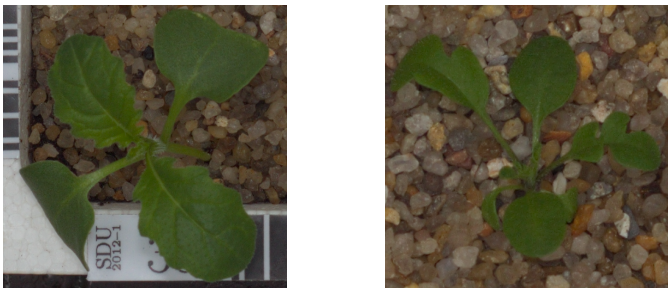
Mihwa Han
January 15, 2018

Deep Learning in the Weeds

1. Definition

1-1. Project Overview

Can we distinguish a weed from a crop seedling? Many weeds and crop seedlings look similar, so sometimes it can be hard to differentiate the two. Although they look alike when they are young, they will eventually grow into totally different plants. This challenge presents itself in many real-world situations. For example, Charlock is an agricultural weed and an invasive species in some areas outside its native range, while Shepherds Purse, considered an herbal medicine, has seedlings of very similar shape to Charlock (See Fig1). Thus, farmers need to differentiate each type of seedling to successfully cultivate without damaging their plants. The goal of this project is to build a model to effectively classify several species of seedlings. This is based on a Kaggle project called "Plant Seedlings Classification" (https://www.kaggle.com/c/plant-seedlings-classification). The dataset contains 4,750 training images and 794 test images, each of which belongs to one of twelve species at several different growth stages. I aim to train several deep learning models using Python/Keras, and test how accurately they can estimate the correct species.



Charlock Shepherds Purse

Fig1. Examples of Charlock and Shepherds Purse. We can see that they look very similar.

1-2. Problem Statement

Why is this project meaningful? As described earlier, many types of seedlings look very similar, so even an experienced gardener can have difficulty differentiating weeds from the seedlings of more helpful plants. Moreover, as each species includes different growth stages, the character of each seedling could change depending on their stage of growth. Furthermore, I will use low resolution images, which are more likely representative of real life, and because of their low resolution it is not always easy for even an experienced human to identify the correct seedling species.

This problem is related to a computer vision challenge, so I will leverage a Convolutional Neural Network (CNN) to train a model to categorize each seedling in the dataset. CNNs are particularly powerful for training on multi-dimensional data, such as images. They are designed to handle 2D spatial information in images particularly well; unlike a regular neural network, CNNs use locally-connected layers and do not use vectors for their hidden layers. To do this, I need to implement models with multiple trials with different CNN layers, such as convolution and pooling layers by using different optimizing parameters and dropout. Furthermore, I will compare my models from scratch with models using the popular technique of transfer learning with Xception, which has been established to help save time when training neural networks on image classification problems. Finally, I will determine the best model to predict seeding species using mean F-score as the evaluation metric, and I will elaborate on it in the following section.

1-3. Metrics

I will use mean F-score to measure model success. The mean F-score is a weighted average of the *precision* and *recall*, which are given by the equations below:

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

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

The summations are over all species of seedling. *TP* means True Positive, which are cases where the correct seedling species is predicted. *FP* means False Positive, which are cases where a seedling is incorrectly predicted to be a particular species. *FN* means False Negative, which are cases where a seedling is not predicted to be its true species.

Therefore, mean F-score is written as the following:

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

Mean F-score is usually more useful than accuracy, especially for situations with uneven class distributions. For example, mean F-score works well when the cost of FP and FN are very different because it takes into account both the precision and recall.

2. Analysis

2-1. Data Exploration

The dataset contains 4,750 training images and 794 test images, each of which belongs to one of twelve species at several different growth stages. I obtained the dataset through a Kaggle competition, but originally the dataset was from the Aarhus University Department of Engineering Signal Processing Group. The training images include the label of seedling species, while the test images do not include labels. I am using only the 4,750 training images to train a model and validate and test the model accuracy.

The input images are all square, but they have a wide range of pixel widths and heights, with some less than 100x100 pixels and some over 1000x1000 pixels, and each image has RGB colors. The figure on left (Fig2) shows the distribution of pixel widths (also equal to the height distribution) for all input images. I will need to rescale the input images to a uniform size to input into my neural network. A smaller uniform size makes the most sense, considering that many images are ~100 pixels on a side, and it's easier to reduce the sizes of images than to reliably scale up the smaller images to a larger size. Also, using a smaller uniform size will help to save computational power. Therefore, I chose to rescale all images to the minimum size of the input data, 47 x 47.

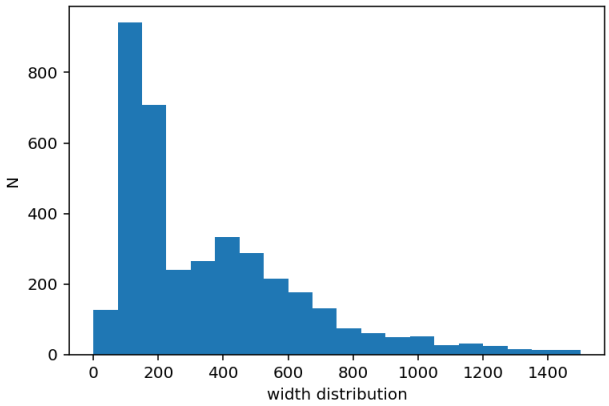


Fig2. Distribution of width/height of input images

Labels	Total
Loose Silky-bent	654
Common Chickweed	611
Scentless Mayweed	516
Small-flowered Cranesbill	496
Fat Hen	475
Charlock	390
Sugar beet	385
Cleavers	287
Black-grass	263
Shepherds Purse	231
Common wheat	221
Maize	221

The input images include twelve seedling species from the following: Black-grass, Charlock, Cleavers, Common Chickweed, Common wheat, Fat Hen, Loose Silky-bent, Maize, Scentless Mayweed, Shepherds Purse, Small-flowered Cranesbill, Sugar beet. The table on the right gives the total number of each species, which range from 221 - 654.

