Seed Species Classification Using Deep Learning

Daniel Sanche
drs801@mail.usask.ca
University of Saskatchewan
drs801
11123105

ABSTRACT

While vital to many agricultural enterprises, identifying individual seed species can be difficult to distinguish, even for trained experts. At the same time, a misidentified seed species could cause immense environmental harm, in the case of invasive species. Currently. Questionable seeds must often be manually sampled and analyzed by a trained expert. Moving this burden onto automated systems could significantly improve efficiency, allowing more seeds to be analyzed in a shorter time. To explore this task, this paper builds off the work of PhD student Xin Yi, who implemented a species classifier using a novel texture descriptor. This project builds off Xin's work by training a convolutional neural network on his data, utilizing image augmentation techniques to assist in training. After analyzing the results, using image augmentation shows a clear increase in performance, but it is apparent that a more substantial dataset is needed for more significant performance gains.

1. INTRODUCTION

Transportation and trade of seeds is an important part of agricultural economies like Saskatchewan's. Various species of seeds are frequently imported and exported across borders for use on farms. As a part of this process, seeds often must undergo analysis by seed experts, both to verify that the correct seed was received, and to ensure that invasive species are kept out of fragile ecosystems. Currently, this process is commonly done manually by highly trained seed experts, but this is a field where automatic image recognition can increase productivity substantially.

Automated software has many advantages over manual human inspection. First of all, the costs are significantly lower. While an expert has limited availability, and likely charges substantial fees for their time, software has no such limitations. After it has been developed once, it can be deployed instantly to multiple locations, and can be run with almost no direct expense.

Replacing human experts with machines not only saves money, but improves throughput as well. The perpetual uptime and parallel nature of automated approaches allow machines to process many more seeds than an expert could feasibly look at manually. Instead of performing analysis on a random sample from a shipment of seeds, the seeds could conceivably all be run through a machine with a camera, in which every seed is quickly analyzed by an algorithm to ensure is no cross-contamination. In this way, lowering the costs associated with analyzing seeds can cause a significant increase in productivity, and improve quality standards as a whole.

The task of seed recognition falls under the more general topic of image classification, in which a program takes in an image as input, and attempts to assign a class label to it from a limited set of potential labels. This was the primary focus of the PhD thesis of Xin Yi, from IMG lab at the University of Saskatchewan. [1] Xin focused on creating a novel texture descriptor that can be used to determine the correct species for each seed. Using his classification method, Xin was able to classify a set of high-quality images with a 95% success rate. When it came time to test his method in a real-world situation in his user study, however, the overall success rate dropped substantially, to ~45%. When he ran a convolutional neural network (CNN) model on the same dataset, he was able to achieve within 65-75% accuracy. This led Xin to speculate that errors in lighting, or movements of the seed between images could have caused issues in his classifier. A CNN approach would be expected to be more resilient to these issues, so that would explain his results.

The CNN used by Xin, however, was very simple. He simply took a pre-existing network that was trained on ImageNet, and replaced the output layer with one trained on his seed images. In theory, it would be expected that he could achieve higher performance if he used a network that was trained directly on seed images instead. Additionally, Xin trained his output layer on a small set of high quality seed images, which had vastly different properties from the final user study images. If a more representative dataset were used in the training stage, that may also increase the network's performance. While Xin's basic CNN could achieve relatively high classification rates, the full potential of a well-trained CNN was unknown. Finding the limits of such an approach was the goal of my project.

In order to improve the training dataset, I created a set of python scripts to augment the original high-quality images. The scripts would read in the original images in sequence, and then apply modifications such as mirroring, noise application, rotation, and scaling in order to create variation in the images. By applying these transformations to the images, I was attempting to make the training set better represent the user study testing dataset. Using these techniques, the original dataset of 11,026 images was expanded to 100,000, which were then used to train the network.

To avoid the issue of having to train a neural network from scratch, I used a pre-trained model of VGG-16, [2] which is similar to the CNN used in Xin's study. This network was originally trained on the ImageNet dataset, which is made up of 10,000,000 images depicting over 10,000 classes. [3] I then fine-tuned the network for 4 epochs on my augmented image dataset, to allow it to learn seed-specific features.

The network was evaluated by running it on the user study dataset used by Xin in his thesis. Like in Xin's paper, the performance of my network was measured using the top-3 metric, in which the identification is deemed a success if the correct class was one of the network's top 3 guesses. In this way, the results can be directly compared to Xin's method, in order to see exactly how much of an impact training and data augmentation has on the overall classification performance. Additionally, I also trained the network directly on the source dataset without applying augmentations in order to create another baseline for evaluation.

2. Related Work

2.1 Seed Classification

The most important reference work used for this project was Xin Yi's thesis, titled "*Plant Seed Identification*". [1] Xin's thesis goes into detail about image classification under the domain of seed images. In his paper, Xin describes a novel texture descriptor he created for this task. The texture descriptor was based on a bag-of-words model, in which key points in the seed images were found at different scales, and pooled together to form a single, unified feature vector that can describe the seed. From there, the feature vectors were used to train a support vector machine (SVM) classifier to identify the associated class label. Additionally, Xin used a simple CNN model, to see how his texture descriptor would perform compared to a deep learning model. The CNN consisted of a VGG-19 [2] network trained on ImageNet, in which the last layer was dropped and replaced with an SVM layer that was trained on his seed images.

