Machine Learning Engineer Nanodegree

Capstone Project

Chris Tharpe
October 14, 2017

I. Definition

Project Overview

This project was based on a leaf classification competition from the Kaggle website [1]. The objective of the competition was to determine the species of each member of a set of leaves when given black-and-white images of the leaves and numerical data about the leaves.

Ninety-nine different species were represented in the dataset. A total of 1584 individual examples were included in the data. The data was subdivided into a test set and a training set. The training set of 990 of examples included the correct species classification. The test set of 594 of examples did not include their species classification.

The original source of the data was the following paper:

Charles Mallah, James Cope, James Orwell. *Plant Leaf Classification Using Probabilistic Integration of Shape, Texture and Margin Features*. Signal Processing, Pattern Recognition and Applications, in press. 2013 [2].

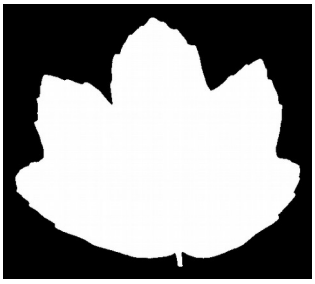
Some examples of the species images supplied, for the species Acer Opalus, Pterocarya Stenoptera, and Quercus Hartwissiana are shown below:

Numerical data about the species included features describing attributes such as the overall shape of the leaf, margin, and leaf texture. A small sample of the numerical data is shown on the following page:

species	margin1	margin2	margin3	margin4	margin5	margin6	margin7	margin8
Acer_Opalus	0.007812	0.023438	0.023438	0.003906	0.011719	0.009766	0.027344	0
Pterocarya_Stenoptera	0.005859	0	0.03125	0.015625	0.025391	0.001953	0.019531	0
Quercus_Hartwissiana	0.005859	0.009766	0.019531	0.007812	0.003906	0.005859	0.068359	0

Problem Statement

The problem objective was to correctly identify the species of the 594 leaves in the test set for which the species information was not provided. Generally, this is done by examining the training set, extracting characteristics of the training set, associating those characteristics with particular species, then applying the same extraction-association process to the test set. For example, item 1 in the test set is an example of the species Acer Opalus. An examination of the image for item 1 shows that the leaf is approximately as wide as it is tall and has five lobes. Looking through the test images, the first image that has those characteristics (hight approximately equal to width, 5 lobes) is item 16. Item 16 is probably the same species as item 1.



Item 1

Item 16

Instead of doing this process manually, a set of machine-learning techniques was used. These techniques were be:

- a Convolutional Neural Network (CNN), which was trained only on the images,
- an multimodal Convolutional Neural Network, which was trained on the images and the numerical data,
- a K-nearest neighbor (KNN) classifier, which will operated only on the numerical data, and
- a backpropagation neural network (NN), which was trained only on the numerical data.

Metrics

Model evaluation was done by submitting a comma separated value (csv) file containing a list of the test set id numbers, a row of species names, and a probability that a particular id is a member of a species to the Kaggle website, for example:

```
id, Acer_Capillipes, Acer_Circinatum, Acer_Mono, Acer_Opalas,...
4 ,        0.0       ,        0.2       ,      0.8     ,     0.0       ,...
7 ,        0.6       ,        0.1       ,      0.0     ,     0.0       ,...
9 ,        0.0       ,        0.7       ,      0.1     ,     0.0       ,...
...
```

After submitting the file, Kaggle evaluated the predictions and returned a logarithmic loss value. The logarithmic loss is defined by Kaggle as:

$$logloss = \frac{-1}{N} \sum_{i=1}^{N} \sum_{j=1}^{M} y_{ij} \log(p_{ij})$$

where N is the number of images in the test set, M is the number of species labels, log is the natural logarithm, yij is 1 if observation i is in class j and 0 otherwise, and pij is the predicted probability that observation i belongs to class j. [3]

The logloss will be zero if all predictions are correct.

Because the natural log of zero is not defined, Kaggle applies a lower limit to the probability estimates of 10 ^ -15. This allows for computation of a percentage accuracy in addition to the logloss value. By submitting a one-hot file (where the highest probability prediction for a test id is set to 1.0, and all other predictions are set to 0.0), each incorrect prediction will add to the logloss by the amount -log(10 ^ -15) / N = 34.5387764 / 594 = 0.058146088. The total number of incorrect predictions can be calculated by dividing the resulting logloss error by 0.058146088. Subtracting the number of incorrect predictions from N then dividing the result by N gives the prediction accuracy. For example, if a one-hot file was submitted and resulted in a logloss error of 0.58146088, the number of incorrect predictions would be 0.58146088 / 0.058146088 = 10. The percentage correctly predicted would then be (594 – 10) / 594 = 98.316%.

Submitting a one-hot version of a file would likely result in a higher logloss error than the probabilistic version of the same results if there are incorrect classifications contained in the file. For example, if we assume a smaller problem with only 3 possible classes, A, B, C, and the correct class for a submission were B, and the probabilistic submission were A = 0.6, B = 0.3, C = 0.1, then the logloss would be (-log(0.3) / 3) = 0.401. However, if the

file were converted to one-hot, with the highest probability class set to 1.0, and the others set to zero, the submission would be A = 1.0, B = 0.0, C = 0.0. Then, the logloss would be (-log(10 ^ -15) / 3) = 11.513. For this reason, two separate files were submitted for each output. A file with probabilistic classifications to obtain the lowest logloss error, and a file with a one-hot version of the classifications to allow percentage accuracy calculations.

At a minimum, the objective was to produce a result better than random guessing. Because there are 99 species, a random guess is likely to be correct 1 time out of 99, or 0.01010101, or 1.01%. This implies a logloss error for random guessing of -log(1 / 99) = 4.59511985. This value is shown on the Kaggle leaderboard for the competition as the Uniform Probability Benchmark, between positions 1519 and 1520 [4].

II. Analysis

Data Exploration

The Kaggle Leaf Classification dataset consists of 1584 leaf image files and two data files containing 192 numerical features for each leaf image. In their original format, the image files are black and white jpg's that range in size from a minimum width of 159 pixels to a maximum width of 1706 pixels, and a minimum height of 189 pixels to a maximum height of 1089 pixels. The average image width is approximately 693 pixels, and the average image height is approximately 493 pixels. These values can be found by running the "image_size.py" python file.

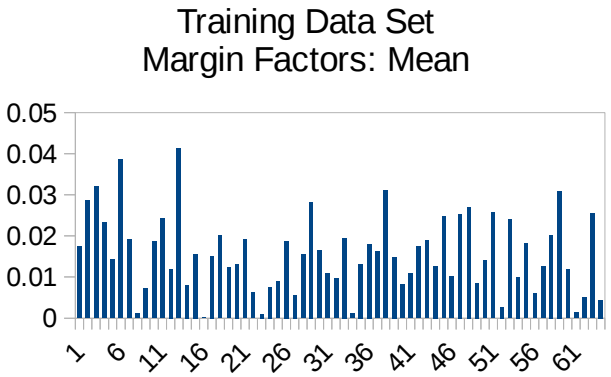
The numerical data files contain factors that describe three leaf classification criteria: shape, margin, and texture. The leaf shape describes the overall shape of a leaf. For example, a leaf may be long and thin or have approximately the same height as its width. It may be curved or straight. The margin factors describe the edges of the leaf. The leaf may have smooth edges or it may have sawtooth like edges. The texture describes the surface of the leaf. The leaf surface might be smooth and waxy, or it might be covered in small hairs. A more detailed discussion of these factors, suitable for a layperson, may be found in [5], [6], and [7]. A mathematical description of the factors may be found in [2] and its associated references.

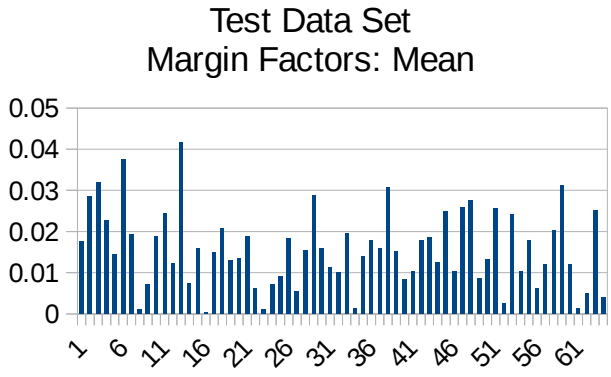
The data contains 64 factors for each of the three shape, margin, and texture features, for a total of 192 values.

