January 2017

Training Convolutional Neural Networks Using An Automated Feedback Loop To Estimate The Population Of Avian Species

Connor Ryan Bowley

Training Convolutional Neural Networks Using an Automated Feedback Loop to Estimate the Population of Avian Species

by

Connor Ryan Bowley
Bachelor of Science, University of North Dakota, 2016

A thesis

Submitted to the Graduate Faculty

of the

University of North Dakota

in partial fulfillment of the requirements

for the degree of

Master of Science

Grand Forks, North Dakota
December
2017

This thesis, submitted by Connor Ryan Bowley in partial fulfillment of the requirements for the Degree of Master of Science from the University of North Dakota, has been read by the Faculty Advisory Committee under whom the work as been done and is hereby approved.

11/27/2017

Dr. Travis Desell

Susan N Ellis-Felege

Dr. Susan Ellis-Felege

Dr. Eunjin Kim

This thesis is being submitted by the appointed advisory committee as having met all of the requirments of the School of Graduate Studies at the University of North Dakota and is hereby approved.

Grant McGimpsey
Dean of the School of Graduate Studies

November 29, 2017

Date

PERMISSION

Title	Training Convolutional Neural Networks Using an Automated Feedback Loop to Estimate the Population of Avian Species
Department	Department of Computer Science
Degree	Master of Science

In presenting this thesis in partial fulfillment of the requirements for a graduate degree from the University of North Dakota, I agree that the library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by the professor who supervised my thesis work or, in his absence, by the Chairperson of the department or the dean of the School of Graduate Studies. It is understood that any copying or publication or other use of this thesis or part thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of North Dakota in any scholarly use which may be made of any material in my thesis.

Connor Ryan Bowley
December 2017

CONTENTS

LIST OF FIGURES

Figure Page

LIST OF TABLES

Table Page

ACKNOWLEDGEMENTS

I would like to thank my advisors for their feedback and support throughout my time on this project. I also appreciate Jennifer Booth and all the citizen scientists on Wildlife@Home who spent time reviewing and classifying images.

Funding was provided by North Dakota EPSCoR, the Hudson Bay Project, Central and Mississippi Flyways, and the UND College of Arts and Sciences. UAS data collection supported by the Hudson Bay Project. Permissions and in-kind assistance were provided by Parks Canada, Wapusk National Park Management Board, and the community of Churchill, Manitoba.

This work has been partially supported by the National Science Foundation under Grant Number 1319700. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

DEDICATION

Dedicated to my wife Kayla for all her support

ABSTRACT

Using automated processes to detect wildlife in uncontrolled outdoor imagery in the field of wildlife ecology is challenging task. This is especially true in imagery provided by an Unmanned Aerial System (UAS), where the relative size of wildlife is small and visually similar to its background. In the UAS imagery collected by the Wildlife@Home project, the data is also extremely unbalanced, with less than 1% of area in the imagery being of wildlife. To tackle these challenges, the Wildlife@Home project has employed citizen scientists and trained experts to go through collected UAS imagery and classify it. Classified data are used as inputs to convolutional neural networks (CNNs) which seek to automatically mark which areas of the imagery contain wildlife. The output of the CNN is then passed to a blob counter which returns a population estimate for the image. A feedback loop was developed to help train the CNNs to better differentiate between the wildlife and the the visually similar background and deal with the disparate amount of wildlife training images versus background training images. When using the feedback loop and citizen scientist provided data, population estimates by the CNN and blob counter are within 3.93% of the manual count by the field biologists. When expert provided data is used the estimates are within 5.24%. This is improved from 150% and 88% error in previous work which did not employ a feedback loop for the citizen science and expert data, respectively. Citizen scientist data worked better than expert data in the current work potentially because a matching algorithm was used on the citizen scientist data but not the expert data.

CHAPTER 1

INTRODUCTION

Image classification is an important problem for wildlife ecology. Many of today's ecological projects use video or imagery for monitoring and tracking species [1–7]. Learning ecological patterns becomes a problem of annotating images and classifying the wildlife they contain. Due to the ease of obtaining video and imagery and the often large geographic area that is covered, the amount of data collected can quickly become too large for ecological researchers to go through manually.

To overcome this problem, some projects [1–4] have turned to citizen scientists to create a larger workforce that can more quickly examine large amounts of data. This requires ordinary people to volunteer their time and brain power to going through sometimes monotonous video and imagery. It is also prone to human errors, such as fatigue, eye strain, or lack of domain knowledge. Also, if a project is unable to gather enough volunteers, progress will advance slowly. To deal with these problems, computer vision techniques can be used to automate the classification of the data.

One such computer vision technique that has grown popular in recent years is the Convolutional Neural Network (CNN). A CNN is a machine learning technique that trains a set of weights using a labeled training dataset. The training data is comprised of multiple classes of data that CNN is trained to differentiate between. Many CNNs have achieved great accuracy on benchmark datasets such as the MNIST handwritten digit dataset [8–12], ImageNet [13–16], and the CIFAR 10 and CIFAR 100 datasets [17]. In general, most datasets used with CNNs have fixed size images where the object of interest fills a large area in the image. The labeled training data also tends to be fairly uniform in the number of training examples for each class.

Wildlife@Home is a ecological project with over 100,000 hours of collected video, over 65,000 images from unmanned aerial systems (UAS), and over 1,800,000 images from trail cameras. One of the end goals of the project is to create an automated system that can classify the video and imagery and differentiate among different species. To obtain labeled data that can be used to train computer vision techniques and test their effectiveness, Wildlife@Home also employs citizen scientists. This involves using a webpage that the citizen scientists can visit to record their observations.

In the collected data, some species observed are visually similar to the their surrounding background. In the UAS imagery, the wildlife takes up only a tiny fraction of each image. For example, a typical lesser snow goose (*Anser caerulescens caerulescens*: hereafter referred to as "snow geese" and the focus of this work) takes up an area less than 18×18 pixels in UAS images that range from 844×755 to over 2000×3000 pixels. It is common for multiple geese to be in one image, and it is even more common that an image contains no geese at all. For these images, the information needed about them is not only if they contain snow geese, but also how many snow geese they contain. The difference in the proportion of imagery containing snow geese relative to the background is great, making Wildlife@Home's UAS dataset extremely unbalanced. These features, and the fact that the background can vary substantially in color and appearance, begin to detail some of the challenges of image classification on the dataset.

Previous work on Wildlife@Home's UAS imagery [18] sought to calculate the population of the white phase lesser snow geese that were contained in the imagery. This work used CNNs that were trained on a dataset labeled by the citizen scientists, which was also labeled separately by expert wildlife ecologists. The separate labellings allowed for the comparison of citizen scientists provided data for training CNNs compared to expert provided data. The end result of that work was a count that overestimated the population compared to a count by experts. When using the expert dataset there was an 88% overestimate, and when using a refined citizen scientist dataset (refined by matching [19], discussed in more detail in Chapter 5) there was a 150% overestimate.

This work is a continuation of that previous work. An automated feedback loop was developed and implemented, which caused a drastic decrease in error. This feedback loop allowed the CNNs to examine the source of false positives that caused the overestimated population count and learn from that information. With this change, an average error of +5.24% was achieved when using the expert provided data and an average error of -3.93% error was achieved when using the citizen scientist provided data with corner-point/intersection observation matching [19].

The rest of this paper proceeds as follows. Chapter 2 focuses on the background information needed to use convolutional neural networks. This includes how they work, what types of layers are used, how they are trained and tested, and common training aids that are often used in practice, such as regularization and batch normalization. Chapter 3 looks at some related works from multiple areas related to this project, including using citizen scientists to generate data, using CNNs for object detection, and case studies that use object detection in ecological research. Chapter 4 examines Wildlife@Home and its datasets in greater depth, such as how the data used for this project was collected and how citizen scientists and experts labeled that data. Chapter 5 describes the work done in this project, including how it builds off of previous work and how the feedback loop used was developed and the reasoning behind it. Chapter 6 details how the project was implemented, such as what data formats and CNN implementations were used. Chapter 7 discusses the particular parameters used for the CNNs and feedback loop and the results obtained with those parameters. Chapter 8 concludes and offers suggestions for future work.

CHAPTER 2

BACKGROUND

This chapter will introduce the classification problem and some of its associated terminology. After this, CNNs will be explained in detail, including the different types of layers, their formulas and derivatives, as well as some common modifications. The chapter will end with an explanation of batch normalization, which is used to enhance the CNNs used in this work.

Classification

A classification problem is a problem that requires data to be categorized into different classes based on a labeled set of training data. In machine learning, classification often corresponds to supervised learning. A *classifier* is an algorithm that does the classification. When classifying data with a positive class and a negative class, if a positive example is classified as positive, this is known as a *true positive*. If a positive example is classified as negative, this would then be known as a *false negative*. Should a negative example be classified as negative, it would be a *true negative*, and if a negative example is classified as positive, it is a *false positive*.

Convolutional Neural Networks

Convolutional Neural Networks (CNNs) have become increasingly common for image classification tasks. They work by having a set of weights and biases that are tuned to create a function that one desires, such as classifying an image as belonging to a particular class of images. Tuning these weights correctly, called training, is an important part of using CNNs.

CNNs consist of many “hidden” layers stacked on top of each other in between an input layer and output layer. The output values of one hidden layer are the input values of the next layer. Each of these layers consists of a number of nodes, called neurons, either in three or one dimensions.

A typical CNN contains four main types of hidden layers: convolutional layers, activation layers, pooling layers, and fully connected layers. Convolutional layers have some of the weights that are tuned during training. Pooling layers have no weights, but instead are used to reduce the size of layers by reducing the width and height. Activation layers typically come after convolutional layers, and run the output of the convolutional layer through some non-linear function. Often times, the activation layer is made to be a function that is added onto the end of the convolutional layers, rather than a separate layer in itself. The fully connected layers are the same as normal artificial neural networks (NNs). These have the rest of the weights to be trained and are the only layers that are 1-dimensional.

Most often, CNNs are trained through supervised learning, where there exists a set of labeled training data that it can learn from. A training example is run through the network, the network generates an predicted output, and that output is compared to the actual value corresponding to that training example. The error between the actual and the predicted output is then used to train the network toward better weights. There are many different ways that the actual weight updates can be done, but backpropagation using gradient descent is by far the most common.

To calculate the gradient needed for backpropagation, an objective function is used to define if an output is correct and/or how correct it is. One simple objective function would be to minimize error in a system. Consider that if the output from running one training example through a CNN (for the purpose of determining which class of images the example belongs to) consists of n values between 0 and 1, one for each of the n classes we are classifying. We can normalize these n values such that they sum to one (as is done by the softmax function, described in more detail below). Each value then represents the confidence the CNN has that the image is of that class. The "true" values would be a 1 for the class the image is actually of, and 0s for the others. By calculating the difference between the true value and the predicted value for each class and summing the absolute value of these differences, we an calculate an error in the system for that image. By doing this over multiple images, we can calculate a total error for all the images seen. The function describing the error we are trying to minimize (the sum of the absolute value of the differences, in this case) is the objective function. Once we have a function, we can derive a gradient for it. Often times extra terms are added to this objective function, as is the case with L2 regularization described below. These extra terms are intended to alter the gradient in order for the result of the training to have some desirable property, such as smaller weights in the case of regularization. It is also common to see objective functions utilize cross entropy loss instead of just using the raw error (some examples are [20–22]).

Feed Forward is the term most often used to describe running a image through the neural network to get a prediction. This is done in both training and inference (classifying a non-training example). Backpropagation, on the other hand, is used to propagate the error "backward" through the network to get the partial derivative at each weight and is done only during training. The partial derivatives can then be used to update the weights so they move down the gradient. Most often, the partial derivatives are calculated analytically using calculus. An approximate can be generated using a numerical approach, but it is slower and less accurate than the analytic approach.

Of great concern when training neural networks is generalization. That is, how well can the network classify data it hasn't seen before? One issue that can hurt the generalization of a network is overfitting on the training data. Overfitting is when the network can achieve very high accuracy on the training data, but the features it picks up on are more specific to a particular training image than the class of images as a whole. Much work has been done on how to increase generalization and reduce overfitting, as large CNNs can easily overtrain and "memorize" the training data. The relevant techniques used in this paper, such as L2 Regularization, weight bounding, and batch normalization, are described below.

Training with Gradient Descent

When training using backpropagation and gradient descent, there are few different terms (some people use them interchangeably) that are often used. These terms describe how often weight updates are applied by the backpropagation.

Stochastic gradient descent (SGD) is a training scheme where weights are updated every time *a single training example* is run through the network. This means that when weights are being updated, *one* example is responsible for the direction and magnitude of the update. That is, we are looking at the gradient in respect to one example at a time. The advantage of this is that by doing a lot of weight updates, updates are made quickly moving along the gradient. The disadvantage is one example might not accurately represent the overall gradient with respect to the entire training set or class of images.

Batch gradient descent (BGD) is the opposite of this. Weights are updated only after *the whole training set* is run through network. The gradients are computed (i.e. backpropagation is run) after every example and the computed gradients are summed (or averaged). The summed (or averaged) gradient is then used to perform one update on all the weights, after which the whole training set is run through again and another weight update is performed. The advantage of this is it should better represent the gradient with respect to the whole training set. While this is good, that gradient might not be representative of the testing set the network should generalize against, or of the class of images as a whole (although a *good* training set should be). A disadvantage of this scheme is that it has a greater computation cost per weight update than SGD, especially with large numbers of training images. The hope is, however, that by moving down a "more accurate" gradient, the network should take less updates to reach the optimum.

Minibatch gradient descent (MGD) is a compromise between the SGD and BGD. Instead of doing a weight update after one or all examples, a weight update is done after N examples were N is referred to as the batch size. These N examples are known as a minibatch. Similar to BGD, the gradients computed at each on these N training examples is summed (or averaged) This way, weight updates are done more frequently and the gradients used *should* be more representative of the true gradient than if only one example was used. Often times, people will use the term SGD when they really mean MGD. Certain optimizations require or suggest the use of MGD, such as batch normalization [23], which will be described later.

Convolutional Layers

Convolutional layers are the layers from which a convolutional neural network derives its name. These layers contain weights to be trained, exist in three dimensions, and go from $R^3 \rightarrow R^3$. Each 2-dimensional plane (width by height) is referred to as feature map. A convolutional layer with a depth of d would then have d feature maps, regardless of width and height values.

The weights in these layers are grouped into filters. Every filter in a particular convolutional layer has the same size, and the dimensions of the filters have the following restrictions: The depth of the filter must be equal to the depth of the input, and the width and height of the filter must be less than or equal to the width and height of the input. A very typical filter size for width and height is 3×3.

Convolutional layers work by striding the filters across the input, looking at one part of the input at a time (the size of the part of the input being looked at is the same size as the filter) and computing the dot product between the filter and the input at each position. The distance the filters are moved across the input each time is called the stride.

Each dot product between a filter and a part of the input will produce one value. As a filter is strided across multiple locations on the input, the values can be combined in a grid to create a 2-dimensional plane of outputs. The location of a number in the grid corresponds to the location on the input that was used to produce the number. The top left number in the grid will correspond to dot product between the filter and the top left portion of the input. The next number in the grid to the right will be from the next portion of the input after moving one stride length to the right. Each filter produces a plane, and the planes stacked on top of each other (so adding a plane increases depth) to make the 3-dimensional output volume. For a visual showing a CNN structure and connections between layers, see Figure 12.

The size of the 3-D output is dependent on the size of the input, the size of the filter, the number of filters, and the stride. The equations to calculate the output size are:

$$w_o = (w_i - w_f)/s + 1 \tag{1}$$

$$h_o = (h_i - h_f)/s + 1 \tag{2}$$

$$d_o = N_f \tag{3}$$

where w_o, h_o, d_o are the width, height, and depth the output respectively, w_i

and h_i are the width and height of the input, w_f and h_f are the width and height of the filters, s is the stride, and N_f is the number of filters. Note that the terms $(w_i - w_f)/s$ and $(h_i - h_f)/s$ must return integers in order for the output size to have a width and height that are whole numbers.