2.2 Neural Networks

This paper is also influenced by research in the field of deep learning, a sub-field of machine learning. Deep learning concerns algorithms that learn to solve problems using artificial neural networks with many layers. The math behind artificial neural networks goes back as far as the 1950s, to the work of Frank Rosenblatt. In 1958, Rosenblatt developed what he called a perceptron, which is a basic piece of hardware meant to simulate a single neuron. [4] Rosenblatt's perceptron takes in a set of binary inputs, computes a weighted sum of the inputs, and then thresholds the value, to result in either a 0 or a 1. The perceptron could learn to classify simple inputs using a basic learning algorithm, in which the weights would be adjusted until the perceptron output the expected value for all items in a training set. Rosenblatt's work caused a great deal of excitement among academics, but it soon failed to live up to expectations, and fell out of fashion.

It wasn't until 1986 that neural networks regained academic interest due to the emergence of the backpropagation algorithm. [5] Backpropagation works by calculating the output of a network for some input, calculating an error score, and then working backwards to find out how the network's weights can be adjusted to lower the error score. Using backpropagation, it became feasible to train larger networks of perceptrons, paving the way for the field of deep learning.

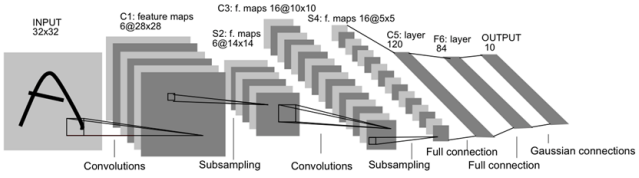

Figure 1: Diagram of Le-Net5, one of the first convolutional neural networks designed by Yann LeCun et al. Most modern architectures are still very similar

In 1989, Yann LeCun and his team at AT&T Bell Labs used backpropagation to produce the first convolutional neural network (CNN). [6] LeCun's network was capable of automatically classifying small images of handwritten digits for the first time. While a neuron from a typical neural network has a distinct weight for each input from the previous layer, a convolutional layer has a small number of weights that are interpreted as a convolutional kernel. The kernel then slides over the previous input layer to produce a new output layer. These networks can be much easier to train than standard neural networks, because the number of weights is severely limited. Additionally, the convolutional layers give the network a spatial invariance effect; the kernels will activate when they recognize their target, no matter where it is in the input image. This feature makes CNNs very effective for image processing tasks. In addition to convolutional layers, modern CNNs typically use pooling layers, which shrink the input size to prevent over-fitting, and standard feed-forward layers, which can be used to produce a vector output at the end of the network.

The neural network architecture used for this project specifically is VGG-16. [2] VGG-16 is a network optimized for large-scale image recognition, in which millions of images are used to train the network. To achieve high performance in this task, VGG-16 uses 16 layers, with a kernel size of 3x3 pixels. Using this architecture, VGG-16 was able to achieve a 7.4% top-5 error on the 2012 ImageNet Large Scale Visual Recognition Challenge. Although VGG-16 is no longer the current state of the art, its relative simplicity and easy availability has made it very useful for fine-tuning applications like this one.

2.3 Image Augmentation

The need for large datasets has always been an issue in training complex deep learning models. Because of this, augmenting the dataset with modified samples has been standard practice. [8] Using image augmentation techniques helps to prevent the network from over-fitting the dataset, which should in theory lead to a performance increase. Typically, image augmentation involves transformations such as cropping, mirroring, rotation, and adding noise to the images. [9]

Recently, some researchers have attempted to take augmentation a step further, by synthesizing new images rather than applying simple transformations. While standard augmentation functions focus on using pixel-level transformations that could be applied to any image, these new techniques involve exploiting existing knowledge about the dataset to generate more meaningful samples. For example, one group of researchers attempted to synthesize new images of faces for their dataset by manipulating facial features in the images already in their dataset. [9] Although this is an exciting development, this topic is outside the scope of this project.

3. Datasets

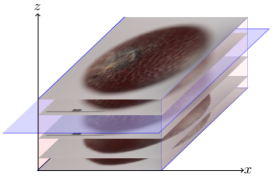

Figure 2: Each seed was photographed from 50-80 focal levels

This project makes use of three major datasets: the source images, the augmented images, and the testing images. The source images were provided to Xin by the Canadian National Seed Herbarium. While Xin's original dataset contained 30 species, he trimmed it down to 15 for his user study, so this subset was used for my analysis. This source image dataset was created by photographing 150 seeds representing the 15 different species. 10 seeds were used for each species in order to capture biological variation within each class. Each image in this dataset has a resolution of 700x640 pixels.

