NEURAL NETWORKS

a Monograph

Paul F. Roysdon, Ph.D.

Neural Networks

a Monograph

Paul F. Roysdon, Ph.D.

NEURAL NETWORKS - A MONOGRAPH

First Edition

ISBN 9781656067814

The text for this book was formated in *LATEX* and the mathematics was formated in $\mathcal{AMS}$-*LATEX* (Donald Knuth's *TEXtext* formatting system) and converted from device-independent to postscript format using `DVIPS`.

To my Applied Mathmatics & Data Science colleagues, thank you.

Contents

Preface

Purpose. This text is the ensemble of notes gathered from my research on Machine Learning (ML) topics. Fortunately, there exists a large body of literature on this subject. However, much of the on-line material is confusing or inaccurate, with inconsistent nomenclature. This text is a simple tutorial of a basic Artificial Neural Network, with worked examples in *Python*, so that anyone can read this text and gain the skills necessary to:

- Perform research of new and exciting literature in ML.
- Apply their *Python* skills to a problem in their field of work.

Audience. A reasonably prepared college graduate should be able to understand and apply all contents of this book.

Prerequisites. None. This text and the accompanying software are for students interested in ML and DS. Most professionals have seen this material in high school or college. If some of the material is new, don't worry, this text will walk you through the necessary steps.

Computing & Associated Software. It is good to be able to solve small problems by hand, but in practice they are large, requiring a computer for their solution. Therefore, to fully appreciate the subject, one needs to solve large (practical) problems on a computer. An important feature of this book is that it comes with software implementing the major algorithms described herein.

This text uses *Python*, specifically *Anaconda* and the *Spyder IDE*. *Spyder* is a nice tool because the user can write scripts and test them in a single application We will use several add-on numerical computing packages with *Python*, e.g. *TensorFlow*, *NumPy*, *SciPy*, *SciKit Learn*. We will use these tools extensively in our examples, so it is suggested that the student installs and familiarize themselves with the tools. You will not be required to be proficient in Python, rather it is expected that you will learn tips and tricks throughout this text.

To install *Spyder* (with *Python* v.3.7), visit `https://www.anaconda.com/distribution/` and download the *Windows 64-Bit Graphical Installer*, or other version of software as appropriate for you hardware. Once installed, open *Anaconda*, launch *Spyder*, and install the following packages by typing the commands in the *console window*. The following list will install most ML packages:

- `pip install --upgrade pip`
- `pip install --upgrade setuptools`
- `pip install opencv-python`

- `pip install theano`
- `pip install tensorflow==1.13.1`
- `pip install keras`
- `pip install matplotlib`
- `pip install pandas`
- `pip install sklearn`
- `pip install -U gensim`
- `pip install pydotplus`

The following steps are needed for *PyTorch* in *Anaconda*:

- `conda update -n base -c defaults conda`
- `conda install PyTorch -c PyTorch`
- `conda install pytorch torchvision -c pytorch`
- `conda list pytorch`

Finally use the shortcut "Ctrl+" to restart *Python* kernel. While it is advisable to use the latest version of each package, *TensorFlow* v.1.13.1 is necessary for compatibility of the tutorial code provided.

Great pains have been taken to make the source code for these programs readable. In particular, the names of the variables in the programs are consistent with the notation of this book. The software can be downloaded from the following web site: `https://github.com/AidedNav/books`

Features. Here are some features that distinguish this text from others:

- The text gives a balanced treatment to both the traditional and newer methods. The notation and analysis is developed to be consistent across the methods.

- From the beginning and consistently throughout the text, example problems are formulated and solved to develop an understanding of the equations and their application to Neural Networks. By highlighting this throughout, it is hoped that the reader will more fully understand and appreciate theory behind Neural Networks.

- There is an extensive treatment of modern methods, including numerically fast solutions systems of equations.

Paul F. Roysdon, Ph.D.
January 5, 2020

Chapter 1

What are Artificial Neural Networks?

An artificial neural network (ANN) is a numerical implementation of the biological brain. Both "systems" are analogous to switches that change their output state based on the strength of the input. While a single switch, or neuron, produces a single output, the interconnection of thousands or millions of neurons represents a structured output. Through *learning*, some neurons are *triggered* differently than others, and the repetition of learning, reinforces those connections and structure. The reinforcement process to produce a desired result is called *feedback*.

The *Artificial* neural networks attempt to simplify and mimic the biological behavior discussed above. *Training* an ANN takes two forms, *supervised* or *unsupervised*. In a *supervised* ANN, the network is trained by providing matched input and output data samples, minimizing the error between. For example: an e-mail spam filter might use specific text within an email to determine the authenticity of the e-mail, and the training of the ANN filter requires many examples, with iterations of feedback, before it will correctly *classify* an e-mail. An *unsupervised* ANN attempts to "understand" the structure of the input "on its own", requiring special *clustering* or *dimension reduction* algorithms; this method will be discussed in a future tutorial.

Chapter 2

Structure of a Neural Network

2.1 The Artificial Neuron

The biological neuron is simulated in an ANN by an *activation function* (AF), or switch. If the input is above a user defined threshold, the AF switches state, e.g. from 0 to 1, -1 to 1 or from 0 to > 0. A commonly used activation function is the sigmoid function,

$$f(z) = \frac{1}{1 + \exp(-z)}.$$

Graphically, the sigmoid function is shown in Figure 2.1.

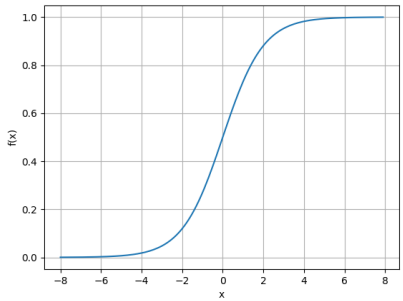

Figure 2.1: The artificial neuron.

The sigmoid function can be written and plotted numerically using Python:

```
import matplotlib.pylab as plt
import numpy as np
x = np.arange(-8, 8, 0.1)
f = 1 / (1 + np.exp(-x))
plt.plot(x, f)
plt.xlabel('x')
plt.ylabel('f(x)')
plt.show()
```

Listing 2.1: The artificial neuron.

In Machine Learning (ML) literature we say that the function is "activated", i.e. it moves from 0 to 1, when the input is greater than some threshold. It is desirable to have a smooth transition between states, as we will see in the following sections. Additionally it is required to have an algebraically smooth function – a function with a derivative at every point – which is required to train the algorithm, as discussed in Section 4.4.

2.2 Nodes

Analogous to biological neurons, ANN's have a similar structure with the output of one neuron connected to the input of another neuron. We represent these networks as connected layers of *nodes*, where each node has many "weighted" inputs from previous nodes. The output of each node is the result of an AF applied to each input, followed by the sum of the AF's, multiplied by the weight of that node. For three inputs plus a weight, the graphical representation of a node is shown in Figure 2.2.

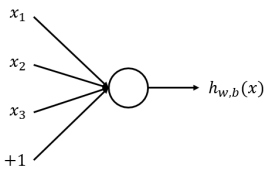

Figure 2.2: The Node Diagram.

The circle in Figure 2.2 represents the node, with inputs x_1, x_2, x_3, weight of +1, and output $h_{w,b}(x)$. In some literature this diagram is called a *perceptron*. Mathematically, Figure 2.2 is

$$x_1 w_1 + x_2 w_2 + x_3 w_3 + b$$

where x_i are the inputs $[x_1, x_2, x_3]$, w_i are weights $[w_1, w_2, w_3]$, and b is the bias.

The weights are scalar values applied to each input, and while randomly initialized, are "optimized" during the learning process so that the structure of all nodes produce the desired "learned" result. The bias increases the flexibility of the node during the learning process.