Zero padding is a technique commonly done to the input of convolutional layers. This involves adding rows and columns of 0s on the outside edges of the input. Doing this changes the equations for the width and height of the output to:

$$w_o = (w_i - w_f + 2p)/s + 1 \tag{4}$$

$$h_o = (h_i - h_f + 2p)/s + 1 \tag{5}$$

where p is the padding amount on each side, and the other variables are the same as above.

One advantage of zero padding is that it can alter the output size of the layer in desirable ways. For example if one had a filter size (width and height) of 3, a padding size of 1, and a stride of 1, the output would have the width and height as the input. Also note that without zero padding, the output will always be smaller than the input. Depending on the size of the input layer, the sizes of the hidden layers might decrease too quickly without padding.

The feed forward equation, a dot product, for each node is given by the following equation.

$$f_j(x, w) = w_{0j} + \sum_{i=1}^{n} w_{ij} x_i \tag{6}$$

In this equation, x is the portion of the input currently being looked at, w_{ij} is the ith weight of the jth filter, w_{0j} is the bias for the jth filter, and n is the number of weights in a filter, which is the same as the number of inputs currently being used. The bias is a learned parameter, just like the weights, and is not affected by the input.

When doing backpropagation through convolutional layers, the partial derivatives for the weights, inputs, and biases all need to be computed.

The partial derivative for the neuron, x_i is

$$\partial x_i = \sum_{k \in K} \sum_{j=1}^{N_f} w_{ij} * \partial y_{kj} \tag{7}$$

where w_{ij} is the ith weight in the jth filter, N_f is the number of filters, ∂y_{kj} is the partial derivative in the output neuron (as in the output neurons from feed forward) at depth j and location k, and K is all locations in the output from feed forward that had x_i in their feed forward computation. If a layer had a filter size of 3 and a stride of 1, the neuron just to the right of the top left neuron would be used in two computations. The indexes of the outputs of these two computations would be what is in the set K.

The partial derivative for the weight, w_l is

$$\partial w_l = \sum_{i \in I} x_i \partial y_{f(i)} \tag{8}$$

where x is the input to the feed forward, I is the set of locations of all input neurons that were multiplied with w_i in feed forward, and $\partial y_{f(i)}$ is the partial derivative of output neuron whose value had $w_l x_i$ as part of its dot product.

The partial derivative for the bias at filter j, ∂w_{0j} is

$$\partial w_{0j} = \sum_{y \in Y_j} \partial y \tag{9}$$

where Y_j is the set of neurons produced by filter j.

An important part of constructing convolutional (and fully connected) layers is proper weight initialization. If weights are not properly initialized, it can cause the CNN to train slowly or poorly. One strategy is to use small random numbers. A better strategy when using ReLU is to use initialize weights based on a Gaussian distribution with a mean of 0 and a variance of $\sqrt{2/n}$ where n is the number of inputs to the neuron [24].

Activation Layers/Functions

After the neuron values are calculated by the convolutional or fully connected layers, an activation function is typically applied to the calculated value. Common activation functions include sigmoid and *tanh*, but recently Rectifier Linear Units (ReLU) and Leaky Rectifier Linear Units (Leaky ReLU) [25] have become popular.

ReLU has the following equation:

$$ReLU(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

This is the same as saying $ReLU(x) = max(0, x)$.

Leaky ReLU is similar, using the equation:

$$Leaky\_ReLU(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{otherwise} \end{cases} \tag{11}$$

Where $0 \leq \alpha \leq 1$. A typical value for α is 0.01.

Backpropagation requires the derivatives of ReLU and Leaky ReLU, and the local derivatives are described by the following equations.

$$dReLU(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

$$dLeaky\_ReLU(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ \alpha & \text{otherwise} \end{cases} \tag{13}$$

To get the global partial derivative of the input to ReLU (or Leaky ReLU), simply multiply $dReLU(x)$ (or $dLeaky\_ReLU(x)$) by ∂y, which is the partial derivative computed on the output of the activation.

Some previous research [26, 27] also put an upper bound on the value that can come out of the ReLU function, sometimes called Clipped ReLU or ReLU clipping. In the case of clipping with Leaky ReLU, there would be a lower bound as well.

Max Pooling Layers

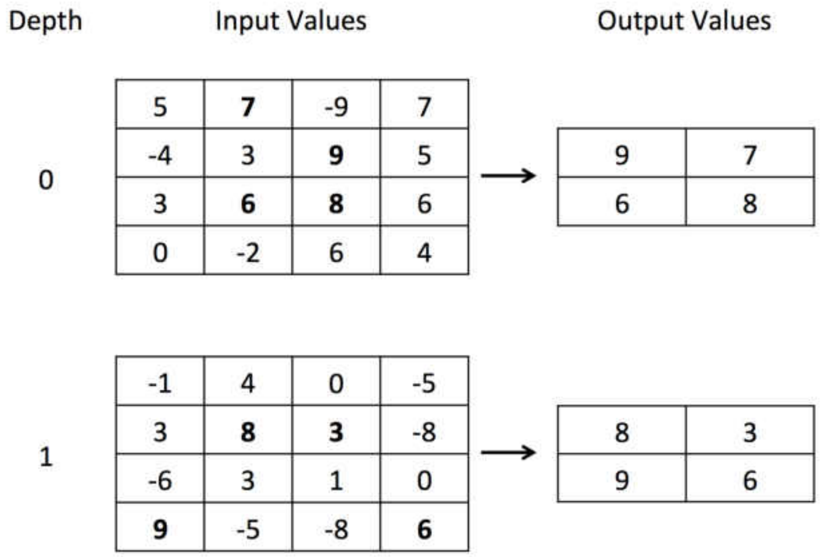
Pooling layers are generally used to reduce the size of the hidden layers of a CNN. They have a 2-dimensional pool size and a stride length, and are strided over their input similar to how convolutional layers are strided. The pool size (width×height) must be less than or equal to the width and height of the input.

The output of the pooling layers always have the same depth as the input, and the width and height can be computed by the following equations.

$$w_o = (w_i - w_p)/s + 1 \tag{14}$$

$$h_o = (h_i - h_p)/s + 1 \tag{15}$$

For these, w_o, w_i, and w_p are the widths of the output, input, and pool,



5	**7**	-9	7
-4	3	**9**	5
3	**6**	**8**	6
0	-2	6	4

9	7
6	8

-1	4	0	-5
3	**8**	**3**	-8
-6	3	1	0
9	-5	-8	**6**

8	3
9	6

Figure 1: Example of a 2×2 max pool operation. For a 2×2 max pool, the pool size is 2 and the stride is 2. The maximum values in each input pool are bold faced. Note that the maximums are computed separately at each depth.

respectively. Similarly, h_o, h_i, and h_p are the heights of the output, input, and pool, and s is the stride length.

The values of each output neuron is dependent on the type of pooling. Most commonly used is max pooling. As said above, the pool is 2-dimensional, so it is computed separately at each depth. The equation for the ith output neuron y_i for max pooling is

$$y_i = max(x \in X_p) \tag{16}$$

where X_p is the set of input neurons in the pool for output i. An example for max pooling can be found in Figure 1.

Backpropagation through max pooling layers involves only routing the derivative through the inputs that were the maximum in their respective pools. If the numbers, 1, 3, 5, and 7 were the input numbers in a pool, changing the values of 1, 3, or 5 by a little would not change the output of the max pooling operation. Because of the this, the partial derivatives for these inputs are 0. The local partial derivative for 7 would be 1, as any change done to that input would have the same change done to the output. Then the global partial derivative at the maximum input of a pool is 1 × ∂o where ∂o is the derivative at the output corresponding to the input. If the stride was smaller than the pool and an input corresponded to multiple outputs, the partials at those outputs would be summed.

Mathematically speaking the equation for the partial derivative going back through a max pool layer is

$$\partial m_i = \begin{cases} \sum_{o \in O_i} \partial o & \text{if } m_i \text{ is the maximum value for 1+ pools} \\ 0 & \text{otherwise} \end{cases} \tag{17}$$

where ∂m_i is the partial derivative at input i of the max pool layer and O_i is the set of outputs which had m_i as the maximum in their pools.

Fully Connected Layers

Fully connected layers typically comprise the last few layers of a CNN. These layers are equivalent to the hidden layers in an artificial neural network. They are called fully connected because every input of the layer has a weighted connection to every output (Figure 2). These are the only layers that tend to be 1-dimensional. As such, if the previous layer is 3-dimensional, it is flattened before becoming the input to the fully connected layer.

A fully connected layer applies the following function to calculate the value of the jth node in the layer.

$$f_j(x, w) = w_{0j} + \sum_{i=1}^{||x||} w_{ij} x_i \tag{18}$$

In this equation, x is the input vector, w is the weights vector, and w_{ij} is the weight connecting the ith input to the jth node.

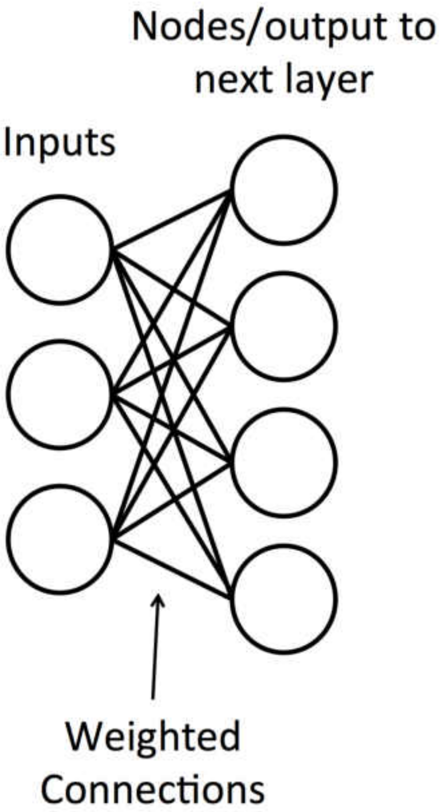


Figure 2: A fully connected layer. The left side is the inputs to the layer, which come from the previous layer. The right side is the nodes contained in this layer, which will be the inputs of the next layer. The lines between them are weighted connections.

In practice, it is easy to “fake” a fully connected layer using a convolutional layer. If one sets the number of filters to the desired output size, sets the filter size to 1 (a flat 1-D input of size N could be considered a 3-D input of size $1 \times 1 \times N$ to satisfy the needs of the convolutional layer), and does not use any padding, the convolutional layer will have a separate weight connecting every input and every output, thereby fully connecting the layer. This trick was used for this project whenever a fully connected layer was needed.

Softmax

The softmax classifier is a classifier that gives normalized probabilities for each class. Softmax is described by the following equation:

$$Softmax(x,i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{19}$$

This gets the ith output of the softmax given the input vector X whose size is n.

While this equation for softmax is fine for mathematical purposes, it can cause numerical instability when run on a computer, as the the denominator can get very large. To deal with this problem, one can subtract the maximum value in the vector from each value, causing all exponents of e to be 0 or negative, increasing numerical stability. Because of the properties of exponents and logarithms, this is does not change the value of the equation. Therefore, in practice, the equation is often

$$Softmax(x,i) = \frac{e^{x_i - max(x)}}{\sum_{j=1}^{n} e^{x_j - max(x)}} \tag{20}$$

The partial derivative for the softmax function is as follows, with j being the index of the true classification:

$$\partial Softmax(x_i) = \begin{cases} 1 - x_i & \text{if } i = j \\ 0 - x_i & \text{otherwise} \end{cases} \tag{21}$$

Note that in both cases, the constant being subtracted from is the true value of the output at that index.

L2 Regularization

Regularization is a common way to reduce overfitting in neural networks. L2 Regularization [28] works by adding

$$L2(W) = \sum_{w \in W} \frac{1}{2} \lambda w^2 \tag{22}$$

to the objective function, where W is the weights vector signifying all the weights and λ is a (usually small) constant. For backpropagation, the derivative of L2 regularization for each weight (applied separately) is

$$\partial L2(w \in W) = \lambda w \tag{23}$$

The effect of this is that every weight, w, is linearly decayed toward 0 by λw after each weight update. This penalizes large weight values and encourages the use of all weights and inputs rather than focusing mostly on inputs connected to large weights.

Weight bounding

Weight bounding [29] can also be useful in reducing overfitting by preventing large weights. Additionally, it can prevent numerical issues by not allowing outputs to become large enough that they reach positive or negative infinity in a computer. Individual weights are bounded such that $|w| < C$ for some $C > 0$, which is a hyperparameter chosen by the user.

Max norm regularization (introduced in [30], while [31] is one example of its use with CNNs) is similar to weight bounding, but instead of bounding on a per weight basis, max norm bounds the magnitude of the weight vector for each neuron. For each neuron's weight vector W, it bounds such that $||W||_2 < c$ for some $c > 0$, which, again, is a hyperparameter.

Momentum

In order to speed up training of neural networks when using backpropagation, several techniques have been developed. One commonly used technique is momentum [32]. Momentum speeds up the movement of weights along the gradient, which can allow the weights to move more quickly toward the optimum. When using momentum, two new variables are created, V representing the "velocity" vector of the gradient of the weights and μ representing the velocity decay (essentially a coefficient of friction, if one considers physics in the real world). All elements of the velocity vector typically start at 0. The velocity vector is updated by the equation

$$V = \mu V - \eta dW \tag{24}$$

where η is the learning rate, dW is the derivative of the weights, and $0 \leq \mu \leq 1$. The new weight update is then

$$W = W + V \tag{25}$$

Nesterov momentum [33] is an adaption of standard momentum. Instead of computing the gradient at the current position, we can instead look ahead to the our future position (or an approximate of it). We know that the weight vector's position is about to be updated by μV, so our future position is $W + \mu V$. Our new update is the result of the following three equations:

$$W' = W + \mu V \tag{26}$$

$$V = \mu V - \eta dW' \tag{27}$$

$$W = W + V \tag{28}$$

Another version of these equations that is a bit easier to implement is:

$$V_{prev} = V \tag{29}$$

$$V = \mu V - \eta dW \tag{30}$$

$$W = W + -\mu V_{prev} + (1 + \mu)V \tag{31}$$

Adjusting the Learning Rate

It is common to adjust the learning rate during training. At the beginning of training, a larger learning rate is typically used to move quickly down the gradient. Lowering the learning rate as training moves on, however, allows the weights to move into valleys in the gradient that it would jump over with a large learning rate.

One common way the learning rate is adjusted is using step decay [34]. Step decay adjusts the learning rate, η such that every N epochs $\eta \leftarrow \alpha\eta$, $0 < \alpha < 1$. Both α and N are hyperparameters. Some adaptive methods for adjusting learning rate have also been developed, such as Adagrad [35], RMSprop [36], and Adam [37].

Batch Normalization

Batch Normalization [23] is a technique that seeks to reduce the *internal covariate shift* in a neural network. Internal covariate shift is a term used by the authors to describe how the distribution of the input values change at each new layer. It is common to normalize the inputs to a network, and this technique seeks to normalize the inputs at each layer of the network instead of just normalizing the original inputs. Whereas the normalization on the inputs to a network are typically done over the entire training set, the normalization at the inner layers are done over each minibatch during training, hence the name.

Batch normalization can be treated as its own layer and placed between the convolutional layer and its activation function (as done in [23]).