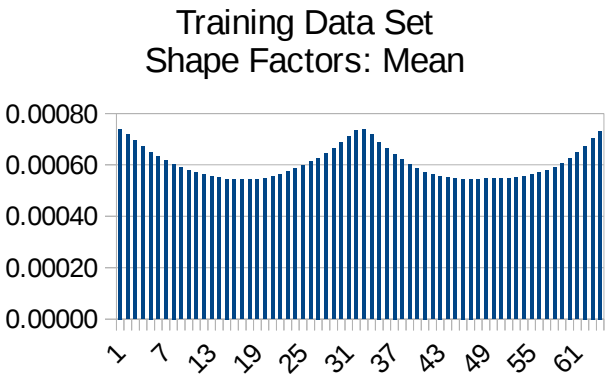
Exploratory Visualization

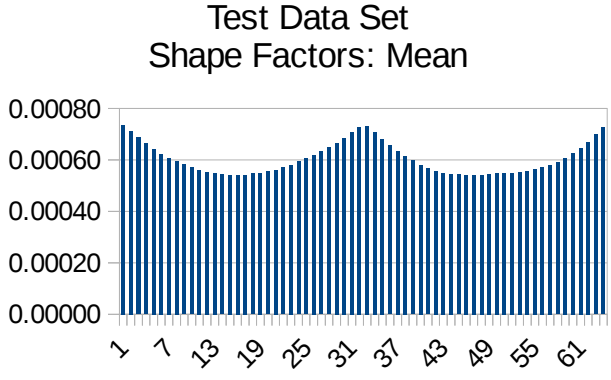
The 192 shape, margin, and texture features all range in value from 0.0 to 1.0. Because there are 192 values for each of the 1584 leaves in the data set, for a total of 304,128 values, displaying all of the values in the report would be space-prohibitive. However, because those 192 values were used as inputs for the multimodal CNN, KNN, and backpropagation neural network models, it was important to have some basic knowledge about he characteristics of the data.

The maximum value, minimum value, mean, and standard deviation was calculated for each of the 64 shape, margin, and texture features in the training and test data sets. Graphs of the mean and standard deviation values are shown below and on the following pages. The training data set graphs are on the left side of the page, and the corresponding test data set graphs are on the right side of the page.

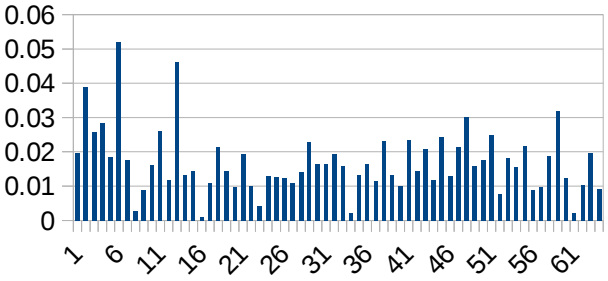




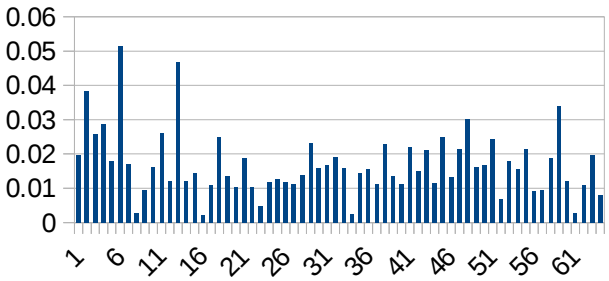




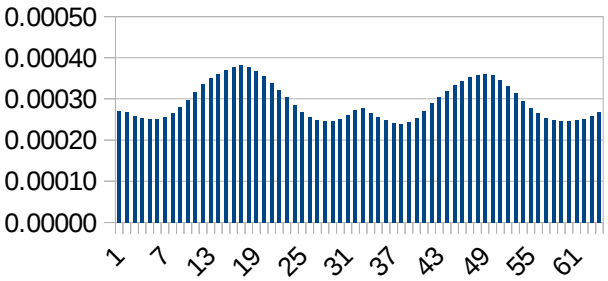
Training Data Set
Margin Factors: Standard Deviation



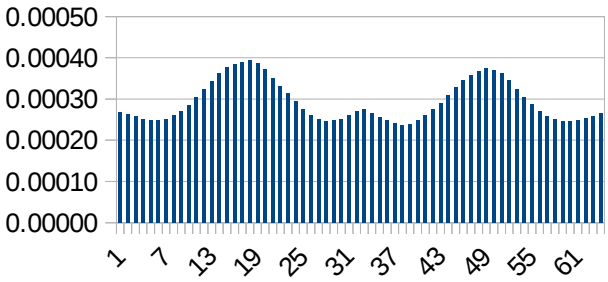
Test Data Set
Margin Factors: Standard Deviation



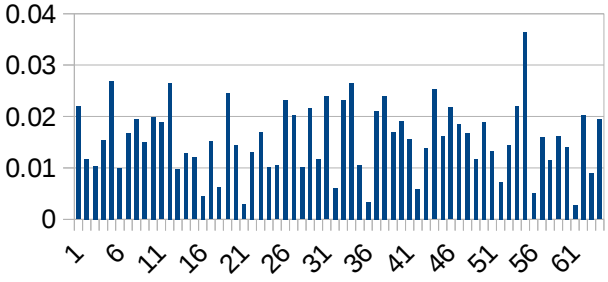
Training Data Set
Shape Factors: Standard Deviation



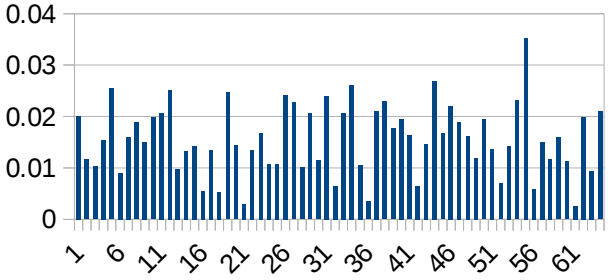
Test Data Set
Shape Factors: Standard Deviation

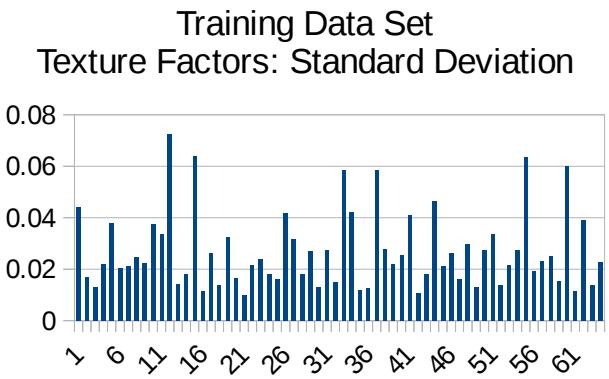


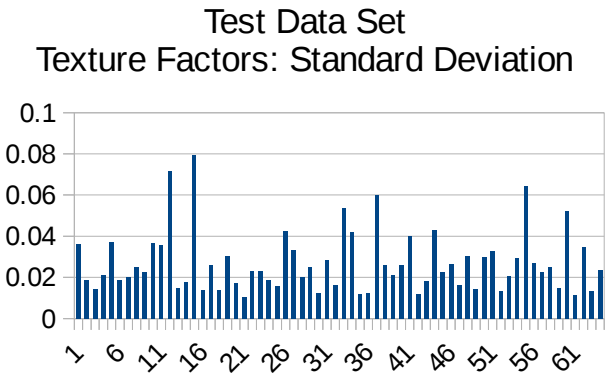
Training Data Set
Texture Factors: Mean



Test Data Set
Texture Factors: Mean







A comparison of the above graphs shows that the training data set and test data set have very similar characteristics. Indicating that the training data is representative of the test data and would be reasonable for use in machine learning models. Further examination shows that the standard deviations are fairly small, less than 0.001 for the shape factors. This implies that some type of data scaling would be necessary in order to adequately distinguish between the species in the data.

Algorithms and Techniques

Classification was attempted using four algorithms:

- Convolutional Neural Network (CNN)
- Multimodal Convolutional Neural Network
- Backpropagation Neural Network (NN)
- K-Nearest Neighbor (KNN)

A CNN was initially chosen to classify the image data because in the past few years, CNNs have been shown to be highly capable in this problem space [8]. A multimodal CNN, trained on both the image data and the numerical data, was used because GPU acceleration was not available for training the CNN, and using numerical data with the image data can decrease the amount of training time required. A backpropagation neural network was trained solely on the numerical data, because it was relatively easy to modify the CNN code to function as a backpropagation network, and training solely on the numerical data required less training time than training on the images. A KNN, operating solely on the numerical data, was also used as a comparison because KNN is fairly straightforward to implement, it is not as computationally intensive as the neural network algorithms, and a version of KNN was used in the original source paper [2].

The neural networks were implemented using Google's TensorFlow, from a modified version of their MNIST example, which was released under the Apache License, Version 2, and allows copying and modification [9, 10].