2.3 Bias

Consider a simple node with only one input and one output (see Figure 2.3). The input to the activation function of the node in this case is simply $x_1 w_1$. Changing w_1 changes the *threshold* above which the AF changes state, as illustrated in Figure 2.4. This can be programmed in Python, as shown below.

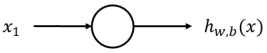

Figure 2.3: The Simple Diagram.

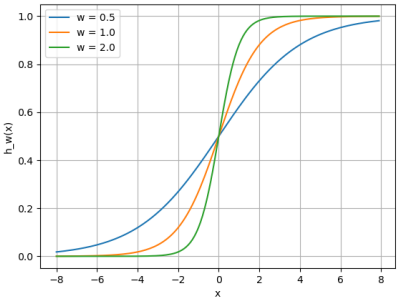

Figure 2.4: Effect of adjusting weights.

```
import matplotlib.pylab as plt
import numpy as np
x = np.arange(-8, 8, 0.1)
f = 1 / (1 + np.exp(-x))
w1 = 0.5
w2 = 1.0
w3 = 2.0
l1 = 'w = 0.5'
l2 = 'w = 1.0'
l3 = 'w = 2.0'
```

```
for w, l in [(w1, l1), (w2, l2), (w3, l3)]:
    f = 1 / (1 + np.exp(-x*w))
plt.plot(x, f, label=l)
plt.xlabel('x')
plt.ylabel('h_w(x)')
plt.legend(loc=2)
plt.show()
```

Listing 2.2: Effect of Adjusting Weights.

Notice that changing the weight, changes the slope of the output of the AF, and is useful when modeling strengths of relationships between input and output variables. However, a bias can be used if we only want the output to change when x is greater than 1. Consider the simple network of Figure 2.3, with the addition of a bias input shown in Figure 2.5.

Adjusting the bias b, translates the AF along the x-axis, resulting in a change of activation (see Figure 2.6, and the Python code below). Therefore, by adding a bias term, you can make the node simulate a generic *if* function, i.e. *if* $(x > z)$ *then* 1 *else* 0. Without a bias term, you are unable to vary z, and the function will be always near 0. This is simple example demonstrates the need for a bias to simulate conditional relationships.

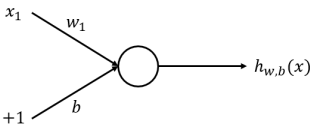

Figure 2.5: Effect of bias.

```
import matplotlib.pylab as plt
import numpy as np
x = np.arange(-8, 8, 0.1)
f = 1 / (1 + np.exp(-x))
w = 5.0
b1 = -8.0
b2 = 0.0
b3 = 8.0
l1 = 'b = -8.0'
l2 = 'b = 0.0'
l3 = 'b = 8.0'
for b, l in [(b1, l1), (b2, l2), (b3, l3)]:
    f = 1 / (1 + np.exp(-(x*w+b)))
```

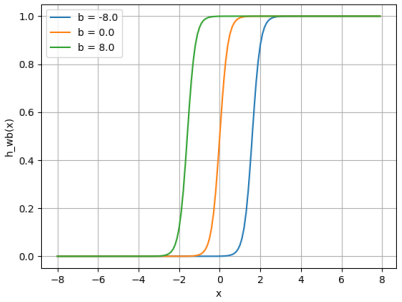

Figure 2.6: Effect of bias.

```
plt.plot(x, f, label=l)
plt.xlabel('x')
plt.ylabel('h_wb(x)')
plt.legend(loc=2)
plt.show()
```

Listing 2.3: Effect of Bias.

2.4 A Simple ANN Structure

Figures 2.2 -2.5 provide the structure of node-weight-bias in an ANN. While there are many forms of interconnected nodes in ANN's, a simple ANN consists of an *input layer*, a *hidden layer* and an *output layer*, as shown in Figure 2.7

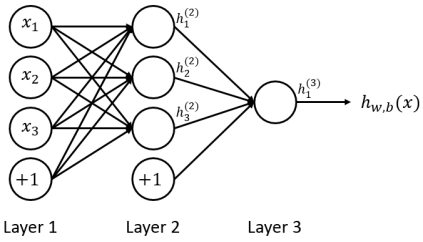

Figure 2.7: Three layer neural network.

Figure 2.7 has three *layers*:

- Layer 1 – **input layer** – where the external input data enters the network.
- Layer 2 – **hidden layer** – as this layer is not part of the input or output. Note: most neural networks can have many hidden layers, however, for simplicity we have only included one.
- Layer 3 – **output layer** – the result of the ANN.

To simplify notation, use L1 for Layer 1, N1 for node 1, etc. As shown, each node in L1 has a connection to all nodes in L2. Likewise, the nodes in L2 each connect to the single output node L3. Each of these connections have an associated weight.

2.5 Notation

The maths in the following chapters require precise notation to accurately track each step. We adopt herein the notation from the Stanford *Deep Learning Tutorial*.

Weights use the notation $w_{ij}^{(l)}$, i refers to the node number of the connection in layer $l+1$, and j refers to the node number of the connection in layer l. Take special note of this order. For example: the notation for the connection between N1 in L1 and N2 in L2 is $w_{21}^{(1)}$. This notation may seem a bit odd, as you would expect the i and j to refer the node numbers in layers l and $l+1$ respectively (i.e. in the direction of input to output), rather than the opposite. However, this notation makes more sense when you consider the bias.

As shown in Figure 2.7, the (+1) bias is connected to each of the nodes in the subsequent layer. So the bias in L1 is connected to the all the nodes in L2. Because the bias is not a true node with an activation function, it has no inputs. The bias notation is $b_i^{(l)}$, where i is the node number in the layer $l+1$ – the same as used for the normal weight notation $w_{21}^{(1)}$. Continuing the previous example; the weight on the connection between the bias in L1 and N2 in L2 is $b_2^{(1)}$.

Note: the values, $w_{ij}^{(l)}$ and $b_i^{(l)}$, are computed and optimized during the training phase of the ANN development process.

The output node notation is $h_j^{(l)}$, where j denotes the node number in layer l of the network. Continuing the previous example; the output of N2 in L2 is $h_2^{(2)}$.

Chapter 3

The Feed-Forward Process

In the context of ML, there are two processes that are commonly confused, *feed-forward* and *feed-back* (or *back-propagation*).

- Feed-forward is the process of computing the output of an ANN when the weights and biases of the nodes are known. This is the process used *after* training to classify input data.
- Feed-back (or back-propagation) is the process of determining, or optimizing, the weights and biases of the ANN. This is the process used *during* training to minimize the error between the known input and desired output.

3.1 A Simple Feed-Forward Example

To illustrate the feed-forward process, the diagram in Figure 2.7 can be represented mathematically as

$$h_1^{(2)} = f\left(w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2 + w_{13}^{(1)}x_3 + b_1^{(1)}\right) \tag{3.1}$$

$$h_2^{(2)} = f\left(w_{21}^{(1)}x_1 + w_{22}^{(1)}x_2 + w_{23}^{(1)}x_3 + b_2^{(1)}\right) \tag{3.2}$$

$$h_3^{(2)} = f\left(w_{31}^{(1)}x_1 + w_{32}^{(1)}x_2 + w_{33}^{(1)}x_3 + b_3^{(1)}\right) \tag{3.3}$$

and the output at L3 as

$$h_{W,b}(x) = h_1^{(3)} = f\left(w_{11}^{(2)}h_1^{(2)} + w_{12}^{(2)}h_2^{(2)} + w_{13}^{(2)}h_3^{(2)} + b_1^{(2)}\right). \tag{3.4}$$

The function, $f(\cdot)$, refers to the node activation function, e.g. the sigmoid function. The first line, $h_1^{(2)}$ is the output of the N1 in L2 with inputs $w_{11}^{(1)}x_1$, $w_{12}^{(1)}x_2$, $w_{13}^{(1)}x_3$, $b_1^{(1)}$. The inputs are summed, then passed to the AF at N1 in L2. This process is repeated for nodes N2 and N3 in L2. The last line is the output of L3, which uses the same process of summation prior to the AF. However, rather than taking the weighted input variables (x_1, x_2, x_3), the final node takes as input the weighted output of the nodes of the L2, $(h_1^{(2)}, h_2^{(2)}, h_3^{(2)})$, plus the weighted bias. The result is a hierarchical structure of an ANN.

