A NEURAL NETWORKS APPROACH TO DETERMINING ANGLE AND SCALE OF PARTIALLY OCCLUDED OBJECTS

A Thesis

Submitted to the School of Graduate Studies and Research

in Partial Fulfillment of the

Requirements for the Degree

Master of Science

Thomas J. Gresavage

Indiana University of Pennsylvania

August 2016

Indiana University of Pennsylvania
School of Graduate Studies and Research
Department of Mathematics

We hereby approve the thesis of

Thomas J. Gresavage

Candidate for the degree of Master of Science

Advisor Howard E. Donley, Ph. D.
Assistant Professor of Mathematics, Advisor

Yu-Ju Kuo, Ph. D.
Assistant Professor of Mathematics

Charles Lambe, Ph. D.
Assistant Professor of Mathematics

ACCEPTED

Randy L. Martin, Ph.D.
Dean
School of Graduate Studies and Research

Title: A Neural Networks Approach to Determining Angle and Scale of Partially Occluded Objects

Author: Thomas J. Gresavage

Thesis Chair: Dr. Howard E. Donley

Thesis Committee Members: Dr. Charles Lamb
Dr. Yu-Ju Kuo

A team of biological researchers wishes to investigate the eating behvaiors of local insects to determine whether they prefer indigenous or invasive plant species. Current techniques involve researchers collecting leaf samples from the field and estimating the percentage of consumed area. This method is especially time consuming but may also be subject to a great deal of error due to individual bias and the erratic nature of human estimation. The task is to construct a program which accepts images of leaves, taken in a controlled environment, and returns estimates for the scale and angle at which the leaf occurs in the image. These values will then be used to limit the search space of the generalized Hough transform algorithm. The scale will be used to determine the missing leaf area. This would reduce the labor intensity of data collection and eliminate humans as a primary source of error.

Table of Contents

Appendices

List of Tables

List of Figures

Chapter 1

INTRODUCTION

Motivation;

Estimating the missing area of a leaf or other known object is generally straightforward concept for humans, however it presents a great many hurdles in the realm of image processing. Since the size of the input image may vary, the images must be scaled or cropped so that the area represented by a single pixel in a reference image represents the same area in a target image. Additionally, the leaves themselves have some standard size deviation, so the original leaf size, location, and angle must be determined from a partially eaten leaf. Since some arbitrary amount of the leaf is missing from the image many useful heuristics for determining the attributes of interest must be abandoned for more robust alternatives. Moreover, this program must be able to accept a wide variety of plant species and still produce accurate results.

Current methods for addressing this problem are computationally costly and thus time consuming. This paper aims to provide a fast and accurate alternative using multilayer neural networks. Neural networks offer an attractive alternative to classical image recognition techniques due to their fast speed. In order for neural networks to be implemented, the image must be converted into a vector of uniform dimension which characterizes the image while maintaining the necessary information to be investigated. This preprocessing step is often the most time-consuming and baffling part of any neural network approach.

Inspired by the GHT, this paper investigates the use of edge curvature at rotationally-uniformly-spaced points to determine the scale and angle offsets of a partially eaten leaf. Using curvature as the defining characteristic of a leaf should produce good results as curvature is sensitive to scaling, and the rotational uniformity of the points will provide sensitivity to rotations. Additionally, the performance of various network topologies at identifying the scale and offset angle will be examined.

Each type of plant will have its own neural network. The neural networks will be trained on images and contours of complete and partially eaten leaves to produce the desired output. The output will be the scale and angle at which the leaf appears with respect to some reference scale and orientation. These parameters may then be passed to other algorithms, such as the GHT, to reduce the search space and thus improve processing time.

A number of different neural structures will be examined to determine the best result. This includes varying the number of inputs and hidden nodes, number of layers, and network topology (fully connected, convolutional, etc.). Two characteristics will be used to determine the performance of the networks; error, and to a lesser degree training time.

Previous Research

In this section we discuss a few of the existing methods used for object recognition and feature extraction. We outline the advantages and disadvantages of these methods and demonstrate the need for a more robust and natural way of dealing with image data.

Curvature Scale Space

Classical approaches for dealing with occluded objects in images vary greatly. For example, an occlusion as encountered by the curvature scale space (CSS, Abbasi et al.) algorithm results in the peak values missing from the scale-space representation. The algorithm is able to overcome this since its score is based on how well the locations of the largest curvature features which do appear correlate to the features which appear in all the models. As one might expect, the more significant the features are that are missing from the image, the greater the likelihood the correct model will not be among the highest scoring models (since less significant features will become relatively more significant and falsely contribute to the match score).

The Generalized Hough Transform

Another method of object recognition which has proven to be effective in dealing with the issue of partial object occlusion is the generalized Hough transform (GHT, Ballard). This algorithm first chooses a reference point in an edge image containing the object to be identified. The radius (r) and angle (θ) from each edge point to the reference point is calculated. For each gradient direction along the edge a list is created containing the radii and angles of all edge pixels which have such a gradient direction and assembled into a table referred to as the *R-table* of the object.

Once the reference image is scanned, and the R-table is built, a four-dimensional array is created representing all possible locations (x, y), scales (s), and angles (φ) at which the object might appear. The image to be matched is converted to an edge image and the gradient direction is calculated. For each edge pixel in the image, the list of (r, θ) values is retrieved from the R-table row which matches the gradient direction of the current edge pixel. These (r, θ) are then used to extrapolate an (x, y) candidate for the true reference point location. Extrapolation is repeated across all s and φ in the search space and the corresponding entry in the accumulator array is then incremented. Possible object locations, angles, and scales are simply maxima in the accumulator. The GHT is robust to occlusions since the edges which appear in the image will still correlate strongly to the original object, provided enough of the object appears in the image for a match to be declared.