The KNN algorithm was implemented using Python 2.7.

K Nearest Neighbor (KNN)

Of the algorithms mentioned above, perhaps the easiest to understand is the k-nearest neighbor (KNN). KNN allows an item to be classified based on the vote of its nearest K neighbors [11]. For this project, a KNN classifier was created that used simple Euclidean distance [12]. To implement this algorithm

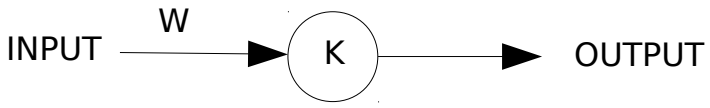
- The 192 shape, margin, and texture numerical values for all 990 items in the training data set were stored in memory
- The 192 numerical values for the test item to be classified were then placed in a list
- The distance between the test item and each training item was calculated based on the Euclidean distance formula $\sqrt{\sum_{i=1}^{192}(train_i - test_i)^2}$
- The maximum distance was saved
- The top K training items that were closest to the test item were placed in a list and sorted in order of distance.
- The distances in the list were then converted to probabilities by
  - subtracting values in the list from the maximum distance
  - summing the new values in the list
  - dividing the values in the list by the sum
  - (for example, if k = 3, the 3 closest distances were [0.1, 0.3, 0.4], and the maximum distance for all training items compared to the test item was 0.5. Then, subtracting each distance from the maximum distance yields [0.4, 0.2, 0.1]. Summing these values gives a result of 0.7. Dividing the items in the list by the sum gives the probabilities [0.57, 0.29, 0.14].)
- If the probabilities were from more than one instance of the same species, the probabilities were added together. (For example, if one entry in the list was associated with Acer_Circinatum and had a value of 0.57, and another entry in the list was also associated with Acer_Circinatum and had a value of 0.29, the total probability for Acer_Circinatum would be 0.86)
- The above steps were completed for each item in the test data
- The resulting probabilities were saved to a csv file under a column heading with the name of the species that was associated with the probability.

For this project, several values of K were experimented with and a final value of K = 5 was selected because it produced reasonable results.

Backpropagation Neural Network (NN)

A backpropagation neural network (referred to as just “NN” below) consists of one or more layers of artificial neurons connected by sets of weights. Training the NN is done with two steps, a forward pass to calculate the NN’s output, and a backward pass to adjust the weights in order to reduce the output error [13].

It may be easiest to understand the backpropagation algorithm by starting with just a single artificial neuron. (The simplified version of the equations shown below for the one and two neuron examples are based on the general equations in Matt Mazur’s “A Step by Step Backpropagation Example” [13].)



The INPUT is multiplied by the weight, W. The product of the INPUT and W is referred to as the NET, because it is the NET input to the neuron. Artificial neurons typically contain an activation function that operates on NET. In this case, the activation function just multiplies the NET by a constant, K. Thus,

$$OUTPUT = INPUT * W * K$$

Calculating the OUTPUT is the forward pass.

The output ERROR is calculated by subtracting the actual OUTPUT from the desired output, the TARGET:

$$ERROR = TARGET - OUTPUT$$

What is now needed is a method to change W so that the ERROR is reduced. We need to know ΔW / ΔERROR, or, abbreviating ERROR as E, ΔW / ΔE. The inverse of ΔW / ΔE can be calculated as follows:

$$\frac{\Delta E}{\Delta W} = \left(\frac{\Delta E}{\Delta OUTPUT}\right) * \left(\frac{\Delta OUTPUT}{\Delta NET}\right) * \left(\frac{\Delta NET}{\Delta W}\right)$$

$$\frac{\Delta E}{\Delta OUTPUT} = \frac{d}{d_{OUTPUT}}\left(TARGET - OUTPUT\right) = -1$$

$$\frac{\Delta OUTPUT}{\Delta NET} = \frac{d}{d_{NET}}\left(K * NET\right) = K$$

$$\frac{\Delta NET}{\Delta W} = \frac{d}{d_W}(INPUT * W) = INPUT$$

Thus,

$$\Delta E / \Delta W = -1 * K * INPUT$$

and $\Delta W / \Delta E$ will be its inverse.

Then, $\Delta W / \Delta E$ multiplied by the ERROR will give the portion of the weight W that was resulted in the ERROR. It basically tells you how far off W is from being the correct value. To move W in a direction to decrease ERROR, $(\Delta W / \Delta E) * ERROR$ must be subtracted from the initial W.

The updated value for W will be the old value for W minus the change in W for the particular error, or

$$W_{NEW} = W_{OLD} - (\frac{\Delta W}{\Delta E}) * ERROR$$

or

$$W_{NEW} = W_{OLD} - \frac{-1}{INPUT * K} * ERROR$$

or, simplified further:

$$W_{NEW} = W_{OLD} + \frac{ERROR}{INPUT * K}$$

Let's work through an example. Assume we want to model a linear equation that is in the form a = bx, where b = 1.2. If we select a value for x = 3.4, a = 1.2 * 3.4 = 4.08. Now, assume that we have an activation function with K = 0.5, and an initial weight value of W = 11.5. NET = INPUT * W. With the INPUT = x = 3.4, NET = 3.4 * 11.5 = 39.1. OUTPUT = K * NET = 0.5 * 39.1 = 19.55. With the TARGET = a = 4.08, the ERROR = 4.08 – 19.55 = -15.47. So,

$$W_{NEW} = W_{OLD} + \frac{-15.47}{3.4 * 0.5}$$

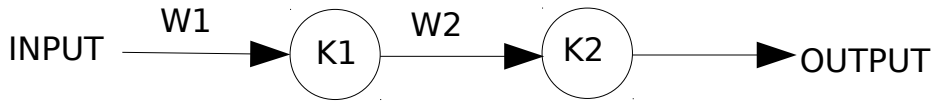
$$W_{NEW} = W_{OLD} - 9.1$$

$$W_{NEW} = 11.5 - 9.1$$

$$W_{NEW} = 2.4$$

If we recalculate the output using the new W, the result is OUTPUT = INPUT * W * K = 3.4 * 2.4 * 0.5 = 4.08. Which matches the value of the function we were trying to model: a = bx = 1.2 * 3.4 = 4.08. We can see that the artificial neuron will match the function a=bx for all x, because W * K * INPUT = 2.4 * 0.5 * INPUT = 1.2 * INPUT, the same as a = 1.2 * x.

Expanding the single neuron example to a two-layer “network” gives the system shown below:



In this system, W2 can be trained as W in the single neuron example above, but the problem becomes how to adjust W1. We need ΔW1 / ΔE. Error, E is defined as above, NET1 = INPUT * W1, and NET2 = OUT1 * W2 = (INPUT * W1 * K1) * W2. OUTPUT = OUT2 = (INPUT * W1 * K1) * W2 * K2. Then ΔE / ΔW1 can be found with the following equation, and ΔW1 / ΔE will be its invers:

$$\frac{\Delta E}{\Delta W1} = \left(\left(\frac{\Delta E}{\Delta OUT2}\right) * \left(\frac{\Delta OUT2}{\Delta NET2}\right) * \left(\frac{\Delta NET2}{\Delta OUT1}\right)\right) * \left(\left(\frac{\Delta OUT1}{\Delta NET1}\right) * \left(\frac{\Delta NET1}{\Delta W1}\right)\right)$$

Solving the above terms for the linear two neuron system gives:

$$\frac{\Delta E}{\Delta OUT2} = \frac{d}{d_{OUT2}}\left(TARGET - OUT2\right) = -1$$

$$\frac{\Delta OUT2}{\Delta NET2} = \frac{d}{d_{NET2}}\left(K2 * NET2\right) = K2$$

$$\frac{\Delta NET2}{\Delta OUT1} = \frac{d}{d_{OUT1}}\left(W2 * OUT1\right) = W2$$

$$\frac{\Delta OUT1}{\Delta NET1} = \frac{d}{d_{NET1}}(K1 * NET1) = K1$$

$$\frac{\Delta NET1}{\Delta W1} = \frac{d}{d_{W1}}(INPUT * W1) = INPUT$$

Which gives

$$\Delta E / \Delta W1 = -1 * K2 * W2 * K1 * INPUT$$

and

$$\Delta E / \Delta W2 = -1 * K2 * OUT1$$

The weight changes would then be:

$$W1_{NEW} = W1_{OLD} - (\frac{\Delta W1}{\Delta E}) * ERROR$$

or

$$W1_{NEW} = W1_{OLD} - \frac{-1}{K2 * W2 * K1 * INPUT} * ERROR$$

and

$$W2_{NEW} = W2_{OLD} - (\frac{\Delta W2}{\Delta E}) * ERROR$$

or

$$W2_{NEW} = W2_{OLD} - \frac{-1}{K2 * OUT1} * ERROR$$

if we then try another example, with:

