Deep Learning: An Introduction for Applied Mathematicians

Catherine F. Higham* Desmond J. Higham†

January 19, 2018

Abstract

Multilayered artificial neural networks are becoming a pervasive tool in a host of application fields. At the heart of this deep learning revolution are familiar concepts from applied and computational mathematics; notably, in calculus, approximation theory, optimization and linear algebra. This article provides a very brief introduction to the basic ideas that underlie deep learning from an applied mathematics perspective. Our target audience includes postgraduate and final year undergraduate students in mathematics who are keen to learn about the area. The article may also be useful for instructors in mathematics who wish to enliven their classes with references to the application of deep learning techniques. We focus on three fundamental questions: what is a deep neural network? how is a network trained? what is the stochastic gradient method? We illustrate the ideas with a short MATLAB code that sets up and trains a network. We also show the use of state-of-the art software on a large scale image classification problem. We finish with references to the current literature.

1 Motivation

Most of us have come across the phrase deep learning. It relates to a set of tools that have become extremely popular in a vast range of application fields, from image recognition, speech recognition and natural language processing to targeted advertising and drug discovery. The field has grown to the extent where sophisticated software packages are available in the public domain, many produced by high-profile technology companies. Chip manufacturers are also customizing their graphics processing units (GPUs) for kernels at the heart of deep learning.

*School of Computing Science, University of Glasgow, UK (Catherine.Higham@glasgow.ac.uk). This author was supported by the EPSRC UK Quantum Technology Programme under grant EP/M01326X/1.

†Department of Mathematics and Statistics, University of Strathclyde, UK (d.j.higham@strath.ac.uk). This author was supported by the EPSRC/RCUK Digital Economy Programme under grant EP/M00158X/1.

Whether or not its current level of attention is fully justified, deep learning is clearly a topic of interest to employers, and therefore to our students. Although there are many useful resources available, we feel that there is a niche for a brief treatment aimed at mathematical scientists. For a mathematics student, gaining some familiarity with deep learning can enhance employment prospects. For mathematics educators, slipping "Applications to Deep Learning" into the syllabus of a class on calculus, approximation theory, optimization, linear algebra, or scientific computing is a great way to attract students and maintain their interest in core topics. The area is also ripe for independent project study.

There is no novel material in this article, and many topics are glossed over or omitted. Our aim is to present some key ideas as clearly as possible while avoiding non-essential detail. The treatment is aimed at readers with a background in mathematics who have completed a course in linear algebra and are familiar with partial differentiation. Some experience of scientific computing is also desirable.

To keep the material concrete, we list and walk through a short MATLAB code that illustrates the main algorithmic steps in setting up, training and applying an artificial neural network. We also demonstrate the high-level use of state-of-the-art software on a larger scale problem.

Section 2 introduces some key ideas by creating and training an artificial neural network using a simple example. Section 3 sets up some useful notation and defines a general network. Training a network, which involves the solution of an optimization problem, is the main computational challenge in this field. In Section 4 we describe the stochastic gradient method, a variation of a traditional optimization technique that is designed to cope with very large scale sets of training data. Section 5 explains how the partial derivatives needed for the stochastic gradient method can be computed efficiently using back propagation. First-principles MATLAB code that illustrates these ideas is provided in section 6. A larger scale problem is treated in section 7. Here we make use of existing software. Rather than repeatedly acknowledge work throughout the text, we have chosen to place the bulk of our citations in Section 8, where pointers to the large and growing literature are given. In that section we also raise issues that were not mentioned elsewhere, and highlight some current hot topics.

2 Example of an Artificial Neural Network

This article takes a data fitting view of artificial neural networks. To be concrete, consider the set of points shown in Figure 1. This shows *labeled data*—some points are in category A, indicated by circles, and the rest are in category B, indicated by crosses. For example, the data may show oil drilling sites on a map, where category A denotes a successful outcome. Can we use this data to categorize a newly proposed drilling site? Our job is to construct a transformation that takes any point in $\mathbb{R}^2$ and returns either a circle or a square. Of course, there are many reasonable ways to construct such a transformation.

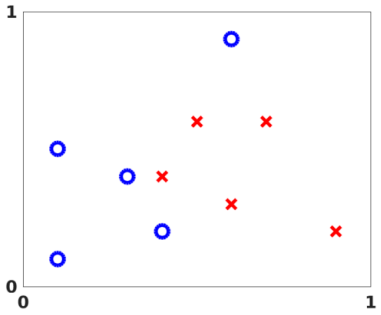

Figure 1: Labeled data points in $\mathbb{R}^2$. Circles denote points in category A. Crosses denote points in category B.

The artificial neural network approach uses repeated application of a simple, nonlinear function.

We will base our network on the sigmoid function

$$\sigma(x) = \frac{1}{1+e^{-x}}, \tag{1}$$

which is illustrated in the upper half of Figure 2 over the interval $-10 \le x \le 10$. We may regard $\sigma(x)$ as a smoothed version of a step function, which itself mimics the behavior of a neuron in the brain—firing (giving output equal to one) if the input is large enough, and remaining inactive (giving output equal to zero) otherwise. The sigmoid also has the convenient property that its derivative takes the simple form

$$\sigma'(x) = \sigma(x)\,(1-\sigma(x))\,, \tag{2}$$

which is straightforward to verify.

The steepness and location of the transition in the sigmoid function may be altered by scaling and shifting the argument or, in the language of neural networks, by *weighting* and *biasing* the input. The lower plot in Figure 2 shows $\sigma\,(3(x-5))$. The factor 3 has sharpened the changeover and the shift -5 has altered its location. To keep our notation managable, we need to interpret the sigmoid function in a vectorized sense. For $z \in \mathbb{R}^m$, $\sigma : \mathbb{R}^m \to \mathbb{R}^m$ is defined by applying the sigmoid function in the obvious componentwise manner, so that

$$(\sigma(z))_i = \sigma(z_i).$$

With this notation, we can set up layers of neurons. In each layer, every neuron outputs a single real number, which is passed to every neuron in the next layer. At the next layer, each neuron forms its own weighted combination of these values, adds its own bias, and applies the sigmoid function. Introducing some mathematics, if the real numbers produced by the neurons in one layer are

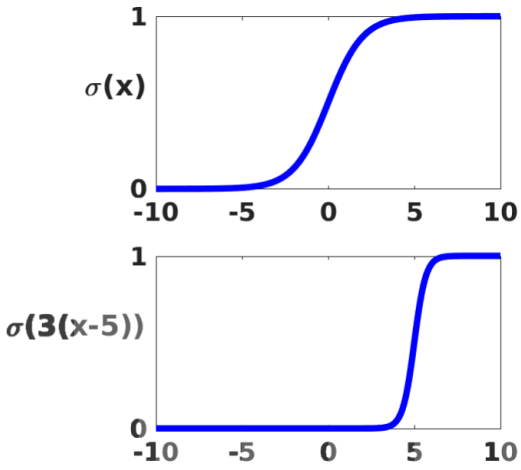

Figure 2: Upper: sigmoid function (1). Lower: sigmoid with shifted and scaled input.

collected into a vector, a, then the vector of outputs from the next layer has the form

$$\sigma(Wa + b). \tag{3}$$

Here, W is matrix and b is a vector. We say that W contains the *weights* and b contains the *biases*. The number of columns in W matches the number of neurons that produced the vector a at the previous layer. The number of rows in W matches the number of neurons at the current layer. The number of components in b also matches the number of neurons at the current layer. To emphasize the role of the ith neuron in (3), we could pick out the ith component as

$$\sigma\left(\sum_j w_{ij} a_j + b_i\right),$$

where the sum runs over all entries in a. Throughout this article, we will be switching between the vectorized and componentwise viewpoints to strike a balance between clarity and brevity.

In the next section, we introduce a full set of notation that allows us to define a general network. Before reaching that stage, we will give a specific example. Figure 3 represents an artificial neural network with four layers. We will apply this form of network to the problem defined by Figure 1. For the network in Figure 3 the first (input) layer is represented by two circles. This is because our input data points have two components. The second layer has two solid circles, indicating that two neurons are being employed. The arrows from layer one to layer two indicate that both components of the input data are made available

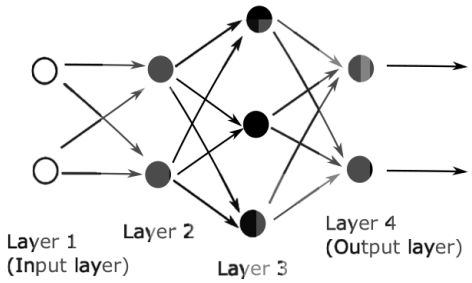

Figure 3: A network with four layers.

to the two neurons in layer two. Since the input data has the form $x \in \mathbb{R}^2$, the weights and biases for layer two may be represented by a matrix $W^{[2]} \in \mathbb{R}^{2\times 2}$ and a vector $b^{[2]} \in \mathbb{R}^2$, respectively. The output from layer two then has the form

$$\sigma(W^{[2]}x + b^{[2]}) \in \mathbb{R}^2.$$

Layer three has three neurons, each receiving input in $\mathbb{R}^2$. So the weights and biases for layer three may be represented by a matrix $W^{[3]} \in \mathbb{R}^{3\times 2}$ and a vector $b^{[3]} \in \mathbb{R}^3$, respectively. The output from layer three then has the form

$$\sigma\left(W^{[3]}\sigma(W^{[2]}x + b^{[2]}) + b^{[3]}\right) \in \mathbb{R}^3.$$

The fourth (output) layer has two neurons, each receiving input in $\mathbb{R}^3$. So the weights and biases for this layer may be represented by a matrix $W^{[4]} \in \mathbb{R}^{2\times 3}$ and a vector $b^{[4]} \in \mathbb{R}^{[2]}$, respectively. The output from layer four, and hence from the overall network, has the form

$$F(x) = \sigma\left(W^{[4]}\sigma\left(W^{[3]}\sigma(W^{[2]}x + b^{[2]}) + b^{[3]}\right) + b^{[4]}\right) \in \mathbb{R}^2. \tag{4}$$

The expression (4) defines a function $F : \mathbb{R}^2 \to \mathbb{R}^2$ in terms of its 23 parameters—the entries in the weight matrices and bias vectors. Recall that our aim is to produce a classifier based on the data in Figure 1. We do this by optimizing over the parameters. We will require $F(x)$ to be close to $[1,0]^T$ for data points in category A and close to $[0,1]^T$ for data points in category B. Then, given a new point $x \in \mathbb{R}^2$, it would be reasonable to classify it according to the largest component of $F(x)$; that is, category A if $F_1(x) > F_2(x)$ and category B if $F_1(x) < F_2(x)$, with some rule to break ties. This requirement on F may be specified through a *cost function*. Denoting the ten data points in Figure 1 by $\{x^{\{i\}}\}_{i=1}^{10}$, we use $y(x^{\{i\}})$ for the target output; that is,

$$y(x^{\{i\}}) = \begin{cases} \begin{bmatrix} 1 \\ 0 \end{bmatrix} & \text{if } x^{\{i\}} \text{ is in category A,} \\ \\ \begin{bmatrix} 0 \\ 1 \end{bmatrix} & \text{if } x^{\{i\}} \text{ is in category B.} \end{cases} \tag{5}$$

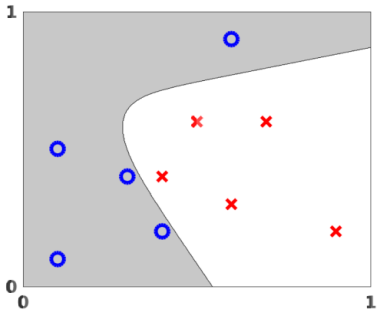

Figure 4: Visualization of output from an artificial neural network applied to the data in Figure 1.