The number of updates made to the accumulator is roughly proportional to the perimeter of the object in the target image. If too much of the object is missing from the image, then the results become erroneous. Ballard mentions a few ways of dealing with this and thus improving the accuracy of the algorithm. First, one may choose to update accumulator entries by the gradient magnitude at the current pixel location. Pixels with large gradients signify a clear divide between foreground and background, thus they make for good indicators of the presence of an object. Additionally, one may update the accumulator by the current pixels curvature. Large curvature values indicate that a corner is present, and corners are unique features and hence excellent object descriptors. Indeed, the this is progenitor of the inspiration for the current research.

Unfortunately, the speed of the GHT suffers from a number of factors including memory usage and processing time. Since the accumulator is a 4-D array, it requires that a huge block of memory be reserved. Moreover, the access time and number of floating point operations puts a large strain on even the most advanced serial CPUs. For this reason, it is incredibly common for the GHT to be executed as parallel code. In fact, the parallelizability of the GHT is one of its main advantages. This advantage is, however, not incredibly accessible to many individuals or industries, nor is it especially deployable for the consumer market.

Artificial Neural Networks

Many early neural network approaches to computer vision generally involve convolving a filter kernel over the entire image to extract features such as edges and corners. The objects contained in the image are then quantifiably characterized in some way by the appearance of features. These characteristics are passed to the network to then be operated on. It is difficult, however, to find a balance between speed and accuracy in this regard. Improving accuracy generally means handling larger data or performing many costly operations on the available information. Speeding up code often means the data is in some way altered from its original form to be more compact. This increased compactness tends to lose information since it is generally some form of mapping, which may have no clear or tractable inverse mapping, to a smaller subspace than the original image space.

The CNN approach deals with occlusions in much the same way that humans do. When the network scans an image for an object the overlapping filter maps allow the network to have some global awareness of the entire image at once. This is advantageous since it gives the network inherent locational independence. That is since the filter maps overlap, if an object lies on the border of a particular filter map, the network is able to use the information contained in the neighboring filter map as if the object lied entirely within the first filter map. This is helpful when dealing with partial occlusions since the activation of each neuron depends only on whether part of the object is found in the filter map.

Moreover, neural networks excel at pattern recognition as a result of the advanced techniques used for training. Since the network will be trained on both whole and and partially eaten leaves. In essence, this assures that the network knows in advance how to handle partial information. Hence why the choice of training data is incredibly important. In practice, good training data is considered to be a random sampling of all possible inputs the network may face. This helps to ensure that the network gives the desired output for all possible inputs, even those which were not included during training. If the sample is not sufficiently random, there may be important characteristics of the pattern not contained in the sample and thus not internalized by the network. In deployment, this yields admirable results when using input data closely related to the training data, but a steep falloff in accuracy when encountering new data.

When compared to the aforementioned methods of handling partial object obstruction or omission, the neural network approach is incredibly straightforward and simple. If one wishes to account for cases where object information is missing, then we must simply include a sampling of these cases during training. On the otherhand, if it is known that the data will, in most cases, be ideal, or if handling occlusions is simply not important, then we may casually omit these cases during training. Perhaps the most advantageous aspect of using neural networks in this case is that the network may be retrained at any time. This means that if at first occlusions are not to be considered, but later it is decided that these cases must indeed be accounted for, we may add new training data at-will and update the weights for the neurons accordingly. This allows for a great deal of versatility and control when designing and implementing a neural solution.

The last and perhaps most important advantage of neural networks is their speed. The simplicity of a neural network is the largest contributing factor to its out-performance of many of the other methods available. By taking tensor inputs adding a bias, multiplying them by a weight tensor, and then computing the resulting activation, neural networks reduce the arduous task of feature extraction and data manipulation to a few simple matrix multiplication and floating point arithmetic operations. The reduced computational cost is so great that http://arxiv.org/abs/1509.06791

Strategies

In this section we discuss the various tactics used to convert between raw image data and object descriptors. It is of the utmost importance that information necessary for determining object characteristics is preserved. Thus, we must carefully choose how the image is preprocessed, and the strategy for making those decisions is herein explained.

Curvature as an Indicator of Scale

The aim of this research is to determine if a multilayer perceptron (MLP), or combination of CNN and MLP is able to learn a rule to determine the scale and angle of partially eaten leaves using edge curvature values. Curvature is a measure of the change in angle of the tangent line to a curve to the change in arc length. The choice of curvature as the defining feature is due to it's sensitivity to scale. If we let ϕ be the angle of the tangent line at some point along a curve, and s be the arc length through some angle θ, then curvature is simply $\frac{d\phi}{ds}$. The sensitivity of curvature to scale comes from the arc length $s = \sqrt{x^2 + y^2}\theta$, since $\phi = \arctan(dy/dx)$ is clearly independent of scale. For a parametric curve given as $c(t) = (x(t), y(t))$, the curvature at position t

is defined as

$$\kappa(t) = \frac{\dot{x}(t)\ddot{y}(t) - \ddot{x}(t)\dot{y}(t)}{\left[\dot{x}(t)^2 + \dot{y}(t)^2\right]^{3/2}} \tag{1.1}$$

