Humpback Whale Identification

By:
Jasmine Cao
Melissa Cheung
Adan Constanzo
Mher Oganesyan
Smitkumar Kaushikkumar Patel

Table of Contents

Introduction

Purpose

Whales are being treated by tougher environments as human intervene from the hunt for whales and oceans becoming warmer. A solution for the whales were to migrate to different locations in the sea. In order to aid the conservation efforts and provide a general surveillance system to the monitor the ocean activity this challenge from kaggel arised. For this kaggle competition, we are gather images from Happywahle's database to classify specific species of whales for surveillance purposes. For the past 40 years, most of analyzing and identify whales have been done manually by individual scientist; leaving a majority of data unused and underutilized. This competition enables the use of helping a scientific community, make use of their data, and provide better survancle for the ocean.

Dataset

The data set consist of over 25,361 images gathered from research institutions and public contributors. Out of the 25,361 images there are over 5,005 unique whale species to be classified. And from the 25,361 images there are around 9,664 whales unidentified. All images come in different dimensions and might be RGB or grayscale. There is a train.csv that includes two columns, image name and whale identification.

Modification to the challenge

Since we are giving multiple images to classify, Dr. Mohammad Pourhomayoun gave us special permission to reduce the complexity of the project. Instead of classifying over 5,005 whales we will be only classifying the top 10 whales in our dataset.

Team Members Responsibilities

Smit and Jasmine were responsible for analyzing the data provided to us. They researched into how we can preprocess the data so we can use it in our project since the images varied in size and were also large. Adan was responsible for researching into different algorithms that we can use for the project, such as the ones we used in class or preprocessed models.

Melissa and Mher worked together to put all the information gathered by everyone else to code and test the models.

Pre-processing

Data Cleaning

A huge proportion of the dataset consisted of an unknown, unlabeled image called new_whale. In order to get a proper results when training/testing we removed 9664 whales consisting of new_whale which happens to be over 38% of our dataset. Then we collected the top ten most occurence of our data set which consisted of around 570 samples.

Image Size Reduction

All images in our dataset contained different dimensions and color schemes. Our convolutional neural network required that all images consisted a specific width and height for the model we constructed for our project. As a result, we resized every images to a specific 100 by 100 pixel resolution and scaled the aspect of the image to prevent loss of information.

Augmenting Images to our dataset

Since we are working with a dataset of around 570 samples a common technique to increase the accuracy of our model was to generate more images by rotating, shifting, horizontally flipping and zooming in on the image. So when it came to training, we increased the number of images to feed to our CNN from 570 images to 3306 images.

Preparing our Features

Since we are using images our features consisted the use of each pixel and rgb value. We first loaded the images with the help of keras.preprocessing.image, condensed the image to a multidimensional array of 100 x 100 x 3 and finally appended it to a training set / testing set. Each image was also divided by 255 (rgb) to help with processing.

Preparing our Labels

We are classifying over 10 unique whales, we used OneHotEncoding to determine what type of whale the model is predicting.

Methodology/Code

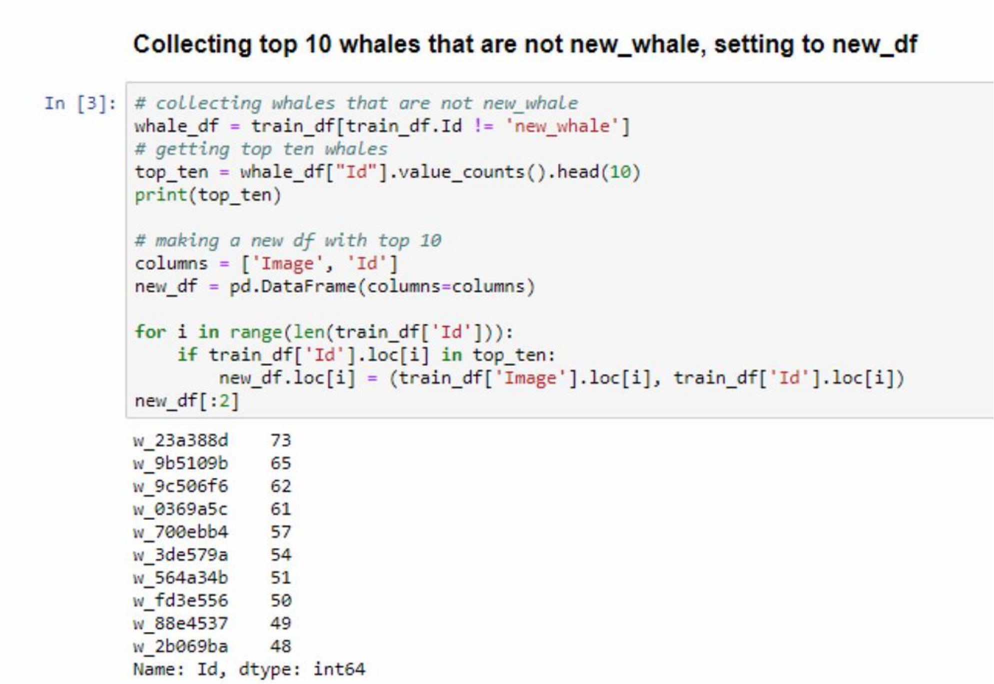
Support Vector Machines (SVM)

