How to easily build a Dog breed Image classification model

> If you're impatient, scroll to the bottom of the post for the Github Repos

Who's a good dog? Who likes ear scratches? Well, it seems those fancy deep neural networks don't have *all* the answers. However, maybe they can answer that ubiquitous question we all ask when meeting a four-legged stranger: what kind of good pup is that?

In this tutorial, we'll walk through building a deep neural network classifier capable of determining a dog's breed from a photo using the

Dog Breed dataset. We'll walk through how to train a model, design the input and output for category classifications, and finally display the accuracy results for each model.

Image Classification

background clutter etc.

How might we go about writing an algorithm that can classify images into distinct categories? Computer Vision researchers have come up with a data-driven approach to solve this. Instead of trying to specify what every one of the image categories of interest looks like directly in code, they provide the computer with many examples of each image class and then develop learning algorithms that look at these examples and learn about the visual appearance of each class. In other words, they first accumulate a training dataset of labeled images, then feed it to the computer in order for it to get familiar with the data.

Given that fact, the complete image classification pipeline can be formalized as follows:

- Our input is a training dataset that consists of *N* images, each labeled with one of *K* different classes.
- Then, we use this training set to train a classifier to learn what every one of the classes looks like.
- In the end, we evaluate the quality of the classifier by asking it to predict labels for a new set of images that it has never seen before. We will then compare the true labels of these images to the ones predicted by the classifier.

Image Classification Using Machine Learning

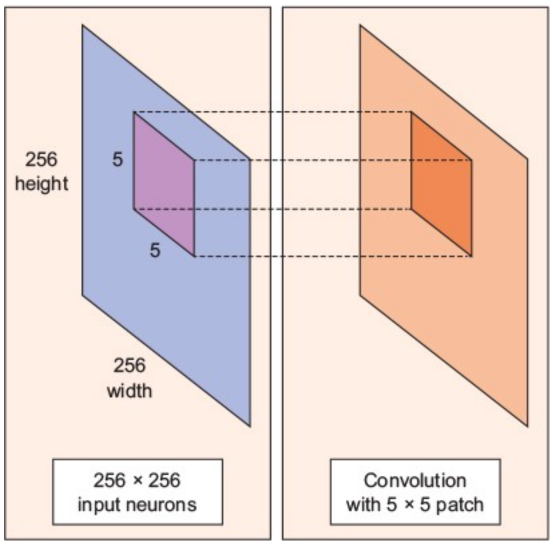
A machine learning image processing approach to image classification involves identifying and extracting key features from images and using them as input to a machine learning model. Image classification, technically speaking, is a machine learning method and it is designed to resemble the way a human brain functions. With this method, the computers are taught to recognize the visual elements within an image. By relying on large databases and noticing emerging patterns, the computers can make sense of images and formulate relevant tags and categories.

Convolutional neural networks

having fewer parameters greatly improves the time it takes to learn as well as reduces the amount of data required to train the model.

Instead of a fully connected network of weights from each pixel, a CNN has just enough weights to look at a small patch of the image. It's like reading a book by using a magnifying glass; eventually, you read the whole page, but you look at only a small patch of the page at any given time.

Consider a 256 × 256 image. Instead of processing the whole image at once, CNN can efficiently scan it chunk by chunk — say, a 5 × 5 window. The 5 × 5 window slides along the image (usually left to right, and top to bottom), as shown in the figure below. How "quickly" it slides is called its stride length. For example, a stride length of 2 means the 5 × 5 sliding window moves by 2 pixels at a time until it spans the entire image. This 5 x 5 window has an associated 5 x 5 matrix of weights.



The sliding-window shenanigans happen in the convolution layer of the neural network. A typical CNN has multiple convolution layers. Each convolutional layer typically generates many alternate convolutions, so the weight matrix is a tensor of 5 × 5 × n, where n is the number of convolutions.

The beauty of the CNN is that the number of parameters is independent of the size of the original image. You can run the same CNN on a 300 × 300 image, and the number of parameters won't change in the convolution layer!

Dog Breed Dataset

The dataset that we'll be working on can be accessed here. We are provided a training set and a test set of images of dogs. Each image has a filename that is its unique id. The dataset comprises 120 breeds of dogs. To make it simpler, we'll reduce the dataset with the 8 main breeds. The tutorial below shows how to use TensorFlow to build a simple CNN with 3 convolutional layers to classify the dog breeds.