I split the 4,750 images into three different datasets: training set (80%), validation set (10%), and a test set (10%). The training set is used to train the model and the validation set is used to adjust weight parameters to check the loss function after training the model. Finally, the test set serves to check the model accuracy using our scoring metrics. The images below are examples of each seedling of the input images.

2-2. Exploratory Visualization

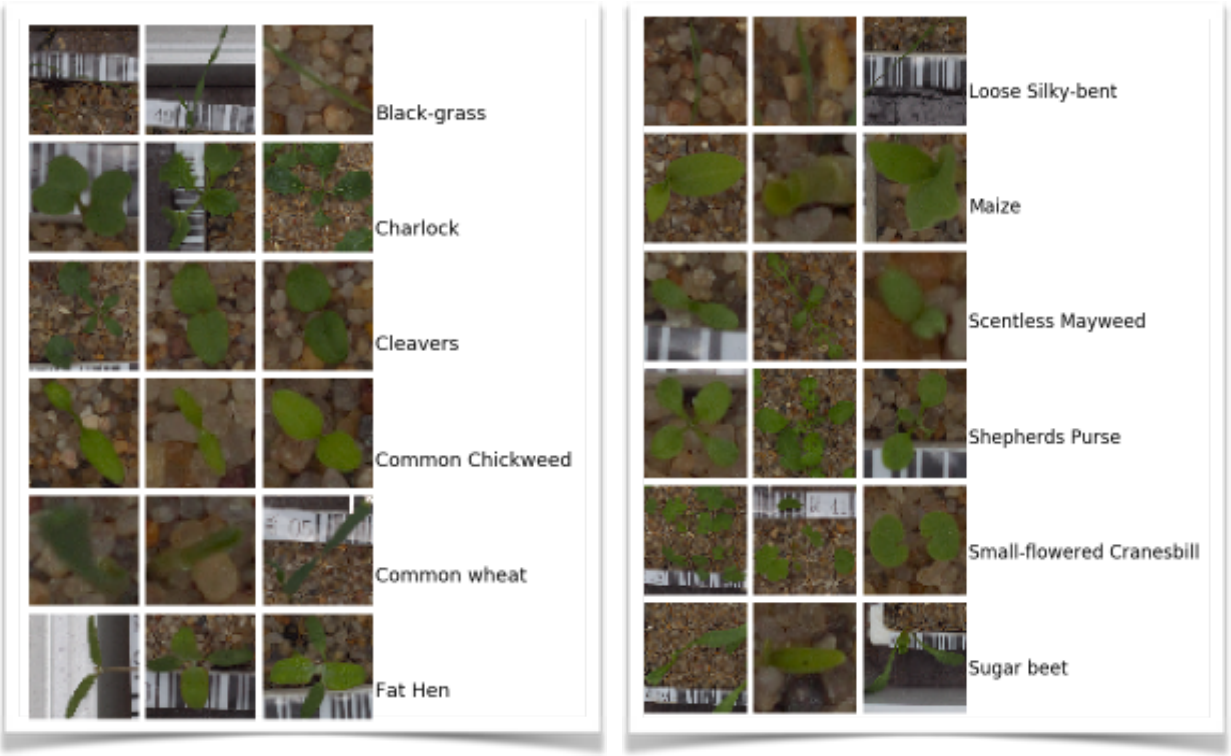


Fig3. Several examples of the twelve kinds seedlings in the input data.

As seen above (Fig3), the input images only include soil, gravel, and measure tape, except seedlings. So, filtering with other objects might not matter much. However, some seedlings, such as Black-grass, have pretty narrow leaves, so it might not be easy to train on those images. Also, some photos are blurry, so it could be hard to predict "by eye" as well. As mentioned earlier, as part of the pre-processing I rescale all input images to the minimum size of the input images, 47 x 47, which helps to save computational time.

2-3. Algorithms and Techniques

Why CNN?

Neural Networks are designed to mimic the process of learning from the human brain. They can be comprised of multiple layers, each containing many nodes to simulate neurons. When training a neural network, the weights applied to each node are adjusted. A regular neural network connects every node to every other node in each successive layer, making them fully connected. To process an image with a regular neural network, the image must first be transformed into a vector. This transformation is not optimal for analyzing 2-D images because it uses a vector for each layer. On the other hand, CNNs are especially powerful when we must train on multi-dimensional data, such as images, because layers of a CNN have neurons arranged in 3 dimensions. CNNs consist of *locally* connected layers, which use far fewer weights compared

to the densly connected layers of a regular neural network. The locally connected layers efficiently prevent overfitting and allow us to easily understand the image data because they naturally handle two dimensional patterns. You can read more details here. Fig4 shows the basic architecture of a regular and a convolutional neural network.

