Automated Pest Detection and Repulsion

A thesis submitted in partial fulfillment of the requirement
for the degree of Bachelor of Science in Physics from
The College of William and Mary

by

Justin Michael Christian and Andrew William Warrick

Advisor: Dr. William Cooke

Senior Research Coordinator: Seth Aubin

Williamsburg, VA

April 24, 2019

Contents

List of Figures

List of Tables

Abstract

With the goal of developing an artificial neural network to detect and repel various unwanted animals, we are in the process of examining and altering pre-existing neural networks: AlexNet and TensorFlow. For AlexNet, we trained the network to be able to detect the presence of a bird with moderate success, and then experimented with taking the differences between sequential images to improve accuracy. We used the latter method to isolate the regions with the largest sequential differences, and AlexNet achieved much greater success in detecting birds within the cropped region. The TensorFlow neural network allows for more adaptability since it is not pre-built like AlexNet, but also proves to be more difficult to set up to full functionality with our own input data. We modified our TensorFlow model to accommodate our own image data with the goal of achieving higher accuracy. With accurate enough models, we can eventually move to automation of training and real-time image analysis.

Chapter 1
Introduction

Our goal was to develop a fully automated system for repelling various creatures whose presence could be detrimental to business or personal practices (e.g. harming vegetable gardens or invading bird feeders). Various passive forms of repellant, for example scarecrows, have been employed in the past with varying degrees of success, but passive systems tend to lose potency over time. An active system such as our own should retain effectiveness much longer. We decided that machine learning, specifically neural networks, would be a good approach, given its successful use in identifying hazards on a road for self-driving cars, live facial recognition, etc., which suggests it would have a capacity for live pest detection as well. With this goal in mind, we each have chosen two separate neural network architectures to explore: AlexNet and Tensorflow. AlexNet has the benefit of being pre-built and easy to modify, whereas the more ground-up approach of TensorFlow could allow us to optimize the neural network more specifically for our task. Andrew is working with AlexNet, and Justin is working with TensorFlow.

1.1 AlexNet Introduction

For easily modifying a pre-existing network, AlexNet has the advantage of possessing a final output layer designed to be substituted with any number of possible outputs, depending on the user's intended purpose. It also comes with a free tutorial that can be found on Matlab's

Onramp page to provide easily accessible information for this implementation [1]. Thus, with little effort one can easily prepare AlexNet to train a novel image set. Much harder, however, is the task of ensuring that AlexNet does not overfit the data and that it can make accurate predictions when given untrained images. Thus, our goal was to adjust our image inputs, the internal AlexNet parameters, and to experiment with image averaging and sequential differences in order to prepare AlexNet to deal with new images. In doing so, we hope to develop an approach that could be extrapolated to images coming from different environments with the task of determining the presence of completely different creatures --e.g. a squirrel instead of a bird.

1.2 TensorFlow Introduction

The goal for working with TensorFlow is to create a neural network from basic components and develop it into a model closely fit to our needs. Our current goal is to increase the accuracy of our working model to at least 90% and apply it to more realistic data sets. So far we have been able to introduce our own images that we have taken, but we have had to modify them to the point where their connection to the real-world scenarios we envision is greatly impacted. Our model can be trained and saved so it persists in computer storage, but until it can be applied with reasonably high accuracy to real-world data, the focus is on tweaking the network.

Chapter 2

Theory

Our research delves into machine learning through the use of artificial neural networks, a rapidly developing computer learning technique that uses logical thresholds to predict an object's classification. Neural networks use a series of input layers with different weights and biases connecting them to each neuron of the next layer, as shown in Figure 2.1. Weights and biases are numerical values that are initially assigned randomly to each neuron, which serves to prioritize some neurons over others. Each neuron in the initial layer is assigned a pixel (or batch of pixels), so the weights and biases determine how much a given pixel or region of the image is prioritized [2]. After running a pre-classified batch of images -- called the training images -- through the system, the neural network observes how well the various categories ended up being separated in the final output layer. The network then modifies the weights and biases and runs a batch of training images that were set aside through the system to see if it ends up with a better or worse separation of categories. If it results in a better separation of categories, it will continue to alter the weights and biases in that direction until the best separation of images is reached. Each iteration of altering weights and biases and running the training images through the network is called an epoch. If a good separation of images occurs, this means that a pattern in the pixels was found that images of one category shared which the others did not [3].

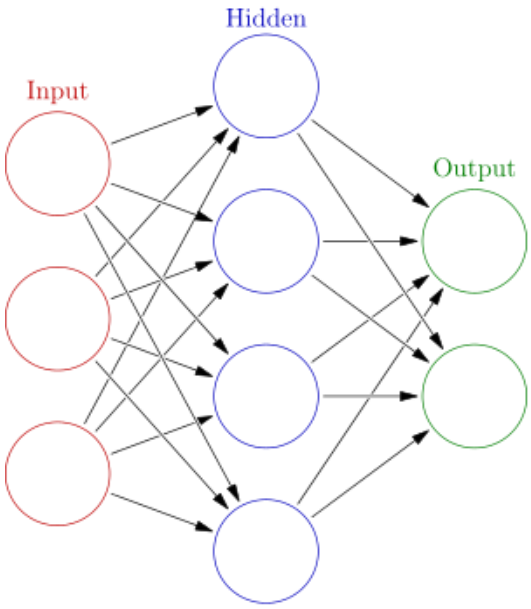

Figure 2.1: This graphic represents the basic structure of a simple neural network [4]. Each circle represents a node (i.e. neuron) in the network layer; a node is responsible for individually processing data fed into it. Data is inputted into the first layer, which begins the initial processing. Every node in the input layer then passes on the information it processed into every node of the next, hidden layer. This process continues until the data reaches the output layer, which is responsible for determining what the data fed into the network represents (e.g. type of animal shown in an image file).

2.1 AlexNet Theory

AlexNet is a convolutional neural network, which is a class of neural networks well-suited for image analysis (see Figure 2.2). It has 25 layers, and operates by creating pooling layers, where the outputs of one pixel region are fed into a single neuron in a successive layer. This process is then repeated at the next layer, to make a yet smaller layer composed of the adjacent neurons of the previous one. This method is particularly useful in analyzing images because it can take into account proximity when trying to make out a pattern for the purpose of qualification.

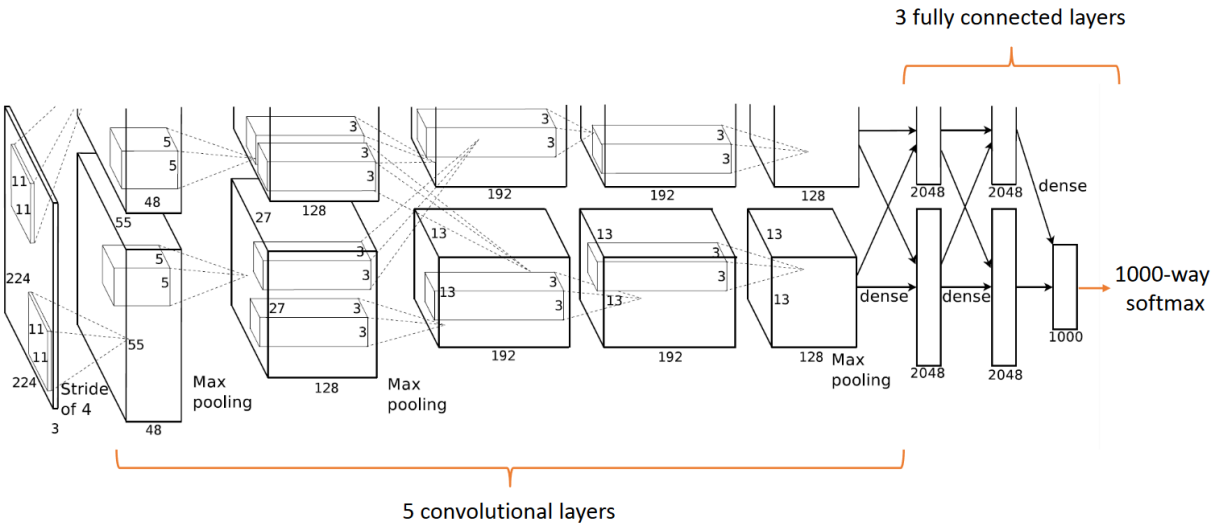

Figure 2.2: Diagram of AlexNet's architecture [5]. A convolutional neural network takes into account the proximity of pixel data within an image. It does so by breaking down an input image into a set of regions, where each region is treated as one entity processed separately from other regions. In this diagram, the network processes the data with convolution and pooling before passing on data to fully connected layers, the standard data processing layers.

The most important layers in AlexNet are the convolutional and fully connected layers, of which there are five and three, respectively [5]. The convolutional layers we have already discussed; the fully connected layers connect every neuron of the previous layer after the convolutional process, and the numbers assigned to each neuron will be used to determine the class score (in our case, whether or not there is a bird) [6].

2.2 TensorFlow Theory

TensorFlow is an open source neural network library which offers advanced customization options for a neural network model with a broad range of possible applications. A

TensorFlow network needs to be built “from scratch”, meaning that in order to implement a network you must write all of the code for it, and train the new network yourself. This allows us to create a network with any number of layers where each layer can be selected from a range of various layer models with different functionalities. TensorFlow also offers the ability to create a convolutional neural network, but we are not implementing one with it at this time.

