Demystifying Deep Convolutional Neural Networks

Adam Harley (adam.harley<at>ryerson.ca)

Version 1.1

Abstract. This document explores the mathematics of deep convolutional neural networks. We begin at the level of an individual neuron, and from there examine parameter tuning, fully-connected networks, error minimization, back-propagation, convolutional networks, and finally deep networks. The report concludes with experiments on geometric invariance, and data augmentation. Relevant MATLAB code is provided throughout, and a downloadable package is available at the end of the document.

Contents

1. Introduction

Artificial neural networks (ANNs) [1] are at the core of state-of-the-art approaches to a variety of visual recognition tasks, including image classification [2] and object detection [3]. For a computer vision researcher interested in recognition, it is useful to understand how ANNs work, and why they have recently become so effective.

An artificial neural network is a type of biologically-inspired pattern recognizer. The basic concept comes from four key insights from psychology:

1. the mammalian brain is comprised of billions of similar-looking cells (called neurons), arranged in a dense and complicated network [Ramon y Cajal];
2. neurons communicate with one another through the propagation of "all-or-none" electrochemical signals [Otto Loewi 1933, Cole, Hodgkin];
3. neurons are classifiers, which specialize with training [Hubel63a, Hubel63b, Hubel63c]; and
4. the brain learns patterns by adjusting the efficacy of communication between pairs of neurons [Hebb].

These observations have been taken to represent the following guidelines for building an artificial pattern recognizer:

1. the recognizer should be built as a network of nearly-identical units;
2. the units should communicate their outputs to one another in binary (i.e. with 1 and 0);
3. the unitary function of the recognizer should be a trainable classifier; and
4. learning should be achieved by adjusting weights on the connections of the network.

Almost all neural network algorithms follow these guidelines, and a few (e.g. [HMAX]) follow the brain's biology even more closely. Inferring algorithmic guidelines from biological findings is partially motivated by the idea that replicating what is known about the biological solution to vision may offer a "head start" to a computational solution, perhaps even before the biological solution is fully understood. In this case, however, the inferred guidelines can also be justified on the basis that they enforce scalability (guideline 1), are computationally convenient (guidelines 2 & 4), and make intuitive sense (guideline 3).

As these insights accumulated, a variety of neural network algorithms were developed (e.g. [McCulloch & Pitts, 1943]), but early attempts suffered from two critical bottlenecks. The first bottleneck was computational. Even a relatively small vision task, such as recognizing handwritten digits, requires a network with hundreds of units, and the computation required to simulate and train such a network is not trivial. It was not until the invention of the backpropagation algorithm [Werbos, PhD thesis 1974] that large (multi-layer) neural networks could feasibly be trained. Today, extremely large neural networks are still computationally expensive to train, and modern implementations rely on GPUs to perform the bulk of the computation.

The second bottleneck was a lack of training data. Without training data that accurately captures the variability in the problem of interest, almost any learning machine is doomed to overfit. Neural networks are especially vulnerable to this problem, since they involve so many trainable parameters. This has not yet been solved algorithmically; the success of neural networks in a problem area is still closely tied to the quality and size of the largest dataset in that area. It is for this reason that neural networks were not successful in categorizing handwritten digits until the U.S. Postal Service scanned, segmented, and labelled 60,000 images of hand-drawn digits for researchers in the 1980s [Lecun90]. Similarly, neural networks were not successful in categorizing images of objects and scenes until after the rise of image-sharing on the internet, which led to the development of datasets with millions of labelled images over hundreds of categories [Krizhevsky, ImageNet]. For problems where training data is limited, this bottleneck remains a serious issue.

In the select areas where the training data bottleneck has been overcome, neural networks have been revealed to have a variety of remarkable attributes. For example, ANNs perform similarly to humans on certain categorization tasks: tasks that are difficult for ANNs tend to also be difficult for humans [Serre, 2007]. Similarly, it has been found that the behaviour of nodes in hierarchical ANNs correlates with the behaviour of neurons at various levels of the human visual system [Yamins, 2014]. These findings suggest that the vision techniques ANNs arrive at through training are similar to the techniques our brains use. From the perspective of inferring a "head start" on computer vision from biology, this is a validating and encouraging result, because it suggests that ANNs truly do mimic the brain.

Due to these successes (and more), ANNs are currently a popular tool in computer vision. Neural networks even see some publicity in news media (e.g. [Robert McMillan]), under names like "artificial intelligence," and "deep learning." But what really goes on in an artificial neural network? Looking past the biology, the history, and the present popularity, what makes ANNs so powerful? As this document will show, moving beyond a "black box" treatment of the topic requires becoming familiar with the underlying mathematics.

The remainder of this document is organized as follows:

- Section 2 defines what a neuron is mathematically, and gives an energy function to minimize.
- Section 3 describes how the “backpropagation” algorithm works.
- Section 4 discusses how a neural network can be implemented in MATLAB, and how to make sense of the various implementation parameters.
- Section 5 shows results on a simple digit categorization task.
- Section 6 gives a short discussion of the results.
- blah blah redo this list.
- at the end of the document, we will revisit the biological guidelines, and review where and how we followed them, and where and how we might like to deviate from them.

Full MATLAB code for the project is available here.

2. Neural Networks as Functions

The answer to "what makes ANNs so powerful?" is this: An ANN is a large, hierarchical, high-dimensional function, which can be easily tuned produce particular outputs, given some range of inputs. Whether the inputs are images, audio files, or stock prices, the fundamental mathematics of this function are the same.

In this document, we will work towards finding a function f that maps from images to digits, i.e. $f: I \mapsto C$, where I is a square image of a hand-drawn digit (28×28 pixels), and C is a column vector of 10 probabilities, one for each digit. For example, we would like to have

$$f\left(\image{0}\right) = \begin{bmatrix}1\\0\\0\\0\\0\\0\\0\\0\\0\\0\end{bmatrix}, \quad f\left(\image{1}\right) = \begin{bmatrix}0\\1\\0\\0\\0\\0\\0\\0\\0\\0\end{bmatrix} \quad \label{goal}$$ $$f\left(\image{9}\right) = \begin{bmatrix}0\\0\\0\\0\\0\\0\\0\\0\\0\\1\end{bmatrix}.$$

That is, given an image of a particular digit, f will assign a 1 to that digit's corresponding element in the output vector C, and a 0 to each of the other elements. This is a difficult problem, because hand-drawn digits are often ambiguous. It is also an appropriate working example, because it is one of the first vision problems that neural networks successfully solved [Lecun89].

We will construct f as a dense network of simple functions, representing neurons. In the human brain, there are billions of neurons, and trillions of connections

between them. Due to the scarcity of computational resources, our ANN will be far less complex, and consequently not as powerful. However, we will eventually achieve 99% accuracy on digit classfication.

3. Artificial Neurons

A neural network is a large function, composed of smaller functions. In biological neural networks, the unitary function is a neuron (see Figure 1).

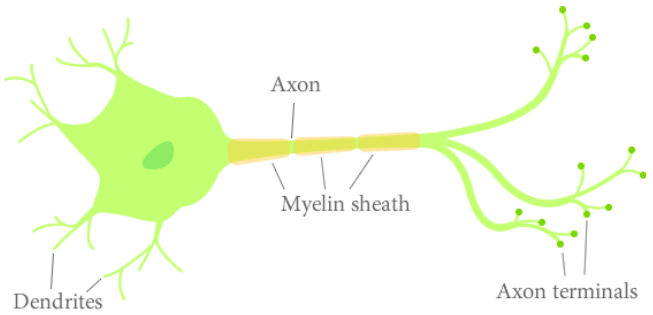

Figure 1. A biological neuron. Electrochemical stimuli from other neurons arrive at the dendrites, causing changes in polarity. If the neuron's electrical potential reaches a certain threshold, the neuron "fires", sending an electrical impulse into the axon. The myelin sheath speeds the propagation of the impulse toward the axon terminals. At each axon terminal, when the signal arrives, the neuron releases molecules into a synapse, where a dendrite of another neuron awaits.

A neuron receives inputs from other neurons, and if the sum of those inputs exceeds a certain threshold, the neuron "fires", sending signals to other neurons. This process can be illustrated in a flow chart, as shown in Figure 2.

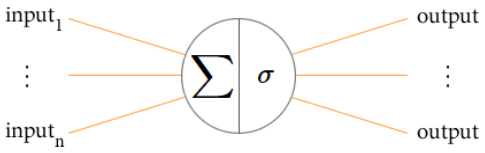

Figure 2. A simplified neuron. Inputs from other neurons are summed, and this sum is passed through a threshold function, denoted by σ. The output of this function is sent to other neurons.

The full picture of biological neurons is not, in reality, so simple, but this flow chart will serve as a good starting point for designing an ANN. Next, we will (1) choose an appropriate inner function for our artificial neurons, (2) allow our artificial neurons to adjust to their input (i.e. learn), (3) organize these neurons in some logical way.

3.1. Definition

We will denote the inner function of artificial neurons with σ. In accordance with biology, this function will take as input a sum of other neurons' outputs, and output some binary-like value, indicating a classification result.

The simplest option for σ is a function that outputs 0 when its inputs are below a certain threshold, and outputs 1 when its inputs are above that threshold (Figure 3, first graph):

$$\sigma(x) = \begin{cases} 0, & \text{if } x < \text{threshold},\\[2ex] 1, & \text{if } x >= \text{threshold},\\ \end{cases}$$

where x is a summation of the neuron's inputs. This function has the advantage of being fast to compute. However, it is discontinuous at the threshold, and therefore not differentiable there. Furthermore, the derivative below and above the threshold is the same: 0. This is an issue because later we will use the derivative to determine on which side of the threshold we are on.

To address these problems, ANNs often use a smooth version of the threshold function: either the logistic sigmoid function (Figure 3, second graph), $$ \sigma(x) = {1\over{1+e^{-x}}},$$ or the $\tanh$ function (Figure 3, third graph), $$\sigma(x) = \tanh(x).$$ These smooth functions have the additional benefit of a "region of uncertainty": rather than always outputting 0 or 1, they can output in-between values, which sometimes may be more appropriate.

Recently, the hinge loss function (Figure 3, fourth graph), $$ \sigma(x) = max(0,x),$$ has also been used. The hinge loss function is discontinuous at the threshold, but its derivative there can be specified manually (to be 0), and everywhere else the derivative is very simple (either 1 or 0). This function also has a "region of uncertainty", since it has a slope near the threshold.

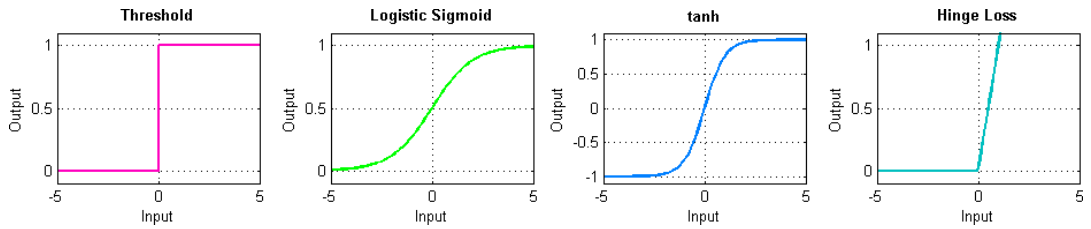

Figure 3. Options for the inner function of an artificial neuron. From left to right: a threshold function (with $\text{threshold} = 0$); the logistic sigmoid ${1\over{1+e^{-x}}}$; $\tanh(x)$; and the hinge loss function $max(0,x)$.

The advantages and disadvantages of the four options are summarized in Table 1.

Table 1

Properties of Four Inner Functions for ANNs

	Threshold	Logistic Sigmoid	tanh	Hinge Loss
Fast to compute?	YES	NO	NO	YES
Simple derivative?	YES	YES	YES	YES
Continuous?	NO	YES	YES	NO
Region of uncertainty?	NO	YES	YES	YES
Directional derivative?	NO	YES	YES	YES

In this document, we will use the logistic sigmoid function,

$${\sigma(x) = }{1\over{1+e^{-x}}}\label{ref1}.$$

This function has the nice property that the derivative can be written in terms of the original function, since

$${d \over dx}{\sigma(x)} = {\sigma(x)(1 - \sigma(x))}\label{ref2}.$$

This means that we once we have the output of the function, we can calculate the derivative with a simple subtraction. We will take advantage of this property later.

3.2. Flexibility

The input to a neuron's inner function, denoted x, is a sum of other neurons' outputs. Rather than using a simple sum, we will use a weighted sum. In the context of real neurons, these weights represent "connection strengths" between pairs of neurons. These weights will add flexibility to our artificial neurons, making it possible for them to learn.