In addition to normalizing the inputs to a layer, batch normalization also adds two new learned variables, γ and β which represent a scale and a shift value, respectively, on the data. These values are trained to optimize the distribution of the inputs, and can be set in such a way as to cause the batch normalization to represent an identity function (i.e. not do anything) if that was optimal. A separate γ and β are used for each dimension (neuron) when used with fully connected layers. A separate γ and β are used for each feature map in a convolutional layer.

Here are the equations needed to to calculate the output of the batch normalization layer during training.

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{32}$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \tag{33}$$

$$\hat{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{34}$$

$$y_i = \gamma\hat{x}_i + \beta \tag{35}$$

For these equations, m is the number of minibatch size, μ_B is the average value of input x over minibatch B, σ_B^2 is the variance over minibatch B, $\hat{x_i}$ is the normalized value of the ith item in the minibatch, y_i the output of the batch normalization layer, and ϵ is small constant to prevent division by 0.

As said above, during training, minibatch statistics are used for normalization, but minibatches are only used during training, not testing or inference. Thus a new equation is needed to compute $\hat{x}$ at run time. This equation is as follows.

$$\hat{x} = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}} \tag{36}$$

$E[x]$ and $Var[x]$ represent the population statistics over the training data for mean and variance, respectively. [23] has equations to calculate these post training, but keeping moving averages of μ_B and σ_B^2 during training can also be used (and was used for this project). The running averages can be computed by running the following equations at each computation of μ and σ^2.

$$E[x] = \alpha\mu_B + (1 - \alpha)E[x] \tag{37}$$

$$Var[x] = \alpha\sigma_B^2 + (1 - \alpha)Var[x] \tag{38}$$

For these equations, α describes how fast E and Var change and $0 \leq \alpha \leq 1$.

To do backpropagation through the batch normalization layers, the partial derivatives for each variable in the layer is needed. The derivatives are used to train γ and β just like they are used to train the weights. If ∂y_i is the partial derivative at the ith output, the partial derivatives are as follows.

$$\partial\hat{x}_i = \partial y_i \cdot \gamma \tag{39}$$

$$\partial\sigma_B^2 = \sum_{i=1}^{m} \partial\hat{x}_i(x_i - \mu_B) \cdot -\frac{1}{2}(\sigma_B^2 + \epsilon)^{-\frac{3}{2}} \tag{40}$$

$$\partial\mu_B = \sum_{i=1}^{m} \partial\hat{x}\frac{-1}{\sqrt{\sigma_B^2 + \epsilon}} \tag{41}$$

$$\partial x_i = \partial\hat{x}_i\frac{1}{\sqrt{\sigma_B^2 + \epsilon}} + \partial\sigma_B^2\frac{2(x_i - \mu_B)}{m} + \frac{\partial\mu_B}{m} \tag{42}$$

$$\partial\gamma = \sum_{i=1}^{m} \partial y_i \cdot \hat{x}_i \tag{43}$$

$$\partial\beta = \sum_{i=1}^{m} \partial y_i \tag{44}$$

CHAPTER 3

RELATED WORK

Citizen Science Projects

There are a number of projects in many disciplines that have used citizen scientists to examine data and generate results. PlanetHunters [38] is a project that had citizen scientists inspect the NASA Kepler public data release to identify potential planets. To reach citizen scientists, they used the Zooniverse tool set [39]. To help ensure consistency and accuracy, each image had 10 separate citizen scientists classify it. Instead of classifying an image directly, the users are asked specific questions about each image, and a decision tree uses their answers to make the actual classification. In the end, the citizen scientists helped identify two new planet candidates.

GalaxyZoo [40], which includes a data release [41], is a project that has citizen scientists classify galaxies in images from the Sloan Digital Sky Survey [42]. The project had more than 100,000 volunteers make over 40 million classifications. The researchers found that the overall results of the citizen scientists were consistent with results from professional astronomers. They also used a website to allow access for citizen scientists, although in order to gain access to the project, users were required to go through a tutorial and correctly identify 11 of 15 galaxies from a standard set. Some classification tasks were not asked of users (such as classifying between different classes of spiral galaxies) so as not to require people to have a high domain knowledge in order to work on the project.

While both of the previous projects use citizen scientists to aid in astronomical research, Snapshot Serengeti [1] employs the use of citizen scientists to aid ecological research by having them classify wildlife in data from camera traps in Serengeti National Park. Like PlanetHunters, Snapshot Serengeti also uses Zooniverse. There are over 50 species that are listed and the user can describe animal features (color, patterns on skin, shape of horns, etc.) to narrow down species that the image might contain.

Cornell has produced multiple projects that employed citizen scientists, such as NestWatch [2, 3] and FeederWatch [2], both of which used citizen scientists to help answer questions about avian species and their population sizes. NestWatch has citizen scientists make observations about common birds at nests usually near their homes, and it uses more experienced users and experts to find and observe more elusive and rare birds. eBird [4] uses citizen scientists not only for classifying data, but also for gathering data, as users are able to upload images of avian observations taken from mobile devices. CamClickr is another citizen scientist project that is used to create a record of nesting behavior and has been used in a university biology class to teach identification of objects to students [43].

Object Detection Techniques

Automated object detection is a popular topic in today's research. Two of the challenges for ImageNet's Large Scale Visual Recognition Challenge for the past few years is object detection from images on 200 fully labeled classes and object detection on 30 fully labeled classes from video [13]. Many techniques have been developed for these challenges and others. Among them are Region-based Convolutional Neural Network(R-CNN) [14], Fast R-CNN [15] and Faster R-CNN [16], which are region based CNNs. These use a region proposal method to identify areas of interest that can be run through the CNN to get a prediction. Fast R-CNN uses a region of interest (RoI) pooling layer to put features from some variable sized region of interest into a fixed size feature map. To increase efficiency in training, different RoIs from the same image share some computation and memory, preventing redoing of the same computations. Faster R-CNN adds a Region Proposal Network that shared convolutional features with the detection network, which greatly reduces the computation needed for generating proposal regions.

Another technique developed for object detection was YOLO [44] which was refined into YOLOv2 [45]. Instead of trying to identify regions of interest to run through the CNN, YOLO runs the whole image through a CNN. The CNN then splits the image into regions and predicts a bounding box and a probability for each region. It does this in a single pass through the CNN, unlike R-CNNs which run many sub-images of the whole image through the CNN.

Object Detection in Ecological Research

Xu and Zhu [5] worked on automatically finding and identifying seabirds with complex and uncontrolled backgrounds. They use a method called Grabcut [46] to find and segment the seabirds. After segmentation, features are extracted and run through three models (k-Nearest Neighbor [47], Logistic Boost [48, 49], and Random Forest [50]) which then voted on the final classification. When their system was run over 900 samples of 6 species of seabirds, their recognition accuracy was 88.1%.

Villa et al. [51] used the data gathered from the Snapshot Serengeti project and trained CNNs over that data. From the Snapshot Serengeti data they created four datasets, a raw unbalanced dataset, a raw balanced dataset, a balanced dataset that only includes animals that are present in the foreground of an image, and a final dataset that included segmented images that contained parts of an animal in them (meant to simulate a segmentation algorithm). Different CNN architectures were tried with each dataset. The CNNs trained on the unbalanced dataset were the worst, with the Top-1 accuracy around 58%

(the worst architecture tried for the unbalanced set had a 35% Top-1 accuracy). The best results were with the final dataset at 88.9% Top-1 accuracy.

Abd-Elrahman et al. [6] used feature-based analysis (with color and shape as the features) to detect birds in video recorded from a UAS. They manually selected the input objects needed for feature-testing. In the end, their system missed less than 20% of the objects and also had a false positive rate that was less than 20%.

Another project by Chrétien et al. [7] used UAS images of white-tailed deer that used both the visible light (RGB) spectrum and the thermal infrared (TIR) spectrum. They were unsuccessful in using supervised and unsupervised pixel-based detection methods to accurately find the deer, but they were able to use object-based image analysis (OBIA) on the RGB and TIR data to achieve 50% detection results with no false positives. This matches manned aerial surveys. However, when using only RGB imagery which contained 4 deer, OBIA detected 1,946 deer. This drastic change in results emphasizes some of the difficulty in using RGB analysis alone in certain project domains.

CHAPTER 4

WILDLIFE@HOME IMAGE DATASET

Wildlife@Home

Wildlife@Home is a citizen science project that seeks to combine crowd sourcing and volunteer computing. Users on their website, which is hosted by the Citizen Science Grid [52], can look through and classify collected data. There are three main types of data that Wildlife@Home uses.

First, there is video that comes from nest cameras. These cameras are placed near the nests of such species as Mallard ducks (*Anas platyrhynchos*), Sharp-Tailed Grouse (*Tympanuchus phasianellus*), Interior Lest Tern (*Sterna antillarum athalassos*), Blue Winged Teal (*Anas discors*), and Piping Plover (*Charadrius melodus*). Between all these species there is over 100,000 hours of video. Users are able to go through the video and annotate what is happening at what times in the video. Examples of events could be a bird on the nest, a bird off the nest and in/out of frame, or a bird preening. Previous work has run background subtraction algorithms across the video dataset [53], as well as training CNNs on the data [54].

Second, there are images that come from trail cameras. These cameras are set up to take an image every 2 minutes and 1 image per second for 30 seconds if motion is detected. As of this writing, there are over 1,800,000 trail camera images. For the trail camera images, users are able to draw bounding boxes around different species that appear in the images, and label them appropriately.

Lastly, there are images taken from UAS. Like the trail cameras, users are able to put bounding boxes around the wildlife and label them. As this is the data used for the work for this thesis, this data is described in more detail below.

In addition to annotating video and imagery, users can also volunteer their computer. The Citizen Science Grid uses the Berkeley Open Infrastructure for Network Computing (BOINC) [55], to allow users to run work for different projects on their computer. This was used to speed up running background subtractions algorithms over the video dataset [53].

Ecological Implications

The UAS images used in this work were taken in Wapusk National Park in Manitoba, Canada. According to Peterson et al. [56], an overabundance of lesser snow geese is causing the destruction of habitat in that area. In order for recovery of the habitat, Peterson et al. [56] says there must be a reduction in the lesser snow goose population. To determine the population trends, the lesser snow geese in the area are counted annually. Ground counts of nesting snow geese are typically used to base reproductive estimates, but the spatial extent that can be reasonably surveyed by a ground count is limited. By using a UAS, a larger area can be covered. However, this creates the downside of needing to go through all the created imagery and count what amounts to small dots in large images. This takes time, effort, and is prone to human error, so an automated detection process could reduce the time from flight to population estimate.

While it is possible to do manned flights and have teams in the air counting the geese directly (rather than through imagery), this is usually more expensive, less safe, and likely more disturbing to the nesting birds. Safety is an issue due to the speed and altitude one must fly at to accurately count birds from an aircraft. Using a UAS has been considered to be a cheaper and safer alternative to manned aircraft [57, 58].

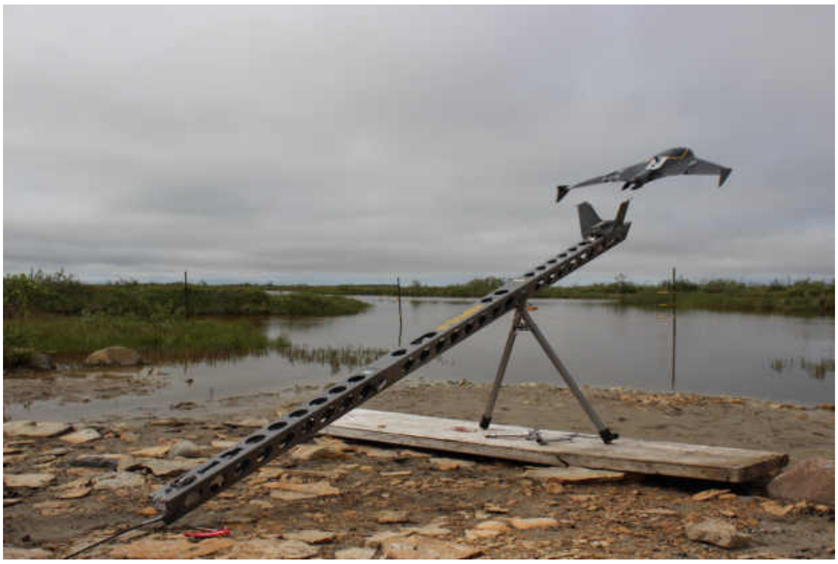


Figure 3: Image of UAS takeoff

Gathering the Data

The UAS imagery used in this project was collected using a Trimble UX5[1] fixed wing UAS (see Figure 3). The images were collected in Wapusk National Park in Manitoba, Canada in 2015 and 2016. Two survey periods were conducted each year, once during the lesser snow geese nesting season and once a month after the nesting season in the post-hatch time frame. Flights were flown at altitudes of 75m, 100m, and 120m above ground level. A 16 megapixel Sony camera placed in the nadir position recorded the images with an 80% overlap between consecutive images. Figure 4 is example of the flight path over an area. Over 65,000 images were taken in total, which reached over 3TB in size.

The images taken were then used to create mosaics for each flight. The Trimble Business Center[2] (version 3.51) was used for the 2015 data and Pix4D[3] (version 3.2.23) was used for the 2016 data. In total, 36 distinct mosaics were created that were over 50GB in size. Each mosaic was then split down into mosaic split images (MSIs) that could be shown to experts and citizen scientists

[1] http://uas.trimble.com/ux5

[2] http://www.trimble.com/Survey/trimble-business-center.aspx

[3] https://pix4d.com/

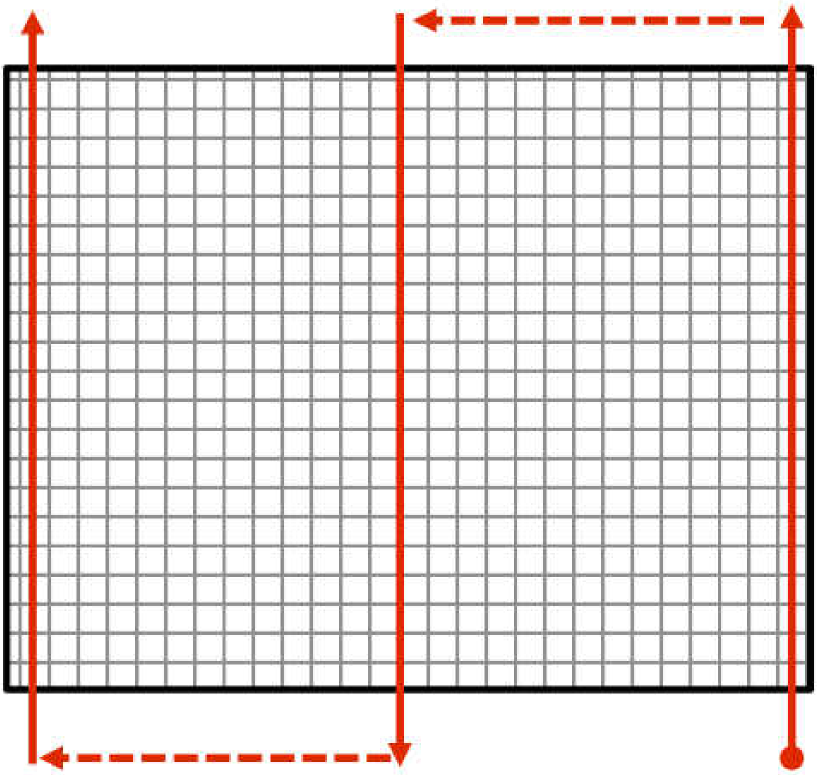
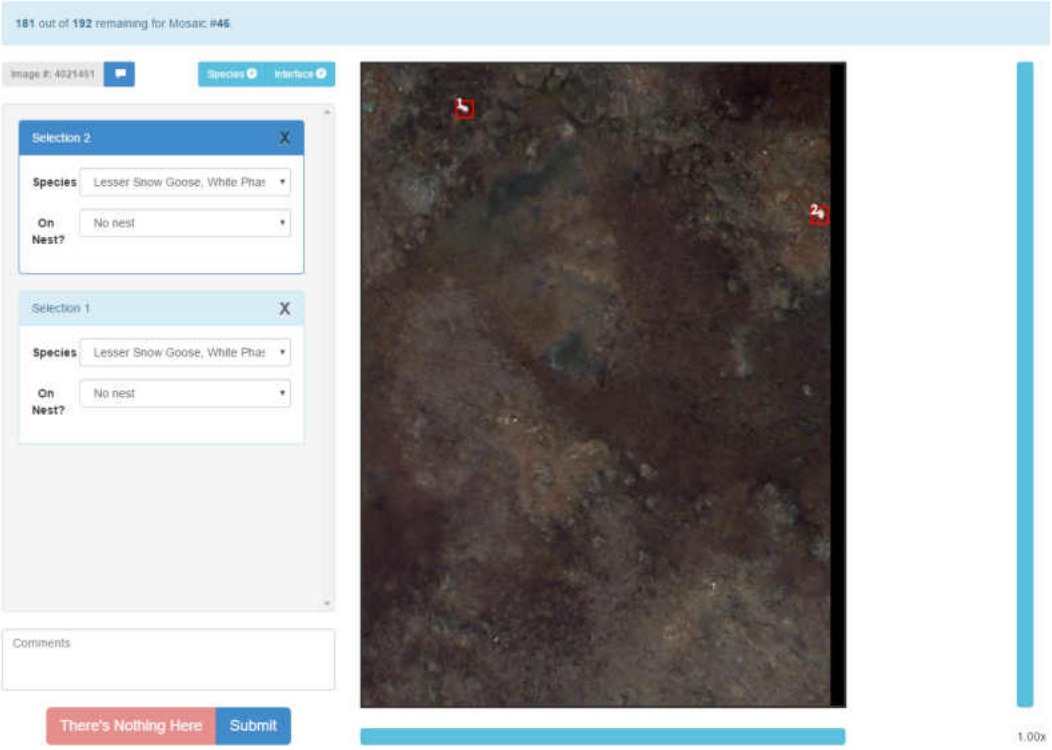


Figure 4: UAS Flight Path

through a web portal. From the 36 mosaics, 8,759 MSIs were created.

