Machine Learning Engineer Nanodegree
Capstone Project
Sraddhanjali Acharya
July 18th, 2019

Aerial Cactus Identification (Determine whether an image contains a columnar cactus)

DEFINITION

Project Overview

VIGIA project [1] is a project based in Mexico developed to assist government led efforts to preserve natural areas. The traditional surveillance mechanisms have been insufficient to log and monitor the impacts of climate change and human activities on the flora and fauna. The goal of VIGIA project is to build an automatic surveillance of such protected areas.

To build such a system various tools and technologies are used such as unmanned aircrafts (drones) equipped with a camera or cameras to take aerial imagery of the protected areas. The imagery can then be fed into a computer vision recognition system to distinguish flora/fauna, log the number of a certain protected species, etc.

Specifically in this project/competition [2], the task of computer vision recognition system is to detect a certain species of cactus. To lead the way in protecting the flora (specifically columnar cactus) in protected natural areas, aerial imagery taken from unmanned aircraft can be fed into a system which utilizes state-of-the-art computer vision and machine learning methods to assist in the recognition task.

Problem Statement

The problem statement of this competition is to predict whether an image captured from the unmanned aircraft is a columnar cactus species (*Neobuxbaumia tetetz*o) or not. This is a two class problem where the predictions will be whether an image contains the columnar cactus or not.

This is a binary classification problem where if an image contains a columnar cactus it is predicted to have a probability closer to 1 else closer to 0. It can also be set as 1 or 0 based on a threshold such as 0.5, i.e if probabilities are greater than 0.5 then set to 1 else 0.

The strategy to solve this problem can be outlined in the following bullet points.

1. Read the train and test data images
2. Convert the JPEG images to RGB pixel information and convert to floating tensors for use in the neural network (specifically CNN)
3. Re-scale the pixel information from 0-255 to a [0, 1] interval.
4. Apply a simple CNN-based model on the data to get the benchmark model to compare against future improved CNN models.
5. Improved models will optimize on this benchmark model using various measures such as image data augmentation, using batch normalization, regularization techniques, etc.
6. Further well-established image classification models such as VGG16/19, InceptionV3, MobileNet, etc. whichever appropriate will be used to get an even better model.
7. The result to be submitted will be in a *.csv* file and of the format as shown below:
    a. **Id -** *name of the image file in the test folder*

   b. **Has_cactus** - *the predicted probability of the image being a cactus*
8. Predict the probabilities in the validation set images and check the validation accuracy.
9. Predict probability of cactus for Kaggle submissions.

Metrics

The final evaluation metric used for Kaggle submission (for test data) is Area Under Receiver Operating Characteristics (*ROC*) curve. ROC curve is a plot of true positive rate (*TPR*) against false positive rate (*FPR*). TPR is the ratio of true positives over all positives. FPR is the ratio of false positives over all negatives.

TPR = TP/P ; FPR = FP/N.

All positives is the sum of true positive and false negative. All negatives is the sum of true negative and false positive.

For this problem, area under ROC curve is used because of class imbalance. Since the number of cactus data is more than no cactus data, using classification error would not give a bias-free analysis of how well the classifier performs in separating the two classes.

For example, if we have 1000 data in class A and 200 in class B. If 20 data items from class A are wrongly classified and 50 data items from class B are wrongly classified, this would be 2% error for class A and 25% error in class B. This doesn't clearly give us an idea of the classifier performance. ROC gives performance measure of classifier at various threshold values and gives an idea about the measure of separability of the classes. Higher the value of AUC better the classifier in separating True Positive and False Positives. Unlike accuracy which works well with balanced dataset, ROC gives more hints on how model would perform if that is not the case.

Another evaluation metric used is accuracy which is a ratio of the sum of true positives and sum of true negatives over the total dataset.

Accuracy = (TP + TN)/total dataset size

ANALYSIS

Data Exploration

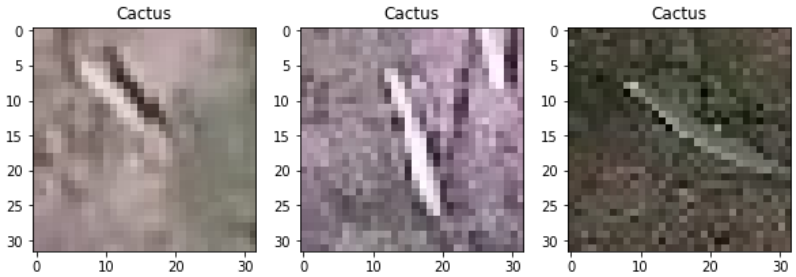
The dataset can be downloaded from the link in [3]. The aerial cactus identification competition [2] contains dataset with large number of 32x32 thumbnail images of aerial photos of cacti. The image dataset has been resized for uniformity.

The dataset consists of train and test images in **train** and **test** folders respectively. **Train.csv** contains two columns **id** and **has_cactus**. The **id** column has the image name which can be fetched from the **train** folder. The **has_cactus** column is training image label, 1 if it has a cactus else 0. There are 17500 training images and 4000 testing images in the train and test folder respectively.

