# Machine Learning Engineer Nanodegree

**Plant Seedlings Classification with Transfer Learning**

Larry Hernandez

October 1, 2018

## I. Definition

### Project Overview

The problem I proposed resides in the domain of Deep Learning for Image Classification. Given images of several types of object (i.e., cats), the goal is to train a deep neural network that classifies each one of those images into exactly one of several sub-categories of that object (i.e., tuxedo cat). This is a common, contemporary problem which arises in many industries, including but not limited to e-commerce, robotics, and dating apps.

Deep convolutional neural networks (CNNs) are known to produce successful image classifiers, including handwritten digit recognition and skin cancer classification. In such cases, it is possible to use "transfer learning" to train a new CNN from an already existing CNN architecture. Rather than spending several hours developing a variety of CNN architectures from scratch, it is faster to take a pre-trained CNN and adapt it to the specific image classification problem at hand. This is because a pre-trained CNN already has the ability to recognize general patterns that are common for a variety of image objects. Subsequent training with the problem-specific data allows the CNN to learn the finer details that are relevant to the problem.

Transfer learning is applied in this capstone project, which is a Kaggle "playground-level" image classification competition called Plant Seedlings Classification. Kaggle community members were given the opportunity to develop algorithms that categorize images of plant seedlings into one of twelve classes. Some classes correspond to valuable crops and the rest correspond to (invaluable) weeds, which compete with crops for the same agricultural resources. If weeds are identified at an early growth stage—when they are seedlings—they can be physically removed, which then gives crops increased chances of prospering since they will not need to compete with the weeds for life-sustaining resources. Consequently, an effective image classifier installed on a smart phone can help farmers keep their plots (mostly) free of weeds and, perhaps, ultimately lead to increased yields of valuable crops.

### Datasets and Inputs

The image data for this project are posted here on the Kaggle website. The images were acquired by researchers affiliated with University of Southern Denmark and Aarhus University in Denmark. Researchers utilized a dSLR camera to obtain 4,750 images of twelve different types of plant seedlings. Images of these seedlings were acquired over a 20 day period at 2-3 day intervals after they emerged from the soil in order to capture their appearance throughout several growth stages.

### Problem Statement

The problem is the following: build a model that effectively classifies the twelve varieties of plant seedlings from their color images. The goal is to achieve as high of an F1-score as possible, with 1.0 being the largest possible value, by adjusting parameters of the convolutional neural network.

Deep CNNs are effective at learning patterns in images if given enough good quality training data and the appropriate architecture. For this project, a technique called "transfer learning" will be used. In this application of transfer learning a pre-trained convolutional neural network will be adapted to develop an image classifier. In particular, the VGG19 architecture (**Figure 1**) with pre-computed weights will be utilized. This pre-trained CNN is capable of categorizing images across 1,000 different classes. Half of these classes are natural objects, such as cats or dogs; the other 500 classes are man-made objects, such as a microwave or oscilloscope. Not one of the classes is a plant. Despite the fact that the plant seedlings would be a new prediction category for this CNN, it should be possible to leverage the pre-trained VGG19 network to create a CNN that adequately classifies the various plant seedlings. The reason is that the bottom layers of the VGG19 network have already learned general features that would be useful for any image object. Adding a specialized set of output layers to the top of the VGG19 base network and subsequently training this aggregated network with the plant seedlings image data should lead to a specialized solution to this problem.

The initial proposal for the transfer learning model includes the following three steps. First, the three fully connected layers of the VGG19 model are removed. These three removed layers of VGG19 are specialized to the 1,000 ImageNet classes that the model was originally developed for. The softmax at the top of the network contains 1,000 outputs, one for each of the 1,000 classes. Since we only care about 12 plant seedlings classes rather than the 1,000 ImageNet classes (none of which include these plant seedlings), discarding these three fully connected layers is important. The resulting architecture is the first 22 layers of VGG19.

In the second step of the original proposal, contiguous blocks of 2D convolutional layers at the top of the truncated VGG19 network are discarded to leave 4, 7, 12, or 17 layers. Removing block E leaves the first 17 layers (**Figure 1**); removing blocks D & E leaves the first 12 layers; removing blocks C, D, and E leaves the first 7 layers, etc.

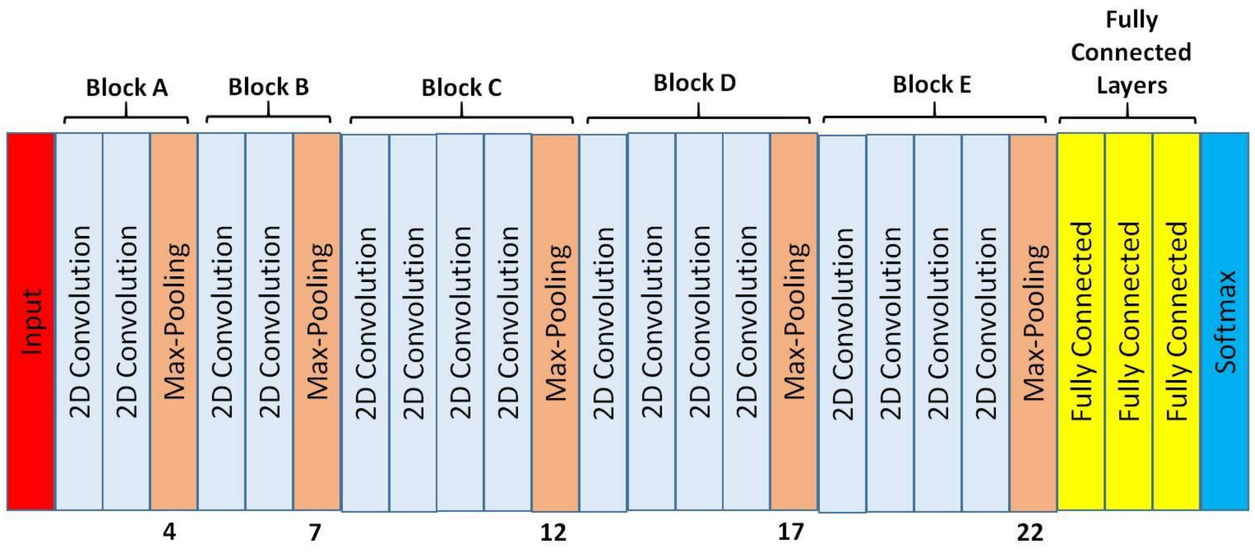


**Figure 1: The full VGG19 architecture. Groups of convolutional and max pooling layers are labeled with letters. The numbers below the max pooling layers indicate which (number) layer they are in the sequence.**

In the third step of the transfer learning process, a small neural network is placed on top of the revised VGG19 network (**Figure 2**). This "top-model" consists of the following: a flattened layer, a dense layer followed by a dropout layer, and another dense layer with a softmax function having 12 outputs. Then, the training and validation plant seedlings images are passed through this aggregated CNN architecture over several epochs. During this step, the initially random weights of the top-model are updated while the weights of the convolutional blocks are held constant. The optimal number of convolutional blocks to remove would be determined through experimentation.