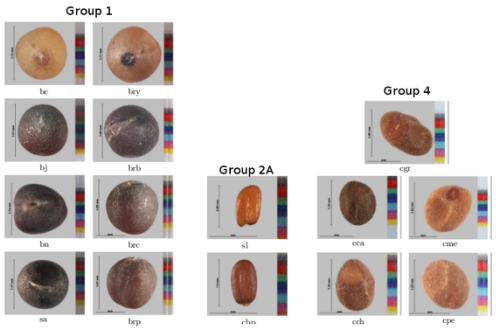

Figure 3: Sample images used in source dataset. For clarity, segmented images were chosen, but some images in the dataset had background artifacts in the image

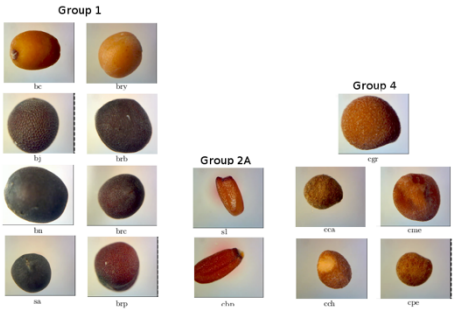

Figure 4: Sample images used in testing dataset

These images were acquired using an AZ100M motorized Multi-Purpose Zoom Microscope. [1] Each seed was placed on the microscope's glass slide and photographed at 50-80 different focal levels, resulting in 11,026 images in total. In Xin's project, these individual focal images were stacked together to form a single image, showing a perspective of the seed that is always in focus. For my project, I treated each focal image as an individual sample image of the seed.

The seed species used for classification can be further broken down into 3 groups: Group 1 contains images from the *Brassicaceae* family, which are round seeds with textured patterns. There are 8 species belonging to this group used in the dataset. Group 2B contains small pill-shaped seeds, of which only 2 were represented in the dataset. Finally, Group 4 contains seeds from the *Cuscata* family, which are more misshapen than other seeds mentioned, and have 5 different species in the dataset. Overall, the seed groups all look very distinct from each other, but determining the specific seed species within the group can be a challenging task.

Group	Species Name	Abbreviation
1	*Brassica corinata*	*bc*
1	Brassica junceai	*bj*
1	*Brassica napus*	*bn*
1	Sinapis arvensis	*sa*
1	Brassica rapa (yellow)	*bry*
1	Brassica rapa (brown)	*brb*
1	Brassica rapa (chinensis)	*brc*
1	Brassica rapa (pekinensis)	*brp*
2B	Sisymbrium loeselii	*sl*
2B	Capsella bursa pastoris	*cbp*
4	Cuscuta campestris	*cca*
4	Cuscuta chinensis	*cch*
4	Cuscuta gronovii	*cgr*
4	Cuscuta megalocarpa	*cme*
4	Cuscuta pentagona	*cpe*

Table 1: List of seed species represented in the dataset

The augmented dataset was generated from the source images using a set of augmentation scripts I developed. This dataset consists of 100,000 images, and was used to train the neural network. The exact augmentation algorithm used will be described in a later section. The images in this dataset were resized to have a resolution of 224x224 pixels, because that is the resolution required for the VGG16 network. [2] As the dataset was being generated, it was ensured that the distribution of species classes was as close to equal as possible to prevent the network from learning to bias certain classes.

Finally, I used a separate dataset of testing images to evaluate the network. This dataset contains the images acquired in the user study portion of Xin's thesis. In Xin's user study, seeds were placed under a microscope, which was used to take photographs of the samples. Like the source image set, the microscope took various images of the seeds at multiple focal levels. Unlike the source image set, there is much more noise and variation in the testing dataset. The environment was much less controlled, with different light sources, background colors, and seed perspectives. This dataset is more representative of what the seeds would look like in a real-world use case, which makes it a good target to evaluate the network against. The images in this dataset were originally 640x480 pixels, but they were cropped and resized to 224x224 for input into the neural network. This dataset contained 12,752 images total, with a varying number of images coming from each species.

4. Augmentation Algorithm

The augmentation algorithm is responsible for taking in an input image from the source dataset, modifying it with various transformations, and saving is as a new image in the augmented image dataset. The functionality for this task is contained in a script called *DataAugmentation.py*. This script contains a number of functions to apply various transformations to the image.

The first three functions contained in the script are basic image filters that are applied at the pixel level. First, there's a function to add random Gaussian noise to the image, parameterized by the noise's mean and variance. Next, there's a function to apply a gamma transformation to one or more of the color channels. Finally, there's a function to adjust the contrast in an image, parameterized by the new mean intensity, and an intensity range. These functions are meant to simulate different possible camera sensors, which would capture different image.

The next set of augmentation functions are provided to simulate different perspectives of the seed itself. The simplest of these transformations is mirroring, in which the pixels in the image are flipped either horizontally or vertically. The rotation function is also relatively simple, rotating the pixels around the center point. The complication with this function is that the pixels in the corners of the resulting image may need to be interpolated. This is a relatively simple task with this dataset, however, because there is

typically nothing of importance in this region of the image anyway. The next function is translation, in which the seed is shifted across the image. This is accomplished by cropping one edge, and adding an equal number of pixels to the opposite edge. Again, the added pixels are usually unnoticeable, because the seed itself is left unaffected. The shrinking function is meant to simulate taking photographs from different distances. In this function, the entire image is scaled to a smaller size, and padding pixels are added to the edges. By shrinking the relative size of the seed in the image, it creates the illusion that the image was taken from a distance. The padding pixels are created by replicating the pixels that were on the edge of the original image. Finally, the lighting function simulates different light sources on the image. This effect was implemented by creating a radial gradient mask that's the same size as the image, and multiplying each color channel in the original image with the gradient mask. This function has parameters to determine the center point of the gradient and the gradient's radius, to simulate different lighting sources.