The customizability of TensorFlow allows us to adjust various parameters of our model with ease. Besides making adjustments to the layer structure of the model itself, one of the most important aspects of the model is the training optimizer. This algorithm determines how the model adapts its neural weights after each training epoch, thus holding great influence over how the model learns. TensorFlow offers several options for optimizers, with the most robust algorithms being Adam and SGD (Stochastic gradient descent) [7]. Each algorithm itself contains further parameters which can be varied to modify the behavior of the algorithm. Currently, we have been working solely with the Adam optimizer: its parameters are learning rate, epsilon, beta1, and beta2. We have only been adjusting the first two parameters; the publication introducing Adam says that the default value of epsilon will not fit every situation, and adjusting the learning rate can tackle overfitting. The learning rate effectively determines the range of magnitude within which the neural weights may change after each epoch, while epsilon is a small constant introduced to prevent dividing by a near-zero number when calculating weight corrections. The two beta parameters are exponential decay rates for moment estimates, but they are not as frequently changed. For more information on Adam and its parameters, see Kingma et al., 2014 [7].

Chapter 3

Progress

3.1 AlexNet Progress

For AlexNet, we analyzed image data taken with a Raspberry Pi camera aimed at a residential bird feeder. We separated the files that contained birds from those that did not. We then altered the final output layer of AlexNet so that it would only split the images into two categories: bird and no_bird. 60% of the images were used to train the network, and the other 40% were used afterwards to test the network after it had been trained.

3.1.a First Attempts

After adjusting the learning rate from 0.01 to 0.001 (the former being too high to successfully train our images with), we made our first attempt running AlexNet. After 30 epochs, AlexNet successfully divided its bird from no_bird training images with a 99.22% accuracy. However, when it came to the untrained test images, it predicted the presence of a bird with only a 45.10% accuracy, worse than the 50/50 odds one would expect from flipping a coin. This is a good example of a neural network overfitting, where AlexNet found a pattern that existed only in the particular images it trained with.

In our second attempt, we altered our approach by cropping our images to focus solely on the bird feeders. This way the birds would make up a much larger portion of the image, and our neural network would not have to analyze as much irrelevant data. This approach proved to be

more effective, and resulted in a prediction accuracy of 73.86% -- a result that leaves something to be desired, but is significantly better than flipping a coin, and proves that our neural network has some capacity to recognize birds.

	Epoch 30 Final Accuracy	Prediction Accuracy
Attempt 1 (no cropping)	99.22%	45.10%
Attempt 2 (cropping)	100%	73.86%

Table 3.1: This table lists the predicted and actual performance accuracy of AlexNet when analyzing our image files. An epoch represents the entirety of the dataset: one epoch concludes when the network has processed the entirety of the test images.

3.1.b Image Differences

To help improve accuracy in distinguishing birds from no bird files, we experimented with taking averages of the pixel values of both categories. Because our camera gave a red tint to some of our pictures, we created a subroutine that would calculate the average pixel value for the red, green, and blue layers of an ideal image and then normalize said values for each individual image by comparing it with the ideal image. We also experimented with taking the difference of sequential images in order to isolate regions where change had occurred between the images, so that by this means we could auto-crop the images to observe the most interesting regions and ignore the rest. A diagram showing how image differences work can be seen in Figure 3.1. We hoped by these means to improve the neural network's accuracy.

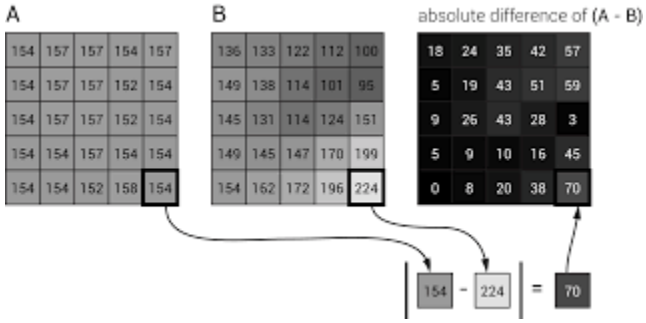

Figure 3.1: A diagram illustrating the process of computing the pixel-wise difference in image data [8]. A pixel from image A and the corresponding pixel from image B have their difference taken, with the result being computed as an absolute value to avoid negative pixel values. This result is stored in a new image, which now shows visual information where the two images differed.

To start, we created an algorithm that sorts each image within its bird or no_bird subfolder by the photo number. We then looked to see whether the next file in the bird folder was the next photo taken sequentially; if not, the code would search through the no_bird folder to find the missing photo, and would subtract it from the bird image to create a difference image. Since bird images tended to follow other bird images and vice versa, there were only 23 places where there was a shift from an image containing a bird, with the following image not containing a bird. We then created a montage of these difference images, with the result being mostly blank pictures except around the birds and the edges of the birdfeeder sometimes, where the light reflection changed frequently (Figure 3.2).

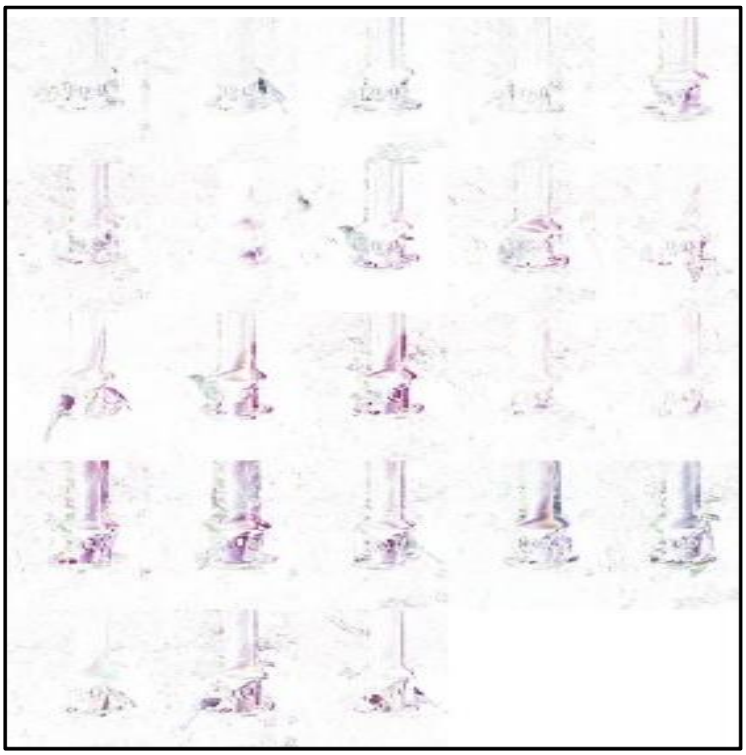

Figure 3.2: A montage of image differences. These images were produced by sequential image differencing. Whenever a bird would disappear from the image, we would take the difference between the image containing the bird and the subsequent image where the bird disappeared. This resulted in images where the only visible features are the bird feeder and the bird itself, which is why the image as a whole is mostly blank.

3.1.c Auto-Cropping

We decided it would be better to develop an algorithm that could determine where an area of interest was, rather than to predetermine where the most likely area of interest would be. Thus, we set on using the image difference method to find out where the largest change of pixels occurred. To ensure that we did not select an anomalous pixel as our center, we lowered the picture resolution so that each new pixel would represent an average of 4 original pixels. The new pixel that would become the center of our cropped image was the weighted average of the position of the largest difference in the red, green, and blue arrays, where the magnitude of the

pixel in each array determined how much it was weighted. We first tried this method by taking the absolute value of the difference between each image and an ideal birdless picture, the thought being that whenever we had a picture with a bird, the greatest difference between the images would occur where the bird was. However, due primarily to the changes in lighting throughout the course of the day, this method did not work very well for huge sections of our images, where the greatest difference between the images would occur where there was greater or lesser light reflection on the feeder. Thus, we instead created a code that would subtract each image from the previous one, as opposed to a standard one. The code would then find the maximum pixel values of the resultant image, and crop a 20x20 image around the corresponding region in the original, unmodified photo.