Data Processing

1 — Packages

Let's import all the packages needed.

```
%matplotlib inline
import matplotlib.pyplot as plt

import tensorflow as tf
import numpy as np
import pandas as pd

import time
from datetime import timedelta

import math
import os

import scipy.misc
from scipy.stats import itemfreq
from random import sample
import pickle

from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split

# Image manipulation
import PIL.Image
from IPython.display import display

# Open a Zip File
```

2 — Unzip Files

We need now to extract the train and test files from the zip. This is the code:

```
# We unzip the train and test zip file

archive_train = ZipFile("Data/train.zip", 'r')

archive_test = ZipFile("Data/test.zip", 'r')
```

```
len(archive_train.namelist()[:]) - 1
```

The last line of code should return a value of 10,222.

3 — Resize and normalize data

The function below creates a pickle file to save all the images unzipped.

```
# This function help to create  a pickle file gathering all the image from a
def DataBase_creator(archivezip, nwidth, nheight, save_name):
    # We choose the archive (zip file) + the new width and height for all the

    # Start-time used for printing time-usage below.
    start_time = time.time()

    # nwidth x nheight = number of features because images have nwidth x nhe
    s = (len(archivezip.namelist()[:])-1, nwidth, nheight,3)
    allImage = np.zeros(s)
    for i in range(1,len(archivezip.namelist()[:])):
        filename = BytesIO(archivezip.read(archivezip.namelist()[i]))
        image = PIL.Image.open(filename) # open colour image
        image = image.resize((nwidth, nheight))
        image = np.array(image)
        image = np.clip(image/255.0, 0.0, 1.0) # 255 = max of the value of a
        allImage[i-1]=image

    # we save the newly created data base
    pickle.dump(allImage, open( save_name + '.p', "wb" ) )

    # Ending time.
    end_time = time.time()
    # Difference between start and end-times.
    time_dif = end_time - start_time
    # Print the time-usage.
```

We then define the new image size applied for all images and call the function above.

```
image_resize = 60

DataBase_creator(archivezip = archive_train, nwidth =
image_resize, nheight = image_resize , save_name = “train")

DataBase_creator(archivezip = archive_test, nwidth =
image_resize, nheight = image_resize , save_name = "test")
```

If using a laptop with CPU, we should see the time usage to be around 40 seconds for the train zip file and 41 seconds for the test zip file.

We have now a train and test pickle files. Next time we open this code in a Jupyter Notebook, we can load them directly and the step above can be skipped if we relaunch the code later.

```
# load TRAIN

train = pickle.load( open( "train.p", "rb" ) )
```

```
# load TEST

test = pickle.load( open( "test.p", "rb" ) )

test.shape
```

The shape of the test data should be (10357, 60, 60, 3).

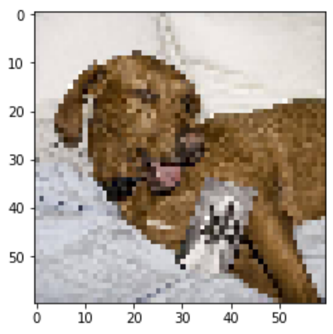
All the images do not have the same shape. For our model, we need to resize them to the same shape. We use the common practice to reshape them as a square. We also need to normalize our dataset by dividing by

255 all the pixel values. The new pixels values will be in the range [0,1].

```
image_resize = 60
nwidth = image_resize
nheight = image_resize

# nwidth x nheight = number of features because images are nwidth x nheight
s = (len(df_train['breed']), nwidth, nheight,3)
allImage = np.zeros(s)

i = 0
for f, breed in df_train.values:
    image = PIL.Image.open('../input/train/{}.jpg'.format(f))
    image = image.resize((nwidth, nheight))
    image = np.array(image)
    image = np.clip(image/255.0, 0.0, 1.0) # 255 = max of the value of a pix
    i += 1
    allImage[i-1]=image

```

Let's check one image from the training dataset:

```
lum_img = train[100,:,:,:]

plt.imshow(lum_img)

plt.show()
```



4 — Check out labels file

Now let's zoom in on the label CSV file from train data.

	id	breed
5662	8eaef67a7269e3e766cce9ce9a1a9ebe	african_hunting_dog
8024	c8c63044e46f67dc58bc676e99ac8a4d	chow
3264	511bfe35ff282294f6129c55bd6c33f6	malinois
2281	38da992c0b9fbcac4127a80fc70708fd	italian_greyhound
137	0341e3d7a4624d6a7b061fdc25b69044	irish_setter

5 — Extract the most represented breeds

We will reduce the database so that we can reduce the complexity of our model. In addition, it will help for the calculation as there will be only N breeds to classify. We will be able to easily run the model in less than 10 minutes.