Labeling of the Data

Wildlife@Home uses a web portal (Figure 5), to allow experts and citizen scientists (collectively known as users) to go through collected imagery and make observations. Users are shown an image and instructed to draw a box around all observed wildlife. They are instructed to draw their boxes around the wildlife in such as way as to completely envelop the wildlife while minimizing the amount of negative space (background) in the box. The users then label the box according to the species and coloration they believe the wildlife to be. Documentation is available for them to compare against. Should they find no wildlife in an image, they can declare there is “nothing here”. The boxes and labels gathered by the users are recorded in a database for further usage.



181 out of **192** remaining for Mosaic **#46**.

Image #: 4021451

Selection 2

Species: Lesser Snow Goose, White Phas

On Nest?: No nest

Selection 1

Species: Lesser Snow Goose, White Phas

On Nest?: No nest

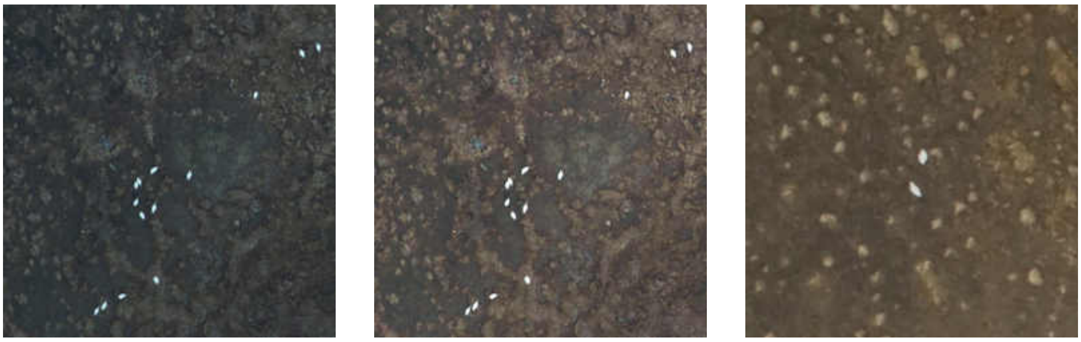
Figure 5: The graphical user interface (GUI) of the web portal for identifying objects in ecological imagery for the Wildlife@Home projects. This screenshot shows a UAS image with two white snow geese identified by the user.

The raw data generated through the web portal is given one of two designations, expert or unmatched. Unmatched observations are the raw observations from the citizen scientists and can then be matched against each other to help increase the accuracy of the data. The matching algorithm used was the 10 pixel corner point method found in [19]. This brings the total number of designations to three, described below.

Designations:

1. Expert - if the recording user is a trained expert. This data is considered to be true without fault (although in reality there are errors) and is considered the baseline by which all others (citizen scientists and CNN predictions) are judged against.

2. Unmatched - if the recording user is a citizen scientist with no training by the project leaders. Considered the least reliable data as if one untrained



(a) Original image from 2015 with blue-shift error (b) Same image from 2015 after the normalizing algorithm (c) Example image from 2016

Figure 6: An example of the blue-shift error on a 2015 UAS image with the resultant image after RGB normalization to closely match the RGB spectrum of the 2016 UAS imagery. The white snow geese are actually white and the ground is correctly brown in the normalized image.

user was wrong, data is mislabeled.

3. Matched - if two citizen scientist observations are matched, the intersection of their bounding boxes is considered a matched observation [19].

For this project, only expert and matched data were considered. The unmatched data was used only to generate the matched data. As Mattingly et al. [19] determined that matched citizen scientist data tends to be better than unmatched data, the unmatched data was not used directly. Thus, for the rest of the paper when user designations are discussed, only the expert and matched designations are being considered.

Technical Issues and Corrections

When the 2015 imagery was collected, there was a mechanical error in the RGB camera used to take the images that resulted in the images having a strong blue tint. To deal with this, the 2015 images were compared and normalized against the 2016 images. Each of the red, green, and blue channels were multiplied by 233.0/150.0, 255.0/189.0, and 236.0/190.0, respectively, and then floored and each channel was capped at 255. These numbers were chosen by sampling several images from both 2015 and 2016 data and comparing the RGB values of white phase snow geese in both datasets. The numerators describe the average integer value of the white phase snow geese color in the 2016 data and the denominators describe the average integer value of them in the 2015 data. Manual inspection of the normalized data appeared to be correct (Figure 6).

CHAPTER 5

METHODOLOGY

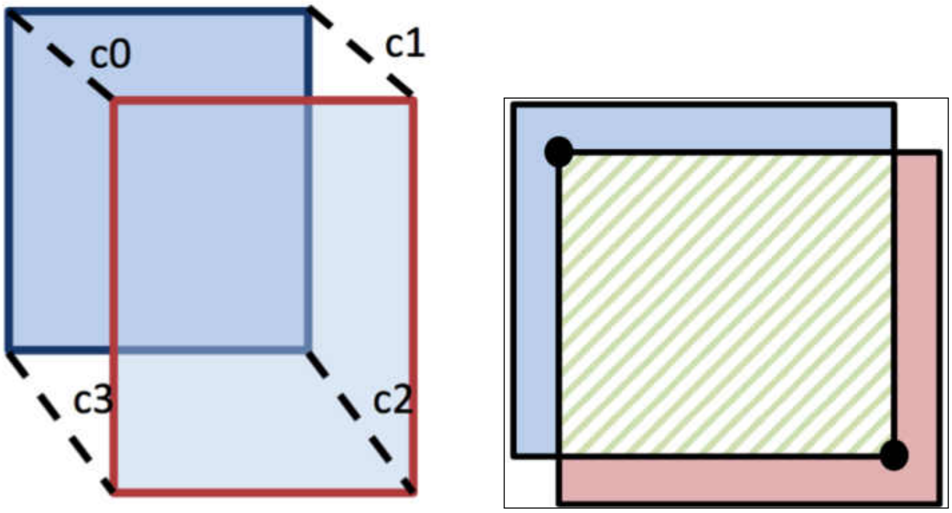
The purpose of this work is to not only identify lesser snow geese in UAS imagery, but also to count them. In this respect it is different than some applications of CNNs which only seek to classify images. Also, in contrast to many benchmark datasets used with CNNs, the objects of interest are relatively small compared to the whole image. These goals and characteristics of the data, as well as the fact that it is an unbalanced dataset, influenced the decisions made below in regard to how to best train CNNs on this data.

Matching User Observations

Before any training can be done with the CNNs, accurate training data is needed. Previously, matched user designation was defined and required that at least two separate users have a matched observation. The matching algorithm used was developed and used by Mattingly et al. [19] to compare citizen scientist observations to expert observations. The matching algorithm they determined worked best was a 10 pixel corner-point distance algorithm for *matching* observations and an intersection method for *extracting* the matched observation.

The N pixel corner point distance algorithm is as follows: if the euclidean distance between the top left corners of two observations is less than or equal to N pixels, the top left corners match. If all four corners between two observations match, the two observations are considered matched (Figure 7a).

The intersection extraction method defines the matched observation to have the corner points of the box describing the intersection of the two user observations. Because of this, the bounding box for the resultant matched



(a) Corner-point distance algorithm (b) Intersection extraction algorithm

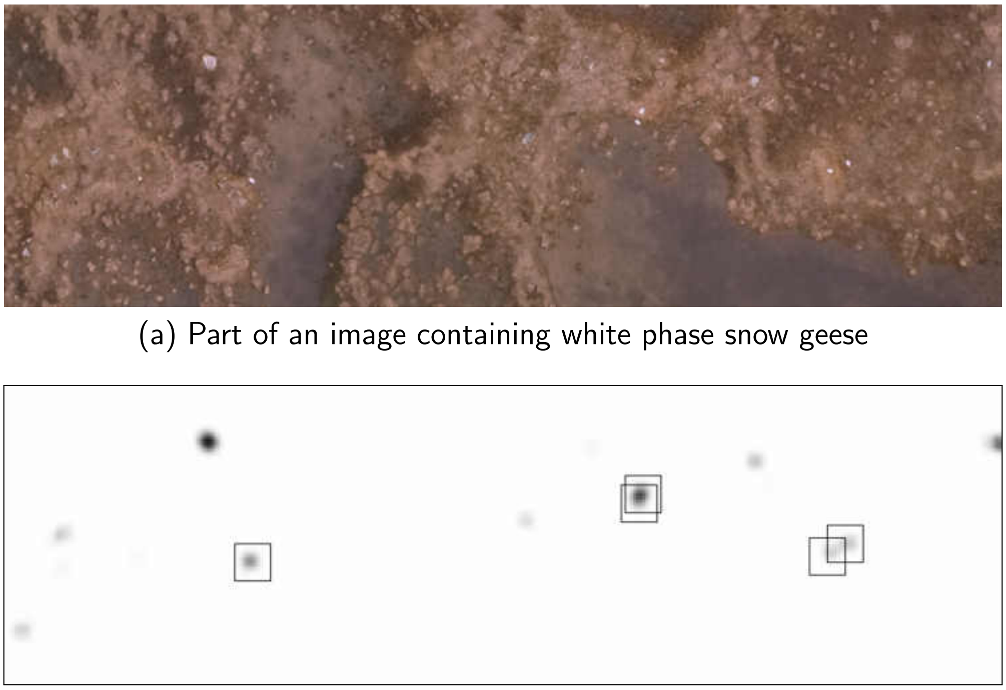
Figure 7: Visual representation of algorithms used for matching two observations

observation will never be larger than either of the users observations and should be a tighter fit around the object of interest (Figure 7b).

Analysis of Previous Work on Wildlife@Home

Previous work on the Wildlife@Home dataset in [18] has promising, albeit not ideal, results. The results of that work had the CNNs trained producing a fair number of false positives, ending with an 88% overestimation of the total population in the best case when training data generated by experts was used. This led to the questions of what was being misclassified as snow geese and why. As seen in Figure 8, there appears to be certain areas of background, such as some of the rocks, that have similar features to white phase snow geese.

In general, CNNs tend to be good at learning and discerning differences between similar images. Consider the MNIST dataset [8] and that CNNs have been trained on it with very high accuracy (e.g. [10][59] amongst many others). This is despite the fact some of the images between classes can look very similar, such as a handwritten 5 looking like a 6 or a 4 looking similar to a 9. So why does the CNN in [18] not always do a good job at discerning between rocks and



(a) Part of an image containing white phase snow geese

(b) A CNN prediction over the image

Figure 8: An example of an image and CNN prediction from previous work [18]. Note that it correctly identifies the white phase snow geese, but it also mis-classifies background that have similar features to the snow geese. The boxes in the prediction were manually placed and show the actual locations of the snow geese.

white phase snow geese? One possible reason is that many of the networks that hold or have held near world record accuracies on benchmark datasets are often very deep and large (e.g. VGGNet with its 140 million weights [60]) compared to the network used in [18].

Another possible reason is it has to do with the nature of the data. The MNIST dataset has roughly the same amount of data for each class. In the Wildlife@Home dataset, the per pixel ratio of foreground to background is incredibly small with over 99% of pixels being background. The unbalanced dataset problem is well defined with many solutions such as undersampling the majority class (used in [18]), oversampling the minority class, and SMOTE [61]. However, it is also important to note that the per pixel percentage of background with similar features to the snow geese is quite small compared to the rest of the background. This can be seen in image shown in Figure 8 (and it holds true in other images in the Wildlife@Home dataset). This brings up another difference in Wildlife@Home data compared to some other datasets. While the images in the MNIST dataset have a fairly similar look between images of the same class (not perfect similarity by any means, but fairly similar), if one wants to classify snow geese versus background, the class of "background" can vary substantially in color and features. As it happens, a small portion of this background class looks more like a snow goose (a different class) than it looks like the rest of background (the same class).

The small subset of background data, thus, is of primary interest. Let us consider defining two subclasses of the background class, the "hard background" consisting of background similar to the foreground, and the "easy background" which consists of everything else. For now, let us define "background similar to the foreground" as "background data that might be marked as a false positive by an arbitrary, trained CNN", and leave the definition somewhat open-ended. Now, if the majority class is undersampled (to deal with the imbalance) and images are taken from the whole background class randomly, the ratio of hard to easy background that the CNN would train on would be small and few hard background images would be used.

So, in a sense, the Wildlife@Home dataset has an unbalanced dataset inside another unbalanced dataset. Background is a strong majority over foreground, and easy background is a strong majority over hard background. One solution, and the one explored in this work, would be to present more hard background images to the CNN, i.e. potentially undersample the easy background and/or oversample the hard background.

One way to do this would be to split the background into two separately labeled classes, hard and easy, and have the CNN consider them separately. The largest inhibitor to this method, however, is labeling of the hard and easy background. Currently, the Wildlife@Home dataset is labeled by the drawing of bounding boxes around wildlife by experts and citizen scientists. It seems infeasible to ask users to also label the open ended “background similar to snow geese” using bounding boxes or any other method. Even if it was a feasible request, what if their definition of “similar” is too strict for the CNN, or too loose?. It could be some of the labeled hard background is actually easy for the CNN to classify, and vice versa. It also leads to the question, should the CNN be penalized in terms of error for misclassifying hard data as easy (or vice versa)? Should the error only consider background vs foreground even though its training on hard background vs easy background vs foreground (how would one even train like this)? These questions and infeasibilities suggest a need for another method.