First, we will consider the case in which an artificial neuron has only one input. Let O denote the output of the neuron, and let

$$ O = \sigma(\phi w + b), \label{singleInput_neuron}$$

where ϕ is the actual input (e.g. a previous neuron's output), w is a "weight" term, and b is a "bias". The parameter w controls the scaling of the sigmoid function, while b translates the function left and right. Together, these two parameters provide full control of the function's shape (see the interactive Figure 4).

Figure 4 (interactive). The effects of changing w and b. Hover your cursor over the first image to see the effects of changing w. Hover your cursor over the second image to see the effects of changing b. The observation to make here is that by adjusting either w or b or both, it is possible to have the function produce any particular value between 0 and 1, for some given input.

If the function has multiple inputs, let $w_1, \ldots, w_n$ be the weights corresponding to the inputs $\phi_1, \ldots, \phi_n$, and let

$$ O = \sigma(\phi_1 w_1 + \ldots + \phi_n w_n + b). \label{finalSigma}$$

With this representation, if the input ϕ is fixed (or varies little), the parameters $w_1, \ldots, w_n$ and b can be adjusted to guide the output O to any value within $(-1,1)$. We will use equation $\eqref{finalSigma}$ as our approximation of an individual neuron.

3.3. Organization

We would like to create a network of the neuron-like functions defined by equation $\eqref{finalSigma}$. That is, we would like to connect the outputs of one function to the inputs of another, in some structured manner. Biologically, this is where neural networks become extremely complicated. Artificial neural networks, on the other hand, can be arbitrarily simple. For our first neural network, we will structure it as a graph with stacked "layers" of nodes, where every pair of neighbouring layers is fully connected (see Figure 5). In this graph, Layer 0 holds the initial input. In the subsequent layers, every node has a bias (b), and a set of incoming weighted edges ($w_1, w_2, \ldots, w_n$). Using these parameters, each node processes the previous layer's outputs (denoted $\phi_1, \phi_2, \ldots, \phi_n$), and sends its own output O along weighted edges to the next layer.

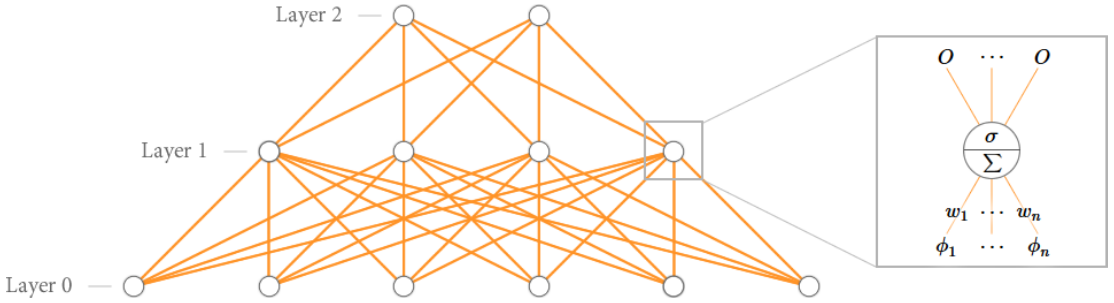

Figure 5. The structure of our artificial neural network. Nodes of neighbouring layers are fully connected. A node receives as input weighted outputs of other neurons, processes them through σ, and outputs the result.

This network of functions will make up our digit-classifying function f, although we will need more nodes in each layer. Layer 0 will hold a one-dimensional version of the 28×28 image, and will therefore need 784 nodes. Layer 2 will need 10 nodes—one for each digit. The centre layer has no predetermined size, and so we will have to experiment to find one. We will also experiment with the number of layers, and with the connection rules that define them, so it is useful to emphasize here that the network architecture is arbitrary.

In our notation for the network, we will distinguish nodes from one another by a single subscript. Since each neuron has its own weights, bias, and output, these terms will all share the subscript of the node. Therefore, to define the output of the qth node on a particular layer, we write

$$O_q = \sigma(\phi_1 w_{1q} + \ldots + \phi_n w_{nq} + b_q), \label{nodeFormula}$$

where ${pq}$ on w_{pq} means that this weight connects the pth node in the previous layer to node q on the current layer. Note that this notation does not specify the layers of the terms involved, but instead uses a short-sighted "current-layer" perspective. If we wished to restate equation $\eqref{nodeFormula}$ with superscripts for layers, we could write

$$O_q^l = \sigma(O_q^{l-1} w_{1q}^l + \ldots + O_n^{l-1} w_{nq}^l + b_q^l).\label{nodeFormula_withLayers}$$

This layer-specific notation is not only hard to read, but unnecessary. As we will see, all of the mathematics of neural networks is carried out one layer at a time.

4. Training

As stated earlier, we have a goal for our function f: given an image of a particular hand-drawn digit, we would like the final layer node corresponding to that digit to output 1, and we would like every other final layer node to output 0, e.g.,

$$f\left(\boxed{}\right) = [1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0]^\top.$$

Since we have a target value for every final layer node, we can define the error of our network as the mean squared error of those nodes,

$$E = {1 \over 2} \sum_{k \in K}(O_{k} - t_{k})^2, \label{error_equation}$$

where K is the set of nodes in the final layer of the network, O_{k} is the output of a final layer node k, and t_{k} is the target value of that node. If we can get O_{k} to be close to t_{k} for all $k \in K $, the network will be producing the output we want, and E will be close to 0.

We want to find the parameters of f that minimize E. We can imagine E to be a surface in a high-dimensional space, and f to be a ball on this surface. The parameters of f, which include $w_{1q}, w_{2q}, \ldots, w_{nq}$ and b_q for every node q, define the position f occupies on E. We want to find the set of parameters that put f on the surface's lowest point.

This can be accomplished through gradient descent: from the current location on E, step in the direction of maximum decrease, and iterate until we reach a minimum. In the one-dimensional case, this simply requires finding the derivative (the slope) of the function at the current location. Using the slope, we can determine which direction is "down," and move in that direction (see the interactive Figure 6, left graph).

In the multi-dimensional case, the direction of steepest descent has as many dimensions as E does, and computing that direction requires finding the partial derivative of E with respect to every parameter involved. Adjusting each parameter along the direction of its corresponding negative partial gradient represents one step downward along the function surface. Therefore, for any node q in the network, we need to find the partial derivative of E with respect to each individual weight w_{pq} connecting to it, i.e. ${\partial E} \over {\partial w_{pq}}$, and also the partial derivative of E with respect to that node's bias, b_q, i.e. ${\partial E} \over {\partial b_q}$. So, for every node q, we will be computing a ${\partial E} \over {\partial w_{pq}}$ for every incoming edge pq, and also a single ${\partial E} \over {\partial b_q}$.

Gradient descent is a greedy algorithm, which means we are not guaranteed to find the global minimum of E. The algorithm may only find a local minimum of the function, or worse, it may overstep a minimum and not find one at all (see the interactive Figure 6, right graph). To address the overstepping problem, the main solution is to use a very low step coefficient. Smaller steps means more time must be spent iterating toward the solution, but speed is less important than correctness. There is no solution to the local vs. global issue. However, we can still justify using gradient descent if we assume that the function we are optimizing is convex.

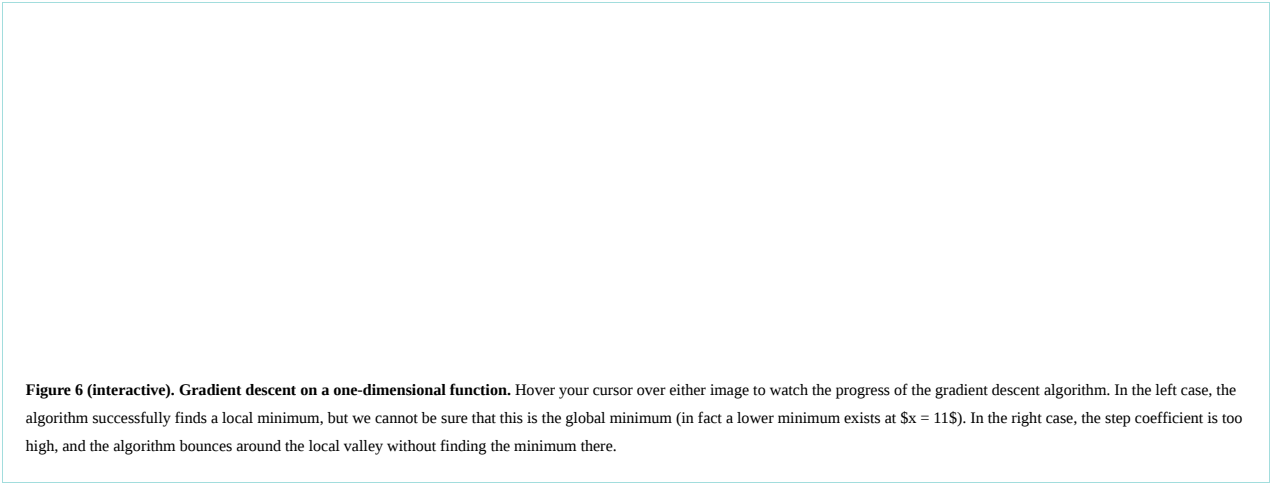

Figure 6 (interactive). Gradient descent on a one-dimensional function. Hover your cursor over either image to watch the progress of the gradient descent algorithm. In the left case, the algorithm successfully finds a local minimum, but we cannot be sure that this is the global minimum (in fact a lower minimum exists at $x = 11$). In the right case, the step coefficient is too high, and the algorithm bounces around the local valley without finding the minimum there.

4.1. Final-layer nodes

For a particular final-layer node k, and an arbitrary weight coming from a node p in a previous layer, we need to find

$$ {{\partial E} \over {\partial w_{pk}}} = {{\partial} \over {\partial w_{pk}}} {1 \over 2} \sum_{k \in K}(O_{k} - t_{k})^2 .$$

Solving, we arrive at the equation

$$ {{\partial E} \over {\partial w_{pk}}} = (O_{k} - t_{k})O_{k}(1 - O_{k})O_p .$$

Since we will be using this formula later, we will make it more compact, by putting all of the terms involving the current node k into a single variable d_k, as in

$${{\partial E} \over {\partial w_{pk}}} = d_{k} O_p \text{, where} \label{dEdwpk} $$

$$d_{k} = (O_{k} - t_{k})O_{k}(1 - O_{k}). \label{dk}$$

Separating "current node" (k) information from "previous node" (p) information will be useful when we implement this algorithm on a computer. The letter d is chosen for d_k because this value essentially captures the delta by which we will adjust the parameter (scaled by O_p in this case).

Computing the derivative of E with respect to a final-node's bias b_{k} requires a very similar process, and the result is similar too:

$$ {{\partial E} \over {\partial b_{k}}} = d_{k} \label{dEdbk}.$$

With equations $\eqref{dEdwpk}$, $\eqref{dk}$, and $\eqref{dEdbk}$, we can calculate the directions in which adjust the weights and biases of any node in the final layer, toward minimizing E.

4.2. Hidden-layer nodes

For any node not in the final layer, we do not know from the outset what that node's target should be. Because the targets are unknown, or "hidden," we call the nodes on these layers "hidden nodes." Since targets are involved in the equation of E, finding the partial derivatives of E for the parameters of hidden nodes is considerably more difficult than it is for final-layer nodes.

To find the partial derivative of E with respect to a weight on a hidden layer, we need to solve

$$ {{\partial E} \over {\partial w_{pq}}} = {{\partial} \over {\partial w_{pq}}} {1 \over 2} \sum_{k \in K}(O_{k} - t_{k})^2 ,$$

where q is not in K. Since our network is fully-connected, the parameters of q are guaranteed to affect every node of the final layer. Therefore, in order to compute this derivative, we will need to keep all terms involving k as variables. Solving, we find that

$$ {{\partial E} \over {\partial w_{pq}}} = O_q(1-O_q)O_p \sum_{k \in K} d_{k} w_{qk}. $$

As before, since this equation will be used heavily later, we will shorten it by putting all terms involving the current node q into a single variable d_q, as in

$${{\partial E} \over {\partial w_{pq}}} = d_q O_p \text{, where} \label{dEdwpq}$$

$$d_{q} = O_q(1-O_q)\sum_{k \in K} d_{k} w_{qk}. \label{dq}$$

We also need to the partial derivative E with respect to any bias on a hidden layer. The formula is

$${{\partial E} \over {\partial b_q}} = d_q . \label{dEdbq}$$

With equations $\eqref{dEdwpq}$, $\eqref{dq}$, and $\eqref{dEdbq}$, we can calculate the directions in which adjust the weights and biases of any hidden-layer node, toward minimizing E.

