Building Neural Networks from Scratch

JC-ProgJava

1st Edition

Building Neural Networks from Scratch

2020

JC-ProgJava

https://jc-progjava.github.io/Building-Neural-Networks-From-Scratch/

This book uses fonts Amiri, Roboto, Palanquin and Source Code Pro, which are all provided by Google Fonts under the Open Font License which can be reviewed here.

Contents

Citation **1**

MLA Style 1

APA Style 1

Citation Information 1

License Notice **2**

Contact **4**

Foreword – Motivation **5**

Chapter 1 - Introduction **8**

What is machine learning? 8

How does machine learning work? 10

Chapter Summary 12

Chapter 2 - The Perceptron **14**

Chapter Summary 18

Chapter 3 - Activation Functions and Derivatives **21**

Identity Activation Function 22

Sigmoid Activation Function 23

TanH Activation Function 24

ReLU Activation Function 25

Leaky ReLU Activation Function 26

SoftPlus Activation Function 27

Notation 27

Derivatives 28

Chapter Summary 30

Chapter 4 - Neural Networks and Learning 34

Single-layer Neural Network Initialization and Learning 35

A Simple Network Example 38

Chapter Summary 44

Task 1 - Recognizing Digits 48

Task Description 48

Goal 48

Background 48

Description 49

Steps to Take 52

Solution 56

Layer.java 56

Neuron.java 63

test2.java 69

Common Questions 72

Saving Models 72

Feeding Networks Different Input Lengths 74

Finding Datasets For Model Training 75

Challenge: A Real-World Application of Digit Recognition 78

Task Description 78

Goal 78

Background 78

Description 78

Steps to Take 81

Estimating Computational Time 83

Solution 84

Layer.java 84

Neuron.java 87

initArray.java 90

application.java 92

test3.java 94

Visualizing Weights 97

visualize.java 97

Chapter Summary 101

Chapter 5 - Overfitting and Underfitting 103

Overfitting 103

Least Squares Method 106

lsm.py 109

lsm.java 110

Underfitting 116

Chapter Summary 118

Chapter 6 - Validation Datasets 121

Separation of Testing, Training, and Validation Datasets 125

Chapter Summary 126

Chapter 7 - Data Preprocessing & Augmentation 130

Image Preprocessing 130

Image Data Augmentation 132

Image Preprocessing and Image Data Augmentation Techniques 133

Other Mediums of Data Augmentation 136

Other Mediums of Data Preprocessing 136

Code Examples 138

Task 1 138

Task 2 141

Task 3 144

Chapter Summary 148

Chapter 8 - Backpropagation 151

Derivation 152

Application 157

Chapter Summary 194

Chapter 9 - Optimizers 196

Chapter 10 - Generative Adversarial Networks 197

Task 2 - Object Recognition 198

Chapter 11 - L1 & L2 Regularization 199

Chapter 12 - Convolutional Neural Networks 200

Chapter 13 - TITLE 201

Chapter 14 - TITLE 202

Chapter 20 - Conclusion 203

Chapter Summaries 204

Further Reading & Resources 205

References 210

A Readable Bibliography 217

Citation

MLA Style

JC-ProgJava. Building Neural Networks from Scratch. 2020, https://jc-progjava.github.io/Building-Neural-Networks-From-Scratch/. Accessed 1 November 2020.

APA Style

JC-ProgJava. (2020). *Building Neural Networks from Scratch*. Retrieved November 1, 2020, from https://jc-progjava.github.io/Building-Neural-Networks-From-Scratch/

Citation Information

Author: JC-ProgJava

Title: Building Neural Networks from Scratch

Publisher: N/A

Published Year: 2020

URL: https://jc-progjava.github.io/Building-Neural-Networks-From-Scratch/

License Notice

The author of this book files **all code** under the "GNU Affero General Public License v3.0" (***GNU AGPLv3***) License. The author also reserves the right to change the license when needed and will be clearly stated on the website hosting the code primarily.

Permissions of use of **code** in this book are:

- The code can be used for commercial purposes.
- The code can be freely distributed.
- The code can be modified.
- The code can be used and modified for private purposes.
- The author reserves the right to patent the code given.

Conditions are:

- The source of the code must be disclosed.
- A copy of the license and copyright notice must be included with the code.
- Users are given the right to access the source code.
- Modifications of the source code must be released under the same license.
- Changes made to source code must be stated.

Limitations are:

- The source code is provided "AS-IS", without warranty.

- The liability of the author for the source code is not ensured. In other terms, you are responsible for any damages that may occur from improperly using the source code.

The author does not include a license or copyright(currently) to the book but states clearly the limitations of use and conditions for using the book are:

- The book may be used privately and for educational purposes, although, once again please provide a source for the text.
- The author should be acknowledged for the work.
- All uses of the text must be credited and cited correctly(see above citation reference), else will otherwise be considered as **plagiarism**.
- The author is not responsible for any technical issues in the book but will appreciate a response to the feedback form provided if any occur in the text.
- The book may **not** be used commercially for any purpose without the author's explicit permission. If unsure of whether your usage of the book is permissible, please contact the author through email at juch5796@gmail.com.
- The book must not be filed for copyright by anyone other than the author.
- The author reserves the right to file a copyright for the book.

- The book must not be recreated in any other form than the given website it is contained in, unless permission is given.
- If any condition is not made clear here, do again contact the author through email at juch5796@gmail.com.