INPUT = 3.4
TARGET = 4.08
W1 = 3.7
K1 = 0.2
W2 = 11.5
K2 = 0.8

Then,

OUTPUT = 3.4 * 3.7 * 0.2 * 11.5 * 0.8 = 23.1472

ERROR = TARGET – OUTPUT = 4.08 – 23.1472 = -19.0672

and

K2 * OUT1 = 0.8 * 3.4 * 3.7 * 0.2 = 2.0128

K2 * W2 * K1 * INPUT = 0.8 * 11.5 * 0.2 * 3.4 = 6.256

Plugging these values into the equations for $W1_{NEW}$ and $W2_{NEW}$ gives:

$W2_{NEW}$ = 11.5 + (1.0 / 2.0128) * -19.0672 = 2.027

$W1_{NEW}$ = 3.7 + (1.0 / 6.256) * -19.0672 = 0.652

If we plug the new weight values in to the equation for OUTPUT we get:

OUTPUT = 3.4 * 0.652 * 0.2 * 2.027 * 0.8 = 0.7189

Then ERROR = 4.08 – 0.7189 = 3.3611. Unlike the single neuron example above, the first iteration through training did not give us an ERROR = 0.0. If we repeat the training iteration with W1 = 0.652 and W2 = 2.027 (calculations not shown, they are just a repeat of the above with different values of W1 and W2), the resulting values for W1 and W2 will be W1 = 3.7 and W2 = 11.5. But these are the same as the original values for W1 and W2!

If the training steps are repeated again, the weights will go back to W1 = 0.652 and W2 = 2.027. The weights are simply oscillating back and forth, passing over the solution without converging to it. This is because the change in the weight values is too large, what is needed is a way to reduce the weight change to allow the solution to converge. This is typically done with a learning rate, usually identified as the Greek letter ETA (η) [13, 14].

If we modify the weight change equations by adding the learning rate, we get

$$W1_{NEW} = W1_{OLD} - \eta\left(\frac{-1}{K2 * W2 * K1 * INPUT}\right) * ERROR$$

$$W2_{NEW} = W2_{OLD} - \eta\left(\frac{-1}{K2 * OUT1}\right) * ERROR$$

If we select a value for η = 0.7, then re-run the training calculations, we get:

$W2_{NEW}$ = 11.5 + 0.7 *(1.0 / 2.0128) * -19.0672 = 4.869

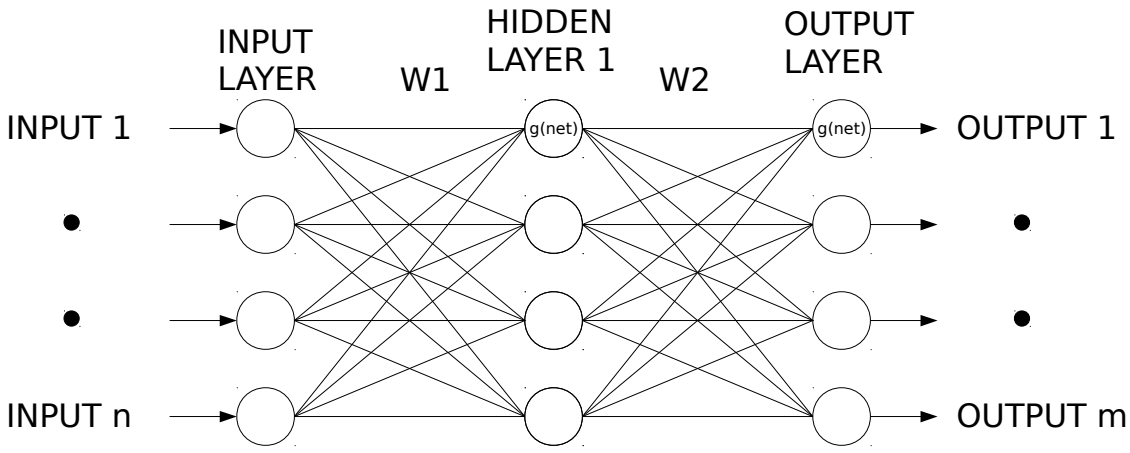
$W1_{NEW}$ = 3.7 + 0.7 *(1.0 / 6.256) * -19.0672 = 1.567

Then, calculating the new output with the new weights:

OUTPUT = 3.4 * 1.567 * 0.2 * 4.869 * 0.8 = 4.151

This gives us an ERROR = 4.08 – 4.151 = -0.071. While still not equal to 0.0, this is a much smaller error than the one achieved without the learning rate.

While the above examples are instructive, for such simple examples, particularly in the single neuron case, it would be easy to simply solve for the weight values directly without using the backpropagation algorithm. The real value of backpropagation comes when trying to model a system that cannot easily be solved directly. There are often more than one set of input and output values, with the inputs and outputs in the form of a multi-element vectors. In this situation, the NN will resemble something more like the system shown below:



The input layer typically does not perform any operations on the input, it merely transmits them into the network. The g(net) function shown within the hidden layer and output layer neurons is called the activation function. It takes the place of the K1 and K2 constants in the simpler examples above. For multiple layer networks, a simple linear activation function (like K * NET above) does not offer any advantage over a single layer network. A multi-layer network with linear activation functions is equivalent to a single layer network [15]. For this reason, non-linear activation functions are used. Some of the more popular activation functions are tanh( ) and the sigmoid function [13, 15].

The elements of the input vector are multiplied by the weights W1, resulting in net values which are then used as inputs into the activation functions g(net). The

output of the hidden layer is used as input into the second layer of weights, W2, and the process is repeated until the outputs are generated.

With multiple sets of multi-element inputs and outputs, the ERROR equation above becomes inadequate. A more general error function is needed. One error function that is used is the squared error function:

$$E_{total} = \sum 1/2 \left(target - output \right)^2 \quad [13]$$

The errors are summed for each output neuron, and for each set of input-output pairs.

With non-linear activation functions operating on multiple weights, more general forms of the weight update equations are also needed.

$$\frac{\Delta E}{\Delta W} = \left(\frac{\Delta E}{\Delta OUTPUT} \right) * \left(\frac{\Delta OUTPUT}{\Delta NET} \right) * \left(\frac{\Delta NET}{\Delta W} \right)$$

Can be generalized to

$$\frac{\partial E}{\partial W2} = \frac{\partial E}{\partial OUTPUT} \frac{\partial OUTPUT}{\partial NET2} \frac{\partial NET2}{\partial W2} \quad [13]$$

and

$$\frac{\Delta E}{\Delta W1} = \left(\left(\frac{\Delta E}{\Delta OUT2} \right) * \left(\frac{\Delta OUT2}{\Delta NET2} \right) * \left(\frac{\Delta NET2}{\Delta OUT1} \right) \right) * \left(\left(\frac{\Delta OUT1}{\Delta NET1} \right) * \left(\frac{\Delta NET1}{\Delta W1} \right) \right)$$

can be generalized to

$$\frac{\partial E}{\partial W1} = \sum \left(\frac{\partial E}{\partial OUT2} \frac{\partial OUT2}{\partial NET2} \frac{\partial NET2}{\partial OUT1} \right) * \left(\frac{\partial OUT1}{\partial NET1} \frac{\partial NET1}{\partial W1} \right) \quad [13]$$

Many variations of the backpropagation neural network are possible, and examples can be found in references [13, 14, 15], but, the basic backpropagation algorithm remains the same:

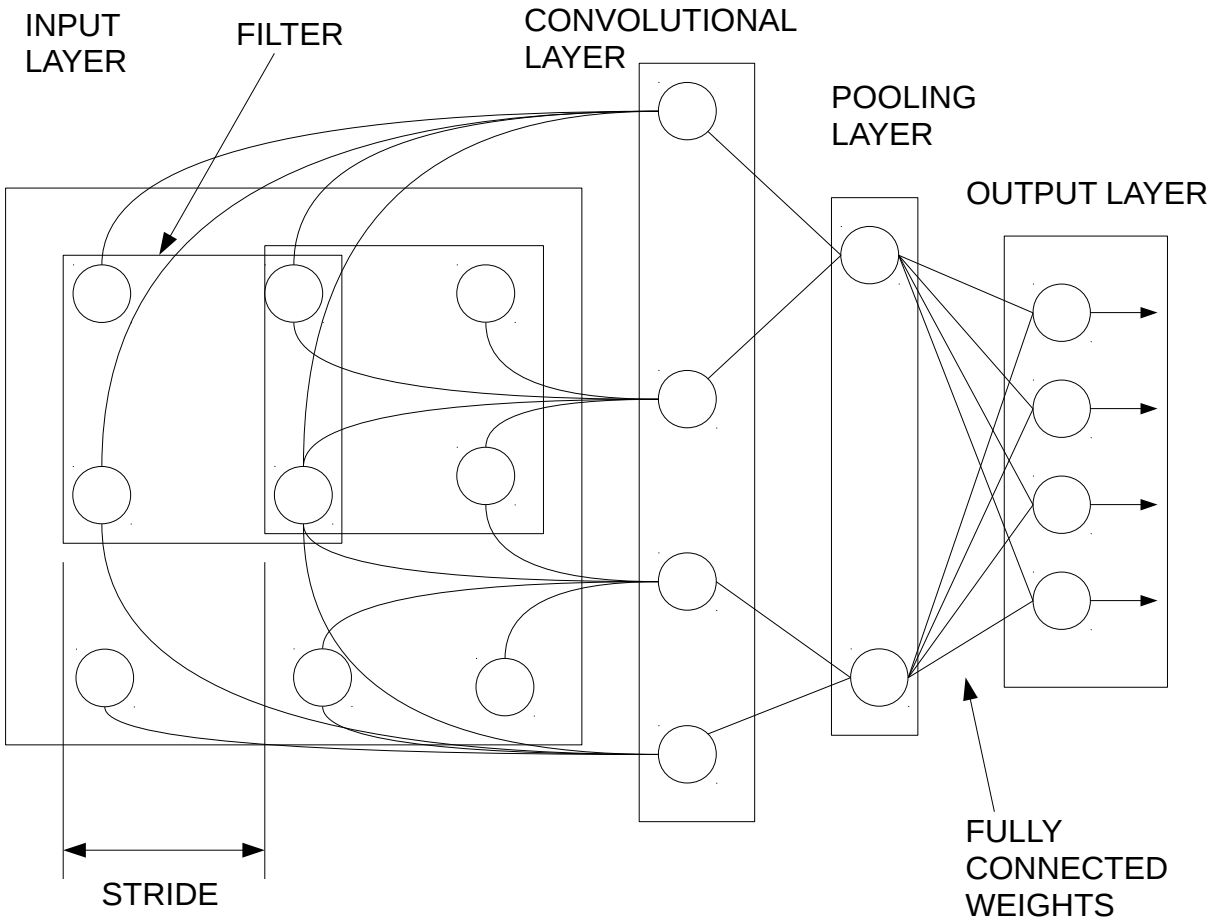
- Submit an input vector to the networks
- Calculate the first layer output by multiplying the inputs by the weights
- Use the values above as inputs into the activation functions
- Use the outputs of the activation functions as inputs to the next layer
- Repeat the above two steps until the final output is calculated
- Use the final output to calculate the error
- Modify the weights to reduce the error

- Repeat the above steps for a set number of iterations, or until the error reaches an acceptable level.

Convolutional Neural Network (CNN)

Convolutional neural networks use the same basic type of training algorithm as the backpropagation networks described above [16]. They were designed to work specifically with images for the inputs, and differ from backpropagation networks in that they have several specialized layers that may not be fully connected. A fully connected layer has a weight connecting each neuron in its layer to each neuron in the previous layer (as shown in the illustration on page 14).

A basic CNN will typically have input layer, a convolutional layer, a pooling layer, a fully connected layer, and an output layer [17]. As with the backpropagation neural network above, understanding CNNs can be facilitated by beginning with a simple example. The example below is partially based on the input, convolutional, and pooling layers from reference [18].



The input layer of a CNN is typically configured to accept an image as an input. In the example above, the input layer is a 3 x 3 grid, which could accept a 3x3 image as input. Filters are applied to the input layer in order to convolve, or group together, sections of the input image. The filter above is 2x2.

Filters are applied over the image with the spacing between each filter known as the “stride.” The stride in the example above is 1 pixel.

The value of the pixels covered by each filter are multiplied by weights and fed into the convolutional layer. This allows the convolutional layer to extract pertinent features from the input layer without being fully connected.

Further dimensional reduction is achieved by passing the outputs of the convolutional layer into a pooling layer [16].

The number of neurons in the output layer corresponds to the number of classes the input images are to be sorted into. In the example above, there are four output neurons, so there would be four classes.

The output layer is typically fully connected to the neurons in its preceding layer.

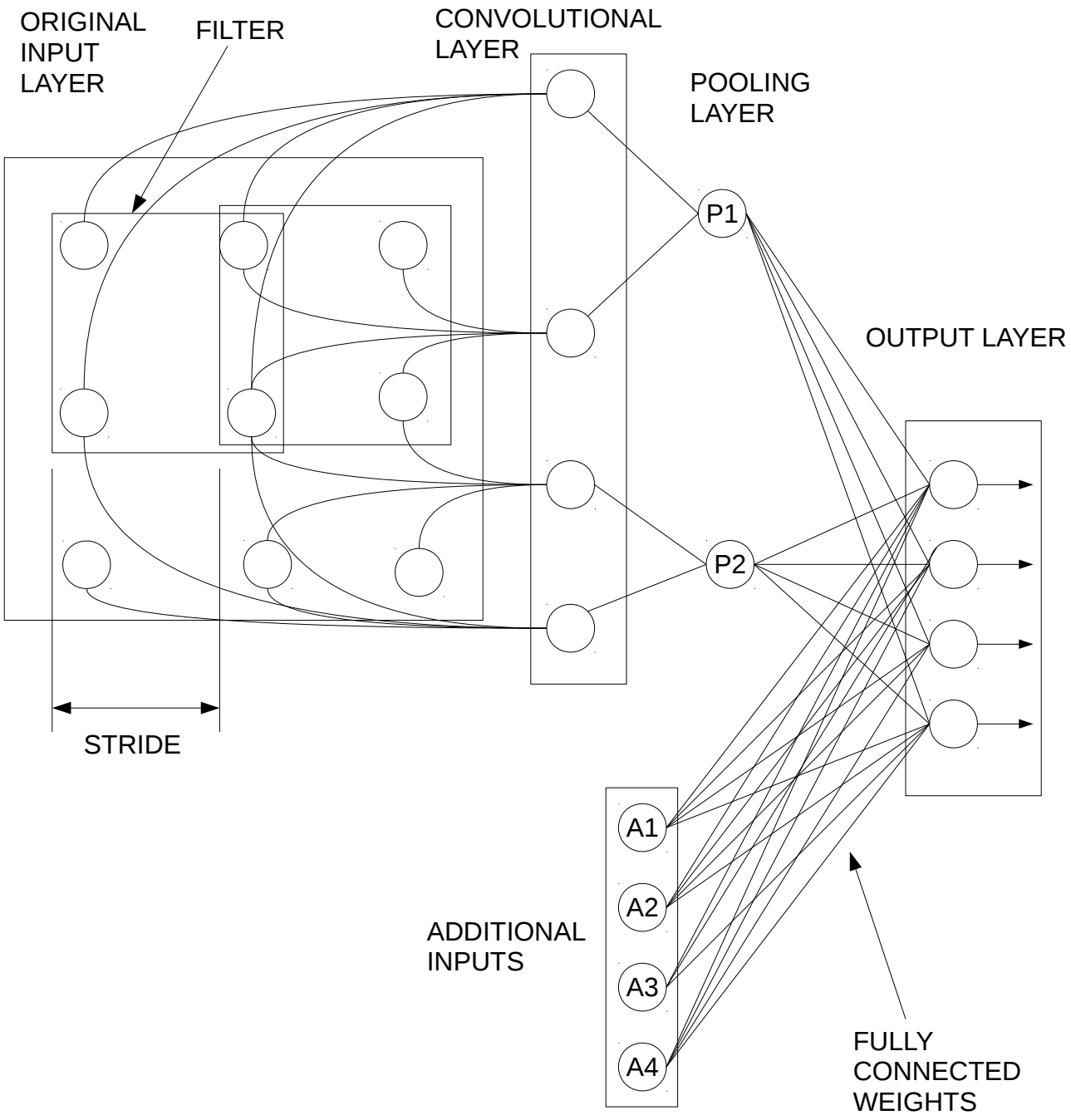
For color images, in addition to inputs corresponding to the height and width of the images, there are also depth inputs. Usually, there are three input layers, accepting inputs corresponding to the red, green, and blue values of the image pixels.

In practice, there are often multiple convolution and pooling layers [16, 18].

Training for the CNN is done in the same manner as the backpropagation neural networks in the previous example. Though the training is done in the same way, a CNN typically requires lower training time than a backpropagation neural network, because there are fewer weights to be trained. Looking at the simple CNN above, it contains 4x4 + 2x2 + 2x4 = 28 weights. A fully connected version of the network, however, would contain 9x4 + 4x2 + 2x4 = 52 weights.