Introduction

Support Vector Machines, or SVM, is a popular machine learning algorithm created by Vladimir N. Vapnik. This algorithm uses learning algorithms to analyze data used for classification or regression analysis. To put SVM in simple terms, SVM tries it’s best to find a hyperplane (a line or plane), or a set of hyperplanes to separate and classify data. The next step is to categorize our data set with maximum-margin line, this line emphasises the clear separation between two or more classifications. Margins play an important role to the algorithm and are know as a hyperparameter to the algorithm.

Preparing our Data

Since our project is modified we first collected the top ten most used whales that did not include not new_whale and created a small dataframe.

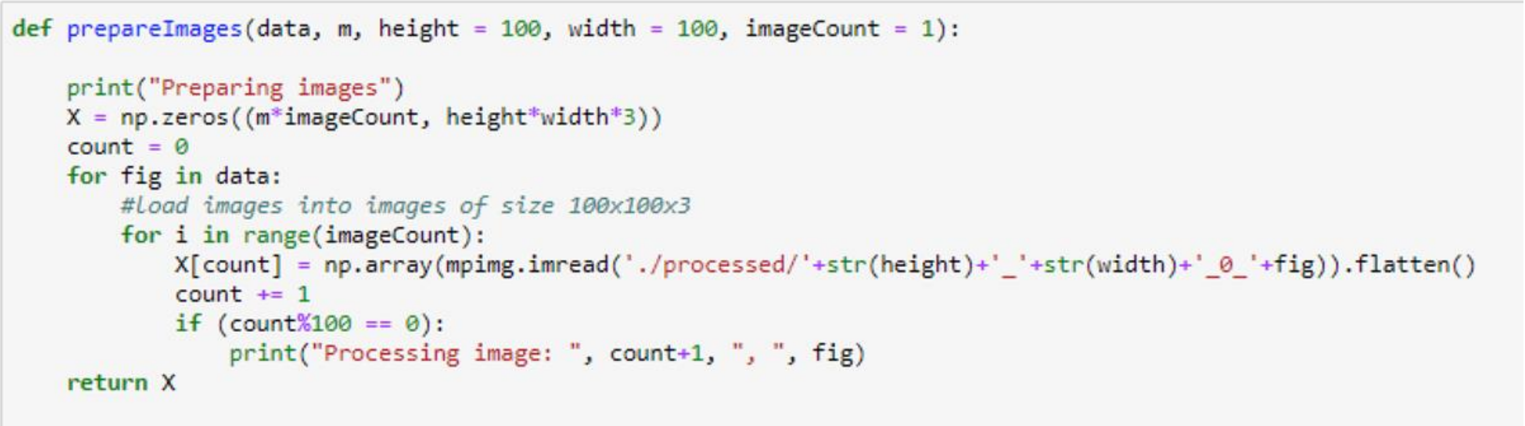


Collecting top 10 whales that are not new_whale, setting to new_df

```
In [3]: # collecting whales that are not new_whale
        whale_df = train_df[train_df.Id != 'new_whale']
        # getting top ten whales
        top_ten = whale_df["Id"].value_counts().head(10)
        print(top_ten)

        # making a new df with top 10
        columns = ['Image', 'Id']
        new_df = pd.DataFrame(columns=columns)

        for i in range(len(train_df['Id'])):
            if train_df['Id'].loc[i] in top_ten:
                new_df.loc[i] = (train_df['Image'].loc[i], train_df['Id'].loc[i])
        new_df[:2]
```

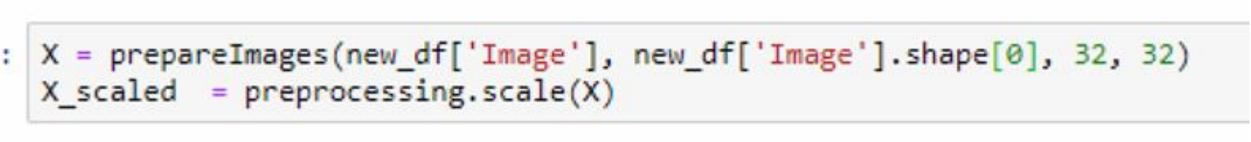
```
w_23a388d    73
w_9b5109b    65
w_9c506f6    62
w_0369a5c    61
w_700ebb4    57
w_3de579a    54
w_564a34b    51
w_fd3e556    50
w_88e4537    49
w_2b069ba    48
Name: Id, dtype: int64
```

After making our dataframe the next important step was to make our features columns by selecting our features to be the images pixels. This function allows us to easily make a dataset by opening an image as an np.array and finally flattening the image and appending it to an array named X.



```
def prepareImages(data, m, height = 100, width = 100, imageCount = 1):

    print("Preparing images")
    X = np.zeros((m*imageCount, height*width*3))
    count = 0
    for fig in data:
        #load images into images of size 100x100x3
        for i in range(imageCount):
            X[count] = np.array(mpimg.imread('./processed/'+str(height)+'_'+str(width)+'_0_'+fig)).flatten()
            count += 1
            if (count%100 == 0):
                print("Processing image: ", count+1, ", ", fig)
    return X
```

After creating our feature column we can then scale the features to reduce computation for the SVM model.