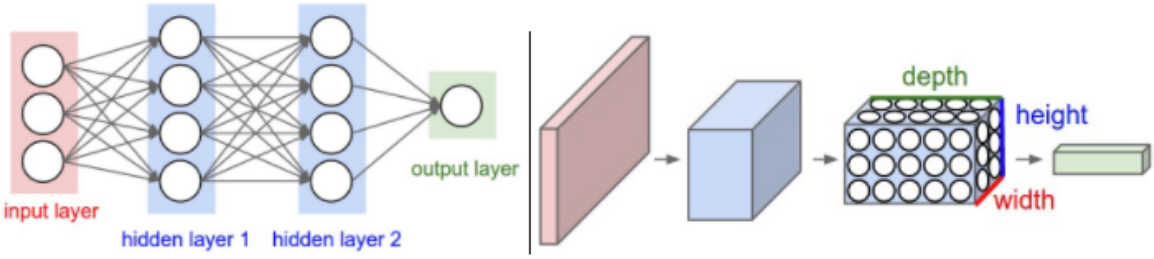


Fig4. (Left) Regular Neural Network (Right) CNN.
(Source : http://cs231n.github.io/convolutional-networks/#overview)

CNN Design

(1) Convolutional Layer

A convolutional Layer consists of locally connected nodes, meaning that the nodes are only connected to a small subset of the previous layers' nodes. To build a convolutional layer, we first select a width (column) and height (row) that defines a convolution filter. The filter is a matrix that can have its own characteristic pattern, and each convolutional layer will have the task of searching for its filters pattern in the image. Fig 5 demonstrates how a convolutional layer works.

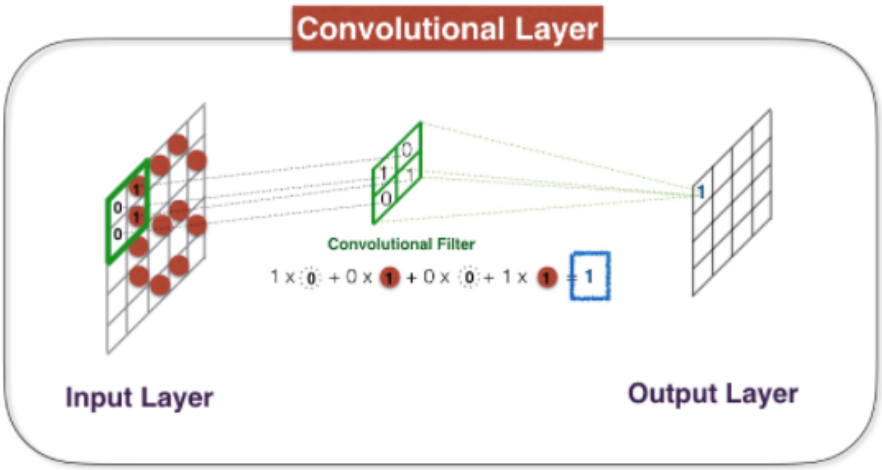


Fig5. Convolutional Layer schematic

(2) Pooling Layer

Recall that a convolutional layer is a stack of feature maps where we have one feature map for each filter. More filters means a larger stack, which means that the dimensionality of our convolutional layers can get quite large. Higher dimensionality means we will need to use more parameters, which can lead to overfitting. Therefore, we need a method to reduce this dimensionality by using pooling layers within a CNN. Generally, there are two popular choices for types of pooling layers, max pooling layer and global average pooling layer. Fig6 shows how a pooling layer works.

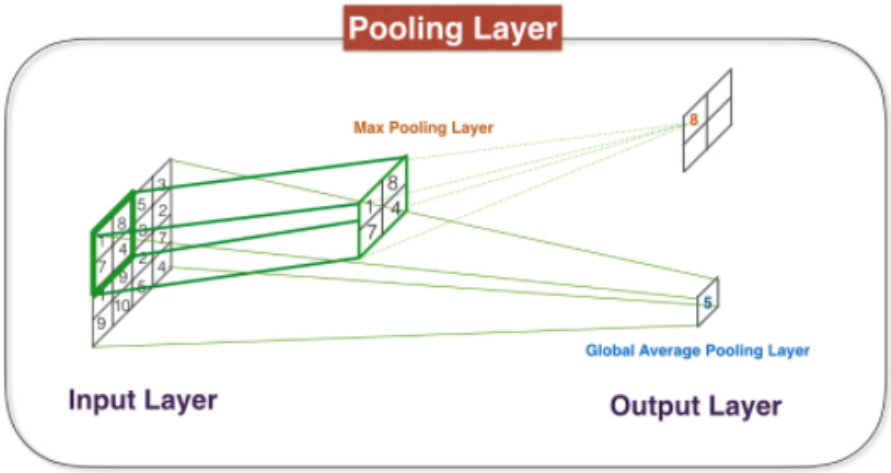


Fig6. Pooling Layer schematic

(3) Activation Function

Activation functions serve as a gateway for nodes in a neural network; meaning, activation functions decide if a neuron should be activated or not. There are different types of activation functions like tanh and the sigmoid function. In this project, I usually use ReLU function as my activation function, whose equation is: $f(x) = max(0,x)$

I will use a CNN to train a model using Keras in Python to categorize each seedling in the dataset. Keras is a deep learning library in Python, allowing for easy and fast prototyping. Keras allows one to build a model with a linear stack of layers, so I can quickly build a basic CNN structure with multiple layers and see how it performs.