Finally, I added a function responsible for segmenting the seed off of its background. This was deemed necessary because many seed images had recognizable artifacts in the background. If some of these background features were more prominent in some classes than others, the neural network may have ended up learning to recognize these correlations instead of learning useful features. For this reason, I added a segmentation function that can replace the seed's background with a random color.

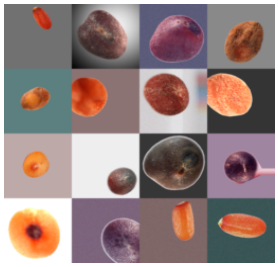

Figure 5: random sample of images in augmented dataset

To bring these functions together, I also created a *ModifyImage* function, that would take in a single training image, run it through the previously discussed functions, and output an augmented image. For each available helper function, *ModifyImage* takes in a 2-tuple for each parameter, representing the range of valid values for that argument. *ModifyImage* then randomly chooses a value within that range, and uses the random value as the argument for the function. Some functions, such as rotation, draw their parameter values from a uniform distribution. Other functions like translation, however, draw their parameters from a normal distribution, ensuring small movements are more likely than large ones. In this way, each augmented image will have slightly different features, drawn out of a known distribution. Additionally, many functions also have a probability parameter, and are only applied if a random float is larger than that value. In this way, not all transformations are applied to every augmented image, providing more variation in the dataset. Finally, *ModifyImage* also produces a *metadata.csv* file during the course of its operation, which logs information about each transformation function that was applied to each image in the dataset.

5. Neural Network Description

For this project, I used a pre-trained VGG-16 model to avoid the computation time that would be required from training a custom network. The weights for VGG-16 were obtained from a GitHub page, in which the network used in the VGG project was converted into Keras. [9] From there, I fine-tuned the weights on our augmented image dataset to increase performance.

VGG-16 is a convolutional neural network originally designed for the Large Scale Visual Recognition Challenge 2012 (ILSVRC-2012), and trained on the ImageNet dataset. ImageNet contains images scraped from the public Internet, and hand-annotated into 1,000 object classes. [3] These classes contain a large variety of objects; from animals, to instruments, to furniture. When originally trained on this ImageNet, VGG-16 was able to obtain a 7.4% top-5 error rate. Because the network was trained extensively on this dataset, it should in theory have learned many useful, generalizable features that can be applied to other image domains. For this reason, adapting the model to work on seeds would be much faster than training a new one from scratch.

VGG-16 contains 16 weighted layers in total: 13 convolutional layers, followed by 3 feed-forward layers. Additionally, there are 5 max-pooling layers, but these have no associated weights to learn. VGG-16 closely resembles VGG-19, which was used by Xin in his thesis. The only difference between the two is that VGG-19 contains 3 extra convolutional layers. VGG-16 was chosen for my project because it's weights were readily available for Keras, my chosen deep learning framework.

Every internal layer in the VGG-16 network is followed by a rectified linear unit (ReLU) as its non-linearity function. ReLU was chosen because this it has become the standard in recent years to prevent the vanishing gradient problem. [10] ReLU layers work by taking in any floating-point value input, and thresholding the value at 0; any negative numbers are changed to 0, while any positive numbers are left unchanged. Sigmoid functions were originally used as non-linearity functions in the past, but ReLU layers have been shown to be more desirable in practice.

While ReLU works great for internal layers, the output layer of VGG-16 uses a different non-linearity function: softmax. Softmax is simply a generalization of the sigmoid function; while sigmoid will convert any floating-point value to a new value between 0 and 1, softmax will take in a vector of numbers, and return a new vector such that the sum of the values in the vector adds up to 1. Using softmax, each neuron in the output layer represents one of the 15 classes, and each neuron will hold a value between 0 and 1, where the sums of all values add up to 1. In effect, these values can be interpreted as probabilities that each class is the appropriate label for the input image.

Python was the primary language used in this project due to its many useful deep learning frameworks, including Keras and TensorFlow. TensorFlow is a machine learning library that interfaces with Python. [10] TensorFlow works by adding mathematical operations, or "tensors" to an internal directed acyclic graph (DAG) that represents the entire model. In this way, an entire neural network can be built by connecting pieces that represent layers into a DAG, and then running data through the graph. Additionally, TensorFlow includes powerful functions for many deep learning tasks, such as training the model to maximize an objective function.

While TensorFlow is a powerful framework, it can be tedious to set up and train networks. For this reason, I also used Keras. [11] Keras is a high-level machine learning framework that sits on top of TensorFlow, and abstracts its complexities into a set of simple

APIs. Using Keras, it is trivial to get a standard neural network architecture up and running using very few lines of code.