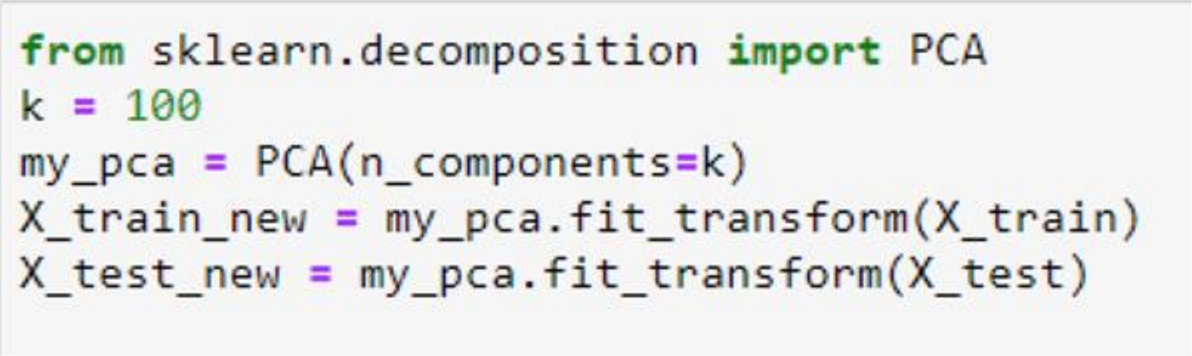


```
: X = prepareImages(new_df['Image'], new_df['Image'].shape[0], 32, 32)
  X_scaled  = preprocessing.scale(X)
```

We then split our data set into training and testing.

Reducing Complexity of Data

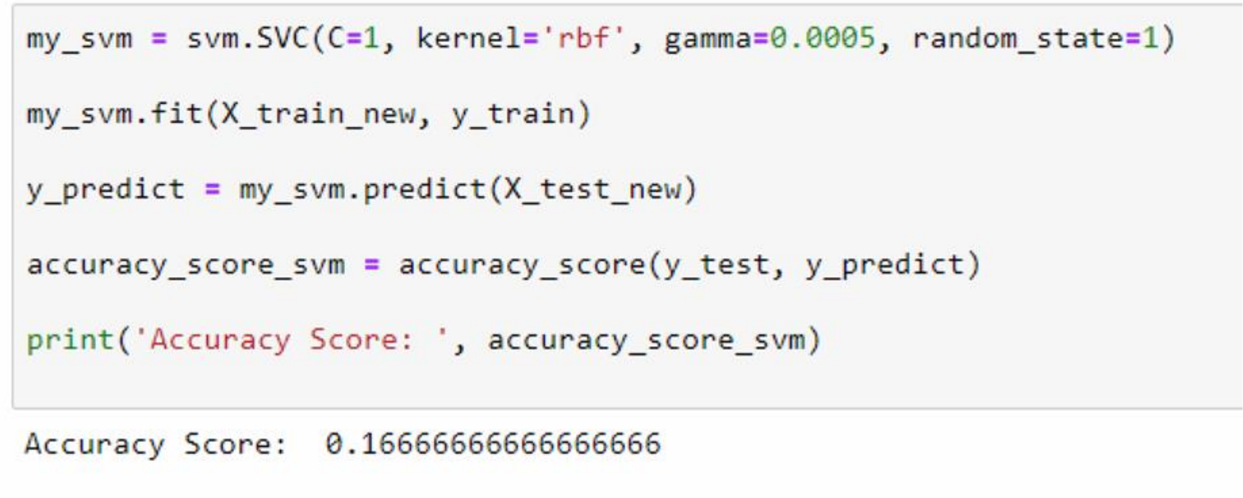
After splitting the data, we must use PCA to reduce the complexity of the image feature so our SVM model can perform better !



```
from sklearn.decomposition import PCA
k = 100
my_pca = PCA(n_components=k)
X_train_new = my_pca.fit_transform(X_train)
X_test_new = my_pca.fit_transform(X_test)
```

Training the Model

Now let’s train the model!