I will start with a basic CNN structure with three convolutional layers and pooling layers. Also, it will have a fully-connected layer with a softmax activation function before the output layer. Once the basic CNN is set up, I will improve the model in several ways such as:

1) adding more CNN layers
2) different size of filters

3) different optimization functions (SGD, Adam)
4) image augmentation

The figure below (Fig7.) is an example design I used after adding more layers to my initial CNN. Later I will compare the results of the CNN models with the results I obtain using transfer learning, by building CNNs on top of the popular models InceptionV3 and Xception.

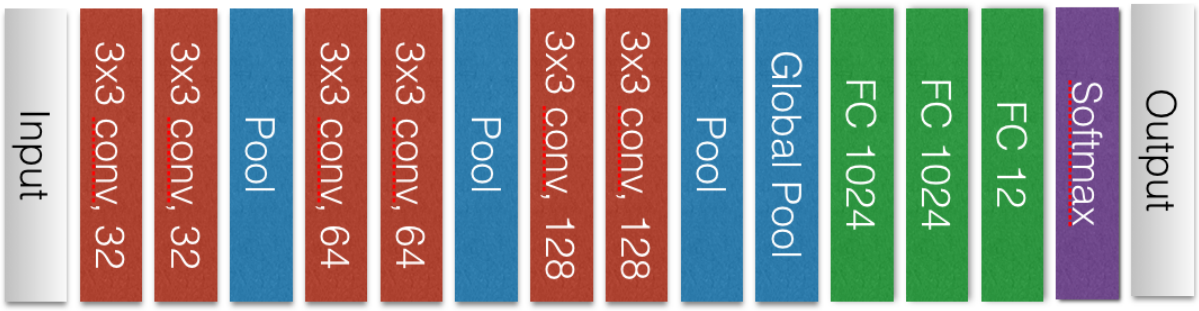


Fig7. An example design used for the CNN model.

2-4. Benchmark

For a benchmark model, I will use the simple CNN model described above with three convolutional layers and max-pooling layers. I tested this simple CNN on the input dataset, and I achieved a mean F-score of 0.40, and a prediction accuracy of 0.4. This accuracy is much better than random chance (1/12), but still not impressive. The aim of this project is to find a model with significantly better mean F-score and accuracy.

3. Methodology

3-1. Data Processing

The data preprocessing function includes three processes as follows: First, the images are reshaped to have the same width and hight (47 pixels by 47 pixels). Second, I convert the image data to Numpy arrays with (number of images, 3, 47, 47). Also, I convert the pixel values to float32 format and normalize all of the arrays. In the case of the label dataset, I simply convert 1-dimensional class arrays to 12-dimensional class matrices. The code below performs this data processing. All codes are available in my github.

```
def img_to_tensor(img_path):
    img = image.load_img(img_path, target_size=(47,47))
    x = image.img_to_array(img)
    return np.expand_dims(x, axis=0)
```

img_to_tensor function to reshape input image as (number of images, 3, 47, 47), which include:

image.load_img : load an image into Python Imaging Library (PIL) format
image.img_to_array : convert a PIL Image instance to a Numpy array.
numpy.expand_dims : insert a new axis

```
def imgs_to_tensor(img_paths):
    list_of_tensors = [img_to_tensor(img_path) for img_path in tqdm(img_paths)]
    return np.vstack(list_of_tensors)
```

imgs_to_tensor function to stack all of the lists to run img_to_tensor on all input images.

```
def load_dataset(path):
    data = load_files(path)
    files = np.array(data['filenames'])
    targets = np_utils.to_categorical(np.array(data['target']), 12)
    return files, targets
```

load_dataset function to load the labels for each image and convert a class vector to binary class matrix using np_utils.to_categorical.

```
labels = listdir("./train")
train_files, train_targets = load_dataset('./train')

test_files = [join("./test/",f) for f in listdir("./test/") if isfile(join("./test/", f))]

rnd = np.random.random(len(train_targets))
train_idx = rnd < 0.8
valid_idx = rnd >= 0.8

X_train = train_files[train_idx]
X_valid = train_files[valid_idx]

y_train = train_targets[train_idx]
y_valid = train_targets[valid_idx]

train_tensors = imgs_to_tensor(X_train).astype('float32')/255
valid_tensors = imgs_to_tensor(X_valid).astype('float32')/255
test_tensors = imgs_to_tensor(test_files).astype('float32')/255
```

(1) Using load_dataset function, I converted the labels for each seedling of the input sample to a binary class matrix.

(2) Using random number generation, I split the input images into three different dataset, including train set (80%), validation set (10%), test set (10%).

(3) Using imgs_to_tensors, I reshape all input images to the appropriate tensor with dimensions (number of images, 3, 47, 47).

(4) I finally converte the pixel values to float32 format and normalize all of the arrays.