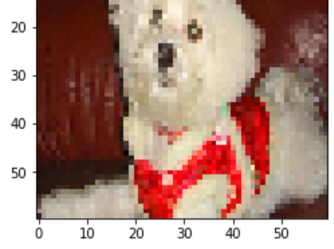
```
Nber_of_breeds = 8

# Get the N most represented breeds
def main_breeds(labels_raw, Nber_breeds , all_breeds='TRUE'):
    labels_freq_pd = itemfreq(labels_raw["breed"])
    labels_freq_pd = labels_freq_pd[labels_freq_pd[:, 1].argsort()[::-1]]

    if all_breeds == 'FALSE':
        main_labels = labels_freq_pd[:,0][0:Nber_breeds]
    else:
        main_labels = labels_freq_pd[:,0][:]

    labels_raw_np = labels_raw["breed"].as_matrix()
    labels_raw_np = labels_raw_np.reshape(labels_raw_np.shape[0],1)
    labels_filtered_index = np.where(labels_raw_np == main_labels)
    return labels_filtered_index

labels_filtered_index = main_breeds(labels_raw = labels_raw, Nber_breeds = N
labels_filtered = labels_raw.iloc[labels_filtered_index[0],:]
train_filtered = train[labels_filtered_index[0],:,:,:]
print('- Number of images remaining after selecting the {0} main breeds : {1
print('- The shape of train_filtered dataset is : {0}'.format(train_filtered
```

We should be able to see this output:

```
- Number of images remaining after selecting the 8 main breeds : (922,)
- The shape of train_filtered dataset is : (922, 60, 60, 3)
```

Let’s look at one image:

```
lum_img = train_filtered[1,:,:,:]

plt.imshow(lum_img)

plt.show()
```



6 — One-Hot Labels

Let’s do one-hot encoding for our labels data.

```
# We select the labels from the N main breeds

labels = labels_filtered["breed"].as_matrix()

labels = labels.reshape(labels.shape[0],1) #labels.shape[0]
looks faster than using len(labels)

labels.shape
```

The labels shape is (922, 1).

```
# Function to create one-hot labels
def matrix_Bin(labels):
    labels_bin=np.array([])

    labels_name, labels0 = np.unique(labels, return_inverse=True)
    labels0

    for _, i in enumerate(itemfreq(labels0)[:,0].astype(int)):
        labels_bin0 = np.where(labels0 == itemfreq(labels0)[:,0][i], 1., 0.)
        labels_bin0 = labels_bin0.reshape(1,labels_bin0.shape[0])

        if (labels_bin.shape[0] == 0):
            labels_bin = labels_bin0
        else:
            labels_bin = np.concatenate((labels_bin,labels_bin0 ),axis=0)

    print("Nber SubVariables {0}".format(itemfreq(labels0)[:,0].shape[0]))
    labels_bin = labels_bin.transpose()
    print("Shape : {0}".format(labels_bin.shape))

```

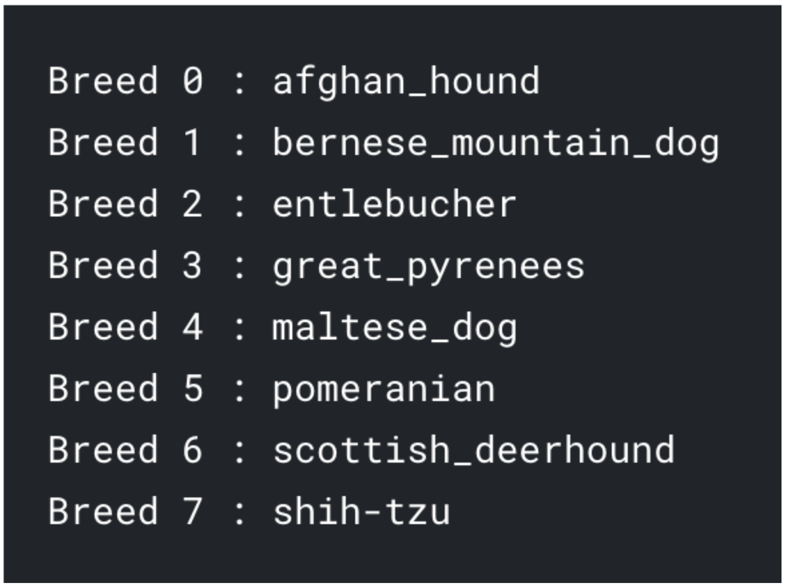
```
labels_name, labels_bin = matrix_Bin(labels = labels)

labels_bin[0:9]
```

```
Nber SubVariables 8
Shape : (922, 8)

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

7 — Quick check on labels

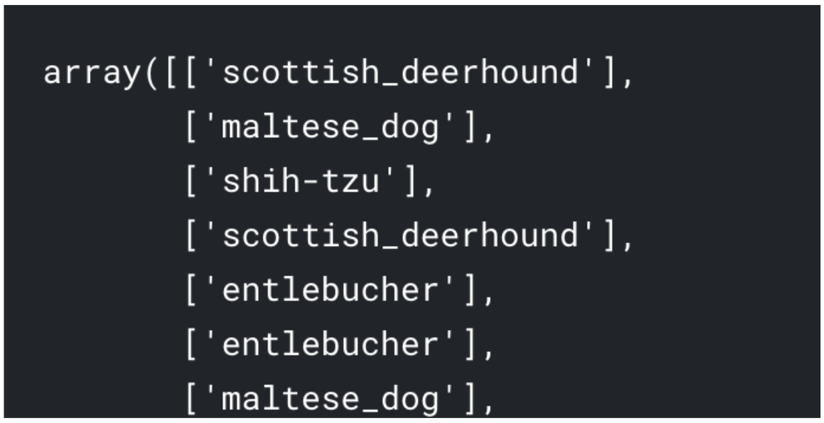
Let's see exactly the N labels we keep. As you will see below from the one-hot labels, you can find which breed it corresponds.

```
for breed in range(len(labels_name)):

    print('Breed {0} : {1}'.format(breed,labels_name[breed]))
```



```
Breed 0 : afghan_hound
Breed 1 : bernese_mountain_dog
Breed 2 : entlebucher
Breed 3 : great_pyrenees
Breed 4 : maltese_dog
Breed 5 : pomeranian
Breed 6 : scottish_deerhound
Breed 7 : shih-tzu
```

```
labels_cls = np.argmax(labels_bin, axis=1)

labels[0:9]
```



```
array([['scottish_deerhound'],
       ['maltese_dog'],
       ['shih-tzu'],
       ['scottish_deerhound'],
       ['entlebucher'],
       ['entlebucher'],
       ['maltese_dog'],
```

Convolutional Neural Networks

1 — Creation of a Train and Validation Data

We split our train data in two: a training set and a validation set. Therefore, we can check the accuracy of the model train made from the 'training set', on the validation set.

```
num_validation = 0.30

X_train, X_validation, y_train, y_validation =
train_test_split(train_filtered, labels_bin,
test_size=num_validation, random_state=6)
```

2 — Creation of a Train and Test Data

Here's the code to split original data to train and test sets:

```
def train_test_creation(x, data, toPred):
  indices = sample(range(data.shape[0]),int(x * data.shape[0]))
  indices = np.sort(indices, axis=None)

  index = np.arange(data.shape[0])
  reverse_index = np.delete(index, indices,0)

  train_toUse = data[indices]
  train_toPred = toPred[indices]
  test_toUse = data[reverse_index]
  test_toPred = toPred[reverse_index]

  return train_toUse, train_toPred, test_toUse, test_toPred

df_train_toUse, df_train_toPred, df_test_toUse, df_test_toPred = train_test_

df_validation_toPred_cls = np.argmax(y_validation, axis=1)
df_validation_toPred_cls[0:9]
```

```
array([5, 1, 5, 6, 7, 2, 5, 1, 1])
```

3 — CNN with TensorFlow — Defining Layers

The architecture will be like this:

- 1st Convolutional Layer with 32 filters
- Max pooling
- Relu
- 2nd Convolutional Layer with 64 filters
- Max pooling
- Relu
- 3rd Convolutional Layer with 128 filters
- Max pooling
- Relu
- DropOut

- DropOut
- Fully Connected Layer with n nodes (n = number of breeds)

Here's a brief explanation of these terms:

- **Convolution Layer:** As explained in the CNN section above, at this layer, we preserve the spatial relationship between pixels by learning image features using small squares of input data. These squares of input data are also called *filters* or *kernels.* The matrix formed by sliding the filter over the image and computing the dot product is called a *Feature Map*. The more number of filters we have, the more image features get extracted and the better our network becomes at recognizing patterns in unseen images.
- **ReLU Layer:** For any kind of neural network to be powerful, it needs to contain non-linearity. ReLU is one such non-linear operation, which stands for Rectified Linear Unit. It is an element-wise operation that replaces all negative pixel values in the feature map by 0. We pass the result from the convolution layer through a *ReLU* activation function.
- **Max Pooling Layer:** After this, we perform a *pooling* operation to reduce the dimensionality of each feature map. This enables us to reduce the number of parameters and computations in the network, therefore controlling overfitting. CNN uses *max-pooling*, in which it defines a spatial neighborhood and takes the largest element from the rectified feature map within that window. After the pooling layer, our network becomes invariant to small transformations, distortions and translations in the input image.
- **Fully-Connected Layer:** After these layers, we add a couple of fully-connected layers to wrap up the CNN architecture. The output from the convolution and pooling layers represent high-level features of the input image. The FC layers use these features for classifying the input image into various classes based on the training dataset. Apart from classification, adding FC layers also helps to learn non-linear combinations of these features.
- **Dropout Layer:** *Dropout* is a regularization technique to help the network avoid overfitting. Basically during training half of neurons on a particular layer will be deactivated. This improves generalization as you force your layer to learn with different neurons. Normally we use Dropout on the fully connected layers, but it is also possible to use dropout after the max-pooling layers, creating some kind of image noise augmentation.

From a bigger picture, a CNN architecture accomplishes 2 major tasks: feature extraction (convolution + pooling layers) and classification (fully-connected layers). In general, the more convolution steps we have, the more complicated features our network will be able to learn to recognize.

Here we define our weights, biases, and other constants.

