Capstone Project
Invasive Species Monitoring

Date – August 6, 2017

Problem Description

Tangles of kudzu overwhelm trees in Georgia while cane toads threaten habitats in over a dozen countries worldwide. These are just two invasive species of many which can have damaging effects on the environment, the economy, and even human health. Despite widespread impact, efforts to track the location and spread of invasive species are so costly that they're difficult to undertake at scale.

Currently, ecosystem and plant distribution monitoring depends on expert knowledge. Trained scientists visit designated areas and take note of the species inhabiting them. Using such a highly qualified workforce is expensive, time inefficient, and insufficient since humans cannot cover large areas when sampling.

Because scientists cannot sample a large quantity of areas, some machine learning algorithms are used in order to predict the presence or absence of invasive species in areas that have not been sampled. The accuracy of this approach is far from optimal, but still contributes to approaches to solving ecological problems.

[Source: Kaggle - https://www.kaggle.com/c/invasive-species-monitoring]

Introduction

Based on the problem description and the initial data analysis, we employ deep learning technique to solve the problem. We will use need the below packages to implement the solution for this problem:

1. Keras – A high Level API written in Python for Tensorflow and Theano convolutional neural networks
2. Tensorflow – Tensorflow is an open source library for numerical computation using data flow graph
3. Numpy/Scipy – A python numerical & scientific computation library
4. Matplotlib – A Python 2D plotting library
5. Python Imaging Library (PIL) – A python image processing library

Deep Learning employs some of the below steps for solving the problem:

1. Dataset Pre-Processing and Dataset Augmentation
2. Convolutional Neural Network (CNN) Model definition
3. CNN Model tuning with Hyper parameters

We will go through each of the above-mentioned steps in more detail in the document. At the end of the document we will discuss the results from our simulation run and discuss the learning as part of this project implementation in the conclusion section.

The CNN built will predict if the species is Invasive or Non-Invasive plant.

Dataset Pre-Processing and Dataset Augmentation

Since the original dataset has limited images, to improve accuracy we augument the data with additional images by applying the below image processing techniques:

1. Randomly change brightness of image
2. Randomly change the sharpness of image
3. Randomly flip the image horizontally
4. Randomly shift the image
5. Randomly rotate the image

As part of image pre-processing we resize the image to (299, 299), convert RGB image to YUV, and then normalize the image. After the preprocessing step is done the image is used to train the Convolutional Neural Network (CNN)