Another (similar) method could be ensuring that hard background is included in the background shown to the CNN at a higher ratio than found in the dataset (essentially oversampling the minority sub-class, or undersampling the majority sub-class). This runs into the same problem of trying to identify hard and easy background as the previous method, although it alleviates some of the questions of how to train it. While manual labeling of hard and easy background seems infeasible, what about an automated solution? Consider in the next section, an automated feedback loop.

Feedback Loop

Let us consider the definition of “similar data” developed in the previous section, which led to the definitions of hard and easy background. Also, let us change definition slightly from an *arbitrary* CNN to a *particular* CNN. Thus, the definition of “background similar to the foreground” now reads “background data that might be marked as a false positive by a *particular*, trained CNN”. This implies that for different CNNs trained on the same dataset, what is hard background and what is easy background might be different, although it seems likely that they will contain similarities. Note that, through this change, hard background and easy background take on firmer definitions. By running the trained CNN over examples from the dataset, one can look at the false positives and define those areas as hard and the rest as easy.

This idea is the basis of the feedback loop. By the definition above, a CNN must already be trained in order to determine hard and easy background. In the feedback loop, a CNN is given feedback by identifying hard background and retraining the CNN over the same overall dataset with care taken to sample more hard background during this retraining. Ideally, after the retraining, the CNN should do better on the data that was "hard" for it before (i.e. less false positives). Also, after the first retraining, what is "hard" for the CNN will most likely change. Some of the data that was hard should no longer be and some data that was not hard before might be. Going through multiple iterations of retraining should help the CNN get better at correctly classifying the hard data and ideally do better overall.

To retrain a CNN at iteration t of the feedback loop, the starting weights will be the weights from the CNN at iteration $t-1$. For the starting weights at iteration 0, which will be the iteration number representing the initial training, they can be initialized however one would choose to typically initialize weights (random, pre-trained on a similar dataset, etc).

One major downside to this algorithm, though, is that for N iterations, it needs to run over the data N times. This factor of N added to the complexity greatly increases the computational cost of training compared to not using the feedback loop. While the linear increase in complexity and computational cost is not ideal, the feedback loop does prevent the need to present all possible background images while training the CNN, which could potentially have even larger computational costs in some extremely unbalanced datasets based on factors such as amount of data in the majority class and the number of epochs trained for.

While it may not be possible to reduce the complexity of the feedback loop

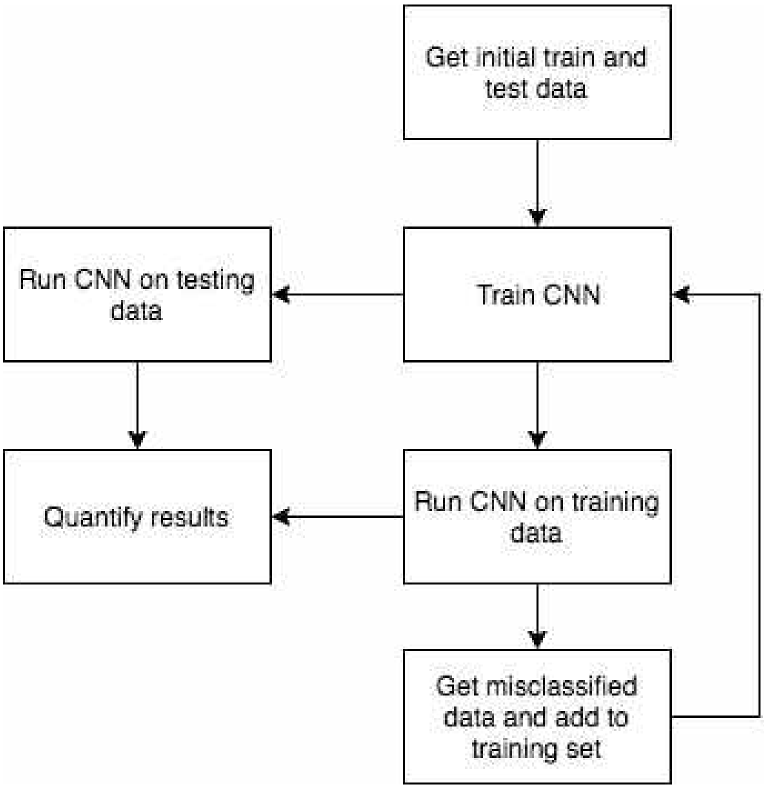

Get initial train and test data
Train CNN
Run CNN on testing data
Quantify results
Run CNN on training data
Get misclassified data and add to training set

Figure 9: Basic flowchart for feedback loop.

from linear, it is possible to reduce its computational cost. Consider an ideal world where retraining the CNN had the possibility of making it better at classifying the training examples it was getting wrong, but the retraining could not cause the CNN to do worse on the examples it was correctly classifying. In such an ideal world, once a CNN at iteration t correctly classified a training example, a CNN at iteration $t + n$, $n > 0$ would not need to look at that particular example. If, at each iteration, all training examples that were correctly identified were removed, the number of examples to train on would go down in size through continued iterations, decreasing overall computational cost.

So then, the following assumption is made. If the network correctly predicted an image at iteration t of the feedback loop, it will *probably* predict that same image correctly at iteration $t + 1$. There is, of course, no guarantee of this. So to decrease computational cost, the following change can be made to the feedback loop. If the CNN at iteration t incorrectly classifies a training example, then the trained CNN at iteration $t + 1$ will be forced to run over that example to see if the new training fixed it. However, if the CNN at iteration t correctly classified the training example or did not run over that example, then the CNN at iteration $t+1$ has some probability, p, of running over that example. This is for the purpose of making sure the retraining did not cause a previously correct classification to become incorrect. While this does not completely negate the increased computational cost of the feedback loop, it does decrease it. Also, for the initial training before the feedback loop is employed, all examples are to be used.

Sampling Amounts

When dealing with an unbalanced dataset and the decision is made to undersample a class, a natural question that arises is how much should the class be sampled? In many cases, a 1:1 ratio is chosen so each class is presented the same number of times. When dealing with an extremely unbalanced dataset like Wildlife@Home, *should* each class be presented the same number of times, or should the extreme majority class be presented more often? If one class is presented much more often while training a CNN than another class, the resulting CNN will usually have a bias toward the class presented more. Is this bias acceptable or even desirable in a situation where one class has a strong majority? And how exactly does this bias in training affect the overall results?

To attempt to answer some of these questions and improve results, this project will use different sampling rates and compare them. When a class is undersampled during training of the CNN, the amount of images of that class used each epoch will be fixed, and each epoch the images used will be taken from the given examples at random. This should help increase the diversity of images shown to the CNN.

Counting objects

The process of training and running the CNNs in such a way that the detected objects can be counted is the same as in [18, 54]. CNNs will be trained on fixed size images which have relatively small dimensions. The fixed size images will be comprised of sub-images of larger images (the MSIs). Experts and citizen scientists have placed bounding boxes around snow geese in the imagery, and these bounding boxes are used to label the sub-images with their appropriate class.

Once a CNN is trained (or retrained) on these fixed size images, it will be run over full size images. To run the CNN over the full size images, the CNN is first run over its sub-image of appropriate size in the top left-hand corner of the image, then it is strided across the image, generating predictions on the sub-images as it goes. Another way to think of this would be to consider a sliding window that is the input size of the CNN. It starts at the top left of the image and slides to right until it hits the right edge, then it moves down and starts again from the left side. It continues to move right and down until it has seen the whole image. The distance the CNN moves over the input each time is called the stride, and, after each movement, the image that can be seen through the window is run through the CNN. See Figure 10 for an example.

The outputs from each of the sub-images are then reconstructed back into a prediction for the whole image. When a sub-image is run through the CNN, a number between 0 and 1 is returned for each class signifying the confidence the CNN has that the sub-image is of that class (as described in Chapter 2). Each pixel in the prediction image also has a vector of confidences that the particular pixel is of a given class. The formula for calculating this vector is as follows:

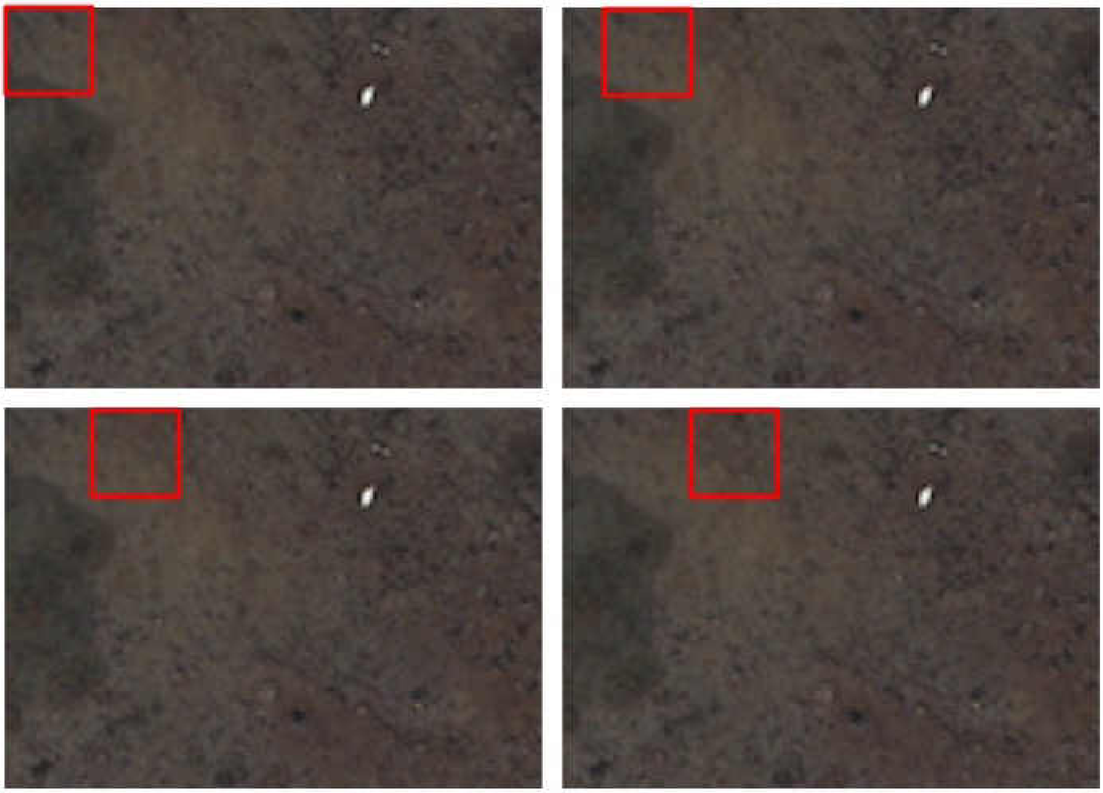


Figure 10: Example of striding a CNN across an image. The red box denotes which part of the image is being run through the CNN. When the CNN reaches the right edge, it will move down and start again at the left edge. The amount the CNN moves over each time is called the stride. This can also be thought of as a sliding window.

$$C_0(p_j) = \sum_{s \in S(p_j)} CNN(s) \tag{45}$$

where p_j is the jth pixel in the image, $C_0(p_j)$ is a function returning an intermediate vector of confidences that pixel j is of each class, $S(p_j)$ is the set of all sub-images containing pixel j, and $CNN(s)$ returns the output vector from running the CNN on sub-image s. As the sums may total to greater than one for a particular class, they can be normalized. The normalization used for this in this work is the square of the value over the sum of squares for all values in the particular vector. The equation for the confidences for each class, c in the set of all classes C, for pixel j is:

$$C(p_{jc}) = \frac{p_{jc}^2}{\sum_{i \in C} p_{ji}^2} \tag{46}$$

Each class is assigned a color, and by finding and counting blobs of the color assigned to the snow geese, a predicted population count can be obtained.

CHAPTER 6

IMPLEMENTATION

Data

Data Formats

Multiple data formats were used to represent the data and its meta-data. The main formats used were IDX, PNG, a custom binary format, and CSV.

IDX files are a type of binary file used in the MNIST dataset [8]. They are able to store multi-dimensional matrices. Their format is as such: the first 4 bytes are what is known as the "magic" number. The first 2 bytes of the magic number are always 0, the 3rd byte encodes the data type, and the fourth byte is the number of dimensions. Following the magic number is the size of each dimension stored as Big Endian Integers (4 bytes each). Following the dimensions is the data. For datasets generated and used in this project, the data is always stored in Little Endian Order, while the dimensions are in Big Endian Order. This allows compatibility with the MNIST dataset while trying to be consistent with the fact that most processors (including the ones used for this project) are Little Endian.

In this project, IDX files are used to store data of a fixed size that is to be fed to the CNN as training data. For a dataset stored as IDX files, two files are used, a data file that stores the image data, and a label file that stores what class each image belongs to. They are stored in a similar way to parallel arrays, where the ith element in the data file matches up to the ith element in the label file. The data file is encoded as unsigned bytes (represented by a 0x08 as the third byte of the magic number) and is of 4 dimensions (a vector of 3 dimensional images). The label file is encoded as signed integers (represented by a 0x0C in the magic number) and has one dimension. How the IDX files were generated is described in the preprocessing steps below. Separate pairs of IDXs were created for each user designation.

PNG files were used to store both the MSIs and CNN predictions on those MSIs.

A custom binary format was used to store user observations. These files will be referred to as "location files". The location files store the number of observations, as well as the location, species, and a hash of the user id of each observation for each MSI. All numbers are 32 bit integers stored in Little Endian Order except for the user id hash which is 128 bit. The format of the files is as follows: the first integer is the MSI number, followed by the number of observations for that MSI. Then for each observation is the species, x location in pixels, y location in pixels, width in pixels, height in pixels of the observation, and then the user id hash, in that order. This is the format for each MSI and is repeated for each MSI in the file.

A separate locations file was created for each designation of users. For images that had multiple users within the same designation look at the image, the observations from each are unioned. That is, if user A had 3 observations and user B had 3 observations, there would be 6 observations in the locations file, even if they were of the same object[1]. These locations files are used in the feedback loop to find misclassified background areas (described below).

CSV files were used to store the count of white phase snow geese for each MSI. The format of each line was "MSI number, white count, blue count" (blue count was not used for this project) and each line had a different MSI number. All of the CNNs were compared against the same count file for consistency. The

[1] Even if two observations are of the same object, it is unlikely that the bounding boxes drawn by the users are in exactly the same location, so the observations are still in some sense distinct

count file used was from the experts only, as this is considered the “most true” count. In the event that a particular MSI had multiple experts look at it who disagreed on the count, an average was taken that was then rounded to the nearest whole number.

Partitioning the Data

One goal of this project was to compare CNNs trained by expert data to CNNs trained by citizen scientist data. Thus, in order to directly compare expert users and matched users against each other, only MSIs that had both expert observations and matched observations (i.e. the intersection of the expert and matched data) was used. This data was then further split into a training set and a testing set. Approximately 20% of the MSIs were reserved for testing and the rest was used for training. Because there are considerably more MSIs that have no observed wildlife in them than MSIs that do (2803 compared to 1351), the 20% for the test set was created by combining 20% of the MSIs with observations in them (262 MSIs) and 20% of the MSIs that did not have observations in them (558 MSIs). The total dataset had 3334 training MSIs and 820 test MSIs.