where $\dot{x}(t)$ is the derivative of $x(t)$ with respect to t. Since the exact equations defining $x(t)$ and $y(t)$ are not known we can not compute curvature directly. Fortunately, convolution provides a workaround. The convolution of any two functions $f(t)$, $g(t)$ is defined as follows:

$$(f \star g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \tag{1.2}$$

It is important to note that $(f \star g)(t) = (g \star f)(t)$. From (1.2) we can show that the derivative of a convolved function may be exactly computed by knowing the derivative of either one of the functions. Symbolically we will show:

$$\frac{d}{dt}(f \star g)(t) = (f \star \dot{g})(t) \tag{1.3}$$

Proof. Let $f(t)$, $g(t)$ be everywhere differentiable on $\mathbb{R}$. Then

$$\begin{aligned}
\frac{d}{dt}(f \star g)(t) &= \frac{d}{dt}\int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \\
&= \int_{-\infty}^{\infty} \frac{d}{dt}f(\tau)g(t-\tau)d\tau \\
&= \int_{-\infty}^{\infty} f(\tau)\frac{d}{dt}g(t-\tau)d\tau \\
&= \int_{-\infty}^{\infty} f(\tau)\dot{g}(t-\tau)d\tau \\
&= (f \star \dot{g})(t)
\end{aligned}$$

■

Thus we may convolve a known function with our parameterized curve $(x(t), y(t))$ and use its derivatives to calculate curvature. The function to be used is the well-known Gaussian distribution centered at t, with standard deviation σ. The choice of σ is at the discretion of the user and should vary depending on the original image size. The contour is parameterized using the method outlined in and the Gaussian is then convolved with the pixel coordinates. This has the effect of smoothing the curve in the edge image as if it had undergone a Gaussian blur and edge detection without the need for additional costly edge finding operations. Increasing σ smooths artifacts, noise, and small features out of the image. Choosing a small value of sigma will result in a noisy signal, whereas too large of a value will blend the important features together. Usually a σ of around one tenth the maximum image dimension yielded good results.

Signatures as an Indication of Angle

An object signature is any way of converting higher dimensional (two or more) image data into lower dimensions. In this study, we will be taking 2-D image data and converting it into one dimension to be fed as the input vector to a neural network. In order for serialization to work, the image features must be parameterizable. We could use t, the location along the curve as the parameter, but this may become infeasable for long contours in large images and unreliable for partially occluded leaves. Instead we will use ψ, the angle between $(x(t), y(t))$ and the horizontal axis in standard counter-clockwise orientation. In this way we extract the curvatures of rotationally-evenly spaced points along the boundary. By ensuring the points are evenly spaced and always starting at the same location the network has a better chance of learning to associate, or expect, certain curvature values to appear at certain nodes. When these values appear elsewhere, the network should be able to identify which node they should have come from, and thereby learn the relationship between the input data and orientation angle.

Network Topologies and Hyperparameters

There will be a total of three different network topologies examined in this research: fully connected MLP with multiple hidden layers, 1-D convolutional feeding into a fully connected network, and 2-D convolutional feeding into a fully connected network. Within these latter two topological paradigms we will further investigate the benefits of pretraining the convolutional network.

The MLP will be used as a baseline to determine the effectiveness of the most basic 1-D convolutional structure. An in-depth explanation of how we relate the MLP structure to the CNN is given in the next chapter. After a baseline analogue has been established, the true testing may begin. The networks' hyperparameters will be varied to determine an ideal configuration to achieve the desired results. The hyperparameters to be investigated are as follows: the number of convolutional layers, number of dense layers, and number of learnable filters.

The convolutional layers will serve as to encode the information into the network, while the fully-connected layers will extrapolate the data. The number of learnable filters determines the depth of the abstractions the network is able to extrapolate from the image and thus is expected to play a major role in the overall network accuracy.

It is important to establish the trade-off between computational effort and accuracy since the final product is aimed at simplifying the work of biologists in the field. If the scientists invest numerous hours training a network with a large number of layers, when similar or even more accurate results would have been achievable with fewer layers and less training time, then the extra effort will have been in vain.

Chapter 2

PREPARING THE DATA

Preprocessing

By choosing the input layer to be convolutional we effectively minimize the amount of preprocessing and information necessary to develop an answer. This is quite an attractive feature of convolutional neural networks as it preserves all of the information. Preprocessing the image data is often the most time consuming part of a neural approach to computer vision. In fact, once the training data is generated, the only preprocessing that must be done is to crop and scale the image to an appropriate size. The training data will be constructed from a single reference leaf. This leaf should be comprised exclusively of whole petioles, otherwise the automatic data generation will fail to produce usable data.

Resizing and Reorienting the Leaves

The object oriented features of the Python programming language are fully implemented to simplify the automatic generation of training data from an image. First, an image containing the reference leaf is loaded into the program. The image is segmented to find the petioles residing in the image. After segmentation, each leaf in the image is extracted and scaled to be 64×64 pixels. The reference leaves are converted to parametric contours using a simple boundary following algorithm which stores edge pixel locations as an ordered list of (x, y) pixel location pairs. This makes removing segments of the leaf a simple matter of deleting splices of the list and replacing them with new location values. The inserted values are generated at random to represent the random nature of how the edge of a consumed leaf may appear.