Our cost function then takes the form

$$\text{Cost}\left(W^{[2]}, W^{[3]}, W^{[4]}, b^{[2]}, b^{[3]}, b^{[4]}\right) = \frac{1}{10}\sum_{i=1}^{10} \tfrac{1}{2}\|y(x^{\{i\}}) - F(x^{\{i\}})\|_2^2. \qquad (6)$$

Here, the factor $\frac{1}{2}$ is included for convenience; it simplifies matters when we start differentiating. We emphasize that Cost is a function of the weights and biases—the data points are fixed. The form in (6), where discrepancy is measured by averaging the squared Euclidean norm over the data points, is often refered to as a *quadratic cost function*. In the language of optimization, Cost is our *objective function*.

Choosing the weights and biases in a way that minimizes the cost function is refered to as *training* the network. We note that, in principle, rescaling an objective function does not change an optimization problem. We should arrive at the same minimizer if we change Cost to, for example, 100 Cost or Cost/30. So the factors $1/10$ and $1/2$ in (6) should have no effect on the outcome.

For the data in Figure 1, we used the MATLAB optimization toolbox to minimize the cost function (6) over the 23 parameters defining $W^{[2]}$, $W^{[3]}$, $W^{[4]}$, $b^{[2]}$, $b^{[3]}$ and $b^{[4]}$. More precisely, we used the nonlinear least-squares solver `lsqnonlin`. For the trained network, Figure 4 shows the boundary where $F_1(x) > F_2(x)$. So, with this approach, any point in the shaded region would be assigned to category A and any point in the unshaded region to category B.

Figure 5 shows how the network responds to additional training data. Here we added one further category B point, indicated by the extra cross at $(0.3, 0.7)$, and re-ran the optimization routine.

The example illustrated in Figure 4 is small-scale by the standards of today's deep learning tools. However, the underlying optimization problem, minimizing a non-convex objective function over 23 variables, is fundamentally difficult. We cannot exhaustively search over a 23 dimensional parameter space, and we cannot guarantee to find the global minimum of a non-convex function.

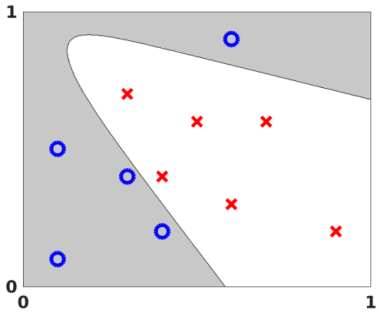

Figure 5: Repeat of the experiment in Figure 4 with an additional data point.

Indeed, some experimentation with the location of the data points in Figure 4 and with the choice of initial guess for the weights and biases makes it clear that `lsqnonlin`, with its default settings, cannot always find an acceptable solution. This motivates the material in sections 4 and 5, where we focus on the optimization problem.

3 The General Set-up

The four layer example in section 2 introduced the idea of neurons, represented by the sigmoid function, acting in layers. At a general layer, each neuron receives the same input—one real value from every neuron at the previous layer—and produces one real value, which is passed to every neuron at the next layer. There are two exceptional layers. At the input layer, there is no "previous layer" and each neuron receives the input vector. At the output layer, there is no "next layer" and these neurons provide the overall output. The layers in between these two are called *hidden layers*. There is no special meaning to this phrase; it simply indicates that these neurons are performing intermediate calculations. Deep learning is a loosely-defined term which implies that many hidden layers are being used.

We now spell out the general form of the notation from section 2. We suppose that the network has L layers, with layers 1 and L being the input and output layers, respectively. Suppose that layer l, for $l = 1, 2, 3, \ldots, L$ contains n_l neurons. So n_1 is the dimension of the input data. Overall, the network maps from $\mathbb{R}^{n_1}$ to $\mathbb{R}^{n_L}$. We use $W^{[l]} \in \mathbb{R}^{n_l \times n_{l-1}}$ to denote the matrix of weights at layer l. More precisely, $w^{[l]}_{jk}$ is the weight that neuron j at layer l applies to the output from neuron k at layer $l-1$. Similarly, $b^{[l]} \in \mathbb{R}^{n_l}$ is the vector of biases for layer l, so neuron j at layer l uses the bias $b^{[l]}_j$.

In Fig 6 we give an example with $L = 5$ layers. Here, $n_1 = 4$, $n_2 = 3$, $n_3 = 4$, $n_4 = 5$ and $n_5 = 2$, so $W^{[2]} \in \mathbb{R}^{3\times 4}$, $W^{[3]} \in \mathbb{R}^{4\times 3}$, $W^{[4]} \in \mathbb{R}^{5\times 4}$, $W^{[5]} \in \mathbb{R}^{2\times 5}$, $b^{[2]} \in \mathbb{R}^3$, $b^{[3]} \in \mathbb{R}^4$, $b^{[4]} \in \mathbb{R}^5$ and $b^{[5]} \in \mathbb{R}^2$.

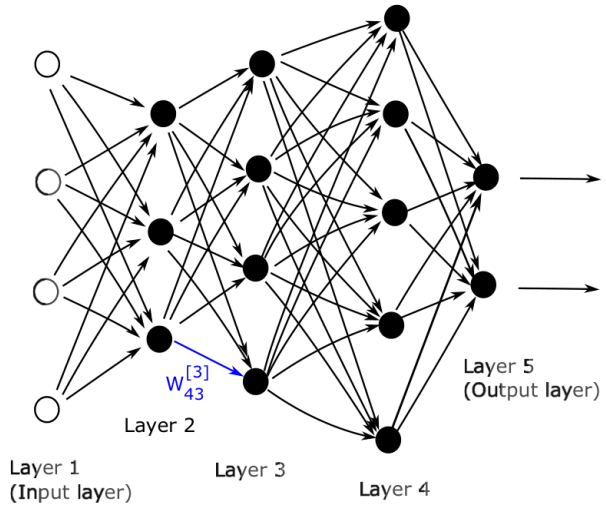

Figure 6: A network with five layers. The edge corresponding to the weight $w_{43}^{[3]}$ is highlighted. The output from neuron number 3 at layer 2 is weighted by the factor $w_{43}^{[3]}$ when it is fed into neuron number 4 at layer 3.

Given an input $x \in \mathbb{R}^{n_1}$, we may then neatly summarize the action of the network by letting $a_j^{[l]}$ denote the output, or *activation*, from neuron j at layer l. So, we have

$$a^{[1]} = x \in \mathbb{R}^{n_1}, \tag{7}$$

$$a^{[l]} = \sigma\left(W^{[l]}a^{[l-1]} + b^{[l]}\right) \in \mathbb{R}^{n_l}, \quad \text{for } l = 2, 3, \ldots, L. \tag{8}$$

It should be clear that (7) and (8) amount to an algorithm for feeding the input forward through the network in order to produce an output $a^{[L]} \in \mathbb{R}^{n_L}$. At the end of section 5 this algorithm appears within a pseudocode description of an approach for training a network.

Now suppose we have N pieces of data, or *training points*, in $\mathbb{R}^{n_1}$, $\{x^{\{i\}}\}_{i=1}^N$, for which there are given target outputs $\{y(x^{\{i\}})\}_{i=1}^N$ in $\mathbb{R}^{n_L}$. Generalizing (6), the quadratic cost function that we wish to minimize has the form

$$\text{Cost} = \frac{1}{N}\sum_{i=1}^{N} \tfrac{1}{2}\|y(x^{\{i\}}) - a^{[L]}(x^{\{i\}})\|_2^2, \tag{9}$$

where, to keep notation under control, we have not explicitly indicated that Cost is a function of all the weights and biases.

4 Stochastic Gradient

We saw in the previous two sections that training a network corresponds to choosing the parameters, that is, the weights and biases, that minimize the cost function. The weights and biases take the form of matrices and vectors, but at this stage it is convenient to imagine them stored as a single vector that we call p. The example in Figure 3 has a total of 23 weights and biases. So, in that case, $p \in \mathbb{R}^{23}$. Generally, we will suppose $p \in \mathbb{R}^s$, and write the cost function in (9) as $\mathrm{Cost}(p)$ to emphasize its dependence on the parameters. So $\mathrm{Cost} : \mathbb{R}^s \to \mathbb{R}$.

We now introduce a classical method in optimization that is often referred to as *steepest descent* or *gradient descent*. The method proceeds iteratively, computing a sequence of vectors in $\mathbb{R}^s$ with the aim of converging to a vector that minimizes the cost function. Suppose that our current vector is p. How should we choose a perturbation, Δp, so that the next vector, $p + \Delta p$, represents an improvement? If Δp is small, then ignoring terms of order $\| \Delta p \|^2$, a Taylor series expansion gives

$$\mathrm{Cost}(p + \Delta p) \approx \mathrm{Cost}(p) + \sum_{r=1}^{s} \frac{\partial \mathrm{Cost}(p)}{\partial p_r} \Delta p_r. \tag{10}$$

Here $\partial \mathrm{Cost}(p)/\partial p_r$ denotes the partial derivative of the cost function with respect to the rth parameter. For convenience, we will let $\nabla \mathrm{Cost}(p) \in \mathbb{R}^s$ denote the vector of partial derivatives, known as the *gradient*, so that

$$\left(\nabla \mathrm{Cost}(p)\right)_r = \frac{\partial \mathrm{Cost}(p)}{\partial p_r}.$$

Then (10) becomes

$$\mathrm{Cost}(p + \Delta p) \approx \mathrm{Cost}(p) + \nabla \mathrm{Cost}(p)^T \Delta p. \tag{11}$$

Our aim is to reduce the value of the cost function. The relation (11) motivates the idea of choosing Δp to make $\nabla \mathrm{Cost}(p)^T \Delta p$ as negative as possible. We can address this problem via the Cauchy–Schwarz inequality, which states that for any $f, g \in \mathbb{R}^s$, we have $|f^T g| \leq \| f \|_2 \| g \|_2$. So the most negative that $f^T g$ can be is $-\| f \|_2 \| g \|_2$, which happens when $f = -g$. Hence, based on (11), we should choose Δp to lie in the direction $-\nabla \mathrm{Cost}(p)$. Keeping in mind that (11) is an approximation that is relevant only for small Δp, we will limit ourselves to a small step in that direction. This leads to the update

$$p \to p - \eta \nabla \mathrm{Cost}(p). \tag{12}$$

Here η is small stepsize that, in this context, is known as the *learning rate*. This equation defines the steepest descent method. We choose an initial vector and iterate with (12) until some stopping criterion has been met, or until the number of iterations has exceeded the computational budget.

Our cost function (9) involves a sum of individual terms that runs over the training data. It follows that the partial derivative $\nabla \mathrm{Cost}(p)$ is a sum over the training data of individual partial derivatives. More precisely, let

$$C_{x^{\{i\}}} = \tfrac{1}{2}\|y(x^{\{i\}}) - a^{[L]}(x^{\{i\}})\|_2^2. \tag{13}$$

Then, from (9),

$$\nabla \mathrm{Cost}(p) = \frac{1}{N}\sum_{i=1}^{N} \nabla C_{x^{\{i\}}}(p). \tag{14}$$

When we have a large number of parameters and a large number of training points, computing the gradient vector (14) at every iteration of the steepest descent method (12) can be prohibitively expensive. A much cheaper alternative is to replace the mean of the individual gradients over all training points by the gradient at a single, randomly chosen, training point. This leads to the simplest form of what is called the *stochastic gradient* method. A single step may be summarized as

1. Choose an integer i uniformly at random from $\{1, 2, 3, \ldots, N\}$.
2. Update
$$p \to p - \eta \nabla C_{x^{\{i\}}}(p). \tag{15}$$

In words, at each step, the stochastic gradient method uses one randomly chosen training point to represent the full training set. As the iteration proceeds, the method sees more training points. So there is some hope that this dramatic reduction in cost-per-iteration will be worthwhile overall. We note that, even for very small η, the update (15) is not guaranteed to reduce the overall cost function—we have traded the mean for a single sample. Hence, although the phrase *stochastic gradient descent* is widely used, we prefer to use stochastic gradient.

The version of the stochastic gradient method that we introduced in (15) is the simplest from a large range of possibilities. In particular, the index i in (15) was chosen by sampling *with replacement*—after using a training point, it is returned to the training set and is just as likely as any other point to be chosen at the next step. An alternative is to sample without replacement; that is, to cycle through each of the N training points in a random order. Performing N steps in this manner, refered to as completing an *epoch*, may be summarized as follows:

1. Shuffle the integers $\{1, 2, 3, \ldots, N\}$ into a new order, $\{k_1, k_2, k_3, \ldots, k_N\}$.
2. for $i = 1$ upto N, update
$$p \to p - \eta \nabla C_{x^{\{k_i\}}}(p). \tag{16}$$

If we regard the stochastic gradient method as approximating the mean over all training points in (14) by a single sample, then it is natural to consider a compromise where we use a small sample average. For some $m \ll N$ we could take steps of the following form.

1. Choose m integers, $k_1, k_2, \ldots, k_m$, uniformly at random from $\{1, 2, 3, \ldots, N\}$.

2. Update
$$p \to p - \eta \frac{1}{m} \sum_{i=1}^{m} \nabla C_{x^{\{k_i\}}}(p). \tag{17}$$

In this iteration, the set $\{x^{\{k_i\}}\}_{i=1}^{n}$ is known as a *mini-batch*. There is a *without replacement* alternative where, assuming $N = Km$ for some K, we split the training set randomly into K distinct mini-batches and cycle through them.

Because the stochastic gradient method is usually implemented within the context of a very large scale computation, algorithmic choices such as mini-batch size and the form of randomization are often driven by the requirements of high performance computing architectures. Also, it is, of course, possible to vary these choices, along with others, such as the learning rate, dynamically as the training progresses in an attempt to accelerate convergence.

Section 6 describes a simple MATLAB code that uses a vanilla stochastic gradient method. In section 7 we use a state-of-the-art implementation and section 8 has pointers to the current literature.

5 Back Propagation

We are now in a position to apply the stochastic gradient method in order to train an artificial neural network. So we switch from the general vector of parameters, p, used in section 4 to the entries in the weight matrices and bias vectors. Our task is to compute partial derivatives of the cost function with respect to each $w_{jk}^{[l]}$ and $b_j^{[l]}$. We saw that the idea behind the stochastic gradient method is to exploit the structure of the cost function: because (9) is a linear combination of individual terms that runs over the training data the same is true of its partial derivatives. We therefore focus our attention on computing those individual partial derivatives.

Hence, for a fixed training point we regard $C_{x^{\{i\}}}$ in (13) as a function of the weights and biases. So we may drop the dependence on $x^{\{i\}}$ and simply write
$$C = \tfrac{1}{2}\|y - a^{[L]}\|_2^2. \tag{18}$$

We recall from (8) that $a^{[L]}$ is the output from the artificial neural network. The dependence of C on the weights and biases arises only through $a^{[L]}$.

To derive worthwhile expressions for the partial derivatives, it is useful to introduce two further sets of variables. First we let
$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]} \in \mathbb{R}^{n_l}, \qquad \text{for } l = 2, 3, \ldots, L. \tag{19}$$

We refer to $z_j^{[l]}$ as the *weighted input* for neuron j at layer l. The fundamental relation (8) that propagates information through the network may then be written
$$a^{[l]} = \sigma\left(z^{[l]}\right), \qquad \text{for } l = 2, 3, \ldots, L. \tag{20}$$

Second, we let $\delta^{[l]} \in \mathbb{R}^{n_l}$ be defined by

$$\delta_j^{[l]} = \frac{\partial C}{\partial z_j^{[l]}}, \qquad \text{for } 1 \le j \le n_l \quad \text{and} \quad 2 \le l \le L. \tag{21}$$

This expression, which is often called the *error* in the jth neuron at layer l, is an intermediate quantity that is useful both for analysis and computation. However, we point out that this useage of the term error is somewhat ambiguous. At a general, hidden layer, it is not clear how much to "blame" each neuron for discrepancies in the final output. Also, at the output layer, L, the expression (21) does not quantify those discrepancies directly. The idea of referring to $\delta_j^{[l]}$ in (21) as an error seems to have arisen because the cost function can only be at a minimum if all partial derivatives are zero, so $\delta_j^{[l]} = 0$ is a useful goal. As we mention later in this section, it may be more helpful to keep in mind that $\delta_j^{[l]}$ measures the sensitivity of the cost function to the weighted input for neuron j at layer l.

At this stage we also need to define the Hadamard, or componentwise, product of two vectors. If $x, y \in \mathbb{R}^n$, then $x \circ y \in \mathbb{R}^n$ is defined by $(x \circ y)_i = x_i y_i$. In words, the Hadamard product is formed by pairwise multiplication of the corresponding components.

With this notation, the following results are a consequence of the chain rule.

Lemma 1 *We have*

$$\delta^{[L]} = \sigma'(z^{[L]}) \circ (a^L - y), \tag{22}$$

$$\delta^{[l]} = \sigma'(z^{[l]}) \circ (W^{[l+1]})^T \delta^{[l+1]}, \qquad \text{for } 2 \le l \le L-1, \tag{23}$$

$$\frac{\partial C}{\partial b_j^{[l]}} = \delta_j^{[l]}, \qquad \text{for } 2 \le l \le L, \tag{24}$$

$$\frac{\partial C}{\partial w_{jk}^{[l]}} = \delta_j^{[l]} a_k^{[l-1]}. \qquad \text{for } 2 \le l \le L. \tag{25}$$

Proof We begin by proving (22). The relation (20) with $l = L$ shows that $z_j^{[L]}$ and $a_j^{[L]}$ are connected by $a^{[L]} = \sigma\left(z^{[L]}\right)$, and hence

$$\frac{\partial a_j^{[L]}}{\partial z_j^{[L]}} = \sigma'(z_j^{[L]}).$$

Also, from (18),

$$\frac{\partial C}{\partial a_j^{[L]}} = \frac{\partial}{\partial a_j^{[L]}} \frac{1}{2} \sum_{k=1}^{n_L} (y_k - a_k^{[L]})^2 = -(y_j - a_j^{[L]}).$$

So, using the chain rule,

$$\delta_j^{[L]} = \frac{\partial C}{\partial z_j^{[L]}} = \frac{\partial C}{\partial a_j^{[L]}} \frac{\partial a_j^{[L]}}{\partial z_j^{[L]}} = (a_j^{[L]} - y_j)\sigma'(z_j^{[L]}),$$

which is the componentwise form of (22).

To show (23), we use the chain rule to convert from $z_j^{[l]}$ to $\{z_k^{[l+1]}\}_{k=1}^{n_{l+1}}$. Applying the chain rule, and using the definition (21),

$$\delta_j^{[l]} = \frac{\partial C}{\partial z_j^{[l]}} = \sum_{k=1}^{n_{l+1}} \frac{\partial C}{\partial z_k^{[l+1]}} \frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}} = \sum_{k=1}^{n_{l+1}} \delta_k^{[l+1]} \frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}}. \tag{26}$$

Now, from (19) we know that $z_k^{[l+1]}$ and $z_j^{[l]}$ are connected via

$$z_k^{[l+1]} = \sum_{s=1}^{n_l} w_{ks}^{[l+1]} \sigma\left(z_s^{[l]}\right) + b_k^{[l+1]}.$$

Hence,

$$\frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}} = w_{kj}^{[l+1]} \sigma'\left(z_j^{[l]}\right).$$

In (26) this gives

$$\delta_j^{[l]} = \sum_{k=1}^{n_{l+1}} \delta_k^{[l+1]} w_{kj}^{[l+1]} \sigma'\left(z_j^{[l]}\right),$$

which may be rearranged as

$$\delta_j^{[l]} = \sigma'\left(z_j^{[l]}\right) \left((W^{[l+1]})^T \delta^{[l+1]}\right)_j.$$

This is the componentwise form of (23).