Preprocessing Steps

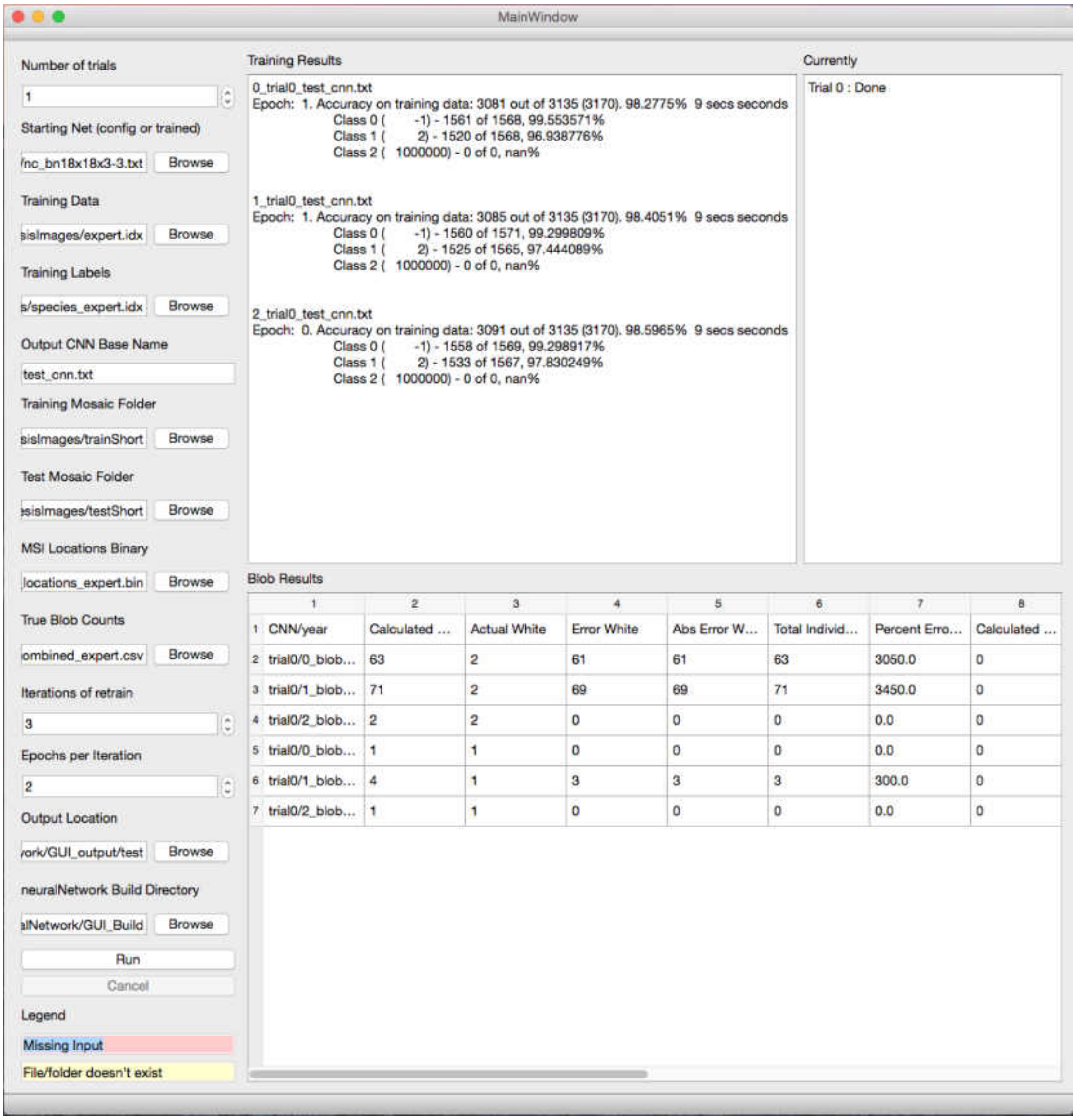
Three preprocessing steps were applied to the MSI data. The first involved choosing which MSIs would be used, and how those used were split into training and testing sets, as described above. The second preprocessing step was the normalization of the blue-shifted 2015 data, described in Chapter 4. The third was the conversion from the MSIs as PNGs to the IDX files. Only MSIs from the training set were used to create the initial training IDXs and the retraining IDXs. The testing MSIs were never made into IDXs.

The observations from the users are contained in bounding boxes of various sizes, and the MSIs themselves are not of a consistent size. However, CNNs take in fixed size input for training and running. Also, a label for each piece of training data is needed for supervised learning. This disparity is what caused the conversion from PNGs and locations to the IDXs. A fixed image size was chosen for the IDX data, which is the same as the input size of the CNN. The foreground images (images of snow geese) were obtained separately for each user designation, while the background images were shared amongst the different designations. For each designation, then, the initial training IDXs were created by combining the unique foreground set with the shared background set.

To get the foreground images, all observations from the location data were extracted and sized to be the given input size. For observations of a size different than the input size, the center of the observation became the center of a new bounding box of the input size, which was then extracted and added to the IDX file[2]. For the expert designation there were 2054 foreground observations, while for the matched designation there were 6560. The reason there is so much difference between the two classes is that more citizen scientists looked at the data than experts. This matters because increasing the number of citizen scientists looking at an MSI can cause an increase in 2-way matched observations that is greater than linear. For instance if 4 citizen scientists looked at an MSI and marked the same bird, 6 two-way matched observations would be created ($4C2$, in this case, $nC2$ in general). On the other hand, if 4 experts marked the same bird, only 4 observations would be made, as the experts are not matched.

There were 8 input sized background images grabbed from each training MSI for a total of 26,672 background examples. The locations within the MSIs were chosen at random while taking care to ensure that they did not overlap with an observation from *any* user designation.

[2] Care was taken to ensure the new box did not run off any of the edges of the image. In this case, the new box was shifted the appropriate direction to ensure that it was entirely on the image.



Number of trials: 1

Starting Net (config or trained): /nc_bn18x18x3-3.txt

Training Data: sisImages/expert.idx

Training Labels: s/species_expert.idx

Output CNN Base Name: test_cnn.txt

Training Mosaic Folder: sisImages/trainShort

Test Mosaic Folder: esisImages/testShort

MSI Locations Binary: locations_expert.bin

True Blob Counts: ombined_expert.csv

Iterations of retrain: 3

Epochs per Iteration: 2

Output Location: /ork/GUI_output/test

neuralNetwork Build Directory: alNetwork/GUI_Build

Legend

Missing Input

File/folder doesn't exist

Training Results

```
0_trial0_test_cnn.txt
Epoch:  1. Accuracy on training data: 3081 out of 3135 (3170). 98.2775%  9 secs seconds
                Class 0 (       -1) - 1561 of 1568, 99.553571%
                Class 1 (        2) - 1520 of 1568, 96.938776%
                Class 2 (   1000000) - 0 of 0, nan%


1_trial0_test_cnn.txt
Epoch:  1. Accuracy on training data: 3085 out of 3135 (3170). 98.4051%  9 secs seconds
                Class 0 (       -1) - 1560 of 1571, 99.299809%
                Class 1 (        2) - 1525 of 1565, 97.444089%
                Class 2 (   1000000) - 0 of 0, nan%


2_trial0_test_cnn.txt
Epoch:  0. Accuracy on training data: 3091 out of 3135 (3170). 98.5965%  9 secs seconds
                Class 0 (       -1) - 1558 of 1569, 99.298917%
                Class 1 (        2) - 1533 of 1567, 97.830249%
                Class 2 (   1000000) - 0 of 0, nan%
```

Currently

Trial 0 : Done

Blob Results

	1	2	3	4	5	6	7	8
1	CNN/year	Calculated ...	Actual White	Error White	Abs Error W...	Total Individ...	Percent Erro...	Calculated ...
2	trial0/0_blob...	63	2	61	61	63	3050.0	0
3	trial0/1_blob...	71	2	69	69	71	3450.0	0
4	trial0/2_blob...	2	2	0	0	0	0.0	0
5	trial0/0_blob...	1	1	0	0	0	0.0	0
6	trial0/1_blob...	4	1	3	3	3	300.0	0
7	trial0/2_blob...	1	1	0	0	0	0.0	0

Figure 11: Screenshot of training interface.

Convolutional Neural Network

The CNN was implemented using C++ and OpenCL. Each type of layer (including batch normalization) had their feed forward and backpropagation functions computed using OpenCL, while the C++ code preprocessed the data and made the appropriate OpenCL calls. The OpenCV library was used for reading and writing images. Separate command line programs were developed for training and testing the CNNs. All code is available at https://github.com/Connor-Bowley/neuralNetwork.

Feedback Loop

The feedback loop was implemented using C++ and Qt. It comprised of a simple interface to get the needed inputs (see Figure 11) and called the C++ programs for training and running the CNNs. As it ran, it showed preliminary results, including the best epoch of each training or retraining, and the blob counts for CNNs at each completed iteration.

The purpose of the feedback loop is to give the CNN feedback on what it is doing wrong by identifying misclassified training data and feeding it back into the CNN. Because the CNNs are trained on IDX files and tested against PNG images, the feedback loop needed to search through the PNGs for areas that were misclassified and convert those areas into IDX files.

Once a trained (or retrained) CNN had generated its prediction image over a training MSI, that prediction was run through another program which strided across it (just like the CNN strided over the training MSI) looking for sub-images that were misclassified. A sub-image was deemed misclassified if it was a false positive. Areas close to a bounding box were exempt from this process because the area predicted to be a snow goose by the CNN was often a little larger than the goose itself (as the CNN predicts the *whole* sub-image to be of a goose if a goose is contained within it). The definition of "close" was set to be: any sub-image with a pixel contained in a box that extends from a user supplied bounding box by N pixels in each direction is exempt from being marked as misclassified. In this work, N was set to be the same size as the CNN input size. All misclassified sub-images were then appended onto the previous iteration's training IDXs.

Reading closely, one will note that false negatives are not included in this implementation of the feedback loop. Early trials (over expert data) did include this, but results improved somewhat when it was taken out. The reason for this is that the expert data was not perfect. Even at the first iteration of retraining, most of the images the feedback loop found to be false negatives were *actually* true negatives. Thus, by the feedback loop returning examples that were background but labeled geese, the CNN was being given mislabeled training data.

CNN Architecture and Settings

The size of the fixed size training sub-images in the IDX files was decided to be 18×18 pixels. This size was chosen because most of the bounding boxes around the snow geese were within this size. Given the 18×18 input, the CNN architecture was created. The architecture can be seen in Figure 12 and Table 1. This architecture is the same as used in [18]. After each convolutional layer, a batch normalization layer and an activation layer was placed, in that order. For batch normalization, all γs were initialized at 1 and βs were initialized at 0. The activation function used was Leaky ReLU with an α of 0.01. The output of the Leaky ReLU was bounded to [-5000.0,5000.0].

Weights for the neurons in the convolutional and fully connected layers were initialized using a normal distribution with mean of 0 and a standard deviation of $\sqrt{2/n}$ where n is the number of inputs to the neuron. After each weight update, the value was bounded such that $|w| \leq 50.0$ for each weight w. The bound here and for Leaky ReLU were to prevent outputs from reaching NaN or positive or negative infinity.

Prior to training or prediction, all data was normalized. When training, the normalization done was to subtract each pixel by the mean and divide by the standard deviation with respect to all pixels from all training images. The mean and standard deviation calculated during training was then used for preprocessing at run time, rather than using the mean and standard deviation of

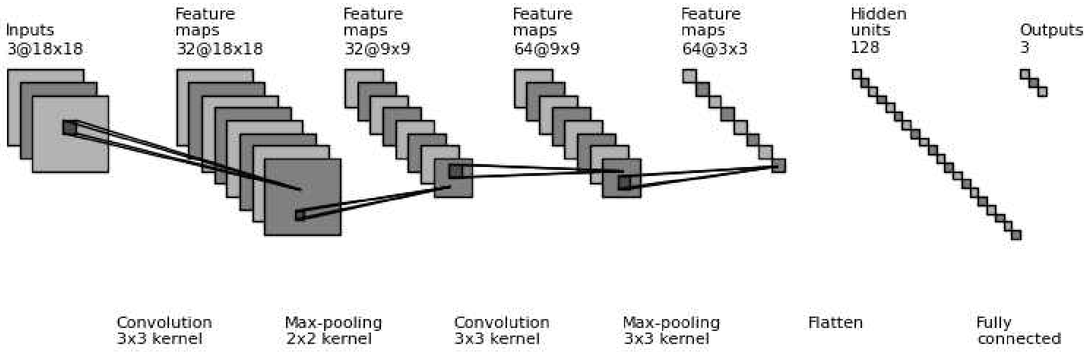


Figure 12: Architecture of the CNNs used in this work

Table 1: Architecture of the CNNs used in this work

Layer Type	**Layer Dims**	**Filter / Pool Size**	**Stride**	**Filters**	**Padding**
Input	18 x 18 x 3				
Convolutional	18 x 18 x 32	3	1	32	1
Max Pooling	9 x 9 x 32	2	2		
Convolutional	9 x 9 x 64	3	1	64	1
Max Pooling	3 x 3 x 64	3	3		
Fully Connected	1 x 1 x 128			128	
Fully Connected	1 x 1 x 3			3	

the test set. For instances of retraining, the mean and standard deviation was from all images ever trained on, including images from previous iterations.

Training

Minibatch gradient descent was used, with minibatch size of 64. The learning rate started at 1×10^{-3} and was multiplied by 0.75 each epoch. L2 Regularization was used with a λ of 0.05. Training was done for 30 epochs, and the epoch whose weights had the best accuracy on the training data was chosen as the final output. Nesterov Momentum was used with a momentum constant of 0.9.

Feedback Loop

For the feedback loop, each dataset and sampling rate pair had 3 separate trials run. Each trial had 5 iterations, consisting of 1 base training and 4 retraining iterations. Each retraining iteration had its initial weights (as well as γs and βs for batch normalization) set to the output of the previous iteration's training. Other than that, the parameters, such as number of epochs, were the same.

Prediction

For predictions over the training and test MSIs, the stride used for striding the CNN across the MSIs was 9 pixels in each direction.

Sampling Rates

To test the effects of sampling rates with this unbalanced dataset, four different ratios of background to foreground were used, 1:1, 3:1, 5:1, and 7:1. In general an N:M ratio would say that the CNN trained on N background examples each epoch for every M foreground examples it trained on that epoch. Because the amount of background to foreground is greater than even 7:1, the subset of background used each epoch was chosen at random from the background in the IDXs and differed each epoch.

Hardware

The CNNs were trained and run on a Mac Pro using a 3.5 GHz 6-Core Intel Xeon E5 processor.

Evaluation of the Results

For the evaluation of the results, the main quantifier used was the difference between the population estimate by the CNNs and the population estimate by the experts. All CNNs, even those trained on data from the matched citizen scientists, were compared against the expert count. This allows a comparison of expert produced data and citizen scientist produced data as training data for neural networks.

How the CNNs performed after the feedback loop compared to before was also examined. The base training iteration, named iteration 0, happens before any retraining and therefore serves as a baseline.

The estimates generated by the CNNs were graphically represented at each iteration. These include the summary statistics of min, mean, and max error for each CNN.

CHAPTER 7

RESULTS

Three runs were conducted for each configuration of training set and background to foreground ratio. The results of the blob counter over the prediction images were averaged (Table 2). CNNs trained on the expert dataset and the CNNs trained on the matched dataset both had low error. Interestingly, the CNNs trained on the matched data performed better under higher background to foreground ratios than the ones trained with expert data. One possible reason for this is that the citizen scientist data is matched while the expert data is not. There was not enough expert data to do matching over it, and there are confirmed cases of expert misclassification.

CNNs that went through the feedback loop were compared to their respective baselines (Table 3). Even one iteration of the feedback loop dropped errors significantly. The decrease in error after one training iteration was larger than the decrease in error that happened when the sampling rates were changed (compare the first two lines in a cell to the first line in two different cells that used the same training set).

As far as sampling rates go, while increasing the sampling of background did reduce error in the baseline, it actually seemed to increase the overall error after using the feedback loop. The exception to this was going from a 1:1 ratio to a 3:1 ratio when using the matched dataset. This suggests that the bias introduced from the large ratios caused too many false negatives in the retraining, as seen by the fact that the population predictions after the feedback loop are consistently low for all ratios other than 1:1.