Contact

Contact the author for any other queries at juch5796@gmail.com. Feedback can be given in the Google Form https://forms.gle/NxrgNB4aSFurgRFq6, all submissions to the form are anonymous, although an email can be included for further contacting(if needed).
Tips & thanks go in a separate form https://forms.gle/RQxrX2h36BgKUK7R9, and the author may include these responses in a section on the book website. Once again, an anonymous name or real name may be given, which may be included in a tips & thanks section on the book website.

Foreword – Motivation

Hi! I know that many of you don't even notice this section, but I would appreciate it if you could read it. Thanks! :-)

In this mini-book, you will (hopefully) learn about different neural network structures. By the end of the book, I hope you also will have a solid conceptual understanding of how neural networks function. NOTE: This book is not written by a data scientist, please do contact me if you find mistakes that should be fixed, thanks! I also welcome suggestions and tips that could be added to different sections of the book. See the contact for more information! I will briefly go through some necessary formulas and equations, but will mainly stick to broad definitions that are comprehendible to even middle school students (I'll try!). Some topics are well beyond those taught in middle school, so you may want to look at more thorough and step-by-step walkthroughs on the topics on platforms like Khan Academy.

As you go through this book, you will also encounter several different projects and real-world problems that I will walk you through and solve. I highly recommend you take a try at solving them beforehand! Solutions that I give may not be the best and most state-of-the-art, so you can also try and tweak it a bit to make it better! I will try to add some optimizations to the code to make it faster and more efficient and give you a summary of what state-of-the-art methods data scientists now use. All code in this book is either pseudocode,

Python (when used, an identical Java program is also given), and Java, so you should be able to roughly interpret it if you have learned any C-like programming languages! You don't need to learn one right away to continue reading either… I provide detailed explanations of each line of code. You can also find it all on GitHub (with this book).

Now that I have given an introduction to what you will learn, I should probably answer an impending question that will some time or later come to mind: "why did you write this book? Surely you didn't do it just for the sake of fun?" I admit, there was a motivation behind it. Nowadays, it is hard to find a detailed (and free) guide to neural networks that explains it all very well. I wrote it, partly also to get a more intuitive understanding of the way neural networks work myself: I had only gotten interested in the subject in March 2020, and yet hadn't been able to find a tutorial that explained it all in detail and provided programs that required no libraries (libraries just do all the work for you, you don't get a good comprehensive understanding on what it does). So, I wrote this, to store all my personal experiences in words, and to make your life in learning about neural networks easier. Thank me later :). People always think that writing the book is the hardest part of the entire process, but I believe that finding materials and sources or references is even harder. Stating information in a readable and easy-to-understand way was probably the most challenging part I tried to accomplish in this book.

On another note, this book took several hundred hours to write and consists of 28000 words. What I am trying to get at is that it took a very long time to write this, so I would appreciate it if all uses (if any) of this book in other forms are attributed and properly cited/credited. It would also be great if this piece of writing could be attributed to the correct authors and not "stolen" or plagiarised. I provide a citation reference which you can look at. I don't use any license, but the limitations for use of this book are also listed there. Any references or sources for images, definitions, or phrases/ideologies are also cited on the references page. If you feel that your work may have been incorrectly cited or misused, you can contact me and I will see to it that we can come to a resolution.

Finally, let's end on a happier note:

Start reading!

JC-ProgJava

October 2020

Chapter 1 - Introduction

> *"Success in creating AI would be the biggest event in human history. Unfortunately, it might also be the last, unless we learn how to avoid the risks."*
> *– Stephen Hawking*

In this introduction, I hope to give clear and coherent answers to a few common questions about machine learning, artificial intelligence, and alike areas in data science.

What is machine learning?

The simple answer is, it is an algorithm that a machine uses to be able to give a correct output for a given set of inputs. Expanding on that, it is one that identifies possible similarities in a given set of inputs and outputs and uses it to classify new inputs that haven't been seen before.

Machine learning can be separated into three categories: supervised and unsupervised learning; and reinforcement learning. Supervised learning requires all data to be labeled, which the computer uses to learn from its mistakes and get better. Unsupervised learning does the opposite, given a set of data with *no*

labels whatsoever, a computer identifies characteristics commonly found among a cluster of the data and uses this to identify or classify different sets of inputs. And lastly, reinforcement learning, which makes a machine learn by "rewarding" it when a correct or ideal interpretation is made.

Here, what I mean by *labeling* is being marked with an answer. For example, supervised learning requires a picture of a cat to be labeled "this is a cat", and for a picture of a dog to be labeled "this is a dog". This form of learning requires an extensive amount of work for humans, who need to manually label an *entire* collection of images, to get a computer to classify different inputs. This is why people will tell you "machine learning is all about the data you have". This is not always entirely true. You can achieve state-of-the-art results(or nearly) with a small dataset. So, I usually say "machine learning is all about the choice of hyperparameters and methods used". I mention another word *hyperparameter*, which just means a parameter that controls a set of parameters. You will get a more thorough definition in later chapters.

You may also wonder about the applications of each category of machine learning. Supervised learning is usually used for classification or regression problems. For example, classifying digits or predicting costs of houses by the number of rooms, location, etc. Unsupervised learning is usually used for identifying unique features in data. This may mean trying to find a correlation between two seemingly unrelated topics or ideas. Reinforcement learning typically occurs in artificial intelligence in the games industry/area. For

example, you could train a computer to play Go, Chess, