Steve Greenberg - Capstone Report

Machine Learning Nanodegree - December 29, 2017

Definition

Project Overview

My Capstone Project classifies twelve different species of plants by image analysis. This project utilizes the dataset from Kaggle's Plant Seedlings Identification competition.

There is a long history of using quantitative analysis to differentiate between species of plants. Wikipedia highlights the Iris Flower Dataset as one of the classic data sets used to teach statistics. More recently, a Flowers dataset has been used to introduce students to Transfer Learning on TensorFlow.

Classifying plant species present an obvious benefit and challenge. Similarly looking plants can sometimes require dramatically different treatments. Harmful, invasive species should be removed. Beneficial species should be cultivated and supported. Humans are only able to differentiate between the two if they have uncommon domain expertise.

Aarhus University in Denmark has made this dataset available for the public. They write:

> *A database of images of approximately 960 unique plants belonging to 12 species at several growth stages is made publicly available. It comprises annotated RGB images with a physical resolution of roughly 10 pixels per mm. To standardise the evaluation of classification results obtained with the database, a benchmark based on f1 scores is proposed.*

Their motivation is to "*provide researchers a foundation for training weed recognition algorithms*".

More details are provided in the authoritative link to the data - https://arxiv.org/abs/1711.05458

Kaggle is re-hosting the dataset as a competition to give it greater exposure.

Problem Statement

The problem is to classify an image of a seedling into one of twelve categories. Figure 1 shows class labels and an example image for each class.

*Figure 1 - Images from **https://vision.eng.au.dk/plant-seedlings-dataset/***

A solution to this problem is a model trained to predict a species for any image.

- **Input**: Input to the model is an arbitrary image.
- **Output**: Output from the model is a predicted label.

I have trained a Convolutional Neural Network to classify seedling images. I considered both CNNs built from scratch and those built using Transfer Learning.

Metrics

F1 Score

Plan Seedlings Identification is a Classification Problem, and so the optimal metric will be one that rates a model's ability to correctly classify pictures across the 12 classes. An initial approach might be to simply measure the percentage of pictures that are correctly classified. This is the model's "accuracy". However, when classes are imbalanced, accuracy can be misleading. (It's easy to have high accuracy in a medical test for a rare condition by never predicting the condition to occur.)

As will become clear in the Data Exploration section, classes are imbalanced in the Plant Seedlings Identification dataset.

F1 Score presents an optimal alternative that works well when classes are imbalanced. F1 Score ensures both that

- images selected to be of a certain class are relevant (aka. Precision)
- relevant images are selected (aka Recall)

There are a few different ways to calculate a **multi-class** F1 Score. (See the discussion of the "average" parameter in scikit-learn's documentation for F1. Kaggle evaluates the **micro-averaged F1 score** as described in the competition's evaluation criteria.

In a micro-averaged F1 Score, precision and recall are evaluated once across all classes. These are each summed across each class (k) in the total set of classes (C). So here are the formulas for Precision and Recall. (TP means True Positives, FP means False Positives and FN means False Negatives.)

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

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

The formula for the F1 score is:

$$\frac{2 \times (precision \times recall)}{(precision + recall)}$$

Operationally, the model's success is quantified by running 795 unlabeled images through the model, combining the predictions into a single CSV and submitting this CSV to Kaggle for evaluation.

Loss Function: Categorical Cross Entropy

In addition to an overall metric for model quality, we need a loss function to use during training. Like the overall metric, the loss function should be chosen to work with a multi-class classification problem. But unlike the overall metric, the loss function should be differentiable and provide direction for optimizing weights. Additionally, unlike the overall metric, the loss function can make use of the probabilities assigned to each class's membership rather than the binary choice of which class is most likely.

I've chosen to use Categorical Cross Entropy for my training.

A low value for Categorical Cross Entropy means that our model has a high probability of predicting target values that align with reality.

The formula for Categorical Cross Entropy is:

$$-\sum_{i=1}^{n}\sum_{j=1}^{m} yij\ ln\ (pij)$$

where …

- **n** is the number of training datapoints
- **m** is the number of classes,
- **yij** is 1 if the **ith** datapoint actually belongs to the **jth** class and 0 otherwise
- **pij** is the probability our model assigns to the jth datapoint belonging to the ith class.

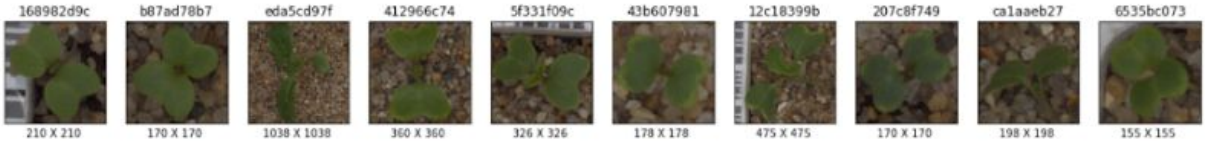
Analysis

Data Exploration

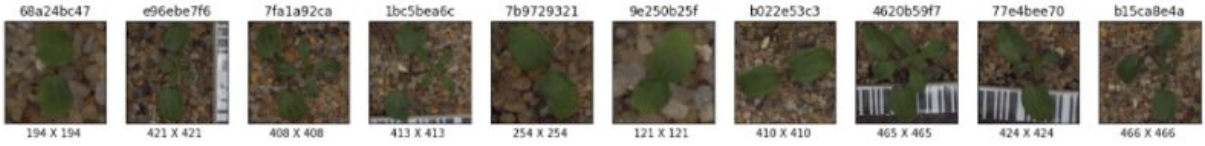
Here are 10 random samples draw from each of the twelve classes in the dataset:

Black-grass

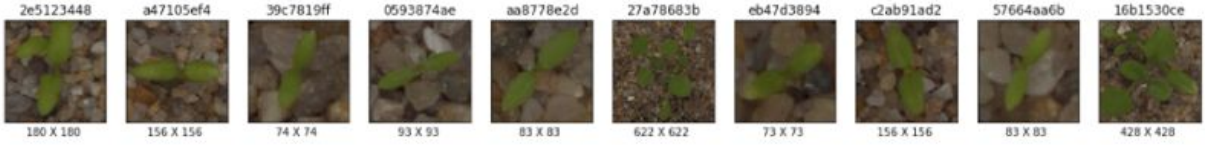
Charlock



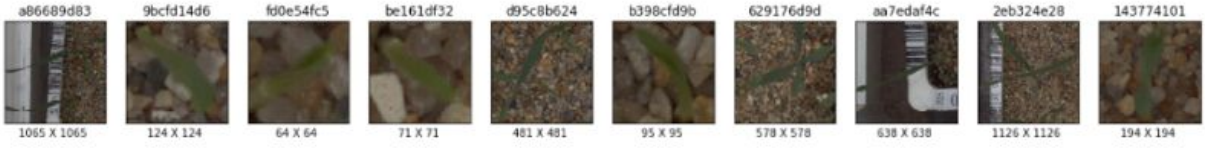
Cleavers



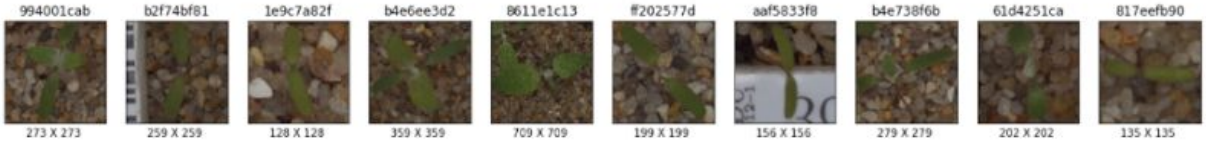
Common Chickweed



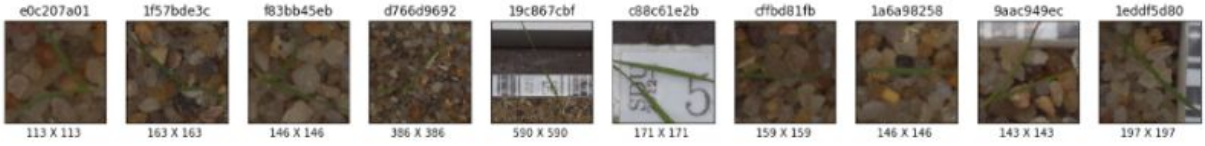
Common wheat



Fat Hen



Loose Silky-bent



Maize

Scentless Mayweed

Shepherds Purse

Figure 2 - 10 Sample Images for each class

Distinguishing Characteristics of Seedlings

A first glance from the human eye reveals some patterns that should be useful for a model to distinguish between these seedling species:

- Charlock often has clumps of four leaves in a "butterfly" pattern.
- Shepherds Purse often has four leaves - separated by thin stems.
- Cleavers often has two fat dark green leaves.
- Common Chickweed has two thin light green leaves.

Nonetheless, this is not a trivial problem. These species are hard to tell apart!

Reference Scales

Many, but not all of the images have reference scales in them. These allow the viewer of the image to understand the actual physical size of the objects in the image. The reference scales look like barcodes - stark black and white stripes in a horizontal or vertical pattern. It's important to acknowledge that these may affect our model.

In particular, Black Grass is often quite small and thin. So the reference scales dominate the images of Black Grass. Our model may identify the reference scale as a distinguishing feature of Black Grass and succeed even with test data as a result. However, such a model will perform poorly when brought out into the wild.

Image Size

The images are not all the same size. Some small images are only 73x73 px, and some large ones are over 1000 px square. Furthermore, the size is meaningful. The dataset notes above specify that every px is roughly 1/10th of a mm.

In order to use a simple Convolutional Neural Network, images are rescaled to a common size. I chose 224 X 224 px. Information - specifically the actual physical size of the seedling - is lost during this rescaling. Although the Reference Scales could continue to provide this information to the network, they are not visible in every picture.

Exploratory Visualization - Class Balance

After loading the labeled data into python and splitting it into test and training sets, I examined the distribution of classes:

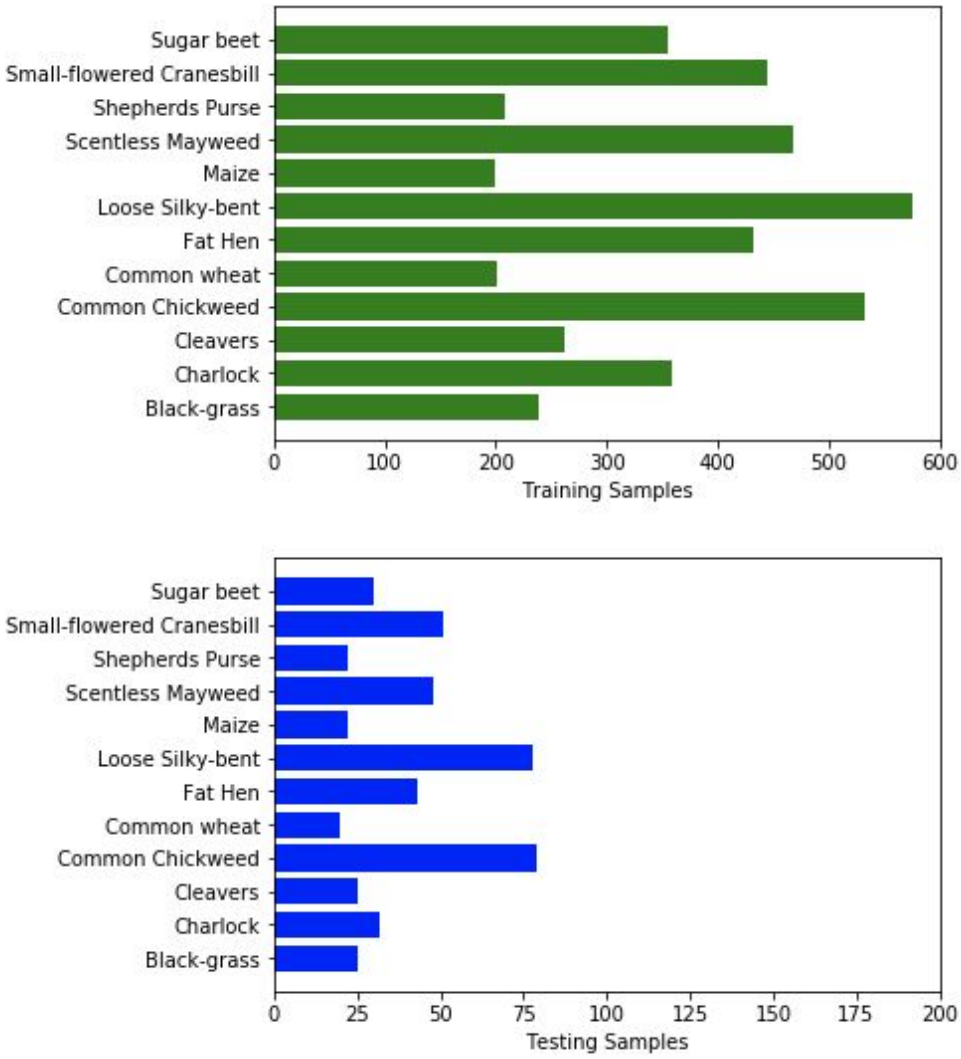


Figure 3 - Number of images in each class in training and test sets.

There is some imbalance of classes here. At the extremes, there are 611 images of Common Chickweed and 221 of Common Wheat and Maize.

After reading through [8 Tactics to Combat Imbalanced Classes in your Machine Learning Dataset] by Dr. Jason Brownlee, I decided to rebalance the classes through use of oversampling to ensure an even distribution.

Algorithms and Techniques

I used three approaches approaches to create a Convolutional Neural Network to classify images:

- Building a CNN from scratch.

- Transfer Learning on the VGG19 network.
- Transfer Learning on the InceptionV3 network.

Common CNN Characteristics

Each of these three approaches shares some common characteristics:

- Each consists of a series of **convolutional layers** - in which n x n matrices of weights (called filters) are trained in sliding windows across the entire image. These convolutional layers are initialized with random weights. They end up discovering latent features in the image. Convolutional layers early in the network discover abstract features - like colors and edges. Later layers learn shapes like ovals and lines. The final convolutional layers discover actual features in the dataset - like stems or leaves.
- In order to prevent an explosion in the number of trainable parameters, convolutional layers are broken up periodically by **pooling layers**.
- The end of our neural network is always a **12-node fully-connected layer with a softmax activation function**. Each of the 12 nodes represents the probability that a particular input image belongs to a certain output class.

Building a CNN from scratch

I've chosen to use Keras to build a simple Convolutional Neural Network. This CNN worked well with the dog identification project.

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_7 (Conv2D)            (None, 223, 223, 16)      208
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 111, 111, 16)      0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 110, 110, 32)      2080
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 55, 55, 32)        0
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 54, 54, 64)        8256
_________________________________________________________________
max_pooling2d_9 (MaxPooling2 (None, 27, 27, 64)        0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 26, 26, 128)       32896
_________________________________________________________________
max_pooling2d_10 (MaxPooling (None, 13, 13, 128)       0
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 12, 12, 256)       131328
_________________________________________________________________
max_pooling2d_11 (MaxPooling (None, 6, 6, 256)         0
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 5, 5, 512)         524800
_________________________________________________________________
max_pooling2d_12 (MaxPooling (None, 2, 2, 512)         0
_________________________________________________________________
```

```
global_average_pooling2d_2 ( (None, 512)              0
_________________________________________________________________
dense_5 (Dense)              (None, 12)                6156
=================================================================
Total params: 705,724.0
Trainable params: 705,724.0
Non-trainable params: 0.0
```

Figure 4 - Architecture of CNN built-from-scratch

- The input shape is a 224 X 224 X 3 tensor representing the three color layers of a 224 X 224 px image.
- 2 X 2 filters are used throughout the network and strides is consistently set to 1.
- 6 pairs of convolutional and pooling layers are used. The height and width of the image are reduced in each subsequent layer. At the same time, the depth increases with each subsequent feature.
- A global average pooling layer and 12 node dense layer are used at the end of the network to classify the image as belonging to one of the 12 classes.

Transfer Learning on VGG19 and Inception V3

A CNN trained from scratch on the seedling data will only find features present in those images. Additionally, training will be limited to what can be achieved in a reasonable time on a single machine with a single K80 GPU. (For my purposes, I never let training last longer than 60 minutes.)

Transfer Learning, in which sophisticated networks and pre-trained weights learned through training on larger datasets and for longer periods, are used as a starting point, often produces better results.

I applied transfer learning on top of both the VGG19 and Inception V3 networks.

- The VGG19 network is quite similar to my homegrown CNN. It uses 3 X 3 convolutional windows rather than 2 X 2 windows, but it retains the 2 X 2 pooling layers. It has six more layers than the network I built.
- The Inception V3 network simultaneously produces 1 X 1, 3 X 3 and 5 X 5 convolutions. Although the number of layers in Inception V3 are daunting, it actually succeeds in having a smaller footprint in the size of its trainable weights.

More about VGG19, Inception V3 and other networks can be found in [ImageNet: VGGNet, ResNet, Inception, and Xception with Keras] by Adrian Rosebrock.

There are a number of specific Transfer learning approaches and the optimal choice depends on both the size of the new dataset and on how similar the data is to the original data used to train the network. For my use case, the **size of the new dataset is fairly small**. I have fewer

than 5000 images of seedlings. The **images in my dataset are dissimilar** from the ImageNet images used to train the Inception network.

Recommended Approach

Given this situation, the approach recommended both in the CNN lessons in the Udacity Course and in [Transfer Learning using Keras] by Prakash Jay is this:

- Remove most of the pre-trained layers from the Inception V3 / VGG19 network, retaining only the beginning of the network which detects generic aspects of images - like edges and shapes.
- Add a new fully connected layer with 12 nodes - matching the number of classes in the seedling dataset.
- Randomize the weights of the my new fully connected layer. Freeze all the weights from retained layers of the Inception V3 / VGG19 network.
- Train the network to update the weights of the new fully connected layer

Here is a visualization of this approach:

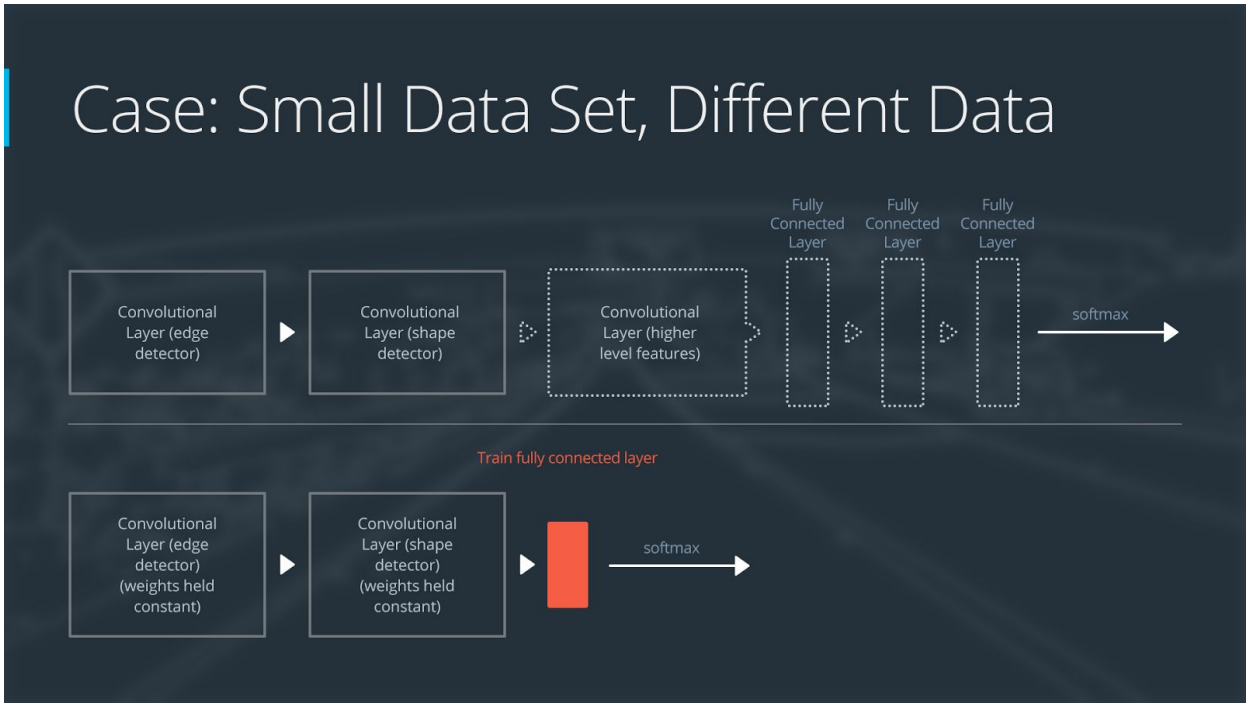


Figure 5 - [Neural Network with Small Dataset, Different Data]

Simpler Approach with Acceptable Results

During my work on this project, I found that I was able to tweak this approach and still get results that I found acceptable. I read [Building powerful image classification models using very little data] by Francois Chollet. Following the advice in the section called "Fine-tuning the top

layers of a pre-trained network", I preserved all convolutional and pooling layers of VGG19 and InceptionV3 and merely adjusted which weights were trainable within the original models.