Non-Invasive Species

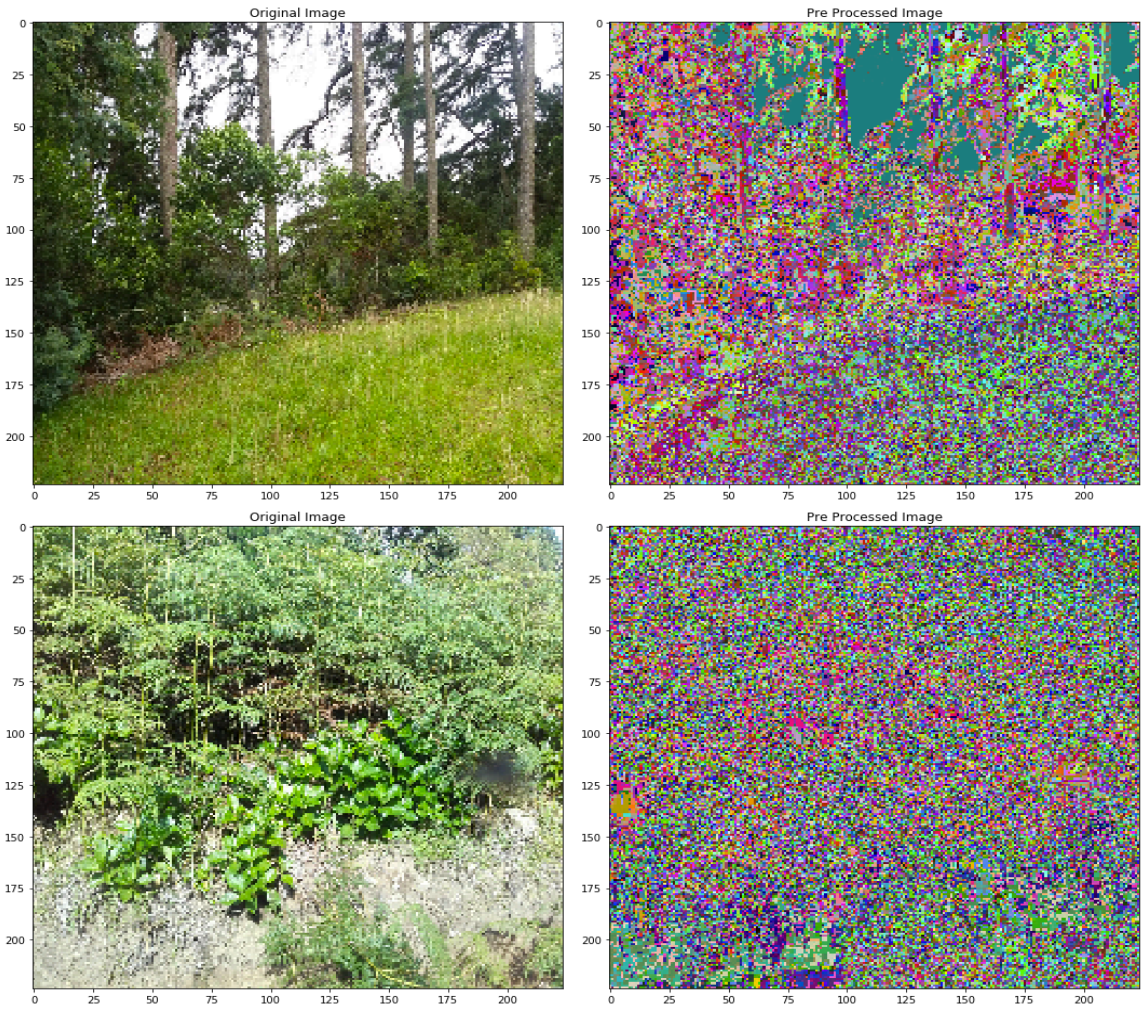
Invasive Species

We apply the pre-processing techniques discussed above, on these set of images. The pre-processing code looks like below:

```
##
Preprocess the image before feeding to the CNN
def pre_process_img(img, enable_zca = False,
                    enable_yuv = True,
                    training = False,
                    normalize = True,
                     target_size = (299, 299)):
    img_path = '../capstone_project_data/train/{}.jpg'.format(img)
    x = image.load_img(img_path, target_size = target_size)
    if training:
        if np.random.random() < 0.5:
            x = apply_modify(x, Brightness, .8, 1.2)
        if np.random.random() < 0.5:
            x = apply_modify(x, Sharpness, 1., 2.)

    x = image.img_to_array(x)


    if training:
```

```
        if np.random.random() < 0.5:
            x = flip_horizontal(x)
            if np.random.random() < 0.5:
                x = image.random_shift(x, 0.2, 0., row_axis = 0, col_axis = 1,
channel_axis = 2)
            if np.random.random() < 0.5:
                x = image.random_rotation(x, 30., row_axis = 0, col_axis = 1,
channel_axis = 2)

    if enable_yuv:
        x = cv2.cvtColor(x, cv2.COLOR_RGB2YUV)

    if enable_zca:
        x = zca_whitening(x)

    if normalize:
        ## Normalize the image
        x = (x / 255. - 0.5).astype('float32')

    return x
```

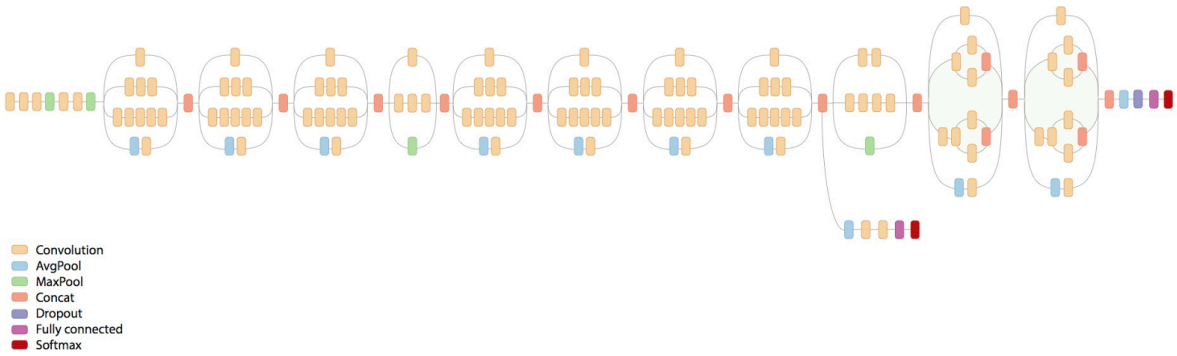
The images look like before after preprocessing through the above method.



Convolutional Neural Network (CNN) Model definition

We here use the Inception V3 model from Keras to build the CNN. An inception V3 model is a trained model on imagenet with 1000 classes. We use transfer-learning technique here, we used an already trained Inception V3 CNN model on imagenet, and then we add layers at the output from the Inception V3 to match the number of classes.