3-2. Implementation

Next, I start with a basic CNN structure with three convolutional layers and Max Pooling layers. Also, it will have fully connected layer with activation function (Relu) before the output layer. Finally, I added Global Average Pooling layers and a fully connected layer to predict the probabilities for each species. The code for this basic CNN structure is below:

```
from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D
from keras.layers import Dropout, Flatten, Dense
from keras.models import Sequential

model = Sequential()
model.add(Conv2D(filters=16,kernel_size=2, activation='relu',
                 input_shape=train_tensors.shape[1:]))
model.add(MaxPooling2D())
model.add(Conv2D(filters=32,kernel_size=2, activation='relu'))
model.add(MaxPooling2D())
model.add(Conv2D(filters=64,kernel_size=2, activation='relu'))
model.add(MaxPooling2D())
model.add(GlobalAveragePooling2D())
model.add(Dense(12,activation='relu'))
model.compile(optimizer='rmsprop', loss='categorical_crossentropy')
model.summary()
```

executed in 155ms, finished 20:06:38 2018-01-09

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_501 (Conv2D)          (None, 46, 46, 16)        208
_________________________________________________________________
max_pooling2d_40 (MaxPooling (None, 23, 23, 16)        0
_________________________________________________________________
conv2d_502 (Conv2D)          (None, 22, 22, 32)        2080
_________________________________________________________________
max_pooling2d_41 (MaxPooling (None, 11, 11, 32)        0
_________________________________________________________________
conv2d_503 (Conv2D)          (None, 10, 10, 64)        8256
_________________________________________________________________
max_pooling2d_42 (MaxPooling (None, 5, 5, 64)          0
_________________________________________________________________
global_average_pooling2d_17  (None, 64)                0
_________________________________________________________________
dense_20 (Dense)             (None, 12)                780
=================================================================
Total params: 11,324
Trainable params: 11,324
Non-trainable params: 0
_________________________________________________________________
```

A callback is a set of functions to be applied at given stages of the training procedure. The code for my callback function implementation is below. For fitting a model, I used 10 epochs and 20 for the batch size.

```
from keras.callbacks import ModelCheckpoint

checkpointer = ModelCheckpoint(filepath='weights.from_scratch.hdf5',
                               verbose=1, save_best_only=True)
model.fit(train_tensors, y_train,
          validation_data=(valid_tensors, y_valid),
          epochs=10, batch_size=20, callbacks=[checkpointer], verbose=1)
```

executed in 1m 12.9s, finished 15:49:42 2018-01-09