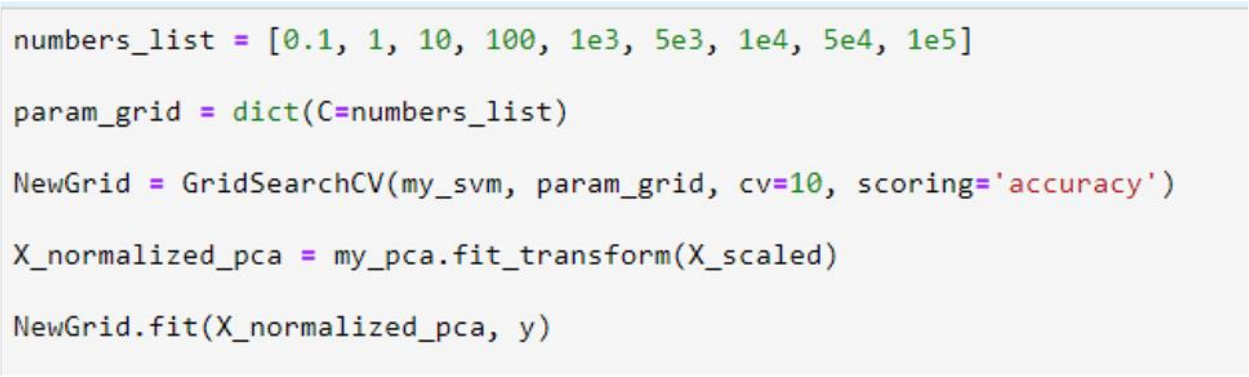


```
my_svm = svm.SVC(C=1, kernel='rbf', gamma=0.0005, random_state=1)

my_svm.fit(X_train_new, y_train)

y_predict = my_svm.predict(X_test_new)

accuracy_score_svm = accuracy_score(y_test, y_predict)

print('Accuracy Score: ', accuracy_score_svm)
```

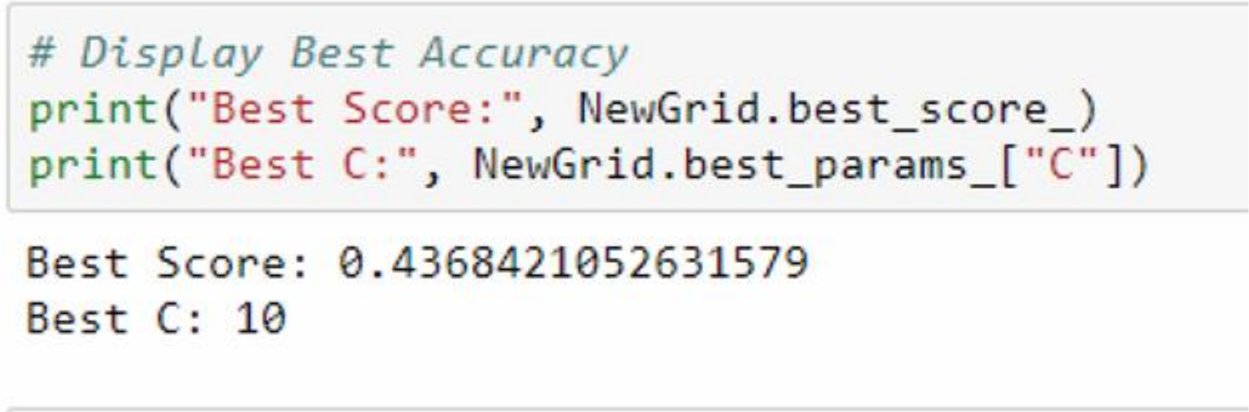
```
Accuracy Score:  0.16666666666666666
```

Improving Accuracy

As you can see, the accuracy isn’t the highest, this is because we used a value of the hyperparameter C = 1. This hyperparameter might not be the best fit for our SVM model and the dataset we have. We can use a library know as GridSearchCV to brute-force a range of possible C values to determine the best accuracy for our model.



```
numbers_list = [0.1, 1, 10, 100, 1e3, 5e3, 1e4, 5e4, 1e5]

param_grid = dict(C=numbers_list)

NewGrid = GridSearchCV(my_svm, param_grid, cv=10, scoring='accuracy')

X_normalized_pca = my_pca.fit_transform(X_scaled)

NewGrid.fit(X_normalized_pca, y)
```



```
# Display Best Accuracy
print("Best Score:", NewGrid.best_score_)
print("Best C:", NewGrid.best_params_["C"])
```

```
Best Score: 0.4368421052631579
Best C: 10
```

This drastically increases our accuracy from 0.1667 to 0.4368.

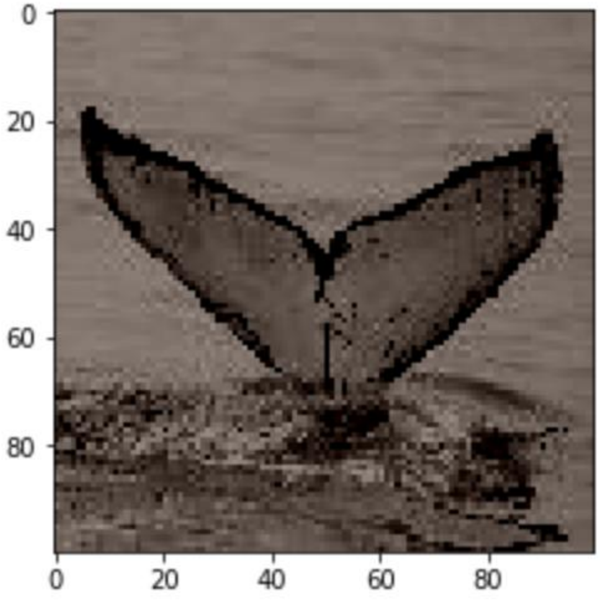
Convolutional Neural Network (CNN)