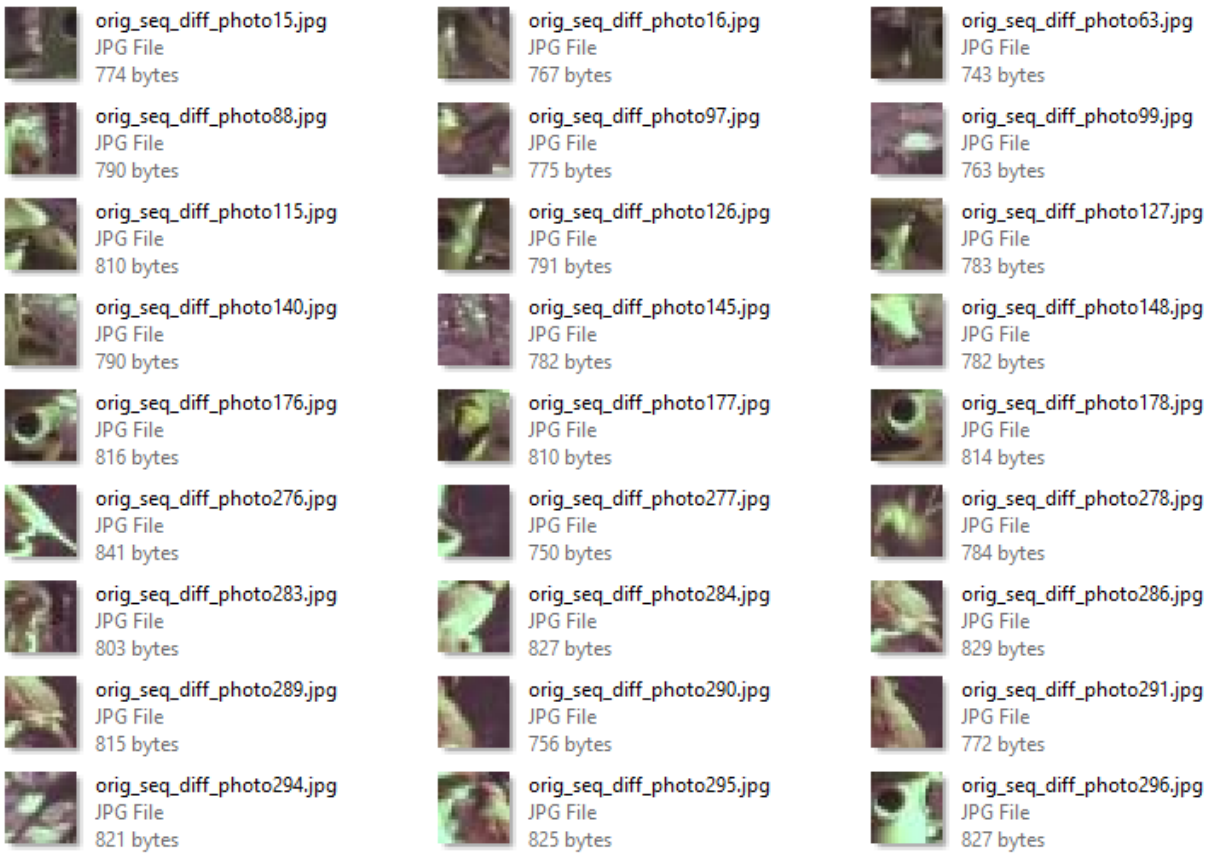

Figure 3.3: A sample of images created using the sequential difference method. Each image was created by subtracting the pixels of a given image by a previous one, and then cropping around the region in the original photo that corresponded with the maximum pixel values in the difference image.

After creating a new folder for these images, we decided to train AlexNet with them. The initial results were not very good, because the labels corresponded with those in the original files. However, since the cropped region was determined using the absolute value of the difference between two consecutive images, the unfortunate consequence is that it would not distinguish between a region where a bird just arrived and a region where one just left. Also, if a bird remained relatively still between two photos, that region would not likely be selected as the most interesting by our code. Thus, we decided to manually reclassify the resultant images and to re-train the network with them. Our results with this approach were far superior, obtaining a test image classification accuracy of 92.13%, which is 18.27% better than the best attempt using just a manually cropped original image.

	Epoch 30 Final Training Accuracy	Prediction Accuracy
Attempt 4 (not relabeled)	97.66%	56.54%
Attempt 5 (relabeled)	100%	92.13%

Table 3.2: A chart showing the AlexNet's accuracy in training and labeling test images before and after manual reclassification.

3.1.d Experimenting with Larger Images

After this, we created a code to store the values of the cropping coordinates, and used these to mark off the cropping region in the original photo with a red square, for the purpose of analyzing how well the method picked out bird locations. We found that the cropped region often only picked out a small part of the bird or -- due to the nature of using the absolute value of image differences to find the maximum point of change -- right next to where a bird was. Thus, we quadrupled the size of the cropped region, reclassified the images accordingly, and then re-

trained AlexNet. To our dismay, this actually slightly lowered our prediction accuracy to 88.89%. This is likely because some types of birds -- most notably the crow -- were nearly impossible to see in the 20x20 photos, and were thus not classified as birds at all during the manual classification stage. They were, however, clearly visible with the 40x40 images, and thus I classified them as birds; indeed, there were 34 more bird images in the 40x40 files than in the 20x20. This means that AlexNet would have had to find a pattern that would encompass both a common songbird and a crow, a difficult task considering we only had 103 bird images for training.

Figure 3.4: The difference between a 20x20 (left) vs a 40x40 (right) cropped photo of the same bird, with a less cropped version of the image for reference.

3.1.e Final Attempts

We attempted to artificially increase the number of training images by making a code that would run each 40x40 image through a gradient, where the brightness would increase from left to right for one new set of images and then from top to bottom for another set, increasing the number of training images by a factor of three. Training with this set resulted in a prediction accuracy of 89.60%, only marginally better than the 88.89% without the additional images. The nearly identical results demonstrates that, through its convolutional and regularization layers, AlexNet effectively filters out the noise of any changes in brightness it determines to be irrelevant in classifying the images.

	Epoch 30 Final Training Accuracy	Prediction Accuracy
Attempt 6 (40x40 auto-cropped)	100%	88.89%
Attempt 7 (40x40 with gradients)	100%	89.60%

Table 3.3: Final AlexNet tests using 40x40 auto-cropped images. Attempt 6 artificially tripled the training images using gradient sets.

3.2 TensorFlow Progress

3.2.a Overview

While TensorFlow does offer the opportunity to work from the ground-up, we do not necessarily want to start completely from scratch. We were able to find an informative, fairly generic example where a simple neural network was applied to an image database, something closely suited to our needs. From there, we worked to introduce our own image data rather than the ones they were working with to see how their model performed on our own pictures.

However, introducing our own data was not as simple since the entire built-in TensorFlow image loading process was encapsulated into one function which only worked on

pre-compiled databases. Our pictures did not match the expected format either in terms of dimensions and color aspect. One other important aspect of our data which did not match theirs: we have many fewer unique images. Given the size of our own data set (~1000 images total) and the total amount of neurons in their model, the model is prone to overfitting as it does not have enough data to learn the relevant patterns. This means that the model will perform with higher accuracy on the training set of data than on the testing; it does not learn what it is supposed to. If we left the images as they are, the total number of parameters within each image, i.e. pixels, far outweighs the number of unique image we possess, leaving no chance for our model to learn anything useful from the data. This issue is tackled in a few different ways which are detailed in the Image Manipulation section.

Once our image data was in the right condition and integrated into the model, we began modifying various parameters of the model to try to reach ~90% accuracy. Before we could even begin seeing promising results, we had to overcome one significant problem: drastic overfitting from an unknown source. This was eventually solved by randomizing the selection of training and testing images from our set of ~1000. Once that was solved and we saw accuracy rise to around 70%, we could move forward with adjusting parameters.

3.2.b Initial Project Framework

We encountered a GitHub tutorial [9] for TensorFlow written in Python which offered a fairly generic implementation of a neural network. This tutorial is useful because it can be modified to explore the functionality of TensorFlow but also to act as a possible working model for our pest image data analysis. This tutorial downloads image data from the Fashion MNIST dataset, a collection of 70,000 grayscale images of articles of clothing. 60,000 of these images are used to train the simple neural network they create, while the remaining 10,000 are used to

test the accuracy of the model. It then builds a three-layer network; the first layer is declared as a flattening layer. This layer takes the two-dimensional data and flattens it into a one-dimensional vector -- no training of the model occurs in this layer, and no parameters or nodes are contained here. This flattening likely causes the data to lose some information regarding pixel proximity; this issue is tackled in other tutorials through convolutional layers, which are more often implemented for image processing. The second and third layers are densely-connected layers, containing 128 and 10 nodes, respectively. These are the standard type of neural network layer, where each node in one layer is connected to every node in the adjacent layers. For these two layers, they specify a particular activation function: tf.nn.relu for the hidden layer and tf.nn.softmax for the output. By specifying the activation function, the behavior of each layer is modified. A tf.nn.relu layer behaves like a standard neural training layer, generating tensors (weights) applied to each node. For a 128-node dense layer, there are 128 * 784 weights, 100,352 parameters to learn. The tf.nn.softmax layer generates probability scores for each category of data, returning a vector containing these scores whose total sums to 1. To see the code for this network structure, see Figure 3.5.

```
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(10, activation=tf.nn.softmax)
])
```

Figure 3.5: Structure of the fashion example's model [9]. This is a very simple model with only one hidden layer which contains 128 neurons, or nodes. It outputs a vector for every image containing 10 values, where each value represents the model's confidence that the image belongs to a particular class. The input_shape seen in the flattening layer refers to the image dimensions: 28x28 pixels.

This TensorFlow tutorial was contained within the Google Colaboratory environment, a web-based IDE which comes pre-built to incorporate the TensorFlow libraries. We could write python code and build models at our convenience, but it was not conducive to automation and self-sufficiency that are necessary for the final stages of the project. Instead, we installed Docker Toolbox and its various software prerequisites. Docker is a container service, meaning the user creates a virtual machine via Docker which hosts various "images" (i.e. file systems). Basically, the user can manage different virtual file systems which for our purposes can store various Python libraries we might need to use. The reason Docker is endorsed by TensorFlow is that Docker has built-in functionality to "pull" a TensorFlow image from cloud storage and essentially set it up for the user within the virtual machine. Once the TensorFlow image is pulled, we can execute TensorFlow scripts within that image.