3.2 Feed-Forward using Linear Algebra

To simplify equations 3.1-3.4, we use Linear Algebra and express the scalar values as a vector (a set of scalars) or matrix (a set of vectors). Let $z_i^{(l)}$ be the summed input of node i of layer l, including the bias term. Equation 3.1 can be expressed as

$$\begin{aligned} z_1^{(2)} &= w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2 + w_{13}^{(1)}x_3 + b_1^{(1)} \\ &= \sum_{j=1}^{3} w_{ij}^{(1)}x_i + b_i^{(1)} \end{aligned}$$

for N1 in L2.

Continuing the example, we can represent equations 3.1-3.4 for L2 as

$$\begin{aligned} \mathbf{z}^{(2)} &= \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \\ \mathbf{h}^{(2)} &= f\left(\mathbf{z}^{(2)}\right) \\ \mathbf{z}^{(3)} &= \mathbf{W}^{(2)}\mathbf{h}^{(2)} + \mathbf{b}^{(2)} \\ \mathbf{h}_{\mathbf{W},\mathbf{b}}(\mathbf{x}) &= \mathbf{h}^{(3)} = f\left(\mathbf{z}^{(3)}\right) \end{aligned}$$

where $\mathbf{W}$ is the matrix of weights, $\mathbf{z}$ is a vector, $\mathbf{h}$ is a vector of node outputs, $\mathbf{b}$ is a vector of biases, and $\mathbf{x}$ is the vector of nodes.

The generalized matrix-vector form is:

$$\begin{aligned} \mathbf{z}^{(l+1)} &= \mathbf{W}^{(l)}\mathbf{h}^{(l)} + \mathbf{b}^{(l)} \\ \mathbf{h}^{(l+1)} &= f\left(\mathbf{z}^{(l+1)}\right) \end{aligned}$$

Here we can see the general feed forward process, where the output of layer l becomes the input to layer $l+1$. We know that $\mathbf{h}^{(1)}$ is simply the input layer $\mathbf{x}$ and $\mathbf{h}^{(n_l)}$ is the output of the final output layer, where n_l is

the number of layers in the network. Notice we have dropped references to the node numbers i and j, as the use of linear algebra simplifies this notation to vector notation. This simplification is necessary to leverage fast linear algebra routines in *Python* (and other languages) rather than using *for*-loops.

To illustrate the numerical implementation, consider the vector $\mathbf{z}^{(l+1)} = \mathbf{W}^{(l)}\mathbf{h}^{(l)} + \mathbf{b}^{(l)}$ for the input layer (i.e. $\mathbf{h}^{(l)} = \mathbf{x}$). The simple linear algebra steps are

$$\begin{aligned}
\mathbf{z}^{(1)} &= \begin{pmatrix} w_{11}^{(1)} & w_{12}^{(1)} & w_{13}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} & w_{23}^{(1)} \\ w_{31}^{(1)} & w_{32}^{(1)} & w_{33}^{(1)} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} + \begin{pmatrix} b_1^{(1)} \\ b_2^{(1)} \\ b_3^{(1)} \end{pmatrix} \\
&= \begin{pmatrix} w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2 + w_{13}^{(1)}x_3 \\ w_{21}^{(1)}x_1 + w_{22}^{(1)}x_2 + w_{23}^{(1)}x_3 \\ w_{31}^{(1)}x_1 + w_{32}^{(1)}x_2 + w_{33}^{(1)}x_3 \end{pmatrix} + \begin{pmatrix} b_1^{(1)} \\ b_2^{(1)} \\ b_3^{(1)} \end{pmatrix} \\
&= \begin{pmatrix} w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2 + w_{13}^{(1)}x_3 + b_1^{(1)} \\ w_{21}^{(1)}x_1 + w_{22}^{(1)}x_2 + w_{23}^{(1)}x_3 + b_2^{(1)} \\ w_{31}^{(1)}x_1 + w_{32}^{(1)}x_2 + w_{33}^{(1)}x_3 + b_3^{(1)} \end{pmatrix}.
\end{aligned}$$

Recall matrix-vector multiplication, the *dot product* states that each element in the *row* of the weight matrix is multiplied by each element in the single *column* of the input vector, then summed to create a new (3×1) vector. Then simply add the bias vector to achieve the final result. Finally, the AF is applied element-wise (to each row separately in the $\mathbf{z}^{(1)}$).

Using the *NumPy* library, the above calculations are demonstrated in the following *Python* functions:

```
def f(z):
    return 1 / (1 + np.exp(-z))
```

Listing 3.1: Activation Function.

```
def matrix_feed_forward_calc(n_layers, x, w, b):
    for l in range(n_layers-1):
        if l == 0:
            node_in = x
        else:
            node_in = h
        z = w[l].dot(node_in) + b[l]
        h = f(z)
    return h
```

Listing 3.2: Feed-Forward Function.

Note the correct use of the dot product function $a.dot(b)$ in line 7. The incorrect use of scalar multiplication, e.g. $a * b$, will produce the wrong result.

While it is possible to perform ANN multiplication using several "nested" *for*-loops, the result is computationally slower, by a factor of 10^3. Leveraging linear algebra libraries like *NumPy* are necessary on large ANN's with thousands or millions of nodes. In fact, deep learning packages such as *TensorFlow* and *Theano* utilize your computer's GPU (rather than the CPU), as the GPU is designed for linear algebra calculations. The GPU provides computationally faster, by a factor of 10^6, results for a given ANN.

Chapter 4

Training a Neural Network

In *supervised learning*, the weights and biases of an ANN are computed by minimizing the error between the input and the desired output. To train an ANN model, several independent input-output pairs are needed. We can specify these input-output pairs as $\{(x^{(1)}y^{(1)}), \ldots, (x^{(n)}y^{(n)})\}$ where n is the number of training samples available to train the weights of the network. Each of these inputs, or outputs, can be expressed as vectors, e.g. $\mathbf{x}^{(m)}$ for some m-dimensional series of values.

For example: training a spam-detection neural network, $x^{(1)}$ could be a count of all the different significant words in an e-mail e.g.:

$$x^{(1)} = \begin{pmatrix} \text{Number of “Prince”} \\ \text{Number of “Nigeria”} \\ \vdots \\ \text{Number of “Swiss bank”} \end{pmatrix} = \begin{pmatrix} 2 \\ 2 \\ \vdots \\ 1 \end{pmatrix}$$

where $y^{(1)}$ is a scalar value, 1 or 0, to designate whether or not the e-mail is spam.

The goal of training an ANN with (x, y) pairs is to increase the likelihood of predicting the correct result y given x. This is performed by varying the weights to minimize the error. Minimizing the error in ML is commonly performed using a method called **gradient descent**, where the ANN is represented by some *cost* or *error function* $\mathcal{J}(w)$, and

a *gradient* is computed at some initial weight to determine the direction of the next step, where the cost is re-computed iteratively until a global cost minimum is reached (see Figure 4.1).

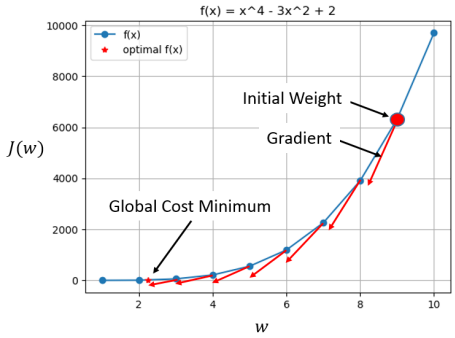

Figure 4.1: Simple, one-dimensional gradient descent.