To show (24), we note from (19) and (20) that $z_j^{[l]}$ is connected to $b_j^{[l]}$ by

$$z_j^{[l]} = \left(W^{[l]} \sigma\left(z^{[l-1]}\right)\right)_j + b_j^{[l]}.$$

Since $z^{[l-1]}$ does not depend on $b_j^{[l]}$, we find that

$$\frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = 1.$$

Then, from the chain rule,

$$\frac{\partial C}{\partial b_j^{[l]}} = \frac{\partial C}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = \frac{\partial C}{\partial z_j^{[l]}} = \delta_j^{[l]},$$

using the definition (21). This gives (24).

Finally, to obtain (25) we start with the componentwise version of (19),

$$z_j^{[l]} = \sum_{k=1}^{n_{l-1}} w_{jk}^{[l]} a_k^{[l-1]} + b_j^{[l]},$$

which gives

$$\frac{\partial z_j^{[l]}}{\partial w_{jk}^{[l]}} = a_k^{[l-1]}, \qquad \text{independently of } j, \tag{27}$$

and

$$\frac{\partial z_s^{[l]}}{\partial w_{jk}^{[l]}} = 0, \qquad \text{for } s \neq j. \tag{28}$$

In words, (27) and (28) follow because the jth neuron at layer l uses the weights from only the jth row of $W^{[l]}$, and applies these weights linearly. Then, from the chain rule, (27) and (28) give

$$\frac{\partial C}{\partial w_{jk}^{[l]}} = \sum_{s=1}^{n_l} \frac{\partial C}{\partial z_s^{[l]}} \frac{\partial z_s^{[l]}}{\partial w_{jk}^{[l]}} = \frac{\partial C}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial w_{jk}^{[l]}} = \frac{\partial C}{\partial z_j^{[l]}} a_k^{[l-1]} = \delta_j^{[l]} a_k^{[l-1]},$$

where the last step used the definition of $\delta_j^{[l]}$ in (21). This completes the proof. ∎

There are many aspects of Lemma 1 that deserve our attention. We recall from (7), (19) and (20) that the output $a^{[L]}$ can be evaluated from a *forward pass* through the network, computing $a^{[1]}$, $z^{[2]}$, $a^{[2]}$, $z^{[3]}$, ..., $a^{[L]}$ in order. Having done this, we see from (22) that $\delta^{[L]}$ is immediately available. Then, from (23), $\delta^{[L-1]}$, $\delta^{[L-2]}$, ..., $\delta^{[2]}$ may be computed in a *backward pass*. From (24) and (25), we then have access to the partial derivatives. Computing gradients in this way is known as *back propagation*.

To gain further understanding of the back propagation formulas (24) and (25) in Lemma 1, it is useful to recall the fundamental definition of a partial derivative. The quantity $\partial C / \partial w_{jk}^{[l]}$ measures how C changes when we make a small perturbation to $w_{jk}^{[l]}$. For illustration, Figure 6 highlights the weight $w_{43}^{[3]}$. It is clear that a change in this weight has no effect on the output of previous layers. So to work out $\partial C / \partial w_{43}^{[3]}$ we do not need to know about partial derivatives at previous layers. It should, however, be possible to express $\partial C / \partial w_{43}^{[3]}$ in terms of partial derivatives at subsequent layers. More precisely, the activation feeding into the 4th neuron on layer 3 is $z_4^{[3]}$, and, by definition, $\delta_4^{[3]}$ measures the sensitivity of C with respect to this input. Feeding in to this neuron we have $w_{43}^{[3]} a_3^{[2]} + \text{constant}$, so it makes sense that

$$\frac{\partial C}{\partial w_{43}^{[3]}} = \delta_4^{[3]} a_3^{[2]}.$$

Similarly, in terms of the bias, $b_4^{[3]} + \text{constant}$ is feeding in to the neuron, which explains why

$$\frac{\partial C}{\partial b_4^{[3]}} = \delta_4^{[3]} \times 1.$$

We may avoid the Hadamard product notation in (22) and (23) by introducing diagonal matrices. Let $D^{[l]} \in \mathbb{R}^{n_l \times n_l}$ denote the diagonal matrix with (i,i) entry given by $\sigma'(z_i^{[l]})$. Then we see that $\delta^{[L]} = D^{[L]}(a^{[L]} - y)$ and $\delta^{[l]} = D^{[l]}(W^{[l+1]})^T \delta^{[l+1]}$. We could expand this out as

$$\delta^{[l]} = D^{[l]}(W^{[l+1]})^T D^{[l+1]}(W^{[l+2]})^T \cdots D^{[L-1]}(W^{[L]})^T D^{[L]}(a^{[L]} - y).$$

We also recall from (2) that $\sigma'(z)$ is trivial to compute.

The relation (24) shows that $\delta^{[l]}$ corresponds precisely to the gradient of the cost function with respect to the biases at layer l. If we regard $\partial C / \partial w_{jk}^{[l]}$ as defining the (j,k) component in a matrix of partial derivatives at layer l, then (25) shows this matrix to be the *outer product* $\delta^{[l]} {a^{[l-1]}}^T \in \mathbb{R}^{n_l \times n_{l-1}}$.

Putting this together, we may write the following pseudocode for an algorithm that trains a network using a fixed number, Niter, of stochastic gradient iterations. For simplicity, we consider the basic version (15) where single samples are chosen with replacement. For each training point, we perform a forward pass through the network in order to evaluate the activations, weighted inputs and overall output $a^{[L]}$. Then we perform a backward pass to compute the errors and updates.

For counter = 1 upto Niter

  Choose an integer k uniformly at random from $\{1, 2, 3, \ldots, N\}$

  $x^{\{k\}}$ is current training data point

  $a^{[1]} = x^{\{k\}}$

  For $l = 2$ upto L

    $z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$

    $a^{[l]} = \sigma\left(z^{[l]}\right)$

    $D^{[l]} = \mathrm{diag}\left(\sigma'(z^{[l]})\right)$

  end

  $\delta^{[L]} = D^{[L]}\left(a^{[L]} - y(x^{\{k\}})\right)$

  For $l = L - 1$ downto 2

    $\delta^{[l]} = D^{[l]}(W^{[l+1]})^T \delta^{[l+1]}$

  end

  For $l = L$ downto 2

    $W^{[l]} \to W^{[l]} - \eta\, \delta^{[l]} {a^{[l-1]}}^T$

    $b^{[l]} \to b^{[l]} - \eta\, \delta^{[l]}$

  end

end

6 Full MATLAB Example

We now give a concrete illustration involving back propagation and the stochastic gradient method. Listing 6.1 shows how a network of the form shown in Figure 3 may be used on the data in Figure 1. We note that this MATLAB code has been written for clarity and brevity, rather than efficiency or elegance. In particular, we have "hardwired" the number of layers and iterated through the forward and backward passes line by line. (Because the weights and biases do not have the the same dimension in each layer, it is not convenient to store them in a three-dimensional array. We could use a cell array or structure array, [18], and then implement the forward and backward passes in `for` loops. However, this approach produced a less readable code, and violated our self-imposed one page limit.)

The function `netbp` in Listing 6.1 contains the nested function `cost`, which evaluates a scaled version of Cost in (6). Because this function is nested, it has access to the variables in the main function, notably the training data. We point out that the nested function `cost` is not used directly in the forward and backward passes. It is called at each iteration of the stochastic gradient method so that we can monitor the progress of the training.

Listing 6.2 shows the function `activate`, used by `netbp`, which applies the sigmoid function in vectorized form.

At the start of `netbp` we set up the training data and target y values, as defined in (5). We then initialize all weights and biases using the normal pseudorandom number generator `randn`. For simplicity, we set a constant learning rate `eta = 0.05` and perform a fixed number of iterations `Niter = 1e6`.

We use the the basic stochastic gradient iteration summarized at the end of Section 5. Here, the command `randi(10)` returns a uniformly and independently chosen integer between 1 and 10.

Having stored the value of the cost function at each iteration, we use the `semilogy` command to visualize the progress of the iteration.

In this experiment, our initial guess for the weights and biases produced a cost function value of 5.3. After 10^6 stochastic gradient steps this was reduced to 7.3×10^{-4}. Figure 7 shows the `semilogy` plot, and we see that the decay is not consistent—the cost undergoes a flat period towards the start of the process. After this plateau, we found that the cost decayed at a very slow linear rate—the ratio between successive values was typically within around 10^{-6} of unity.

An extended version of `netbp` can be found in the supplementary material. This version has the extra graphics commands that make Figure 7 more readable. It also takes the trained network and produces Figure 8. This plot shows how the trained network carves up the input space. Eagle-eyed readers will spot that the solution in Figure 8. differs slightly from the version in Figure 4, where the same optimization problem was tackled by the nonlinear least-squares solver `lsqnonlin`. In Figure 9 we show the corresponding result when an extra data point is added; this can be compared with Figure 5.

```
function netbp
%NETBP  Uses backpropagation to train a network

%%%%%% DATA %%%%%%%%%%
x1 = [0.1,0.3,0.1,0.6,0.4,0.6,0.5,0.9,0.4,0.7];
x2 = [0.1,0.4,0.5,0.9,0.2,0.3,0.6,0.2,0.4,0.6];
y = [ones(1,5) zeros(1,5); zeros(1,5) ones(1,5)];

% Initialize weights and biases
rng(5000);
W2 = 0.5*randn(2,2); W3 = 0.5*randn(3,2); W4 = 0.5*randn(2,3);
b2 = 0.5*randn(2,1); b3 = 0.5*randn(3,1); b4 = 0.5*randn(2,1);

% Forward and Back propagate
eta = 0.05;                % learning rate
Niter = 1e6;               % number of SG iterations
savecost = zeros(Niter,1); % value of cost function at each iteration
for counter = 1:Niter
    k = randi(10);         % choose a training point at random
    x = [x1(k); x2(k)];
    % Forward pass
    a2 = activate(x,W2,b2);
    a3 = activate(a2,W3,b3);
    a4 = activate(a3,W4,b4);
    % Backward pass
    delta4 = a4.*(1-a4).*(a4-y(:,k));
    delta3 = a3.*(1-a3).*(W4'*delta4);
    delta2 = a2.*(1-a2).*(W3'*delta3);
    % Gradient step
    W2 = W2 - eta*delta2*x';
    W3 = W3 - eta*delta3*a2';
    W4 = W4 - eta*delta4*a3';
    b2 = b2 - eta*delta2;
    b3 = b3 - eta*delta3;
    b4 = b4 - eta*delta4;
    % Monitor progress
    newcost = cost(W2,W3,W4,b2,b3,b4)   % display cost to screen
    savecost(counter) = newcost;
end

% Show decay of cost function
save costvec
semilogy([1:1e4:Niter],savecost(1:1e4:Niter))

  function costval = cost(W2,W3,W4,b2,b3,b4)
     costvec = zeros(10,1);
     for i = 1:10
         x =[x1(i);x2(i)];
         a2 = activate(x,W2,b2);
         a3 = activate(a2,W3,b3);
         a4 = activate(a3,W4,b4);
         costvec(i) = norm(y(:,i) - a4,2);
     end
     costval = norm(costvec,2)^2;
  end % of nested function

end
```

Listing 6.1: M-file **netbp.m**.

```
function y = activate(x,W,b)
%ACTIVATE  Evaluates sigmoid function.
%
%  x is the input vector,  y is the output vector
%  W contains the weights, b contains the shifts
%
%  The ith component of y is activate((Wx+b)_i)
%  where activate(z) = 1/(1+exp(-z))

y = 1./(1+exp(-(W*x+b)));
```

Listing 6.2: M-file `activate.m`.