```
# Our images are 100 pixels in each dimension.
img_size = image_resize

```

```

# Number of classes : 5 breeds
num_classes = Nber_of_breeds

def new_weights(shape):
    return tf.Variable(tf.truncated_normal(shape, stddev=0.05))
    #outputs random value from a truncated normal distribution

def new_biases(length):
    return tf.Variable(tf.constant(0.05, shape=[length]))
```

Here we define our convolution layer.

```
def new_conv_layer(input,              # The previous layer.
                   num_input_channels, # Num. channels in prev. layer.
                   filter_size,        # Width and height of each filter.
                   num_filters,        # Number of filters.
                   use_pooling=True,
                   use_dropout=True):  # Use 2x2 max-pooling.

    # Shape of the filter-weights for the convolution.
    # This format is determined by the TensorFlow API.
    shape = [filter_size, filter_size, num_input_channels, num_filters]

    # Create new weights aka. filters with the given shape.
    weights = new_weights(shape=shape)

    # Create new biases, one for each filter.
    biases = new_biases(length=num_filters)

    # Create the TensorFlow operation for convolution.
    # Note the strides are set to 1 in all dimensions.
    # The first and last stride must always be 1,
    # because the first is for the image-number and
    # the last is for the input-channel.
    # But e.g. strides=[1, 2, 2, 1] would mean that the filter
    # is moved 2 pixels across the x- and y-axis of the image.
    # The padding is set to 'SAME' which means the input image
    # is padded with zeroes so the size of the output is the same.
    layer = tf.nn.conv2d(input=input,
                         filter=weights,
                         strides=[1, 1, 1, 1],
                         padding='SAME')

    # Add the biases to the results of the convolution.
    # A bias-value is added to each filter-channel.
    layer += biases

    # Use pooling to down-sample the image resolution?
    if use_pooling:
        # This is 2x2 max-pooling, which means that we
        # consider 2x2 windows and select the largest value
        # in each window. Then we move 2 pixels to the next window.
        layer = tf.nn.max_pool(value=layer,
                               ksize=[1, 2, 2, 1],
                               strides=[1, 2, 2, 1],
                               padding='SAME')

    # Rectified Linear Unit (ReLU).
    # It calculates max(x, 0) for each input pixel x.
    # This adds some non-linearity to the formula and allows us
    # to learn more complicated functions.
    layer = tf.nn.relu(layer)

    if use_dropout:
        layer = tf.nn.dropout(layer,keep_prob_conv)

    # Note that ReLU is normally executed before the pooling,
    # but since relu(max_pool(x)) == max_pool(relu(x)) we can
    # The shape of the input layer is assumed to be:
    # layer_shape == [num_images, img_height, img_width, num_channels]

    # The number of features is: img_height * img_width * num_channels
    # We can use a function from TensorFlow to calculate this.
    num_features = layer_shape[1:4].num_elements()

    # Reshape the layer to [num_images, num_features].
    # Note that we just set the size of the second dimension
    # to num_features and the size of the first dimension to -1
    # which means the size in that dimension is calculated
    # so the total size of the tensor is unchanged from the reshaping.
    layer_flat = tf.reshape(layer, [-1, num_features])

    # The shape of the flattened layer is now:
    # [num_images, img_height * img_width * num_channels]

    # Return both the flattened layer and the number of features.
```

Here we define our fully-connected layer.

```
def new_fc_layer(input,          # The previous layer.
                 num_inputs,     # Num. inputs from prev. layer.
                 num_outputs,    # Num. outputs.
                 use_relu=True,
                 use_dropout=True): # Use Rectified Linear Unit (ReLU)?

    # Create new weights and biases.
    weights = new_weights(shape=[num_inputs, num_outputs])
    biases = new_biases(length=num_outputs)

    # Calculate the layer as the matrix multiplication of
    # the input and weights, and then add the bias-values.
    layer = tf.matmul(input, weights) + biases

    # Use ReLU?
    if use_relu:
        layer = tf.nn.relu(layer)

    if use_dropout:
        layer = tf.nn.dropout(layer,keep_prob_fc)

```

4 — CNN with TensorFlow — Set up placeholder tensor

Here we set up a placeholder for the tensor in TensorFlow.