From Calculus we know that the gradient (the first-derivative) of a function is the "slope" of that function evaluated at a desired point. For gradient descent, we compute the gradient to find the direction of *negative*-slope, so that the algorithm will "move" in the direction of the minimum. The magnitude of the gradient is the "steepness" of the slope, and thereby dictates the step size and direction. Numerically we can adjust step-size to increase or decrease the "speed" at which the algorithm approaches the minimum. Algebraically, gradient descent is

$$w_{new} = w_{old} - \alpha \times \nabla error,$$

where w_{new} is the new w position, w_{old} is the current w position, $\nabla error$ is the gradient of the error at w_{old} and α is the step size.

If the step-size α is too large, the algorithm will "bounce" between the left and right sides of the minimum, and likely diverge. A step-size that is too small will result in small improvements to the error, possibly without ever reaching the minimum. Often the step-size can be tuned, and it depends on the cost function.

While in most cases a global minimum exists, reaching the exact minimum may be computationally expensive. Instead, a "stopping criteria" can be used to terminate the gradient descent if the error or rate-of-change-of-error is below a user-defined threshold.

These concepts are illustrated in the following section.

4.1 A One-Dimensional Gradient Descent Example

Consider the polynomial $f(x) = x^4 - 3x^3 + 2$. From Calculus, we know that the gradient at any point x of $f(x)$ is found by computing the first-derivative, e.g. $f'(x) = 4x^3 - 9x^2$. The minimum of this function is found by solving for x when $f'(x) = 0$, i.e. $x = 2.25$. In Optimization literature, $f'(x)$ is called a *cost function*. Both the function, and its minimum are shown in Figure 4.2, and the method to find the minimum is shown numerically with a *Python* example.

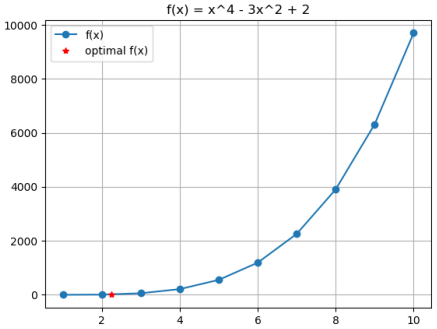

Figure 4.2: Simple, one-dimensional gradient descent.

```
x_old = 0 # The initial value does not matter
x_new = 6 # The algorithm starts at x=6
gamma = 0.01 # step size
threshold = 0.00001

def df(x):
    y = 4 * x**3 - 9 * x**2
    return y

while abs(x_new - x_old) > threshold:
    x_old = x_new
    x_new += -gamma * df(x_old)

print("The local minimum occurs at %f" % x_new)
```

Listing 4.1: Simple Gradient Function.

This *Python* function prints "The local minimum occurs at 2.249965", which agrees with the exact solution within the precision.

This example implements the **gradient descent** algorithm through the iteration of the function $f(x)$, using its derivative $f'(x)$, starting from a random point x_{new}, and iterating to a user-defined threshold. Mathematically we represent this as

$$x_{new} = x_{old} - \gamma \cdot \nabla f \tag{4.1}$$

where $\nabla f = f'(x)$ is the *gradient* of the function $f(x)$ and can be thought of as the error at each iteration, and the iteration step-size is γ.

While in one-dimension, the gradient descent algorithm is obvious, in higher-dimensions, like the multi-dimensional ANN, these methods are not as obvious. To train our ANN, we will use a common method called **back-propagation**.

4.2 The Cost Function

In Section 4.1, we introduced the concept of a cost function $f(x)$ to minimize the error for a single variable x. To minimize the cost over many variables, as is the case for an ANN, we use a method called the sum-of-squares error (SSE). The SSE applies the L_2-norm, represented by the function $\|\cdot\|$, and is *squared* to increase the values furthest from the mean, e.g. outliers, so that a larger weight may be applied to those errors. Geometrically, this would find the geometric distance between each error, and seek to minimize the result. The SSE for the single training pair (x^z, y^z) of the z^{th} training sample for the cost function $\mathcal{J}$ is

$$\begin{aligned}\mathcal{J}(\mathbf{w}, \mathbf{b}, \mathbf{x}, \mathbf{y}) &= \frac{1}{2}\|\mathbf{y}^z - \mathbf{h}^{(n_l)}(\mathbf{x}^z)\|^2 \\ &= \frac{1}{2}\|\mathbf{y}^z - \mathbf{y}_{pred}(\mathbf{x}^z)\|^2,\end{aligned} \tag{4.2}$$

where $h^{(n_l)}$ is the output of the final layer of the neural network, i.e. the output of the neural network. Because $h^{(n_l)}$ is the output, we can rewrite this as y_{pred} to highlight the prediction of the neural network given the input x^z. Instead of a single element, where the absolute error might be used, e.g. $abs(y_{pred,i}(x_i^z) - y_i^z)$, we compute the SSE using vectors. The constant $\frac{1}{2}$ is used to simplify the expression when we differentiate the cost function.

Note that Equation 4.2 is for a single (x, y) training pair. To minimize the cost function over all m training pairs we want to find the minimum

mean squared error (MSE), such that

$$\begin{aligned}\mathcal{J}(\mathbf{w},\mathbf{b}) &= \frac{1}{m}\sum_{z=0}^{m}\frac{1}{2}\|\mathbf{y}^z - \mathbf{h}^{(n_l)}(\mathbf{x}^z)\|^2 \\ &= \frac{1}{m}\sum_{z=0}^{m} J\left(\mathbf{w},\mathbf{b},\mathbf{x}^{(z)},\mathbf{y}^{(z)}\right) \end{aligned} \tag{4.3}$$

In a later section we will apply the cost function $\mathcal{J}$ above to train the weights of our network.

4.3 Gradient Descent

Gradient descent for each, scalar, weight $w_{(ij)}^{(l)}$ and bias $b_{(ij)}^{(l)}$ of the ANN, has the general form

$$w_{(ij)}^{(l)} = w_{(ij)}^{(l)} - \alpha\frac{\partial}{\partial w_{(ij)}^{(l)}}\mathcal{J}(w,b)$$

$$b_{(i)}^{(l)} = b_{(i)}^{(l)} - \alpha\frac{\partial}{\partial b_{(i)}^{(l)}}\mathcal{J}(w,b)$$

where $\mathcal{J}(w,b)$ represents the cost function. Similar to Eqn. 4.1, the above equation implements the gradient descent algorithm as

$$w_{new} = w_{old} - \alpha \times \nabla error.$$

Again, an iterative process is performed, whereby the weights are updated in each iteration.

Notice the values $\frac{\partial}{\partial w_{(ij)}^{(l)}}$ and $\frac{\partial}{\partial b_{(i)}^{(l)}}$. These are *partial derivatives* of the cost function, meaning that they are a derivative of the cost function *with respect to* (w.r.t.) the variable that they are trying to optimize. Notice that the cost function $\mathcal{J}(w,b)$, is a function of both w and b. Therefore the partial derivative is used to find the *slope* of the *error or cost* w.r.t. the variable of that cost, e.g. the cost w.r.t. b is $\frac{\partial}{\partial b_{(i)}^{(l)}}$. These partial derivatives highlight the complexity of computing multi-dimensional cost functions.

4.4 Back-Propagation

In this section, and subsequent sections, we will review the math of back-propagation. While tedious, the math is invaluable to understand the key ideas in *deep learning*, instead of being a "code monkey" who doesn't understand how the code actually works.