Before the leaves may be used to generate data, they are used to instantiate a `Leaf` object. This object is itself a subclass of a `Contour` object. A `Contour` has a number of methods which are useful when dealing with parametrized objects. One such method is `curvature`, which calculates the curvature at each point along the contour using the approach outlined in the work of Abbasi et al. (1999). This curvature is used to determine the orientation of the leaf from a completely verticle position. To accomplish this, the point with the highest curvature at a given Gaussian standard deviation σ is found. The angle between the centroid of the contour and the pixel of interest is assumed to be the angle at which the leaf is offset from vertical and the entire contour is rotated by this amount. Additionally, the contour list is shifted so that the point of interest is the first element of the list.

Indexing and Chain Codes

Before continuing much further it is important to establish some common notation. When referring to pixel locations, (x, y) will refer to the coordinates of a pixel in the standard plane, assuming the downward direction for y is positive. However, $< y, x >$ refers to the same pixel. This is so that there is consistency with the indexing convention in Python where indeces are designated as (row index, column index).

Consider the 8-neighbors of an arbitrary pixel labeled from 0 to 7 in a clockwise fashion beginning with the pixel immediately to the left. This will be the convention for identifying the relative positions of neighboring pixels. The label will be referred to as the pixels $c$hain code. So the pixel with chain code 4 is the neighbor immediately to the right of the current pixel. Figure 2.1 illustrates this concept.

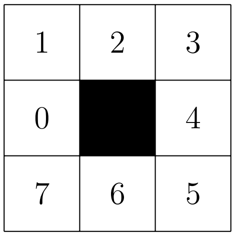


Figure 2.1. The 8-neighbors of an arbitrary pixel labeled by *chain codes*.

The Pixelwalk Algorithm

The `Leaf` class has its own useful methods for generating random data. One such method is the `pixelwalk` method. This function walks from left to right starting at (0,0) and ending when it reaches some specified width. The walk pseudo-randomly determines the next pixel based on a Gaussian distribution centered on the straight-line direction to the ending point. For example, the user wishes to have their walk be 128 pixels to the right of where they started. The destination pixel is then set to $< 0, 128 >$. Let's say that at some time in the walk we find ourselves at the point $< y, x >$. The destination pixel will lie at some angle θ given by:

$$\theta = \arctan 2(y, 128 - x) \tag{2.1}$$

Where $\arctan 2$ is the IEEE standard two-argument arctangent function. Since we are dealing with pixel movements we cannot move in the exact direction specified by this angle. So we must map $(-\pi, \pi] \rightarrow [0, 8)$. When creating the mapping we need to ensure that $-\pi$ maps to 8, 0 maps to 4, and π maps to 0 (since down is defined to be positive). This is easily accomplished using the mapping c defined below.

$$c(\theta) = 4\left(1 - \frac{\theta}{\pi}\right) \tag{2.2}$$

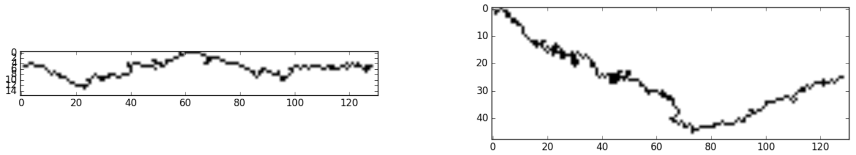
This value gives us the chain code of the next pixel we will move to if we were to head directly for the destination. Since we wish to introduce some randomness to this we will generate another mapping $g : [0, 8) \to (-\infty, \infty)$ so that we may use the standard Gaussian distribution. This mapping and its inverse are given by:

$$g(c) = \ln\left(\frac{c}{8-c}\right) \tag{2.3}$$

$$g^{-1}(t) = \frac{8}{1+\exp(-t)} \tag{2.4}$$

where $c = c(\theta)$ is the chain code from the previous mapping. We need the inverse mapping so that we can take the random sample from $\mathbb{R}$ back to our chain code. Next we simply call the random number generator to grab a sample from the desired distribution; Gaussian with $\mu = g(c)$ and $\sigma = \sigma_u \beta$. Here, σ_u is a user-specified standard deviation, and β is itself a random variable from a β-distribution where $\alpha_\beta = 2$, $\beta_\beta = 5$. The values for α_β and β_β are somewhat arbitrary.

The reason the standard deviation of the Gaussian is chosen to be random is so that the eccentricity of the path varies in a manner that is similar to what one might find in the field. For additional realism, a constant bias is added to each chain code found this way. That is, instead of heading directly for the destination $< 0, 128 >$ with some minor deviations, add a bias to the direction of travel. This has the effect of causing the walk to begin at an angle so the path has a more natural curve to it. The bias is a random integer in the range [-1,1], which corresponds to an angle bias of $\pm\frac{\pi}{4}$ radians. It can be seen from Figure 2.2 that the paths are indeed reminiscent of what one might find in the field. The plot in Figure 2.2(b) is especially exemplary of the effect of the bias and increased σ_u on the path.



(a) No bias, $\sigma_u = 2.1678$

(b) $-\frac{\pi}{4}$ radian bias, $\sigma_u = 3.2940$

Figure 2.2. Output of 128 pixel-width `pixelwalk` with various parameters

Chapter 3

NEURAL NETWORKS

Background

Convolutional Neural Networks

Another solution to the issue of data loss during preprocessing is to use a 2-D convolutional neural network (CNN). The goal of a convolutional approach is to provide the neural network with an abstract understanding of image features in a process similar to that of the visual cortex.

In a eukaryotic eye, incoming light activates either a rod, or cone receptor. If the light hits a rod, the receptor is activated at three levels corresponding to red, green, and blue. These individual activations make up the total activation of the entire receptor. In the visual cortex, neurons receiving inputs from the receptors are not fully connected to the receptor layer. Downstream layers are instead connected to a small subset of proximal receptors, with each receptor belonging to many such neighborhoods, or feature maps. The collection of feature maps forms a cover of the image space.