```
Train on 3772 samples, validate on 480 samples
Epoch 1/10
3740/3772 [============================>.] - ETA: 0s - loss: 2.4031 - acc: 0.1580Epoch 00000:
val_loss improved from inf to 2.36962, saving model to weights.from_scratch.hdf5
3772/3772 [==============================] - 10s - loss: 2.4035 - acc: 0.1569 - val_loss: 2.3
696 - val_acc: 0.1688
```

The mean F1 score of this first basic CNN is approximately 0.34, and accuracy is 0.4. I will use this as my benchmark model.

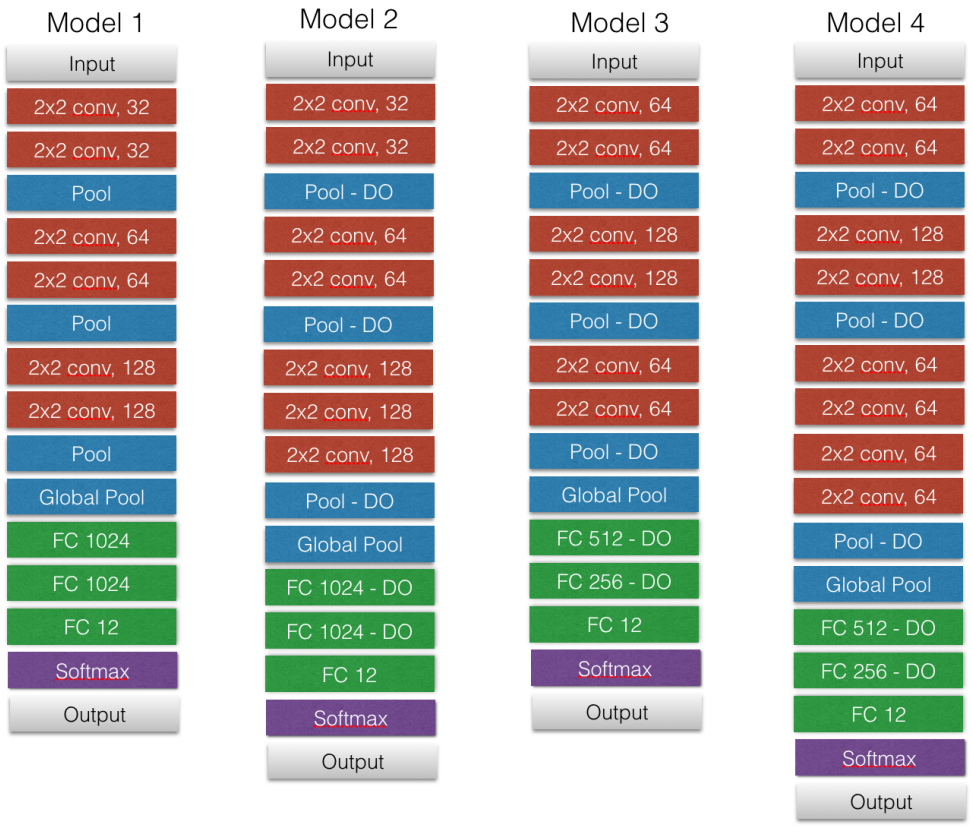


Fig8. Four CNN models designed for this project. FC is a fully connected layer and DO is dropout layer.

3-3. Refinement

Once the basic CNN is set up, I improved upon the model in several ways such as: (1) adding more CNN layers, (2) different size of filters, (3) different optimization functions (SGD, Adam), and (4) Image augmentation.

(1) I designed four different models shown in Fig8. One of the biggest challenges I faced was designing different models to test, because there are so many free parameters. I looked to other models people have used for inspiration in designing these four models in Fig8. For compiling these models, I used 'categorical_crossentropy' as loss function and 'Adam' as optimizer with the default values of the parameters (e.g., learning rate=0.001). Then, I iterated each model with 20 epochs with batch_size=20 with training set and validation set. Another difficult challenge was determining a reasonable number of epochs to train these models, because I needed to find the right balance between saving time and getting reliable estimates for the performance of these models.

The following table gives the results of each model. The accuracy and mean F-1 score values are calculated with the test set, which I did not use to train models. The running time is estimated by my macbook, which has 2.9 GHz Intel Core i5.

	Accuracy	mean F-1	Running Time
Model 1	0.79	0.74	20m
Model 2	0.74	0.68	22m
Model 3	0.83	0.81	40m
Model 4	0.70	0.64	45m

As seen above, Model 3 has the best result, with Accuracy=0.83 and mean F-1 score=0.81 with test data. However, when I compare the running time, Model 1 and Model 2 have much better time efficiency. Thus, I picked two models, Model 1 (best time efficiency with the second best mean F-1) and Model 3 (best mean F-1 value) to use in the next trial to refine my process.

(2) With Model 1 and Model 3, I trained them with a different filter size (3 x 3) compared to the original filter size (2 x 2). The table below compares the previous results to the new trials, with a larger filter size, for the two models I'm testing.

	Accuracy	mean F-1	Running Time
Model 1 (filter - 2 x 2)	0.79	0.74	20m
Model 1 (filter - 3 x 3)	0.74	0.68	28m
Model 3 (filter - 2 x 2)	0.83	0.81	40m
Model 1 (filter - 3 x 3)	0.70	0.67	65m

Overall, we can see there is no improvement for both models. So, I decided to use the first filter size (2 x 2) hereafter for my convolutional layers.

(3) I have used the Adam optimization algorithm with the default value for the learning rate. I decided to test my models using a range of learning rates: 0.01, 0.0005, and 0.0001. Also, I tried to use the Root Mean Square Propagation (RMSProps) as an alternative optimization algorithm. The results of these tests are in the table below.

	Optimizer	Accuracy	mean F-1	Running Time
Model 1	**Adam (lr=0.0001)**	0.55	0.48	20m
	Adam (lr=0.0005)	0.77	0.77	20m
	Adam (lr=0.001)	0.79	0.74	20m
	RMSProps	0.80	0.76	20m
Model 3	**Adam (lr=0.0001)**	0.62	0.60	45m
	Adam (lr=0.0005)	0.80	0.79	45m
	Adam (lr=0.001)	0.83	0.81	45m
	RMSProps	0.74	0.74	42m

Overall, the default value of the learning rate for Adam optimization function seems to be the best choice. The lower learning rates, like 0.0001, did not work well because it is too slow, and so it requires many more epochs to train the model. Considering the time efficiency, I concluded that a lower learning rate is not appropriate for the models. Also, I tried using a larger learning rate than 0.001 with the Adam algorithm, but most of them did not work well. So, I left those models off the table above. On the other hand, RMSProps works pretty well, but using Adam optimization function seems to have a slightly better result. Therefore, I decided to use an Adam optimization algorithm and used the Keras default values (e.g., lr=0.001).

```
datagen = ImageDataGenerator(horizontal_flip=True,
                             vertical_flip=True)