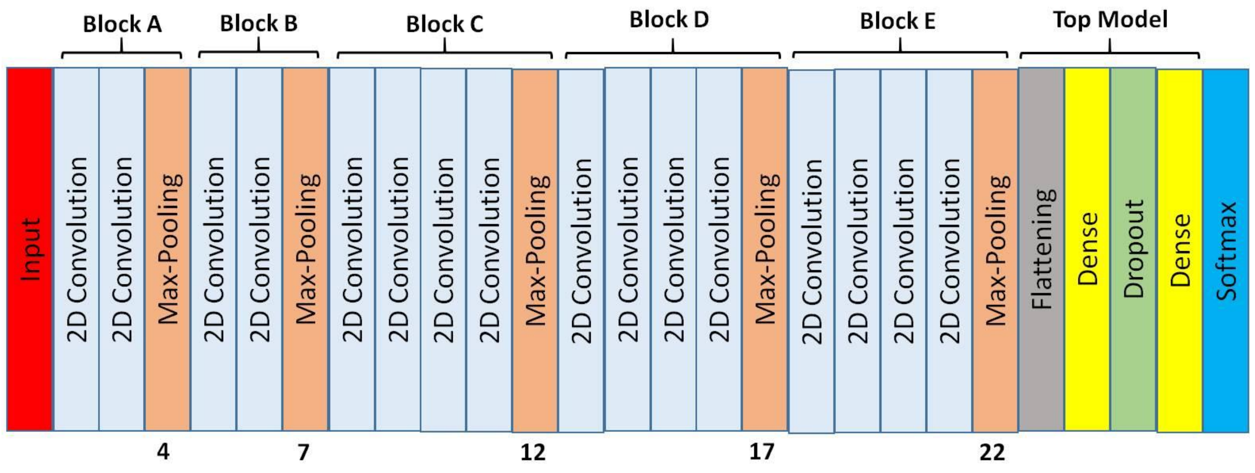


**Figure 2: The architecture of the combined VGG19 base (Blocks A-E) and the "top-model". This architecture ultimately serves as the solution model in this project.**

During the course of this project, it was determined that completely removing the blocks of 2D convolutional layers (i.e. blocks A, B, C) resulted in poor performance. F1-scores ranged between 0.08 and 0.12. There appears to be an issue with the Keras function *pop()* that is meant to remove these layers. Coding workarounds were attempted but efforts were fruitless. Consequently, the second and third steps in the originally proposed transfer learning process were modified. Instead of completely removing blocks of convolutions at the top of the VGG19 base model, every one of the first 22 VGG19 layers was kept in the model. From there, two additional steps were undertaken. In the first step, called "tuning", a top-model is added to the VGG19 base. The weights of all 22 VGG19 layers were held fixed while the weights of the top-model were updated. In the second step, called "fine tuning", the weights of the top-model and the weights of select, contiguous convolutional blocks (i.e., Blocks D & E) at the top (i.e. end) of the truncated, 22-layer VGG19 architecture are updated, while the weights of the remaining convolutional blocks (i.e. Blocks A, B, and C) at the bottom (i.e., beginning) of the network are held constant. Consequently, all models tested in this final iteration of the project include the first 22 layers of the pre-trained VGG19 model, with revised weights toward the top of the architecture.

Several model parameters were tested. These include the number of VGG19 layers to freeze, the number of densely connected neurons to use in the top-model, dropout rate of the top-model, and learning rate for the optimizer. Model development utilized training and validation sets. The CNN yielding the smallest loss on the validation set was chosen as the final model. The final model was then used to make predictions with a third image set, called the testing set, which has not been used until this point. The F1-score achieved on the testing set is the evaluation metric for the final model.

## Metrics

The evaluation metric for this problem was defined in the competition rules. It is the micro-averaged F1 score. This metric is appropriate for classification problems having unbalanced classes. The mathematical expression for this metric is the following:

$$F1_{micro} = \frac{2Precision_{micro}Recall_{micro}}{Precision_{micro}+Recall_{micro}} \tag{1}$$

where

$$Precision_{micro} = \frac{\sum_{k\in C} TP_k}{\sum_{k\in C} TP_k + FP_k} \tag{2}$$

and

$$Recall_{micro} = \frac{\sum_{k\in C} TP_k}{\sum_{k\in C} TP_k + FN_k} \tag{3}$$

In these equations, TP represents the number of true positive predictions, FP represents the number of false positive predictions, FN is the number of false negative predictions, and the index $k$ represents the image classes.

In classification problems, it is not uncommon to utilize accuracy as an evaluation metric. However, accuracy is only suitable when the classes are (mostly) balanced; that is, when the number of observations is approximately the same for all classes. In this problem, the classes are not balanced, so accuracy is not suitable and can be rather misleading. The micro-averaged F1-score is a suitable metric for this problem, because it is a (geometric) average of the algorithm's rates for (1) predicting the correct class among all of its class predictions, and (2) predicting the correct class compared to the total number of available correct class instances. In other words, the F1 score equally emphasizes the algorithm's precision and recall scores, and it does so through a geometric mean. This equal emphasis is important for this particular problem, because identifying the valuable plant seedlings is just as important as identifying the (invaluable) ones.

While the F1-score is used to evaluate the final model's predictions, the metric that is used to determine the optimal model during training is categorical cross entropy. This is because the categorical cross entropy is a function that can be directly minimized via the forward and backward propagation processes. The F1-score, on the other hand, cannot be directly optimized. It makes sense to utilize the categorical cross entropy as the loss function in this problem, since it is for multiple output classes.

## II. Analysis

### Data Exploration

The data set contains 4,750 images of the twelve different types of plant seedlings. Images were captured at 2-3 day intervals over a 20-day period after the seedlings first emerged from the soil. As a result, the seedlings appear in a variety of growth stages (**Figure 3**). The ability to recognize seedlings at various growth stages is an important aspect of this problem.

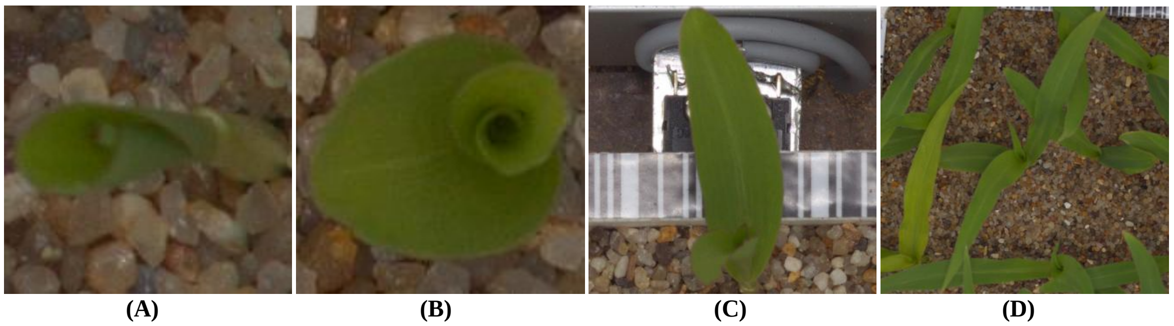


**Figure 3: Seedlings of the Maize plant at a variety of growth stages. Note: the seedlings are not segmented in the images and that in (D) there are multiple seedlings within the image.**

The number of images for each seedling type varies between 231 and 654. This is definitely an unbalanced data set (**Figure 5 D**). It is important to note that when the data are split into training, validation, and testing sets, this relative distribution should be preserved.

The images have red, green, and blue (RGB) channels, in that order. The widths and heights of the images vary. Consequently, the image processing step should address this; each image needs to be resized to a common width and height for the machine learning algorithms. The plant seedlings are NOT segmented (**Figure 3**), and occasionally the seedlings comprise only a small portion of the images. As previously mentioned, this should not pose too much of a problem. The CNN should learn to ignore the non-plant portions of the images.

The images were reviewed for basic image quality. It was determined that some of the image data are outliers and required removal. For example, some images did not contain a plant seedling (**Figure 4**). In some cases, the images were very small, extremely blurry, or both. If the image did not contain a plant seedling or was too blurry, then it was discarded (**Figure 4**).