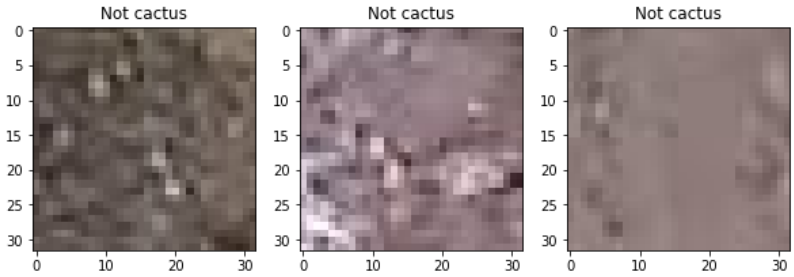
The dataset in file **train.csv** looks like the following.

	id	has_cactus
0	0004be2cfeaba1c0361d39e2b000257b.jpg	1
1	000c8a36845c0208e833c79c1bffedd1.jpg	1
2	000d1e9a533f62e55c289303b072733d.jpg	1
3	0011485b40695e9138e92d0b3fb55128.jpg	1
4	0014d7a11e90b62848904c1418fc8cf2.jpg	1

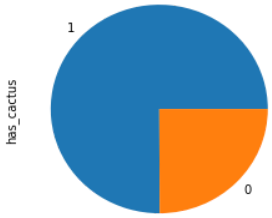
Some sample images of images having cactus are shown below:



Some sample images of images having no cactus are shown below:

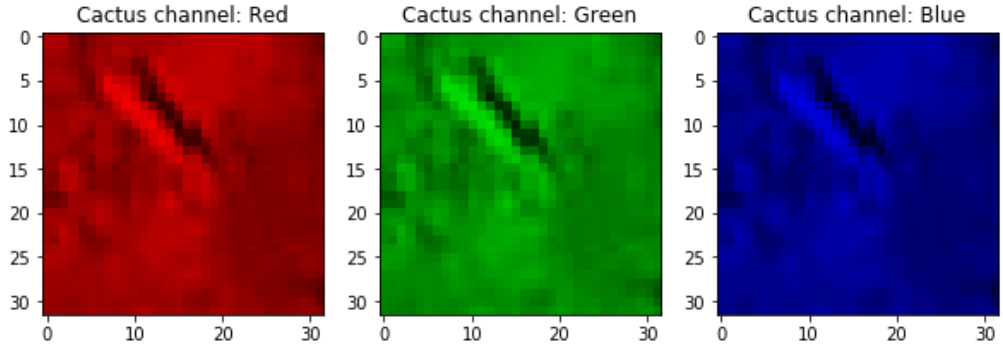


We can see in the pie-chart below that, there are a larger number of cactus images (13,136) than there are no cactus (4363) images in the dataset. This class imbalance issue will be addressed later in the report.

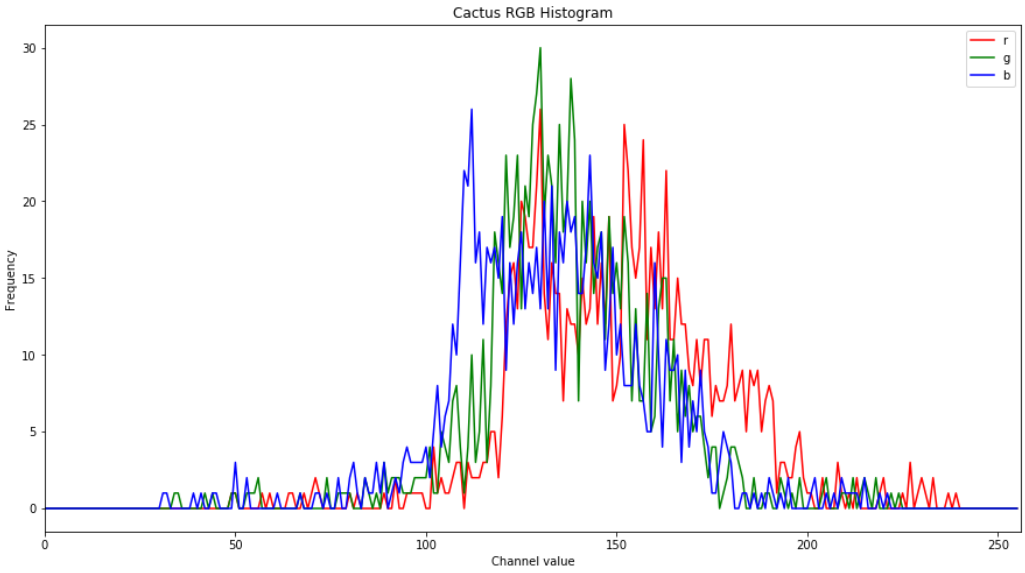


Exploratory Visualization

The image data has three dimensions for their RGB values and can be viewed for a cactus image in the following plot. All these three dimensions will be used to extract features in the CNN model which will be discussed later.



To get a better idea about if these channels represent different information, wecan view the histogram plot of individual channels of the cactus image in the plot below:



The red, blue and green channels vary in their channel value distribution. **For example,** we can observe that at channel value around 200, the red channel frequency is higher than it is for the other channels.

Algorithms and Techniques

CNN (convolutional neural network) classifier is a state-of-art classifier for image classification. It uses convolution kernel/filter to extract features from each dimension of images which can be represented as a matrix of RGB values. The convolution operation superimposes the filter onto the image matrix to obtain a feature map out of the image matrix. For example, the convolution of a 32x 32 input image matrix with a 3x3 filter is done by superpositioning the filter on the image, and then adding the product of the values from filter and the values from the image matrix. This is done over the image matrix to give an output

feature map. Stride is a parameter which can be used to determine how the filter should shift. Some description of layers and key terms are explained below:

Max pooling layer extracts features from input feature map in a certain window size, say 2x2 and outputs maximum value in this region.

Relu activation layer adds nonlinearity like tanh and sigmoid but relu helps solve the vanishing gradient problem which occurs especially when training a large network. Vanishing gradient problem is where the gradient decreases exponentially in the backpropagation algorithm while updating the weights resulting in slow to no update to the weights.

Dropout layers are used to reduce overfitting which is a technique where given a probability value such a 0.5, 50% of the neurons are randomly chosen and are not used in the training phase. In the next run of the training, another set of 50% randomly chosen nodes will be skipped in the training and so on.

Above building blocks can be used to build a simple CNN model to train the images in the dataset.

To further improve the model, **data augmentation** can be done. Let's take a scenario where a classifier is being trained on perfectly centered images, despite good training accuracy, on not-so-perfectly centered images in the test dataset the model will underperform. To cater to diverse images with shift in their position from center, horizontal flip of the images, etc. data augmentation techniques can be employed in the training data.

Transfer learning takes a pre-trained model which has its own network architecture and weights, which can be frozen and only the lower layers can be trained for the problem dataset at hand. The high level features weights from the top layers are then fine-tuned to the problem at hand by connecting this top layers to a self-designed fully connected network architecture to obtain a final model. For this project, VGG16 model with weights from 'imagenet' dataset is used for transfer learning.

For the **loss function**, since it is a binary classification problem we use binary cross entropy or log loss function where y is the label and p(y) is the predicted probability of being y for all N points.

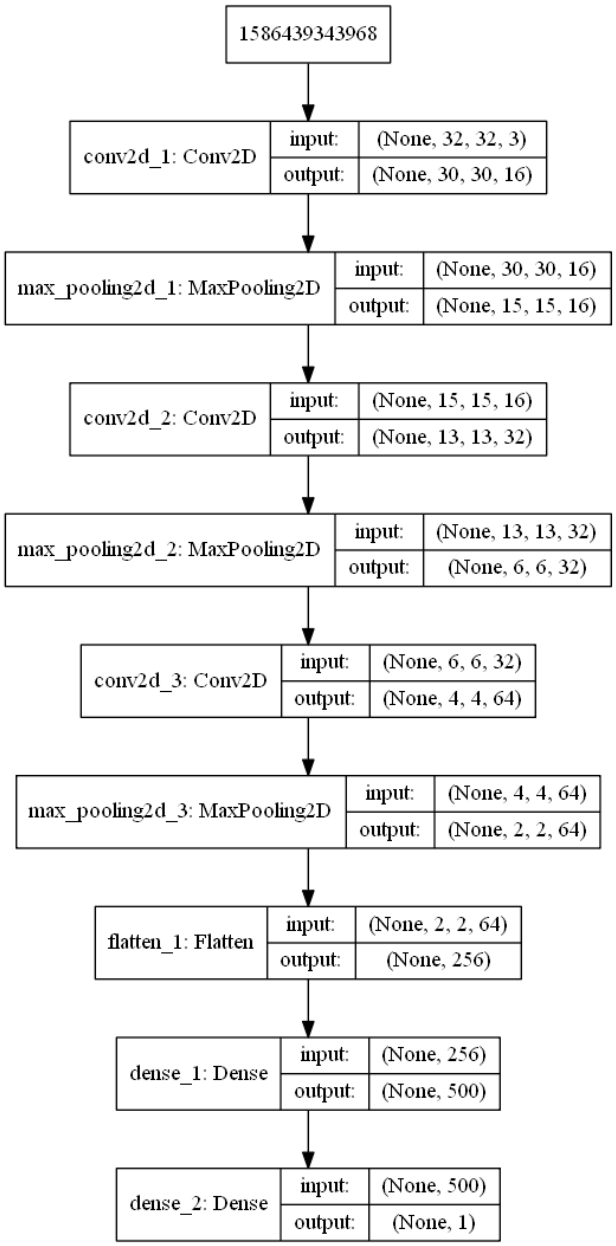
$$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i))$$

As for **optimizers,** adam optimization algorithm is used. Unlike stochastic gradient descent algorithm which maintains a single learning rate which doesn't change throughout the training, adam has a learning rate for each network weight or parameter and is adapted during the training phase.

For example: A sample CNN network in working is described below bullet points:

```
model = Sequential()
model.add(Conv2D(16, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(MaxPool2D(2, 2))
model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(MaxPool2D(2, 2))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPool2D(2, 2))
model.add(Flatten())
```

model.add(Dense(500, activation='relu'))
model.add(Dense(1, activation='sigmoid'))



1. Input image of shape 32x32x3 (RGB channels) are fed as an input to the convolution 2D layer.
2. 16 3x3 sized filters are superimposed on the image with default strides=1, padding='valid', resulting in 16 30x30 sized feature map.
3. Then a max pooling layer of 2x2 kernel is applied resulting in 16 15x15 sized feature map.
4. Then, 32 3x3 sized filters are superimposed on the 16 15x15 sized feature map to get 32 13x13 sized feature map and so on.

5. Before entering Dense layer, we flatten the 64 2x2 sized feature map to get a single array of 256 values.
6. These 256 values are connected to 500 node Dense (fully connected) layer and finally these 500 nodes are connected to a single node.

As we move from first convolution layer to final layers, the network first create feature maps to figure out colored regions and edges in the input image, then finds high-level shapes and contours out of them. Unlike a fully connected network where all nodes in a layer are connected to all nodes in the next layer, in a CNN layer, since each neuron is connected to only a subset of input image. This helps in parameter sharing or sharing of weights by all neurons in a particular feature map and this helps reduce the number of parameters to learn compared to a fully connected layer.