I found that I was able to achieve good results by freezing the weights in the first third of the model and retraining those in the remaining two thirds.

Benchmarks

Results from my model were benchmarked against other submissions in the Kaggle Leaderboard for this contest.

Additionally, another Kaggler has put together an example CNN using Keras and is hosting it as a Kaggle Kernel. After I completed my solution I ran their code as a benchmark.

Finally, it's important to take a step back and ask how the model might be used for real purposes. It might be used to try to classify weeds seen in the wild. The author of the study has provided a folder with a few dozen pictures of weeds in the wild without artificial lighting. I benchmarked my model with this data.

Methodology

Note: In this methodology section, I wanted to allow the reader to quickly find the code corresponding to the my descriptions. So I refer to cells in the accompanying notebook by the Heading.

Data Preprocessing

Initial Preprocessing

Notebook Heading	**Description**
Load Data	Loads data from the labeled directory into memory. Returns arrays of filenames and target classes. Note that this doesn't actually load the images into memory.
Train Test Split	Splits data into training and testing sets.
Load Images into Tensors	Images are loaded into memory and converted into tensors. They are all resized to size (224, 224,3). Each is 224px wide by 224px high with 3 RGB color layers.

	Each pixel in the tensor is converted to a scalar value in the range [0,1] by dividing it by 255
One Hot Encode Targets	Targets are one-hot-encoded by converting them from integers to 12-item vectors

Augmenting and Oversampling

As noted above in the Data Exploration section, there is a class imbalance in the dataset.

Additionally, I wanted to augment the dataset with random transformations. Following the advice of Francois Chollet in Building powerful image classification models using very little data, I applied random rotations, height and width shifts and zooms. I filled in points outside the boundaries of the input with copies of the nearest pixels.

Keras's ImageDataGenerator provided a convenient tool for both addressing both the class imbalance and augmenting the dataset.

Here's an example of an image of Charlock that has been transformed by this method:

Figure 6 - An example of a transformed image used for Dataset Augmentation

Notebook Heading	**Description**
Augment and Oversample Data	Creates 2000 Augmented Training Images for each class. These are saved to the data/augmented folder.

This chart confirms that classes are now equally balanced:

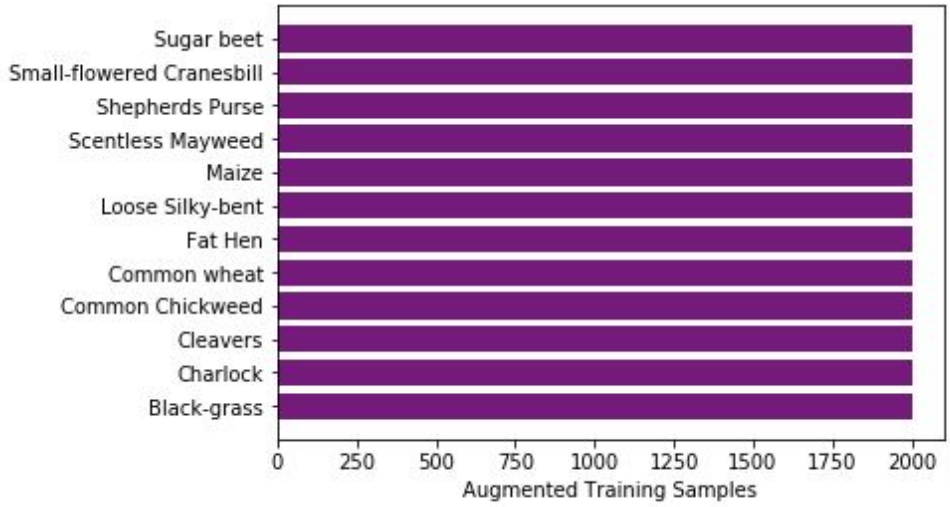


Figure 7 - Class Balance in Training Samples after Oversampling

Implementation

Notebook Heading	**Description**
Define F1 Score for Training in Keras	Keras does not offer F1 Score as built-in metrics function a F1 Score as metric for training in keras. As a result F1 Score is provided as a custom metric. More Information: https://keras.io/metrics/ Code source: https://stackoverflow.com/questions/43547402/how-to-calculate-f1-macro-in-keras
Compile basic CNN from scratch	See description above in "Algorithms and Techniques"
Compile new CNN based on VGG 19 using Transfer Learning	See description above in "Algorithms and Techniques"
Compile new CNN based on Inception V3 using Transfer Learning	See description above in "Algorithms and Techniques"
Define Confusion Matrix Plot	Neither scikit-learn nor keras offer a built-in confusion matrix with readable formatting. This function fills that need.

	Source: http://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html#sphx-glr-auto-examples-model-selection-plot-confusion-matrix-py
Train the CNN	Train a model. The best model definition and weights are saved out to an hdf5 file with the model's name. Parameters: - model - The compiled model to train. - epochs - Training is run for a series of epochs. Each epoch covers the entire training dataset. - x defaults to x_training_tensors, but can optionally be set to x_augmented_tensors. - y defaults to y_training_targets, but can optionally be set to y_augmented_targets. - validate_using_test_set. Two options for validation in the training loop: - True: Validation uses the same test set that's also used in the evaluation step below. - False: 20% of the training data can be withheld for validation.
Evaluate Results	Evaluate the results of a model. The F1 Score for the Test Sample is displayed. A confusion matrix showing all True Positives, False Negatives and False Positives is displayed.
Create Predictions to submit to Kaggle	Load the unlabeled examples provided by kaggle. Create predictions using the model and save these out to a csv that can be submitted to Kaggle.

Refinement

I went through five major stages in refining my model. These are described in this section along with the resulting metrics.

1 - Initial Basic CNN

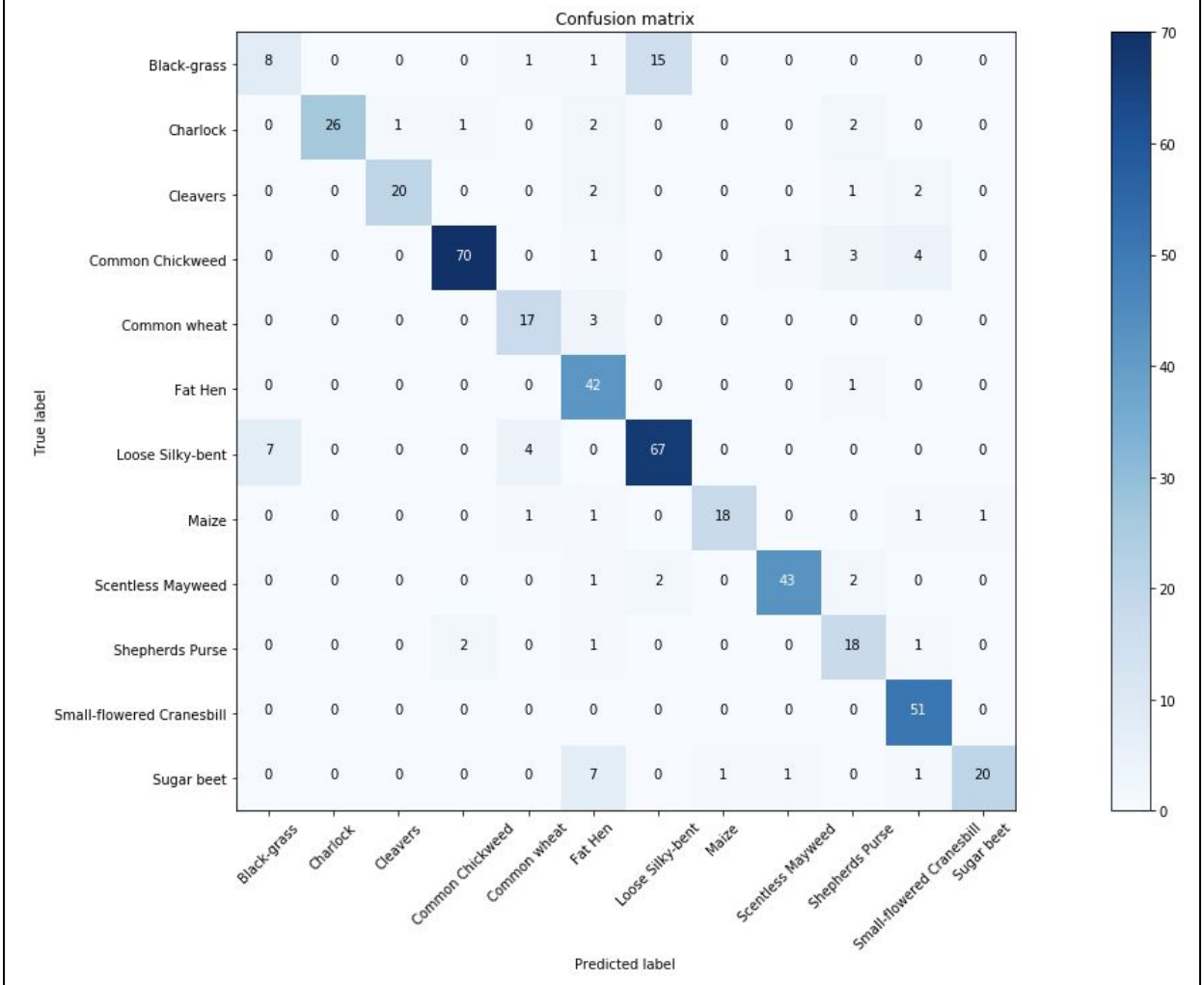
The initial basic CNN was created from scratch and run with the following hyperparameters:

n_epochs	10

x	x_train_tensors
y	y_train_targets
validate_using_test_set	True
optimizer	rmsprop

Resulting Metrics:

F1 on Test Data	0.84211
F1 on Kaggle Leaderboard	0.85012



2 - VGG19 Transfer Learning

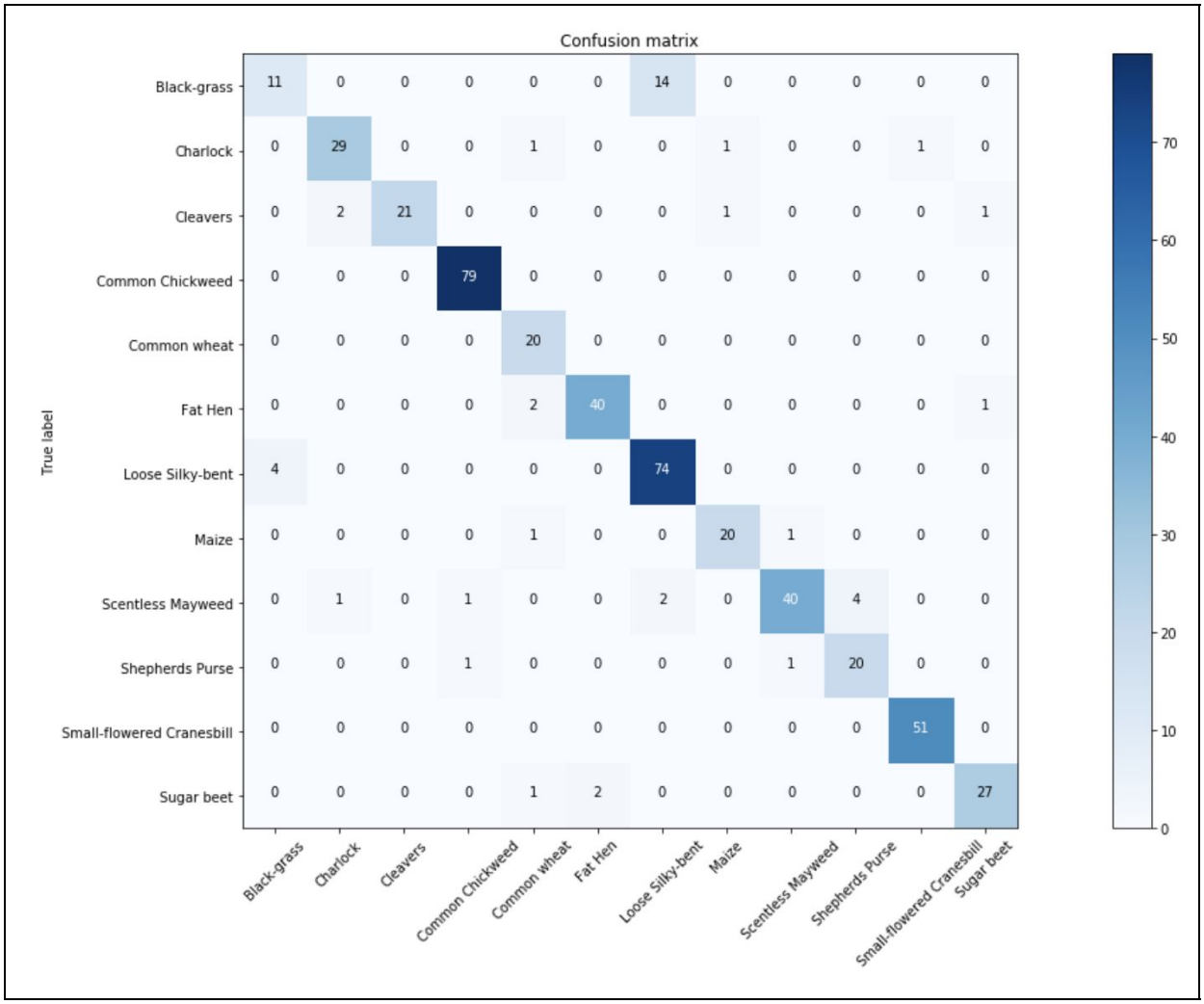
Next I attempted transfer learning with VGG19. Weights were frozen on the first 10 layers of 22 total. So 12 layers were trained.

The following hyperparameters were used:

n_epochs	10
x	x_train_tensors
y	y_train_targets
validate_using_test_set	True
last_layer_to_freeze	10
optimizer	Stochastic Gradient Descent (Learning Rate=0.0001, Momentum=0.9)

Resulting Metrics:

F1 on Test Data	0.90947
F1 on Kaggle Leaderboard	0.85264



3 - Inception V3 Transfer Learning

Next I attempted transfer learning with Inception V3. Weights were frozen on the first 200 layers of 313 total. So 113 layers were trained.

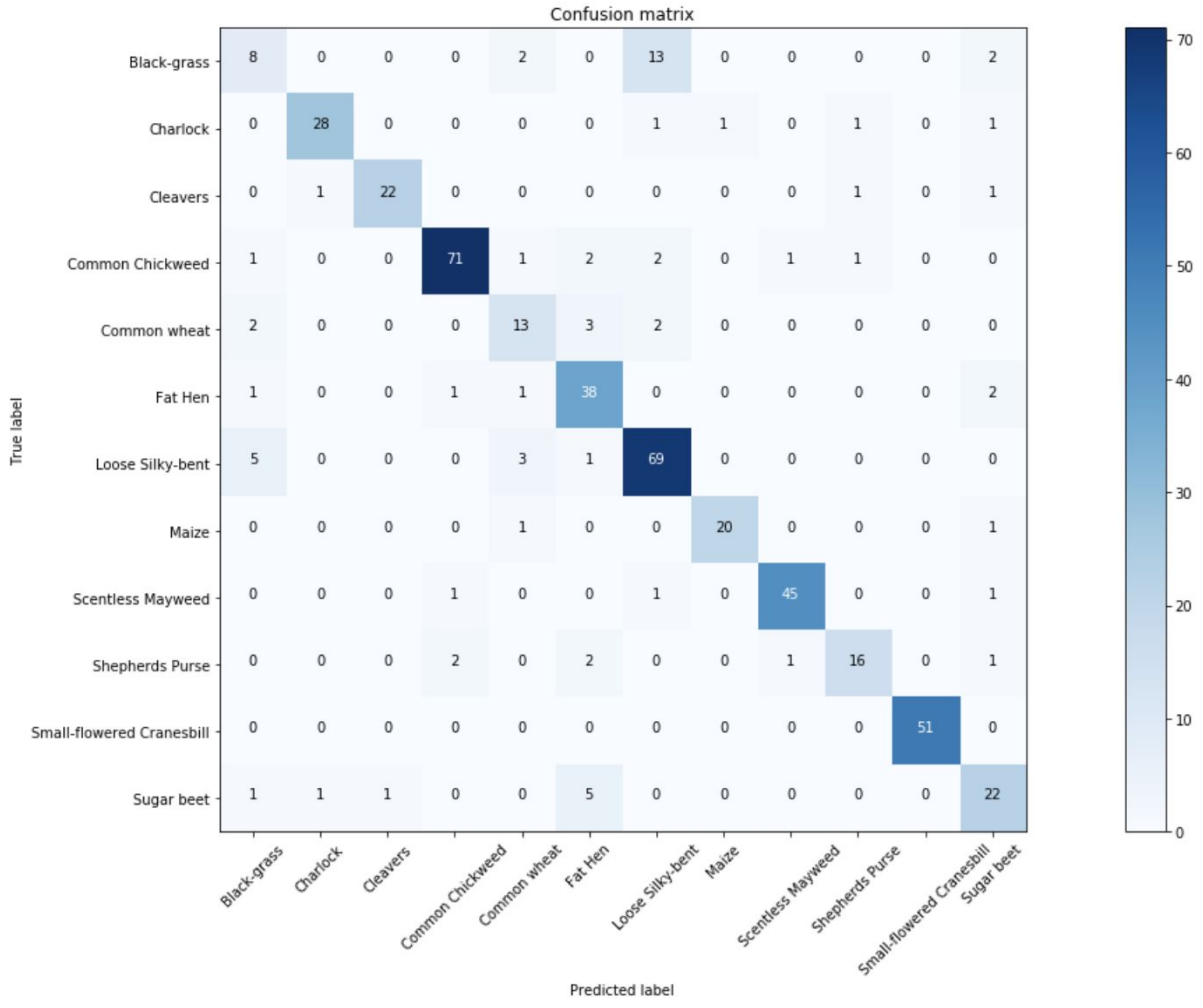
The following hyperparameters were used:

n_epochs	10
x	x_train_tensors
y	y_train_targets
validate_using_test_set	True

last_layer_to_freeze	200
optimizer	Stochastic Gradient Descent (Learning Rate=0.0001, Momentum=0.9)

Resulting Metrics:

F1 on Test Data	0.84842
F1 on Kaggle Leaderboard	0.85390



4 - Freezing fewer layers of Inception V3

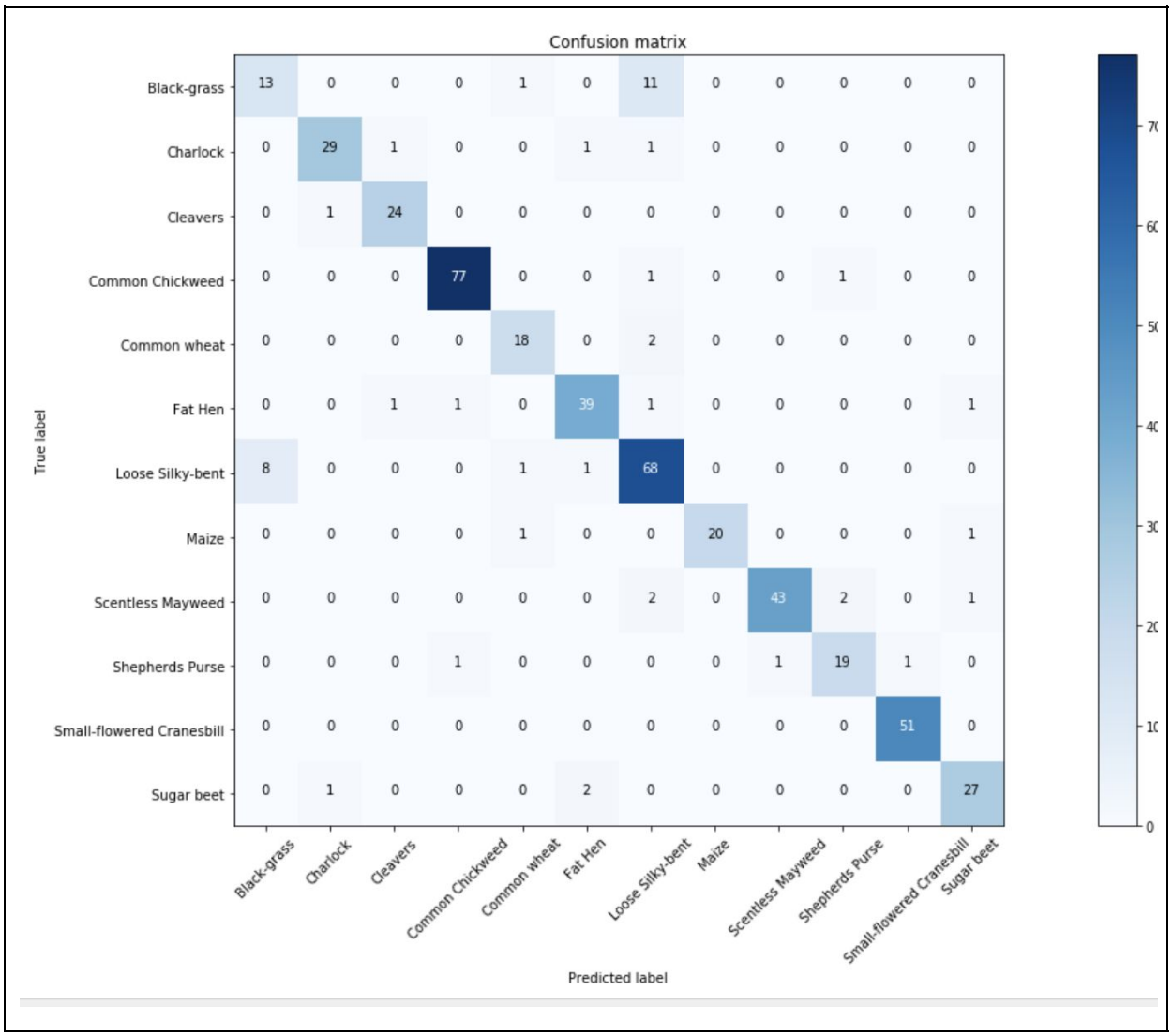
I thought it might be useful to retrain more of Inception V3. In this training run, weights were frozen on the first 50 layers of 313 total. So 263 layers were trained.

The following hyperparameters were used:

n_epochs	10
x	x_train_tensors
y	y_train_targets
validate_using_test_set	True
last_layer_to_freeze	50
optimizer	Stochastic Gradient Descent (Learning Rate=0.0001, Momentum=0.9)

Resulting Metrics:

F1 on Test Data	0.90105
F1 on Kaggle Leaderboard	0.89546



5 - Switching to Augmented / Oversampled Data

As described in the Data Preprocessing section above, I introduced code to balance classes and augment the data. As a result of these two changes, the number of training samples increased from ~ 4000 to 24,000. In order to ensure that the augmented data would be used in the validation step, I used a random sample of 20% of the new augmented training data as a validation set inside Keras's training loop.

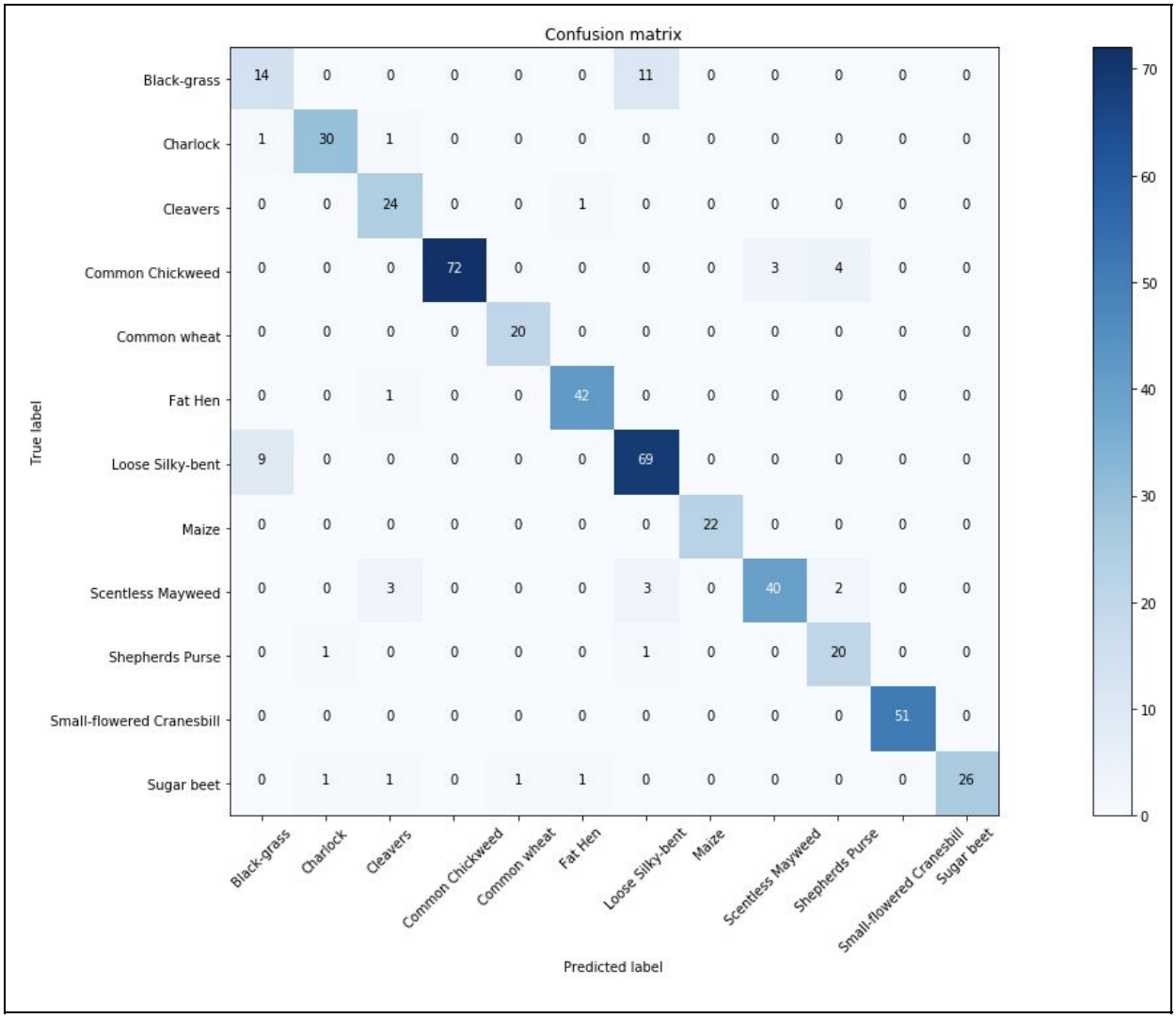
I continued to hold the original unaugmented test data out for model evaluation after training was complete.

The following hyperparameters were used:

n_epochs	10
x	x_augmented_tensors
y	y_augmented_targets
validate_using_test_set	False
last_layer_to_freeze	50
optimizer	Stochastic Gradient Descent (Learning Rate=0.0001, Momentum=0.9)

Resulting Metrics:

F1 on Validation Data (20% subset of augmented data)	0.9648
F1 on Test Data	0.90526
F1 on Kaggle Leaderboard	0.92065



A quick note about optimizers