4.3. Multiple hidden layers

In the calculations for the hidden-layer partial derivatives, we assumed that the final layer of the network resided directly above the hidden layer. If we have more than one hidden layer, then this will not always be the case. However, the equations worked out above will still apply if we make the following generalization: replace d_k in $\eqref{dq}$ with d_κ, and let

$$d_{\kappa} = \begin{cases} d_k, & \text{if the following layer is the final layer, or} \\[2ex] d_q^{l+1}, & \text{if the following layer is another hidden layer,} \\ \end{cases} \label{dkappa}$$

where d_q^{l+1} refers to d_q evaluated at the following layer. Then, equation $\eqref{dq}$ becomes

$$d_{q} = O_q(1-O_q)\sum_{\kappa \in K} d_{\kappa} w_{q\kappa}. \label{dq_generalized}$$

This reformulation of d_q fits into equation $\eqref{dEdwpq}$ with no further changes necessary. To understand why this formulation is correct, it will help to establish a good interpretation of what these partial derivatives represent.

4.4. Interpretation

Let us focus for a moment on equation $\eqref{dEdwpk}$, $ {{\partial E} \over {\partial w_{pk}}} = (O_{k} - t_{k})O_{k}(1 - O_{k})O_p $. This equation says that the partial derivative of E with respect to a weight w_{pk}, which connects node p in the previous layer to node k in the final layer, equals the product of three terms: $O_k - t_k$, which is the error of the node's output; $O_k(1-O_k)$, which is the derivative of the node's function σ; and O_p, which is the data that travelled across the edge weighted by w_{pk} toward the current node. In other words,

$$ {{\partial E} \over {\partial w_{pk}}} = (\text{derivative of } \sigma(x_k))*(\text{error at } w_{pk} \text{'s destination})*(\text{data passed through } w_{pk}). \label{dEdwpk_words}$$

We can interpret the first term as providing the sign for the partial derivative we are computing. The second two terms contribute magnitude to the derivative. If w_{pk} caused a large error at the final layer, then the partial derivative of E with respect to that parameter will be large. The derivative will be especially large if the data passing through w_{pk} is also large. This sounds reasonable.

We can use the same interpretation strategy on equation $\eqref{dEdwpq}$, $ {{\partial E} \over {\partial w_{pq}}} = O_q(1-O_q)O_p \sum_{k \in K} d_{k} w_{qk} $. The first part of the equation has some familiar terms, so we can translate them directly:

$$ {{\partial E} \over {\partial w_{pq}}} = (\text{derivative of } \sigma(x_q))*(\text{data passed through }w_{pq})*\sum_{k \in K} d_{k} w_{qk} .$$

Considering that $d_k = O_k(1-O_k)(O_k-t_k)$, the summation term computes, for each k in K, $\text{derivative of } \sigma(x_k)$ times $\text{error at } w_{qk} \text{'s destination}$ times w_{qk}, and sums these products together over K. Therefore, this step collects a sum of errors at the final layer, weighted by the connections that lead from the current node (q) to the nodes in the final layer ($k \in K$). In other words, this step collects all of the error at the final layer that w_{pq} is connected to:

$$ {{\partial E} \over {\partial w_{pq}}} = (\text{derivative of } \sigma(x_q))*(\text{data passed through }w_{pq})*(\text{the total error connected to } w_{pq} \text{'s destination}). \label{dEdwpq_words}$$

This is almost an exact match of equation $\eqref{dEdwpk_words}$.

Finally, we may interpret a generalized version of equation $\eqref{dEdwpq}$, using d_κ from $\eqref{dkappa}$ to accomodate the case of multiple hidden layers: ${{\partial E} \over {\partial w_{pq}}} = O_q(1-O_q)O_p \sum_{\kappa \in K} d_{\kappa} w_{q\kappa}. \nonumber$ All of the reasoning from earlier applies here as well, but with the added complication that for every layer between the current layer and the final layer, we must recursively unpack d_κ into $d_q^{l+1} = O_q^{l+1}(1-O_q^{l+1})\sum_{\kappa \in K} d_{\kappa} w_{q\kappa}^{l+1}$, as per equations \eqref{dkappa} and \eqref{dq_generalized}. Once again, we need to recognize some familiar terms: $O_q^{l+1}(1-O_q^{l+1})$ is simply a derivative, which provides a sign, and $\sum_{\kappa \in K} d_{\kappa} w_{q\kappa}^{l+1}$ computes $\text{the total error connected to }$ $w_{q\kappa}^{l+1} \text{'s destination}$. Following this product recursively up the network simply maintains our original formula:

$$ {{\partial E} \over {\partial w_{pq}}} = (\text{derivative of } \sigma(x_q))*(\text{data passed through }w_{pq})*(\text{the total error connected to } w_{pq} \text{'s destination}). \nonumber \tag{\ref{dEdwpq_words}}$$

This interpretation of a neural network's partial derivatives is depicted in Figure 7. Interpreting the derivatives in this way will be useful when coding the network. It will also simplify the derivation of derivatives for more complicated network types, such as convolutional neural networks.

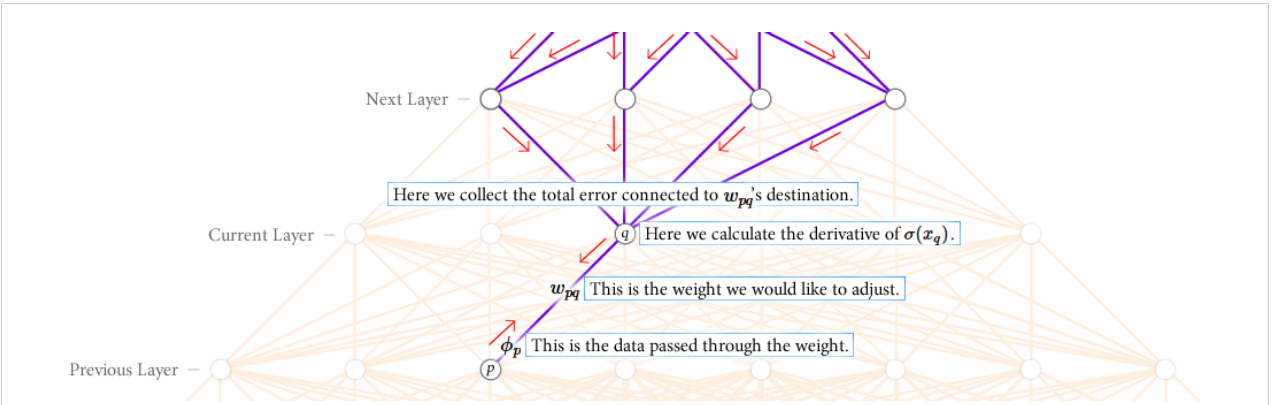

Figure 7. Partial derivatives in the ANN graph. Sources of information relevant to the calculation of the partial derivative of E with respect to a weight w_{pq} are highlighted. Arrows indicate the direction in which the information must travel for this computation to take place; all arrows lead to w_{pq}, where the final calculation takes place. Note that every value that w_{pq} affects contributes to the derivative, as should be the case.

5. Backpropagation

The backpropagation algorithm is essentially a dynamic programming algorithm for finding all of the partial derivatives in an ANN. The algorithm operates on the fact that the derivatives on one layer contain partial solutions to the derivatives on the layer below. Specifically, once we calculate

$${{\partial E} \over {\partial w_{pk}}} = d_{k} O_p, \nonumber \tag{\ref{dEdwpk}} $$

then we will be able to re-use d_k for calculating

$${{\partial E} \over {\partial w_{pq}}} = d_q O_p \text{, where} \nonumber \tag{\ref{dEdwpq}}$$

$$d_{q} = O_q(1-O_q)\sum_{k \in K} d_{k} w_{qk}, \nonumber \tag{\ref{dq}}$$

in the layer below. Similarly, we will be able to re-use d_q when calculating the derivatives for the next layer below. In this way, we can travel from the top layer to the bottom layer, computing derivatives for the parameters of the network one layer at a time, and assembling partial solutions to the lower layers' derivatives as we go. The algorithm's name comes from how it propagates partial solutions backward through the network (from the final layer to the first layer).

Before starting the algorithm, initialize all weights to small random numbers, and initialize all biases to 0. Then, follow this loop:

1. Do a complete forward pass through the network. That is, input the image to the bottom layer, and compute the outputs of every node in the network, one layer at a time.
2. For each final-layer node k, calculate the delta value d_{k}, and update the weights and bias at that node, according to the following rules:
   - $w_{pk} = w_{pk} - {{\partial E} \over {\partial w_{pk}}} = w_{pk} - d_{k} O_p$
   - $b_{k} = b_{k} - {{\partial E} \over {\partial b_{k}}} = b_{k} - d_{k}$

   Notice that we subtract the gradient (rather than add it), because the gradient points in the direction of maximum increase, and we want to descend the error surface.
3. For each hidden layer (iterating from top to bottom), for each node q in that layer, calculate d_q, and update the weights and bias, according to the following rules:
   - $w_{pq} = w_{pq} - {{\partial E} \over {\partial w_{pq}}} = w_{pq} - d_{q} O_p$
   - $b_{q} = b_{q} - {{\partial E} \over {\partial b_{q}}} = b_{q} - d_{q}$
4. Loop to 1, until some stopping criteria is reached.

This loop will accomplish gradient descent on the error function, optimizing every parameter of our ANN.

In the next several sections, we will implement the backpropagation algorithm, and demonstrate how ANNs work, with the use of MATLAB code and interactive examples.

6. First Implementation: A Single-Node "Network"

In this section, we will implement our first neural "network", which will contain a single node. We will begin by giving this node a single incoming edge, and a single fixed input. This type of network is shown in Figure 8. A single node can also have several inputs across a single edge, and have multiple incoming edges. We will add this functionality incrementally.

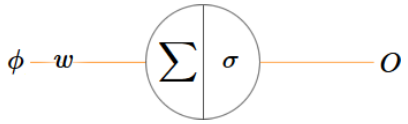

Figure 8. A single node with one input. The input ϕ is multiplied by a weight w, and that product is passed through the neuron's inner function σ, producing an output O. In this simple case, summing the weighted inputs is redundant.

6.1. Optimizing one connection for one input

Recall that the output of a node with one input is defined by

$$ O = \sigma(\phi w + b), \nonumber \tag{\ref{singleInput_neuron}}$$

where ϕ is the input, w is a "weight" term, and b is a "bias". In the ANN graph, w and b represent the parameters of the connection between the input ϕ and the node's inner function. We can optimize these parameters to solve the problem "Given an input of -0.23, produce the output 0.74." This can be phrased algebraically, as

$$ \sigma((-0.23)w + b) = 0.74. \label{in23out74}$$

Since there are two unknowns, w and b, and only one equation, there are infinite possible solutions. Two extreme solutions are shown in Figure 9. As we will see, solutions found by gradient descent are characteristically different from these examples.

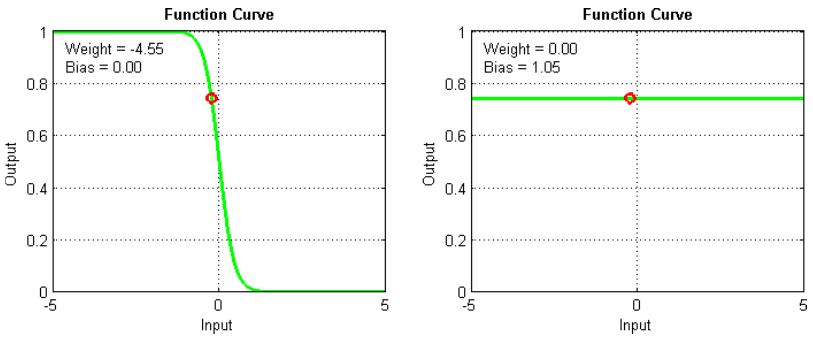

Figure 9. Two possible solution to the example problem. If $b = 0$, then $w = -4.54769$ solves the equation, as shown in the left plot. If $w = 0$, then $b = 1.04597$ solves the equation, as shown in the right plot.

The ANN will find a solution by descending the error surface E, defined by

$$ E = {1 \over 2}(O - 0.74)^2, $$

where O is the output of the node. Note that this is the same as the original error function $\eqref{error_equation}$, except the sum over $k \in K$ is removed (since there is only one node), and t is replaced with an actual target value.