Benchmark

A simple benchmark model comprises of three convolutional 2D layers (16, 32, 64) of kernel size 3x3. All of these layers use relu activation. Three max pooling layers (2x2) follow each of these convolutional 2D layers. And, later these are trained on 500 node fully connected layer with relu activation, and a 1 node layer since it is a binary classification problem. The final layer of single node is set with sigmoid activation to give probability estimates for each class.

```
model = Sequential()
model.add(Conv2D(16, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(MaxPool2D(2, 2))
model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(MaxPool2D(2, 2))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPool2D(2, 2))
model.add(Flatten())
model.add(Dense(500, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

This model gave a val_loss of 0.0355 and a validation accuracy of 98.88%.

Also, results from [4] show a very high recognition accuracy of 95% on the validation set of columnar cactus identification with an image dataset of more than 10, 000 images. These two models, simple CNN and another benchmark of 95% accuracy will be used to assess the quality of the machine learning model.

METHODOLOGY

Data Preprocessing

Normalization:

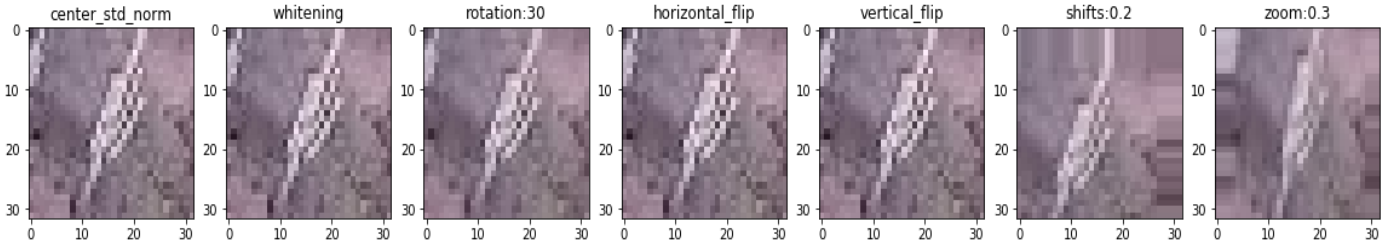
In the data preprocessing step all the field values of train data column has_cactus was made into a string field. Also, all the images are scaled to be between 0 and 1 by dividing them by 255 value.

Handling class imbalance:

There is a *class imbalance* issue, since there are **13,136** images of cactus vs **4363** images of no cactus in the dataset. Using *sklearn* library's *class_weight* module, class weight were computed from the training data and used in the training.

Augmentation:

In the refinement of model, for image augmentation keras' *ImageDataGenerator* module is used to transform the training images. Sample augmentation on a cactus image can be viewed below:



Implementation

The training dataset was split into 70-30 where 70% were used in training and 30% were used in validation for the first two models. For the remaining models, *StratifiedKFold* was used with number of five number of splits. Details of these models can be viewed below.

Model 1: Benchmark model (as discussed above)

Model 2: Benchmark model with Dropout layers of 0.1 probability to reduce overfitting.

```
model = Sequential()
model.add(Conv2D(16, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(MaxPool2D(2, 2))
model.add(Dropout(0.1))
model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(Dropout(0.1))
model.add(MaxPool2D(2, 2))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Dropout(0.1))
model.add(MaxPool2D(2, 2))
model.add(Flatten())
model.add(Dense(500, activation='relu'))
model.add(Dropout(0.1))
model.add(Dense(1, activation='sigmoid'))
```

Refinement

In the refinement step, image augmentation techniques and pre-trained classifier VGG16 model was explored and the predictions obtained from these models on the test dataset were submitted to Kaggle.

Model 3. Model 3 but trained with augmented image data.
In the image augmentation step, the parameters were set as, *width_shift_range=0.2*, *heigh_shift_range=0.2*, *rotation_range=30*, *zoom_range=0.3*, and both *vertical_flip* and *horizontal_flip* were set to *True*.

Model 4. VGG16 model.
The VGG16 top layers were not included and for the lower layers, three dense layers of node size 500, 300 and 100 were used with activation function *'relu'*. *BatchNormalization* and *Dropout* layers with 0.5 probability of dropout were used following each dense layer. And for the final output layer a single node dense layer with *'sigmoid'* activation was used.

```
vgg_model = Sequential()
vgg_model.add(vgg16)
vgg_model.add(Flatten())
vgg_model.add(Dense(500, activation = 'relu'))
vgg_model.add(BatchNormalization())
vgg_model.add(Dropout(0.5))
vgg_model.add(Dense(300, activation = 'relu'))
vgg_model.add(BatchNormalization())
vgg_model.add(Dropout(0.5))
vgg_model.add(Dense(100, activation = 'relu'))
vgg_model.add(BatchNormalization())
vgg_model.add(Dropout(0.5))
vgg_model.add(Dense(1, activation = 'sigmoid'))
```

Difficulties encountered in the project:
For deep learning models, to fit all of the dataset at once into memory isn't ideal so keras api has generators to input data in a batch wise fashion in the model. These generators also allow dataset augmentation on the fly without having to do it in memory. An example of how model predicts classes of an image is described below along with an issue encountered and how it was resolved:

```
# create a image generator that scales all images by 255 values to lie within interval [0, 1]
gen = ImageDataGenerator(rescale=1./255)