model.fit_generator(datagen.flow(train_tensors, y_train, batch_size=batch_size),
                    steps_per_epoch=len(train_tensors)/batch_size,
                    validation_data=datagen.flow(valid_tensors, y_valid, batch_size=batch_size),
                    validation_steps=len(valid_tensors)/batch_size,
                    callbacks=checkpointer,
                    epochs=epochs,
                    verbose=1)
```

Finally, I added Image Augmentation, which is the process of taking images that are already in a training dataset and manipulating them to create many altered versions of the same images. For example, we can have more images with different rotation angles or vertically/ horizontally flipped versions of the original images. This provides more images to train on, which could help a model to better generalize its training. The function `ImageDataGenerator()` in Keras generates batches of the input images with many different variations depending on its parameters. You can read more details here. `.flow()` and `.fit_generator()` are used to fit the model in batches with data augmentation. We can use these at the same time, as the code above shows. I only applied the horizontal_flip and vertical_flip options. This is my last refinement factor, so I trained the models with many epochs (30 - 50).

I took the best Model 1 in the table above, and the best Model 3 above, and then trained those models on an additional 30 epochs using image augmentation. The results of these tests are given in the table below.

	Accuracy	mean F-1	Running Time
Model 1 (w/ RMSProps + Image Augmentation)	0.94	0.93	20m
Model 3 (w/ Adam + Image Augmentation)	0.85	0.84	59m

Model 1 results in a better accuracy and higher mean f1 score (even with less time). Therefore, I decided to use Model 1 with RMSProps + Image Augmentation. Having settled on a model, I next decided to train it using larger input image sizes (256 x 256) with 50 epochs. At the same time, for comparison I used a pre-trained Xception model with fully-connected layers added (to do transfer learning). As with the model above, I first trained the model with Xception without image augmentation, and then trained it with image augmentation. The table below has the results using Xception. It seems surprising that the Xception model performed slightly worse than the Model 1 results above, but this was mainly due to a limitation in my CPU power. The training of the Xception model, which took 200 min, only got through five epochs. If I had more time or a more powerful processor, I would have achieved better results through transfer learning. However, given my computational resources, I would choose to use Model 1 above to train a model for this project.

	Accuracy	mean F-1	Running Time
Xception w/ Adam + Image Augmentation)	0.90	0.89	200m

4. Results

The Model 1 (+RMSprops optimization and Image Augmentation) is my final chosen CNN model made from scratch. The code for the model is saved on my Github page.

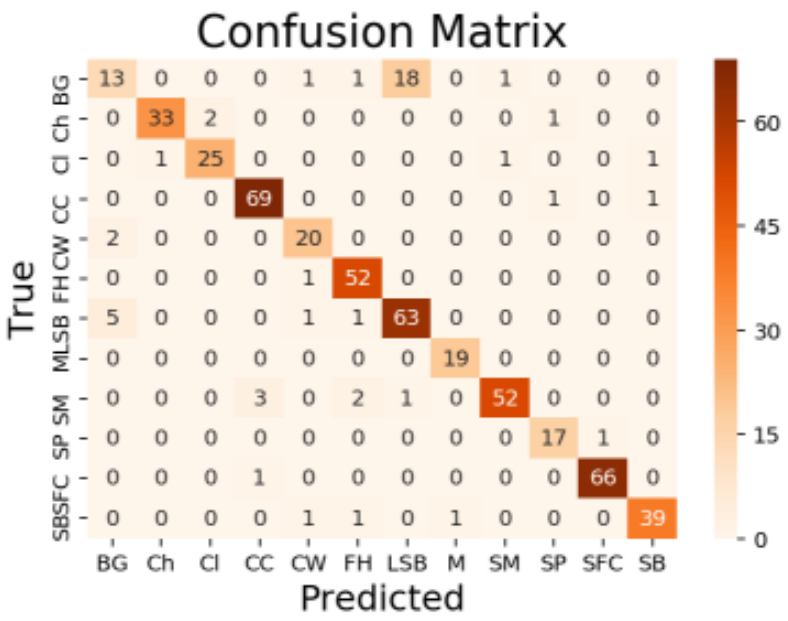
4-1. Model Evaluation and Validation

The table in the previous section has my final accuracy (0.94) and mean F-1 value (0.93) of the model using the test sets. The result is a vast improvement over the first CNN model that I tried, especially considering that the total run time is less than 30 min on my macbook. The inclusion of image augmentation in my final model allows that model to better generalize the training dataset, and therefore to achieve better accuracy in the test dataset.

4-2. Justification

- **Benchmark model** : mean F1 score is 0.34, and accuracy is 0.4
- **Final model** : mean F1 score is 0.93, and accuracy is 0.94

The comparison between my bench mark model and the final model shows that I have more than doubled my accuracy, and the F1 score is nearly triple.



abbreviation	class
BG	Black-grass
Ch	Charlock
Cl	Cleavers
CC	Common Chickweed
CW	Common wheat
FH	Fat Hen
LSB	Loose Silky-bent
M	Maize
SM	Scentless Mayweed
SP	Shepherds Purse
SFC	Small-flowered Cranesbill
SB	Sugar beet

Above I plot the confusion matrix to more closely analyze the results of the model. The x-axis is the predicted class and y-axis is the true class for 10% seedings of overall sample (the test

sample). The diagonal numbers correspond to the cases where the predicted class is matched with the true classes. The sum of each row gives the **TNs** (**True Negatives**), which are called "Negative" although the seedlings are truly identified. The sum of each column gives the **FPs** (**False Positives**), which although they are called "Positive", the seedlings are falsely identified. Thus, Precision and Recall are each 0.93. Based on the definition of mean F-1 score I mentioned in the previous section, I derive 0.93 as mean F-1 score. The equations used to explicitly calculate these are given later in this section.