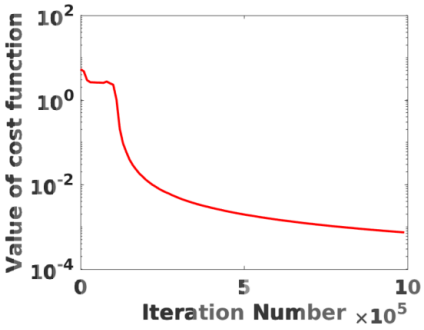

Figure 7: Vertical axis shows a scaled value of the cost function (6). Horizontal axis shows the iteration number. Here we used the stochastic gradient method to train a network of the form shown in Figure 3 on the data in Figure 1. The resulting classification function is illustrated in Figure 8.

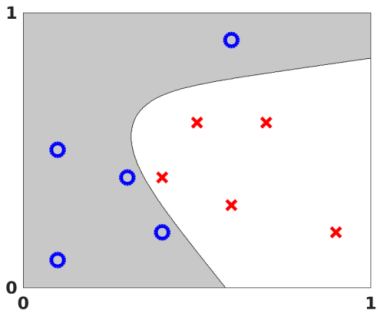

Figure 8: Visualization of output from an artificial neural network applied to the data in Figure 1. Here we trained the network using the stochastic gradient method with back propagation—behaviour of cost function is shown in Figure 7. The same optimization problem was solved with the `lsqnonlin` routine from MATLAB in order to produce Figure 4.

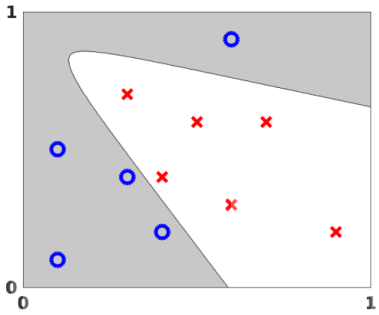

Figure 9: Visualization of output from an artificial neural network applied to the data in Figure 1 with an additional data point. Here we trained the network using the stochastic gradient method with back propagation. The same optimization problem was solved with the `lsqnonlin` routine from MATLAB in order to produce Figure 5.

7 Image Classification Example

We now move on to a more realistic task, which allows us to demonstrate the power of the deep learning approach. We make use of the MatConvNet toolbox [33], which is designed to offer key deep learning building blocks as simple MATLAB commands. So MatConvNet is an excellent environment for prototyping and for educational use. Support for GPUs also makes MatConvNet efficient for large scale computations, and pre-trained networks may be downloaded for immediate use.

Applying MatConvNet on a large scale problem also gives us the opportunity to outline further concepts that are relevant to practical computation. These are introduced in the next few subsections, before we apply them to the image classification exercise.

7.1 Convolutional Neural Networks

MatConvNet uses a special class of artificial neural networks known as Convolutional Neural Networks (CNNs), which have become a standard tool in computer vision applications. To motivate CNNs, we note that the general framework described in section 3 does not scale well in the case of digital image data. Consider a color image made up of 200 by 200 pixels, each with a red, green and blue component. This corresponds to an input vector of dimension $n_1 = 200 \times 200 \times 3 = 120,000$, and hence a weight matrix $W^{[2]}$ at level 2 that has $120,000$ columns. If we allow general weights and biases, then this approach is clearly infeasible. CNNs get around this issue by constraining the values that are allowed in the weight matrices and bias vectors. Rather than a single full-sized linear transformation, CNNs repeatedly apply a small-scale linear kernel, or filter, across portions of their input data. In effect, the weight matrices used by CNNs are extremely sparse and highly structured.

To understand why this approach might be useful, consider premultiplying an input vector in $\mathbb{R}^6$ by the matrix

$$\begin{bmatrix} 1 & -1 & & & & \\ & 1 & -1 & & & \\ & & 1 & -1 & & \\ & & & 1 & -1 & \\ & & & & 1 & -1 \end{bmatrix} \in \mathbb{R}^{5\times 6}. \tag{29}$$

This produces a vector in $\mathbb{R}^5$ made up of differences between neighboring values. In this case we are using a filter $[1, -1]$ and a *stride* of length one—the filter advances by one place after each use. Appropriate generalizations of this matrix to the case of input vectors arising from 2D images can be used to detect *edges* in an image—returning a large absolute value when there is an abrupt change in neighboring pixel values. Moving a filter across an image can also reveal other features, for example, particular types of curves or blotches of the same color. So, having specified a filter size and stride length, we can allow the training process to learn the weights in the filter as a means to extract useful structure.

The word "convolutional" arises because the linear transformations involved may be written in the form of a convolution. In the 1D case, the convolution of the vector $x \in \mathbb{R}^p$ with the filter $g_{1-p}, g_{2-p}, \ldots, g_{p-2}, g_{p-1}$ has kth component given by

$$y_k = \sum_{n=1}^{p} x_n g_{k-n}.$$

The example in (29) corresponds to a filter with $g_0 = 1$, $g_{-1} = -1$ and all other $g_k = 0$. In the case

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} a & b & c & d & & & & & & \\ & & a & b & c & d & & & & \\ & & & & a & b & c & d & & \\ & & & & & & a & b & c & d \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ 0 \end{bmatrix}, \tag{30}$$

we are applying a filter with four weights, a, b, c, and d, using a stride length of two. Because the dimension of the input vector x is not compatible with the filter length, we have *padded* with an extra zero value.

In practice, image data is typically regarded as a three dimensional tensor: each pixel has two spatial coordinates and a red/green/blue value. With this viewpoint, the filter takes the form of a small tensor that is successsively applied to patches of the input tensor and the corresponding convolution operation is multi-dimensional. From a computational perspective, a key benefit of CNNs is that the matrix-vector products involved in the forward and backward passes through the network can be computed extremely efficiently using fast transform techniques.

A convolutional layer is often followed by a *pooling* layer, which reduces dimension by mapping small regions of pixels into single numbers. For example, when these small regions are taken to be squares of four neigboring pixels in a 2D image, a *max pooling* or *average pooling* layer replaces each set of four by their maximum or average value, respectively.

7.2 Avoiding Overfitting

Overfitting occurs when a trained network performs very accurately on the given data, but cannot generalize well to new data. Loosely, this means that the fitting process has focussed too heavily on the unimportant and unrepresentative "noise" in the given data. Many ways to combat overfitting have been suggested, some of which can be used together.

One useful technique is to split the given data into two distinct groups.

- *Training data* is used in the definition of the cost function that defines the optimization problem. Hence this data drives the process that iteratively updates the weights.

- *Validation data* is not used in the optimization process—it has no effect on the way that the weights are updated from step to step. We use the validation data only to judge the performance of the current network. At each step of the optimization, we can evaluate the cost function corresponding to the validation data. This measures how well the current set of trained weights performs on unseen data.

Intuitively, overfitting corresponds to the situation where the optimization process is driving down its cost function (giving a better fit to the training data), but the cost function for the validation error is no longer decreasing (so the performance on unseen data is not improving). It is therefore reasonable to terminate the training at a stage where no improvement is seen on the validation data.

Another popular approach to tackle overfitting is to randomly and independently remove neurons during the training phase. For example, at each step of the stochastic gradient method, we could delete each neuron with probability p and train on the remaining network. At the end of the process, because the weights and biases were produced on these smaller networks, we could multiply each by a factor of p for use on the full-sized network. This technique, known as *dropout*, has the intuitive interpretation that we are constructing an average over many trained networks, with such a consensus being more reliable than any individual.

7.3 Activation and Cost Functions

In Sections 2 to 6 we used activation functions of sigmoid form (1) and a quadratic cost function (9). There are many other widely used choices, and their relative performance is application-specific. In our image classification setting it is common to use a *rectified linear unit*, or ReLU,

$$\sigma(x) = \begin{cases} 0, & \text{for } x \leq 0, \\ x, & \text{for } x > 0, \end{cases} \tag{31}$$

as the activation.

In the case where our training data $\{x^{\{i\}}\}_{i=1}^N$ comes from K labeled categories, let $l_i \in \{1, 2, \ldots, K\}$ be the given label for data point $x^{\{i\}}$. As an alternative to the quadratic cost function (9), we could use a *softmax log loss* approach, as follows. Let the output $a^{[L]}(x^{\{i\}}) =: v^{\{i\}}$ from the network take the form of a vector in $\mathbb{R}^K$ such that the jth component is large when the image is believed to be from category j. The *softmax* operation

$$(v^{\{i\}})_s \mapsto \frac{e^{v_s^{\{i\}}}}{\sum_{j=1}^K e^{v_j^{\{i\}}}}.$$

boosts the large components and produces a vector of positive weights summing to unity, which may be interpreted as probabilties. Our aim is now to force the softmax value for training point $x^{\{i\}}$ to be as close to unity as possible in component l_i, which corresponds to the correct label. Using a logarithmic rather than quadratic measure of error, we arrive at the cost function

$$-\sum_{i=1}^{N} \log\left(\frac{e^{v_{l_i}^{\{i\}}}}{\sum_{j=1}^{K} e^{v_j^{\{i\}}}}\right). \tag{32}$$

7.4 Image Classification Experiment

We now show results for a supervised learning task in image classification. To do this, we rely on the codes `cnn_cifar.m` and `cnn_cifar_init.m` that are available via the MatConvNet website. We made only minor edits, including some changes that allowed us to test the use of dropout. Hence, in particular, we are using the network architecture and parameter choices from those codes. We refer to the MatConvNet documentation and tutorial material for the fine details, and focus here on some of the bigger picture issues.