# from a dataframe train_df and it's index, flow_from_dataframe gets data in train_batch_size
# from the dataframe to feed into the model
train_gen = gen.flow_from_dataframe(dataframe=train_df[:ind],directory=train_dir,x_col='id',
                                    y_col='has_cactus',class_mode='binary',batch_size=train_batch_size,
                                        target_size=(target_size,target_size), shuffle=False)
```

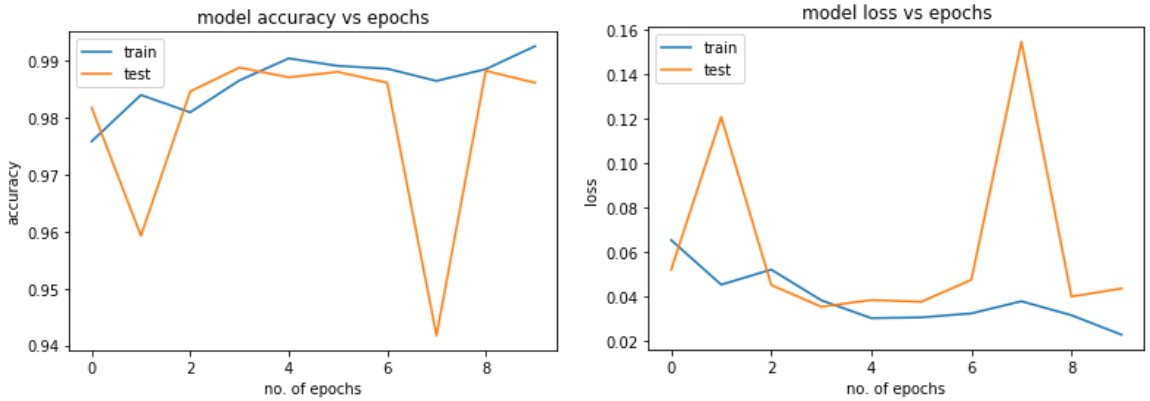
Such generators for both the training and validation data are fit to the model using *fit_generator*. And, function *predict_classes* is used to predict the class of an image. However, in the *flow_from_dataframe function*, *shuffle=False* has to be set. If *shuffle=False* is not set, then by default it is set to *True* which means the indices of train classes and train predictions get shuffled when model trains on it, and similarly for validation data too. This shuffling of indices gives incorrect predictions when using *predict_classes* function and affects the calculation of metrics such as accuracy, roc scores, etc.

RESULTS

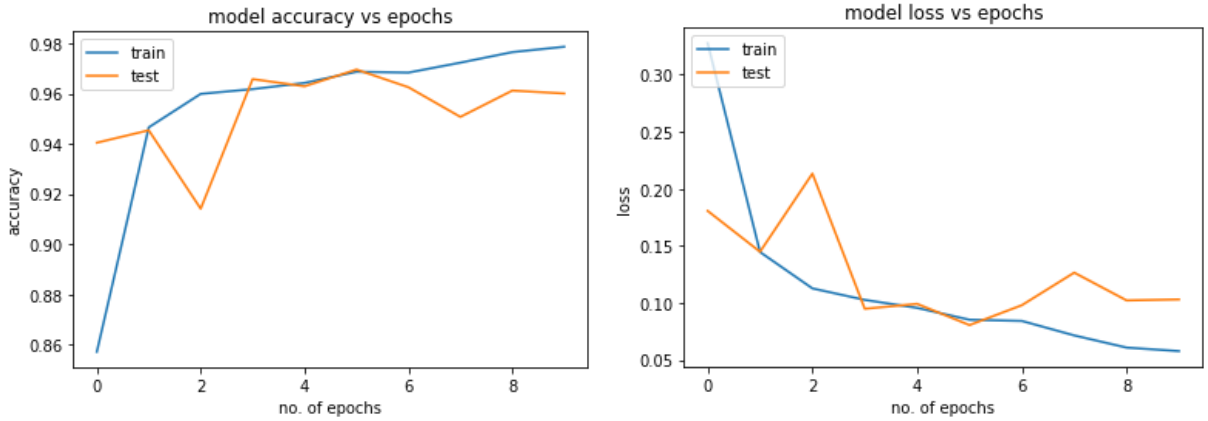
Model Evaluation and Validation

Model 1 and 2 were trained for 10 epochs with 70-30 train-test split data. Model 3 and 4 were run over 5 splits using *StratifiedKFold* for 20 epochs.

Model 1: The best model has a val_loss of 0.0355 and a validation accuracy of 98.88%.

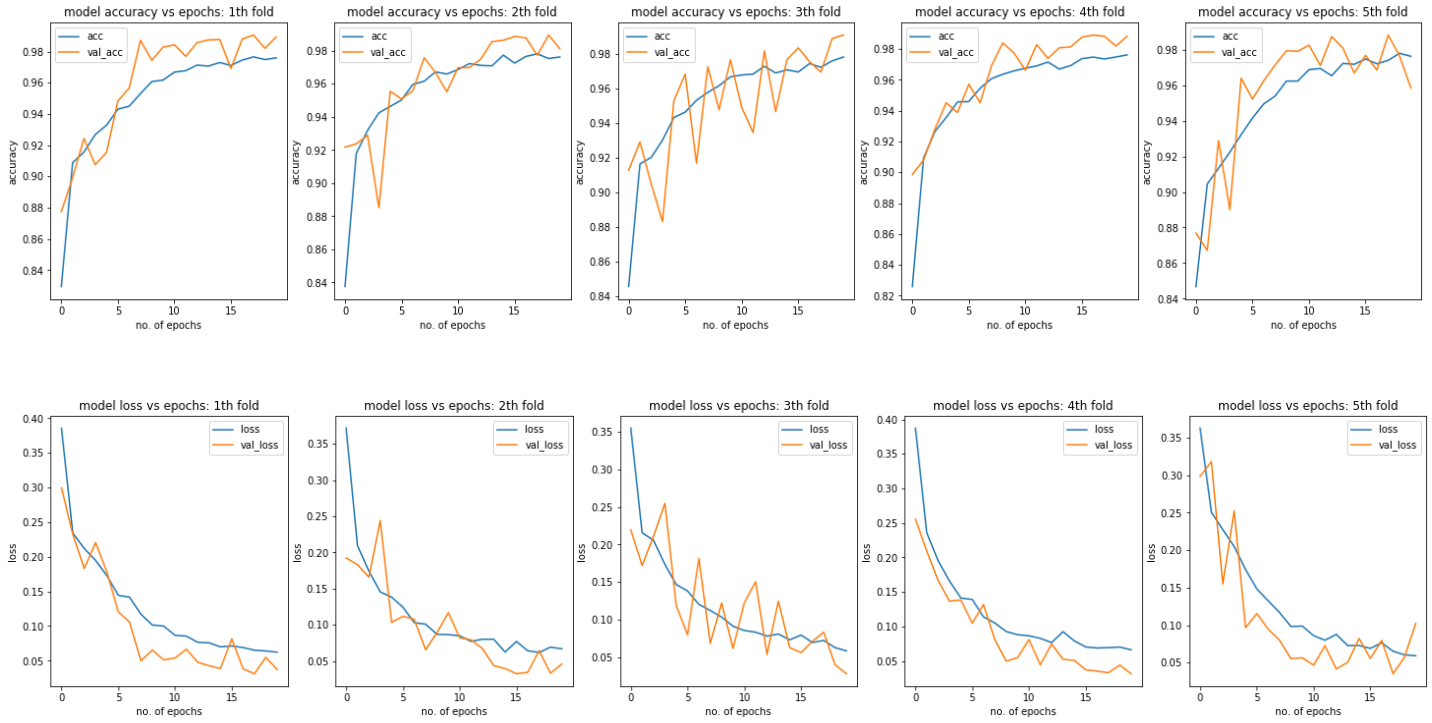


Model 2: The best model has a val_loss of 0.0806 and a validation accuracy of 96.95%.

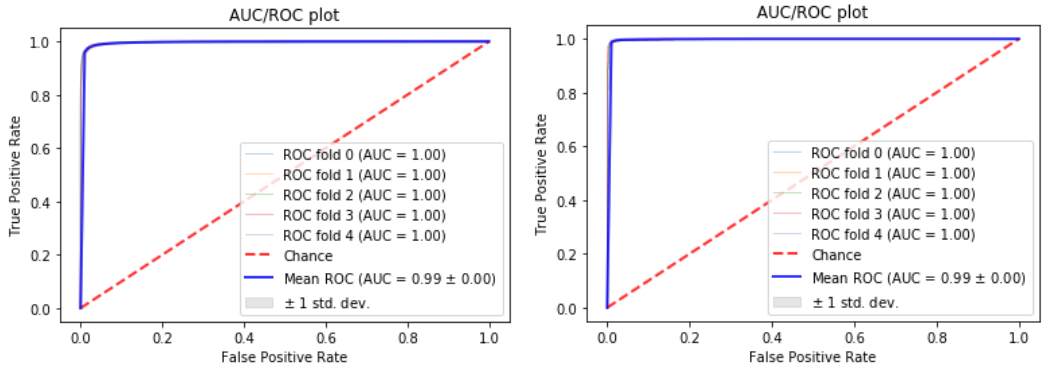


Model 3: The following plots are model accuracy vs no.of epochs for split size 1 to 5 respectively. The best model has a val_loss of 0.0312 and a validation accuracy of 99.06% at *fold=1* to val_loss of 0.0352

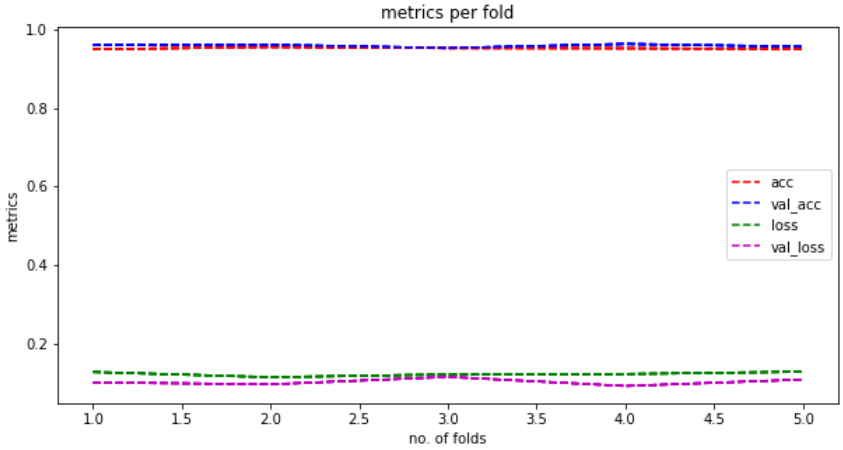