Because I used a pre-trained network rather than designing a custom one, there were few parameters to the network available for me to adjust. One part of the network I was able to modify was the training function, however. For this project, I chose to use Keras' standard stochastic gradient descent (SGD) function, with a learning rate of 0.0001. Stochastic gradient descent works by calculating an error function, determining the partial derivatives of that function with respect to each weight in the network, and adjusting the weights in the direction of its derivative, to slowly decrease the error value over time. While Keras provides the option to modify momentum and decay values during training in order to modify the learning rate changes over time, I left these parameters to the default of 0 for simplicity. The network was trained for 4 epochs through the augmented image dataset. At the end of the 4th epoch, the accuracy on the training set was over 99% for the top-1 error, indicating that the network was unlikely to improve without introducing more training data.

6. Results

The performance of the network during the training process can be seen in Figure 6. The top-1, top-3, and top-5 metrics have been tracked for the network through each training epoch. This chart shows that the majority of the training takes place within the first epoch, which is unsurprising due to the large size of the training dataset. From there, the performance modestly increases until after 4 epochs, at which point the classification rate dips slightly, and then appears to plateau. From this chart, I found that 4 training epochs was sufficient to properly train the network, so the weights at this point were used for the rest of the analysis. At epoch 4, the top-1 result was at approximately 44%, while the top-3 result was 69% and the top-5 result was 78%.

The accuracy of the network can be clearly analyzed by breaking down the accuracy for each individual species, as seen in figure 7. This chart shows the top-1, top-3, and top-5 accuracies for each seed species as different shades of blue. Analyzing this chart makes it apparent that some species perform much better than others. As mentioned previously, the overall top-3 performance for all classes is 69%. When broken down for each species, the highest score is 100% (for *bj*, *sl*, and *cbp*), and the lowest is 6% (for *brb*). The standard deviation between all species is 30%.

The results can be further broken down into a covariance matrix, as seen in Table 2. Covariance matrices are an efficient way to describe the top-1 rankings of the network, in such a way that it is easy to analyze where the errors are occurring. The columns in the matrix represent classes assigned to the images in the dataset, and each row indicates how often that column's class was predicted to be the row's class. For example, the cell in column "*sa*" under column "*bn*" tells us how many times the seed "*sa*" was mistaken for the seed "*bn*". An ideal covariance matrix would be a diagonal identity matrix, in which each class is always properly identified.

In addition to the tabular covariance matrix, I have generated a more visual representation in Figure 8. This plot helps to visualize the covariance matrix in a more obvious way. Each vertical bar represents a column of the covariance matrix, broken down and colour coded. In this chart, all seeds in Group 1 are coloured shades of blue, those in Group 2B are shades of red, and those in Group 4 are shades of green. In this way, it is clear how often seeds are mistaken for other seeds in the same group, and how often they are confused with other groups.

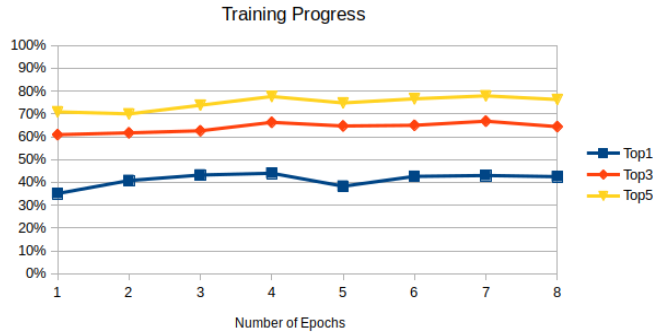

Figure 6: Network performance through the training process.

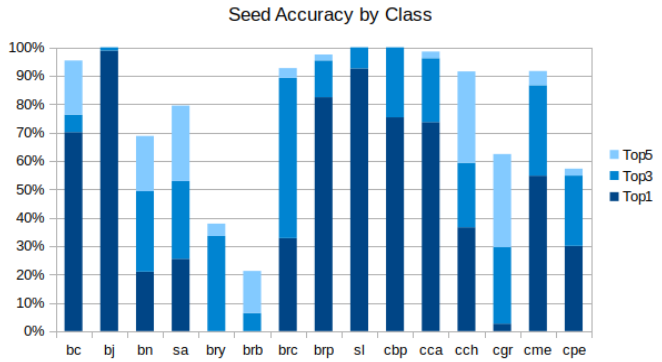

Figure 7: Training accuracy for each species class

	bc	bj	bn	sa	bry	brb	brc	brp	sl	cbp	cca	cch	cgr	cme	cpe
bc	667				609										2
bj		890		6		47	38	13			1				
bn			169	450				27							
sa															
bry				225											
brb			11	156				15							
brc			231	38		150	332	45			17				
brp			343	7	49	700	630	752			23	106	31	57	20
sl	6				3				613						
cbp									43	358		98	228		25
cca		10	53	1		192	13	55			599	189	23		442
cch	278				244	1		5			77	259	373	221	48
cgr					14								23	58	23
cme									6	117		6	148	460	67
cpe											96	50	57	44	269

Table 2: Covariance matrix of the classifier. Columns represent the true class, rows represent the predicted class