First recall the foundational equations from Section 3.2 for the three-layer neural network in Figure 2.7. The output of this neural network can be calculated by

$$h_{\mathbf{W},\mathbf{b}}(x) = h_1^{(3)} = f\left(w_{11}^{(2)}h_1^{(2)} + w_{12}^{(2)}h_2^{(2)} + w_{13}^{(2)}h_3^{(2)} + b_1^{(2)}\right),$$

which simplifies to $h_1^{(3)} = f(z_1^{(2)})$ by defining $z_1^{(2)}$, where

$$z_1^{(2)} = w_{11}^{(2)}h_1^{(2)} + w_{12}^{(2)}h_2^{(2)} + w_{13}^{(2)}h_3^{(2)} + b_1^{(2)}.$$

Note: throughout this section, to simplify notation, we will assume that we are operating on scalar, not vectors.

To compute the change in weight $w_{12}^{(2)}$ relative to the cost $\mathcal{J}$, evaluate $\frac{\partial J}{\partial w_{12}^{(2)}}$. To compute this derivative, we first must apply the *chain-rule* from Calculus:

$$\frac{\partial \mathcal{J}}{\partial w_{12}^{(2)}} = \frac{\partial \mathcal{J}}{\partial h_1^{(3)}} \frac{\partial h_1^{(3)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial w_{12}^{(2)}}. \tag{4.4}$$

Evaluating the last term in Equation 4.4

$$\begin{aligned}
\frac{\partial z_1^{(2)}}{\partial w_{12}^{(2)}} &= \frac{\partial}{\partial w_{12}^{(2)}}\left(w_{11}^{(1)}h_1^{(2)} + w_{12}^{(1)}h_2^{(2)} + w_{13}^{(1)}h_3^{(2)} + b_1^{(3)}\right) \\
&= \frac{\partial}{\partial w_{12}^{(2)}}\left(w_{12}^{(1)}h_2^{(2)}\right) \\
&= h_2^{(2)}.
\end{aligned}$$

Recall from Calculus that, in this instance, the partial derivative of $z_1^{(2)}$ with respect $w_{12}^{(2)}$ only operates on one term within the parentheses, $w_{12}^{(1)}h_2^{(2)}$. The result is $h_2^{(2)}$, which is simply the output of N2 in L2.

The partial derivative of the middle term, $\frac{\partial h_1^{(3)}}{\partial z_1^{(2)}}$, is the partial derivative of the AF of the $h_1^{(3)}$ output node. Recall in Section 2.1 that we require the AF to be a "smooth" function and therefore differentiable. This requirement is demonstrated here. The derivative for the sigmoid function is

$$\frac{\partial h}{\partial z} = f'(z) = f(z)(1 - f(z)),$$

where $f(z)$ is the activation function.

To compute the partial derivative of the first term, $\frac{\partial J}{\partial h_1^{(3)}}$, in Equation 4.4, we use the MSE of the cost function

$$\mathcal{J}(w, b, x, y) = \frac{1}{2}\|y_1 - h_1^{(3)}(z_1^{(2)})\|^2.$$

Define y_1 as the training target for the output node. Applying the chain-rule,

$$\frac{\partial \mathcal{J}}{\partial h} = \frac{\partial \mathcal{J}}{\partial u}\frac{\partial u}{\partial h},$$

where $u = \|y_1 - h_1^{(3)}(z_1^{(2)})\|$ and $\mathcal{J} = \frac{1}{2}u^2$, then

$$\frac{\partial \mathcal{J}}{\partial h} = -\left(y_1 - h_1^{(3)}\right).$$

To simplify terms, using the above results, let

$$\delta_i^{(n_l)} = -\left(y_i - h_i^{(n_l)}\right) \cdot f'\left(z_i^{(n_l)}\right),$$

where i is the node number of the output layer. Note: in this example there is only one layer, therefore $i = 1$.

The partial derivative for the cost function simplifies to

$$\frac{\partial}{\partial W_{ij}^{(l)}}\mathcal{J}(W, b, x, z) = h_j^{(l)}\delta_i^{(l+1)}, \tag{4.5}$$

where, the output layer $l = 2$ and i remains the node number.

4.5 Propagating Hidden Layers

At the output layer, the derivative $\frac{\partial \mathcal{J}}{\partial h} = -\left(y_i - h_i^{(n_l)}\right)$ is computed because the cost function can be directly calculated by comparing the output layer to the training data. However in the hidden layer, a direction relation to the cost function is not apparent. Here, both weights and biases are embedded deep within the neural network. Thus, we use the *back-propagation* method.

To propagate $\delta_i^{(n_l)}$ backward through the network, we again use the chain rule. To illustrate, consider node j in L2; this node contributes to $\delta_i^{(n_l)}$ through the weight $w_{ij}^{(2)}$ (see Figure 4.3 for the case of $j = 1$ and $i = 1$). Consider the following two cases:

1. In Figure 4.3, the output layer δ is *communicated* to the hidden node by the weight of the connection. In the case where there is only one output layer node, the generalized hidden layer δ is defined as
$$\delta_j^{(l)} = \delta_1^{(l+1)} w_{1j}^{(l)} f'\left(z_j^{(l)}\right),$$
where j is the node number in layer l.

2. However, in the case with multiple output nodes, the weighted sum of all communicated errors are used to calculate $\delta_j^{(l)}$ (see Figure 4.4)

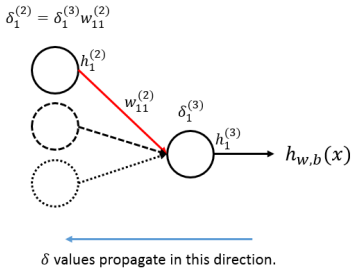

Figure 4.3: Simple back-propagation illustration.

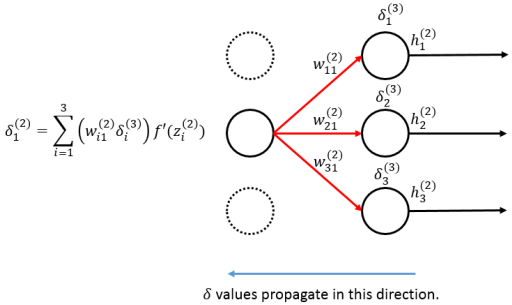

Figure 4.4: Back-propagation illustration with multiple outputs.

As shown in Figure 4.4, each δ value from the output layer is included in the sum used to calculate $\delta_1^{(2)}$, but each output δ is weighted according to the appropriate $w_{i1}^{(2)}$ value; e.g. N1 in L2 contributes to the error of three output nodes, therefore the measured error (or cost function value) at each of these nodes must be "passed back" to the δ value for this node.

The generalized expression for the δ values of the nodes in the hidden layers is

$$\delta_j^{(l)} = \left(\sum_{i=1}^{s_{(l+1)}} w_{ij}^{(l)} \delta_i^{(l+1)} \right) f'(z_j^{(l)}),$$

where j is the node number in layer l, i is the node number in layer $l+1$, and the value $s_{(l+1)}$ is the number of nodes in layer $(l+1)$.

Applying the steps used to derive Equation 4.5, the reader can also derive the bias function

$$\frac{\partial}{\partial b_i^{(l)}} J(W, b, x, z) = \delta_i^{(l+1)}.$$

The final weight and bias functions are

$$w_{(ij)}^{(l)} = w_{(ij)}^{(l)} - \alpha \frac{\partial}{\partial w_{(ij)}^{(l)}} J(w,b) \tag{4.6}$$

$$b_{(i)}^{(l)} = b_{(i)}^{(l)} - \alpha \frac{\partial}{\partial b_{(i)}^{(l)}} J(w,b) \tag{4.7}$$

To increase computational performance, the next section will reformulate Equations 4.6 and 4.7 in vector form.