Introduction

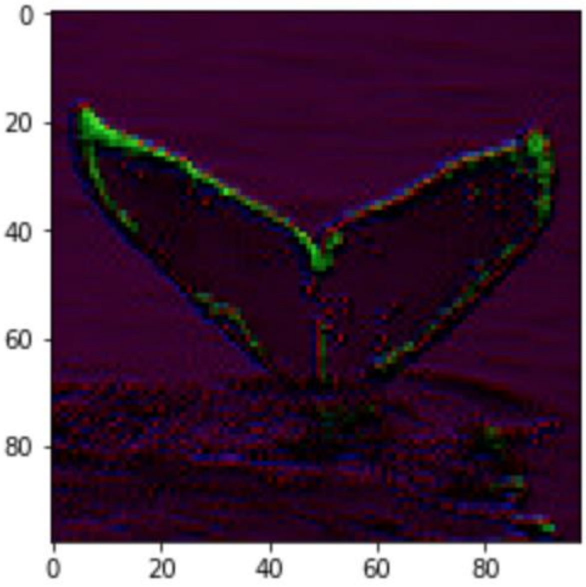
A convolutional neural network (CNN) is a type of deep neural network inspired by the human brain visual cortex. In comparison to SVM, a CNN extracts meaningful patterns from images such as edges or basic shapes to be more efficient than using raw pixels. A CNN consist a series of layers including convolutional layers (for feature extractions), ReLu layers (for nonlinearity), and pooling layers (for further dimensionality reduction) and finally fully connected layers (for final classification).

What makes a CNN so special is that it applies filters to each image to reduce the complexity of the image and collect features such as edge detection. The model then figures out the best filters to apply to each image in order to increase the accuracy for prediction.

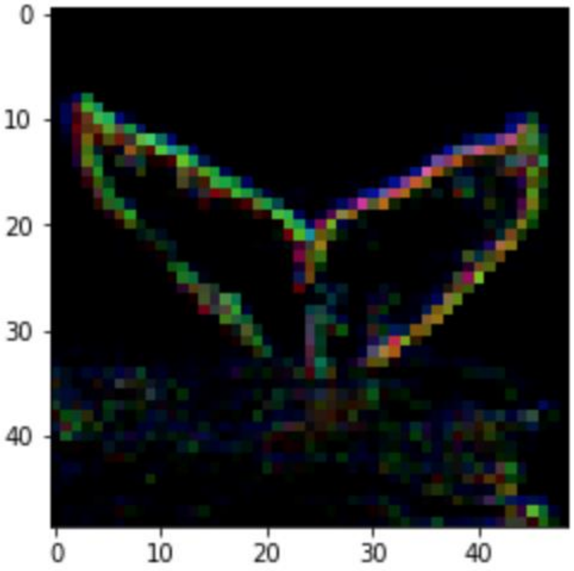
Here is a picture of a 100 x 100



Here is an same image being applied l 3, (3 x 3) filter to an image



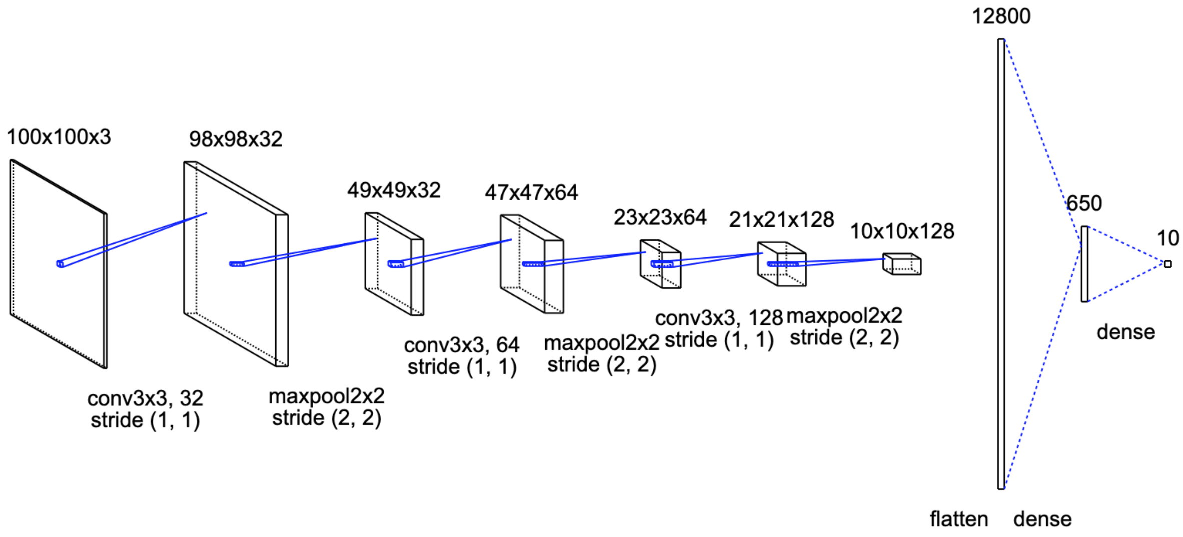
Finally, here is the same image being applied 3 , (3x3) filter and reducing dimensionality with dropout (randomly disconnecting weights)



CNN Architecture

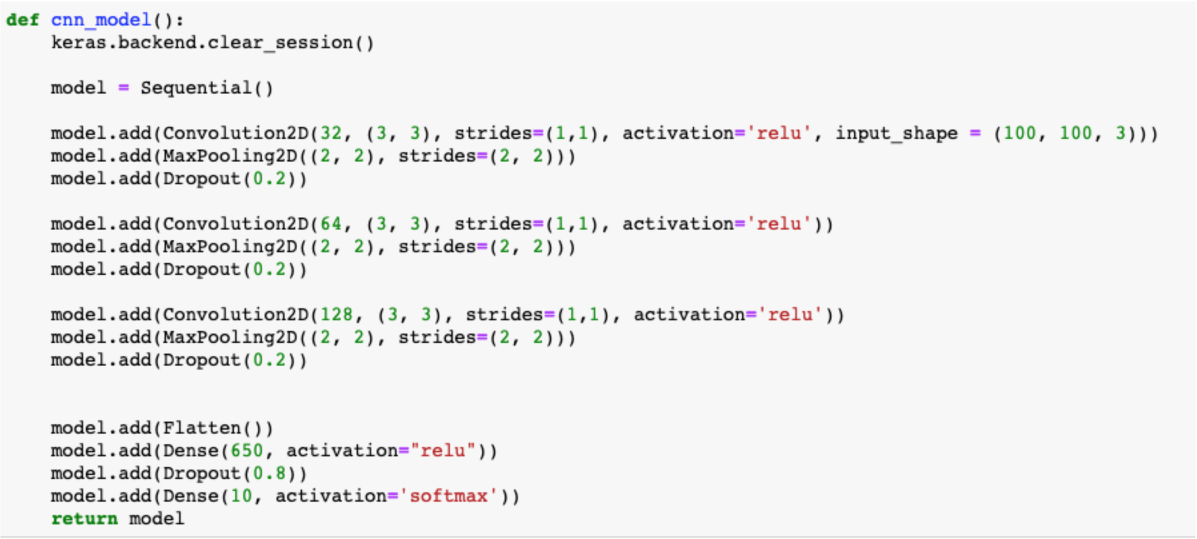
The model we constructed for our CNN was inspired from the VGG_16 model. We increased the numbers by powers of 2 as the layer got deeper and applied both Max2DPooling and dropout . Since CNN is a deep learning network the nature of this model means computing multiple layers with weights. For our basic model we are computing over 8.4 million parameters and, as a result, had to stop after applying filters when we reached to 128 filters.

Here is a diagram of our CNN.



For our model we applied an activation layer ReLu for each convolution layer.

Here is our code for the CNN model.



```
def cnn_model():
    keras.backend.clear_session()

    model = Sequential()

    model.add(Convolution2D(32, (3, 3), strides=(1,1), activation='relu', input_shape = (100, 100, 3)))
    model.add(MaxPooling2D((2, 2), strides=(2, 2)))
    model.add(Dropout(0.2))

    model.add(Convolution2D(64, (3, 3), strides=(1,1), activation='relu'))
    model.add(MaxPooling2D((2, 2), strides=(2, 2)))
    model.add(Dropout(0.2))

    model.add(Convolution2D(128, (3, 3), strides=(1,1), activation='relu'))
    model.add(MaxPooling2D((2, 2), strides=(2, 2)))
    model.add(Dropout(0.2))


    model.add(Flatten())
    model.add(Dense(650, activation="relu"))
    model.add(Dropout(0.8))
    model.add(Dense(10, activation='softmax'))
    return model
```

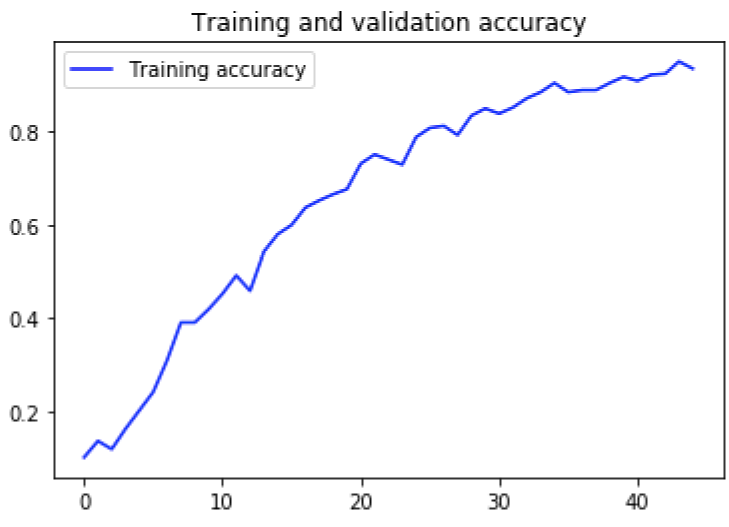
Preparing our Data

Since our CNN requires that all inputs be uniform we had to resize all images to a standard of 100 x 100. Since each image can either be grayscale or rgb we decided to not flatten the images and instead keep the shape of (100, 100, 3) and store them into a list.