Multimodal Convolutional Neural Network

Multimodal learning involves training a network on multiple types of input information [19]. The different data types can be combined in multiple ways, one method is late fusion. With late fusion, the additional data is combined near the output layer of the network [20]. A modified version of the above CNN showing one way to achieve this is shown on the following page.



In the above example, additional input neurons are added and fully connected to the output neurons. In the original CNN, the input into the fully connected layer would only have been output from the pooling layer. Expressed as a vector: [P1, P2]. To use the additional inputs, the outputs from the additional input neuron output is appended to the output from P1 and P2, so that the new input vector

into the fully connected weights would be [P1, P2, A1, A2, A3, A4]. Training can then be done as shown in the previous neural network examples.

Benchmark

Benchmarking was based on the logarithmic loss value and the percentage classification accuracy (both defined in the **Metrics** section, above). At a bare minimum, the classifiers should perform better than random guessing. Random guessing would result in a log loss value approximately equal to 4.59511985 (again, see **Metrics** section). The lower the log loss, the better the performance, with a perfect classifier resulting in a log loss of 0. Therefore, at minimum, the classifier should produce a log loss value less than 4.59511985.

At a higher level of performance, it would be desirable for the classifier to meet or exceed the percentage accuracy of the KNN algorithm in the source paper. The highest percentage accuracy achieved in the source paper was 96% [2].

Finally, a further benchmark would be the placement of the classifier performance on the Kaggle leaderboard. However, the top 24 teams on the Kaggle leaderboard show a log loss of 0.00000, indicating perfect performance, which might be difficult to match [4].

III. Methodology

Data Preprocessing

Images

For a fully connected neural network, the number of multiplication operations needed for the first layer will be equal to the (size of the input vector) * (number of neurons in first layer). Therefore, without the benefit of GPU acceleration, the computation time required will be proportional to the size of the input vector. The input vector size for an image will be its (width in pixels) * (height in pixels) * (number of color channels). Doubling the dimensions of a square image approximately quadruples the required computational time. For example, a 32x32x1 image would have an input vector size of 1024, and a 64x64x1 image would have an input vector size of 4096.

The average size of the data images is approximately 693x493 pixels, for an average input vector size of 341,649 elements. In order to reduce time required for training, two sets of scaled images, one set of 32x32 pixels and another set of 64x64 pixels, were created. As a comparison, if a neural net using 32 x 32 pixel input images required 30 minutes for training, a comparable network trained on the original images would take approximately (341,649 / 1024) * 0.5 hours =

166.82 hours, or about 7 days.

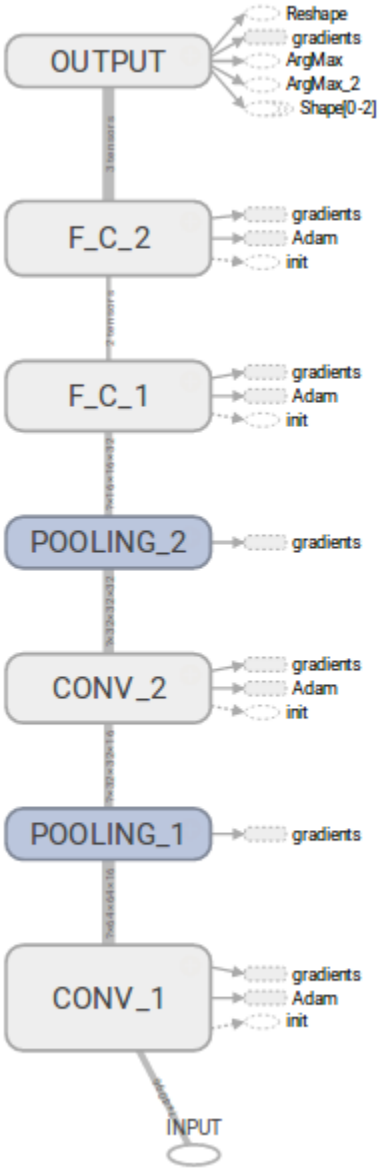
The images were scaled by first determining the maximum height and width of the original images by running *images_size.py.* The maximum height was 1089 pixels and the maximum width was 1706 pixels. The re-purposed MNIST TensorFlow CNN was designed to accept square images for its inputs, therefore the images were re-sized into squares, with each side having the same number of pixels as the maximum width. This was done using *normalize_images.py.* The images were then cropped to remove any excess border space around the leaves using *crop_images.py.* The resulting cropped images were then scaled to 32x32 and 64x64 pixel images using *scale_images.py.*

Numerical Data

Because the differences between the numerical data were small (see **Exploratory Visualization** section, above) scaling was needed to adequately differentiate between the leaf species. The numerical data was first scaled by applying a Min-Max scaler. Then a standard deviation scaler was applied to the resulting Min-Max values [21]. The min-max scaling was performed using *min_max_scaler.py* and the standard scaler was applied using *standard_scaler.py*.

Implementation

A convolutional neural network was created using a modified version of Google's MNIST example [9][10]. The resulting CNN had the configuration shown at the right, with the INPUT being either the 32x32 or the 64x64 processed images. The first convolutional layer (CONV_1), used a 5x5 pixel patch size, with a stride of 1,1 and computes 16 features for each 5x5 patch. The first pooling layer (POOLING_1), performs max pooling with 2x2 filters and a stride of 2. The second convolutional layer (CONV_2) also uses a 5x5 patch with a 1,1 stride, but computes 32 features for each patch. The second pooling (POOLING_2) performs max pooling with 2x2 filters and a stride of 2. POOLING_2 is followed by a first fully connected layer (F_C_1), which contains 512 neurons, and is then followed by second fully connected layer (F_C_2) which also contains 512 neurons. Finally, the output from F_C_2 is multiplied by a



512x99 matrix to produce the 99 element OUTPUT vector.

The input images are supplied in a random order to the network, and are randomly rotated by ± 15 degrees. The neurons in each layer use a hyperbolic tangent (tanh) activation function. To reduce overfitting, dropout is used in F_C_2 [22]. The network was trained using the AdamOptimizer [23] and a softmax function was applied to the output [24] [25].

Refinement

Four convolutional neural networks were trained, using 32x32 and 64x64 images, a batch size of 99, for 10,000 and 15,000 epochs. The results are summarized in the following table:

Image Size	Training Epochs	Logloss error	% Accuracy	Output file
32x32	10,000	1.26751	68.18	softmax_32_10k.csv
32x32	15,000	1.3096	71.04	softmax_32_15k.csv
64x64	10,000	1.20017	70.54	softmax_64_10k.csv
64x64	15,000	1.26329	72.05	softmax_64_15k.csv

These results are much better than the random logloss error of 4.59511985 or the random % accuracy of 1.01%, but are far from the best logloss errors of 0.0 achieved on the Kaggle leaderboard [4] or the 96% accuracy achieved by Mallah [2].

To verify that the correct training data and test data were being used, and to verify that the correct format was being used for the submission file, a k-nearest neighbor (KNN) classifier, using only the numerical data and not the images, was created using Python 2.7. The numerical data was first scaled using a min-max scaler, then the resulting min-max scaled data was again scaled using a standard deviation scaler. The KNN used simple Euclidean distance on the scaled data to determine distance between neighbors, and was testing using k = 5. For k = 5, the resulting logloss error of the submission file was 0.14051, when converted to a one-hot file, this resulted in a percentage accuracy of 96.97%. This percentage accuracy is close to the 96% accuracy achieved in [2], and indicates that the basic format of the test data, training data, and submission files was correct.

After this basic verification step using KNN, the next step was to attempt to increase the accuracy of the CNN. A review of the Kaggle leaderboard for published kernels that achieved relatively high accuracy revealed the kernel used by Xiao Wang. Wang placed at position 161 on the leaderboard with a logloss error of 0.01551 [4]. Further review of Xiao Wang's kernel showed that Wang used a multimodal CNN, combining the image inputs with the numerical inputs, to achieve higher accuracy [26].

To implement the multimodal CNN, the original CNN was modified as shown below:

The image input (IMG_IN) was fed into the first convolutional layer (CONV_1) as with the non-multimodal CNN. From there, calculation continued as before until the output of the first fully connected layer (F_C_1). The output of F_C_1 was combined with the numerical input (NUM_IN) corresponding to IMG_IN. This COMBINED vector was then used as input to the second fully connected layer, F_C_2. Aside from this difference in input configuration, training was performed as with the original CNN. Four multimodal CNNs were trained, using 32x32 and 64x64 images, at 10,000 and 15, epochs. The results are shown in the table on the following page.