4.6 Vectorizing Back-Propagation

Before vectorizing the back-propagation, first consider the simplified (but currently incorrect) cost functions for the weight and bias

$$\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(l)}} = \mathbf{h}^{(l)} \boldsymbol{\delta}^{(l+1)}$$
$$\frac{\partial \mathcal{J}}{\partial \mathbf{b}^{(l)}} = \boldsymbol{\delta}^{(l+1)}.$$

Notice that $h^{(l)}$ is a $(s_l \times 1)$ vector, or of *dimension* $(s_l \times 1)$, where s_l is the number of nodes in layer l. Similarly $\boldsymbol{\delta}^{(l+1)}$ is dimension $(s_{l+1} \times 1)$. However, from Linear Algebra we know that $\mathbf{W}^{(l)}$ must be the same size as $\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(l)}}$. Linear Algebra rules for matrix multiplication state that a matrix of dimension $(n \times m)$ multiplied by a matrix of dimension $(o \times p)$ will have a product matrix of size $(n \times p)$. Therefore, to get the dimension $(s_{l+1} \times s_l)$, we use the *transpose* of $h^{(l)}$:

$$\boldsymbol{\delta}^{(l+1)} (\mathbf{h}^{(l)})^\intercal \equiv (s_{l+1} \times 1) \times (1 \times s_l) \equiv (s_{l+1} \times s_l).$$

The resulting "vectorized" back-propagation function is

$$\begin{aligned} \boldsymbol{\delta}_j^{(l)} &= \left(\sum_{i=1}^{s_{(l+1)}} w_{ij}^{(l)} \delta_i^{(l+1)} \right) f'\left(z_j^{(l)}\right) \\ &= \left(\left(\mathbf{W}^{(l)}\right)^\intercal \boldsymbol{\delta}^{(l+1)} \right) \otimes f'\left(\mathbf{z}^{(l)}\right). \end{aligned}$$

The $\otimes$ symbol denotes the *Hadamard product*, an element-by-element multiplication.

4.7 Gradient Descent for Back-Propagation

As shown in Equation 4.3, the total cost function is a function of the MSE, such that

$$\mathcal{J}(\mathbf{w}, \mathbf{b}) = \frac{1}{m} \sum_{z=0}^{m} \mathcal{J}\left(\mathbf{w}, \mathbf{b}, \mathbf{x}^{(z)}, \mathbf{y}^{(z)}\right). \tag{4.8}$$

Reformulating Equation 4.6 into matrix form for the cost function in Equation 4.8:

$$\begin{aligned}
\mathbf{W}^{(l)} &= \mathbf{W}^{(l)} - \alpha \frac{\partial}{\partial \mathbf{W}^{(l)}} \mathcal{J}(\mathbf{w}, \mathbf{b}) \\
&= \mathbf{W}^{(l)} - \alpha \left[\frac{1}{m} \sum_{z=1}^{m} \frac{\partial}{\partial \mathbf{W}^{(l)}} \mathcal{J}\left(\mathbf{w}, \mathbf{b}, \mathbf{x}^{(z)}, \mathbf{y}^{(z)}\right) \right].
\end{aligned}$$

Intuitively this means that as we iterate through the training samples, we have to have a term that sums the partial derivatives of the individual sample cost functions. Summing these terms is required to later calculate the mean.

Let this summation term be represented as $\Delta \mathbf{W}^{(l)}$ and $\Delta \mathbf{b}^{(l)}$ for the weights and biases, respectively. Therefore, at each sample iteration of the training, compute the following steps:

$$\begin{aligned}
\Delta \mathbf{W}^{(l)} &= \Delta \mathbf{W}^{(l)} + \frac{\partial}{\partial \mathbf{W}^{(l)}} \mathcal{J}\left(\mathbf{w}, \mathbf{b}, \mathbf{x}^{(z)}, \mathbf{y}^{(z)}\right) \\
&= \Delta \mathbf{W}^{(l)} - \delta^{(l+1)} \left(\mathbf{h}^{(l)}\right)^{\intercal} \\
\Delta \mathbf{b}^{(l)} &= \Delta \mathbf{b}^{(l)} + \delta^{(l+1)}.
\end{aligned}$$

Upon iterating through all training samples – the Δ values have already been summed – we update the weight parameters

$$\begin{aligned}
\mathbf{W}^{(l)} &= \mathbf{W}^{(l)} + -\alpha \left[\frac{1}{m} \Delta \mathbf{W}^{(l)} \right] \\
\mathbf{b}^{(l)} &= \mathbf{b}^{(l)} + -\alpha \left[\frac{1}{m} \Delta \mathbf{b}^{(l)} \right].
\end{aligned}$$

4.8 The Final Gradient Descent Algorithm

The final gradient descent algorithm is shown in Algorithm 1 (next page).

Algorithm 1: Final Gradient Descent Algorithm.

Initialization: randomly initialize the weights for each layer in $\mathbf{W}^{(l)}$;
while *iterations* $<$ *iteration limit* **do**
- Set $\Delta\mathbf{W}$ and $\Delta\mathbf{b}$ to zero;
- **for** *samples* 1 *to* m **do**
  - Perform a feed-forward pass through all the n_l layers. Store the activation function outputs $\mathbf{h}^{(l)}$;
  - Calculate the value $\delta^{(n_l)}$ for the output layer;
  - Use back-propagation to calculate the values $\delta^{(l)}$ for layers 2 to $n_l - 1$;
  - Update $\Delta\mathbf{W}^{(l)}$ and $\Delta\mathbf{b}^{(l)}$ for each layer;
- **end**
- Perform a gradient descent step:

$$\mathbf{W}^{(l)} = \mathbf{W}^{(l)} + -\alpha \left[\frac{1}{m}\Delta\mathbf{W}^{(l)}\right]$$
$$\mathbf{b}^{(l)} = \mathbf{b}^{(l)} + -\alpha \left[\frac{1}{m}\Delta\mathbf{b}^{(l)}\right]$$

- Repeat the algorithm until the user-desired threshold is met or the average cost function has reached a minimum;

end
Result: The result is a trained ANN.

Chapter 5

Implementing a Neural Network

In the previous sections we looked at ANN theory, now we will implement the theory. To illustrate the theory we will perform training and prediction on the MNIST dataset, attempting to estimate the digits that these pixels represent (using neural networks of course). The MNIST dataset is a standard dataset in neural network and deep learning literature. The dataset consists of images of hand-written digits that are labeled, or "tagged", so that we can train and compare results of images against the true labeled value. Each image is of dimension 8×8 gray-scale pixels, a total of 64 values that indicate pixel intensity. We will use the *Python* Machine Learning library, *scikit learn*. An example of the image (and conveniently part of the *scikit learn* dataset) is shown in the code below (for an image of 1):

```
from sklearn.datasets import load_digits
digits = load_digits()
print(digits.data.shape)
import matplotlib.pyplot as plt
plt.gray()
plt.matshow(digits.images[1])
plt.show()
```

The code above prints $(1797, 64)$ to show the shape of input data matrix, and a plot of the digit "1" in Figure 5.1.

As is typically the case, we must first *pre-process* the data-set. Here we will perform two steps:

1. Scale the data.
2. Split the data into *train* and *test* datasets.

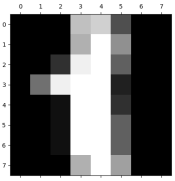

Figure 5.1: MNIST digit “1”.

5.1 Scaling the Data

To correctly utilize the activation function that has an x-axis *sensitivity* range of ± 1, we need to scale our input data to a range of ± 1.