and a validation accuracy of 98.83% at *fold=5*. The results confirm higher the split, lower the number of training examples for the model to learn from. Due to space restriction for this graph, please see this graph in the ipython notebook.



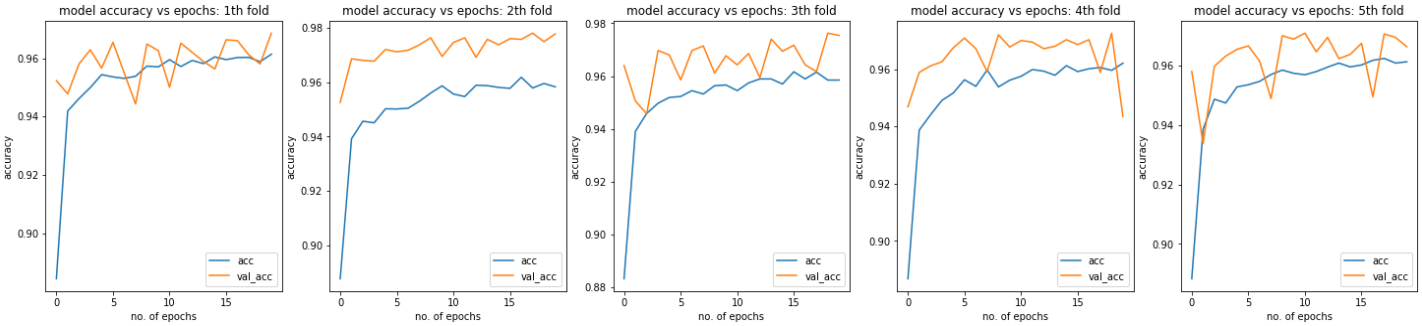
ROC curves observed for the train (left) and validation data (right) is shown below: The AUC score show how good the classifier is at separating true positives from false positives. For the data we observe a very good roc plot with area under the curve=1 for all k-folds.



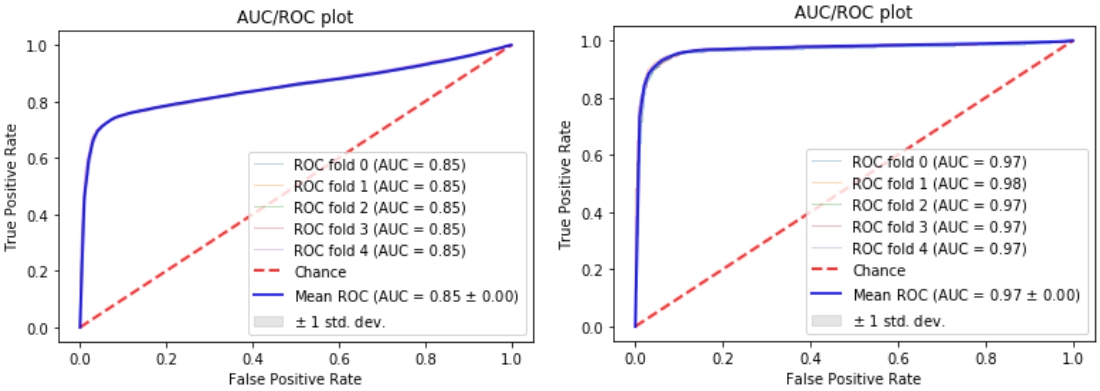
The following plot is the average of metrics such as accuracy, loss over no.of epochs for the model.



Model 4: The following plots are model accuracy vs no.of epochs for split size 1 to 5 respectively. The best model has a val_loss of 0.0950 and a validation accuracy of 96.60% at *fold=1* to val_loss of 0.0584 and a validation accuracy of 97.80% at *fold=5*. Due to space restriction for this graph, please see this graph in the ipython notebook.

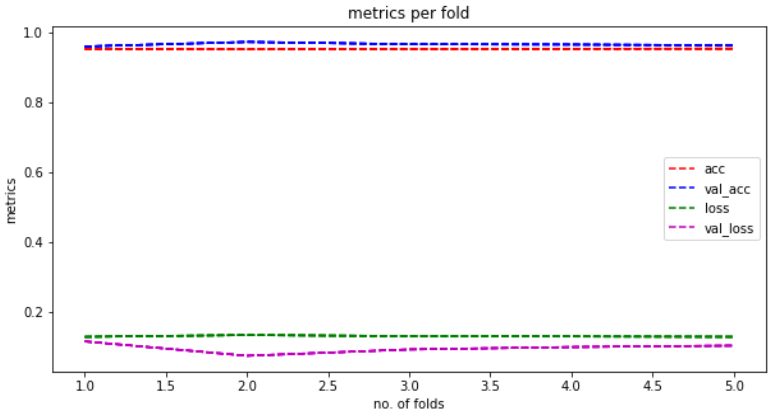


The plots of ROC curve for train(left) and test(right) can be viewed below:



The AUC score for train data for all splits average at 0.85 whereas the trained classifier does a good job at predicting the validation dataset with auc score for validation data for all splits at an average of 0.97. The

closer the AUC score is to 1, the better it is at separating true positives from false positives. This AUC score shift from train to validation data could mean classifier is generic enough and is not overfitting.
The following plot is the average of metrics such as accuracy, loss over no.of epochs for the model.



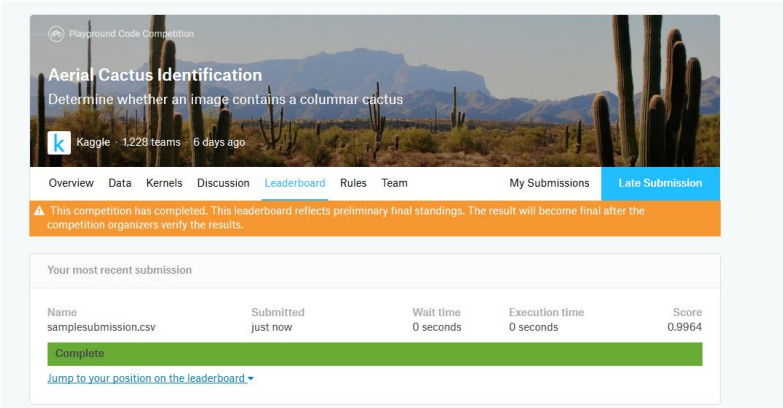
Justification

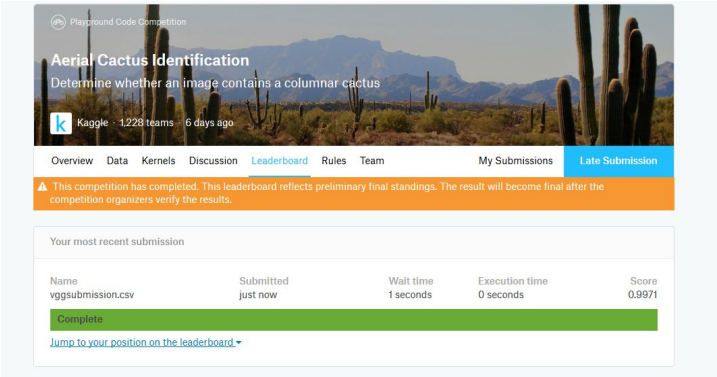
All the models validation accuracy is over 95% accuracy obtained in [4], which makes the models good for solving the problem at hand. Model 1 outperforms all of them, yet it is a simple CNN model without Dropout, Image Augmentation. This could mean overfitting.