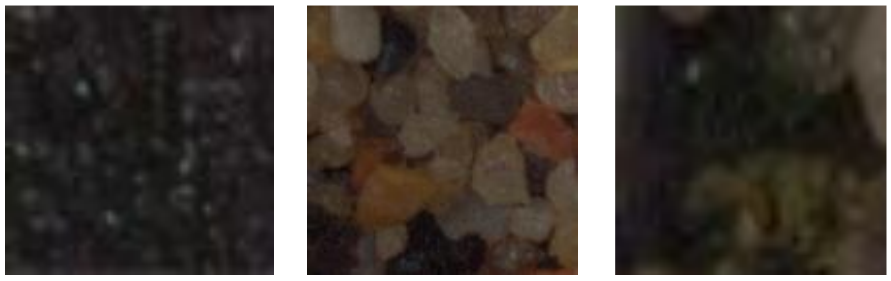
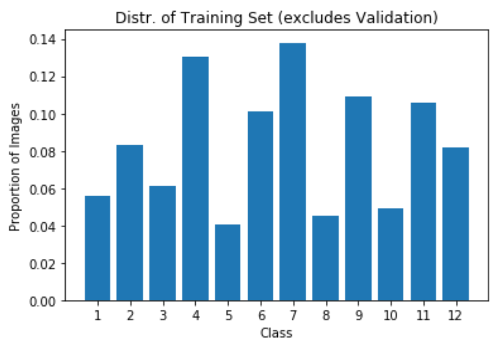


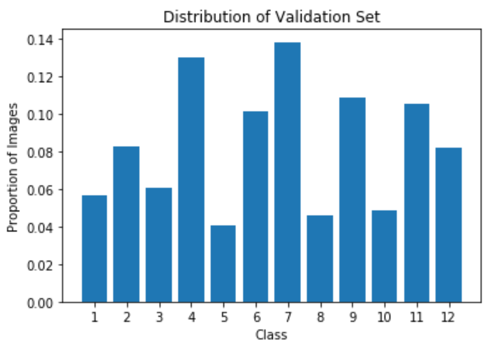
**Figure 4: Examples of images that seem not to contain a plant seedling and that should be discarded include (A) Common Chickweed, (B) Loose Silky-bent, and (C) Sugar beet.**

## Exploratory Visualization

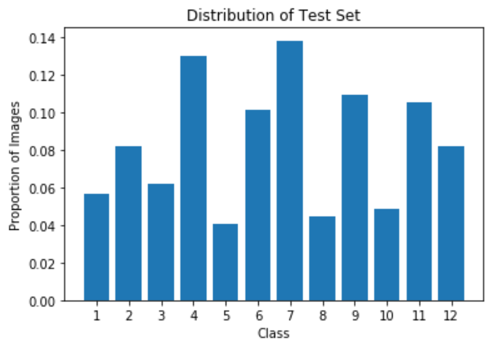
The data set is imbalanced (**Figure 5 D**). This reveals the importance of using the F1-score rather than accuracy as an evaluation metric. Another important consequence is that the training and validation data sets should have the same distribution as the full data set. Fortunately, that is possible. **Figure 5** reveals that the training, validation, and testing sets all have approximately the same distribution as the overall image data set. For example, class 1 constitutes approximately 6% of the training, validation, testing, and combined image data sets. Using training, validation, and testing data sets with comparable distributions is important for developing a reliable model.



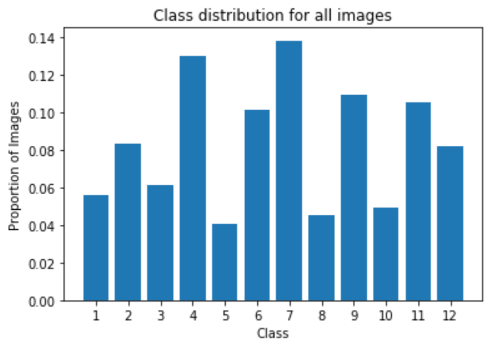
(A)



(B)



(C)



(D)

**Figure 5: The training (A), validation (B), testing (C) sets all have the same distribution of the unbalanced classes as the full data set (D).**

## Algorithms and Techniques

The machine learning framework that this project relies upon is called a convolutional neural network (CNN). Convolutional neural networks are a type of artificial neural network consisting of three or more convolutional layers, along with drop-out layers, densely connected layers, and pooling layers. The core building block of a CNN is a convolutional layer during which a convolution operation with a pre-specified 'receptive field' is applied to the full width and depth of the input. The result of convolving input data with a filter is a two-dimensional activation map of that filter. Consequently, the network learns filters that activate when a specific type of feature, such as a horizontal edge, at some spatial position in the input is detected. Each convolutional layer utilizes several filters, resulting in several 2D activation maps as output. Those activation maps are then stacked along the depth of the input to form a volumetric output data set.

The activation step is handled with a rectified linear unit (ReLu) layer. The function 'max(0,x)' is applied to the resulting convolutional operation to determine which neurons are "activated" in the output. The use of this ReLu introduces a nonlinear decision functional process to the network in a simple yet effective manner.

Pooling layers are a useful component to the CNNs here. They are effective means of nonlinearly down-sampling the data. In these processes, each "slice" of the volumetric data set is partitioned into non-overlapping, often small rectangular regions. A result is produced from these partitions. Typically, this result is the maximum value within each rectangle, but other options such as minimum or average value exist. These pooling layers serve several purposes, including reducing the spatial size of the features, number of parameters, and computation time. They also help to control overfitting. Pooling layers are often placed in between sequential convolutional layers.

Drop-out layers are utilized in the CNNs here. In a drop-out layer, a percentage of the nodes are eliminated or "dropped out" of the network resulting in a reduced network. Consequently, it is the reduced network that is trained on the data during that layer. By training the network on a reduced set of nodes with all of the training data, the dropout process decreases overfitting. An additional benefit is that training time is reduced somewhat.

Dense, fully connected layers are also employed at the end of the CNN. Dense fully connected layers are intended to perform the highest level of decision making. The neurons in these layers are connected to all activations in the previous layer. In this project, a flattening layer connects the output of max pooling layers with dense, fully connected layers. Additionally, a soft-max activation function is utilized with the last dense layer. The soft-max function produces 12 probabilities, each of which corresponds to a plant seedling class.

The filter weights, and hence extracted features, of the CNNs are learned through an iterative, optimization process known as backpropagation. This consists of passing the training data "forward" through the network, extracting weights, and calculating the value of a loss function. Then, a backward pass through the network; weights are adjusted by an amount that is proportional to the product of a "learning rate" constant and the derivative of the loss function with respect to the weights. This process is repeated over a pre-specified number of steps or epochs. The weights which lead to the lowest value of the loss function on the validation set are saved.

There are advantages to using CNNs for this type of problem. For one, filter sizes and weights are shared for convolution operations applied to the input. Weight sharing reduces the number of parameters that need to be trained, thereby making CNNs computationally efficient. Additionally, the convolution operation exploits spatially local correlation of image pixels via the convolution filter size. Each neuron of the CNN is only connected to a small area of the input, though they extend to the entire depth of the input. This ability of CNNs to preserve and learn from spatial correlations between adjacent image pixels is the primary reason that CNNs are a powerful approach for this image classification problem.

In this work, a method called "transfer learning" is used. The idea is that a pre-trained VGG19 convolutional neural network can be adapted to produce a plant seedlings image classifier. This general technique has worked for a variety of applications, including the aforementioned skin cancer classification problem.

The VGG19-based "transfer learning" solution proposed here is expected to perform reasonably well. At the very least, it is expected to outperform the proposed benchmark model that will be built from scratch (described below). The reason is that the model built with transfer learning has the advantage of starting with pre-computed weights obtained from prior training on the ImageNet database. Consequently, the pre-trained VGG19 model will have the ability to recognize general patterns through its pre-computed weights. Attaching a few dense layers to the end of the pre-trained VGG19 model, and subsequently training it with the plant seedlings images should yield a fairly effective classifier.