Inception V3 Model Architecture



Transfer Learning

Transfer learning provides the opportunity to adapt a pre-trained model to new classes of data with several advantages. In this case we load the Inception V3 model with pre-trained weights on imagenet. This method saves significant time in training.

In our model, we add layer of Dropout to improve Generalization and avoid overfitting.

The code for the Inception V3 model with transfer learning is below:

```
def get_model(num_classes):

        base_model = InceptionV3(weights='imagenet', include_top=False)
        x = base_model.output
        x = GlobalAveragePooling2D()(x)
        x = Dense(256, activation='relu')(x)
        x = Dropout(0.25)(x)
        x = Dense(16, activation='relu')(x)
        x = Dropout(0.25)(x)
        pred = Dense(num_classes, activation='softmax')(x)
```

```
    model = Model(inputs=base_model.input, outputs=pred)
    #adam = Adam(lr=0.001)
    #sgd = SGD(lr=0.0001, momentum=0.9)
    model.compile(loss = 'categorical_crossentropy',
                  optimizer = Adam(lr=0.0001),
                  metrics = ['accuracy'])
    return model
```

CNN Model tuning with Hyper parameters

We use Adam optimizer in our model to minimize the loss. The reason for using Adam optimizer is because it converges faster compared to Stochastic Gradient Descent (SGD), with very less tuning. However with proper tuning SGD can also achieve comparable or better performance than Adam Optimizer in Practice.

Below are the CNN model hyper parameters tuned for this model.

1. Learning rate - Tried different learning rate and the best learning rate of 0.0001 worked better accuracy
2. Number of epochs - This parameter was primarily limited in my simulation to time used, so choose value of 4
3. Batch Size - Tried batch sizes of 8, 16, 32, 64. The best that worked was 16 on my system

Below is the code for the tuned and compiled CNN model.

```
def train_model():
    df = pd.read_csv('../capstone_project_data/train_labels.csv')
    images = df['name'].values
    labels = df['invasive'].values
    batch_size = 16
    train_x, test_x, train_y, test_y = train_test_split(images, labels,
    test_size = 0.2,
    random_state = 42)
    print("Train Size: %d" % len(train_x))
    print("Test Size: %d" % len(test_x))
    train_y = np_utils.to_categorical(train_y)
    test_y = np_utils.to_categorical(test_y)
    num_classes = train_y.shape[1]
    print("Num classes %d" % num_classes)
    model = get_model(num_classes)
    model.fit_generator(training_generator(batch_size, train_x, train_y),
                        steps_per_epoch=len(train_x) // batch_size,
                        epochs = 4,
                        verbose = 1,
                        callbacks = [],
                        validation_data = validation_generator(batch_size, test_x,
```

```
test_validation_steps = len(test_x) // batch_size)

with open("model.json", "w") as fp:
        json.dump(model.to_json(), fp)
        model.save_weights("model.h5", overwrite = True)

print("Done!!!")
```

Metrics

Here we use the Cross entropy loss function for the Adam optimizer. It is a standard practice to used cross entropy function in deep neural networks, with softmax activation. The Cross entropy loss functions is defined as below,

$$L(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} H(p_n, q_n) = -\frac{1}{N}\sum_{n=1}^{N} \left[y_n \log \hat{y}_n + (1-y_n) \log (1-\hat{y}_n) \right],$$

Simulation Results

The output from the model training is as below:

```
Train Size: 1836
Test Size: 459
Num classes 2
Epoch 1/4 114/114 [==============================] - 2173s - loss: 0
.2650 - acc: 0.8942 - val_loss: 0.1534 - val_acc: 0.9554
Epoch 2/4 114/114 [==============================] - 2161s - loss: 0
.1295 - acc: 0.9518 - val_loss: 0.0596 - val_acc: 0.9688
Epoch 3/4 114/114 [==============================] - 2259s - loss: 0
.1083 - acc: 0.9622 - val_loss: 0.0862 - val_acc: 0.9598
Epoch 4/4 114/114 [==============================] - 2321s - loss: 0
.1089 - acc: 0.9583 - val_loss: 0.1139 - val_acc: 0.9576
Done!!!
```

Based on the above results. The training accuracy is: **0.9583** and the validation accuray is: **0.9576**

Conclusion:

1. CNN are best suited for Deep Learning
2. Data pre-processing and data augmentation are necessary steps for imporved accuracy and faster convergence
3. Tuning hyper-parameters like learning rate, number of epochs, batch_size are equally important for good accuracy score
4. Keras library is great way to learn and implement Deep Learning using CNN
5. With only 4 epochs, I was able to achieve > 95% accuracy

Reference:

1. https://www.kaggle.com/c/invasive-species-monitoring
2. https://keras.io/
3. https://www.tensorflow.org/
4. https://research.googleblog.com/2016/03/train-your-own-image-classifier-with.html