In order to check the result in detail, I compared TP, FP, TN, Precision, and Recall for each seeding species. The table below shows the results. As you can see, most of the seedling species are relatively well-recognized by the model, with the obvious exception being BG (which is often mis-identified as LSB).

	TP	FP	TN	Precision	Recall
BG	13	7	21	0.65	0.38
Ch	33	1	3	0.97	0.92
CI	25	2	3	0.93	0.89
CC	69	4	2	0.95	0.97
CW	20	4	2	0.83	0.91
FH	52	5	1	0.91	0.98
LSB	63	19	7	0.77	0.9
M	19	1	0	0.95	1
SM	52	2	6	0.96	0.90
SP	17	2	1	0.89	0.94
SFC	66	1	1	0.99	0.99
SB	39	2	3	0.95	0.93
Total	468	50	50		

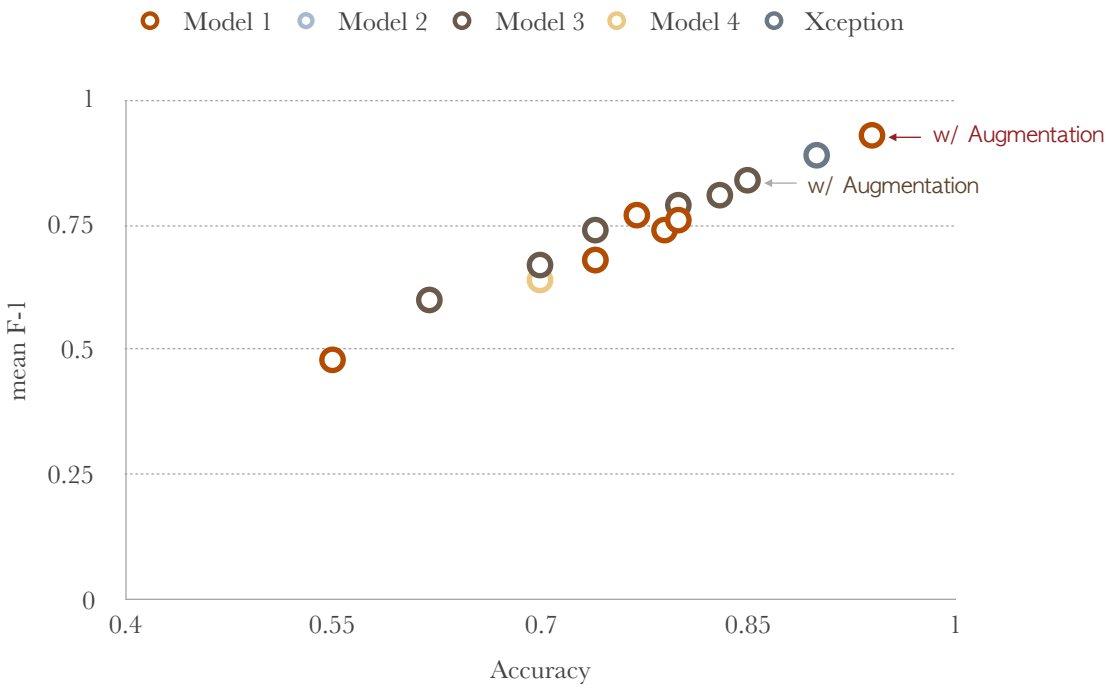
Here, I use the micro-average mean F-1 score, which is given by:
Precision = (TP1+TP2+…+TP12) / (TP1+TP2+……+TP12 + FP1 + FP2 +…. +FP12)
Recall = (TP1+TP2+…+TP12) / (TP1+TP2+…+TP12 + FN1 + FN2 +…. +FN12)
Thus, precision & recall are 0.93, and F-1 score is the same.

5. Conclusion

End-to-End Summary

I tested a variety of different CNN models, using different numbers of filters, sizes of filters, optimization functions, and with or without image augmentation. I also used a pre-trained model (based on Xception) as a comparison to my CNN built from scratch, and I found the results to be slightly better with the pre-trained model. Both of my refined models have significantly better results when compared to the first basic model with three simple convolutional layers. I verified the test score using the leaderboard on Kaggle, so I got the result of 0.93 using their test data.

5-1. Visualization



The figure above shows the F-1 score vs. accuracy for all of the models tested in this project. Different colored symbols indicate the different model tested. This figure highlights the significant improvements made with the variation in the models that were tested. Furthermore, the linear trend results from the fact that F-1 score correlates with accuracy.

5-2. Reflection

One of the biggest challenges for me in the project was the sheer number of free parameters available when building a CNN from scratch. It was hard for me to even know where to start in testing a set of models, and the fact that training each one takes at least 20 minutes only added to the difficulty. However, I came away from this process with a much greater

understanding of what each of these components of a CNN does, and the effects they have on the computational time and on the final qualities of the models. So my take-away from this project will certainly be that the process of testing out all of these models was really valuable for me. I also have a greater appreciation of the value of using Cloud Computing (e.g., AWS) with GPU processors, which would allow me to more rapidly prototype model variations to speed up this process in the future.

5-3. Improvement

(1) When examining the confusion matrix above, the most obvious flaw is the mis-identification between Black-grass (BG) and Loose silky-bent (LSB). The precision and recall for Black-grass is the worst overall, since a large fraction of the BG are considered by the model to be LSB. On the other hand, the precision and recall for LSB are not bad, because I believe the number of input images for LSB could be enough to train the model to predict with at least a 90% accuracy. Thus, the most important step I would take to improve my model would be to include more images, especially for the BG class. At the same time, if we want to use this model to categorize BG, we need to accept that there will be a significant number of FPs and TNs (and a lower precision and recall). If you look back at Fig3, which shows example images of each seedling species, it's apparent that the features of BG are not very easy to see, and that visually they look very similar to the LSB seedlings. This no doubt contributes to the problems that the model has with identifying them.

(2) Cloud : Everything in this project was executed on my laptop, which is not the most efficient way to train models such as these. In the future, I will use GPU computing on an Amazon Web Services EC2 instance, which can be extremely helpful because it will allow me to train more models, and with a greater number of epochs, thanks to a large improvement in speed. On AWS, I could also use larger image input sizes than 47x47. Also, I could test many pre-trained models with different parameters, which could also lead me find a better overall model.

Reference

https://www.kaggle.com/c/plant-seedlings-classification
https://www.tensorflow.org/tutorials/layers
https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/
https://towardsdatascience.com/image-augmentation-for-deep-learning-using-keras-and-histogram-equalization-9329f6ae5085
https://www.kaggle.com/gaborfodor/seedlings-pretrained-keras-models
https://keras.io/applications/
http://cs231n.github.io/convolutional-networks/#overview