The final CNN will be the sequential combination of two architectures, the revised pre-trained VGG19 model and a trained "top-model" that is added to the VGG19 base (**Figure 2**). The primary purpose of the top-model is to specialize the CNN to the 12 plant seedlings classes. It also utilizes a dropout layer intended to prevent the network from overfitting. The top-model will first be trained in the following way. All of the weights of the VGG19 architecture will be held fixed but the weights of the top-model will be adjustable. The training and validation data sets will be used to train this version of the combined network. This process is really just training or "tuning" the top-model portion of the combined CNN. The weights of the VGG19 layers are frozen so that they are not drastically changed or destroyed by the large gradient updates that are triggered during the training of the top-model. The optimizer for this step will be the RMS-Prop. This optimizer will allow the weights of the top-model to be updated rather quickly, since it is adaptive, while the weights of the VGG19 base are held constant. The next step is to “fine-tune” the combined CNN. In this step, the weights of the top-model and a few of the top-most convolutional blocks of the VGG19 base are adjustable. The optimizer for the fine-tuning step will be the non-adaptive stochastic gradient descent (SGD) method with a slow learning rate. The reason for this choice of optimizer and small learning rate is that this combination ensures that the updates to the weights will be small in magnitude and, therefore, will not destroy the features that the model previously learned during the “tuning” step.

The input data will be pre-processed using the Keras function ‘preprocess_input’. This function resizes all images to the same size, scales them, and re-arranges their color channels so that they can be passed through the pre-loaded VGG19 neural network architecture. More details about the data preprocessing steps are provided in section III of this document.

### Benchmark

The benchmark model is a 14-layer convolutional neural network (**Figure 6**) trained from scratch. It has the following architecture: 2D convolution, 2D convolution, max pooling, 2D convolution, 2D convolution, dropout, max pooling, 2D convolution, 2D convolution, dropout, max pooling, flatten, dense layer, dense final output layer. Several values for the dropout rates, number of neurons for the first dense layer (i.e., 13th layer), and the max pool size were tested to yield the lowest validation loss (i.e. categorical cross entropy). The architecture of this benchmark model is similar to that of VGG19, but it has fewer layers and also incorporates two dropout layers to avoid overfitting. Given that its architecture is similar to VGG19, this benchmark model is expected to perform reasonably well with sufficient training.

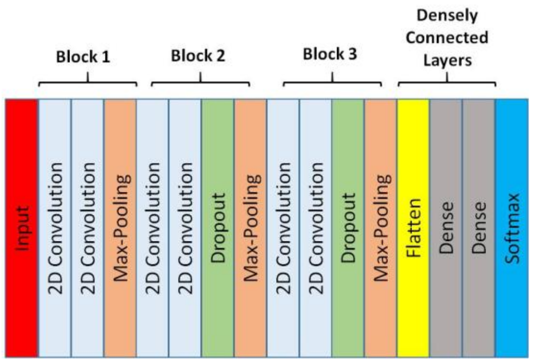


**Figure 6: Architecture for benchmark CNN.**

The benchmark was trained with the training and validation image sets using several values for the two dropout rates, number of neurons for the two dense layers, and the pool size. The combination of hyper-parameters that ultimately yielded the highest F1-score on the validation set were the following: imageSize = 224; dropout = 0.10 (Block 2 & 3); neurons = 500. This model achieved an F1-score of 0.75 on the test set. This F1-score of 0.75 is the value that needs to be exceeded by the solution model.

## III. Methodology

### Data Preprocessing

Data pre-processing began with the removal of outlier images. This includes images in which the plant seedling was too blurry for the green portion of the image to be deemed a plant or if discerning the plant type would be impossible. Images not containing plant seedlings were removed. This was done by manually inspecting every image in the full data set. After refining the image data set, one-hot encoding is applied to each image. This is important for utilizing the categorical cross entropy function too. The images are partitioned into training, validation and testing sets, containing 60%, 20%, and 20% of the full (revised) data set, respectively. Each partition maintains the relative proportion of the twelve classes.

The remaining image pre-processing steps for the benchmark and the transfer learning based models are the same. The difference is that for the benchmark model a few steps are performed manually; whereas, for the transfer learning based method using the pre-built VGG19 CNN, there is a prebuilt Keras function which handles most of the pre-processing steps automatically.

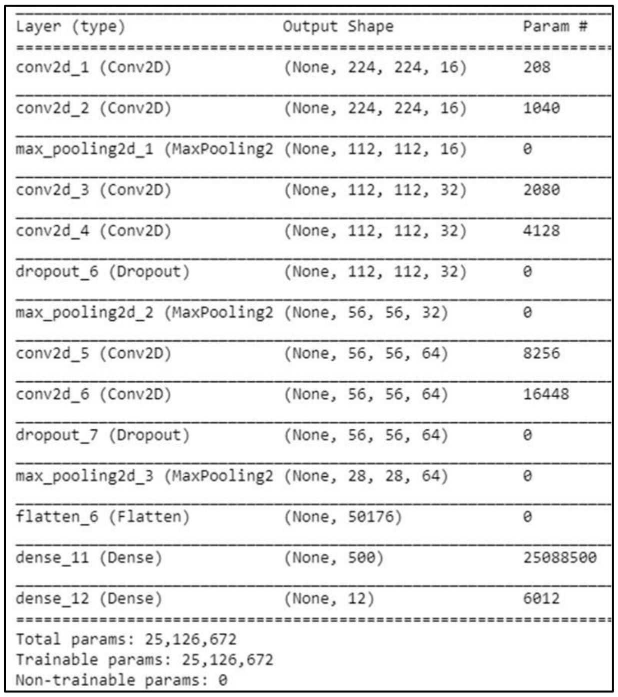
The manual steps of the image pre-processing steps for the benchmark model include the following. Images are loaded as PIL type images. Then they are resized to 224x224x3 and then converted into 4D tensors so that they can be used by the Keras software. Then, they are re-scaled so that all pixel values reside between 0 and 1. This is achieved by dividing each image pixel by the integer value 255, the largest possible pixel value for all images.

For the transfer learning-based with the built-in VGG19 models in Keras, the Keras function *preprocess_input()* handles some of the pre-processing. Before using it, however, the images were loaded as 224x224x3 PIL images and converted to 4D tensors, just as for the benchmark model. The *preprocess_input()* function then re-scales these tensors so pixel values are between 0 and 1.0. It also swaps the order of the green and blue color channels, since that is required to use the pre-trained VGG19 model.

## Implementation

After the image data were split into the three aforementioned sets and processed, the benchmark model and the VGG19-based transfer-learning CNN were implemented in Keras. Since this project utilizes deep convolutional neural networks, GPUs were utilized for their computational power. This project was implemented on a Deep Learning Amazon Machine Image (AMI) hosted by Amazon Web Services (AWS). The AMI comes pre-loaded with common tools for deep learning, including Keras, the conda environment, and several popular python libraries. This project was completed using the pre-installed environment called *tensorflow_p36*, which comes equipped with python 3.6 and tensorflow.