As applied to artificial neural networks this equates to convolving a kernel having shared weights (all red receptors behave similarly, all blue receptors behave similarly, etc.) over an input. A feature is considered to be present if it causes any one of the feature maps in a small neighborhood to be activated. This is the essence max pooling which is widely used in conjunction with convolutional layers, however other pooling or downsampling methods may also be used. Traditionally, the pooling layers are strided in such a way as to form a perfect cover of the convolutional layers' output, however this is not required.

In addition to greatly reducing the input size, pooling adds translational invariance to the network since the appearance of a feature anywhere in the pool will cause its activation. Specifically, a pool with shape $(2, 2)$ will have the same activation if a feature appears in any one of the four quadrants. Moreover, this reduces the dimensionality of the input by a multiplicative factor. In the case of a $(2, 2)$ pooling layer with a stride of 2, it would reduce the dimensionality by half along each axis.

Figure 3.1 shows the typical structure of a 1-D convolution feeding into a max pool layer before ultimately being passed to an output layer with a single node. At the convolutional layer, all of the connections which share a color also share their weights. The pooling layer has a pool size of 2 and a stride of 2. It is clear from this example that the convolutional layer reduces the input dimension by 2 (filter width - 1), and the pooling layer downsamples the convolutional layer by a factor of 2.

Since feature maps are convolved over the input space it creates a ”contiguous comprehension” of the image. That is, instead of interpreting data as a collection of singletons, the network is able to relate neighboring points to one another. Stacking convolutional and pooling layers provides increasingly global awareness of pixel relationships, allowing for greater abstraction. If color image data is used, it also inherently provides the ability to isolate objects and features based on color. Convolutional layers also tend to have shorter training times compared to traditional MLPs taking serialized images as their inputs.

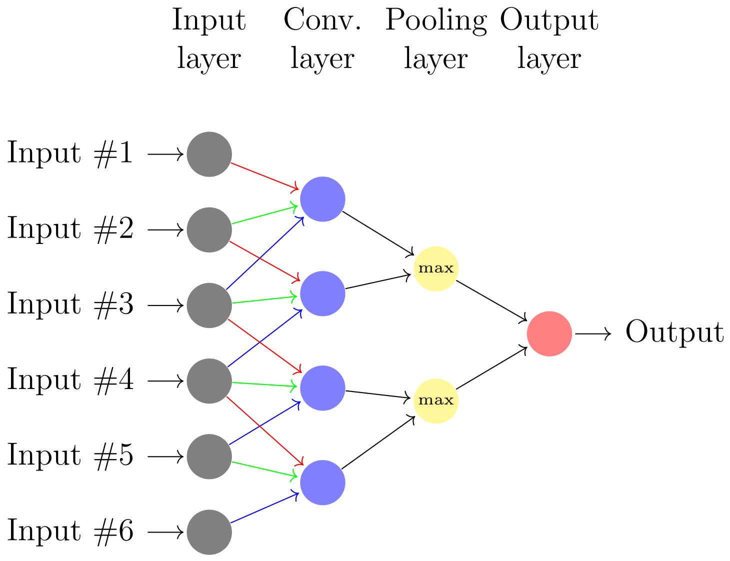


Figure 3.1. 1-D convolutional network with one convolutional, and one pooling layer.

Autoencoders

An approach to overcome this, which has been implemented to varying degrees of success, is to use principal component analysis (PCA, also PC) to determine the salient parameters and reduce the input dimensionality. This can be an arduous task and is certainly limited in its ability to reduce input size while preserving information. Artificial neural networks provide another alternative; the use of an autoencoder. An autoencoder is a feed-forward multilayered unsupervised learning model which lowers the dimensionality of the input before re-expanding it to the original space. Say, for instance, we have an input vector $x \in \mathbb{R}^n$ which we want to be able to express as a vector $y \in \mathbb{R}^m$, with $m < n$. An autoencoder would use some number of intermediate fully connected or convolutional layers to reduce the dimensionality to m. Later layers re-expand the result into $\mathbb{R}^n$ with the objective of reproducing exactly the original vector, hence it is unsupervised. The stages of the autoencoder which together decrease the dimensionality to m are referred to as the encoder. Conversely,

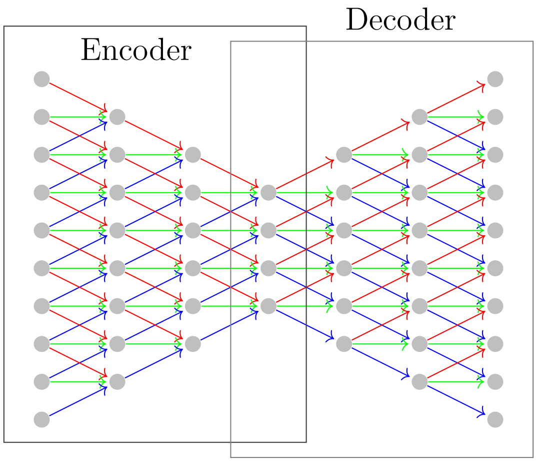


Figure 3.2. 1-D convolutional autoencoder with four encoding and decoding layers.

the stages which restore the original input size are referred to as the decoder.