Coding this in MATLAB is fairly straightforward. First, we define the inner function, initialize the network's parameters, and specify a maximum number of iterations for gradient descent. Typically, one initializes the weight to a small random number, and the bias to 0. For reproducibility, we will initialize the weight to 1 in this

example.

```
%% Setup
sigma=@(x) 1./(1+exp(-x));
input = -0.23;
target = 0.74;
weight = 1;
bias = 0;
maxIterations = 100;
```

We can test the network immediately.

```
%% First test
node_output = sigma(weight * input + bias);
fprintf('Input = %.2f. Target = %.2f\n', input, target);
fprintf('Before training, w = %.4f, b = %.4f, output = %.4f\n', ...
    weight, bias, node_output);
```

The next step is gradient descent. Gradient descent takes place in a loop, which is split into two parts. First, there is a forward pass, in which the input is run through the network, and an error is calculated at the end. Second, there is a backward pass, in which derivatives are calculated for each parameter, and the parameters are adjusted. The code here follows from the mathematics worked out earlier.

```
%% Training
for iter=1:maxIterations
    % Forward pass
    node_output = sigma(weight * input + bias);
    error = 0.5*((node_output - target)^2);

    % Progress check!
    if mod(iter,10)==0
        fprintf('Iteration = %d: output = %.4f\n', iter, node_output);
    end

    % Backward pass.
    % Create d_k
    derivative = node_output*(1-node_output);
    d_k = (derivative).*(node_output-target);
    % Apply it!
    weight = weight + (-1)*d_k*input;
    bias = bias + (-1)*d_k;
end
fprintf('Done training!\n');
```

After some number of iterations, the output of the network should be close to the target output. We can check this by testing the network after training, and printing the result to the console.

```
%% Final test
node_output = sigma(weight * input + bias);
fprintf('After training, w = %.4f, b = %.4f, output = %.4f\n', ...
    weight, bias, node_output);
```

Given the initialization $w = 1$ and $b = 0$, the algorithm arrives at $w = 0.7241, b = 1.1996$, producing the output $O = 0.7375$. This is very close to the target output.

The progress of gradient descent can be illustrated effectively by plotting the function curve, and the position of the network on the error surface, at every iteration. These plots are shown in Figure 10.

Figure 10 (interactive). Gradient descent on a one-node network with a single input/target pair. Hover your cursor over either of the plots to watch the progress of the gradient descent algorithm. The left plot shows the curve of the network, and the target (in red). Simultaneously, the right plot shows the position of the network on its error surface. The weight was initialized to 1, and the bias was initialized to 0.

There are several important things to observe in Figure 10.

- First, note that the left graph presents the problem of fitting a curve to a single point. As deduced earlier, this type of problem has infinite solutions.
- Second, note that the error surface has a large low-error area, and the entire surface slopes into it. This suggests that the gradient descent algorithm will easily find the minimum.
- Third, observe that the error surface looks like a plane. This suggests that only one of the two parameters has a strong effect on error. In this case, it appears that it is easier to minimize error by adjusting the bias than it is to do so by adjusting the weight.
- Finally, note that the algorithm found a solution close to the initialization, which resulted in a curve similar in shape to the canonical logistic sigmoid curve. This is good, because unlike the curve of the first plot in Figure 9, this curve will produce outputs near the target value even if the input is perturbed slightly. Furthermore, unlike the second curve in Figure 9, this curve will produce different outputs if the inputs are widely different. This is desirable behaviour, because we want the ANN to be robust to noise, but we also want it to be discriminative.

Our observations on the algorithm and solution space for this simple case will help build intuition about the types of problems ANNs can solve, and the types of solutions ANNs tend to find.

6.2. Optimizing one connection for many inputs

In practice, it is usually the case that an ANN must optimize their output to multiple inputs. For digit classification, for example, a single network will be used for all of the digits, so the weights that produce $\begin{bmatrix}1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\end{bmatrix}$ for the image of a hand-drawn 0 must also produce $\begin{bmatrix}0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$ for the image of a hand-drawn 9. We can try a similar problem with a single-node network.

We will train our single-node ANN to solve the problem posed by the following system of equations:

$$ \left\{ \begin{aligned} \sigma((-3.4)w + b) & = 0.7\\ \sigma((0.9)w + b) &= 0.6 \\ \end{aligned} \right\}. $$

There are two unknowns, and two equations, meaning there is a single solution. Computing the solution directly, we find that $w \approx 0.102752, b \approx -0.497942$ solves the system. We will see if gradient descent finds these values also.

In our MATLAB code, there is not much to change. This time, the inputs and targets are vectors.

```
%% Setup
sigma=@(x) 1./(1+exp(-x));
inputs = [-3.4, 0.9];
targets = [0.7, 0.6, ];
weight = randn(1);
bias = 0;
maxIterations = 200;
```

As we test the network, we must now iterate through the input/target pairs. In the code, we use the term "example" to indicate the pair number, because this essentially represents a "training example". Once the network learns to produce the target outputs for the given examples, it will hopefully be able to produce desirable output for new examples.

```
%% First test
fprintf('Before training, w = %.4f, b = %.4f\n', ...
    weight, bias);
for example=1:length(inputs)
    node_output = sigma(weight * inputs(example) + bias);
    fprintf('For input = %.1f, output = %.2f (target=%.2f)\n',...
        inputs(example), node_output, targets(example));
end
```

We can only train the network for one input/output pair at a time, so we use a random number to choose what to train on. Alternatively, we could iterate through the pairs in a loop.

```
%% Training
for iter=1:maxIterations
    % Randomly choose an input/target
    example = randi(length(inputs));

    % Forward pass
    node_output = sigma(weight * inputs(example) + bias);
    error = 0.5*((node_output - targets(example))^2);

    % Progress check!
    if mod(iter,10)==0
        fprintf('Iteration = %d: output %d = %.4f (target = %.4f)\n',...
            iter, example, node_output, targets(example));
    end

    % Backward pass.
    % Create d_k
    derivative = node_output*(1-node_output);
    d_k = (derivative).*(node_output-targets(example));
    % Apply it!
    weight = weight + (-1)*d_k*inputs(example);
    bias = bias + (-1)*d_k;
end
fprintf('Done training!\n');
```

The network may not find the exact solution to the training data, but it should come close.

```
%% Final test
fprintf('After training, w = %.4f, b = %.4f\n', ...
    weight, bias);
for example=1:length(inputs)
    node_output = sigma(weight * inputs(example) + bias);
    fprintf('For input = %.1f, output = %.2f (target=%.2f)\n',...
        inputs(example), node_output, targets(example));
end
```

Given initialization $w = 1$ and $b = 0$, the algorithm finds $w = -0.1033$, and $b = 0.4933$. This is very close to the computed solution, and produces the desired output: for input -3.4, output $O = 0.7$, and for input 0.6, output $O = 0.9$.

Figure 11 shows the function curve, and the position of the network on the error surface, at every iteration. Studying this figure will help us understand the solution the algorithm arrived at.

Figure 11 (interactive). Gradient descent on a one-node network, with two input/target pairs. Hover your cursor over either of the plots to watch the progress of the gradient descent algorithm. The left plot shows the curve of the network, and the two targets (in red). Simultaneously, the right plot shows the position of the network on its error surface.

We can make several observations in Figure 11.

- First, note that the left graph presents the problem of fitting a curve to two points. As deduced earlier, this type of problem has only one solution.
- Second, note that the error surface is more complicated than it was with a single target. Clearly, finding a good solution to the problem will require optimizing both w and b to the input.
- Third, note that the error surface is convex (bowl-shaped). This indicates that there is a single solution to the problem (as we already know), and also that gradient descent should be able find it.

These observations will be true for almost any one-node, two-target problem.

Using the same code, we can also solve more difficult problems. For example, we can task the ANN with solving the problem posed by the following system of equations:

$$ \left\{ \begin{aligned} \sigma((-0.9)w + b) &= 0.7 \\ \sigma((0)w + b) &= 0.9 \\ \sigma((3.4)w + b) &= 0.6 \\ \sigma((4)w + b) &= 0.3 \end{aligned} \right\}. $$

There are four unknowns, and two equations, meaning there may be a single solution, or there may be no solutions. We will find out with gradient descent.

In our MATLAB code, the only thing to change is the vector of inputs, and the vector of targets.

```
inputs = [-0.9, 0, 3.4, 4];
targets = [0.7, 0.9, 0.6, 0.3];
```

With no further changes, the algorithm can iterate toward a solution, arriving at $w = -0.2365, b = 1.1449$. This solution is plotted in Figure 12.

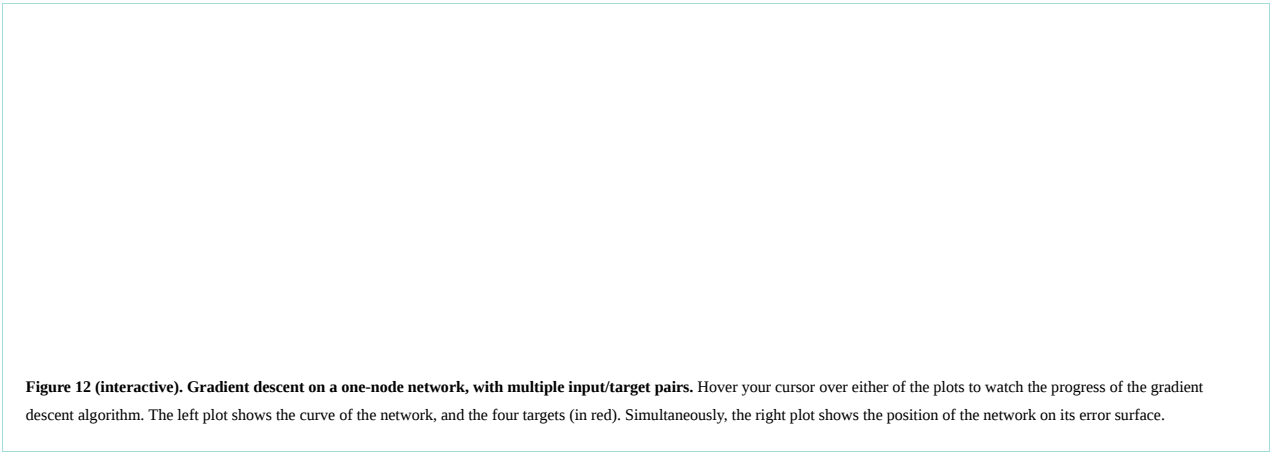

Figure 12 (interactive). Gradient descent on a one-node network, with multiple input/target pairs. Hover your cursor over either of the plots to watch the progress of the gradient descent algorithm. The left plot shows the curve of the network, and the four targets (in red). Simultaneously, the right plot shows the position of the network on its error surface.

We can make several observations in Figure 12.

- First, note that the four input/target pairs do not fit a sigmoid line, so there is no perfect solution to the problem. At best, we can expect gradient descent to find a curve somewhere in between all of the points.
- Second, note that the error surface appears to have a single deep valley, so gradient descent should be able to descend into it.
- Finally, when watching the function curve move toward the targets, note that its movement becomes erratic as it nears the targets. This is an example of the "overstepping" problem, where the algorithm misses a good solution because its step size is too large. We can address this by introducing a step coefficient. In the context of ANNs, this coefficient is often called a "learning rate". Typically, the coefficient is set up to decrease over time, so that the algorithm takes smaller and smaller steps as it nears the final solution. Our next implementation will include this coefficient.

Visualizing the progress of an algorithm can sometimes reveal weaknesses in the approach, prompting ideas for improvements. However, it will not be so easy to make observations and deductions with more complicated networks. In this case, the ANN has only two parameters (w and b), and so the error space is two-dimensional, meaning it can be shown as a surface. For an ANN with several thousand parameters, the error space will have several thousand dimensions, so simple visualizations will be impossible. Nonetheless, intuitions built up from simple cases will still be useful for more complicated cases.

6.3. Optimizing many connections for one set of inputs

The nodes of an ANN typically have many incoming connections, representing many channels of input. For each connection, the node has a unique weight. A single-node network of this type is displayed in Figure 13.

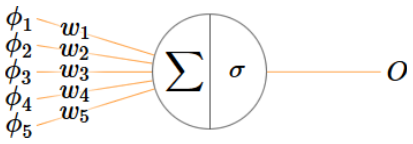

Figure 13. A single node with five inputs. The inputs are multiplied by their weights, and the sum of those products is passed through σ, producing an output O.

This graph corresponds to the equation

$$ O = \sigma(\phi_1 w_1 + \ldots + \phi_n w_n + b), \nonumber \tag{\ref{finalSigma}}$$

where n equals 5. The more weights a network has, the greater its representational power. With five weights instead of one, we should now be able to solve larger problems than before. We will begin by solving

$$ \sigma((-0.5)w_1 + (0.4)w_2 + (0.1)w_3 + (0.4)w_4 + (0.5)w_5 + b) = 0.5 . $$

There are six unknowns, and only one equation, so there are infinite solutions. Similar to the problem in Section 6.1 for a single-input node, there is one input per connection, but this time there are several connections going into the node simultaneously.