The benchmark model was implemented first, since it is meant to serve as a basis for comparison. The F1-score for the optimized benchmark model then becomes the metric that the VGG19-based transfer learning CNN is required to exceed. This benchmark model is a 14-layer convolutional neural network (**Figure 7**). It is implemented using the 'Sequential' function in Keras. This CNN consists of three blocks of 2D convolutional layers (two layers per block), followed by 2D max-pooling, a dropout layer (second and third blocks only), and two densely connected layers. The convolutional layers each use a stride of one, 'same' padding, and a 'Relu' activation function. The two convolutions in the first block each have filter size of 16x16, the two convolutions in the second block have filter size 32x32, and the two convolutions in the second block have filter size 64x64. The dropout layers in the second and third convolutional blocks use a 10% dropout rate. These dropout layers are meant to prevent the model from overfitting. The 2D max-pooling steps each have size 2x2. The densely connected portion of the model consists first of a flattening layer. Then, there is a densely connected layer utilizing 500 neurons, followed by a second densely connected layer having 12 neurons, one for each of the 12 plant seedlings. This output layer uses a *softmax* activation function. This benchmark model uses the ‘Adam’ optimizer for updating the network weights with “categorical cross entropy” for the loss function. This model is implemented in a user-defined function called *create_benchmark_CNN*, which is included in the code repository for this project. The tune-able model parameters are dropout rates and number of neurons in the first densely connected layer.



```
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_1 (Conv2D)            (None, 224, 224, 16)      208       
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 224, 224, 16)      1040      
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 112, 112, 16)      0         
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 112, 112, 32)      2080      
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 112, 112, 32)      4128      
_________________________________________________________________
dropout_6 (Dropout)          (None, 112, 112, 32)      0         
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 56, 56, 32)        0         
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 56, 56, 64)        8256      
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 56, 56, 64)        16448     
_________________________________________________________________
dropout_7 (Dropout)          (None, 56, 56, 64)        0         
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 28, 28, 64)        0         
_________________________________________________________________
flatten_6 (Flatten)          (None, 50176)             0         
_________________________________________________________________
dense_11 (Dense)             (None, 500)               25088500  
_________________________________________________________________
dense_12 (Dense)             (None, 12)                6012      
=================================================================
Total params: 25,126,672
Trainable params: 25,126,672
Non-trainable params: 0
```

**Figure 7: Detailed architecture of the benchmark.**

This benchmark model is trained with 15 epochs and a batch size of 20. The training image set is used for learning features, and the validation set is used at the end of each epoch to provide a more accurate estimate of the loss function. The learning curve in **Figure 8** reveals that the loss function does not change much after about 10 epochs, so 15 epochs is sufficient for training. Throughout the training process, the weights of the model achieving the lowest loss on the validation set are saved.

The benchmark network is trained with several combinations of the model parameters, including dropout rates, number of neurons, max-pooling size, and convolution filter sizes. The models having the following values achieved the best loss scores: convolutional filter sizes of 16, 32, 64 for the convolutions in blocks 1, 2, and 3, respectively; kernel size = 2x2; stride = 1; padding = ‘same’; activation function for convolutions = ‘relu’; and max-pooling size = 2x2. The function *create_benchmark_CNN()* was created with fixed values for those model parameters. Consequently, model development focused on values for drop-out rates and the number of neurons in the densely connected layers (**Figure 1**). The model weights that achieved the absolutely lowest loss score were recorded for re-use. These weights were then used to make predictions with the testing data. As previously mentioned, this model with “best” weights achieves an F1-score of 0.75 with the testing data.

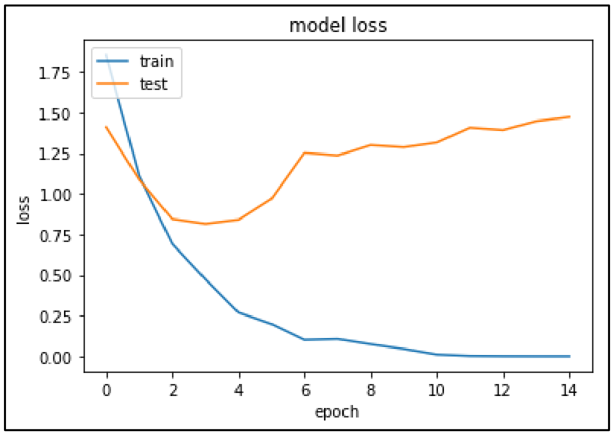


**Figure 8: The validation loss for the benchmark model changes very little after about 10 epochs.**

| Rank | Model Parameters & Values | Validation Loss | Validation F1-Score |
|---|---|---|---|
| 1 | (0.10, 0.20, 250) | 0.815 | 0.71 |
| 2 | (0.20, 0.20, 500) | 0.854 | 0.72 |
| 3 | (0.20, 0.10, 500) | 0.877 | 0.73 |

**Table 1: Validation losses and F1-scores for benchmark model. The model parameters listed in the 3-tuple include dropout rate of Block 2, dropout rate of Block 3, and number of neurons in the densely connected layers.**

The two-stage process for implementing the VGG19-based transfer learning CNN is described here. The first stage is called "tuning" and the second stage, which builds upon the results of the first stage, is called "fine tuning". To utilize transfer learning with the pre-trained VGG19 network, the *VGG19()* function from the "keras.applications" module is utilized.

In the "tuning" stage, the *VGG19()* function is called with its first argument, "include_top", set to "False". This means that the fully connected layers—that is, layers 23, 24, and 25—of the model are excluded. This is important, because the weights of those last three layers are highly adapted to the 1,000 classes of the ImageNet database. After this discard, the weights of these layers are frozen. A small "top-model" is then added to the pre-trained VGG19 base (**Figure 2**).

The top-model consists of four layers: (1) flatten layer, (2) a densely connected layer with 'relu' activation, (3) a dropout layer and (4) a second densely connected layer consisting of 12 outputs with 'softmax' activation. The aggregated network (**Figure 9**), consisting of the modified VGG19-base and the top-model, is compiled with the adaptive 'RMS Prop' optimizer so that the weights of the top-model can be updated from their random values relatively quickly. Recall that the weights of the VGG19 base are frozen during this step. The only tune-able parameters for this aggregated top-model include (1) the number of neurons in the first densely connected layer, (2) the dropout rate, and (3) the learning rate for the RMS Prop optimizer (**Table 2**).

This first aggregated model is developed using the training and validating sets and the parameter values listed in **Table 3**. Just as with the benchmark model, the "tuned" model is developed using a batch size of 20 and 15 epochs. The batch size and number of epochs is kept the same for all models developed in this project in order to achieve fair comparisons. Just as before, it is also clear from the learning curves for the loss function (**Figure 10**) that 15 epochs are sufficient for training the model. During each epoch, features are learned from the training set; at the end of each epoch, the validation set is used to generate a loss score. The algorithm saves the weights that lead to the lowest loss among all 15 epochs.

| **Model Parameter** | **Values** |
|---|---|
| Num. VGG19 Layers to Keep | 4,7,12,17 |
| Num. Dense Neurons (Top Model) | 64, 128, 250 |
| Dropout Rate (Top Model) | 0.10, 0.20 |
| Optimizer | SGD, RMS Prop |
| Learning Rate | 0.0001, 0.0005 |