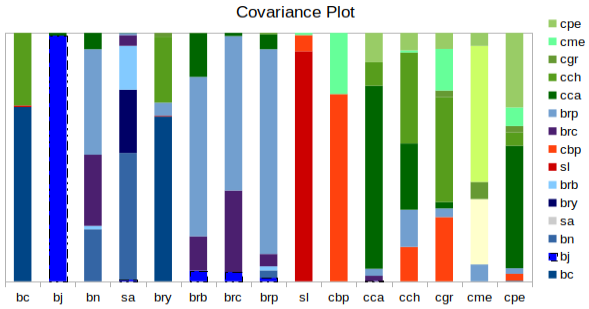

Figure 8: Covariance matrix represented as a bar graph. Each column shows the distribution of predictions for seeds of that type

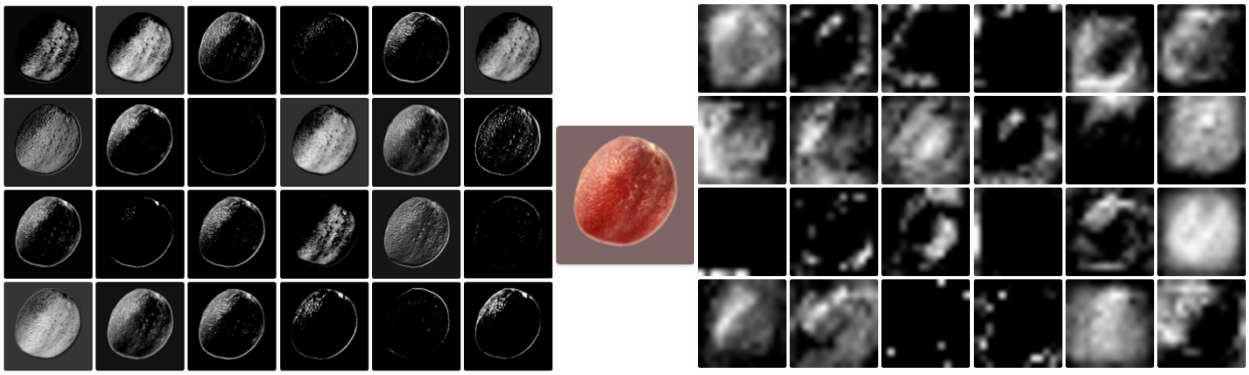

Figure 9: Network activations in the 1st convolutional layer

Figure 10: Network activations in the 13th convolutional layer

Unfortunately, despite the additional training, the performance of the network is roughly comparable to those achieved by Xin on the same dataset. The VGG-19 network used in Xin's thesis showed results with a margin of error ranging between 65-80%, which puts it right in the same range as the 69% reported by my network.

To further investigate this issue, I trained the network directly on the source image dataset, rather than the augmented images. This network was trained for 6 epochs, due to the fact that the source dataset was much smaller than the augmented dataset. After training was complete, it reported a 56% top-3 accuracy. This shows that although augmenting the images in the dataset lead to a 13% performance increase over the raw source images, surprisingly, it seems that the network still performed worse than one that was trained only on the general ImageNet dataset. This could be due to over-fitting; it's possible that training on the seed images caused the network to lose some more general features, that ended up being useful when used with the testing dataset. Alternatively, it's possible that some other factors in Xin's experiment caused him to achieve higher results. Perhaps differences between the VGG-16 and VGG-19 architectures can explain the discrepancy.

A visualization of the weights learned by the network can be seen in Figures 9 and 10. These visualizations were created using Quiver, [13] a Python framework created to visualize CNN architectures constructed in Keras. Recall that CNNs are built out of layers, in which each layer contains a set of many learned kernel filters. These filters take in input images from the previous layer in the network, and produce activation maps based on features they detect in the image. Quiver can visualize these filters by producing a gray-scale image, in which the value at each pixel represents how strongly the filter responds to the corresponding pixel in the original input image. The seed image in between Figures 9 and 10 represents an input image that is fed through the network. Figure 9 represents a subset of filters stored in the very first convolutional layer of the network, while Figure 10 shows a subset of the filters in the final convolutional layer.

This visualization helps to demonstrate the differences between the filters in early layers, from those in later layers. The earlier filters shown in the Figure 9 are much simpler. They represent basic image processing operations, such as edge detectors or thresholding functions. The later filters shown in the Figure 10 are made up of many of those simple filters stacked on top of each other, combined with pooling and ReLU operations. These filters are much more messy and complex. If the network was trained on a dataset like ImageNet, these filters might activate for high-level features, such as eyes on a face or wheels on a car. Because seeds are relatively simple objects, the filters in this case seem to activate primarily for lighting and texture features.

7. Discussion

The most surprising result of this project was that the network achieved very similar accuracies to the one used by Xin, even though his network had no training on seed images, and used a very small training set. This result makes more sense after seeing the "t-distributed stochastic neighbor embedding" (t-SNE) visualization of Xin's network, included in Figure 11. This figure attempts to project the 4,096-dimension feature vector from the final layer of the CNN into a 2D space, so it can be easily visualized by humans. In a t-SNE plot, the further any two points are from one another other on the 2D plane, the easier it is for the classifier to tell their classes apart. In Xin's figure, dots are the testing set of images, and triangles are the training images.