We translated our Python code from Colaboratory into a single Python script, responsible for training, testing, and saving neural networks. To run the script, we have to initialize our Docker TensorFlow image while mounting our host machine's file system onto the virtual one. This allows us to read in our own pictures, stored on a personal computer. We then can execute the script within the Docker image while accessing our own image files, thus implementing a process which can be recreated on any computer we would need, including a Raspberry Pi.

This tutorial provides a basic framework to implement image processing. In order to modify it to match our needs, our efforts can be broken down into two major portions: image manipulation and network parameters. We needed to introduce our own data to the model to even begin our project; once this was done, we could begin fine-tuning our parameters to attempt to increase our model's accuracy.

3.2.c Image Manipulation

In order to develop a model that can tackle our problem, we needed to load in our own image data to the TensorFlow environment. The TensorFlow neural network model accepts data in the IDX format, a simple image format which consists of a 2-dimensional array of pixel data, where each pixel has a value ranging from 0 to 255. A pixel value of 0 corresponds to a completely white pixel, whereas 255 represents a completely black pixel. The IDX format does not support color images. For an example of how these files appear visually, see Figure 3.6.

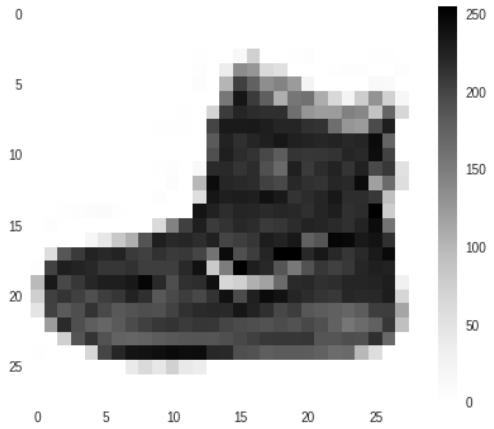

Figure 3.6: A sample test image generated from the Fashion MNIST Database [9]. The format is 28x28 pixels in grayscale, with a pixel value of 255 corresponding to the darkest pixel value. In this case, this is an image of an ankle boot, one of the 10 categories of articles of clothing in the database.

However, the incoming data from the Raspberry Pi camera is in .jpg format with 640x480 pixels in color. Using the image processing app XnConvert, we have been making various modifications to the images, from cropping and compressing to converting to grayscale. We used the same base images as AlexNet, but they deviate in the extent that they are cropped and compressed. Not only do we have to worry about matching the image format, but we have to

consider the volume of data that is required to properly train our model. There are two solutions to this issue: cropping and/or compressing the images. Cropping allows us to focus the model's attention on the most interesting or important part of an image, but for the case of TensorFlow this has been done manually thus far. We also compress the images to reduce the number of pixels while also reducing noise, but information is also lost in the process. The images must also be converted to grayscale for them to be compatible with our TensorFlow model; there must only be one value associated with each pixel, which is or course not true for color images.

TensorFlow does not offer any built-in utilities for dealing with image data in various file formats. Although we could modify the images to match the qualitative needs of our model, we also had to be able to actually access the files and load them into the script. This is not so straightforward if you are not using a pre-compiled database like Fashion MNIST; we could not simply call one load function and have our image arrays with corresponding labels made for us. Instead, we import Python Image Library (PIL) to handle the image data. In order to access the images, we had to separate them into separate folders based on their category and store them in the public user directory so they are available to the script. Only then can we read in each image, knowing what label they should have, and store them in our own compiled arrays.

We then had the framework for introducing our own image data. But we were not done with handling images; we are still making modifications to the image data to find the ideal format and volume of data to suit our model. By adjusting the level of compression and cropping particular portions of the images, we are able to optimize our images to fit the functionality of our model. It may very well be that our current model is close to handling the goal we have envisioned for it, but the area that we are most restricted with is our volume of data. The number of unique images we possess limits not only the size of our network, but also that of the volume

of data (i.e. number of pixels) that can be associated with each image. If we have too many pixels per image, the model will be very prone to overfitting, or fail to find the small but pivotal difference between our images when there is so much commonality to get caught on. As an example of a healthy ratio of images and pixel data, the fashion MNIST database has 70,000 images who each contain 784 pixels. This may be a higher ratio than strictly necessary, but it certainly is a lot better than our ~1000 images with 300,000 pixels apiece. This is what the cropping and compressing addresses, which has led us to working with 100-pixel images instead. We have cropped to the point where only the bottom of the bird feeder is in the image (given its high volume of bird traffic) but we still require about 10x compression, which is likely too much to yield the best results. Ideally, we would try to match the number of neurons in our hidden layer to the number of inputs (pixels per image), in this case doing so by limiting the number of input [10]. The cropping also has led some images to be misclassified given that the bird could have been in a cropped section of the image.

Another approach to improving our data artificially inflated our data with several forms of picture transformations. We applied several different actions on our original data set to produce "new" images which are distinct from the old ones without introducing the possibility of confusing the model (i.e. introduce "new" birds in pictures that should not have them). This was done by rotating the images in 90° increments and through adjusting the bit depth of the images. To see example of these actions, see Figure 3.7. We have essentially created a new data set 5x the original size, preserving the uniqueness of every image. However, after training the model on both the original set and the inflated one, the model actually performed a few percentages better on the original set (around 83% currently). This approach does have merit, however, it just seems to not yet be beneficial in our case [11].

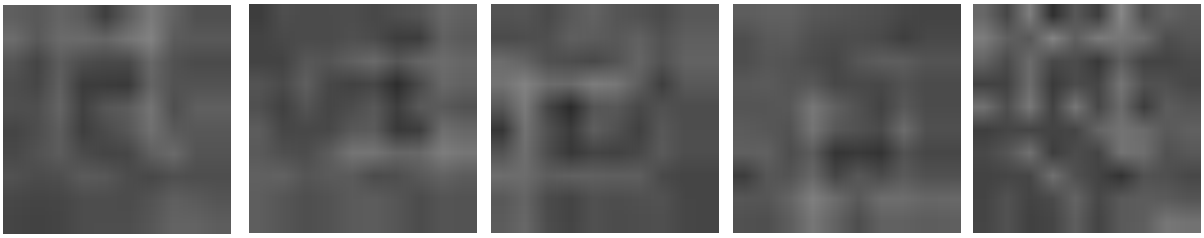

Figure 3.7: Sample images of original picture with 4 variants. From left to right: original, 90° clockwise rotation, 90° counterclockwise rotation, 180° rotation, adjustment from 255-value scale to 8-value scale. All 5 of these images are selected randomly to either be a training or testing image.

Another important aspect of the image data is the means by which we determine which images are designated as training versus testing. When we first introduced our image data to the script, we took some fraction of files in the image directories and placed them in a training image array. Importantly, the images we took from the file system were always the same ~600 images that the model would then train on. Naturally this meant that the testing images were always the same, as well. This meant that any time we trained and then tested the model, we received and accuracy of about 50% ± 5% without fail. To combat this, we introduced a method to randomly select the training and testing images by first shuffling the arrays that contain the source images. This simple method, in combination with the image variants mentioned above, saw our accuracy jump to around 70% after repeated bouts of training and testing.