To implement this type of ANN in MATLAB, we need to make a few changes to our code. In this problem we need to initialize a vector of inputs, a vector of weights, and a single target. For this example, we randomly generate the weights. Also, we specify a learning rate schedule, as prompted by the observations on Figure 12.

```
%% Setup
sigma=@(x) 1./(1+exp(-x));
inputs = [-0.5, 0.4, 0.1, 0.4, 0.5];
target = 0.5;
nInputs = length(inputs);
weights = randn(1,nInputs);
bias = 0;
maxIterations = 100;
learningRateSchedule = [0.3 0.2 0.1];
```

This establishes the network's architechture, so we can test it immediately.

```
%% First test
fprintf('Before training...\n');
node_output = sigma(weights * inputs' + bias);
for i=1:nInputs
    fprintf('Weight %d = %.2f\n', i, weights(i));
    fprintf('Input %d = %.2f\n', i, inputs(i));
end
fprintf('Node output = %.2f (target = %.2f)\n', ...
    node_output, target);
```

To train the network, we first retrieve a learning rate from the schedule, and then carry out the standard computations. One task here is multiply the weights by the inputs pointwise, and sum them together. This is just a dot product. Since in our code `weights` and `inputs` are both 1×5 vectors, we multiply `weights` by the transpose of `inputs` to accomplish this in MATLAB.

```
%% Training
for iter=1:maxIterations
    % Get the learning rate for this iteration from the schedule
    if iter < 60
        learningRate = learningRateSchedule(1);
    elseif iter < 80
        learningRate = learningRateSchedule(2);
    else
        learningRate = learningRateSchedule(3);
    end

    % Forward pass
    node_output = sigma(weights * inputs' + bias);
    error = 0.5*((node_output - target)^2);

    % Progress check!
    if mod(iter,10)==0
        fprintf('Iteration = %d, learningRate = %.2f, output = %.4f (target = %.4f)\n', ...
            iter, learningRate, node_output, target);
    end

    % Backward pass.
    % Create d_k
    derivative = node_output.*(1-node_output);
    d_k = (derivative).*(node_output-target);
    % Apply it!
    weights = weights + learningRate*(-1)*d_k.*inputs;
    bias = bias + learningRate*(-1)*d_k;
end
fprintf('Done training!\n');
```

The solution we arrive at depends greatly on the initialization, but with enough iterations the algorithm should find a good solution.

```
%% Final test
fprintf('After training...\n');
node_output = sigma(weights * inputs' + bias);
for i=1:nInputs
    fprintf('Weight %d = %.2f\n', i, weights(i));
    fprintf('Input %d = %.2f\n', i, inputs(i));
end
fprintf('Node output = %.2f (target = %.2f)\n', ...
    node_output, target);
```

Unfortunately, now that our ANN has more than two parameters, it is impossible to effectively plot the solution space of the problem. However, it is still informative to

look at the final output, and the corresponding error. With a fixed input, error should decrease on every iteration.

6.4. Optimizing many connections for many sets of inputs

Our last single-node network will optimize a single set of weights for many sets of inputs. This problem is comparable to the one in Section 6.2: the network is "overloaded" with input, and the goal is to find a solution that satisfies all input/target pairs.

We will attempt to solve the system

$$ \left\{ \begin{aligned} \sigma((-0.5)w_1 + (0.4)w_2 + (0.1)w_3 + (0.4)w_4 + (0.5)w_5 + b) &= 0.8\\ \sigma((-0.4)w_1 + (0.3)w_2 + (0.2)w_3 + (0.1)w_4 + (0.3)w_5 + b) &= 0.6\\ \sigma((0.1)w_1 + (-0.1)w_2 + (0.3)w_3 + (-0.1)w_4 + (0.9)w_5 + b) &= 0.1\\ \sigma((-0.1)w_1 + (0.4)w_2 + (0.6)w_3 + (0.3)w_4 + (-0.1)w_5 + b) &= 0.2\\ \end{aligned} \right\}. $$

With six unknowns and four equations, there are infinite solutions. With enough iterations, gradient descent should be able to find a good one.

In MATLAB, we will again be able to reuse most of our earlier code. This time, we have a vector of inputs for each target, so we will the inputs as a matrix, and we will once again have a vector of targets.

```
%% Setup
sigma=@(x) 1./(1+exp(-x));
inputs = [-0.5, 0.4, 0.1, 0.4, 0.5;
          -0.4, 0.3, 0.2, 0.1, 0.3;
          0.1, -0.1, 0.3, -0.1, -0.9;
          -0.1, 0.4, 0.6, 0.3, -0.1];
targets = [0.7; 0.6; 0.1; 0.3];
nInputs = size(inputs,2);
nExamples = size(inputs,1);
weights = randn(1,nInputs);
bias = 0;
maxIterations = 1000;
learningRateSchedule = [0.3 0.2 0.1];
```

The rest of the code is almost exactly the same, except we iterate through input/target pairings during testing, and randomize the pair number during training.

```
%% First test
fprintf('Before training...\n');
```

Click to expand

Since the weights are initialized randomly, the algorithm will find a different solution on each run. With a bad initialization, the algorithm may not find a good solution at all. In most cases, however, the algorithm will work well.

Nothing more can be done with a single node ANN. The next several sections will explore more complicated ANNs, in which multiple nodes are connected to one another.

7. Second Implementation: A Fully-Connected Network

In this section, we will implement our first multi-node ANN. Figure 5 (copied below) shows the architecture of the network we will design. This will be our first network to make use of "hidden" nodes, and backpropagation.

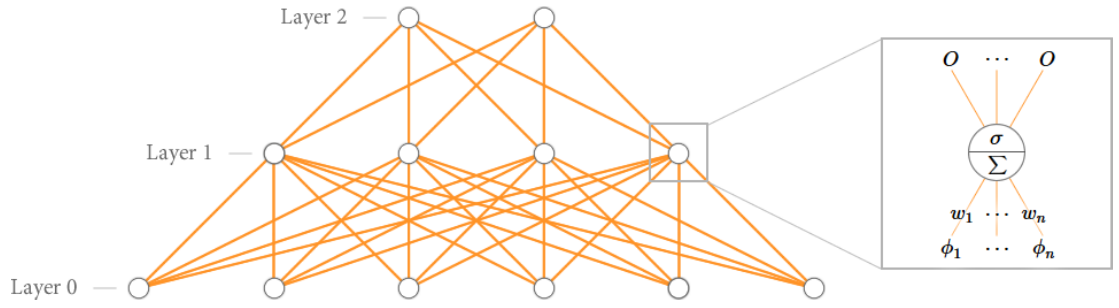

Figure 5. The structure of our artificial neural network. Nodes of neighbouring layers are fully connected. A node receives as input weighted outputs of other neurons, processes them through σ, and outputs the result.

7.1. Classifying digits from computer fonts

In this section, we will build an ANN capable of classifying the digits of various computer fonts. This is very close to our end-goal of classifying hand-drawn digits. The reason we start with computer fonts is that they have much less variability than handwriting. Less variability in the training data means less parameters are necessary, and training time can be faster. Therefore, rather than solve equation $\eqref{goal}$, we will solve

$$ f(\blacksquare 0) = \begin{bmatrix} 1\\0\\0\\0\\0\\0\\0\\0\\0\\0 \end{bmatrix}, \quad f(\blacksquare 1) = \begin{bmatrix} 0\\1\\0\\0\\0\\0\\0\\0\\0\\0 \end{bmatrix}, \quad f(\blacksquare 9) = \begin{bmatrix} 0\\0\\0\\0\\0\\0\\0\\0\\0\\1 \end{bmatrix}. \label{simplified_goal}$$

That is, given a 16×16 computer-generated image of a particular digit, f will assign a 1 that digit's corresponding element in the output vector, and a 0 to each of the other elements. Since the images input to f are actually two-dimensional arrays of unsigned integers, we can rewrite the first equality of $\eqref{simplified_goal}$ as

$$f \left (\begin{bmatrix} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ & 0 & 0 & 0 & 0 & 0 & 0 & 32 & 112 & 128 & 51 & 0 & 0 & 0 & 0 & 0 & 0 \\ & 0 & 0 & 0 & 0 & 0 & 51 & 240 & 224 & 204 & 255 & 112 & 0 & 0 & 0 & 0 & 0 \\ & 0 & 0 & 0 & 0 & 0 & 192 & 176 & 0 & 0 & 112 & 240 & 16 & 0 & 0 & 0 & 0 \\ & 0 & 0 & 0 & 0 & 0 & 16 & 255 & 64 & 0 & 0 & 0 & 240 & 80 & 0 & 0 & 0 & 0 \\ & 0 & 0 & 0 & 0 & 64 & 255 & 0 & 0 & 0 & 0 & 192 & 128 & 0 & 0 & 0 & 0 \\ & 0 & 0 & 0 & 0 & 64 & 255 & 0 & 0 & 0 & 0 & 192 & 128 & 0 & 0 & 0 & 0 \\ & 0 & 0 & 0 & 0 & 64 & 255 & 0 & 0 & 0 & 0 & 192 & 128 & 0 & 0 & 0 & 0 \\ & 0 & 0 & 0 & 0 & 32 & 255 & 16 & 0 & 0 & 0 & 204 & 112 & 0 & 0 & 0 & 0 \\ & 0 & 0 & 0 & 0 & 0 & 240 & 96 & 0 & 0 & 32 & 255 & 51 & 0 & 0 & 0 & 0 \\ & 0 & 0 & 0 & 0 & 0 & 128 & 240 & 96 & 80 & 204 & 192 & 0 & 0 & 0 & 0 & 0 \\ & 0 & 0 & 0 & 0 & 0 & 16 & 160 & 255 & 255 & 176 & 16 & 0 & 0 & 0 & 0 & 0 \\ & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \end{bmatrix} \right) = \begin{bmatrix} 1\\ 0\\ 0\\ 0\\ 0\\ 0\\ 0\\ 0\\ 0\\ 0\\ \end{bmatrix}, $$

but keep in mind that there are several versions of each digit in the dataset. We will train the parameters of the function with 80 different images of each digit, for a total of 800 equalities with which to train the parameters. As usual, we will have many more parameters than equations, meaning an infinite number of solutions are possible.

In order to classify digits accurately, the function f must be very large. As discussed earlier, the bottom layer of the network must have one node per pixel, which in this case means 256 nodes. The "hidden" section of the network, can have any number of layers, and any number of nodes per layer. We will begin with a single hidden layer, containing 20 nodes. Finally, the top layer must contain 10 nodes—one for each digit. This network is displayed in Figure 14.

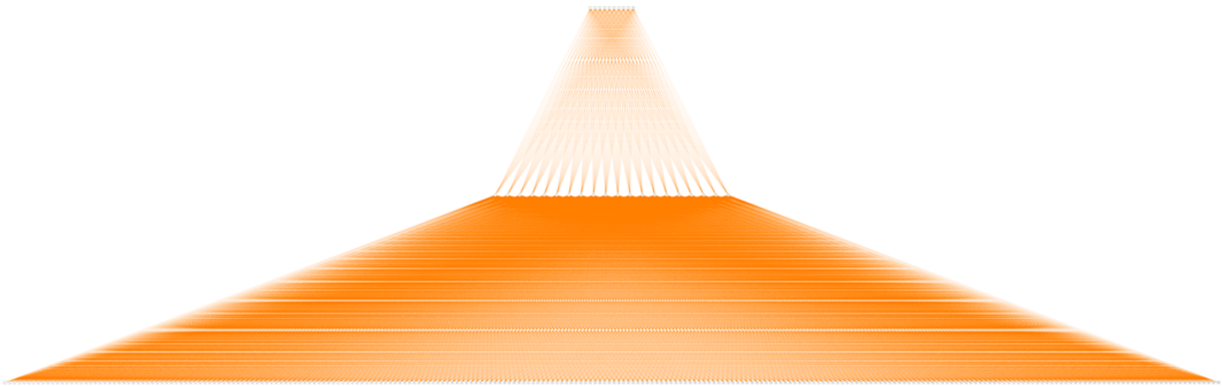

Figure 14 (interactive). A fully-connected network for classifying digits from computer fonts. Hover your cursor over the image to zoom in. This network could be described as 256-20-10, where the numbers specify the layer sizes from bottom to top. The bottom layer of the network has 256 nodes, which hold the pixels of the image. The middle (hidden) layer has 20 nodes. The top layer has 10 nodes, which hold the elements of the output vector. Every pair of neighbouring layers is fully connected.

As before, we will code this in MATLAB in a single script. Many sections of code will look similar to the code for single-node networks. First, we initialize the basic parameters of the network, including the number of hidden nodes, the number of iterations for gradient descent, and the learning rate schedule. These are variables that we may want to experiment with later, so it is convenient to put them near the top of the program.

```
%% Network parameters
nHiddenNodes = 20;
maxIterations = 800*20; % The dataset has 800 training images
learningRateSchedule = [0.5000; % Set the learning rate for each pass through
                        0.5000; % the training images.
                        0.5000;
                        0.5000;
                        0.3000;
                        0.3000;
                        0.3000;
                        0.3000;
                        0.1000;
                        0.1000;
                        0.1000;
                        0.1000;
                        0.0500;
                        0.0500;
                        0.0500;
                        0.0500;
                        0.0100;
                        0.0100;
                        0.0100;
                        0.0100];
```

Note that there is no variable called `nHiddenLayers`. It is a good idea to encode this variable implicitly in `nHiddenNodes`. When `nHiddenNodes` is a scalar (as it is for this example), it implies that `nHiddenLayers = 1`. When `nHiddenNodes` is a vector, such as `[40 20]`, the number of elements in the vector specifies the number of hidden layers, and the values of the vector specify the number of nodes on each layer. Also, note that `maxIterations` is set to twenty times the size of the training set. In ANN terminology, each loop over the training set is called an epoch. Therefore, `maxIterations` here indicates that we will train for twenty epochs. The learning rate schedule specifies the learning rate for each epoch.