Each triangle (representing a testing image) is likely to be assigned the class of the dot (training image) that appears closest to it in the plot. When viewing Xin's figure, it is clear that his ImageNet-trained network still found many of the same clusterings as mine, which was trained on seeds. For example, Xin's t-SNE shows the seed class *bry* to have its testing samples separated from the training samples by a cluster representing *bc*; in my covariance matrix, *bry* was never correctly identified, and it was often mistaken with *bc*. Additionally, in my covariance matrix, *brb* is often mistaken with *brp*, *brc*, or *cca*; in Xin's t-SNE, *brb* is located in a space in between those three classes.

This analysis of Xin's t-SNE helps to demonstrate that although my network was trained on hundreds of thousands of images of seeds, it is still primarily using the same general ImageNet features that it started with. The primary explanation for this is that there was no incentive to learn new filters because of the homogeneity of the training dataset. My network was able to achieve over 99% top-1 accuracy on the training dataset after 4 epochs using primarily the ImageNet features, so there was no incentive for it to learn more seed-specific features, even if they would have helped on the testing set. Introducing new data to the training set may have allowed the network to learn new useful features, because it would have more samples to learn from. Training the network from scratch may also have assisted in this goal, but that too would require more training images.

In hindsight, it should come as no surprise that a small training dataset was the limiting factor to improving performance. Although my network trained on 100,000 images, these came from a source dataset of 11,026 images. From these 11,026, the majority are

simply different focal slices from a smaller set of only 150 different seeds. In total, only 10 seeds for each species were used to generate the entire dataset. With a dataset this small, it should be no surprise that the network was unable to fully capture all the biological diversity possible for each species. It is very likely that the user study images contained seeds with slightly different shapes or textures than the classifier had encountered before, which caused it to make erroneous predictions.

This last point gets at the core of this project's findings: although the image augmentation functions show a clear benefit in my testing over an un-augmented dataset, there are limits as to what augmentation can accomplish. Augmentation can compensate for different camera settings, like noise, viewing angle, and lighting, but it will do little to help if the dataset doesn't comprehensively contain all the natural variation you may encounter in the wild. If the seeds in the testing dataset are significantly different from those in the training set, data augmentation alone won't solve the problem

This project demonstrates that the results Xin achieved with his simple CNN were likely close to the performance ceiling for the datasets he used. In order to achieve higher scores on the user study images, he would likely need to acquire more seed images to train on. Additionally, this demonstrates a possible reason as to why his texture descriptor model had such a large drop in performance: it is likely that it too would need more texture samples to properly capture the seed variations.

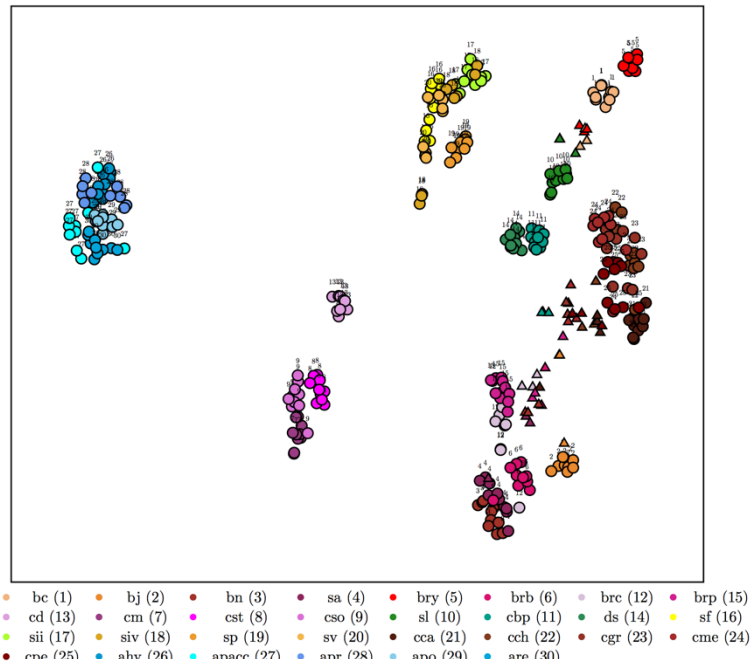

Figure 11: t-SNE visualization of Xin's VGG-19 Network. Circles represent training images, triangles represent testing images

8. Future Work

As mentioned previously, in order to improve performance, it is likely that more source images are required. The yellow and brown *Brassica rapa* (*bry/brb*) classes in particular appear to need more images, as demonstrated by the fact that both seeds had a 0% top-1 accuracy in my network on the testing dataset. This is evidence that a limited dataset is the primary bottleneck to performance at this time.

In addition to acquiring more data, there are a number of additional augmentations that could be added to potentially improve performance. One potential augmentation would be to apply distortions to the seed shapes. Currently, the augmentation algorithm can rotate, scale, and transform seeds around an image, but the overall shape boundary stays consistent. It is possible that the network is learning shape features in its training, which could be a source of over-fitting. If the shape of the seeds in the augmented dataset were distorted, it may cause the network to focus more on textures and other features which would be more consistent in real seeds.