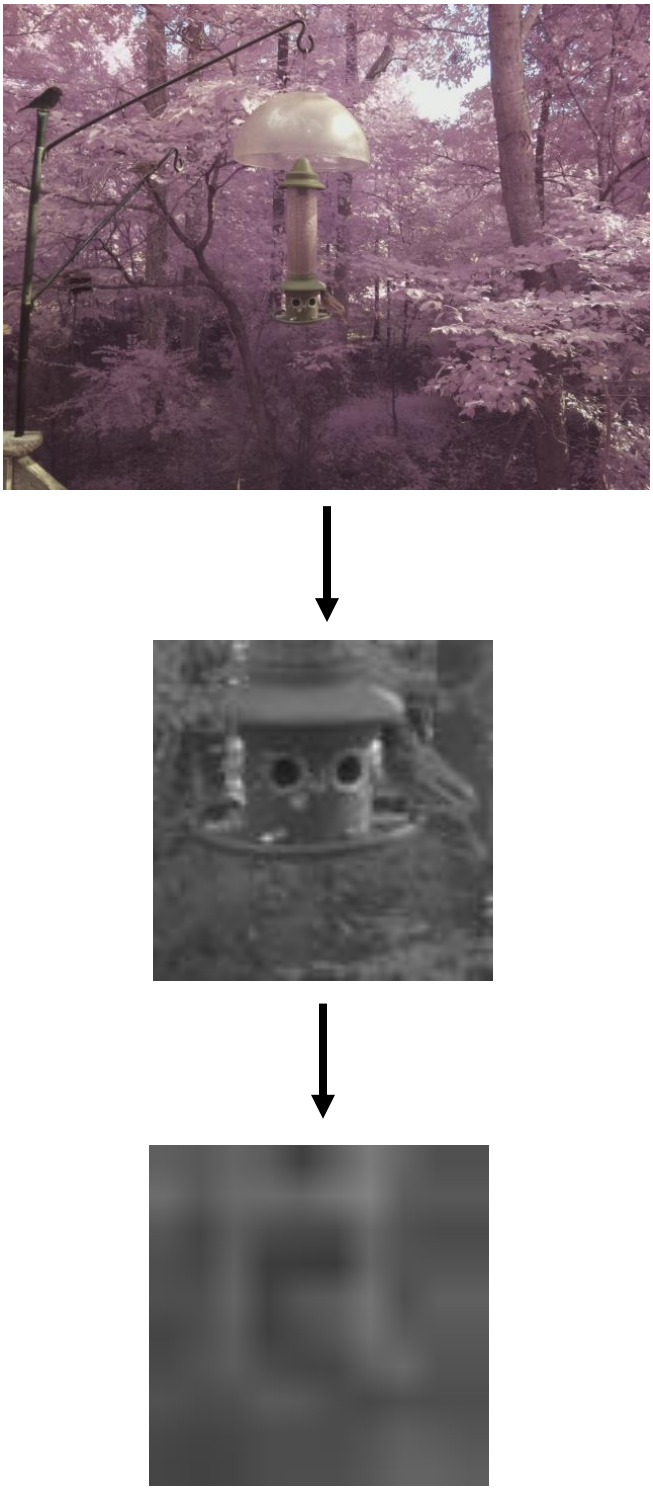

Figure 3.8: Progression of image processing from original full-color image to grayscale, cropped, compressed image. Note how the final product is completely meaningless upon manual inspection, but our model has gleaned some pattern from it.

3.2.d Developing Network Model

Given that the Fashion example's model was simple and functional, we made no modifications to it until after the Python scripts were running as intended. We attempted to apply it to our own images, cropping and compressing to various degrees, but the model would overfit and only predict the same class for every test image. Both the data we inputted and the parameters of the model itself led to this lack of functionality. To address this, we made changes to the model's structure, optimizer, and image data to attempt to make a trainable model.

```
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(10,10)),
    keras.layers.Dense(50, activation=tf.nn.relu),
    keras.layers.Dense(1, activation='sigmoid')
])
```

Figure 3.9: The current TensorFlow model we are implementing. Keras is the neural network library from which much of the code originates. Note that the first layer does not train any parameters, but simply reformats the data into a one-dimensional array.

The layers of our model can be changed in several different ways, including the total number of layers we implement. After some experimentation, the number of layers and their activations (i.e. functionality) have been kept the same, but we have changed parameters of the layers themselves. One important difference between the fashion example and our own current implementation is that we are only trying to conclude if there is a bird present in our images; we have binary classification rather than 10 class possibilities. This means that our output layer has to be modified so it only outputs one classification or the other: the ideal activation for this is sigmoid, which is designed specifically for binary classification.

The hidden layer, where the network actually contains parameters to learn, has been modified so it only contains 50 nodes rather than 128. The reason for this is that a layer

containing fewer nodes forces the model to pick out the most important features of the data, which is particularly useful when the data has little variation [10].

We have experimented with using both Adam and SGD algorithms, with the current working choice being Adam. We have adjusted the learning rate and epsilon parameters of the algorithm, but so far have found no conclusive result with any of the changes. They are now left at the default values. The only other change that has been made to the optimizer is swapping out the standard loss function, which measures the accuracy of the training, for one designed to handle binary classification. To fine tune our model further, the optimizer is certainly an important place to make further adjustments.

Finally, we have increased the number of epochs when training to 150; between this and the hidden layer size of 50 neurons, we have achieved 85% accuracy when training and roughly 83% accuracy when testing. This suggests we are still overfitting a little, but this does not come as a surprise given all the circumstances mentioned previously. Again, these values vary each time a new model is trained, but we are mostly looking for a qualitative increase in accuracy to determine the efficacy of adjustments made to the model, particularly when these changes have been as dramatic as they have thus far.

We also have created a few graphs delineating the output generated by the network after training. Given that we are using the sigmoid activation for the output layer, we would expect an ideal dataset and model to output a true sigmoid curve. We can see in our test cases that we achieve a fairly reasonable sigmoid curve, but there are some qualitative differences that are immediately noticeable.

	Trial	Testing Accuracy
Before Shuffling	Introduced Data	~50%
	Crop and Compress Images	~50%
	Re-Crop Closer, Compress Further	~50%
	Modified Model Parameters (# of Nodes, # of Layers, Optimizer)	~50%
	Introduced Image Transformations	~50%
After Shuffling	Introduced Shuffling of Training, Testing Images	~70%
	Increased Hidden Layer Neurons to 50	~80%
	Revert to Image Set w/o Transformations	~84%

Table 3.4: Resulting model accuracy after each stage of research. The most important realization was the necessity of shuffling the selection of training versus testing data each time we train the model. This implies that there perhaps was some time dependence in the data (e.g. position of the bird feeder as it swung freely) or other factor which was local to a particular subsection of the data.

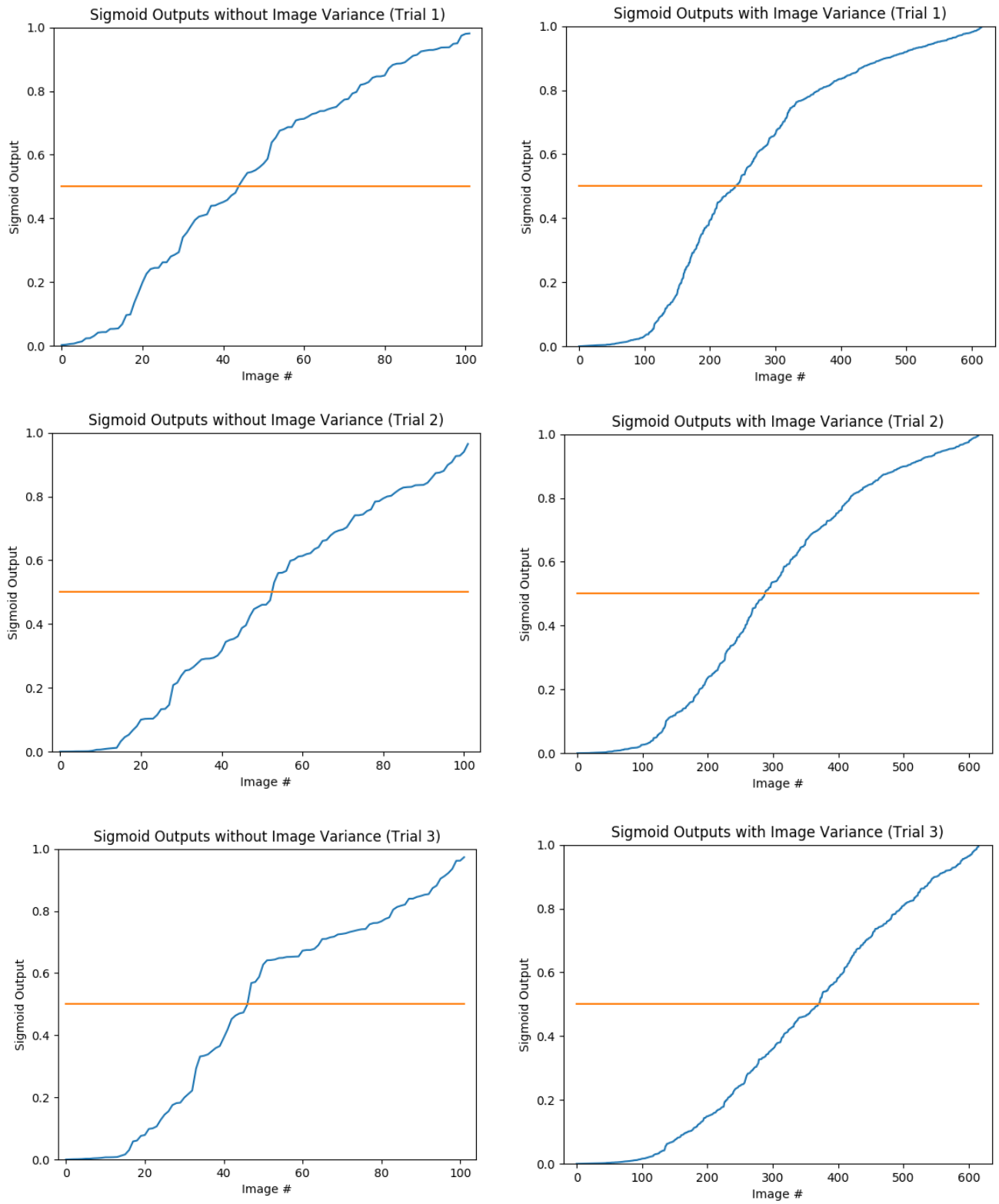

Figure 3.10: Sigmoid outputs from our model after training on images with variants (expanded image set) and on the set without. Interestingly, though the curves are smoother and appear more qualitatively like a sigmoid function, the test accuracy for the model trained without variances is higher (~83% versus ~80%, again always varying).