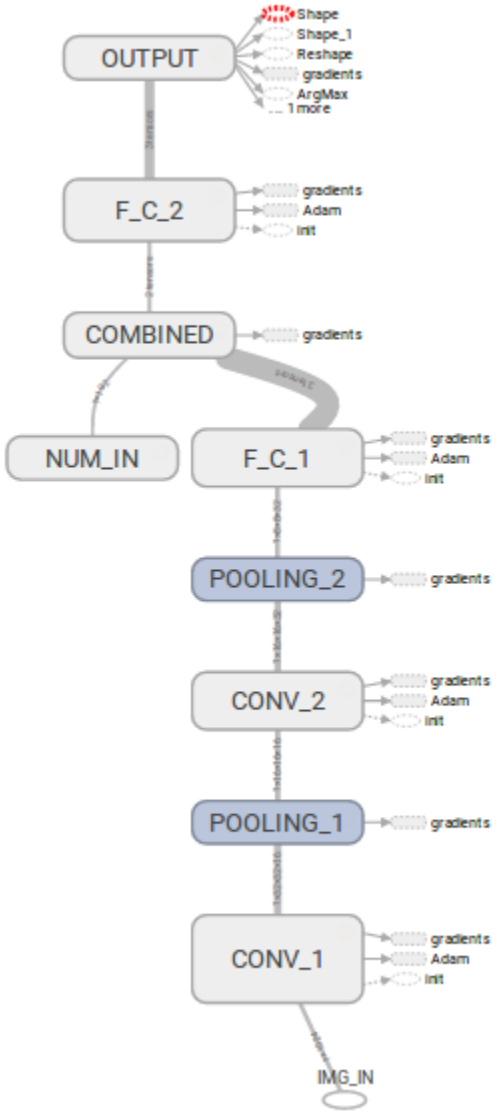


Image Size	Training Epochs	Logloss error	% Accuracy	Output file
32x32	10,000	0.65401	82.32	mm_32_10k.csv
32x32	15,000	0.61976	84.85	mm_32_15k.csv
64x64	10,000	1.04863	73.57	mm_64_10k.csv
64x64	15,000	1.10034	75.47	mm_64_15k.csv

These results were a noticeable improvement on the non-multimodal CNN. The greatest improvement was with the 32x32 image CNN trained for 15,000 epochs, which went from an accuracy of 71.04% to an accuracy of 84.85%, a 13.81% improvement. However, while these results are an improvement, they are still shy of the results achieved with the KNN above, the percentage accuracy in [2], or the logloss of 0.0 on the Kaggle leaderboard [4].

Further review of the Kaggle leaderboard for entries with a low logloss error and a publicly available kernel indicated that one of the lower logloss errors of 0.00545, at position 41, was achieved by David Prakash [4]. Prakash's publicly available kernel achieved a logloss error of 0.02204 using a neural network trained only on the numerical data, without use of the image data [27]. This logloss error of 0.02204 was lower than any of the logloss errors achieved with any of the methods previously used in this project.

Implementation Challenges

Possibly the most challenging aspect of implementing the project was learning how to use Google Tensorflow. While it is an extremely powerful tool, there is a learning curve.

I found it somewhat difficult to debug, as there is no traditional style of integrated development environment (IDE) that allows for running a project in debug mode and stepping through it to see how variable values change. I also did not see a way to print the variable values to a terminal while a Tensorflow session was running.

Some helpful ways of dealing with the Tensorflow learning curve are:

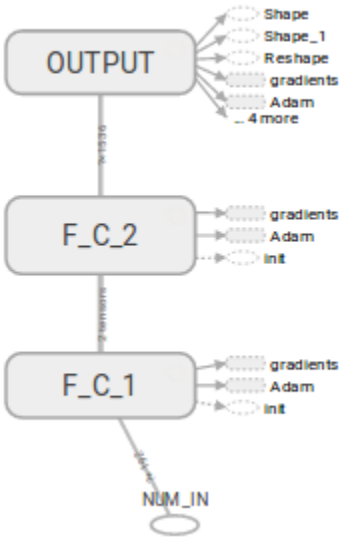
- Start with an example problem that is licensed for copying and modification, then modify that example to meet your needs. One such example is Google's MNIST [9].
- Use the TensorBoard tool to graphically view your network.
- While I haven't found a way to print values from within a learning session, it is possible to print the output of a session using sess = tf.InteractiveSession() and print(sess.run([y], feed_dict={x: X_VALUES}))
  - feed_dict assigns values to the tensor "x"

- The session will run up to the point where variable “y” is evaluated and print the result

IV. Results

Model Evaluation and Validation

In an attempt to replicate the low logloss error achieved by Prakash, a neural network using only the numerical data as input was created. The basic configuration of this network is shown in the graph to the right. The numerical input data (NUM_IN) is fed into the first fully connected layer (F_C_1), the output of which goes into the second fully connected layer (F_C_2), and emerges as the classification output (OUTPUT).



Both of the fully connected layers contain 1536 neurons. Prakash used a rectified linear unit (relu, [28]) activation function for his first layer, and a sigmoid activation function [29] for his second layer [27]. A relu activation function was used for F_C_1, and a sigmoid activation function was used for F_C_2.

Justification and Training Robustness

The numerical input only neural network was trained for 30,000 epochs using a batch size of 99. The resulting logloss error was 0.02209, extremely close to the 0.02204 logloss error achieved by Prakash [27]. The output file for this network is saved as “NN_30k.csv” A one-hot version of this output file was created to test the percentage accuracy. The one hot file is saved as “one_hot_NN_30k.csv” and resulted in a logloss error of 0.17444, which equates to a percentage accuracy of approximately 99.5%.

While the final resulting logloss error of 0.02209 is still greater than the logloss of 0.00000 achieved by the highest-placed entries on the Kaggle leaderboard [4], if submitted while the competition was still underway, it would have placed at position 254, after the logloss error of 0.02202 submitted by Manish Yathnalli [4]. There were 1598 entries recorded on the Kaggle leaderboard, placing at 254 would have put this model in the top 16%.

The 99.5% accuracy of the model is 3.5% better than the 96% accuracy achieved by Mallah [2].

Robustness of the final neural network model was evaluated by training the network four additional times. The initial values of the weights were random, as were the order of submission of the numerical data for each species. If the model produced similar results from these randomly different starting conditions, then it would be sufficiently robust for use. The results of the robustness testing are shown in the table below:

Model	Logloss error	Percent accuracy
Initial training	0.02209	99.50
Robustness check 1	0.03300	98.99
Robustness check 2	0.02906	99.33
Robustness check 3	0.02145	99.66
Robustness check 4	0.02134	99.16

The standard deviation for the logloss errors is 0.005506, and the percentage accuracy for all instances of the model is close to 99%. These results are sufficiently similar to indicate that the model is robust enough for the problem

Robustness with Noisy Inputs

One method of quantifying noise is with a signal to noise ratio (SNR). The ratio compares the level of a desired signal to the level of noise [30]. If the variance of the signal and noise are known, and the signal and noise are both zero-mean, SNR can be defined as

$$SNR = \frac{[\sigma^2_{signal}]}{[\sigma^2_{noise}]}$$ [30]

Where the variance, σ^2, is the standard deviation squared.

Fortunately, the standard scaler inputs to the network meet these criteria.

A new version of the backpropagation neural network was created which allowed for the addition of noise into the input signal. The standard deviation of the standard scaler data is, by definition, 1.0. Therefore its variance is also 1.0 (1 ^ 2).

The amount of noise to be added to the signal could then be defined by solving the above equation for the desired standard deviation of the noise. For example, if the desired SNR was 10.0, sigma_noise = sqrt(1.0 / 10.0) = 0.3162. The desired standard deviation for the noise was then plugged into Python’s Gaussian random

function, so that the noise to be added would be noise = random.gauss(0.0, sigma_noise). This noise value was then added to each of the values in the input data, with a different noise value for each data point. The random nature of Python's Gaussian function results in an actual SNR that is slightly different from the desired SNR, but the actual SNR is close enough to the desired value to adequately test the model. (For example, the desired SNR might be 10.0, but the actual SNR might wind up being 9.9 or 10.1).