This concept is diagrammed in Figure 3.2, which is a 1-D fully-convolutional autoencoder with four encoding and decoding layers. Looking carefully at the similarity between the encoding connections and decoding connections an important property of convolutional networks appears: symmetry. That is, if we isolate the connections between two layers we see that downwards convolution (encoding) is symmetric to upwards convolution (decoding). This is made even more explicitly clear in Figure 3.3.

During training, the mapping onto the subspace is created synchronously with its inverse, thus circumventing the tribulations of classical methods. Once trained, an autoencoder is broken in twain and made to encapsulate some other network or machine. During training of the complete network, the weights and biases of the encoder and decoder may or may not be further trained. The effect of an autoencoder is to extract the m most characteristic features which are useful in describing an object within the problem scope. One potential advantage to pretraining a network with

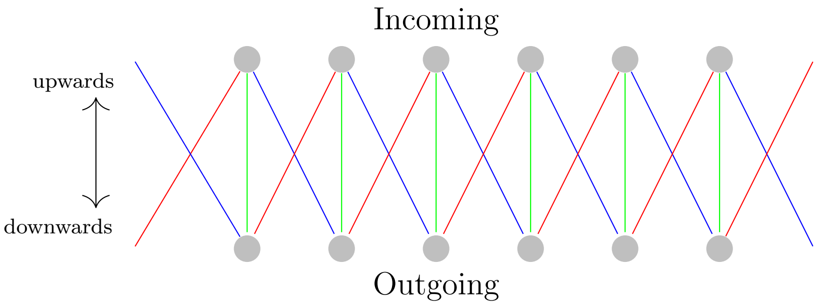


Figure 3.3. Two 1-D convolutional layers showing the symmetry between downwards and upwards convolution.

an autoencoder is a reduced training time, as will be demonstrated in this work. It has been shown that an autoencoder with one hidden layer and linear activation is equivalent to PCAHinton and Salakhutdinov (2006).

Chapter 4

BUILDING THE NETWORK

Initialization

All layer activations and weight initializations are chosen to be tanh and the normalized initialization distribution, hereinafter referred to as the Glorot uniform distribution, given in (4.1) following suggestions of Glorot and Bengio Glorot and Bengio (2010). These authors were able to show that the choice of symmetric activation functions, such as tanh and softmax, are favorable to non-symmetric activations, such as sigmoids, due to their ability to effectively propagate error to lower layers. They were also able to show that initialization with the Glorot uniform distribution further improved training times and parameter updates (especially weight updates) over the standard uniform distribution.

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right] \tag{4.1}$$

Here, $U[-a, a]$ is the standard uniform distribution over the interval $[-a, a]$ and n_j is the number of nodes in the jth layer.

Normalization

For speed and accuracy it is recommended that the inputs to a network be normalized to $[-1, 1]$Ioffe and Szegedy (2015). For this we use a feature provided by Lasagne called `BatchNormLayer` which normalizes the inputs according to

$$x' = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}\gamma + \beta$$

where μ and σ^2 are the mean and standard deviation for the batch, ϵ is a small positive number. Of notable importance is that μ and σ^2 are computed across examples in a batch and not per exampleIoffe and Szegedy (2015); Contributors (2016).

Multilayer Perceptron

In order to determine whether or not there is an advantage to the convolutional topology when working with curvature information we must also build a fully connected MLP with which to compare the results. We must also make sure that this fully connected network has identical hyperparameters to the 1-D CNN whenever possible. That is, after each convolution and pooling layer the number of neurons decreases in a consistent manner. Specifically, using a filter stride of 1 (convolving the filter along every neuron, without skipping) the dimensionality reduces by `filter_size - 1` after each convolutional layer. After max pooling with a pool size of 2 the dimensionality is halved. Convolutional networks have an additional hyperparameter which is the number of filters, or feature maps, the network has. In order to minimize discrepancy the number of feature maps will be set to 1, as there is no MLP analogue for a 1-D CNN with more than this many.

1-D Convolutional Network

By representing the leaf as an edge contour the data is reduced from a 2-D pixel matrix to a 1-D vector of edge pixel locations. The vector is then analyzed to determine the curvature at each of these edge pixels which further reduces the (x, y) coordinate pairs to a single value, $\kappa_{x,y}$. The nature of the curvature data is convolutional, in that the curvature at one point is related to the curvature at its neighbors so we will use a 1-D convolutional network to bolster the performance and improve training times.

The vector of curvatures will have filters of width 3 convolved along it. The output of these filters is fed into a max pooling layer with a pool shape of 2; reducing the dimensionality by half. Similar layer pairs are stacked three more times for a total of four convolutional and four max pooling layers. Lasagne provides 1-D convolutional and max-pooling layer functionality natively through`Conv1DLayer` and `MaxPool1DLayer` respectively.

2-D Convolutional Network

The 2-D convolutional network will take 64×64 pixel images of leaves as its input. Then, using square filters of shape $(3, 3)$ and max pooling with a pool shape of $(2, 2)$ the image is encoded in this way total of four times, as in the 1-D case with curvature, before being passed to a feedforward MLP. As mentioned before, the main purpose of the convolutional layers is simply to encode the information into the network. The task of determining the scale and angle will ultimately be the work of the MLP layers.

Autoencoding

Pretraining layers of a network improves the overall training time since convergence to extrema is sensitive to the initial position or guess, and saturation prevents propagation of error through layers of a especially with non-symmetric activation functionsGlorot and Bengio (2010). That is, during early training iterations the layers nearest to the output see the greatest changes to their weights and biases, but may become saturated and unable to improve their output. By the time the error is back-propagated to earlier layers the updates to weights and biases is virtually non-existent. Thus, upstream layers must wait for convergence to occur on downstream layers first before they experience very much error being back-propagated, i.e. updates to their weights and biases. Autoencoding avoids this issue by ensuring the networks weight and bias initialization is nearer to a local maximum or minimum.

