a softmax activation function is the classifier.

```python
def CNN(filters):
  model = Sequential()
  first = True
  for n_filters in filters:
    if first:
      model.add(Conv2D(n_filters, (3, 3), strides=1, input_shape=(28, 28, 1)))
    else:
      model.add(Conv2D(n_filters, (3, 3), strides=1))
    model.add(ReLU())
    model.add(MaxPooling2D((2, 2), strides=2))
    model.add(Dropout(0.25))
  model.add(Flatten())
  model.add(Dense(10))
  model.add(Activation('softmax'))

  # For a multi-class classification problem
  model.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['accuracy'])
  model.summary()
  return model
```

Let’s start with a CNN with a single convolutional layer of 16 filters.

```python
model = CNN([16])
model.fit(x_train, y_train, epochs=10, batch_size=32, shuffle=True, verbose=2)
score = model.evaluate(x_test, y_test, verbose=1)
print("test", score)
```

The output from `model.summary()`:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 16)        160
_________________________________________________________________
re_lu_1 (ReLU)               (None, 26, 26, 16)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 13, 13, 16)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 13, 13, 16)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 2704)              0
_________________________________________________________________
```

```
dense_1 (Dense)              (None, 10)                 27050
_________________________________________________________________
activation_1 (Activation)    (None, 10)                 0
=================================================================
Total params: 27,210
Trainable params: 27,210
```

Below is the result from our training of the CNN:

```
test [0.05741905354047194, 0.9809]
```

You can see we can get comparable accuracy (98%) on the test data with a CNN with a lot less trainable parameters (27K vs 400K+).

Let’s see if this improves the accuracy on our alternate test data.

```
inverted [2.1893138484954835, 0.5302]
shifted [2.231996842956543, 0.5682]
```

Yes, it made some measurable difference. We can see we went from a previous high of 10% accuracy on the inverted data to 50% accuracy. Thus, it does seem the convolutional layers help filter out (not learn) the background or whiteness of the digits. But it’s still far too low in accuracy. For the shifted, we increased to 57%. Still below our target, but we can also see now have the convolutional layers are learning the spatial relationships.

If one convolutional layer improved things, let’s see how much better we can do with two convolutional layers. We will use two layers, the first with 16 filters and the second with 32 filters. It’s a common practice to double the number of filters as you get successively deeper into a CNN.

```
model = CNN([16, 32])
model.fit(x_train, y_train, epochs=10, batch_size=32, shuffle=True, verbose=2)
score = model.evaluate(x_test, y_test, verbose=1)
print("test", score)
```

Below is the result from our training of the CNN:

```
test [0.03628469691830687, 0.9882]
```

Again, we get comparable with slight improvement in accuracy of ~99% on the test data. Let's see if by adding convolutional layers will improve the accuracy on our alternate test data.

```
inverted [1.2761547603607177, 0.6332]
shifted [0.6951200264453888, 0.7679]
```

We do see some incremental improvement. Our inverted test data went up to 63%. So it's more learning to filter out the background and whiteness of the digits, but still not there. Our shifted test jumped to 76%. So you can see how convolutional layers are learning the spatial relationships in the digits vs. position in the image (compared to a DNN).

Image Augmentation

Finally, let's use image augmentation. Image augmentation is a process for generating new samples from existing samples by making some small modifications, such that the modification would not change what the image would be classified as, and the image would be still recognized by the human eye as being that class.

In addition to adding more samples to the training set, certain types of augmentation can also aid in generalizing the model to accurately classify images outside of the test (holdout) dataset that it would've otherwise failed on.

As we saw on our above CNN, we still had insufficient accuracy on shifted images, thus our model has not fully learned the spatial relationships of the digits separate from the location and background in the image. We could add more filters and convolutional layers in an attempt to increase the accuracy on shifted images. This would make the model more computational complex, longer to train, larger memory print, and longer latency when deployed to do predictions (inference).

Alternately, we are going to improve this by using image augmentation to randomly shift the image left or right upto 20%. Since our images are 28 pixels wide, 20% would mean that the image gets shifted a maximum of six pixels in either direction. We have a minimum of a four pixel boundary, so there will be little to no clipping of the digits.

We will use the `ImageDataGenerator` class in Keras to do the image augmentation. In the code example below, we do the following:

- Create the same CNN model as before.
- Instantiate an `ImageDataGenerator` generator object where the parameter `width_shift_range=0.2` will augment the dataset during training by randomly shifting images +/- 20%.

- We invoke the `fit_generator()` method to train the model using our image augmentation generator with our existing training data.
- We specify the number of `steps_per_epoch` in the generator as the number of training samples divided by the batch size; otherwise, the generator would loop indefinitely on the first epoch.