```
x = tf.placeholder(tf.float32, shape=[None, img_size,
img_size, num_channels], name='x')

x_image = tf.reshape(x, [-1, img_size, img_size,
num_channels]) #-1 put everything as 1 array

y_true = tf.placeholder(tf.float32, shape=[None,
num_classes], name='y_true')

y_true_cls = tf.argmax(y_true, axis=1)

keep_prob_fc=tf.placeholder(tf.float32)

keep_prob_conv=tf.placeholder(tf.float32)
```

5 — CNN with TensorFlow — Design the layer

In this part, you can play with the filter sizes and the number of filters.
The best model is one with the proper number of layers but also a good

```


# Convolutional Layer 2.
filter_size2 = 4          # Convolution filters are 4 x 4 pixels.
num_filters2 = 64      # There are 64 of these filters.


# Convolutional Layer 3.
filter_size3 = 3          # Convolution filters are 3 x 3 pixels.
num_filters3 = 128      # There are 128 of these filters.


# Fully-connected layer.
fc_size = 500

layer_conv1, weights_conv1 = \
    new_conv_layer(input=x_image,
                   num_input_channels=num_channels,
                   filter_size=filter_size1,
                   num_filters=num_filters1,
                   use_pooling=True,
                   use_dropout=False)

layer_conv2, weights_conv2 = \
    new_conv_layer(input=layer_conv1,
                   num_input_channels=num_filters1,
                   filter_size=filter_size2,
                   num_filters=num_filters2,
                   use_pooling=True,
                   use_dropout=False)

layer_conv3, weights_conv3 = \
    new_conv_layer(input=layer_conv2,
                   num_input_channels=num_filters2,
                   filter_size=filter_size3,
                   num_filters=num_filters3,
                   use_pooling=True,
                   use_dropout=True)

layer_flat, num_features = flatten_layer(layer_conv3)

#Train
layer_fc1 = new_fc_layer(input=layer_flat,
                         num_inputs=num_features,
                         num_outputs=fc_size,
                         use_relu=True,
                         use_dropout=True)

layer_fc2 = new_fc_layer(input=layer_fc1,
                         num_inputs=fc_size,
                         num_outputs=num_classes,
                         use_relu=False
```

6 — CNN with TensorFlow — Cross-entropy loss

Here we define our loss function to train our model.

```
cross_entropy =
tf.nn.softmax_cross_entropy_with_logits(logits=layer_fc2,
labels=y_true)

cost = tf.reduce_mean(cross_entropy)

optimizer = tf.train.AdamOptimizer(learning_rate=1e-
4).minimize(cost)

correct_prediction = tf.equal(y_pred_cls, y_true_cls)

accuracy = tf.reduce_mean(tf.cast(correct_prediction,
tf.float32))
```

```
session = tf.Session()

def init_variables():

    session.run(tf.global_variables_initializer())
```

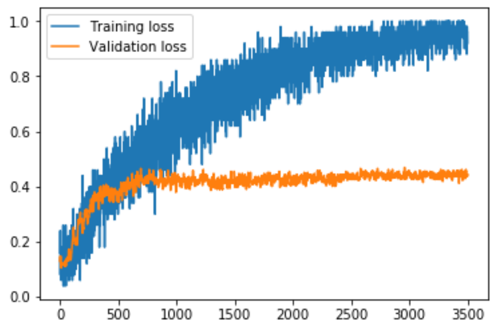
The function below creates a batch from a dataset. We use a batch to train our model.

```
batch_size = 50

#function next_batch
def next_batch(num, data, labels):
    '''
    Return a total of `num` random samples and labels.
    '''
    idx = np.arange(0 , len(data))
    np.random.shuffle(idx)
    idx = idx[:num]
    data_shuffle = [data[i] for i in idx]
    labels_shuffle = [labels[i] for i in idx]


    return np.asarray(data_shuffle), np.asarray(labels_shuffle)
```

```
def optimize(num_iterations, X):
    global total_iterations

    start_time = time.time()

    #array to plot
    losses = {'train':[], 'validation':[]}

    for i in range(num_iterations):
            total_iterations += 1
            # Get a batch of training examples.
            # x_batch now holds a batch of images and
            # y_true_batch are the true labels for those images.
            x_batch, y_true_batch = next_batch(batch_size, X_train, y_train)


            # Put the batch into a dict with the proper names
            # for placeholder variables in the TensorFlow graph.
            feed_dict_train = {x: x_batch,
                               y_true: y_true_batch,
                               keep_prob_conv : 0.3,
                               keep_prob_fc : 0.4}
            feed_dict_validation = {x: X_validation,
                               y_true: y_validation,
                               keep_prob_conv : 1,
                               keep_prob_fc : 1}

            # Run the optimizer using this batch of training data.
            # TensorFlow assigns the variables in feed_dict_train
            # to the placeholder variables and then runs the optimizer.
            session.run(optimizer, feed_dict=feed_dict_train)

            acc_train = session.run(accuracy, feed_dict=feed_dict_train)
            acc_validation = session.run(accuracy, feed_dict=feed_dict_valid
            losses['train'].append(acc_train)
            losses['validation'].append(acc_validation)

            # Print status every X iterations.
            if (total_iterations % X == 0) or (i ==(num_iterations -1)):
            # Calculate the accuracy on the training-set.

    # Difference between start and end-times.
    time_dif = end_time - start_time


    # Print the time-usage.
    print("Time usage: " + str(timedelta(seconds=int(round(time_dif)))))
```

```
init_variables()

total_iterations = 0

optimize(num_iterations=3500, X=250)
```

```
Iteration:    250, Training Accuracy:  28.0%, Validation Accuracy:  29.6%
Iteration:    500, Training Accuracy:  50.0%, Validation Accuracy:  36.5%
Iteration:    750, Training Accuracy:  48.0%, Validation Accuracy:  43.7%
Iteration:   1000, Training Accuracy:  50.0%, Validation Accuracy:  40.4%
Iteration:   1250, Training Accuracy:  70.0%, Validation Accuracy:  43.7%
Iteration:   1500, Training Accuracy:  80.0%, Validation Accuracy:  41.5%
Iteration:   1750, Training Accuracy:  80.0%, Validation Accuracy:  44.0%
Iteration:   2000, Training Accuracy:  82.0%, Validation Accuracy:  43.0%
Iteration:   2250, Training Accuracy:  92.0%, Validation Accuracy:  42.2%
Iteration:   2500, Training Accuracy:  94.0%, Validation Accuracy:  43.3%
Iteration:   2750, Training Accuracy:  88.0%, Validation Accuracy:  44.4%
Iteration:   3000, Training Accuracy: 100.0%, Validation Accuracy:  44.0%
Iteration:   3250, Training Accuracy:  92.0%, Validation Accuracy:  44.0%
Iteration:   3500, Training Accuracy:  92.0%, Validation Accuracy:  44.4%
Time usage: 2:08:14
```



As you can see, the model tends to overfit and is not very good.

8 — CNN with TensorFlow — Results

The results are not so good as the accuracy is only 44%. Using a pre-trained model with Keras will give you a better result but with this model, you will know how to build from scratch your own CNN with TensorFlow.

By having more photos of dogs, we can increase the accuracy. in addition, we can create new images in our training dataset by rotating the images. it's what we call image augmentation. It will help the model to detect a pattern which can have different 'position' in the space.

Below are some functions to show some images from the new test data

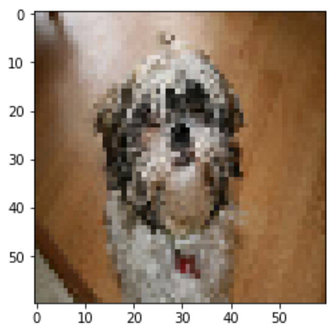
```
    # Create figure with 3x3 sub-plots.
    fig, axes = plt.subplots(4, 3)
    fig.subplots_adjust(hspace=0.3, wspace=0.3)


    for i, ax in enumerate(axes.flat):
        # Plot image.
        ax.imshow(images[i].reshape(img_shape), cmap='binary')


        # Show true and predicted classes.
        if cls_pred is None:
            xlabel = "True: {0}".format(cls_true[i])
        else:
            xlabel = "True: {0}, Pred: {1}".format(cls_true[i], cls_pred[i])


        # Show the classes as the label on the x-axis.
        ax.set_xlabel(xlabel)

        # Remove ticks from the plot.
        ax.set_xticks([])
        ax.set_yticks([])

    # Ensure the plot is shown correctly with multiple plots
```

```
def plot_confusion_matrix(data_pred_cls,data_predicted_cls):
    # This is called from print_test_accuracy() below.


    # cls_pred is an array of the predicted class-number for
    # all images in the test-set.

    # Get the confusion matrix using sklearn.
    cm = confusion_matrix(y_true=data_pred_cls,
                          y_pred=data_predicted_cls)


    # Print the confusion matrix as text.
    print(cm)


    # Plot the confusion matrix as an image.
    plt.matshow(cm)


    # Make various adjustments to the plot.
    plt.colorbar()
    tick_marks = np.arange(num_classes)
    plt.xticks(tick_marks, range(num_classes))
    plt.yticks(tick_marks, range(num_classes))
    plt.xlabel('Predicted')
    plt.ylabel('True')


    # Ensure the plot is shown correctly with multiple plots
```

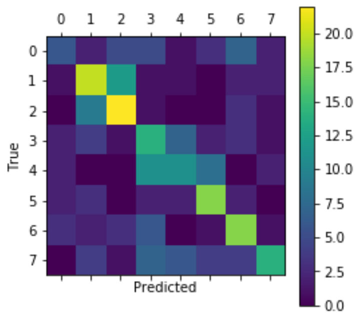
Let’s look at some results!