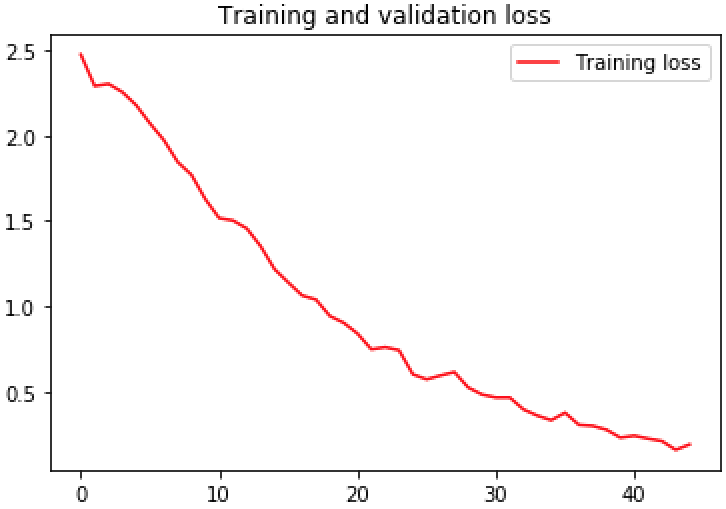
Training

During training we used hyperparameters of batch_size to 64 due to limitation of memory and epochs to 45 . This gave us an accuracy of 0.71.

The image below represents the accuracy on validation set as epoch increases .

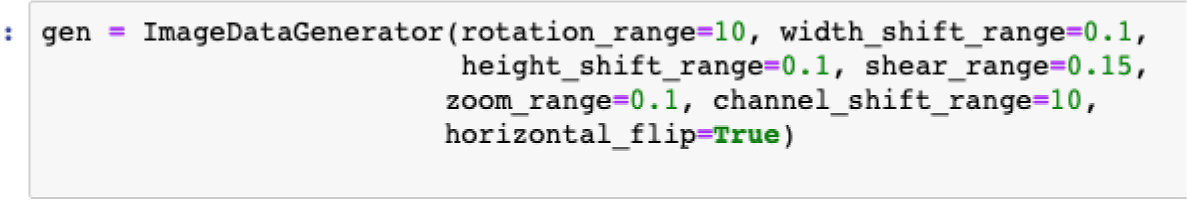


The image below represents the loss on validation set as epoch increases.



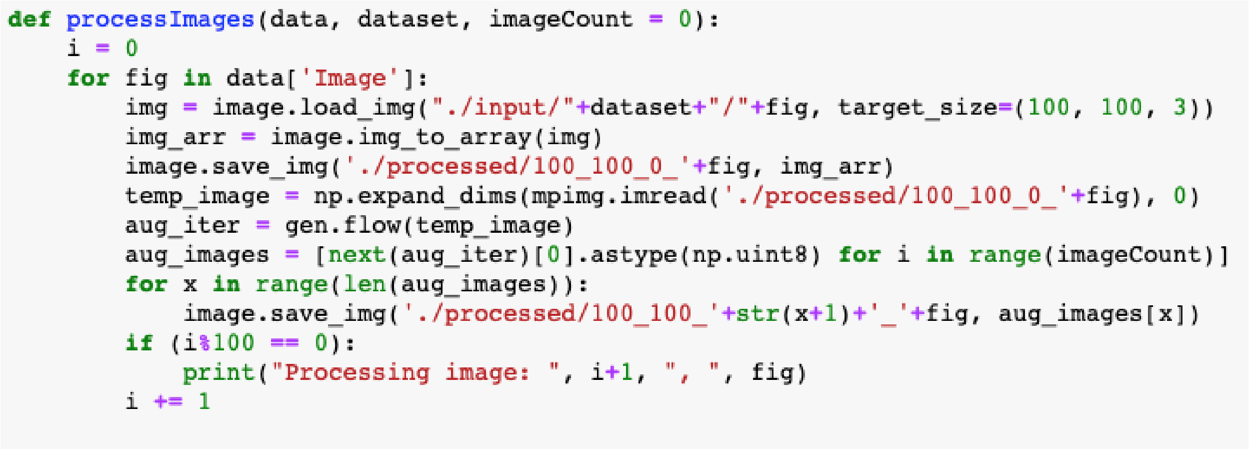
Improving Training by Augmenting Images to Increase Data Set