Next, we load the training dataset, and set up some helper variables for the code.

```
%% Setup
sigma=@(x) 1./(1+exp(-x));
load('datasets/font_train_data.mat');
imageSize = 16;
nTargets = 10; % 10 digits
nTrainingImages = size(images,3);
trainingImage = 0;
```

Since we will be looping over the training set several times, we need to keep an index for the training image we are on. This index is stored in `trainingImage`, here initialized to `0`.

As with the single-node network, we initialize weights to random numbers, and biases to 1. The first hidden layer needs as many weights per node as there are pixels in the image (16×16). Each hidden layer after the first needs as many weights per node as there are outputs from the previous hidden layer. In the final layer, there need to be weights connecting the outputs of the last hidden layer to each final-layer node. Every node, in the entire network, only needs one bias.

```
%% Initialize variables for the fully-connected network
sizeOfInput = imageSize^2;

nHiddenLayers = size(nHiddenNodes,2);
hidden_weights = cell(nHiddenLayers,1);
hidden_biases = cell(nHiddenLayers,1);
hidden_outputs = cell(nHiddenLayers,1);

for l=1:nHiddenLayers
    if l==1
        hidden_weights{l} = randn(nHiddenNodes(l), sizeOfInput);
    else
        hidden_weights{l} = randn(nHiddenNodes(l), nHiddenNodes(l-1));
    end
    hidden_biases{l} = zeros(nHiddenNodes(l), 1);
    hidden_outputs{l} = zeros(nHiddenNodes(l), 1);
end

final_weights = randn(nTargets, nHiddenNodes(nHiddenLayers));
final_biases = zeros(nTargets, 1);
final_outputs = zeros(nTargets,1);
```

Note that the weight matrices are organized so that the number of rows corresponds to the number of nodes in the layer, and the number of columns correspond to the number of incoming connections. So, for a particular layer's `weights` matrix, `weights(2,4)` holds the weight going to the 2nd node in the current layer, from the 4th node in the previous layer. The `biases` and `outputs` matrices for each layer are similarly designed, but are only one column wide, since each node only has one bias and one output.

The next step is gradient descent. As before, this will take place in a loop, which is split into two parts. First, there is a forward pass, in which the input is run through the network, and an error is calculated at the end. Second, there is a backward pass, in which derivatives are calculated for each parameter, and the parameters are adjusted. Since hidden layers are involved, backpropagation will be used to find these derivatives. Since the code for this section is fairly complicated, we will split it up and discuss it in smaller parts.

The first section of the loop is straightforward. Update the `trainingImage` index, and load the correct learning rate, image, and target output for the iteration. When loading the image, convert it to double precision, converting its range to $[0,1]$. This will reduce the amount the sigmoid functions must transform to effectively classify the data. To do an even better job at this step, we could also center the input according to the dataset's mean and standard deviation. We will do that in the next implementation.

```
%% Training
for iter=1:maxIterations
    % Iterate
    trainingImage = trainingImage + 1;
    if trainingImage > nTrainingImages
        trainingImage = 1;
    end

    % Set the learning rate based on the epoch number
    epoch = round(iter/nTrainingImages);
    learningRate = learningRateSchedule(epoch+1);

    % Load the image
    im = im2double(images(:,:,trainingImage)); % min=0, max=1

    targets = zeros(nTargets,1);
    targets(labels(trainingImage)+1) = 1;
```

The second section of the loop is the "forward pass" through the ANN. The nodes of the first hidden layer receive the image (reshaped into a column vector) as input. At hidden layers beyond the first, each node receives the output of the entire previous layer as input. At the final layer, each node receives the output of the highest hidden layer as input. On every layer, the inputs are multiplied by their weights, and the sum of those products is added to the node's bias, and that entire sum is passed through the logistic sigmoid function. Once the outputs at the final layer are calculated, we can calculate the error of the network.

```
    %% Forward pass
    [m, n] = size(im);
    input = reshape(im, m*n, 1);

    % Hidden layer(s)
    for l=1:nHiddenLayers
        for i=1:nHiddenNodes(l)
            if l==1
                hidden_outputs{l}(i) = sigma(...
                    hidden_weights{l}(i,:)*...
                    input...
                    + hidden_biases{l}(i));
            else
                hidden_outputs{l}(i) = sigma(...
                    hidden_weights{l}(i,:)*...
                    hidden_outputs{l-1}...
                    + hidden_biases{l}(i));
            end
        end
    end

    % Final layer
    final_outputs = sigma(...
        final_weights*...
        hidden_outputs{nHiddenLayers}...
        + final_biases);

    error = 0.5*(sum((final_outputs - targets).^2));
```

The third section of the loop is the "backward pass" through the ANN. In this section, we will calculate the partial derivative of the error function with respect to every parameter in the network, and update the parameters according to these derivatives. We will calculate the derivatives layer-by-layer, backpropagating information from the end of the network to the front. An easy mistake to make at this step is to update the parameters at one layer, and then use the updated parameters to calculate the derivatives for the next layer. It is a good idea to update all of the parameters in one pass, but all of the calculations should be calculated using the original parameters. Therefore, we begin the backward pass by making copies of the parameters, and saving them with the suffix `beforeUpdate`.

```
    %% Backward pass.
```

```
    % Here we update the weights and biases for every node in the network

    % We need to use the same old weights & biases for the entire backward
    % pass, so we save them here, before starting.

    final_weights_beforeUpdate = final_weights;
    final_biases_beforeUpdate = final_biases;
    hidden_weights_beforeUpdate = hidden_weights;
    hidden_biases_beforeUpdate = hidden_biases;
```

For the final layer of the network, finding the derivatives and deltas requires almost exactly the same code as for the single-node network.

```
    % Final layer
    % Collect the d_ks
    derivatives = final_outputs.*(1-final_outputs);
    d_k = (derivatives).*(final_outputs-targets);
```

Applying the deltas is slightly more complicated. Referring to equation $\eqref{dEdwpk}$, we currently have d_k for every node k, but we need O_p for every single weight w_{pk}, which (from $\eqref{dEdwpk_words}$) we know represents $\text{data passed through } w_{pk}$. The simplest way to find this value is with a double loop that iterates through all nodes and all weights, individually finding the input to every weight. However, we can eliminate one of these loops if we realize that every final-layer node receives the same inputs: every node's nth weight receives the output of the nth node in the top hidden layer. Therefore, there are only as many O_p values as there are nodes in the top hidden layer. Iterating through the subscript p on O_p will be enough to update every final-layer node's pth weight, regardless of the number of final-layer nodes. Therefore, recalling that the `final_weights` matrix has one column for each incoming connection, we can update the values in this matrix one column at a time.

```
    % Apply them!
    for i=1:nHiddenNodes(nHiddenLayers)
        final_weights(:,i) = final_weights(:,i)+...
            learningRate*(-1)*d_k*hidden_outputs{nHiddenLayers}(i);
    end
    final_biases = final_biases + learningRate*(-1)*d_k;
```

For the hidden layers of the network, finding and applying the deltas requires a similar process. As before, finding the "basic derivative" ($O_q(1-O_q)$) for each node q, is very easy. Referring to equations $\eqref{dEdwpq}$, $\eqref{dq_generalized}$, and $\eqref{dEdwpq_words}$, the next step is to compute, for every incoming weight w_{pq}, $\text{the total error connected to } w_{pq} \text{'s destination}$. Will store this value in the variable `sumAbove`.

Computing `sumAbove` for a node requires scaling the "errors" (deltas) found above the node by the appropriate connecting weights. The backpropagation algorithm guarantees that if we are on layer l, the deltas for layer $l+1$ are already calculated. Therefore, we can find `sumAbove` by taking the dot product of the "outgoing" weights at node q and the delta at the layer above. This requires one extra trick: since every layer's weight matrix stores "incoming" weights (so that `weights(q,:)` references all weights leading to node q in the current layer), we need to make use of the following layer's weight matrix to find "outgoing" weights (so that `weights(:,q)` references all weights that come from node q in the previous layer).

Multiplying `sumAbove` with the "basic derivative" will provide d_q.

```
    % Hidden layers
    % Collect the d_qs
    d = cell(nHiddenLayers);
    for l=nHiddenLayers:-1:1
        d{l} = zeros(nHiddenNodes(l),1);
        derivatives = hidden_outputs{l}.*(1-hidden_outputs{l});
        for targetNode=1:nHiddenNodes(l)
            if l==nHiddenLayers
                sumAbove = final_weights_beforeUpdate(:,targetNode)'*d_k;
            else
                sumAbove = hidden_weights_beforeUpdate{l+1}(:,targetNode)'*d{l+1};
            end
            d{l}(targetNode) = derivatives(targetNode)*sumAbove;
        end
    end
```

Applying the deltas on the hidden layer parameters requires the same trick as we used in the final layer: taking advantage of the fact that every node's nth weight receives the same input, we can update every node's nth weight simultaneously by indexing the weight matrices one column at a time. Similar to the code in the foward pass, this step requires treating the first hidden layer differently from other hidden layers: the first hidden layer receives the image as input, whereas other layers receive the output of a previous hidden layer as input.

```
    % Apply them!
    for l=nHiddenLayers:-1:1
        for targetNode=1:nHiddenNodes(l)
            if l==1
                hidden_weights{l}(targetNode,:) = ...
                    hidden_weights{l}(targetNode,:) + ...
                    learningRate*(-1)*d{l}(targetNode)*...
                    input';
            else
                hidden_weights{l}(targetNode,:) = ...
                    hidden_weights{l}(targetNode,:)+...
                    learningRate*(-1)*d{l}(targetNode)*...
                    hidden_outputs{l-1}';
            end
            hidden_biases{l}(targetNode) = ...
                hidden_biases{l}(targetNode) + ...
                learningRate*(-1)*d{l}(targetNode);
        end
    end
end
fprintf('Done!\n');
```

This concludes the training loop. Thanks to the vectorized code, the full 16,000 training iterations will only take 20 seconds to complete.

To test the network, we simply do more forward passes. If we test on the training data, the error rate should be extremely low. Since the ANN has more variables than there are equations, and the algorithm does gradient descent, the error rate on the training data should eventually reach 0. In general, it is more interesting to see how the ANN performs on new data, since this is a measure of how well the ANN might be able to solve similar problems in practice.

```
nTestImages = 200;
top1Correct = 0;
```

Since there are only 200 testing images, this step will take less than 1 second.

After 20 epochs of training, this network typically achieves approximately 98% accuracy on the training set, and approximately 92% accuracy on the test set. Since gradient descent depends greatly on the initialization, these results are not the same on every run. However, once a network is trained, its performance on any dataset is deterministic.

To improve the accuracy of our ANN, we have two options. First, we can train the network for more epochs. That is, increase `maxIterations`, and extend the `learningRateSchedule`, and retrain the network. The only problem with this idea is that the parameters of the network may eventually fit so closely to the training data that they no longer generalize well to unseen data. This is called overfitting. If this happens, performance on the test data will actually begin to decrease. Figure 15 illustrates this phenomenon.

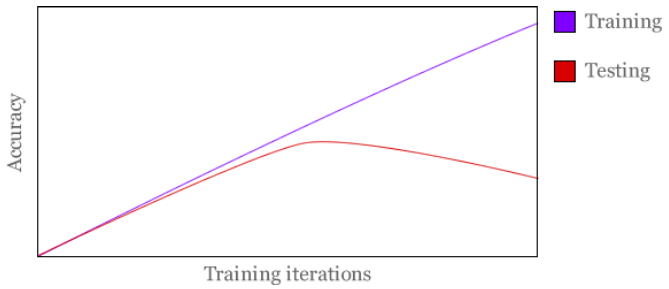

Figure 15. Overfitting. After a certain amount of training iterations, accuracy on the testing data may begin to decrease, while accuracy on the training data continues to increase.

The second option is to add more nodes to the network. In our code, this only requires changing `nHiddenNodes`: if we set the variable to a larger scalar value, we define a network with more nodes in the single hidden layer; if we set the variable to a row vector with several scalars, we define a network with multiple hidden layers. The number of elements of `nHiddenNodes` implies the number of hidden layers, and the values of those elements specify the number of nodes in each hidden layer.

```
nHiddenNodes = [40 20];
```

With double the number of nodes on the lowest hidden layer, and hence twice as many weights, this new network should have approximately twice as much representational power as the original network. However, it will also take twice as long to train. The new network is displayed in Figure 16, below.