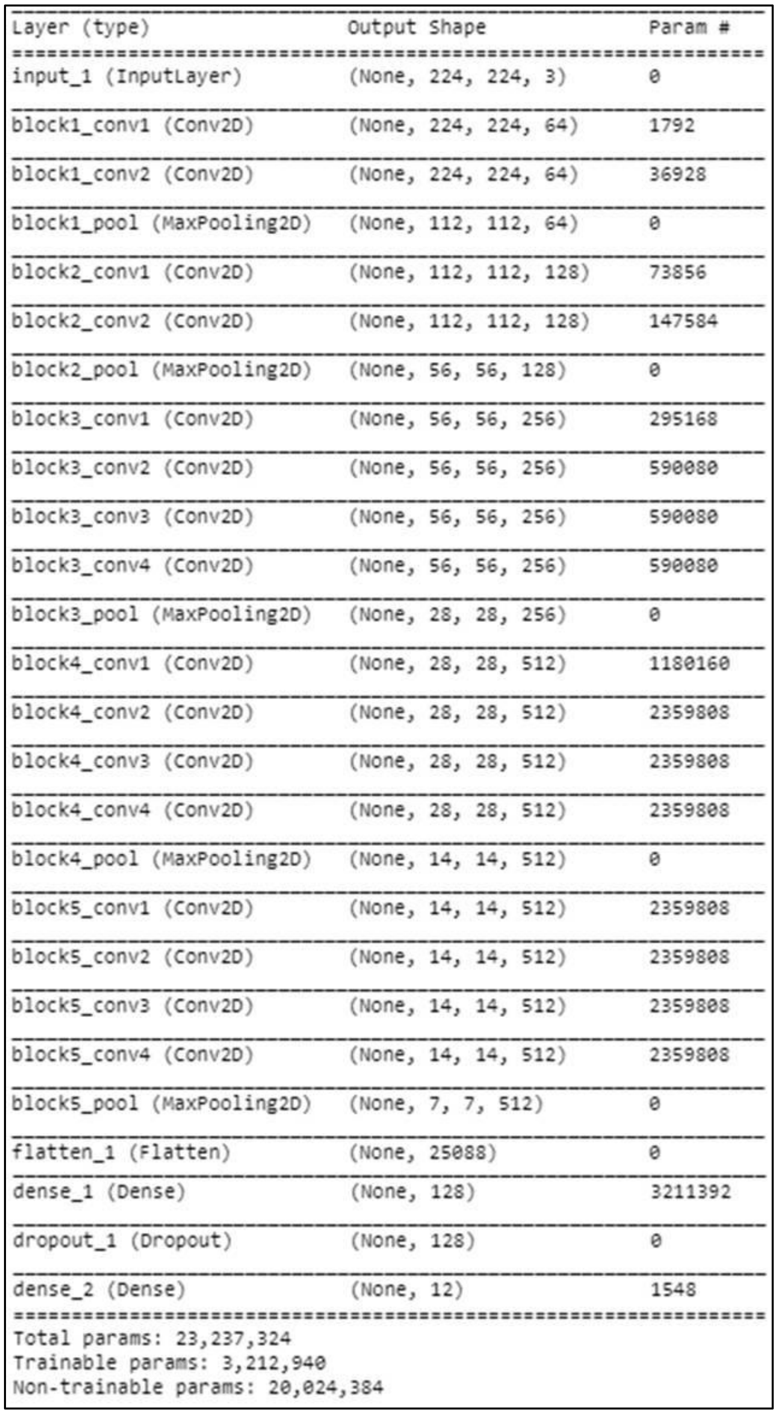
**Table 2: Tuneable model parameters for the aggregated convolutional neural network presented in this work.**

| Rank | Model Parameters & Values | Validation Loss | Validation F1-Score |
|---|---|---|---|
| 1 | (22, 64, 0.20, 0.0001) | 1.213 | 0.74 |
| 2 | (22, 128, 0.10, 0.0001) | 1.244 | 0.73 |
| 3 | (22, 128, 0.20, 0.0001) | 1.270 | 0.69 |

**Table 3: Three best models after the "tuning" stage for the aggregated model. The 4-tuple of model parameters is the following 1. Number of bottom-most VGG19 layers to freeze. 2. Number of dense neurons in the top-model, 3. Dropout rate of top-model, 4. Learning rate for RMS Prop.**

The "fine-tuning" stage utilizes the result of the previous stage. In this stage, the weights of the top-model and the weights from one or more contiguous convolutional blocks at the top of VGG19 base (i.e., Blocks C, D & E) are adjustable, while the weights of convolutional blocks near the bottom of the VGG19 base (i.e. blocks A & B) are fixed. This leads to the notion of another parameter to optimize: "number of initial VGG19 layers to freeze". For this particular project, the "number of VGG19 layers to freeze" were 4, 7, 12, and 17. Note: these frozen layers always reside at the bottom of the VGG19 base. For example, if the value for this parameter is 12, then only the first 12 layers of the VGG19 base are frozen.

Just as before, the tuned, aggregated model is trained and validated using 15 epochs and a batch size of 20. During this training, the parameter called "number of initial VGG19 layers to freeze" is again varied between 4, 7, 12, and 17. The algorithm saves the model weights whenever the validation loss achieves a new minimum. Finally, the model that achieved the lowest validation loss score is used to make predictions with the testing set. The F1-score from this prediction serves as the final evaluation of the method.



```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 224, 224, 3)       0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv4 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv4 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv4 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 25088)             0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               3211392
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_2 (Dense)              (None, 12)                1548
=================================================================
Total params: 23,237,324
Trainable params: 3,212,940
Non-trainable params: 20,024,384
```

**Figure 9: Architecture of the transfer learning model.**

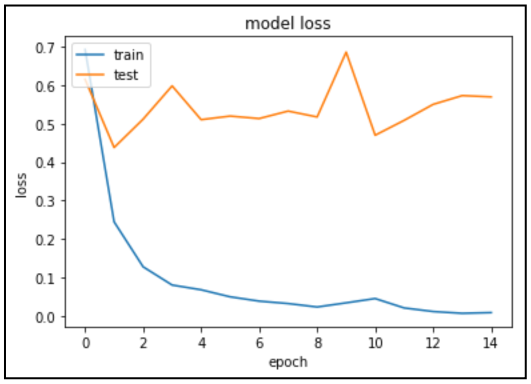


**Figure 10: The learning curves for the loss function of the solution model indicate that 15 epochs is sufficient.**

As previously mentioned, the original plan for this transfer learning approach did not work. Attempts to completely remove convolutional blocks at the top of the VGG19 model were fruitless and led to very high validation losses. The resultant network was as useless as random guessing. The *layers.pop()* function intended for removing layers from the CNN did not work properly, as verified by inspecting the dimensions of the weights of the resultant model's topmost layer. Other workarounds were developed, such as creating a 'Sequential' network and manually copying the layers from the pre-trained VGG19 network. The networks generated by these workarounds also yielded very high validation losses.

| Rank | Model Parameters & Values | Validation Loss | Validation F1-Score |
|---|---|---|---|
| 1 | (17, 128, 0.20, 10e-5) | 13.21 | 0.18 |
| 2 | (17, 128, 0.20, 10e-6) | 13.45 | 0.16 |
| 3 | (17, 128, 0.20, 10e-4) | 14.03 | 0.13 |

**Table 4: Three best models after the "fine-tuning" stage for the aggregated model. The 4-tuple of parameters represents the following 1. Learning rate for SGD, 2. Number of dense neurons in the top-model, 3. Dropout rate of top-model, 4. Number of bottom-most VGG19 layers to freeze.**

The implementation of the F1-score was somewhat straightforward. The F1-score is calculated via the *f1_score* function in the sklearn.metrics module. Utilizing it during training involves defining a Metrics class that is fed to the 'callbacks' parameter to the model's *fit()* function. The class definition was borrowed from a web blog on Medium. The Metrics class uses two functions, *on_train_begin()* which is called at the beginning of the training process, and *on_epoch_end()* which is called at the end of each epoch. At the beginning of training, *on_train_begin()* initializes a list for storing any desired metrics that will be calculated at the end of each epoch. For this problem, a single array is initialized to store F1-scores. At the end of each epoch, *on_epoch_end()* accesses the model and validation data. It then makes predictions, computes F1-scores, and stores them in the array that was generated by *on_train_begin()*.