CNN thrives with big datasets. Since we are only utilizing around 570 images the accuracy of the model suffers. One viable solution is to augment more images that can be either rotated, flipped, or zoomed in. To do this we used a library from keras called, ImageDataGenerator. We first include instance of the object with our specific requirements.



```
: gen = ImageDataGenerator(rotation_range=10, width_shift_range=0.1,
                              height_shift_range=0.1, shear_range=0.15,
                             zoom_range=0.1, channel_shift_range=10,
                             horizontal_flip=True)
```

Next we simply create and save images that our ImageDataGenerator created for us

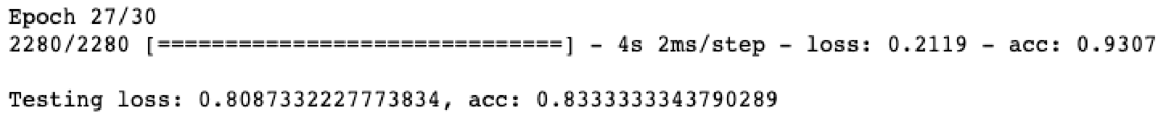


```
def processImages(data, dataset, imageCount = 0):
    i = 0
    for fig in data['Image']:
        img = image.load_img("./input/"+dataset+"/"+fig, target_size=(100, 100, 3))
        img_arr = image.img_to_array(img)
        image.save_img('./processed/100_100_0_'+fig, img_arr)
        temp_image = np.expand_dims(mpimg.imread('./processed/100_100_0_'+fig), 0)
        aug_iter = gen.flow(temp_image)
        aug_images = [next(aug_iter)[0].astype(np.uint8) for i in range(imageCount)]
        for x in range(len(aug_images)):
            image.save_img('./processed/100_100_'+str(x+1)+'_'+fig, aug_images[x])
        if (i%100 == 0):
            print("Processing image: ", i+1, ", ", fig)
        i += 1
```

Finally, we can then add our new images to our X_training, leaving the X_test untouched from these new images.
This increases our training set from 456 to 2280.

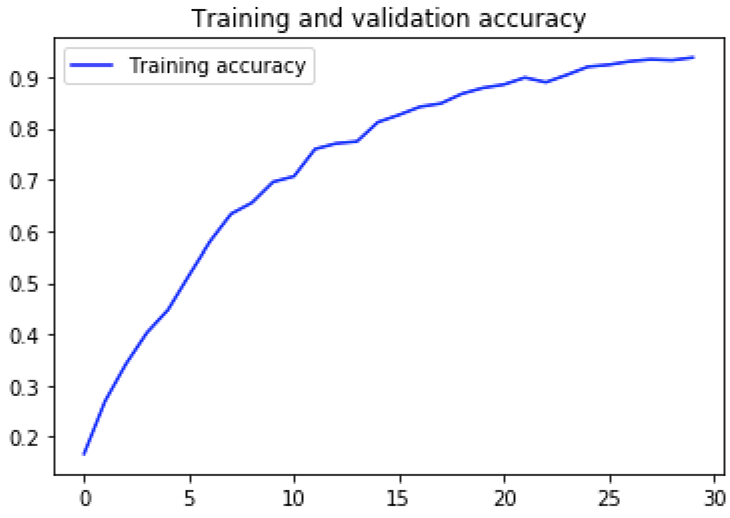
Final Training with Augmented Images

After fitting the data one more time but of course restarting the model we achieved an accuracy of over 0.83!

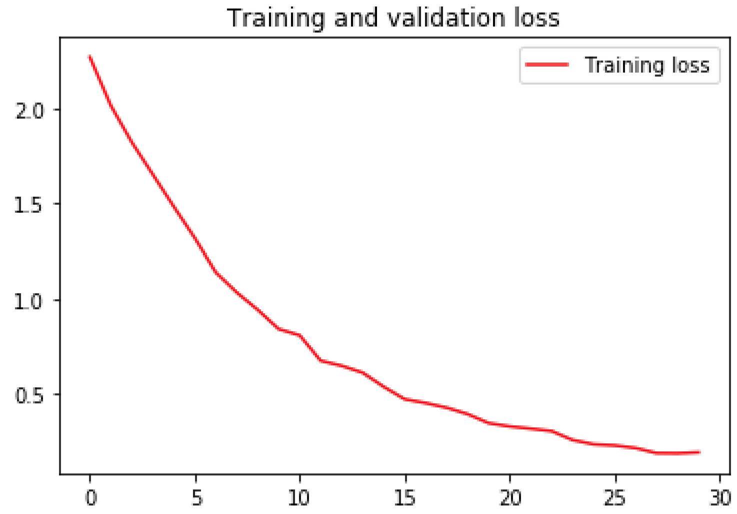


```
Epoch 27/30
2280/2280 [==============================] - 4s 2ms/step - loss: 0.2119 - acc: 0.9307

Testing loss: 0.8087332227773834, acc: 0.8333333343790289
```

The image below represents the accuracy on validation set as epoch increases .



The image below represents the loss on validation set as epoch increases.



Final Results

Model	**Accuracy**
SVM	0.43
CNN (non augmented images)	0.71
CNN (augmented images)	0.83