We consider a set of images, each of which falls into exactly one of the following ten categories: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck. We use labeled data from the freely available CIFAR-10 collection [20]. The images are small, having 32 by 32 pixels, each with a red, green, and blue component. So one piece of training data consists of $32 \times 32 \times 3 = 3,072$ values. We use a training set of 50,000 images, and use 10,000 more images as our validation set. Having completed the optimization and trained the network, we then judge its performance on a fresh collection of 10,000 test images, with 1,000 from each category.

Following the architecture used in the relevant MatConvNet codes, we set up a network whose layers are divided into five blocks as follows. Here we describe the dimemsions of the inputs/outputs and weights in compact tensor notation. (Of course, the tensors could be stretched into sparse vectors and matrices in order to fit in with the general framework of sections 2 to 6. But we feel that the tensor notation is natural in this context, and it is consistent with the MatConvNet syntax.)

Block 1 consists of a a convolution layer followed by a pooling layer and activation. This converts the original $32 \times 32 \times 3$ input into dimension $16 \times 16 \times 32$. In more detail, the convolutional layer uses 5×5 filters that also scan across the 3 color channels. There are 32 different filters, so overall the weights can be represented in a $5 \times 5 \times 3 \times 32$ array. The output from each filter may be described as a *feature map*. The filters are applied with unit stride. In this way, each of the 32 feature maps has dimension 32×32. Max pooling is then applied to each feature map using stride length two. This reduces the dimension of the feature maps to 16×16. A ReLU activation is then used.

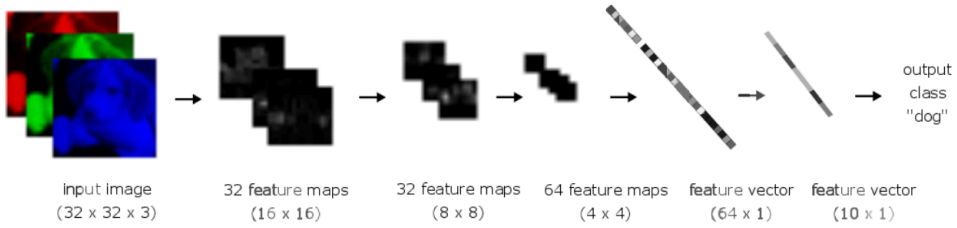

Figure 10: Overview of the CNN used for the image classification task.

Block 2 applies convolution followed by activation and then a pooling layer. This reduces the dimension to $8 \times 8 \times 32$. In more detail, we use 32 filters. Each is 5×5 across the dimensions of the feature maps, and also scans across all 32 feature maps. So the weights could be regarded as a $5 \times 5 \times 32 \times 32$ tensor. The stride length is one, so the resulting 32 feature maps are still of dimension 16×16. After ReLU activation, an average pooling layer of stride two is then applied, which reduces each of the 32 feature maps to dimension 8×8.

Block 3 applies a convolution layer followed by the activation function, and then performs a pooling operation, in a way that reduces dimension to $4 \times 4 \times 64$. In more detail, 64 filters are applied. Each filter is 5×5 across the dimensions of the feature maps, and also scans across all 32 feature maps. So the weights could be regarded as a $5 \times 5 \times 32 \times 64$ tensor. The stride has length one, resulting in feature maps of dimension 8×8. After ReLU activation, an average pooling layer of stride two is applied, which reduces each of the 64 feature maps to dimension 4×4.

Block 4 does not use pooling, just convolution followed by activation, leading to dimension $1 \times 1 \times 64$. In more detail, 64 filters are used. Each filter is 4×4 across the 64 feature maps, so the weights could be regarded as a $4 \times 4 \times 64 \times 64$ tensor, and each filter produces a single number.

Block 5 does not involve convolution. It uses a general (fully connected) weight matrix of the type discussed in sections 2 to 6 to give output of dimension $1 \times 1 \times 10$. This corresponds to a weight matrix of dimension 10×64.

A final softmax operation transforms each of the ten ouput components to the range $[0, 1]$.

Figure 10 gives a visual overview of the network architecture.

Our output is a vector of ten real numbers. The cost function in the optimization problem takes the softmax log loss form (32) with $K = 10$. We specify stochastic gradient *with momentum*, which uses a "moving average" of current

and past gradient directions. We use mini-batches of size 100 (so $m = 100$ in (17)) and set a fixed number of 45 epochs. We predefine the learning rate for each epoch: $\eta = 0.05$, $\eta = 0.005$ and $\eta = 0.0005$ for the first 30 epochs, next 10 epochs and final 5 epochs, respectively. Running on a Tesla C2075 GPU in single precision, the 45 epochs can be completed in just under 4 hours.

As an additional test, we also train the network with dropout. Here, on each stochastic gradient step, any neuron has its output re-set to zero with independent probability

- 0.15 in block 1,
- 0.15 in block 2,
- 0.15 in block 3,
- 0.35 in block 4,
- 0 in block 5 (no dropout).

We emphasize that in this case all neurons become active when the trained network is applied to the test data.

In Figure 11 we illustrate the training process in the case of no dropout. For the plot on the left, circles are used to show how the objective function (32) decreases after each of the 45 epochs. We also use crosses to indicate the objective function value on the validation data. (More precisely, these error measures are averaged over the individual batches that form the epoch—note that weights are updated after each batch.) Given that our overall aim is to assign images to one of the ten classes, the middle plot in Figure 11 looks at the percentage of errors that take place when we classify with the highest probability choice. Similarly, the plot on the right shows the percentage of cases where the correct category is not among the top five. We see from Figure 11 that the validation error starts to plateau at a stage where the stochastic gradient method continues to make significant reductions on the training error. This gives an indication that we are overfitting—learning fine details about the training data that will not help the network to generalize to unseen data.

Figure 12 shows the analogous results in the case where dropout is used. We see that the training errors are significantly larger than those in Figure 11 and the validation errors are of a similar magnitude. However, two key features in the dropout case are that (a) the validation error is below the training error, and (b) the validation error continues to decrease in sync with the training error, both of which indicate that the optimization procedure is extracting useful information over all epochs.

Figure 13 gives a summary of the performance of the trained network with no dropout (after 45 epochs) in the form of a *confusion matrix*. Here, the integer value in the general i, j entry shows the number of occasions where the network predicted category i for an image from category j. Hence, off-diagonal elements indicate mis-classifications. For example, the (1,1) element equal to 814 in Figure 13 records the number of airplane images that were correctly

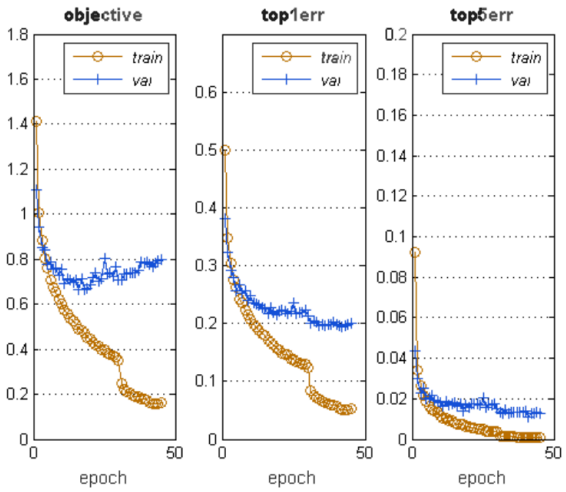

Figure 11: Errors for the trained network. Horizontal axis runs over the 45 epochs of the stochastic gradient method (that is, 45 passes through the training data). Left: Circles show cost function on the training data; crosses show cost function on the validation data. Middle: Circles show the percentage of instances where the most likely classification from the network does not match the correct category, over the training data images; crosses show the same measure computed over the validation data. Right: Circles show the percentage of instances where the five most likely classifications from the network do not include the correct category, over the training data images; crosses show the same measure computed over the validation data.

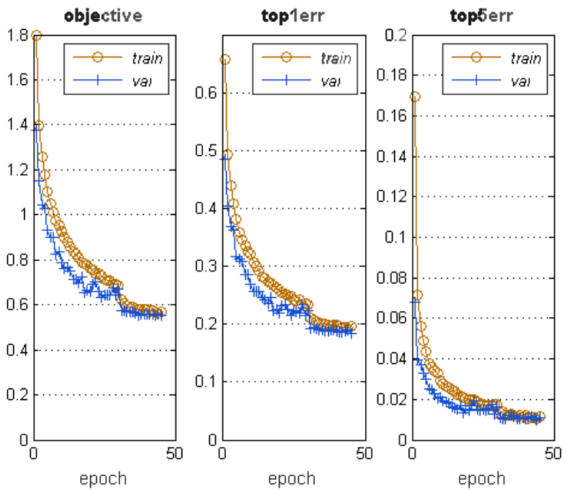

Figure 12: As for Figure 11 in the case where dropout was used.

classified as airplanes, and the (1,2) element equal to 21 records the number of automobile images that were incorrectly classified as airplanes. Below each integer is the corresponding percentage, rounded to one decimal place, given that the test data has 1,000 images from each category. The extra row, labeled "all", summarizes the entries in each column. For example, the value 81.4% in the first column of the final row arises because 814 of the 1,000 airplane images were correctly classified. Beneath this, the value 18.6% arises because 186 of these airplane images were incorrectly classified. The final column of the matrix, also labeled "all", summarizes each row. For example, the value 82.4% in the final column of the first row arises because 988 images were classified by the network as airplanes, with 814 of these classifications being correct. Beneath this, the value 17.6% arises because the remaining 174 out of these 988 airplane classifications were incorrect. Finally, the entries in the lower right corner summarize over all categories. We see that 80.1% of all images were correctly classified (and hence 19.9% were incorrectly classified).

Figure 14 gives the corresponding results in the case where dropout was used. We see that the use of dropout has generally improved performance, and in particular has increased the overall success rate from 80.1% to 81.1%. Dropout gives larger values along the diagonal elements of the confusion matrix in nine out of the ten categories.

To give a feel for the difficulty of this task, Figure 15 shows 16 images randomly sampled from those that were misclassified by the non-dropout network.