While most of the code developed for this project was recycled from a previous project, two minor but impactful coding challenges occurred. These coding challenges were actually coding "bugs", but the impact of these bugs was disheartening, almost scary, when first discovered. The first complication was in accessing and visualizing the model histories (**Figure 8** and **Figure 10**) that were recorded for the 40 different tuning and fine-tuning stages of the transfer learning model. I did not realize that during these two specific stages of model development, I made the mistake of saving only the "History.history" attribute (i.e. a dictionary) containing the training and validation metrics rather than the full History object that is returned by *model.fit()*. This was essentially a typo in my code. Consequently, my function *plot_model_acc_loss()* that I had been using for visualizing training histories of the benchmark model, did not work! This was particularly troubling and disheartening because, as previously mentioned, training these models took 40+ hours for all 40 models! Would I have to re-run them after fixing my bug?! I hoped not! Rather than re-running the models I sought a solution to my predicament, namely determining if the files that were saved actually contained the history data. The reason that *plot_model_acc_loss()* did not work and produced a scare was that it accepts a History object as its input argument. It then accesses the 'history' attribute to generate graphs of the training and validation losses. Fortunately, I realized this after reading the error message and by printing the data to the console. Printing the data revealed that the structure was a dictionary with the following keys: 'loss', 'val_loss', 'acc', 'val_acc'. A simple solution was to write a new function, which is currently called '*plot_saved_model_acc_loss()*' in the code repository, that accepts this 'history' dictionary as its only input and generates a plot of the training and validation history losses. In retrospect, my naming convention for these data files contributed to the confusion.

The second challenge also sparked frightful confusion after training these models for 40+ hours. This challenge was also a mental mistake: I utilized the wrong version of the test data set to make final predictions with the solution model. I incorrectly used the data set called 'test_tensors', which was processed exclusively for the benchmark model. The data set I should have used with the solution model is called 'testData'. The 'test_tensors' data do not have the green and blue color channels swapped; whereas, 'testData' do have those two color channels swapped since that is required for the VGG19 transfer learning based solution. Consequently, the solution model, when utilizing the incorrect 'test_tensors' data, yielded an F1-score of 0.10. This score is substantially lower than the F1-scores of 0.82+ produced with the validation set. I discovered the solution to this mental bug after verifying that the model weights were correct (by re-training the model and reproducing an abysmal F1-score of about 0.10) and subsequently reviewing the code. A better naming convention for the two separate test data sets would have helped to avoid this problem in the first place.

## Refinement

As mentioned in the section entitled "Implementation", several parameter values were tested to obtain the model achieving the lowest validation loss. The initial implementation of this transfer learning based approach utilized the following parameter values: RMS Prop learning rate of 0.0001, 128 neurons for the first dense layer of the top-model, a dropout rate of 0.20, and kept the weights of all 22 layers of the VGG19 base frozen. The best "tuned" model achieved a validation loss of 1.213 and a validation F1-score of 0.74 ().

Listed in **Table *5*** are the parameters used for training the transfer learning based CNN:

| Model Parameter | Values Tested |
|---|---|
| Number of VGG19 Layers to Freeze | 4,7,12,17,22 |
| Number of Dense Neurons (Top Model) | 64, 128 |
| Dropout Rate (Top Model) | 0.10, 0.20 |
| Learning Rate for RMS Prop (Top Model) | 0.0001, 0.0005 |

**Table 5: Model parameters and values used in developing the aggregated VGG19-based "transfer learning" style convolutional network in this work.**

This combination of parameter values means that 40 different models were trained. With each model taking approximately one to two hours (or more) to train, it is easy to see that model development is rather time consuming.

After testing all 40 combinations of parameters, it was determined that the best model (**Table 6**) requires keeping the first 12 layers of VGG19 frozen, uses 128 neurons in the top-model, a dropout rate of 0.10, and a learning rate of 0.0005 with the RMS Prop optimizer. This best model achieves a validation loss of 0.438 and a validation F1-score of 0.86.

| Rank | Model Parameter Values | Validation Loss | Validation F1-Score |
|---|---|---|---|
| 1 | (12, 128, 0.10, 0.0005) | 0.438 | 0.86 |
| 2 | (4, 128, 0.10, 0.0005) | 0.446 | 0.90 |
| 3 | (7, 128, 0.20, 0.0005) | 0.451 | 0.87 |

**Table 6: The 4-tuple of model parameters represents: 1. Number of bottom-most VGG19 layers to freeze. 2. number of dense neurons of top-model, 3. Dropout rate of top-model, 4. Learning rate for Stochastic Gradient Descent.**

## IV. Results

### Model Evaluation and Validation

The best model is defined as the one that achieves the lowest validation loss during training. The reason is for this selection criterion is that the optimizer operates directly on the validation loss function. There is no way to optimize the F1-score in a way that directly relates to the model parameters. Once the model with the lowest validation loss is determined, it is used to make predictions with the held-out test set. In this work, the lowest validation loss is 0.438 (**Table 6**). The F1-score that this model achieves on the test set is 0.85.

The various values of hyper-parameters that were used in model development are presented here. The best model utilizes a learning rate of 0.0005 for the RMS Prop optimizer, 128 neurons in the top-model, a dropout rate of 0.10, and keeps the first 12 layers of the VGG19 base frozen. These values are reasonable. Dropout rates smaller than 0.50 are common in deep learning models. Freezing the bottom 12 layers of the VGG19 base is reasonable, because, as mentioned previously, these layers of the pre-trained VGG19 model should already have the ability to identify generic features (i.e., basic shapes) common to most image objects. The remaining layers would then increasingly specialize on features for the plant seedlings.

The F1-score that the best model achieves on the testing set is comparable to the best value that it achieved on the validation set. This indicates reliability in the model, since it is able to achieve similar results with an entirely different set of input data. A more robust measure of this model's stability could be based on training the model with 30+ different training and testing splits. The use of these 30+ training-testing splits would account for model variance. The mean value of these evaluation metrics could be recorded and saved as one measurement. This process would be repeated across 30+ trials to account for the stochastic nature inherent to deep learning models. These 30+ mean values would then be averaged to yield the final evaluation metric. This approach would provide an estimate of the F1-score as well as variability in this estimate. Unfortunately, the computation time (approximately 900+ hours) and corresponding financial cost for training deep convolutional neural networks make this approach extremely undesirable and prohibitive. Fortunately, the use of the training, validation, and held-out testing sets is considered acceptable practice for evaluating models.

## Justification

The transfer-learning based model definitely outperforms the benchmark in its ability to classify images of the 12 plant seedlings in this problem. The F1-score of the solution model is 0.85, which is larger than the F1-score of 0.75 that is achieved by the benchmark. Since the best possible F1-score is 1.0 and the worst possible score is 0.0, it is clear that the solution model outperforms the benchmark.

The solution model presented here is significant enough to have solved the problem reasonably well. An F1-score of 0.85 indicates that the model performs better than random guessing, and in fact that it identifies the plant seedlings fairly well. This result is not surprising given that the solution model here is based on VGG19, a famous image classifier.

This solution model demonstrates the effectiveness of transfer learning. As already mentioned, the solution developed here is based on a pre-trained VGG19 network. The VGG19 network contains a series of blocks, each of which contains several consecutive 2D convolutions and a 2D max-pooling layer, followed by fully connected layers and a softmax activation layer. Since this network earned 2$^{nd}$ place in the Image Net Large Scale Visual Recognition Challenge (ILSVRC) in 2014, I anticipated that it would achieve good results on this problem. Fortunately, that is the case. The top-model is an effective, albeit, required component of the solution. Its primary purpose is to produce 12 outputs, one for each of the 12 plant seedlings classes. It is possible that the dropout layer reduces the degree to which the CNN overfits the data; although, a comparison study would be necessary to evaluate this hypothesis.