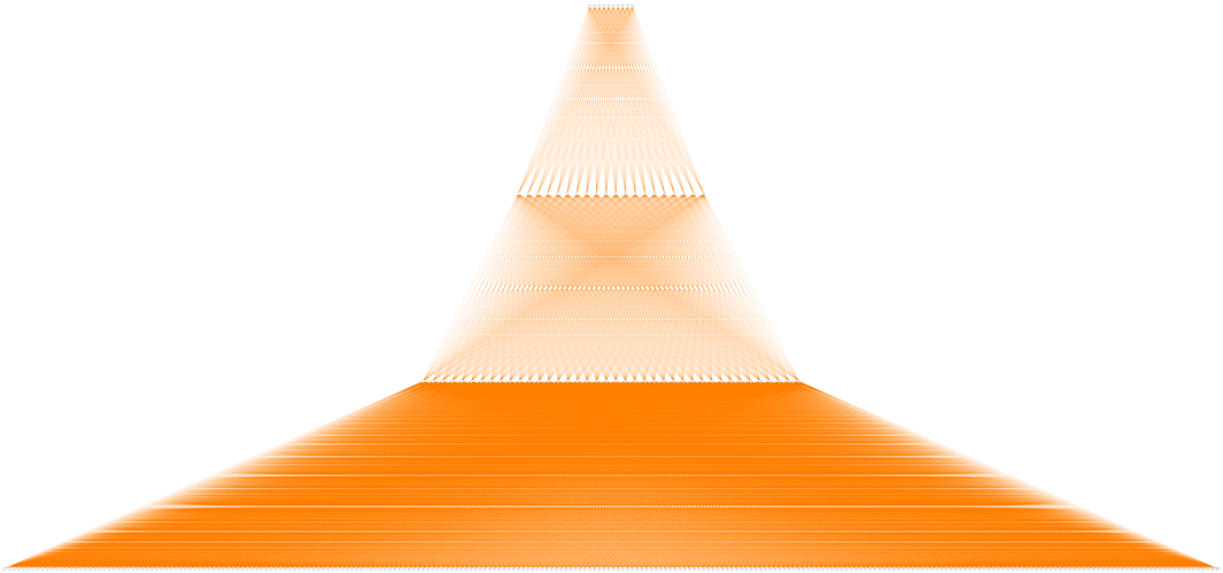

Figure 16 (interactive). A larger fully-connected network for classifying digits from computer fonts. Hover your cursor over the image to zoom in. This network could be described as 256-40-20-10, where the numbers specify the layer sizes from bottom to top. The bottom layer of the network has 256 nodes, which hold the pixels of the image. The two middle (hidden) layers have 40 nodes and 20 nodes, respectively. The top layer has 10 nodes, which hold the elements of the output vector. Every pair of neighbouring layers is fully connected.

After 20 epochs of training, this network typically achieves approximately 99% accuracy on the training set, and approximately 94% accuracy on the test set. This is not a large improvement in performance, but more extreme efforts along these lines may be worthwhile, depending on the application.

7.2. Classifying handwritten digits

With some small adjustments, the same code can be used to train a network to classify handwritten digits. That is, we can finally attempt to find our end-goal function f, which produces the behaviour

$$ f\left(\boxed{0}\right) = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad f\left(\boxed{1}\right) = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \text{and} \quad f\left(\boxed{9}\right) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}. $$

That is, we will create an ANN that takes in a 28×28 image as input, and produces an output vector of ten elements, in which the element corresponding to the correct digit has the value 1, and all other elements have the value 0.

Handwritten digits are significantly more difficult to classify than computer font digits, so the network will have to be significantly larger than our previous networks. We can copy the network setup of Lecun [1989]: two hidden layers, containing 300 nodes and 100 nodes; twenty passes through the training set; and decrease the learning rate in steps, from 0.01 on the first pass to 0.0001 on the final pass.

```
%% Network parameters
nHiddenNodes = [300 100];
maxIterations = 60000*20; % The dataset has 60000 training images
learningRateSchedule = [0.01; % Set the learning rate for each pass through
                        0.01; % the training images. This here is the
                        0.005;% schedule recommended by Lecun, Bottou,
                        0.005;% Bengio, & Haffner (1998).
                        0.005;
                        0.001;
                        0.001;
                        0.001;
                        0.0005;
                        0.0005;
                        0.0005;
                        0.0001;
                        0.0001;
                        0.0001;
                        0.00005;
                        0.00005;
                        0.00005;
                        0.00001;
                        0.00001;
                        0.00001;];
```

There are no theoretical guidelines for deciding on these parameters. The network defined by this new architecture is shown in Figure 17.

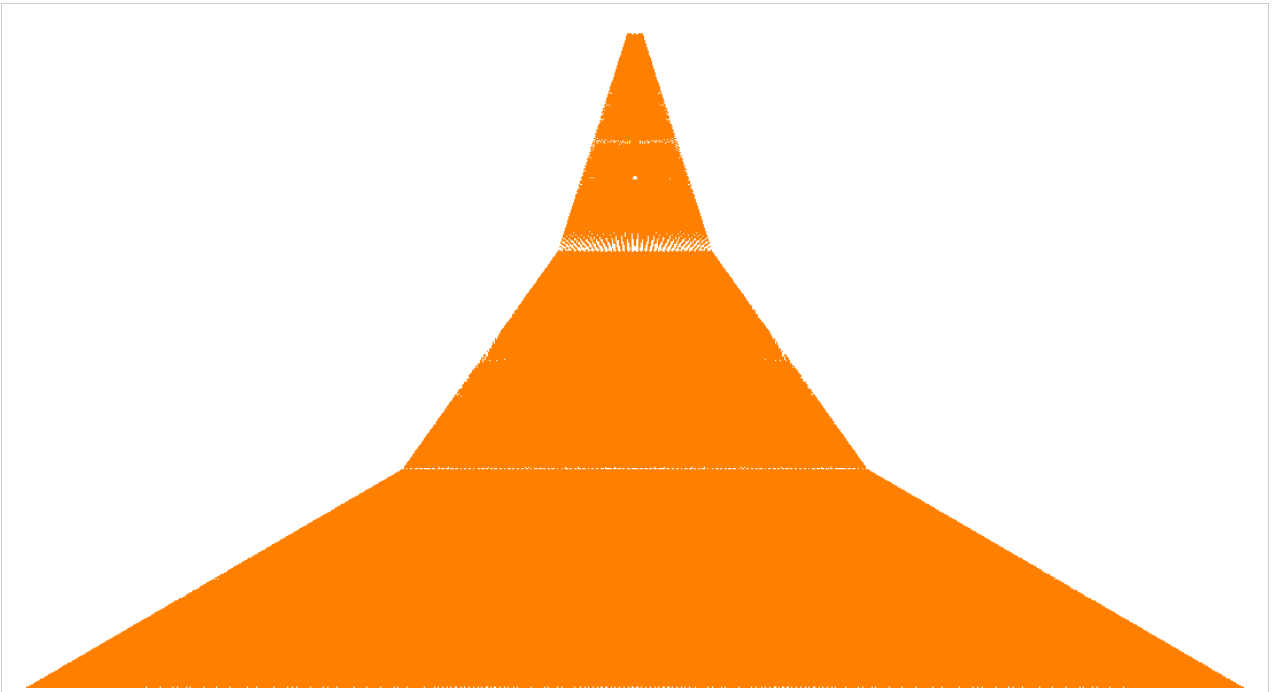

Figure 17 (interactive). A fully-connected network for classifying handwritten digits. Hover your cursor over the image to zoom in. This network could be described as 784-300-100-10, where the numbers specify the layer sizes from bottom to top. The bottom layer of the network has 784 nodes, which hold the pixels of the image. The two middle (hidden) layers have 300 nodes and 100 nodes, respectively. The top layer has 10 nodes, which hold the elements of the output vector. Every pair of neighbouring layers is fully connected. Note that the connections in this network are so dense that the space between the layers appears to be filled solid.

The next and final change to the code is in loading the data. There are only two lines to change.

```
load('datasets/MNIST_train_data.mat');
imageSize = 28;
```

After approximately 8 hours of training, the network achieves an accuracy of 96% on the test data. If we increase the number of nodes to 500 in the first hidden layer and 150 in the second hidden layer, the network reaches 97% accuracy. This is impressive, but we would like to reach 99%. For this to be possible, we will have to fundamentally change the architecture of the network.

7.3. Analysis

To better understand what fully-connected ANNs learn, it helps to visualize the learned parameters. Our earlier visualizations of ANNs displayed the weights of the network uniformily, illustrating the architecture of the network. Here, we will focus our attention on the weights. Figure 18 shows a network trained on computer font digits, using the architecture from Figure 16. The inputs and outputs of each layer are bordered red; these values will change from iteration to iteration. The parameters of each layer are bordered blue; these are the learned parameters, which stay fixed after training.

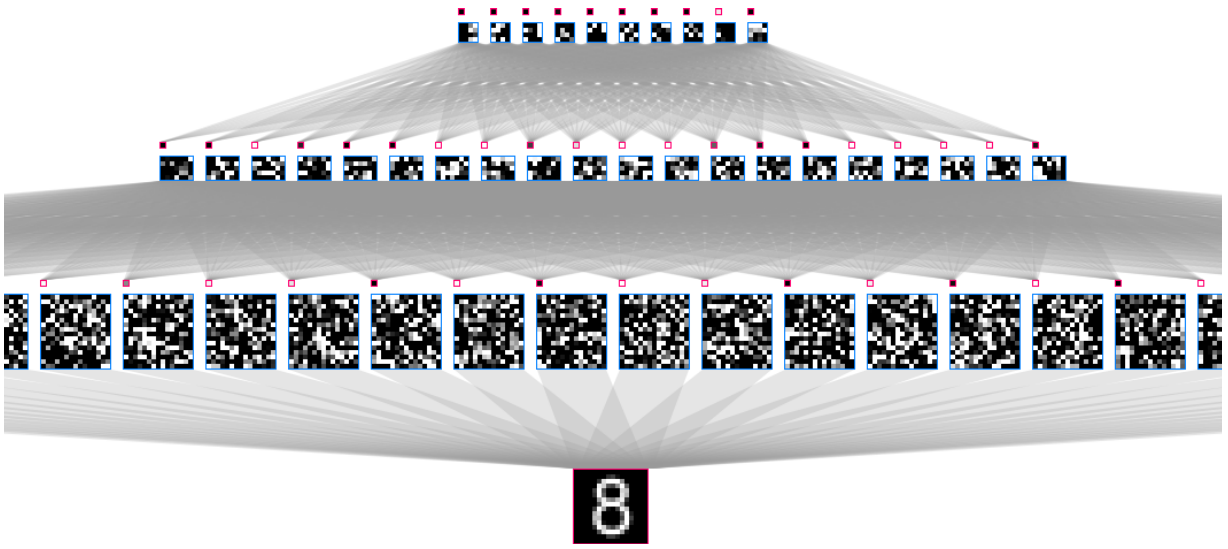

Figure 18 (interactive). An alternate visualization of a fully-connected network. Hover your cursor over the image to pan the viewport. This figure depicts the input, the weights of each layer, and the outputs of each layer, all as images. The images are shown with coloured borders, so that white pixels can be differentiated from the background. The pixel intensities of the images correspond to actual values of a trained network, so that bright pixels indicate large values and dark pixels indicate low values. Overlapping quadrangles between layers depict the transfer of information from layer to layer. Note that the outputs at the top layer of the network are as expected: the second last output (corresponding to the digit 8) is bright, and the other outputs are dark.

8. Convolutional Neural Networks

- show the features the first hidden layer is extracting, and explain them in words, if possible
- we know that pixels are locally correlated
- if we know something is true then we should take the burden off the classifier to figure it out

% Lecun advises to make the background pixels equal to -0.1, and the
% foreground pixels equal to 1.175. This will make the mean roughly 0,
% and the variance roughly 1. (Although in my own testing, the variance
% is lower

% A matrix representation for weights in a layer makes a lot of sense,
% because we are interested sometimes in all connections incoming to a
% weight (i.e. the elements of a row), and also interested in all
% connections leaving a weight (i.e. the elements of a column of the
% following matrix).