```
feed_dict_validation = {x: X_validation, y_true:
y_validation, keep_prob_conv : 1, keep_prob_fc : 1}

df_validation_Predicted_cls = session.run(y_pred_cls,
feed_dict=feed_dict_validation)

plot_images(images=X_validation[50:62],
cls_true=df_validation_toPred_cls[50:62],
cls_pred=df_validation_Predicted_cls [50:62])
```

```
i = 63

print(("True : {0} /
{1}").format(df_validation_toPred_cls[i],
labels_name[df_validation_toPred_cls[i]]))

print(("Pred : {0} /
{1}").format(df_validation_Predicted_cls[i],
labels_name[df_validation_Predicted_cls[i]]))

lum = X_validation[i,:,:,:]

plt.show()
```



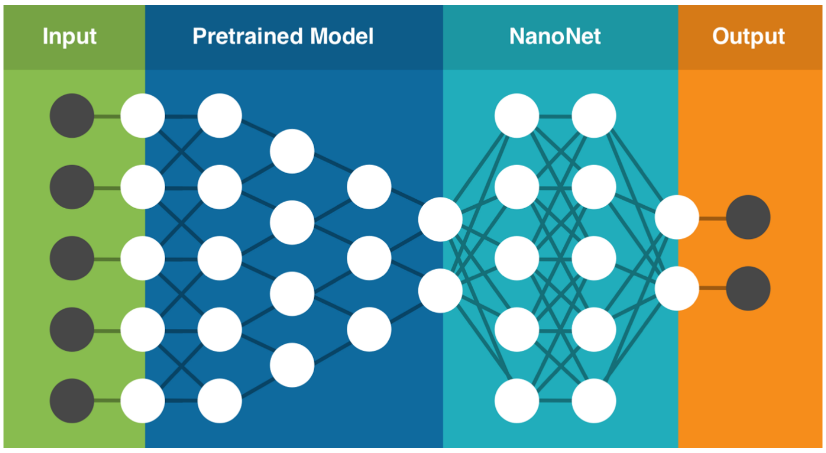
```
plot_confusion_matrix(df_validation_toPred_cls,df_validation_
Predicted_cls)
```



As you can see, the model has difficulties to differentiate Breed 1: **bernese_mountain_dog** and Breed 2: **entlebucher**. These 2 breeds look very similar to each other(same color and shape). So, it looks normal that our model has made some mistakes between these two breeds.

Nanonets makes Transfer Learning easier

Having experienced the accuracy problem with transfer learning, I set out to solve it by building an easy to use cloud-based Deep Learning



Because the NanoNets models are heavily pre-trained, I used a much smaller training dataset of only ~100 images per class. From this model, I got 83.3% test accuracy. This is 7% more than the VGG19 model in spite of using 1/60th of the data! The reason that NanoNets model performs better is: large amount of pre-training, optimal hyper-parameter selection, and data augmentation.

The great thing about NanoNets is that anyone can upload data and build their own models. You can build models in 2 ways:

1. Using a GUI: https://app.nanonets.com

2. Using NanoNets API:

https://github.com/NanoNets/image-classification-sample-python

Below, we will give you a step-by-step guide to training your own model using the Nanonets API, in 9 simple steps.

Step 1: Clone the Repo

```
git clone https://github.com/NanoNets/image-classification-
sample-python.git
cd image-classification-sample-python
sudo pip install requests
```

Step 2: Get your free API Key

Get your free API Key from http://app.nanonets.com/#/keys

Step 3: Set the API key as an Environment Variable

```
export NANONETS_API_KEY=YOUR_API_KEY_GOES_HERE
```

Step 4: Create a New Model

```
export NANONETS_MODEL_ID=YOUR_MODEL_ID
```

Step 6: Upload the Training Data

Collect the images of the objects you want to detect. Once you have dataset ready in folder `images` (image files), start uploading the dataset.

```
python ./code/upload-training.py
```

Step 7: Train Model

Once the Images have been uploaded, begin training the Model

```
python ./code/train-model.py
```

Step 8: Get Model State

The model takes ~30 minutes to train. You will get an email once the model is trained. In the meanwhile, you check the state of the model

```
watch -n 100 python ./code/model-state.py
```

Step 9: Make Prediction

Once the model is trained. You can make predictions using the model

```
python ./code/prediction.py PATH_TO_YOUR_IMAGE.jpg
```

Thanks to Rushabh Nagda and Shiva Manne