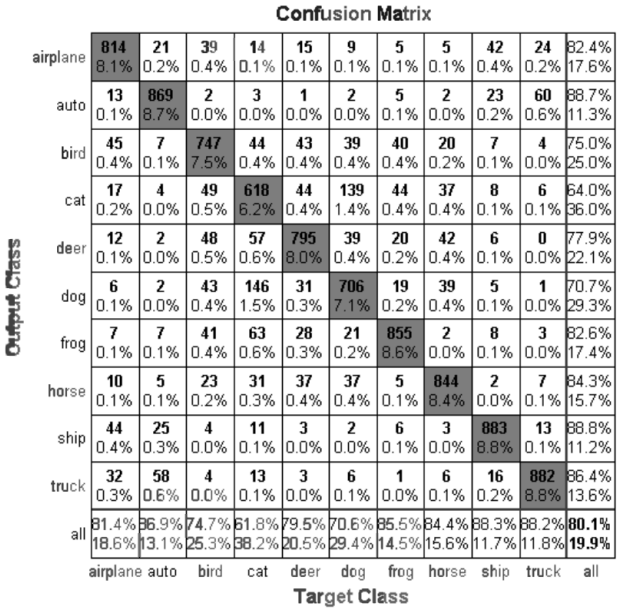

Figure 13: Confusion matrix for the the trained network from Figure 11.

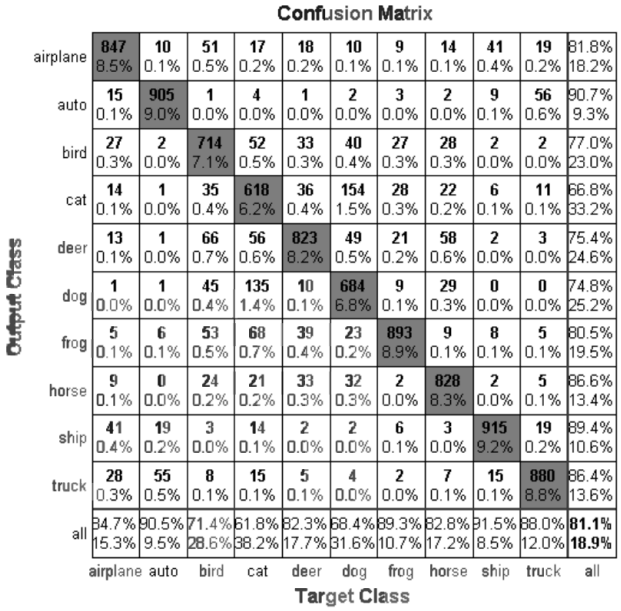

Figure 14: Confusion matrix for the the trained network from Figure 12, which used dropout.

Figure 15: Sixteen of the images that were misclassified by the trained network from Figure 11. Predicted category is indicated, with correct category shown in parentheses. Note that images are low-resolution, having 32×32 pixels.

8 Of Things Not Treated

This short introductory article is aimed at those who are new to deep learning. In the interests of brevity and accessibility we have ruthlessly omitted many topics. For those wishing to learn more, a good starting point is the free online book [26], which provides a hands-on tutorial style description of deep learning techniques. The survey [22] gives an intuitive and accessible overview of many of the key ideas behind deep learning, and highlights recent success stories. A more detailed overview of the prize-winning performances of deep learning tools can be found in [29], which also traces the development of ideas across more than 800 references. The review [35] discusses the pre-history of deep learning and explains how key ideas evolved. For a comprehensive treatment of the state-of-the-art, we recommend the book [10] which, in particular, does an excellent job of introducing fundamental ideas from computer science/discrete mathematics, applied/computational mathematics and probability/statistics/inference before pulling them all together in the deep learning setting. The recent review article [3] focuses on optimization tasks arising in machine learning. It summarizes the current theory underlying the stochastic gradient method, along with many alternative techniques. Those authors also emphasize that optimization tools must be interpreted and judged carefully when operating within this inherently statistical framework. Leaving aside the training issue, a mathematical framework for understanding the cascade of linear and nonlinear transformations used by deep networks is given in [24].

To give a feel for some of the key issues that can be followed up, we finish with a list of questions that may have occured to interested readers, along with brief answers and further citations.

Why use artificial neural networks? Looking at Figure 4, it is clear that there are many ways to come up with a mapping that divides the x-y axis into two regions; a shaded region containing the circles and an unshaded region containing the crosses. Artificial neural networks provide one useful approach. In real applications, success corresponds to a small *generalization error*; the mapping should perform well when presented with new data. In order to make rigorous, general, statements about performance, we need to make some assumptions about the type of data. For example, we could analyze the situation where the data consists of samples drawn independently from a certain probability distribution. If an algorithm is trained on such data, how will it perform when presented with *new data from the same distribution*? The authors in [15] prove that artificial neural networks trained with the stochastic gradient method can behave well in this sense. Of course, in practice we cannot rely on the existence of such a distribution. Indeed, experiments in [36] indicate that the worst case can be as bad as possible. These authors tested state-of-the-art convolutional networks for image classification. In terms of the heuristic performance indicators used to monitor the progress of the training phase, they found that the stochastic gradient method appears to work just as effectively

when the *images are randomly re-labelled*. This implies that the network is happy to learn noise—if the labels for the unseen data are similarly randomized then the classifications from the trained network are no better than random choice. Other authors have established negative results by showing that small and seemingly unimportant perturbations to an image can change its predicted class, including cases where one pixel is altered [32]. Related work in [4] showed proof-of-principle for an *adversarial patch*, which alters the classification when added to a wide range of images; for example, such a patch could be printed as a small sticker and used in the physical world. Hence, although artificial neural networks have outperformed rival methods in many application fields, the reasons behind this success are not fully understood. The survey [34] describes a range of mathematical approaches that are beginning to provide useful insights, whilst the discussion piece [25] includes a list of ten concerns.

Which nonlinearity? The sigmoid function (1), illustrated in Figure 2, and the rectified linear unit (31) are popular choices for the activation function. Alternatives include the *step function*,

$$\begin{cases} 0, & \text{for } x \leq 0, \\ 1, & \text{for } x > 0. \end{cases}$$

Each of these can undergo *saturation*: produce very small derivatives that thereby reduce the size of the gradient updates. Indeed, the step function and rectified linear unit have completely flat portions. For this reason, a *leaky rectified linear unit*, such as,

$$f(x) = \begin{cases} 0.01x, & \text{for } x \leq 0, \\ x, & \text{for } x > 0, \end{cases}$$

is sometimes preferred, in order to force a nonzero derivative for negative inputs. The back propagation algorithm described in section 5 carries through to general activation functions.

How do we decide on the structure of our net? Often, there is a natural choice for the size of the output layer. For example, to classify images of individual handwritten digits, it would make sense to have an output layer consisting of ten neurons, corresponding to $0, 1, 2, \ldots, 9$, as used in Chapter 1 of [26]. In some cases, a physical application imposes natural constraints on one or more of the hidden layers [16]. However, in general, choosing the overall number of layers, the number of neurons within each layer, and any constraints involving inter-neuron connections, is not an exact science. Rules of thumb have been suggested, but there is no widely accepted technique. In the context of image processing, it may be possible to attribute roles to different layers; for example, detecting edges, motifs and larger structures as information flows forward [22], and our understanding of biological neurons provides further insights [10]. But specific roles cannot be completely hardwired into the network design—the

weights and biases, and hence the tasks performed by each layer, emerge from the training procedure. We note that the use of back propagation to compute gradients is not restricted to the types of connectivity, activation functions and cost functions discussed here. Indeed, the method fits into a very general framework of techniques known as *automatic differentiation* or *algorithmic differentiation* [13].

How big do deep learning networks get? The AlexNet architecture [21] achieved groundbreaking image classification results in 2012. This network used 650,000 neurons, with five convolutional layers followed by two fully connected layers and a final softmax. The programme *AlphaGo*, developed by the Google DeepMind team to play the board game Go, rose to fame by beating the human European champion by five games to nil in October 2015 [30]. AlphaGo makes use of two artificial neural networks with 13 layers and 15 layers, some convolutional and others fully connected, involving millions of weights.

Didn't my numerical analysis teacher tell me never to use steepest descent? It is known that the steepest descent method can perform poorly on examples where other methods, notably those using information about the second derivative of the objective function, are much more efficient. Hence, optimization textbooks typically downplay steepest descent [9, 27]. However, it is important to note that training an artificial neural network is a very specific optimization task:

- the problem dimension and the expense of computing the objective function and its derivatives, can be extremely high,
- the optimization task is set within a framework that is inherently statistical in nature,
- a great deal of research effort has been devoted to the development of practical improvements to the basic stochastic gradient method in the deep learning context.

Currently, a theoretical underpinning for the success of the stochastic gradient method in training networks is far from complete [3]. A promising line of research is to connect the stochastic gradient method with discretizations of stochastic differential equations, [31], generalizing the idea that many deterministic optimization methods can be viewed as timestepping methods for gradient ODEs, [17]. We also note that the introduction of more traditional tools from the field of optimization may lead to improved training algorithms.

Is it possible to regularize? As we discussed in section 7, overfitting occurs when a trained network performs accurately on the given data, but cannot *generalize* well to new data. *Regularization* is a broad term that describes attempts to avoid overfitting by rewarding smoothness. One approach is

to alter the cost function in order to encourage small weights. For example, (9) could be extended to

$$\text{Cost} = \frac{1}{N}\sum_{i=1}^{N} \tfrac{1}{2}\|y(x^{\{i\}}) - a^{[L]}(x^{\{i\}})\|_2^2 + \frac{\lambda}{N}\sum_{l=2}^{L} \| W^{[l]} \|_2^2. \qquad (33)$$

Here $\lambda > 0$ is the regularization parameter. One motivation for (33) is that large weights may lead to neurons that are sensitive to their inputs, and hence less reliable when new data is presented. This argument does not apply to the biases, which typically are not included in such a regularization term. It is straightforward to check that using (33) instead of (9) makes a very minor and inexpensive change to the back propagation algorithm.

What about ethics and accountability? The use of "algorithms" to aid decision-making is not a recent phenomenon. However, the increasing influence of black-box technologies is naturally causing concerns in many quarters. The recent articles [7, 14] raise several relevant issues and illustrate them with concrete examples. They also highlight the particular challenges arising from massively-parameterized artificial neural networks. Professional and governmental institutions are, of course, alert to these matters. In 2017, the Association for Computing Machinery's US Public Policy Council released seven *Principles for Algorithmic Transparency and Accountability*[1]. Among their recommendations are that

- "Systems and institutions that use algorithmic decision-making are encouraged to produce explanations regarding both the procedures followed by the algorithm and the specific decisions that are made", and
- "A description of the way in which the training data was collected should be maintained by the builders of the algorithms, accompanied by an exploration of the potential biases induced by the human or algorithmic data-gathering process."