Another potential augmentation would be to sample ranges from the image focal stacks, rather than using single slices. As described previously, the seed datasets were generated by taking 50-80 image of the seed with different focal levels, and these images can then be combined into a focal stack. For my project, I simply used each focal level as an individual input image. One idea that we had discussed prior to the project was to merge different focal images together. This would help to simulate cameras with different depth-of-field properties. This feature was scrapped due to time constraints, and because we found that the increased focal range didn't accurately represent the images contained in our testing dataset.

Along with adding new augmentations, it may be worthwhile to experiment with different parameters to the augmentation algorithm. I didn't have the time to extensively experiment with the parameters, so I found a setting that seemed representative to what one would expect out of typical consumer cameras, and stuck with those settings for the entirety of the project. It may be worthwhile to extensively experiment with different settings to see if any augmentations had a larger impact on the results of the network. In the end, however, it is likely that the augmentations had little or no impact on the actual results, so this is likely not very important without a larger source dataset anyway.

Using larger images is one change that could have a significant impact on the results. My project used 224x224 images, because that is the required image size for the VGG-16 network. These images are relatively small, however. It is likely that any small, intricate texture patterns would be imperceptible at that scale, but may be visible in higher resolution images. My results showed that the features learned by the network were no better than the generic ones learned on ImageNet, but that may not be the case if the network had larger images to gather features from.

The reason the image sizes weren't increased for my project was because that would require training the network from scratch, which is another area for potential future work. The network used in the project was a standard VGG-16 model, that was fine-tuned to the seed dataset. Because the original weights were created for a specific network architecture, any changes to the network would cause them to be unusable. If the network were trained from scratch, however, there would be no limitations on customizations. We would be free to add or remove layers or change the image size to see how that impacted our results. This freedom comes at a cost, however; in order to train a network from scratch, we would need significantly more training time or more powerful hardware, which was out of scope for this project.

9. Conclusion

This project has demonstrated both the advantages of augmenting images in a dataset, and the limitations of such an approach. Generating images with simple transformations can help prevent over-fitting on the training dataset, as is demonstrated by

comparing my results between the network trained on augmented images and the network trained on the source images alone. At the same time, augmenting images will not make up for a lack of representation in the original dataset. These results show that in order to train a classifier that will reach higher accuracies on Xin's user study images, more data for each seed species must be collected. With a larger dataset, more powerful models can be created. Using these more powerful classifiers, seeds can be quickly analyzed in an automated setting, freeing up the time of experts, and allowing better scrutiny of the seed when needed.

10. REFERENCES

[1] Yi, X. *Plant Seed Identification.* Unpublished thesis, University of Saskatchewan, Saskatoon, SK, Canada, 2016.

[2] Karen Simonyan and Andrew Zisserman. *Very deep convolutional networks for large-scale image recognition.* arXiv preprint arXiv:1409.1556, 2014.

[3] Russakovsky, O., Deng, J., Su, H. Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A.C., Fei-Fei, L. *ImageNet Large Scale Visual Recognition Challenge.* IJCV, 2015.

[4] Rosenblatt, F. *The perceptron: A probabilistic model for information storage and organization in the brain.* Psychological Review, Vol 65(6), Nov 1958, 386-408

[5] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning representations by back-propagating errors. Nature, 323, 533–536, 1986

[6] LeCun, Y, Boser, B, Denker, J, Henderson, D, Howard, R, Hubbard, W, Jackel, L. *Backpropagation Applied to Handwritten Zip Code Recognition.* Neural Computation, vol.1, no.4, pp.541-551, Dec. 1989

[7] K. Chatfield, K. Simonyan, A. Vedaldi, and A. Zisserman. *Return of the devil in the details: Delving deep into convolutional nets*. *Proc. British Mach. Vision Conf, 2014*.

[8] Masi, I, Tran, A.T., Leksut, J.T., Hassner, T., Medioni, G.G. *Do We Really Need to Collect Millions of Faces for Effective Face Recognition?* 2016. Retrieved March 30, 2016, from arXiv: https://arxiv.org/abs/1603.07057

[9] baraldilorenzo. 2015. GitHub Gist, Retrieved March 30, 2016, from http://gist.github.com/baraldilorenzo/07d7802847aaad0a35d3

[10] X. Glorot, A. Bordes, and Y. Bengio. *Deep Sparse Rectifier Neural Networks.* JMLR W&CP: Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, volume 15, pp. 315–323, 2011.

[11] Abadi, M. Ashish, A., Barham, P., et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed System. 2015.* Retrieved March 30, 2016, from http://download.TensorFlow.org/paper/whitepaper2015.pdf

[12] Chollet, F. 2015. *Keras*. GitHub Repository. Retrieved March 30, 2016, from https://github.com/fchollet/keras

[13] Bian, J. 2016. *Quiver*. GitHub Repository Retrieved March 30, 2016, from https://github.com/keplr-io/quiver