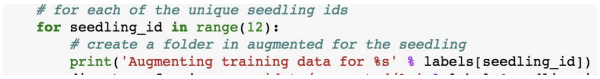
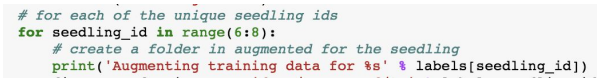
I experimented with both traditional **stochastic gradient descent** and the **rmsprop** optimizer. According to Sebastian Ruder in [An overview of gradient descent optimization algorithms](), rmsprop is one of a family of improved optimizers which allow learning rates to differ between parameters and be adjusted dynamically during optimization.

- I trained the from-scratch CNN with both optimizers and found that rmsprop dramatically outperformed SGD.
- I used SGD for all the Transfer Learning runs. I did do try the rmsprop optimizer with my preferred model as a final check, but it performed more poorly than SGD.

Challenges Overcome

For those who wish to repeat this work or improve upon it, here are a few of the ways I dealt with potential stumbling blocks. In some noted cases, the problems have only been partially solved.

Challenge / Why it's important	**How I addressed it / Open Questions**
How to get access to state-of-the-art GPUs. CNNs can have many trainable parameters and can be slow to train without the parallelization factor of GPUs.	Provisioned Google Compute Engine instances with NVIDIA Tesla K80 GPUs. Here's a starting point to learn now.
How to provision the Compute Instance so it can use GPUs? It's not obvious. Required drivers and libraries must be manually installed.	The Compute Engine survival training: TensorFlow Estimator guide was indispensable.
How to expose jupyter so that I could access it from my local machine? This was an incredibly finicky process.	I had to ensure firewall rules were correct. It was helpful to follow the instructions in Running Jupyter Notebook on Google Cloud Platform in 15 min. Of particular note, I found it critical to specify both the port and ip_address when starting jupyter.
How to do transfer learning in Keras?	As noted above, the approach I read about in Building powerful image classification models using very little data ended up being quite a bit easier to implement than the method of discarding the last layers of the pre-trained models. But implementing the original approach is a good next step. The Transfer Learning with Keras post would be a good place to start.
How to avoid running out of memory when creating augmented images?	In order to avoid running out of memory, I ended up doing a few classes at a time. In the "Augment and Oversample Training Data" notebook cell

	 was briefly rewritten to only cover some of the seedlings. 
How to avoid running out of memory when training using augmented images?	I initially used a Google Compute Engine instance with 15 GB of memory. I ran out of memory when training using augmented data. I increased the machine size to have 100 GB of memory. I watched Memory usage during training and saw that it used ~ 40 GB. As a future improvement, note that Keras's DataImageGenerator does not require the images to all be in memory at the same time. Scaling up Google Compute Engine instance with more memory was simply easier than modifying the code.
How to fully explore the hyperparameter space?	I only spot-checked a few combinations of hyperparameters as noted in the Refinement section above. Next steps could include more exploration of the space. In particular, I believe significant progress could be made by changing how many layers of Inception V3 weights are frozen.

Results

Model Evaluation and Validation

The fifth model I built - the Inception V3-based model trained on augmented data - is my preferred model because it performs best in a generalizable way.

The second model I built - the VGG19-based model - achieves the highest F1 score on the Test Dataset amongst all my training runs. However, it doesn't generalize well and has a much lower F1 score on the unlabeled Kaggle data. This indicates that the model has overfit the test dataset.

Justification

Compared to other Kaggle Competition Submissions