First consider one of the dataset pixel representations:

```
digits.data[0,:]
Out[2]:
array([  0.,   0.,   5.,  13.,   9.,   1.,   0.,   0.,   0.,   0.,  13.,
        15.,  10.,  15.,   5.,   0.,   0.,   3.,  15.,   2.,   0.,  11.,
         8.,   0.,   0.,   4.,  12.,   0.,   0.,   8.,   8.,   0.,   0.,
         5.,   8.,   0.,   0.,   9.,   8.,   0.,   0.,   4.,  11.,   0.,
         1.,  12.,   7.,   0.,   0.,   2.,  14.,   5.,  10.,  12.,   0.,
         0.,   0.,   0.,   6.,  13.,  10.,   0.,   0.,   0.])
```

Notice that the input data ranges from 0 up to 15. Scaling to the range of ± 1, 1σ, using *scikit learn* results in

```
from sklearn.preprocessing import StandardScaler
X_scale = StandardScaler()
X = X_scale.fit_transform(digits.data)
X[0,:]
Out[3]:
array([ 0.        , -0.33501649, -0.04308102,  0.27407152, -0.66447751,
       -0.84412939, -0.40972392, -0.12502292, -0.05907756, -0.62400926,
        0.4829745 ,  0.75962245, -0.05842586,  1.12772113,  0.87958306,
       -0.13043338, -0.04462507,  0.11144272,  0.89588044, -0.86066632,
       -1.14964846,  0.51547187,  1.90596347, -0.11422184, -0.03337973,
        0.48648928,  0.46988512, -1.49990136, -1.61406277,  0.07639777,
        1.54181413, -0.04723238,  0.        ,  0.76465553,  0.05263019,
       -1.44763006, -1.73666443,  0.04361588,  1.43955804,  0.        ,
       -0.06134367,  0.8105536 ,  0.63011714, -1.12245711, -1.06623158,
        0.66096475,  0.81845076, -0.08874162, -0.03543326,  0.74211893,
        1.15065212, -0.86867056,  0.11012973,  0.53761116, -0.75743581,
       -0.20978513, -0.02359646, -0.29908135,  0.08671869,  0.20829258,
       -0.36677122, -1.14664746, -0.5056698 , -0.19600752])
```

By default, *scikit learn* normalizes the input by subtracting the mean and dividing by the standard deviation. As shown, the data is centered around zero with a 1σ standard deviation of ± 1.

Note: the output y is not scaled.

5.2 Creating Training & Testing Datasets

In the context of ML, the term "over-fitting" implies the tendency for ANN models very accurately predict specific inputs, based on extensive training, but poorly predict inputs that slightly deviate from mean of the training data. Simply stated, "over-fitting" results in the inability to predict anything the ANN has not "seen" previously. Therefore, given a set of data, $60-80\%$ of the data is used for training, while the remaining data is used for testing.

Using *scikit learn*, we can split the data into training and testing sets, as show below

```
from sklearn.model_selection import train_test_split
y = digits.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4)
```

In this case, we have used the 60/40-rule to split the data into training and testing, respectively.

5.3 Setting up the Output Layer

To predict digits from 0 to 9, we need 10 nodes in the output layer. For example: the prediction of the digit "2" should produce the output layer result $[0, 0, 1, 0, 0, 0, 0, 0, 0, 0]$. However, in reality the result will more closely resemble $[0.05, 0.05, 0.8, 0, 0, 0.05, 0, 0.02, 0.01, 0.02]$, which sums to 1, and the most likely value indicated by the largest value 0.8, representing the the digit "2".

The MNIST data supplied in *scikit learn*, the "targets" or the classification of the handwritten digits is in the form of a single number. We need to convert that single number into a vector so that it lines up with our 10 node output layer. In other words, if the target value in the dataset is "1" we want to convert it into the vector $[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]$, as shown below

```
import numpy as np
def convert_y_to_vect(y):
    y_vect = np.zeros((len(y), 10))
    for i in range(len(y)):
        y_vect[i, y[i]] = 1
```

```
    return y_vect
y_v_train = convert_y_to_vect(y_train)
y_v_test = convert_y_to_vect(y_test)
y_train[0], y_v_train[0]
Out[8]:
(1, array([ 0., 1., 0., 0., 0., 0., 0., 0., 0., 0.]))
```

5.4 Creating the Neural Network

Next we will specify the structure of the neural network. The input layer requires 64 nodes for the 64 pixels in each image. The output layer requires 10 nodes to predict the digits, as discussed Section 5.3. The hidden layer requires enough nodes to account for the complexity of the data. Using the relation $\frac{N_{\text{input}}}{2}$, where N_{input} represents the number of nodes in the input layer, define 30 nodes for the hidden layer. Therefore:

```
nn_structure = [64, 30, 10]
```

Define the sigmoid function and its derivative:

```
def f(x):
    return 1 / (1 + np.exp(-x))
def f_deriv(x):
    return f(x) * (1 - f(x))
```

5.5 Training the Neural Network

Recall Algorithm 1 from Section 4.8.

The first step is to initialize the weights for each layer. To simplify the code, we'll use Python dictionary objects (initialized by). Next, initialize the weights to random values, using the *NumPy* function *random_sample*, to ensure convergence during training. The weight initialization function is shown below.

```
import numpy.random as r
def setup_and_init_weights(nn_structure):
    W = {}
    b = {}
    for l in range(1, len(nn_structure)):
        W[l] = r.random_sample((nn_structure[l], nn_structure[l-1]))
        b[l] = r.random_sample((nn_structure[l],))
    return W, b
```

Listing 5.1: Initialize Weights Function.

Next, set the mean accumulation values $\Delta\mathbf{W}$ and $\Delta\mathbf{b}$ to zero (these need to be the same size as the weight and bias matrices).

```
def init_tri_values(nn_structure):
    tri_W = {}
    tri_b = {}
    for l in range(1, len(nn_structure)):
        tri_W[l] = np.zeros((nn_structure[l], nn_structure[l-1]))
        tri_b[l] = np.zeros((nn_structure[l],))
    return tri_W, tri_b
```

Listing 5.2: Set Mean Accumulation Values Function.

Next, a single step is performed in the gradient descent loop, compute a feed-forward pass through the network.

```
def feed_forward(x, W, b):
    h = {1: x}
    z = {}
    for l in range(1, len(W) + 1):
        # if it is the first layer, then the input to the weights is x,
        # otherwise, it is the output of the last layer.
        if l == 1:
            node_in = x
        else:
            node_in = h[l]
        z[l+1] = W[l].dot(node_in) + b[l] # z^(l+1) = W^(l)*h^(l) + b^(l)
        h[l+1] = f(z[l+1]) # h^(l) = f(z^(l))
    return h, z
```

Listing 5.3: Feed-Forward Function.

Finally, we calculate the output layer $\boldsymbol{\delta}^{(n_l)}$, and any hidden layer values $\boldsymbol{\delta}^{(l)}$ to perform the back-propagation.

```
def calculate_out_layer_delta(y, h_out, z_out):
    # delta^(nl) = -(y_i - h_i^(nl)) * f'(z_i^(nl))
    return -(y-h_out) * f_deriv(z_out)

def calculate_hidden_delta(delta_plus_1, w_l, z_l):
    # delta^(l) = (transpose(W^(l)) * delta^(l+1)) * f'(z^(l))
    return np.dot(np.transpose(w_l), delta_plus_1) * f_deriv(z_l)
```

Listing 5.4: Calculate Output Layer Deltas.

Assembling all of the steps together.

```
def train_nn(nn_structure, X, y, iter_num=3000, alpha=0.25):
    W, b = setup_and_init_weights(nn_structure)
    cnt = 0
    m = len(y)
    avg_cost_func = []
    print('Starting gradient descent for {} iterations'.format(iter_num))
    while cnt < iter_num:
        if cnt%1000 == 0:
            print('Iteration {} of {}'.format(cnt, iter_num))
        tri_W, tri_b = init_tri_values(nn_structure)
        avg_cost = 0
        for i in range(len(y)):
            delta = {}
            # perform the feed-forward pass and return the stored h and z
            # values to be used in the gradient descent step.
            h, z = feed_forward(X[i, :], W, b)
            # loop from nl-1 to 1 back-propagating the errors
            for l in range(len(nn_structure), 0, -1):
                if l == len(nn_structure):
                    delta[l] = calculate_out_layer_delta(y[i,:], h[l], z[
    l])
                    avg_cost += np.linalg.norm((y[i,:]-h[l]))
                else:
                    if l > 1:
                        delta[l] = calculate_hidden_delta(delta[l+1], W[l
    ], z[l])
                    # triW^(l) = triW^(l) + delta^(l+1) * transpose(h^(l)
    )
                    tri_W[l] += np.dot(delta[l+1][:,np.newaxis], np.
    transpose(h[l][:,np.newaxis]))
                    # trib^(l) = trib^(l) + delta^(l+1)
                    tri_b[l] += delta[l+1]
        # perform the gradient descent step for the weights in each layer
        for l in range(len(nn_structure) - 1, 0, -1):
            W[l] += -alpha * (1.0/m * tri_W[l])
            b[l] += -alpha * (1.0/m * tri_b[l])
        # complete the average cost calculation
        avg_cost = 1.0/m * avg_cost
        avg_cost_func.append(avg_cost)
        cnt += 1
    return W, b, avg_cost_func
```

Listing 5.5: Train Neural Net Function.