There are many different strategies one may choose when taking an autoencoding approach. In this study, we choose to tie the weights and biases of the encoding and decoding layers so they are the same. After training of an autoencoder, it happens quite frequently that the encoding weights are the same as the decoding weightsIm et al. (2015). The work of Im, Belghazi and MimisevicIm et al. (2015) show that every autoencoder with tied weights always defines a conservative vector field, and thus able to reconstruct the input vector accurately.

While training the main network, the pretrained weights and biases are allowed to be updated so that any information of special importance necessary to extract the scale and angle from an image may be incorporated to these encoding layers.

Choosing the Training Algorithm

Perhaps the most important aspect when developing an effective neural network solution is choosing an appropriate training algorithm. Traditionally, stochastic gradient descent (SGD) is used and has been proven immensely effective at locating optima. Additionally, Lasagne provides a number of SGD variants, as well as RMSProp, AdaGrad, AdaDelta, Adam, and Adamax. This research chose *Adam* as its training algorithm due to its superior performance over SGD, its variants, and

AdaGrad, Kingma and Ba (2014).

In short, Adam is an adaptive learning algorithm based on first and second moment estimates of the true gradients. The Adam algorithm tunes the learning rate at each time step or iteration based on these moment estimates and is able to overcome many obstacles of sparse and non-stationary data, Kingma and Ba (2014). While the data used in this study is stationary, it is indeed sparse since much of the image is, in fact, not the leaf. Curvature, when calculated from an image, is relatively sparse since digitalization of the subpixel boundary results in many straight lines, i.e. zero-curvature points. Hence the attractiveness of the *Adam* algorithm.

The network is to be feed-forward and consist of two convolutional two perceptron layers. The input layer is chosen to be convolutional so that the network may learn the feature of the leafs continuity. In so-doing it provides the ability to extrapolate the angle.This should also provide additional robustness to occlusions. This improvement is attributed to the shared weights among feature maps which prevent bad leaf segments from adversely affecting the system during training. This may be further improved by associating a weight to the teaching data corresponding to some relative measure of the intactness of the leaf. In this implementation, the measure shall be the relationship between the number of points defining the partially eaten leaf contour and the original contour whence it came.

Table 4.1. 1D Convolutional Test Cases

No. Filters	No. Conv. Layers	No. Dense Layers
2	2	2
	3	2
	3	3
	3	4
	4	2
	4	3
	4	4
4	2	2
	3	2
	3	3
	3	4
	4	2
	4	3
	4	4
8	2	2
	3	2
	3	3
	3	4
	4	2
	4	3
	4	4
16	2	2
	3	2
	3	3
	3	4
	4	2
	4	3
	4	4

Varying Parameters

In order to determine the relationship between hyperparameters and the performance of the network we deployed the design of experiments methodology. Tables 4.1 and 4.2 show the test space for the 1D and 2D networks respectively in their entirety.

Table 4.2. 2D Convolutional Test Cases

No. Filters	No. Conv. Layers	No. Dense Layers
6	2	2
	3	2
	3	3
	3	4
	4	2
	4	3
	4	4
12	2	2
	3	2
	3	3
	3	4
	4	2
	4	3
	4	4
24	2	2
	3	2
	3	3
	3	4
	4	2
	4	3
	4	4
48	2	2
	3	2
	3	3
	3	4
	4	2
	4	3
	4	4

Chapter 5

RESULTS

Chapter 6

CONCLUSIONS

Chapter 7

FUTURE WORK

Performance Improvements

While the code performs admirably, there are plenty of areas where further improvement may be made. This section explores just some of the potential avenues for advancement.

Graphics Processing Unit Support

Since training and execution of neural networks is heavily reliant on linear algebraic operations, processing times may be vastly improved (10× to 50× fasterContributors (2016)) employing the graphics processor of a computer instead of the CPU. Theano provides support for this but only for *NVidia* cards with *CUDA* support. The vast majority of this research was conducted on a machine which used an on-board *Intel* graphics card with no CUDA cores. For this reason, no effort was made to implement GPU processing. Since the training times for convolutional networks with many filters are exceptionally large, implementing GPU processing would be quite favorable.

Parallel Processing Support

As was mentioned in the previous subsection, any effort to improve the training times of the large networks is quite favorable. To this end, future work should include support for parallel processing. There are a number of steps where parallel processing can be easily implemented such as generating the random data, training of the various networks, and solving minibatches of data.

Parallelizing of Random Data Generation

Currently, random data is generated using one leaf at a time as a seed for a number of randomly eaten leaves. This is quite a time consuming step thus it is an attractive candidate for parallelization. A simple parallelization would have the scheduler assign one leaf seed to each node so that there is no need for communication between nodes. This guarantees processing time would fall off inversely proportional to the number of parallel nodes.

Parallelizing of Network Training

Depending on the hyperparameters of the network, training quickly becomes the most computationally expensive step in the entire process. To expedite this process, it would be advantageous to parallelize this segment of the code. To accomplish this, each parallel node would be assigned to train one or more networks. In so doing, processing time fall-off would be bounded by the inverse of the number of parallel nodes. The advantage is a bound, and not direct proportion since the training of one network may take longer than another and is serial in nature, i.e. an unused node may not take up the process of training a network without significant overhead from the scheduler negating the advantage of parallelization. Parallelization is especially ideal for processing on machines which do not meet the requirements for GPU processing.