```
from keras.preprocessing.image import ImageDataGenerator

model = CNN([16, 32])
datagen = ImageDataGenerator(width_shift_range=0.2)
model.fit_generator(datagen.flow(x_train, y_train, batch_size=32),
                    steps_per_epoch= 60000 // 32 , epochs=10)
score = model.evaluate(x_test, y_test, verbose=1)
print("test", score)
```

Below is the result from our training of the CNN:

```
test [0.046405045648082156, 0.986]
```

Let’s see if this improves the accuracy on our alternate test data.

```
inverted [4.463096208190918, 0.2338]
shifted [0.06386796866590157, 0.9796]
```

Wow, our accuracy on the shifted data is now nearly 98%. So we were able to train the model to learn the spatial relationships of digits when their shifted in the image without increasing the complexity of the model.

Let’s now tackle training the model to filter out the background and whiteness of the digits. In the code below we take 10% of the training data (`x_train_copy[0:6000]`) and invert it like we did with the test data. Why 10% instead of the whole training data? When we want to train a model to filter out something, we generally can do it with as little as 10% of the distribution of the entire training data.

Next we combine the original training data with the additional inverted training data by appending the two training sets together (both x_train --the data, and y_train --the labels), for a total of 66,000 images (vs. 60,000) in our training set.

```
# Select 10% of the (copy of) training data and invert it.
x_train_invert = np.invert(x_train_copy[0:6000])
x_train_invert = (x_train_invert / 255.0).astype(np.float32)
x_train_invert = x_train_invert.reshape(-1, 28, 28, 1)

# Select the same 10% of the corresponding labels
y_train_invert = x_train[0:6000]

# Next, combine the two training datasets into a single training set
x_combine = np.append(x_train, x_train_invert, axis=0)
y_combine = np.append(y_train, y_train_invert, axis=0)

model = CNN([16, 32])
datagen = ImageDataGenerator(width_shift_range=0.2)
datagen.fit(x_train_combine)
model.fit_generator( datagen.flow(x_combine, y_combine, batch_size=32),
steps_per_epoch= 66000 // 32 , epochs=10)
score = model.evaluate(x_test, y_test, verbose=1)
print("test", score)
```

Below is the result from our training of the CNN:

```
test [0.04763028650498018, 0.9847]
```

Let’s see if this improves the accuracy on our alternate test data.

```
inverted [0.13941174189522862, 0.9589]
shifted [0.06449916120804847, 0.979]
```

Wow, our test accuracy on the inverted images is nearly 96%.

Final Test

As a final test, I randomly selected “in the wild” images of a hand-drawn single digit from a Google image search. These included images which were colored, drawn with a felt pen, a paint brush, another with a crayon drawn by a young child, etc. After I did my “in the wild” testing, I got only 40% accuracy with the above trained CNN.

Why only 40% and how would one diagnose the cause? The question should be ‘what subpopulation distribution’ did the model learn? Did the model learn to generalize the contours of the digits independent of the contrast to the background; or did it simply learn digits are either white or black? What would happen if one tested with a black digit on a gray background (instead of white)?

The training and test data from MNIST are digits drawn with a pen or pencil, where the lines are thin. Some of my “in the wild images” were thicker from a felt pen to paint brush to crayon. Did the model learn to generalize the thickness of the lines? What about texture? The crayon and paint drawn digits had uneven texture --were these differences in texture got learned as edges in the convolutional layers?

As a final example. Presume you developed a model for use in a factory to detect defects in parts, where the camera is in a fixed position and perspective over a gray colored conveyor belt, with ridges running down it. All works well until one day the conveyor belt is replaced and the owner replaced it with a nice yellow smooth belt to add some color to the factory, and now the defect detection model fails. What happened?