At the time of this writing, an F1 score of 0.92065 ranks at #143 of 263 on the public leaderboard for the plant-seedlings-classification competition. In my opinion, this is a respectable showing.

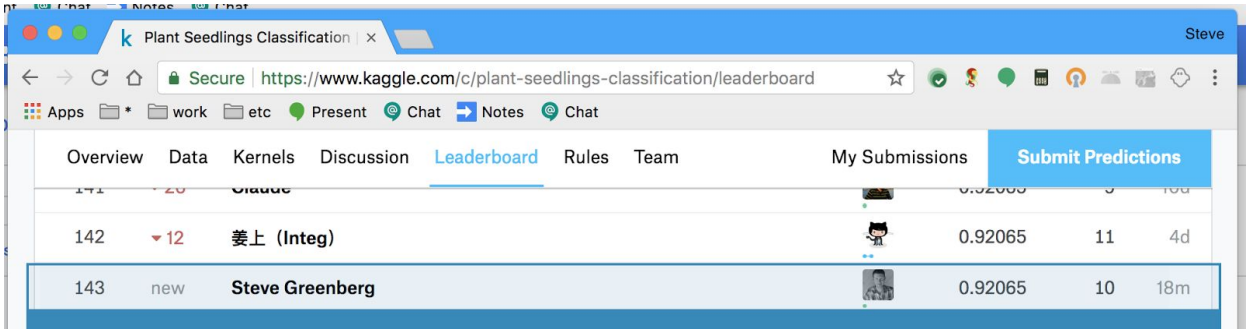


Figure 8 - Position of Preferred Solution within the Kaggle Leaderboard at the time of this writing

Compared to the "Getting Started" Kaggle Kernel

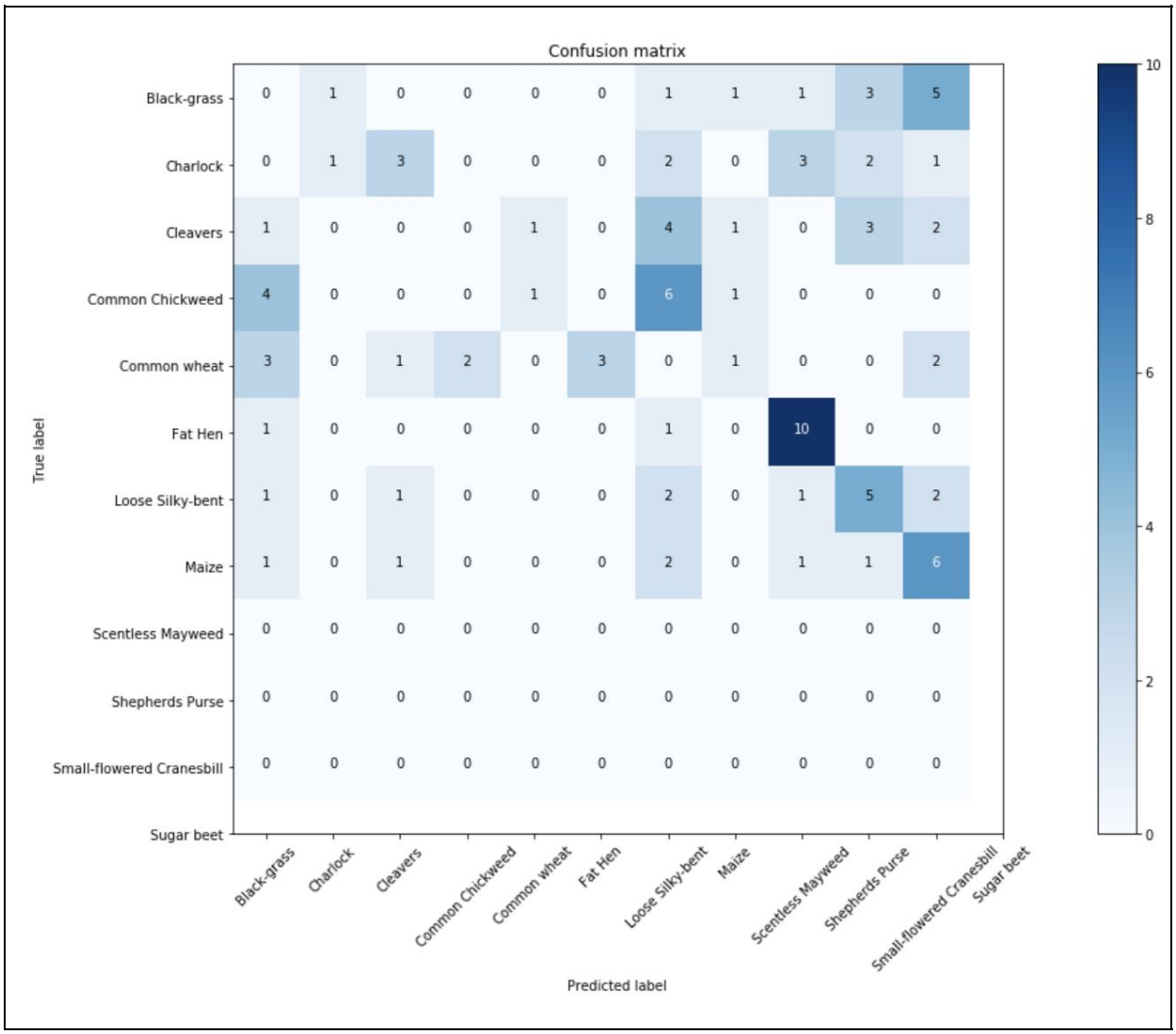
At outset of this project I noted that there was a popular Getting Started Kaggle Kernel tutorial that used Transfer Learning. Though this is excellent work, I wanted to use more primary sources for my work on this Capstone project, so I did not refer to it.

After completing my model and getting to an F1 Score of 0.92065, I ran the code in this tutorial as a benchmark. It achieves an F1 Score of 0.85264 against the same data. So my work compares favorably with this Getting Started tutorial.

Evaluated against images from the Wild

I tested my model on images from the wild, and the results were not good. Only 3 of 120 samples (2.5%) were correctly classified. We would expect random chance guessing to produce about 8.3% correct classifications in a dataset with samples evenly divided between the 12 classes.

F1 on Images from the Wild	0.03125



The conclusion is that the model generalizes well to classify weeds in the artificial conditions created for the Kaggle competition dataset. However, it does not generalize to classify weeds in natural conditions.

Conclusion

Recap of Process

- I looked through active Kaggle competitions to find one that would work well for this project and fulfill my professional goals.
- Because this competition would benefit from the use of Public Cloud resources and GPUs, I felt like it was a good fit.

- I created a Google Compute Engine instance, provisioned Keras with access to local GPUs and exposed Jupyter across the internet to my local machine.
- I downloaded the datasets from Kaggle.
- After some basic spot checking using my computer's file explorer, I built some basic data visualizations to explore aspects of the dataset - image sizes, class balance, and to preview samples of the data.
- I put the data through a basic from-scratch CNN, and then researched more about how to do Transfer Learning in Keras on small datasets.
- I submitted initial CSVs to Kaggle and saw an F1 score in the mid-80s on the unlabeled data.
- I decided that two changes to the model would help - oversampling to solve class imbalance and augmenting the data.
- I introduced Image Augmentation and Oversampling to solve these problems.
- I arrived at my preferred model.
- I examined the balance of image sizes across the classes to see if it imbalance could be affecting our model.

Illustration of End to End Process

The following diagram outlines the end to end process for training and evaluating my preferred model.

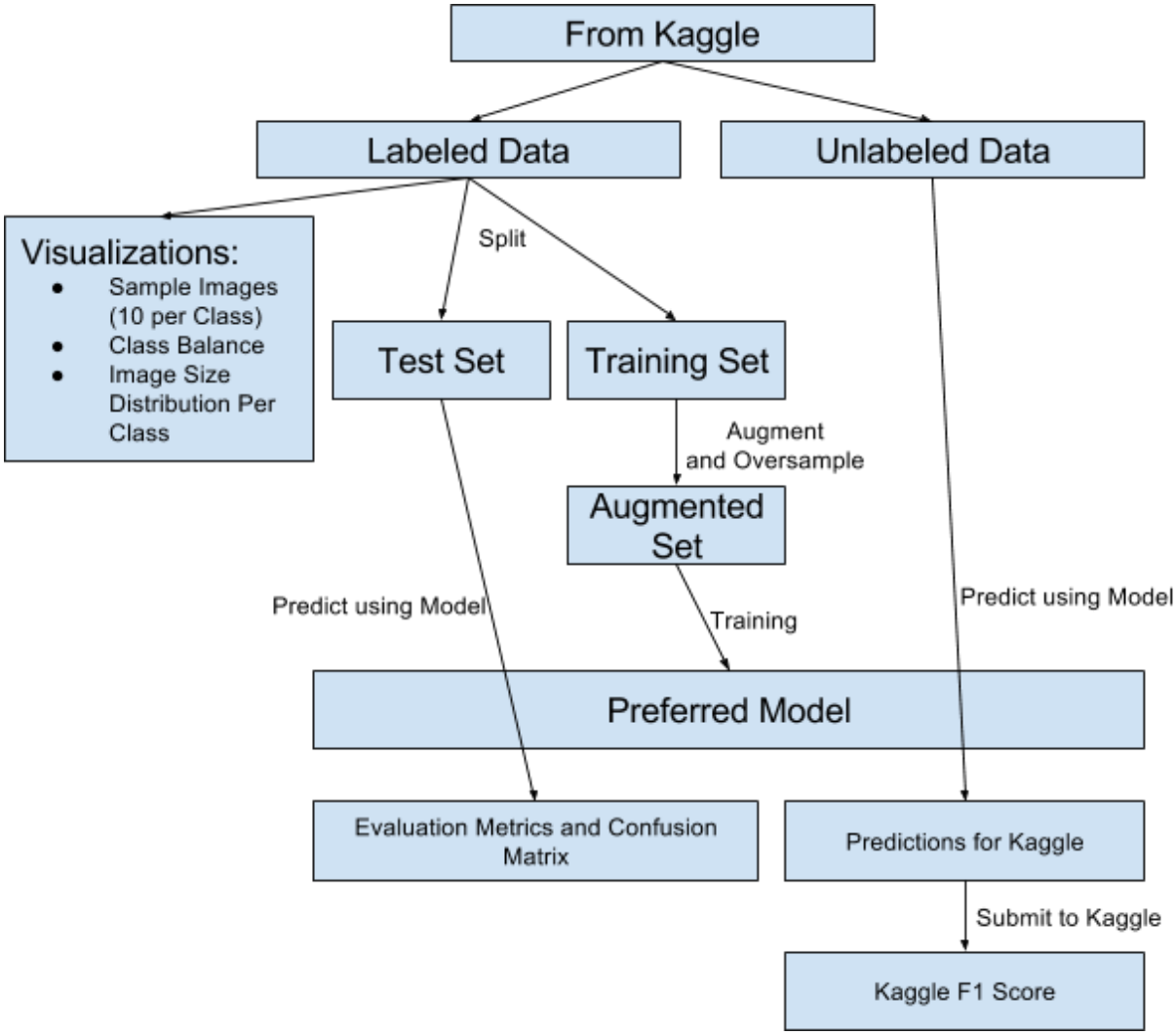


Figure 9 - The end-to-end process for my preferred model.

Reflection

It's satisfying to have built a model that performs reasonably well within the kaggle competition. The model does an excellent job of distinguishing between 10 of the classes.

However, the confusion matrices make it clear that the model often confuses Loose Silky-bent and Black-grass. Both of these plants are thin whisps of grass. It's incredibly hard for me to see the difference between them in the samples.

If we want to improve our model's performance within the Kaggle competition, the next steps should clearly be targeted at improving disambiguation between Loose Silky-Bent and Black-Grass.

However, before setting off on that workstream, I would want to step back and think about the model's real-world usability. Its inability to generalize to data from the wild is concerning. This model may be useful for a Kaggle competition but it's not clear that it's useful for any real-world application.