Chapter 4

Conclusion

4.1 AlexNet Conclusion

While some fine-tuning could still be done, AlexNet -- when coupled with our auto-crop technique -- has demonstrated a remarkable accuracy in the detection of birds, considering the relatively small sample it had to work with. While the 40x40 images have thus far performed worse than our 20x20 ones, the flaw was likely due to different kinds of birds making it more difficult to reliably train the network for the larger images, which is a problem that would not carry over if we were merely trying to detect the presence of one kind of pest. The desired cropping size may well vary depending on the circumstances of the images being analyzed. For future work, it may be useful to look for online databases of squirrels, and then to properly size those images based on the anticipated cropping value. At that point, the network as we have designed it should be ready to begin live detection testing.

4.2 TensorFlow Conclusion

TensorFlow has taken a long time to become as functional as AlexNet, and it has yet to reach the accuracy that AlexNet has. However, we are dealing with a limited data set towards a specific purpose which is not the same goal we hold for the final product. TensorFlow still seems to be a powerful approach to problems like these, but it has taken a lot to get to that point. We would like to raise our evaluation accuracy from ~83% to at least 90% if we want to transition to

the next stage of the project. A good step forward would be to investigate switching to a convolutional network, whose functionality is fully supported by TensorFlow. We also could consider re-introducing image transformations, something which has precedence in other studies. Perhaps most crucial would be to collect more robust data to feed into our model. This seems to be a significant bottleneck in the logistics of our TensorFlow network.

4.3 Future Work

Once we are able to reliably identify pests in our images with high accuracy, we will move on to the challenge of analyzing live video data. We will need to take into consideration the challenges introduced when implementing our networks in different environments; thus far we have only been using images taken of the same bird feeder at the same location. There are further complications associated with using video data rather than pre-recorded images: automating the image processing, classification, and consequential instructions the pest-repelling device should follow. In order to repel detected pests, we are currently considering developing a gentle water gun guided by the camera, but the logistics of such a device are not currently under investigation.

Appendix A

Code Samples

A.1 List of MATLAB Scripts

bird_or_no_bird: this code creates AlexNet, splits the image datastore into testing and training categories, modifies the final layer around the bird/no_bird categories, alters the learning rate, trains the network, and then tries to predict whether there is a bird in each test image.

```
net1 = alexnet;

[trainImgs,testImgs] = splitEachLabel(SDD,0.6);
numClasses = numel(categories(SDD.Labels));
layers = net1.Layers;
fc = fullyConnectedLayer(numClasses);
layers(23) = fc;
cl = classificationLayer;
layers(25) = cl;
opts = trainingOptions('sgdm', 'InitialLearnRate', 0.001);
[birdnet,info] = trainNetwork(trainImgs, layers, opts);
testpreds = classify(birdnet,testImgs);
```

bird_test: determines the success of the network and creates a confusion matrix to determine the number of false positives, false negatives, true positives, and true negatives.

```
bd = imageDatastore('BIRDS', 'IncludeSubfolders', true,
'LabelSource', 'foldernames', … 'ReadFcn', @wormread);
[trainImgs,testImgs] = splitEachLabel(bd,0.6);
f = figure;
plot(info.TrainingLoss)
```

```
testpreds = classify(birdnet,testImgs);
birdActual = testImgs.Labels;
numCorrect = nnz(testpreds == birdActual);
fracCorrect = numCorrect/numel(testpreds);
f2 = figure;
confusionchart(testImgs.Labels,testpreds)
```

orig_seq_diff: this code sorts the bird files by photo number and rearranges the labels accordingly. It then creates an array of the absolute value of each image subtracted by the previous image, and then finds the largest pixel value in a lower resolution photo for the red, green, and blue arrays, and uses those values to determine the center for cropping. It then creates a new file for each cropped region, with the file name based on the for-loop index. The new photo number thus corresponds with the original photo number.

```
global img_standard
bdf = imageDatastore('Birdfeeder', 'IncludeSubfolders', true,
'LabelSource', 'foldernames');
img_standard = readimage(bdf, 95);
% creates a standard image for the red, green, and blue array
averages

bdx = imageDatastore('BIRDS', 'IncludeSubfolders', true,
'LabelSource', 'foldernames', 'ReadFcn', @bigwormread3_no_crop);
B = bdx.Labels;
A = sort_nat(bdx.Files);
bdx.Files = A;
bdx.Labels = B;
% sorts labels and files

sz = size(A, 1);
sb = nnz(B == 'b');
snb = nnz(B == 'n b');
n = 1;

for i = 1:sz
     C = regexp(A(i),'\d*','Match');
     for ii= 1:length(C)
     if ~isempty(C{ii})
          Num(ii,1)=str2double(C{ii}(end));
     else
          Num(ii,1)=NaN;
```

```
     end
     end
     A(i) = num2cell(Num);
end
% lists file numbers only accord to the photo number

[firstsorted, firstsortorder] = sortrows(A);
bdx.Files = bdx.Files(firstsortorder);
bdx.Labels = B(firstsortorder);
% sorts files and labels based on the photo numbers

arraySize = [480 640 3];
result = cell(1, sz);
for q = 1:sz
     result{q} = zeros(arraySize);
     if (q-1) > 0
     result{q} = uint8(abs(int64(readimage(bdx, q)) -
int64(readimage(bdx, (q-1)))));
     end
end
% finds the absolute value of the sequential image differences

result1 = result;
for i = 1:sz
     orig_img = readimage(bdx,i);
     img = result1{i};
     img1 = imresize(img, 0.25);
     r = img1(:,:,1);
     g = img1(:,:,2);
     b = img1(:,:,3);
     rm = max(r(:));
     gm = max(g(:));
     bm = max(b(:));
     [rr, rc] = find(r == rm);
     [gr, gc] = find(g == gm);
     [br, bc] = find(b == bm);
     row_norm = 4*(int64(rr(1))*int64(rm) +
int64(gr(1))*int64(gm) +
...int64(br(1))*int64(bm))/(int64(rm)+int64(gm)+int64(bm));
     column_norm = 4*(int64(rc(1))*int64(rm) +
int64(gc(1))*int64(gm) +
...int64(bc(1))*int64(bm))/(int64(rm)+int64(gm)+int64(bm));
     img = imcrop(img,[(column_norm-10) (row_norm-10) 20 20]);
     result1{i} = img;
     orig_img = imcrop(orig_img,[(column_norm-10) (row_norm-10)
20 20]);
     imwrite(orig_img, sprintf('orig_seq_diff_photo%d.jpg', i));
```

```
end
% finds largest region of the image difference photo, and
creates a cropped
% image around that region

-bigwormread3_no_crop: subroutine that normalizes each color
array based on an ideal photo.

function img = bigwormread3_no_crop(file)
     global img_standard
     img = imread(file);
     Ravg = mean(int64(img_standard(:,:,1)), 'all');
     Gavg = mean(int64(img_standard(:,:,2)), 'all');
     Bavg = mean(int64(img_standard(:,:,3)), 'all');
     x = mean(int64(img(:,:,1)), 'all');
     y = mean(int64(img(:,:,2)), 'all');
     z = mean(int64(img(:,:,3)), 'all');
     r = img(:,:,1)*(Ravg/x);
     g = img(:,:,2)*(Gavg/y);
     b = img(:,:,3)*(Bavg/z);
     img = cat(3, r, g, b);
End
```