The estimates generated by the CNNs for each configuration of training set

Table 2: Blob Counter Results

Training set	**BG:FG**	**Predict**	**Actual**	**Error**	**|%Error|**
Expert	**1:1**	**348.33**	**331**	**17.33**	**5.24**
Expert	3:1	288.67	331	-42.33	12.79
Expert	5:1	255.00	331	-76.00	22.96
Expert	7:1	218.00	331	-113.00	34.14
Matched	1:1	398.67	331	67.67	20.44
Matched	***3:1***	***318.00***	***331***	***-13.00***	***3.93***
Matched	5:1	301.33	331	-29.67	8.96
Matched	7:1	271.33	331	-59.67	18.03

CNNs were trained using given training set and the background to foreground sampling ratio given by BG:FG. The Predict column is the population prediction on the test set. The Actual column is the actual count over the test set by our expert users. The numbers given are the average of the best iteration results of each run. Bold face rows are best for their training set. Italicized row is best overall.

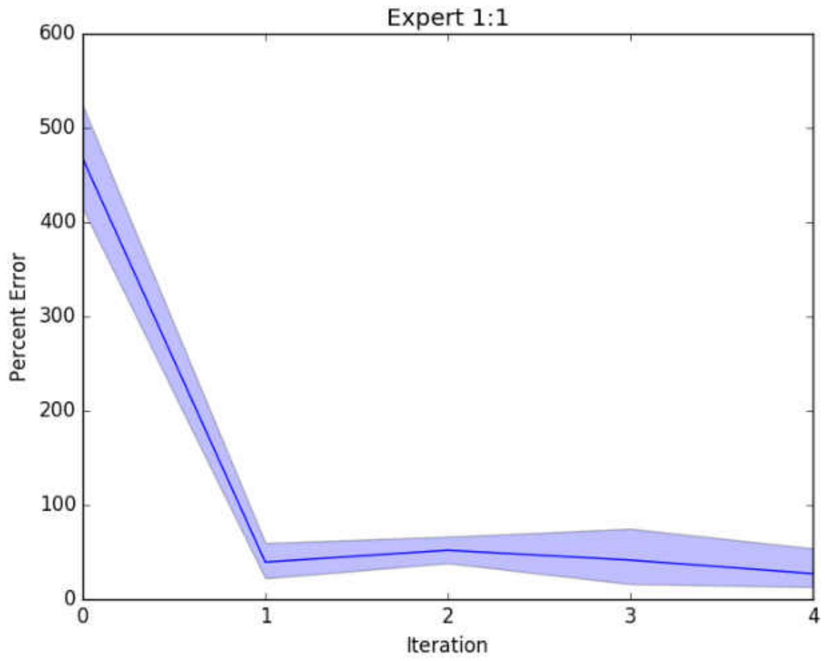
and background to foreground ratio were graphically represented at each iteration. The worst error obtained by any CNN that had been through the feedback loop at all, did better than the very best baseline (Figure 13; a 215 goose under-estimate for the worst feedback CNN over expert 7:1 compared to 273 over-estimate for the best baseline run over matched 7:1).

Table 3: Comparison of feedback loop to baseline.

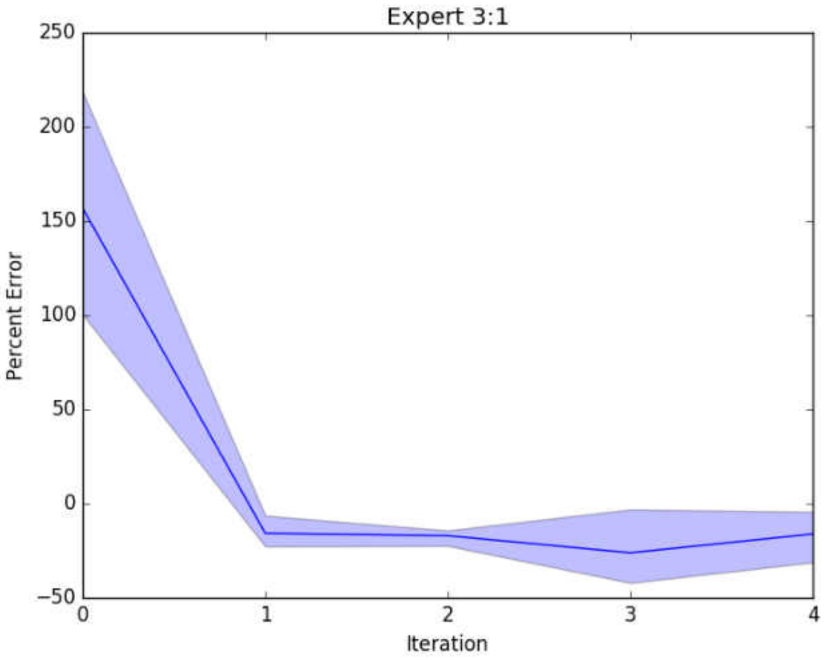
Training set	BG:FG	Iteration	Predict	Actual	|%Error|
Expert	1:1	0	2518.33	331	660.83
Expert	1:1	1	468.67	331	41.59
Expert	1:1	best (3.67)	348.33	331	5.24
Expert	3:1	0	850.00	331	156.80
Expert	3:1	1	279.00	331	15.71
Expert	3:1	best (3.00)	288.67	331	12.79
Expert	5:1	0	699.00	331	111.18
Expert	5:1	1	224.00	331	32.33
Expert	5:1	best (1.67)	288.67	331	22.96
Expert	7:1	0	626.33	331	89.22
Expert	7:1	1	203.33	331	38.57
Expert	7:1	best (1.33)	218.00	331	34.14
Matched	1:1	0	1878.33	331	467.47
Matched	1:1	1	461.67	331	39.48
Matched	1:1	best (3.67)	398.67	331	20.44
Matched	3:1	0	1054.33	331	218.53
Matched	3:1	1	330.00	331	0.30*
Matched	3:1	best (2.67)	318	331	3.93
Matched	5:1	0	856.00	331	151.61
Matched	5:1	1	272.33	331	17.72
Matched	5:1	best (2.67)	301.33	331	8.96
Matched	7:1	0	708.00	331	113.90
Matched	7:1	1	251.00	331	24.17
Matched	7:1	best (2.67)	271.33	331	18.03

* While these numbers averaged to a very low amount of error from the actual, the individual numbers themselves were not the best in their respective runs.

At iteration 0, the feedback loop has not yet been employed, which makes it an effective baseline. It can be seen that even one iteration of retraining drastically cuts the error. The best iteration varied between trials. The average best iteration for each CNN is given in parentheses.



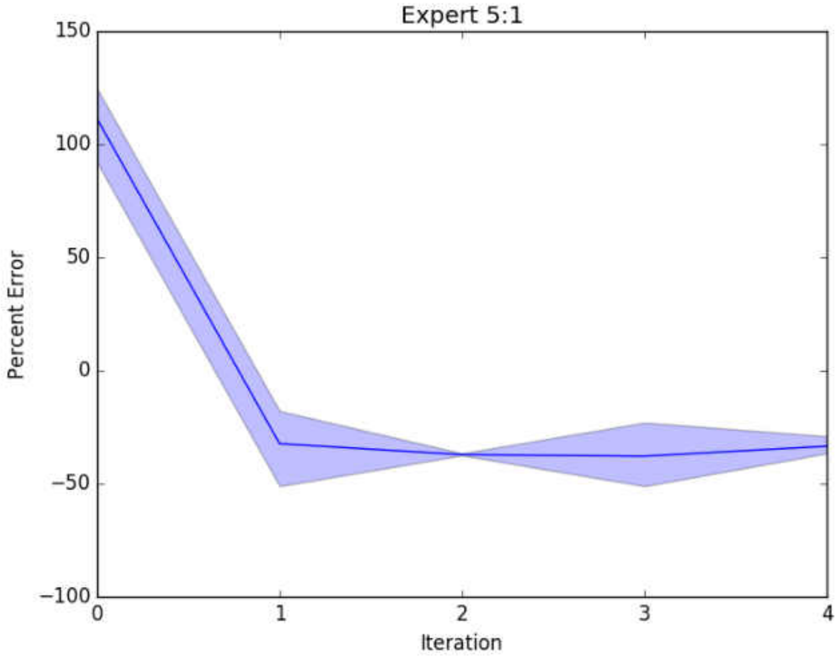
(a) Expert 1:1



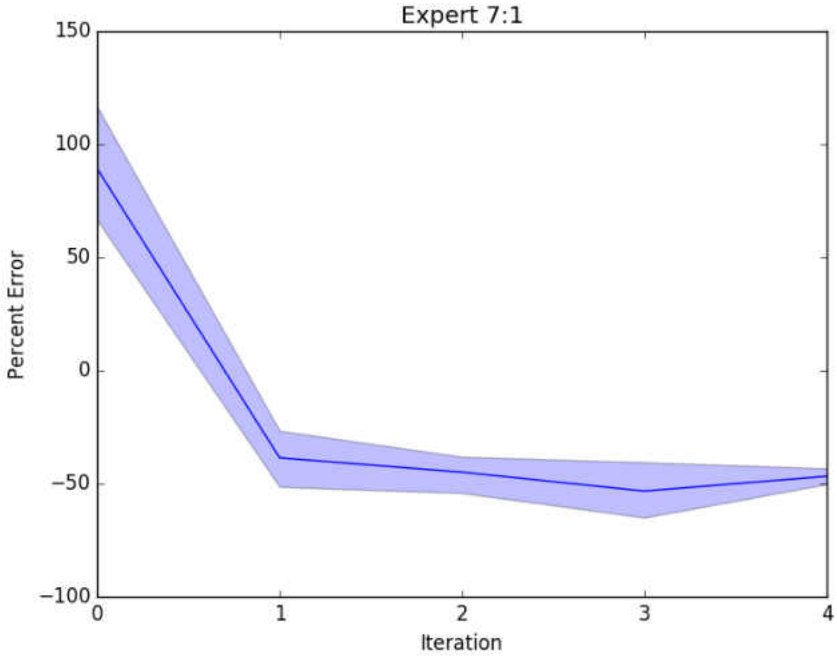
(b) Expert 3:1

Figure 13: Average error based on iteration for each dataset and background to foreground sampling ratio. The line is the average, with the filled in portion showing the maximum and minimum values seen at each iteration. A thinner filled in region has less variance than a thicker one.

Figure 13: cont.

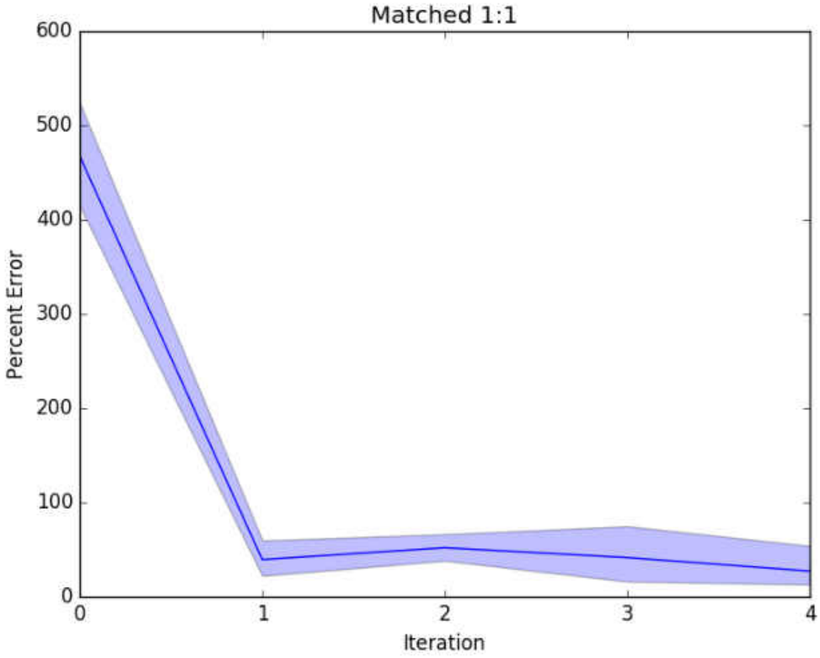


(c) Expert 5:1

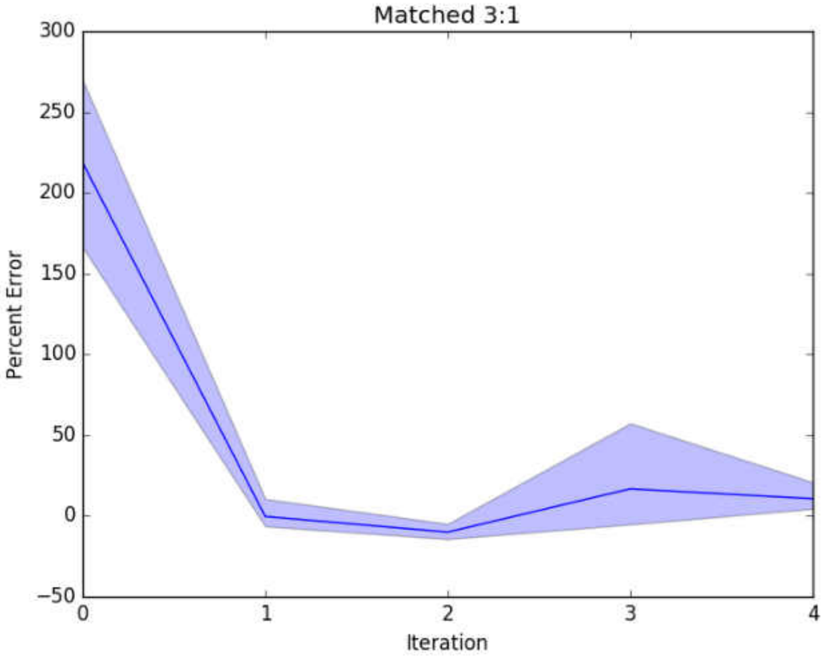


(d) Expert 7:1

Figure 13: cont.