In the Data Exploration section of this report, I talked about the Image Size problem. In the Free-Form Visualization section, I'll return to this.

Free-Form Visualization

There are two potential problems introduced by rescaling seedling images:

Potential Problem 1: Does rescaling throw out useful information?

When we rescaled at the images to be 224 X 224 pixels, we threw out some critical information - namely that each pixel shows 1/10th of a mm of the seedling. Two plants may have similar appearances but actually be different sizes. If this were the case, our preprocessing could hurt our ability to differentiate between them. In fact there is some evidence that this may be the case with Loose Silky-Bent and Black Grass. It looks like Loose Silky Bent's Ligule is larger than Black Grass's.

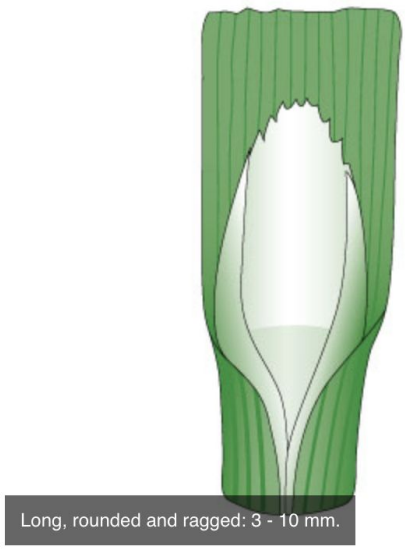


Figure 10 - Illustration of Loose Silky-Bent Ligule

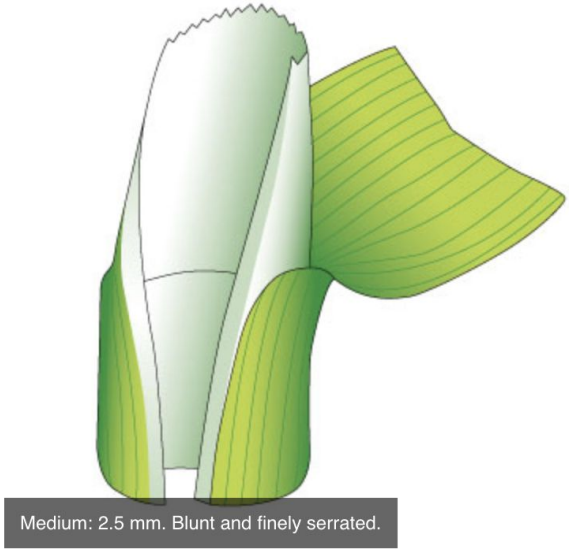


Figure 11 - Illustration of Black-Grass Ligule

Potential Problem 2: Does rescaling create latent features that don't generalize?

If there are class-imbalances in the size of images, rescaling might unintentionally create features specific to images taken in artificial conditions. If this were the case, the model might not be learning the visual properties of seedlings but instead be learning to distinguish between images scaled to 224 X 224 pixels from different source sizes.

To understand if this may be the case, it's helpful to understand the distribution of image sizes across the 12 classes:

Figure 12 - Distribution of Image Sizes within each class

This visualization doesn't definitively tell us if classes are being distinguished by the original size of their image, but there are significant differences. Notably:

- The median size of a Loose Silky-bent image is about 250 X 250 px, meaning that the pictures are not rescaled very much.
- The median size of a Black-grass image is about 600 X 600 px, meaning that they are scaled down by ½.

Further work would need to be done to determine if either of these potential problems is actually occuring in a significant way.

Improvement

Next steps to improve the model depend on one's goal.

If the goal is to improve the score in the Kaggle competition:

- Additional hyper-parameter tuning and image augmentation could make a difference.
- However, the model already performs quite well for 10 classes of data. The most improvement will come from improving the model's ability to distinguish between Loose silky-bent and Black-grass.
- I would be tempted to isolate those two classes of seedlings and try to build a secondary model optimized for them.

If the goal is to perform better on images in the wild:

- The first step will be to increase the available dataset. 120 images is not enough to get started.
- Removing the Reference Scales from the pictures by modifying the images to replace stark white and black pixels with background noise (like the small brown pebbles in the images) might help the pictures more accurately represent those found in the outdoors.

Final Thoughts - Additional Reflections

- I came away from this project very impressed with the power of Convolutional Neural Networks. The performance of even the basic CNN built from scratch after merely ten epochs of training is really impressive. It's certainly able to distinguish between these seedlings better than a human would be able to do with a similar time spent training!

- Cloud Resources - in particular the ability to quickly spin up a powerful Virtual Machine with 100 GB of RAM and with state-of-the-art GPUs - is an incredibly powerful tool for Data Scientists.

- It's easier to scale up than to scale out. I consciously scoped this project to avoid the challenge of distributing training, and I did all my image augmentation in-memory.

- Overall, this felt like a satisfying end to the Udacity ML Engineering Nanodegree.