Parallelizing of Batch Processing

Once the network has been trained, it is likely that the researcher will be passing large sets of data to the network. This is known as a batch and by its nature batch processing is highly parallelizable. Each node would be assigned to either a minibatch of the whole batch or an individual sample from the batch (minibatch of one). There is little overhead cost since analysis of one minibatch is independent of the analysis of other minibatches. Therefore, parallelization here would result in the processing time falling off inverse-proportionally to the number of parallel nodes.

Code Optimizations

There are a number of places where brute force measures were taken for the sake of simplicity during development. This results in a great deal of inefficiencies and extensive processing time. The portion of code with the greatest room for improvement is the preparation of the image data. This includes the generation of random leaf data, reorientation, scaling, and resizing of leaves, parameterizing of leaf boundaries, calculation of curvature, etc. Since the construction and training of the network is handled almost entirely by Lasagne and Theano, there is likely little room for improvement here by this code.

Additional Features

The current realization has a great deal of useful features that one may wish to tinker with. There are, however, a great many additional bells and whistles which would prove useful for a positive research experience.

Coordinate Predictions

The current network implementation only predicts the scale and angle offset of a leaf. As has been demonstrated, the various networks are able to do this satisfactorily. There are two remaining parameters necessary to describe the leaf for the purpose of feedback and they are the x- and y-coordinates for the center of the uneaten leaf. Future work will add predictions for these two parameters to the output of the network. An advantage of the neural network approach here is that the prediction for each of these coordinates will be in $\mathbb{R}$ and thus potentially able to overcome the discretization of the subpixel image.

Area Calculation

Since the work of this project feeds into the task of determining missing area, it would be useful if a neural network itself would be able to determine the missing leaf area. This way, a neural network would handle the task from start to finish and thereby contribute its advantages wholly to the undertaking of calculating the amount of leaf eaten.

further adjust weights of training data inversely proportional to SD during pixel walk -¿ high SD means perimeter probably longer.

Design of Experiments

The need for a form of design of experiments functionality became painfully apparent while testing the various network architectures. It would be useful to automate sweeping the number of filters, convolutional and dense layers, learning rate, etc. through a range of values. The user would specify which fields they wish to test, what ranges they wish to test over, and what discretization they wish to use, and the program would automatically build a design space to build and test the various configurations.

Performance Analysis

It would be incredibly useful if the code were able to automatically save the performance analysis such as training time, training error, processor time, etc. to a file for later analysis. The current implementation simply prints these values to the screen, but does not save them to a file anywhere.

Network Visualization

Due to the graphical nature of neural networks, one feature which may prove useful to various researchers and students would be the ability to visualize the network graph. A future implementation would be able to take the hyperparameters passed to the network constructor and then build an image of the network layers and its connections. The user would be able to turn on and off the plotting of various attributes such as "only show convolutional layers", or "hide connections", etc. These would provide the users with an accessible way of approaching neural networks as well as the inherent usefulness of visual feedback when building these structures.

Furthermore, it would be useful to visualize the output of each layer as an image or other form of data visualization so the behavior of the network, i.e. its black-box nature may be unmasked.

Progress Feedback

Since generating the data and training the network is quite time-consuming, feedback as to the progress of the program would be immensely helpful. This is already lightly implemented during data generation where an optional progress bar may be displayed. This implementation uses a package in Python, ProgressBar, which provides support for simple construction and updating of a progress bar.

Chapter 8

ACKNLOWEDGEMENTS

I would like to acknowledge the various parties who contributed to the success of this project. Firstly, without the support of the Indiana University of Pennsylvania Mathematics and Computer Science departments numerous hurdles would have been otherwise insurmountable. Special thanks to my advisor, Dr. Edward Donley, whose advisement directly contributed to the overcoming of a great many image processing and general debugging obstacles. Thanks to the S-COAM scholarship program provided by Indiana University of Pennsylvania, funded by grants from the National Science Foundation. Thanks to the Indiana University of Pennsylvania Department of Biology whose research interest provided the inspiration for this thesis. The Indiana University of Pennsylvania graduate assistantship program, without which I would likely not have been introduced to Dr. Donley or the research of the Biology department. Finally, my grandfather, James Gresavage Sr., whose mathematical ability, work ethic, and never-ending perseverance provided the inspiration to pursue an advanced degree in applied mathematics, always aim high, and accomplish my goals.

Bibliography

Abbasi, S., F. Mokhtarian, and J. Kittler (1999, nov). Curvature scale space image in shape similarity retrieval. *Multimedia Syst. 7*(6), 467–476.

Ballard, D. H. (1987). Readings in computer vision: Issues, problems, principles, and paradigms. Chapter Generalizing the Hough Transform to Detect Arbitrary Shapes, pp. 714–725. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Contributors, L. (2016). Welcome to lasagne.

Glorot, X. and Y. Bengio (2010). Understanding the difficulty of training deep feedforward neural networks.

Hinton, G. E. and R. R. Salakhutdinov (2006). Reducing the dimensionality of data with neural networks. *Science 313*(5786), 504–507.

Im, D. J., M. I. D. Belghazi, and R. Memisevic (2015). Conservativeness of untied auto-encoders. *CoRR abs/1506.07643*.

Ioffe, S. and C. Szegedy (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR abs/1502.03167*.

Kingma, D. P. and J. Ba (2014). Adam: A method for stochastic optimization. *CoRR abs/1412.6980*.