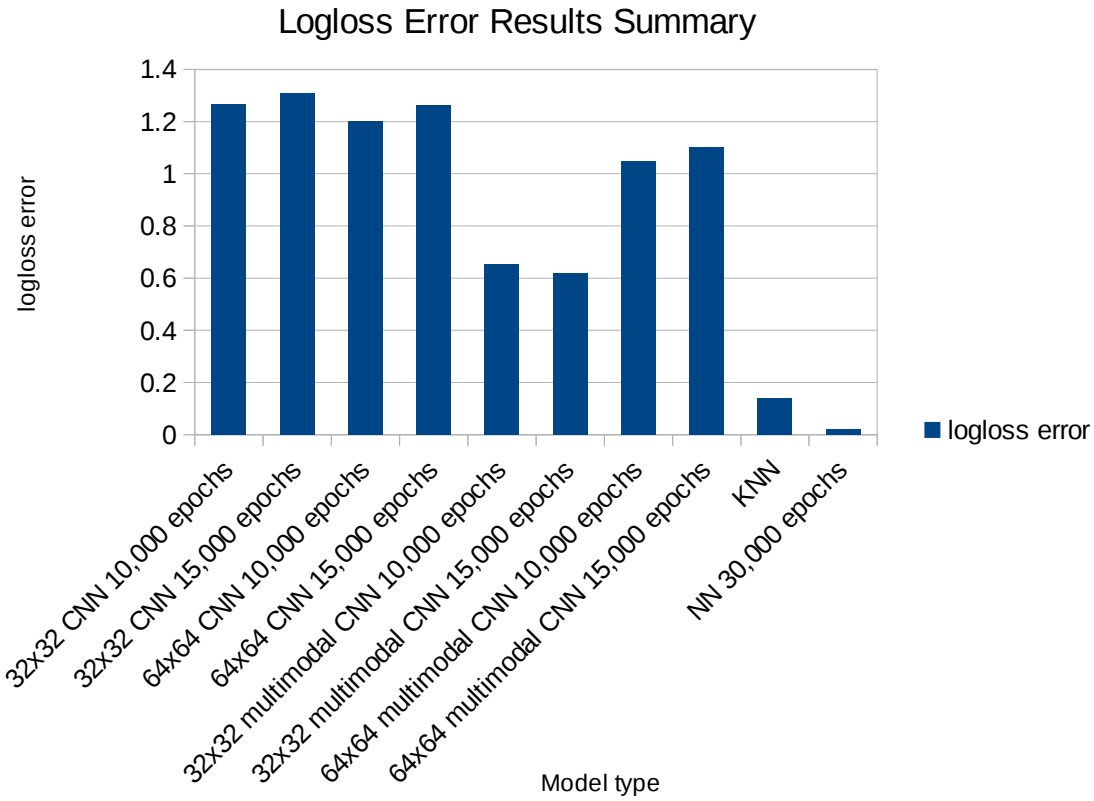
A new neural network was trained for 30,000 epochs, and noisy inputs corresponding to SNRs of [NAN, 1.0, 5.0, 10.0, 50.0, 100.0, 1000.0] were submitted to the network. (Note that NAN stands for "not a number", because with zero noise, the SNR is undefined, due to division by zero. Thus the NAN input corresponds to a zero-noise input.) The results are shown in the table below:

Approximate SNR	Logloss error	% accuracy
NAN	0.0276	99.33
1000	0.0269	99.33
100	0.03010	99.16
50	0.04503	99.33
10	0.03448	98.82
5	0.04503	98.99
1	0.21975	94.44

The table above shows that the logloss error remains low and that the percentage accuracy remains high, with most percent accuracy values near 99%, until the SNR gets to 1.0, or, when the noise level is equal to the signal level. This indicates that the model is quite robust when given noisy inputs.

Conclusion

Free-form Visualization



A summary of the logloss errors produced by each of the methods used is shown in the chart below.
One interesting fact that can be derived from the chart is that the models which relied *less* on the images for training performed better than the models which relied *more* on the images. The KNN and pure numerical neural network (NN) performed better, by far, than the CNNs and multimodal-CNNs. The multimodal-CNNs performed better than the image-only CNNs. This shows a somewhat counterintuitive result of a smaller training vector input size producing a better result than a larger training vector input size. This is probably an example of the "curse of dimensionality" [31]. As the size of the input training vector increases, the number of training examples needed also increases, and may increase exponentially [31]. The number of elements in the numerical-only input vector is just 192, however, for even the smallest 32x32 image, the size of the input vectors is 32x32 = 1024.

Using base 2 logarithms to compare input vectors, if the number of training examples for a 192 element vector is K * (2 ^ 192), the number of examples for a 1024 element input vector would be K * (2 ^ 1024) / (2 ^ 192) = K * (2 ^ 832). Converting to log base 10, approximately 10 ^ 251 times more examples would be required to train a 1024 element input vector network as compared to 192 element input vector network.

Even assuming a simple linearly proportional increase in training examples required for each input dimension, the 1024 element vector would require 1024 / 192 or 5.33 times more training examples than a network trained with a 192 element input vector.

Applied to this project, a minimum of 5.33 * 594, or 3168, training image examples would probably be need for the image-only CNN to approach the accuracy of the numerical factor-only NN.

Reflection

The initial intent of the project was to create a species classification model using only the images as input into a CNN. After attempting multiple CNN configurations, using various epoch batch sizes and training for differing numbers of total epochs, it became clear that a CNN trained solely on the image data was not going to produce logloss or percentage accuracy results comparable to those shown on the Kaggle leaderboard or achieved by Mallah [2][4]. This led to an exploration of different methods

First, a KNN program was written to attempt to replicate the results achieved by Mallah [2]. The KNN results were comparable and gave me confidence that I was using the input data correctly and using the proper configuration for the Kaggle submission files. Then, I reviewed publicly available kernels for the higher ranking scores on Kaggle. Which led to the multimodal CNN, and finally to the numerical-only NN. These achieved better results. Unexpectedly, the best result was achieved by the NN without using any image data. The final robustness checks worked well, and indicate that the NN would be a good model to use for similar types of data sets.

Improvement

Because the best results were achieved with a NN that did not use the images for input, it might be interesting to see if results could be improved by using an even smaller training vector size. For example, instead of using all 192 numerical

factors in a training vector, use only half of them, perhaps selecting the factors having the greatest standard deviation.

The Kaggle competition that was the source of this project ended in February 2017. This report was written approximately eight months after the end of the competition, and a summary of the winning model, submitted by Ivan Sosnovik, is available on-line [32]. Sosnovik used an ensemble method [33] employing techniques such as principle component analysis (PCA), logistic regression, and random forest. Using one or all of these techniques combined with the NN that produced the best results might also improve the model.

References

1. https://www.kaggle.com/c/leaf-classification

2. https://www.researchgate.net/profile/Charles_Mallah/publication/266632357_Plant_Leaf_Classification_using_Probabilistic_Integration_of_Shape_Texture_and_Margin_Features/links/547072f40cf216f8cfa9f72c.pdf

3. https://www.kaggle.com/c/leaf-classification#evaluatin

4. https://www.kaggle.com/c/leaf-classification/leaderboard

5. http://www.karensgardentips.com/botany-for-gardeners/botany-for-gardeners-the-leaf-shape/

6. http://www.karensgardentips.com/botany-for-gardeners/botany-for-gardeners-the-leaf-margins/

7. http://www.karensgardentips.com/botany-for-gardeners/botany-for-gardeners-the-leaf-texture-2/

8. http://colah.github.io/posts/2014-07-Conv-Nets-Modular/

9. https://www.tensorflow.org/get_started/mnist/pros

10. https://github.com/tensorflow/tensorflow/blob/r1.2/tensorflow/examples/tutorials/mnist/mnist_softmax.py

11. https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm

12. https://en.wikipedia.org/wiki/Euclidean_distance

13. https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

14. https://en.wikibooks.org/wiki/Artificial_Neural_Networks/Error-Correction_Learning

15. https://www.cs.cmu.edu/afs/cs/academic/class/15883-f15/slides/backprop.pdf

16. http://cs231n.github.io/convolutional-networks/

17. https://en.wikipedia.org/wiki/Convolutional_neural_network

18. http://jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks

19. http://mfile.narotama.ac.id/files/Umum/JURNAR%20STANFORD/Multimodal%20deep%20learning.pdf

20. https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/42455.pdf

21. https://en.wikipedia.org/wiki/Feature_scaling

22. https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf

23. https://www.tensorflow.org/api_docs/python/tf/train/AdamOptimizer

24. https://en.wikipedia.org/wiki/Softmax_function

25. https://www.tensorflow.org/api_docs/python/tf/nn/softmax

26. https://www.kaggle.com/xiaowang111/keras-convnet-lb-0-0052-w-visualization

27. https://www.kaggle.com/davidprakash/nn-kernel

28. https://en.wikipedia.org/wiki/Rectifier_(neural_networks)

29. https://en.wikipedia.org/wiki/Sigmoid_function

30. https://en.wikipedia.org/wiki/Signal-to-noise_ratio

31. https://en.wikipedia.org/wiki/Curse_of_dimensionality

32. http://blog.kaggle.com/2017/03/24/leaf-classification-competition-1st-place-winners-interview-ivan-sosnovik/

33. https://en.wikipedia.org/wiki/Ensemble_learning