## V. Conclusion

### Free-Form Visualization

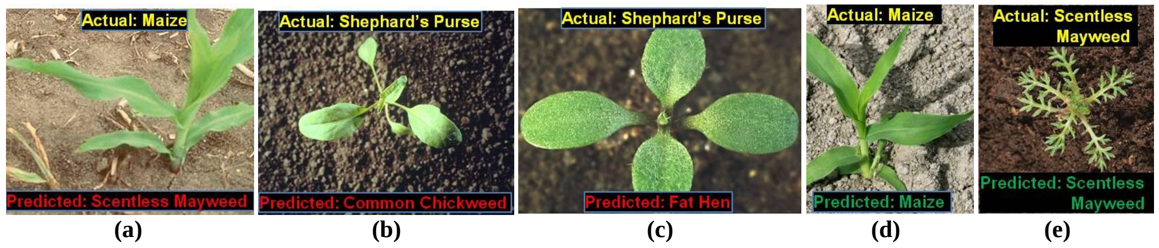


**Figure 11: The solution model was used to make classifications with 5 images of plant seedlings taken from the internet. It was correct twice.**

The solution model was used to make predictions with five images of plant seedlings taken from the internet. It was correct two times (**Figure 11d&e**). While two out of five correct predictions is not great, the sample size is small and therefore the rate of success is not worth being concerned about. It would be great to make predictions with several hundred or thousand new images, but acquiring such a high volume of images is not feasible for me at this time.

The model incorrectly classified Maize as Scentless Mayweed (**Figure 11a**). This misclassification is confusing, since those two plant seedling types generally do not resemble each other. Furthermore, the two Maize seedlings in **Figure 11a & d** resemble each other quite well, so the fact that a misclassification occurred once rather than twice or not at all is surprising. Perhaps additional image processing, such as background removal via green-channel filtering would improve the results of the classifier. The misclassification of the Shephard's Purse as Common Chickweed in **Figure 11b** is definitely understandable—the plant seedling in that figure looks remarkably similar to Common Chickweed! The same is true of **Figure 11b**—the

Shepherd's Purse in the figure looks very similar to some Fat Hen seedlings in the training data. In general, no classifier will make perfect predictions, especially when there are classes of image objects that are strikingly similar, as seen here.

## Reflection

The solution presented here takes advantage of transfer learning, wherein a previously trained model was adapted to the data given for this project. Some data processing steps were required. First, outliers were identified by manually seeking images that were too blurry to be useable or that did not contain plant seedlings. The remaining images were then converted to 4D tensors, scaled to have pixel values residing between 0 and 1, and resized to have the same dimensions (224x224). The ordering of the last two color channels, green and blue, were interchanged to blue and green. This pre-processing is typical for image data but also required for use with the VGG19 functions in Keras. The pre-processed images were then split into training, validation, and testing sets in such a way that the class distribution of the three resulting data sets resembled that of the full data set.

The transfer learning approach is based upon using the VGG19 network architecture. This network architecture and pre-computed weights, obtained by training on the ImageNet database, are loaded via the aforementioned *VGG19()* function. The network is loaded without the top three layers, since the weights of those layers are specific to the 1,000 output categories of the ImageNet data set. This is an important step, since in this project there are only 12 classes of plant seedlings, none of which are included in the ImageNet classes. In place of these three layers, a multilayer top-model is used. This top-model has two fully connected layers with a single dropout layer placed in between. The dropout layer is intended to reduce overfitting. The output layer produces 12 output classes, one for each of the 12 plant seedlings. This combination of VGG19 base and top-model constitute a new "aggregated model".

The aggregated model is developed in two stages. In the first stage, the model is tuned. Tuning the model amounts to training it in the following way: the weights of the VGG19 layers are held constant but the initial, random-valued weights of the top-model are updated during training. The training and validation image data are used for the tuning stage. In the second stage, the tuned model is fine-tuned. Just as before, the weights of the "tuned" top-model are allowed to be updated; additionally, the weights of one or more convolutional blocks near the top of the VGG19 network are allowed to be updated. The weights of the rest of the VGG19 base network, near the bottom of the architecture, are held constant. A gradient descent optimizer is utilized with a small learning rate so that the weights of the adjustable portions of the VGG19 network can be gradually updated, or finely tuned, to connect the fixed weights of the bottom portion of the model with those of the top-model. Just as before, the training and validation image data are used for fine-tuning.

The fine-tuning step is repeated several times for a variety of model parameters. These parameters include the chosen number of convolutional blocks that are held fixed, the number of neurons to use in the fully connected top-model, the learning rate of the optimizer, and the dropout rate of the top-model. The weights of the model achieving the lowest validation loss are recorded across 15 epochs. This tuning and fine-tuning process is performed for all 40 combinations of model parameters. The model achieving the lowest validation loss is considered the best model and subsequently used to make predictions with the testing set. The F1-score is then calculated and considered to be the final evaluation metric. The best model that was developed in this project achieved an F1-score of 0.85. This score is good enough to consider the model an effective and sufficient solution to the plant seedlings image classification problem, though there are ways to improve this model.

This model performed reasonably well, as expected. It is well-known that transfer learning based neural networks are effective and can be used to build a model relatively quickly. Despite the success demonstrated in this project, there certainly was an interesting challenge to address. The original plan for developing the model had to be abandoned, since its implementation led to poor results. As previously mentioned, discarding additional layers of the VGG19 network using the *layers.pop()* function seemed to severely compromise the performance of the resulting network. A couple of workarounds were attempted, including copying layers of the VGG19 network into a Sequential model, and writing a function that was supposed to achieve what the *layers.pop()* function was meant to do. Neither of those workarounds was effective. Fortunately, the solution presented in this paper worked well.

## Improvement

While the solution model created in this project is effective, the use of certain approaches could improve the results. First, instead of using the first 22 layers of the pre-trained VGG19 CNN, it is theoretically possible to remove additional layers before adding the top-model to it. Attempts to achieve this here were unsuccessful, but doing so could potentially yield lower validation losses and higher F1-scores. None of the solutions posted online by other programmers were effective when implanted here. A benefit of this modification would be reduced training time because the network would be smaller. A second area that has the potential to improve the model is an increased hyper-parameter space. The development of the solution model involved tuning only four parameters. It would be possible to tune additional parameters, such as expanding the choices for model optimizer. Only stochastic gradient descent and the RMS Props optimizers were employed in this work. The use of Adagrad, Adadelta, or Adam would yield additional models to compare with the solution model. Third, a method that was not used but that would probably yield better results is image augmentation via generators. Utilizing image generators to artificially expand the training data—via image rotations, translations, reflections, and shearing—could lead to a better predictive model. The model would have more image examples to learn from; the downside to this, however, is the increased computational and financial costs.

A statistical technique that was not used here, but that would be helpful for making a robust assessment of the model, was mentioned earlier in this work. That technique is to use 30+ random training-testing splits for training across 30+ trials to generate sets of validation losses and F1-scores. As previously mentioned, this approach would utilize about 900+ hours and a significant financial cost. However, in principle it would be an excellent way of accounting for model variance associated with train-test splits and the stochastic variation inherent to the optimization processes of deep learning model development.

It is possible to generate an ensemble of transfer-learning based CNNs. For example, the final model could be comprised of the three models listed in **Table 6**, where a voting system is implemented to determine the final prediction for each input image. Alternatively, an ensemble of models generated via transfer-learning with other pre-trained networks would be possible. This could include a variety of transfer-learning based models based on VGG19, ResNet50, etc. There are many possibilities, but such an ensemble may be less sensitive to variations in the input data when compared to the solution model presented here. The downside is the increased time and financial cost necessary to train each additional transfer-learning based model.