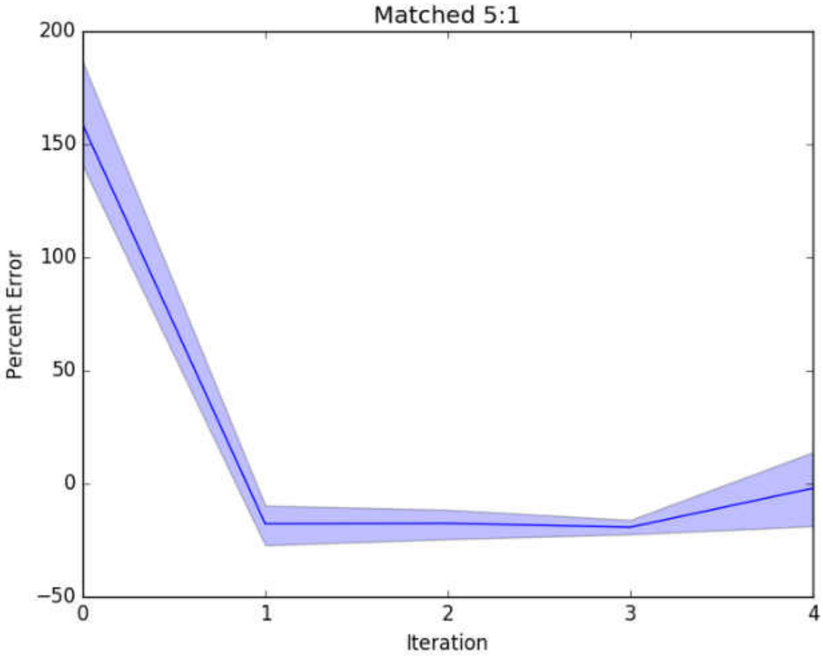


(e) Matched 1:1

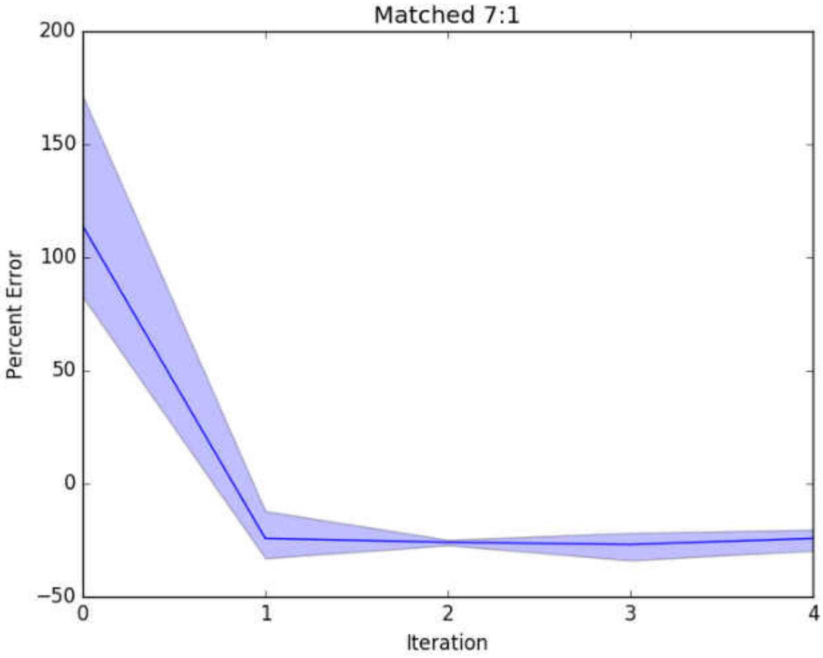


(f) Matched 3:1

Figure 13: cont.



(g) Matched 5:1



(h) Matched 7:1

CHAPTER 8

CONCLUSION

This paper used data gathered from citizen scientists and experts to train convolutional neural networks. These networks were able to provide estimates of the population of white phase snow geese collected from from UAS imagery. While previous work yielded a large number of false positives [18], the addition of a feedback loop in this work drastically reduced the error and yielded runs whose population estimates were not always overestimates.

The feedback loop introduced is simple, yet effective, way to increase accuracy on massively unbalanced datasets. It provided an automated approach to choosing which examples from the majority class were most important to include in training. As the focus of the feedback loop was more the data itself than the CNNs, any new improvements in CNN training techniques could be easily applied to system. In fact, any image classification method that uses supervised training could most likely be used with the feedback loop.

The best results for CNNs trained on the data provided by the citizen scientists had an average error of only 3.93% for their population estimates, down from 150% in previous work. Similarly, CNNs trained on expert provided data had an average error of 5.24% down from 88% in previous work. The low error for both datasets shows both the viability of using citizen scientists to produce training data for CNNs and the viability of using CNNs in ecological research.

Future Work

This project focused on white phase lesser snow geese collected from UAS imagery. In these same images are blue phase lesser snow geese. These blue phase geese are less in number (approximately 1/3 of the population) and more camouflaged than their white phase brethren. As the feedback loop did well on the white phase geese, applying it to the more difficult blue phase geese would be an obvious first step for future work.

Further improvements to accuracy could be made on the predictions for the white phase snow geese. The CNNs used in this work were fairly small and shallow compared to other works. This is good as far as speed goes, but there is the possibility it harmed accuracy. Training deeper networks with more filters could be a way to further improve accuracy. Another potential improvement could be to combine the feedback loop with a system like EXACT [9] which would generate an ideal CNN architecture in an automated fashion.

One of the largest downsides to the feedback loop is the increase in computational cost. The computational cost can never be as low as normal training. This is because the 0th iteration of the feedback loop is simply a normal training, and everything else is then extra. Thus, future work could look into advancements for dealing with datasets such as Wildlife@Home's which is not only unbalanced, but also has a majority class that contains examples which are visually similar to the minority class. The feedback loop makes the assumption that some examples of the majority class are "harder" to identify than others, and seeks to find these hard examples. While what is misclassified by one CNN might not be the same as what is misclassified by another, there are likely similarities. If one could identify the "hard" background prior to the initial training, the feedback loop could possibly be eliminated. Even if it wasn't eliminated entirely, it could "jump start" and need fewer iterations.

Other improvements could be made by applying region based CNNs or a system like YOLOv2 to the Wildlife@Home dataset and seeing if and how those systems could be integrated into the feedback loop.

BIBLIOGRAPHY

[1] Lion Research Center, University of Minnesota, [Accessed Online, 2012] http://www.snapshotserengeti.org/.

[2] R. Bonney, C. B. Cooper, J. Dickinson, S. Kelling, T. Phillips, K. V. Rosenberg, and J. Shirk, “Citizen science: a developing tool for expanding science knowledge and scientific literacy,” *BioScience*, vol. 59, no. 11, pp. 977–984, 2009.

[3] T. Phillips and J. Dickinson, “Tracking the nesting success of north america’s breeding birds through public participation in nestwatch,” 01 2008.

[4] C. Wood, B. Sullivan, M. Iliff, D. Fink, and S. Kelling, “ebird: engaging birders in science and conservation,” *PLoS biology*, vol. 9, no. 12, p. e1001220, 2011.

[5] S. Xu and Q. Zhu, “Seabird image identification in natural scenes using grabcut and combined features,” *Ecological Informatics*, vol. 33, pp. 24–31, 2016.

[6] A. Abd-Elrahman, L. Pearlstine, and F. Percival, “Development of pattern recognition algorithm for automatic bird detection from unmanned aerial vehicle imagery,” *Surveying and Land Information Science*, vol. 65, no. 1, p. 37, 2005.

[7] L.-P. Chrétien, J. Théau, and P. Ménard, “Visible and thermal infrared remote sensing for the detection of white-tailed deer using an unmanned aerial system,” *Wildlife Society Bulletin*, vol. 40, no. 1, pp. 181–191, 2016.

[8] Y. LeCun and C. Cortes, "Mnist handwritten digit database," *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, 2010.

[9] T. Desell, "Developing a volunteer computing project to evolve convolutional neural networks and their hyperparameters," in *e-Science (e-Science), 2017 IEEE 12th International Conference on.* IEEE, 2017.

[10] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[11] P. Y. Simard, D. Steinkraus, J. C. Platt *et al.*, "Best practices for convolutional neural networks applied to visual document analysis." in *ICDAR*, vol. 3, 2003, pp. 958–962.

[12] D. C. Ciresan, U. Meier, J. Masci, L. Maria Gambardella, and J. Schmidhuber, "Flexible, high performance convolutional neural networks for image classification," in *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, vol. 22, no. 1. Barcelona, Spain, 2011, p. 1237.

[13] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[14] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," *CoRR*, vol. abs/1311.2524, 2013. [Online]. Available: http://arxiv.org/abs/1311.2524

[15] R. Girshick, "Fast r-cnn," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1440–1448.

[16] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Advances in neural information processing systems*, 2015, pp. 91–99.

[17] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," 2009.

[18] C. Bowley, M. Mattingly, S. Ellis-Felege, and T. Desell, "Toward using citizen scientists to drive automated ecological object detection in aerial imagery," in *e-Science (e-Science), 2017 IEEE 12th International Conference on*. IEEE, 2017.

[19] M. Mattingly, A. Barnas, S. Ellis-Felege, R. Newman, D. Iles, and T. Desell, "Developing a citizen science web portal for manual and automated ecological image detection," in *e-Science (e-Science), 2016 IEEE 12th International Conference on*. IEEE, 2016, pp. 223–232.

[20] J. C. Fernández, C. Hervás, F. J. Martínez, P. A. Gutiérrez, and M. Cruz, "Memetic pareto differential evolution for designing artificial neural networks in multiclassification problems using cross-entropy versus sensitivity," in *International Conference on Hybrid Artificial Intelligence Systems*. Springer Berlin Heidelberg, 2009, pp. 433–441.

[21] L. Wan, M. Zeiler, S. Zhang, Y. L. Cun, and R. Fergus, "Regularization of neural networks using dropconnect," in *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, S. Dasgupta and D. Mcallester, Eds., vol. 28, no. 3. JMLR Workshop and Conference Proceedings, May 2013, pp. 1058–1066. [Online]. Available: http://jmlr.org/proceedings/papers/v28/wan13.pdf

[22] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol, "Stacked denoising autoencoders: Learning useful representations in a deep

network with a local denoising criterion," *Journal of Machine Learning Research*, vol. 11, no. Dec, pp. 3371–3408, 2010.

[23] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International Conference on Machine Learning*, 2015, pp. 448–456.

[24] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

[25] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *Proc. ICML*, vol. 30, 2013, p. 1.

[26] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates *et al.*, "Deep speech: Scaling up end-to-end speech recognition," *arXiv preprint arXiv:1412.5567*, 2014.

[27] Z. Cai, X. He, J. Sun, and N. Vasconcelos, "Deep learning with low precision by half-wave gaussian quantization," *arXiv preprint arXiv:1702.00953*, 2017.

[28] A. Y. Ng, "Feature selection, l 1 vs. l 2 regularization, and rotational invariance," in *Proceedings of the twenty-first international conference on Machine learning*. ACM, 2004, p. 78.

[29] Y. Liao, S.-C. Fang, and H. L. Nuttle, "A neural network model with bounded-weights for pattern classification," *Computers & Operations Research*, vol. 31, no. 9, pp. 1411 – 1426, 2004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0305054803000972

[30] N. Srebro, J. Rennie, and T. S. Jaakkola, "Maximum-margin matrix factorization," in *Advances in neural information processing systems*, 2005, pp. 1329–1336.

[31] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting." *Journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[32] D. E. Rumelhart, G. E. Hinton, R. J. Williams *et al.*, "Learning representations by back-propagating errors," *Cognitive modeling*, vol. 5, no. 3, p. 1, 1988.

[33] Y. Nesterov, "A method of solving a convex programming problem with convergence rate o (1/k2)," in *Soviet Mathematics Doklady*, vol. 27, no. 2, 1983, pp. 372–376.

[34] A. Karpathy, "Stanford University CS231n: Convolutional Neural Networks for Visual Recognition." [Online]. Available: http://cs231n.github.io/

[35] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[36] T. Tieleman and G. Hinton, "Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude," COURSERA: Neural Networks for Machine Learning, 2012.

[37] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980

[38] D. A. Fischer, M. E. Schwamb, K. Schawinski, C. Lintott, J. Brewer, M. Giguere, S. Lynn, M. Parrish, T. Sartori, R. Simpson, A. Smith, J. Spronck, N. Batalha, J. Rowe, J. Jenkins, S. Bryson, A. Prsa, P. Tenenbaum, J. Crepp, T. Morton, A. Howard, M. Beleu, Z. Kaplan, N. vanNispen, C. Sharzer, J. DeFouw, A. Hajduk, J. P. Neal, A. Nemec, N. Schuepbach, and V. Zimmermann, "Planet hunters: the first two planet

candidates identified by the public using the kepler public archive data," *Monthly Notices of the Royal Astronomical Society*, vol. 419, no. 4, pp. 2900–2911, 2012.

[39] R. Simpson, K. R. Page, and D. De Roure, "Zooniverse: observing the world's largest citizen science platform," in *Proceedings of the 23rd international conference on world wide web.* ACM, 2014, pp. 1049–1054.

[40] C. J. Lintott, K. Schawinski, A. Slosar, K. Land, S. Bamford, D. Thomas, M. J. Raddick, R. C. Nichol, A. Szalay, D. Andreescu, P. Murray, and J. Vandenberg, "Galaxy zoo: morphologies derived from visual inspection of galaxies from the sloan digital sky survey," *Monthly Notices of the Royal Astronomical Society*, vol. 389, no. 3, pp. 1179–1189, 2008.

[41] C. Lintott, K. Schawinski, S. Bamford, A. Slosar, K. Land, D. Thomas, E. Edmondson, K. Masters, R. C. Nichol, M. J. Raddick, A. Szalay, D. Andreescu, P. Murray, and J. Vandenberg, "Galaxy zoo 1: data release of morphological classifications for nearly 900,000 galaxies," *Monthly Notices of the Royal Astronomical Society*, vol. 410, no. 1, pp. 166–178, 2011.

[42] D. G. York, J. Adelman, J. E. Anderson Jr, S. F. Anderson, J. Annis, N. A. Bahcall, J. Bakken, R. Barkhouser, S. Bastian, E. Berman *et al.*, "The sloan digital sky survey: Technical summary," *The Astronomical Journal*, vol. 120, no. 3, p. 1579, 2000.

[43] M. A. Voss and C. B. Cooper, "Using a free online citizen-science project to teach observation & quantification of animal behavior," *The american biology Teacher*, vol. 72, no. 7, pp. 437–443, 2010.

[44] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[45] J. Redmon and A. Farhadi, “YOLO9000: better, faster, stronger,” *CoRR*, vol. abs/1612.08242, 2016. [Online]. Available: http://arxiv.org/abs/1612.08242

[46] C. Rother, V. Kolmogorov, and A. Blake, “Grabcut: Interactive foreground extraction using iterated graph cuts,” in *ACM transactions on graphics (TOG)*, vol. 23, no. 3. ACM, 2004, pp. 309–314.

[47] T. Cover and P. Hart, “Nearest neighbor pattern classification,” *IEEE transactions on information theory*, vol. 13, no. 1, pp. 21–27, 1967.

[48] Y. Freund, R. E. Schapire *et al.*, “Experiments with a new boosting algorithm,” in *Icml*, vol. 96, 1996, pp. 148–156.

[49] J. H. Friedman, “Additive logistic regression: a statistical view of boosting,” *Ann. Statist.*, vol. 28, pp. 337–407, 2000.

[50] L. Breiman, “Random forests,” *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[51] A. Gomez, A. Salazar, and F. Vargas, “Towards automatic wild animal monitoring: identification of animal species in camera-trap images using very deep convolutional neural networks,” *arXiv preprint arXiv:1603.06169*, 2016.

[52] T. Desell, “Citizen science grid,” 2017, [Accessed Online 2017] https://csgrid.org/csg/.

[53] K. A. Goehner, “Using computer vision and volunteer computing to analyze avian nesting patterns and reduce scientist workload,” Master’s thesis, University of North Dakota, 2015.

[54] C. Bowley, A. Andes, S. Ellis-Felege, and T. Desell, “Detecting wildlife in uncontrolled outdoor video using convolutional neural networks,” in

e-Science (e-Science), 2016 IEEE 12th International Conference on. IEEE, 2016, pp. 251–259.

[55] D. P. Anderson, "Boinc: A system for public-resource computing and storage," in *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on.* IEEE, 2004, pp. 4–10.

[56] S. L. Peterson, R. F. Rockwell, C. R. Witte, and D. N. Koons, "The legacy of destructive snow goose foraging on supratidal marsh habitat in the hudson bay lowlands," *Arctic, Antarctic, and Alpine Research*, vol. 45, no. 4, pp. 575–583, 2013.

[57] G. P. Jones IV, L. G. Pearlstine, and H. F. Percival, "An assessment of small unmanned aerial vehicles for wildlife research," *Wildlife Society Bulletin*, vol. 34, no. 3, pp. 750–758, 2006.

[58] K. S. Christie, S. L. Gilbert, C. L. Brown, M. Hatfield, and L. Hanson, "Unmanned aircraft systems in wildlife research: current and future applications of a transformative technology," *Frontiers in Ecology and the Environment*, vol. 14, no. 5, pp. 241–251, 2016.

[59] D. Ciregan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," in *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on.* IEEE, 2012, pp. 3642–3649.

[60] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[61] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.