Article 15 of the European Union's General Data Protection Regulation 2016/679[2], which takes effect in May 2018, concerns "Right of access by the data subject," and includes the requirement that "The data subject shall have the right to obtain from the controller confirmation as to whether or not personal data concerning him or her are being processed, and, where that is the case, access to the personal data and the following information:."Item (h) on the subsequent list covers

- "the existence of automated decision-making, including profiling, referred to in Article 22(1) and (4) and, at least in those cases, meaningful information about the logic involved, as well as the significance

[1] https://www.acm.org/

[2] https://www.privacy-regulation.eu/en/15.htm

and the envisaged consequences of such processing for the data subject."

What are some current research topics? Deep learning is a fast-moving, high-bandwith field, where many new advances are driven by the needs of specific application areas and the features of new high performance computing architectures. Here, we briefly mention three hot-topic areas that have not yet been discussed.

Training a network can be an extremely expensive task. When a trained network is seen to make a mistake on new data, it is therefore tempting to fix this with a local perturbation to the weights and/or network structure, rather than re-training from scratch. Approaches for this type of *on the fly* tuning can be developed and justified using the theory of measure concentration in high dimensional spaces [12].

Adversarial networks, [11], are based on the concept that an artificial neural network may be viewed as a *generative model*: a way to create realistic data. Such a model may be useful, for example, as a means to produce realistic sentences, or very high resolution images. In the adversarial setting, the generative model is pitted against a *discriminative model*. The role of the discriminative model is to distinguish between real training data and data produced by the generative model. By iteratively improving the performance of these models, the quality of both the generation and discrimination can be increased dramatically.

The idea behind *autoencoders* [28] is, perhaps surprisingly, to produce an overall network whose output matches its input. More precisely, one network, known as the *encoder*, corresponds to a map F that takes an input vector, $x \in \mathbb{R}^s$, and produces a lower dimensional output vector $F(x) \in \mathbb{R}^t$. So $t \ll s$. Then a second network, known as the *decoder*, corresponds to a map G that takes us back to the same dimension as x; that is, $G(F(x)) \in \mathbb{R}^s$. We could then aim to minimize the sum of the squared error $\|x - G(F(x))\|_2^2$ over a set of training data. Note that this technique does not require the use of labelled data—in the case of images we are attempting to reproduce each picture without knowing what it depicts. Intuitively, a good encoder is a tool for dimension reduction. It extracts the key features. Similarly, a good decoder can reconstruct the data from those key features.

Where can I find code and data? There are many publicly available codes that provide access to deep learning algorithms. In addition to MatConvNet [33], we mention Caffe [19], Keras [5], TensorFlow [1], Theano [2] and Torch [6]. These packages differ in their underlying platforms and in the extent of expert knowledge required. Your favorite scientific computing environment may also offer a range of proprietary and user-contributed deep learning toolboxes. However, it is currently the case that making serious use of modern deep learning technology requires a strong background in numerical computing. Among the standard benchmark data sets are

the CIFAR-10 collection [20] that we used in section 7, and its big sibling CIFAR-100, ImageNet [8], and the handwritten digit database MNIST [23].

Acknowledgements

We are grateful to the MatConvNet team for making their package available under a permissive BSD license. The MATLAB code in Listings 6.1 and 6.2 can be found at

http://personal.strath.ac.uk/d.j.higham/algfiles.html

as well as an exteneded version that produces Figures 7 and 8, and a MATLAB code that uses `lsqnonlin` to produce Figure 4.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, *TensorFlow: A system for large-scale machine learning*, in 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), 2016, pp. 265–283.

[2] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky, Y. Bengio, A. Bergeron, J. Bergstra, V. Bisson, J. Bleecher Snyder, N. Bouchard, N. Boulanger-Lewandowski, X. Bouthillier, A. de Brébisson, O. Breuleux, P.-L. Carrier, K. Cho, J. Chorowski, P. Christiano, T. Cooijmans, M.-A. Côté, M. Côté, A. Courville, Y. N. Dauphin, O. Delalleau, J. Demouth, G. Desjardins, S. Dieleman, L. Dinh, M. Ducoffe, V. Dumoulin, S. Ebrahimi Kahou, D. Erhan, Z. Fan, O. Firat, M. Germain, X. Glorot, I. Goodfellow, M. Graham, C. Gulcehre, P. Hamel, I. Harlouchet, J.-P. Heng, B. Hidasi, S. Honari, A. Jain, S. Jean, K. Jia, M. Korobov, V. Kulkarni, A. Lamb, P. Lamblin, E. Larsen, C. Laurent, S. Lee, S. Lefrancois, S. Lemieux, N. Léonard, Z. Lin, J. A. Livezey, C. Lorenz, J. Lowin, Q. Ma, P.-A. Manzagol, O. Mastropietro, R. T. McGibbon, R. Memisevic, B. van Merriënboer, V. Michalski, M. Mirza, A. Orlandi, C. Pal, R. Pascanu, M. Pezeshki, C. Raffel, D. Renshaw, M. Rocklin, A. Romero, M. Roth, P. Sadowski, J. Salvatier, F. Savard, J. Schlüter, J. Schulman, G. Schwartz, I. V. Serban, D. Serdyuk, S. Shabanian, E. Simon, S. Spieckermann, S. R. Subramanyam, J. Sygnowski, J. Tanguay, G. van

Tulder, J. Turian, S. Urban, P. Vincent, F. Visin, H. de Vries, D. Warde-Farley, D. J. Webb, M. Willson, K. Xu, L. Xue, L. Yao, S. Zhang, and Y. Zhang, *Theano: A Python framework for fast computation of mathematical expressions*, arXiv e-prints, abs/1605.02688 (2016).

[3] L. Bottou, F. Curtis, and J. Nocedal, *Optimization methods for large-scale machine learning*, arXiv:1606.04838, version 2, (2017).

[4] T. B. Brown, D. Mané, A. R. M. Abadi, and J. Gilmer, *Adversarial patch*, arXiv:1712.09665 [cs.CV], (2017).

[5] F. Chollet et al., *Keras*, GitHub, (2015).

[6] R. Collobert, K. Kavukcuoglu, and C. Farabet, *Torch7: A Matlab-like environment for machine learning*, in BigLearn, NIPS Workshop, 2011.

[7] J. H. Davenport, *The debate about algorithms*, Mathematics Today, (2017), p. 162.

[8] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and F.-F. Li, *ImageNet: A large-scale hierarchical image database.*, in CVPR, IEEE Computer Society, 2009, pp. 248–255.

[9] R. Fletcher, *Practical Methods of Optimization*, Wiley, Chichester, second ed., 1987.

[10] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, MIT Press, Boston, 2016.

[11] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. C. Courville, and Y. Bengio, *Generative adversarial nets*, in Advances in Neural Information Processing Systems 27, Montreal, Canada, 2014, pp. 2672–2680.

[12] A. N. Gorban and I. Y. Tyukin, *Stochastic separation theorems*, Neural Networks, 94 (2017), pp. 255–259.

[13] A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, Society for Industrial and Applied Mathematics, Philadelphia, second ed., 2008.

[14] P. Grindrod, *Beyond privacy and exposure: ethical issues within citizen-facing analytics*, Phil. Trans. of the Royal Society A, 374 (2016), p. 2083.

[15] M. Hardt, B. Recht, and Y. Singer, *Train faster, generalize better: Stability of stochastic gradient descent*, in Proceedings of the 33rd International Conference on Machine Learning, 2016, pp. 1225–1234.

[16] C. F. Higham, R. Murray-Smith, M. J. Padgett, and M. P. Edgar, *Deep learning for real-time single-pixel video*, Scientific Reports, (to appear).

[17] D. J. Higham, *Trust region algorithms and timestep selection*, SIAM Journal on Numerical Analysis, 37 (1999), pp. 194–210.

[18] D. J. Higham and N. J. Higham, *MATLAB Guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, third ed., 2017.

[19] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, *Caffe: Convolutional architecture for fast feature embedding*, arXiv preprint arXiv:1408.5093, (2014).

[20] A. Krizhevsky, *Learning multiple layers of features from tiny images*, tech. rep., 2009.

[21] A. Krizhevsky, I. Sutskever, and G. E. Hinton, *Imagenet classification with deep convolutional neural networks*, in Advances in Neural Information Processing Systems 25, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds., 2012, pp. 1097–1105.

[22] Y. LeCun, Y. Bengio, and G. Hinton, *Deep learning*, Nature, 521 (2015), pp. 436–444.

[23] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, *Gradient-based learning applied to document recognition*, Proceedings of the IEEE, 86 (1998), pp. 2278–2324.

[24] S. Mallat, *Understanding deep convolutional networks*, Philosophical Transactions of the Royal Society of London A, 374 (2016), p. 20150203.

[25] G. Marcus, *Deep learning: A critical appraisal*, arXiv:1801.00631 [cs.AI], (2018).

[26] M. Nielsen, *Neural Networks and Deep Learning*, Determination Press, 2015.

[27] J. Nocedal and S. J. Wright, *Numerical Optimization*, Springer, Berlin, second ed., 2006.

[28] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1*, MIT Press, Cambridge, MA, USA, 1986, ch. Learning Internal Representations by Error Propagation, pp. 318–362.

[29] J. Schmidhuber, *Deep learning in neural networks: An overview*, Neural Networks, 61 (2015), pp. 85–117.

[30] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, *Mastering the game of Go with deep neural networks and tree search*, Nature, 529 (2016), pp. 484–489.

[31] J. Sirignano and K. Spiliopoulos, *Stochastic gradient descent in continuous time*, SIAM J. Finan. Math., 8 (2017), pp. 933–961.

[32] J. Su, D. V. Vargas, and S. Kouichi, *One pixel attack for fooling deep neural networks*, arXiv:1710.08864 [cs.LG], (2017).

[33] A. Vedaldi and K. Lenc, *MatConvNet: Convolutional neural networks for MATLAB*, in ACM International Conference on Multimedia, Brisbane, 2015, pp. 689–692.

[34] R. Vidal, R. Giryes, J. Bruna, and S. Soatto, *Mathematics of deep learning*, Proc. of the Conf. on Decision and Control (CDC), (2017).

[35] H. Wang and B. Raj, *On the origin of deep learning*, arXiv:1702.07800 [cs.LG], (2017).

[36] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, *Understanding deep learning requires rethinking generalization*, in 5th International Conference on Learning Representations, 2017.