Model 3 was a refinement over Model 2 with that training data being fed were augmented. However, for an unseen dataset, a simple CNN model such as in Model 1 may not be adaptive to new and unseen dataset, so using a pre-trained VGG16 model could help build a generic enough classifier. This was the goal in building Model 4.

CONCLUSION

Free-Form Visualization

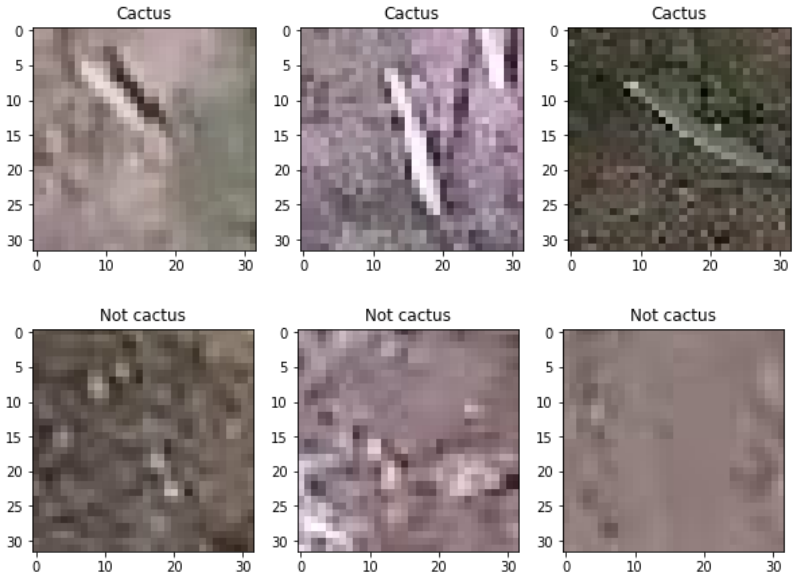
The test dataset was predicted using Model 3 and submitted to Kaggle to obtain a score of 0.9964 score for area under ROC curve.



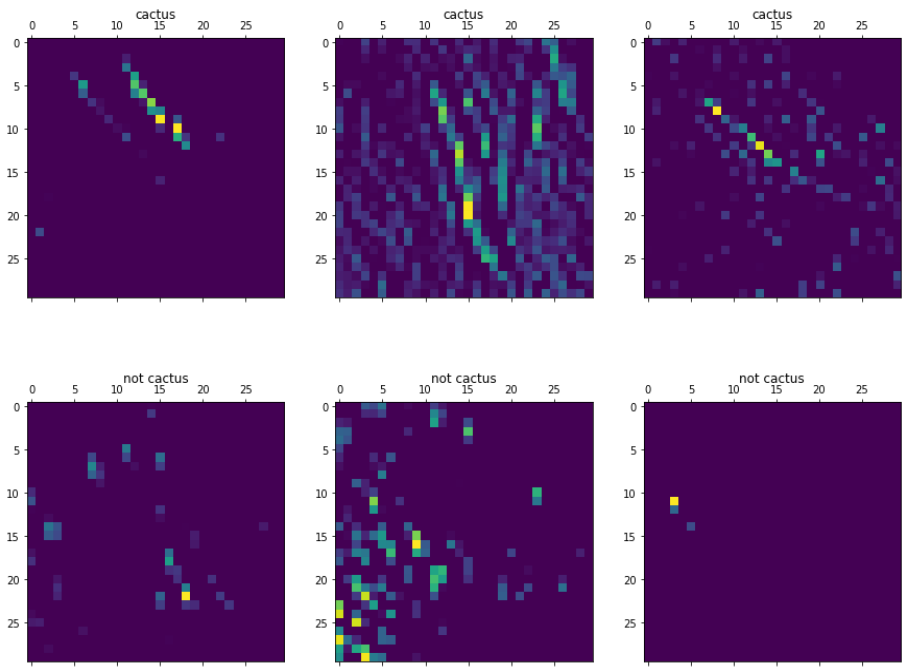


The test dataset was predicted using Model 4 and submitted to Kaggle to obtain a score of 0.9971 score for area under ROC curve. Snapshots from the kaggle submission has been shown here to show that VGG model performed better in the unseen dataset as it has a better area under the ROC curve than Model 3.

Some image prediction plot using Model 4 has been shown below, where cactus has been correctly labeled as cactus (shown in the title of each plot) and similarly for no cactus image:



Below are the plots of the activation function working on the images using Model 3 for cactus and no cactus images respectively.



Reflection

Reflecting back on the strategy used in this project, key points can be summarized below:

1. Read the train and test data images.
2. Convert JPEG images values to RGB pixel information and to floating tensors to be fed into neural network.
3. Re-scaling is done to all images to a [0,1] interval by dividing by 255.
4. Benchmark model is built to get a simple CNN architecture.
5. Refinement over benchmark model is done by adding Dropout layers and doing image augmentation.
6. To cater for large unseen dataset, a well-established pre-trained model like VGG16 model was used.