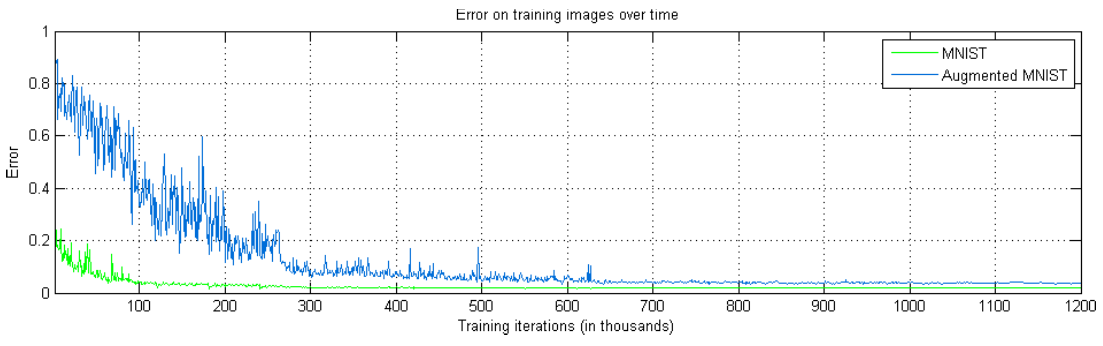

Figure 1. Training error over time. Electrochemical stimuli from other neurons arrive at the dendrites, causing changes in polarity. If the neuron's electrical potential reaches a certain threshold, the neuron "fires", sending an electrical impulse down the axon. At each axon terminal, the neuron releases molecules into a synapse, where the dendrite of the another neuron awaits.

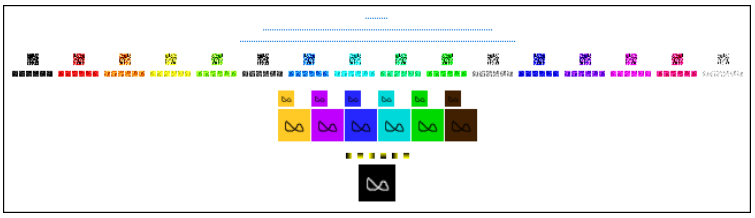

Figure 1. A convolutional network. Electrochemical stimuli from other neurons arrive at the dendrites, causing changes in polarity. If the neuron's electrical potential reaches a certain threshold, the neuron "fires", sending an electrical impulse down the axon. At each axon terminal, the neuron releases molecules into a synapse, where the dendrite of the another neuron awaits.

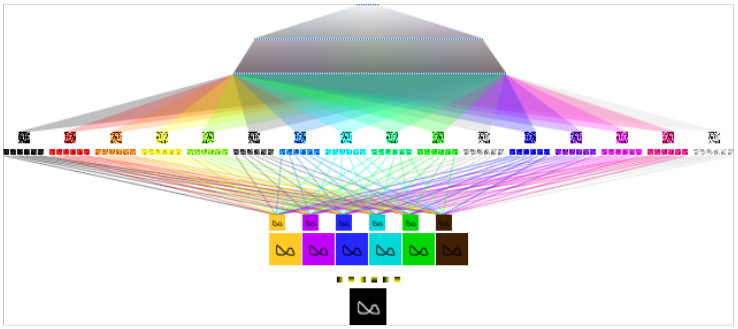

Figure 1. A convolutional network. Electrochemical stimuli from other neurons arrive at the dendrites, causing changes in polarity. If the neuron's electrical potential reaches a certain threshold, the neuron "fires", sending an electrical impulse down the axon. At each axon terminal, the neuron releases molecules into a synapse, where the dendrite of the another neuron awaits.

Once all of the nodes are connected correctly, we can iterate through forward and backward passes of the backpropagation algorithm. On the forward pass, we compute the output each node produces, and eventually the error at the end of the network. On the backward pass, we compute d_{k} for every node (or d_q if we're in a hidden layer), and work backward across the layers, to update all of the weights and biases in the network.

There are a few choices to make in the finer details of the implementation, each of which have interesting implications:

- **How many nodes to have in the hidden layer:** The size of the input layer is fixed by the input space (e.g. one node per input pixel), and the size of the output layer is fixed by the output space (e.g. one node per category, in a classification task), but the inner hidden layer has no limitations on size. If the hidden layer is too small, it won't be able to separate out important "patterns" in the high-dimension space it works in. If the layer is too large, the relative contribution of each node will be small, and it will probably take longer to train. The ideal number of nodes depends on the problem the network is tasked with.
- **How to initialize the weights:** Is it better to initialize all of the weights to 1, or initialize them randomly? Most sources seem to agree it's better to initialize the weights randomly, because this seems to decrease the probability that the optimization algorithm will get stuck in a bad local minimum. There are also recommendations to initialize the weights to small numbers (i.e. close to 1), so that no weight overpowers the others from the outset.
- **Stopping criteria:** When do we stop training? Do we attempt to reach a certain error rate in the classification of the input set? Or do we iterate a specified number of times? It's probably best to give the algorithm a combination of different stopping criteria.
- **Learning rate:** Many sources recommend damping the d_q (for any node q in the network) with a learning rate, which is usually somewhere around 0.3. This helps make sure that the gradient descent does not jump past a local minimum and miss it entirely.
- **How to feed in training data:** Given a large set of input data, it may be tempting to train the network on a small portion of it until it excels there, and then gradually increase the data set size. This will not work: getting the network to excel on a small portion of the data will over-fit the network to that data. To learn new information afterward, the network will essentially have to forget what it learned before. A better way to train the network is to sample randomly (or in some uniform manner) from the overall dataset, so that the node never over-fits to its input.

First, we use 6 filters on the 32x32 image. 6x(5x5+1)=156 params. The resulting feature maps are of size 28x28, which we then downsample to 14x14.

In the next layer, we have 16 nodes, and each node takes in some unique combination of the 6 14x14 feature maps. A node uses one filter per input, producing several new feature maps, which it sums together, and finally adds a bias. Parameter counting is harder, because it depends on the "keepers" matrix. In Lecun's matrix, there are 60 keepers, so 60x(5x5)+16=1516 params. There are 16 outputs of size 10x10, which we then downsample to 5x5.

In the final layer, we have 120 nodes, and each node takes in ALL of the downMaps of the previous layer. Once again, a node uses one filter per input, producing several

temporary feature maps, which are summed together to produce one final feature map. 120x(16x(5x5)+1) = 48120 params. There are 120 outputs of size 1x1, which we just leave as is and serve to the fully-connected network.

% Here we check and calculate the derivatives individually,using Newton's difference quotient

After testing a variety of different settings, I eventually settled on a learning rate of 0.3, and a backpropagation iteration limit of 10,000 (which takes approximately 100 seconds). Also, I created a slight modification to the backpropagation algorithm, to account for the fact that overall error does not necessarily decrease on every iteration: I store matrices of "best weights" and "best biases" for both layers of the network, so that if on the 10,000th (or some earlier) iteration the algorithm takes a bad step, I can undo that step and use the best solution computed so far.

5. Results

I used my neural network to categorize images of digits. Figure 4 shows the images used. A total of 100 16x16 pixel images were created for this experiment, each generated in an image editing program. The numbers vary in size, style, and position. Each column of numbers in Figure 4 corresponds to a unique font style. The variability in this data set is obviously not as drastic as would be found in a dataset of hand-written digits, but reading these numbers automatically is a difficult problem nonetheless.

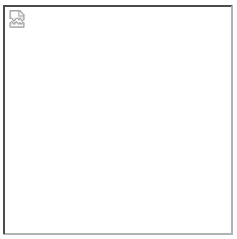

Figure 4. Set of digits used for training and testing a neural network. (Any column used for training was not used for testing.)

The network was trained on the first 8 columns of the dataset above, and tested on the final two columns. Across trials, I manipulated the number of hidden nodes in the network. Using 10 hidden nodes, the algorithm achieved an accuracy of 85% at testing. With 12 hidden nodes, the accuracy increased to 90%. With 15 hidden nodes, the accuracy increased even further, to 95%. Finally, using 20 hidden nodes, the algorithm was able to achieve a 100% correct classification rate.

6. Discussion

The performance of a neural network partly depends on chance. Since the weights in the network are initialized randomly, it is possible to get lucky (or unlucky) and start out at a very unusual place in the high-dimensional area where the nodes work. For example, the second time running the algorithm with 20 nodes yielded only 95% accuracy. It is still fairly clear that 20 nodes outperforms 10 nodes, but the real difference in performance would take much more testing to judge.

The results reported here are fairly impressive, but it is important to note that this problem is far easier than a typical problem in image analysis today. It would be much more challenging to train and test the network on hand-written digits, where there is more within-class variability, and probably also less between-class variability. This network probably has too few layers, and too few nodes per layer, to be successful in such a problem. However, it is easy to expand the network and extend the training period, so perhaps more impressive results are still possible.

7. Conclusions

Neural networks are extremely powerful pattern-recognizers. Even a simple artificial neural network can do gradient descent in a high-dimensional space, and update its own effectiveness in a fast, iterative manner. The number of iterations required for backpropagation may seem like a lot, but once the network is trained, processing time (for any input) is almost instantaneous.

In this report, I covered (1) the derivation of a mathematical node, (2) a brief overview of the “backpropagation” algorithm, (3) an exploration of a node's behaviour on its own and in a network, and also (4) the results of training a custom-built ANN on digit recognition. In the future, I would like to study other types of neural networks, such as convolutional neural networks, and so-called "deep belief" networks. I believe there is great promise in biologically-inspired algorithms for image analysis and computer vision, and I hope this report provides a good introduction to that field.

References

1. LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P.: Gradient-Based Learning Applied to Document Recognition. Proceedings of the IEEE, 86(11):2278-2324 (1998) [↑]
2. Krizhevsky, A., Sutskever, I., Hinton, G.: Imagenet Classification with Deep Convolutional Neural Networks. Advances in Neural Information Processing Systems 25:1106–1114 (2012) [↑]
3. Girshick, R., Donahue, J., Darrell, T., and Malik, J.: Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. arXiv preprint arXiv:1311.2524 (2013) [↑]
4. Razavian, A. S., Azizpour, H., Sullivan, J., and Carlsson, S.: CNN Features off-the-shelf: an Astounding Baseline for Recognition. arXiv preprint arXiv:1403.6382 (2014) [↑]
5. Fukushima, K.: Neocognitron: Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position. Biological Cybernetics, 36:193-202 (1980) [↑]
6. Hubel, D. H., and Wiesel, T. N.: Receptive Fields of Single Neurones in the Cat's Striate Cortex. Journal of Physiology 148(3): 574–591 (1959) [↑]
7. Hubel, D.H., and Wiesel, T.N.: Receptive fields, binocular interaction and functional architecture in the cat’s visual cortex. Journal of Physiology 160: 106–154 (1962) [↑]

8. Wiesel, T.N., and Hubel, D.H.: Single-cell responses in striate cortex of kittens deprived of vision in one eye. J Neurophysiol 26: 1003–1017, 1963 (http://jn.physiology.org/cgi/reprint/26/6/1003). [↑]
9. Wiesel, T.N., and Hubel, D.H.: Comparison of the effects of unilateral and bilateral eye closure on cortical unit responses in kittens. J Neurophysiol 28: 1029–1040, 1965 (http://jn.physiology.org/cgi/reprint/28/6/1029). [↑]
10. Wiesel, T.N., and Hubel, D.H.: Binocular interaction in striate cortex of kittens reared with artificial squint. J Neurophysiol 28: 1041–1059, 1965 (http://jn.physiology.org/cgi/reprint/28/6/1041). [↑]
11. Hebb, D.O.: The Organization of Behavior. Wiley, New York. (1949) [↑]
12. Cole, K.S, Curtis, H.J.: Electric impedance of the squid giant axon during activity. Journal of General Physiology 22:649–670. (1939) [↑]
13. Hodgkin A.L., Huxley A.F.: A quantitative description of membrane current and its application to conduction and excitation in nerve. J Physiol (Lond) 117:500–544. (1952) [↑]
14. LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D.: Handwritten digit recognition with a back-propagation network. Advances in Neural Information Processing Systems (1990) [↑]
15. Deng, J., Berg, A., Satheesh, S., Su, H., Khosla, A., and Fei-Fei, L.: ILSVRC-2012. http://www.image-net.org/challenges/LSVRC/2012/ (2012) [↑]

Harris, R., 2013. Neural network tutorial: The backpropagation algorithm (Parts 1-2). https://www.youtube.com/watch?v=aVId8KMsdUU

Harris, R., 2013. Visualize Back Propagation (Parts 1-5). http://www.youtube.com/watch?v=EAhog_7fSDo

Krizhevsky, 2009. Convolutional Neural Networks for Object Classification in CUDA. http://www.eecg.toronto.edu/~moshovos/CUDA08/arx/convnet_report.pdf

Robert Gordon University. The Backpropagation Algorithm. http://www4.rgu.ac.uk/files/chapter3%20-%20bp.pdf

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

Yamins et al., 2014. Performance-optimized hierarchical models predict neural responses in higher visual cortex

Serre, Oliva, & Poggio, 2007. A feedforward architecture accounts for rapid categorization.

Robert McMillan, 2014. Inside the Artificial Brain That’s Remaking the Google Empire. Wired. http://www.wired.com/2014/07/google_brain/

Listen to this as a text document, removing images manually, and citations with this:

\[[0-9]*\]

\[[0-9]*,* *[0-9]*,* *[0-9]*,* *[0-9]*,* *[0-9]*\]

new experiment: test the statement "performance scales linearly with dataset size"