Notice that the function above does not terminate upon reaching a threshold. Instead, the function terminates at $3,000$ iterations so that we can monitor the change in the average cost function, see avg_cost_func. In each gradient descent iteration, the function cycles through each training sample ($range(len(y))$ and performs the feed-forward pass followed by the back-propagation. The back-propagation step is an iteration through the layers starting at the output layer and working backwards – $range(len(nn\_structure), 0, -1)$. The average cost is calculated at the output layer ($l == len(nn_structure)$). The mean accumulation values, $\Delta\mathbf{W}$ and $\Delta\mathbf{b}$, designated as tri_W and tri_b, are updated for every layer. After iterating through all training samples, accumulating the tri_W and tri_b values, the gradient descent step is computed to change and the values for the weight and bias are updated

$$\mathbf{W}^{(l)} = \mathbf{W}^{(l)} - \alpha[\frac{1}{m}\Delta\mathbf{W}^{(l)}]$$

$$\mathbf{b}^{(l)} = \mathbf{b}^{(l)} - \alpha[\frac{1}{m}\Delta\mathbf{b}^{(l)}]$$

At termination, the function returns the trained weight and bias values, as well as the tracked average cost for each iteration.

Running the function $train\_nn$ may take a few minutes, depending on the capabilities of your computer.

```
W, b, avg_cost_func = train_nn(nn_structure, X_train, y_v_train)
```

After the function terminates, we can plot the average cost for each iteration. As shown in Figure 5.2, by $3,000$ iterations the average cost has started to "plateau", implying that additional iterations are not likely to improve performance.

```
plt.plot(avg_cost_func)
plt.ylabel('Average J')
plt.xlabel('Iteration number')
plt.show()
```

5.6 Testing the Neural Network

With an adequately trained MNIST neural network model, we can test a (64 pixel) input from the MNIST dataset. This is performed by a *single* feed-forward pass through the network using our trained weight and bias values. As discussed previously, we assess the prediction of the output layer by taking the node with the maximum output as the predicted digit using the $numpy.argmax$ function.

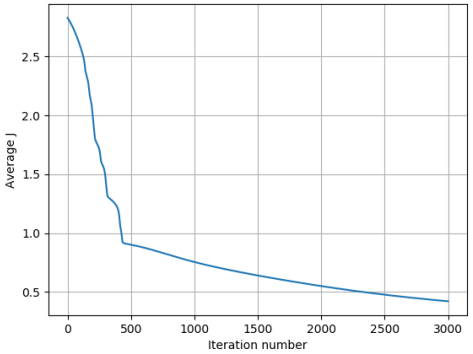

Figure 5.2: Average cost vs. iteration

```
def predict_y(W, b, X, n_layers):
    m = X.shape[0]
    y = np.zeros((m,))
    for i in range(m):
        h, z = feed_forward(X[i, :], W, b)
        y[i] = np.argmax(h[n_layers])
    return y
```

Finally, we assess the accuracy of the prediction (i.e. the percentage of times the network predicted the handwritten digit correctly), by using the *scikit learn accuracy_score* function:

```
from sklearn.metrics import accuracy_score
y_pred = predict_y(W, b, X_test, 3)
accuracy_score(y_test, y_pred)*100
```

In this example the *accuracy_score* function computes an 86% accuracy of correctly predicting the digits. While this is good for a simple Artificial Neural Network, the current state-of-the-art Deep Learning algorithms achieve accuracy scores of 99.7%.

Leveraging the lessons learned in this tutorial, the reader is encouraged to learn and investigate the current state-of-the-art algorithms, e.g. convolution neural networks (CNN), deep belief nets (DBN), and recurrent neural networks (RNN).

Appendix A

Conventions & Symbols

A.1 Notation

The notational conventions that are used throughout this text are summarized in Table A.1. A non-bold face symbol denotes a scalar quantity. A bold face symbol denotes either a vector (typically lower case) or a matrix (typically upper case). It is important to make the distinction between a *true* value, a *calculated*, *estimated*, or a *measured* value. As shown in Table A.1, the true value has no additional mark; the calculated value has a "hat" on it; the measured value has a "tilde" above it. The error is defined as the true value minus the estimated value. The error quantity is indicated with a δ, for example $\delta\mathbf{x} = \mathbf{x} - \hat{\mathbf{x}}$.

Table A.1: Notational conventions.

x	non-bold face variables denote *scalars*
$\mathbf{x}$	boldface lower-case denotes *vector* quantities
$\mathbf{X}$	boldface upper-case denotes *matrix* quantities
$x_{i,j}$	row i and column j entry of matrix $\mathbf{X}$
$\mathbf{x}$	true value of $\mathbf{x}$
$\hat{\mathbf{x}}$	calculated value of $\mathbf{x}$
$\tilde{\mathbf{x}}$	measured value of $\mathbf{x}$
$\delta\mathbf{x}$	error $\mathbf{x} - \hat{\mathbf{x}}$
$\mathbf{R}_a^b$	transformation matrix from reference frames a to b
$\mathbf{x}^a$	vector $\mathbf{x}$ represented with respect to frame a
$\mathbb{R}, \mathbb{R}^+, \mathbb{R}^n$	real numbers, reals greater than 0, n-tuples of reals
$\mathbb{N}$	natural numbers $\{0, 1, 2, \dots\}$
$\mathbb{C}$	complex numbers
$\mathbb{Z}$	integer numbers
$\mathbf{0}_{n\times m}$ or $\mathbf{0}$	zero matrix
$\mathbf{I}_{n\times n}$ or $\mathbf{I}$	identity matrix
$\lvert\mathbf{X}\rvert$	determinant of matrix $\mathbf{X}$
R, N	range space, null space
R_∞, N_∞	generalized range space and null space
$\mathcal{N}$	Normal or Gaussian random variable

This text is a simple tutorial of a basic Artificial Neural Network, with worked examples in *Python*, so that anyone can read this text and gain the skills necessary to:

- Perform research of new and exciting literature in ML.
- Apply their *Python* skills to a problem in their field of work.

Features that distinguish this text from others:

- The text gives a balanced treatment to both the traditional and newer methods. The notation and analysis is developed to be consistent across the methods.
- From the beginning and consistently throughout the text, example problems are formulated and solved to develop an understanding of the equations and their application to Neural Networks. By highlighting this throughout, it is hoped that the reader will more fully understand and appreciate theory behind Neural Networks.
- There is an extensive treatment of modern methods, including numerically fast solutions systems of equations.

Paul F. Roysdon holds a Ph.D. Electrical Engineering (focus in Applied Mathematics & Statistics), from the University of California, as well as an M.S. in Aeronautical Engineering, M.S. in Electrical Engineering, M.S. in Mechanical Engineering, and B.S. in Aeronautical & Mechanical Engineering. He has nearly twenty years experience in engineering and applied mathematics, solving real-world problems. He formerly worked in the private sector, with experience in aircraft design of military subsonic and supersonic unmanned vehicles, as well as software development and hardware testing of autopilots and navigation systems. He currently serves as a Chief Data Scientist at the Department of Defense.