7. For each model, their validation losses and validation accuracies were observed.
10. The result were submitted will be in a *.csv* file to Kaggle and of the format as shown below:
    a. **Id -** *name of the image file in the test folder*
    b. **Has_cactus** - *the predicted probability of the image being a cactus*

Interesting/Challenges of the project:

Most interesting as well as challenging part of the project was trying to solve the issue of incorrect predictions because of shuffle being set to *True* in the *flow_from_dataframe*. Another challenge was long training times which meant some issues such as the aforementioned one couldn't have been known beforehand. Also, figuring out what a benchmark model could be for comparing against further refinements to a model was a challenge.

Improvement

Some improvements for the project could be the following points

1. Using *GridSearchCV* to obtain best model parameters to train.
2. Add gaussian noise to an image and see how the model performs.
3. Using other pre-trained models and see how they perform.
4. Exploring other image augmentation techniques.
5. Exploring other fully connected network architecture to connect as lower layers of pre-trained classifiers such as VGG16.
6. Training for more epochs could be done.
7. Try something like in paper [5], which is cycling learning rates for training neural networks. The idea is to change learning rate from small value until loss stops decreasing before bumping it up again. And, observe the plot of learning rate across batches.

References:

[1] https://jivg.org/research-projects/vigia/
[2] https://www.kaggle.com/c/aerial-cactus-identification/data
[3] https://www.kaggle.com/c/13435/download-all
[4] López-Jiménez, Efren, et al. "Columnar cactus recognition in aerial images using a deep learning approach." *Ecological Informatics* 52 (2019): 131-138.
[5] https://arxiv.org/abs/1506.01186