col_row_saver: this code performs all the functions of orig_seq_diff, but instead of creating new files it merely saves the cropping center information into a new array, and then assigns labels that correspond with those manually given to the auto-cropped photos. It then determines the number of each label, and creates an array that stores the last 40% of each label (this corresponds to the testing images separated in the bird_or_no_bird code).

```
global img_standard
bdf = imageDatastore('Birdfeeder', 'IncludeSubfolders', true,
'LabelSource', 'foldernames');
img_standard = readimage(bdf, 95);

bdq = imageDatastore('BIRDS', 'IncludeSubfolders', true,
'LabelSource', 'foldernames', 'ReadFcn', @bigwormread3_no_crop);
Bn = bdq.Labels;
An = sort_nat(bdq.Files);
bdq.Files = An;
bdq.Labels = Bn;

sz = size(An, 1);
```

```
sb = nnz(Bn == 'b');
snb = nnz(Bn == 'n b');
n = 1;

for i = 1:sz
     C = regexp(An(i),'\d*','Match');
     for ii= 1:length(C)
     if ~isempty(C{ii})
          Num(ii,1)=str2double(C{ii}(end));
     else
          Num(ii,1)=NaN;
     end
     end
     An(i) = num2cell(Num);
end

[firstsorted1, firstsortorder1] = sortrows(An);
bdq.Files = bdq.Files(firstsortorder1);
bdq.Labels = Bn(firstsortorder1);

arraySize = [480 640 3];
result = cell(1, sz);
for q = 1:sz
     result{q} = zeros(arraySize);
     if (q-1) > 0
     result{q} = uint8(abs(int64(readimage(bdq, q)) -
int64(readimage(bdq, (q-1)))));
     end
end

result1 = result;
col_row_array = zeros(sz, 3);
for i = 1:sz
     orig_img = readimage(bdq,i);
     img = result1{i};
     img1 = imresize(img, 0.25);
     r = img1(:,:,1);
     g = img1(:,:,2);
     b = img1(:,:,3);
     rm = max(r(:));
     gm = max(g(:));
     bm = max(b(:));
     [rr, rc] = find(r == rm);
     [gr, gc] = find(g == gm);
     [br, bc] = find(b == bm);
```

```
     row_norm = 4*(int64(rr(1))*int64(rm) +
int64(gr(1))*int64(gm) +
...int64(br(1))*int64(bm))/(int64(rm)+int64(gm)+int64(bm));
     column_norm = 4*(int64(rc(1))*int64(rm) +
int64(gc(1))*int64(gm) +
...int64(bc(1))*int64(bm))/(int64(rm)+int64(gm)+int64(bm));
     col_row_array(i,1) = row_norm;
     col_row_array(i,2) = column_norm;
end
col_row_array(:, 3) = orig_ns.Labels;
% saves coordinates of cropped images and adds labels

num_of_nobird = 0;
num_of_bird = 0;
for k = 1:765
     if col_row_array(k,3) == 2
     num_of_nobird = num_of_nobird + 1;
     else
     num_of_bird = num_of_bird + 1;
     end
end
%counts number of birds and no_birds

num_of_nobirdtrain = round(0.6*num_of_nobird);
num_of_birdtrain = round(0.6*num_of_bird);
no_bird_count = 0;
bird_count = 0;
new_col_row = zeros(round(0.4*765), 3);
test_count = 0;
for p = 1:765
     if col_row_array(p,3) == 2
     no_bird_count = no_bird_count + 1;
     if no_bird_count > num_of_nobirdtrain
          test_count = test_count + 1;
          new_col_row(test_count,:) = col_row_array(p,:);
     end
     else
     bird_count = bird_count + 1;
     if bird_count > num_of_birdtrain
          test_count = test_count + 1;
          new_col_row(test_count,:) = col_row_array(p,:);
     end
     end
end
% separates the test images from the training images
```

Error_array: this code creates a montage of the original photos with the cropping region marked for every case where the label did not line up with what AlexNet predicted.

```
num = numel(testImgs.Files);
numError = num - numCorrect;
errorArray = zeros(numError,1);
n = 1;
% counts # of errors and creates array

for i = 1:num
     if testpreds(i) ~= birdActual(i)
     errorArray(n) = i;
     n = n + 1;
     end
end
% records where errors occurred

resultArray = cell(numError, 1);
for j = 1:numError
     resultArray{j} = readimage(bdxtest, errorArray(j));
end
% stores original datastore images where an error occurred

new_resultArray = resultArray;
for l = 1:numError
     image(resultArray{l})
     hold on
     rectangle('Position', [(new_col_row(errorArray(l), 2) - 10)
(new_col_row(errorArray(l), … 1) - 10) 20 20], 'EdgeColor', 'r',
'LineWidth', 4)
     F = getframe;
     [X,map] = frame2im(F);
     new_resultArray{l} = X;
     close
end
montage(new_resultArray)
% shows where cropped image was relative to original with a red
square
```

A.2 Python TensorFlow Script

APDR_TF.py: This script performs either the training and storage of a new model, or the loading and testing of an old one. It begins by prompting the user to select either training or testing; if testing is selected, the script loads our images from the local file system and places them into the training and testing sets. These sets are saved into local files for future testing or other use. Then the model is compiled, trained, and stored in a file. If testing is selected, the saved model and testing images are loaded, and the model is evaluated using them. The script also incorporates graphing utilities which are implemented in the evaluation section. This code is executed in a bash session within the Docker Quickstart Terminal, and is stored within the public user directory of a personal machine.

```
#Justin Christian, APD&R, April 2019
#This file contains the TensorFlow Keras neural network model we
#are implementing for our research project.
#It supports functionality for both training and testing models,
#as well as storing model, image, and graphical data within the
#public user directory on the local machine.
#Credit to Francois Challet (2017) and the TensorFlow authors
#(2018) for providing initial inspiration and basic structure
#for this model.

#only the first 4 import calls are necessary for a basic
tensorflow model to function
from __future__ import absolute_import, division, print_function
import tensorflow as tf
from tensorflow import keras
import numpy as np

#Python Image Library assists with handling our own image data
from PIL import Image

#we import os to deal with accessing the file system
```

```
import os

#matplotlib is a robust graphing utility
import matplotlib

#while attempting to get matplotlib to actually work, we found a
#tip suggesting to include this following line.
#it might not be necessary anymore, but it's supposed to be
#called here
matplotlib.rcParams["backend"] = "TkAgg"
import matplotlib.pyplot as plt
import matplotlib.rcsetup as rcsetup

#all keras api aimed at reducing overfitting
from tensorflow.keras.backend import
manual_variable_initialization
from tensorflow.keras.constraints import max_norm
from tensorflow.keras.constraints import unit_norm
from tensorflow.keras.regularizers import l2
from tensorflow.keras.layers import Dropout

#this was incorporated as a suggestion when testing accuracy did
#not reflect training accuracy: likely not needed anymore
manual_variable_initialization = True

print(tf.__version__)

#used to read in files without including unexpected hidden files
def listdir_nohidden(path):
    for f in os.listdir(path):
        if not f.startswith('.'):
            yield f

#the following 4 methods were designed to test our model on
#distinct, easily categorizable images.
#we would simply call one of these methods on one half of the
#images and use another on the other half to make 2 distinct,
#easily separable groups that the model should have an easy time
#with
def to_black(Image):
  new_image = Image
  for r in range(new_image.width):
    for c in range(new_image.height):
      new_image.putpixel((r,c), 255)
  return new_image

def to_white(Image):
```

```
  new_image = Image
  for r in range(new_image.width):
    for c in range(new_image.height):
      new_image.putpixel((r,c), 0)
  return new_image

def diagonal(Image):
  new_image = Image
  for r in range(new_image.width):
    for c in range(new_image.height):
      old_pixel = new_image.getpixel((r,c))
      if (r == c):
          new_image.putpixel((r,c), 255)
      else:
          new_image.putpixel((r,c), 0)
  return new_image

def box(Image):
  new_image = Image
  for r in range(new_image.width):
    for c in range(new_image.height):
      old_pixel = new_image.getpixel((r,c))
      if (r == 0 or r == 4 or c == 0 or c == 4):
          new_image.putpixel((r,c), 255)
      else:
          new_image.putpixel((r,c), 0)
  return new_image

#this code enables us to implement an image dictionary so we can
#keep track of our images as they go through the script
#(particularly after the shuffle)
#by saving the random state as a variable, we can shuffle the
#actual images and the corresponding dictionary the same way
def shuffle_unison(a, b):
    state = np.random.get_state()
    np.random.shuffle(a)
    np.random.set_state(state)
    np.random.shuffle(b)

#this suppresses one of the several warnings that pop up when
#executing our current version of tensorflow code
#(this one didn't matter, the other probably don't either)
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'

#variable which determines functionality of the script from user
#input (training vs. testing)
```

```
activation = input("To train a new model, enter 't'. To evaluate
an existing model, enter 'e'. To exit, enter anything else. ")

#training
if (activation == 't' or activation == 'T'):

    #this next portion is all about reading in files from the
    #local file system.
    #basically, your images must be separated by classification
    #in separate folders in the Users/<name> subdirectory
    #we have a 'Bird' and 'No Bird' folder in C:/Users/Justin;
    #the script creates empty numpy arrays to fill with image
    #data as it changes directories
    os.chdir('/tmp/Bird')

    #integer of size equal to the amount of files in the current
    #directory
    bird_size = len([name for name in os.listdir('.') if
    #os.path.isfile(name)])

    #empty array of length bird_size to fill with images
    #containing birds
    bird_images = np.empty([bird_size, 10, 10], np.uint8)

    #dictionary for keeping track of images containing birds.
    #data type is U40 (unicode max length 40 characters, solves
    #strange behavior with printing)
    bird_dict = np.empty([bird_size], dtype='U40')

    os.chdir('/tmp/No Bird')

    nobird_size = len([name for name in os.listdir('.') if
    os.path.isfile(name)])

    nobird_images = np.empty([nobird_size, 10, 10], np.uint8)

    nobird_dict = np.empty([nobird_size], dtype='U40')

    os.chdir('/tmp/Bird')

    bird_iterator = 0

    print("Loading images with birds.")

    #we now make use of PIL. we are accessing image files one at
    #a time (excluding hidden files), opening them, and
    #converting #the data to a numpy array.
```

```
#we then copy the data to our empty array, as well as copy
#the filename to our dictionary
for filename_bird in listdir_nohidden('/tmp/Bird'):
    bird_image = Image.open(os.path.abspath(filename_bird))
    Image.Image.load(bird_image)
    bird_data = np.asarray(list(bird_image.getdata()))
    bird_data = np.reshape(bird_data, (10,10))
    bird_images[bird_iterator] = bird_data
    bird_dict[bird_iterator] =
    os.path.basename(filename_bird)
    bird_iterator = bird_iterator + 1

os.chdir('/tmp/No Bird')

nobird_iterator = 0

print("Loading images without birds.")

for filename_nobird in listdir_nohidden('/tmp/No Bird'):
    nobird_image =
    Image.open(os.path.abspath(filename_nobird))
    Image.Image.load(nobird_image)
    #invert(nobirdtrain_image)
    nobird_data = np.asarray(list(nobird_image.getdata()))
    nobird_data = np.reshape(nobird_data, (10,10))
    nobird_images[nobird_iterator] = nobird_data
    nobird_dict[nobird_iterator] =
    os.path.basename(filename_nobird)
    nobird_iterator = nobird_iterator + 1

#a significant note when working with our current image
#data:
#we have to randomize the images that are selected to be
#training vs. testing.
#we suffered from overfitting for a good while before
#realizing it was because we used the same images to test
#with every time
shuffle_unison(bird_images, bird_dict)

shuffle_unison(nobird_images, nobird_dict)

#we have now read in our image data and created bird and
#no_bird arrays, and have shuffled those arrays
#the next step is to divide the data into training and
#testing arrays
#here you can define the ratio of training images to testing
#images
```

```
train_ratio = 4 / 5
test_ratio = 1 - train_ratio

train_size = int(len(bird_images) * train_ratio)
test_size = int(len(bird_images) * test_ratio)

train_images = np.empty([train_size*2, 10, 10], np.uint8)
train_labels = np.empty([train_size*2], np.uint8)
train_dict = np.empty([train_size*2], dtype='U40')

test_images = np.empty([test_size*2, 10, 10], np.uint8)
test_labels = np.empty([test_size*2], np.uint8)
test_dict = np.empty([test_size*2], dtype='U40')

print("Compiling training data.")

birdtrain_iterator = 0

#we will grab the first pre-defined number of images to call
#training images,
#but the array we're grabbing from was shuffled, so the
#images will be different each time
while birdtrain_iterator < train_size:
    train_images[birdtrain_iterator] =
    bird_images[birdtrain_iterator]
    train_dict[birdtrain_iterator] =
    bird_dict[birdtrain_iterator]
    train_labels[birdtrain_iterator] = 1
    birdtrain_iterator = birdtrain_iterator + 1

nobirdtrain_iterator = 0

while nobirdtrain_iterator < train_size:
    train_images[nobirdtrain_iterator+train_size] =
    nobird_images[nobirdtrain_iterator]
    train_dict[nobirdtrain_iterator+train_size] =
    nobird_dict[nobirdtrain_iterator]
    train_labels[nobirdtrain_iterator+train_size] = 0
    nobirdtrain_iterator = nobirdtrain_iterator + 1

print("Compiling testing data.")

birdtest_iterator = 0

while birdtest_iterator < test_size:
    test_images[birdtest_iterator] = bird_images
    [birdtest_iterator+train_size]
```

```
    test_dict[birdtest_iterator] =
    bird_dict[birdtest_iterator+train_size]
    test_labels[birdtest_iterator] = 1
    birdtest_iterator = birdtest_iterator + 1

nobirdtest_iterator = 0

while nobirdtest_iterator < test_size:
    test_images[nobirdtest_iterator+test_size] =
    nobird_images[nobirdtest_iterator+train_size]
    test_dict[nobirdtest_iterator+test_size] =
    nobird_dict[nobirdtest_iterator+train_size]
    test_labels[nobirdtest_iterator+test_size] = 0
    nobirdtest_iterator = nobirdtest_iterator + 1

os.chdir('/tmp')

#the images data has to be normalized
train_images = train_images / 255.0

test_images = test_images / 255.0

#here we save the various arrays storing our data to be
loaded in the evaluation section
np.save('train_images', train_images)
np.save('train_labels', train_labels)
np.save('train_dict', train_dict)
np.save('test_images', test_images)
np.save('test_labels', test_labels)
np.save('test_dict', test_dict)

print("Images saved.")

class_names = ['No Bird', 'Bird']

print("# of training images: " + str(len(train_images)))
print("# of testing images: " + str(len(test_images)) +
"\n")

#the implementation of our neural network. it's a very
#simple model with a single hidden layer, not the standard
#for image processing
#a good move forward would be to switch to a convolutional
#network
#the flattening layer is necessary because we use a dense
#layer
```

```
    #the relu activation is standard for the hidden layer, while
    #the sigmoid activation is used because this is binary
    #classifcation
    #the choice of 50 nodes came from the fact that when
    #overfitting is a problem,
    #limiting your neurons will help (otherwise I would start at
    #100 to match the number of inputs i.e. pixels)
    model = keras.Sequential([
        keras.layers.Flatten(input_shape=(10,10)),
        keras.layers.Dense(50, activation=tf.nn.relu),
        keras.layers.Dense(1, activation='sigmoid')
    ])

    #define the algorithm for modifying weights
    #Adam or SGD seem good choices for this (see paper)
    #binary_crossentropy is necessary for binary classification
    model.compile(optimizer=keras.optimizers.Adam(),
        loss='binary_crossentropy',
        metrics=['accuracy'])

    #the only parameter to mess with here is the number of
    #epochs,
    #with the optimal apporoximate number right now being 150
    #iterations
    model.fit(train_images, train_labels, epochs=150)

    #save the trained model into the user directory
    #note that file access on the local machine is active while
    #using Docker in the intended manner:
    #the model would appear immediately in your personal file
    #system,
    #or you can modify this script and save it without needing
    #to remount the directory
    model.save('my_model.hdf5')

    print("Model trained and saved.")

#evaluation section
elif (activation == 'e' or activation == 'E'):

    model = keras.models.load_model('my_model.hdf5')

    print("Model loaded.")

    #load image data. note that they are saved as numpy files,
    #so they cannot be read if you were to open one in a text
    #editor (could be addressed if needed)
```

```
train_images = np.load('train_images.npy')
train_labels = np.load('train_labels.npy')
train_dict = np.load('train_dict.npy')
test_images = np.load('test_images.npy')
test_labels = np.load('test_labels.npy')
test_dict = np.load('test_dict.npy')

#this function is what returns the accuracy of the model on
#new testing data
model.evaluate(test_images, test_labels)

#this array contains the predicted classification of the
#test images:
#currently, 0 for an image without a bird and 1 for an image
#with a bird
predictions = model.predict_classes(test_images)

#this array contains the actual value that the output layer
#is producing,
#which is interpreted either as one class or the other.
#the threshold is 0.5: anything above is a bird, anything
#below isn't
sigmoid = model.predict(test_images)

#for analysis purposes
for i in range(20):
    print("True Label=[%s], Predicted Label=%s,
    Percentage=%s, File Name= %s" %
    (test_labels[len(test_labels)-i-1],
    predictions[len(test_labels)-i-1],
    sigmoid[len(test_labels)-i-1],
    test_dict[len(test_labels)-i-1]))

print()

for i in range(20):
    print("True Label=[%s], Predicted Label=%s,
    Percentage=%s, File Name= %s" % (test_labels[i],
    predictions[i], sigmoid[i], test_dict[i]))

#for more analysis
#weights = np.asarray(model.get_weights())

#weights_flat = weights.flatten()

#print(weights_flat)
```

```
    #print(len(weights_flat))

    #this segment produces a graph plotting the sigmoid layer
    #output.
    #for an ideal model and image data, we wold expect it to
    #match the sigmoid shape
    fig = plt.figure()
    x = np.arange(0, len(test_images), 1)
    y = np.sort(sigmoid, axis=None)
    fig, ax = plt.subplots()
    ax.plot(x, y, x, np.full((len(test_images)), 0.5))
    plt.axis([-20,len(test_images)+20,0,1])
    plt.xlabel('Image #')
    plt.ylabel('Sigmoid Output')
    plt.title('Sigmoid Outputs with Image Variance (Trial 3)')
    plt.show(block=True)
    plt.savefig('sigmoid_graph_variance3.png')

else:
    exit()
```

Bibliography

[1] https://matlabacademy.mathworks.com/R2018a/portal.html?course=deeplearning

[2] K. Suzuki, H. Abe, H. MacMahon and K. Doi, "Image-processing technique for suppressing ribs in chest radiographs by means of massive training artificial neural network (MTANN)," in *IEEE Transactions on Medical Imaging*, vol. 25, no. 4, pp. 406-416, April 2006. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1610746&isnumber=33827

[3] Michael A. Nielsen, “Neural Networks and Deep Learning,” Determination Press, 2015. http://neuralnetworksanddeeplearning.com/chap1.html

[4] https://en.wikipedia.org/wiki/Artificial_neural_network#/media/File:Colored_neural_network.svg, accessed Oct 5, 2018

[5] http://cvml.ist.ac.at/courses/DLWT_W17/material/AlexNet.pdf

[6] http://cs231n.github.io/convolutional-networks/#fc

[7] Kingma et al., 2014, https://arxiv.org/pdf/1412.6980.pdf

[8] https://openframeworks.cc/ofBook/chapters/image_processing_computer_vision.html, accessed Nov 11, 2018

[9] https://github.com/tensorflow/docs/blob/master/site/en/tutorials/keras/basic_classification.ipynb

[10] Jason Brownlee, Binary Classification Tutorial with the Keras Deep Learning Library, https://machinelearningmastery.com/binary-classification-tutorial-with-the-keras-deep-learning-library/

[11] https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html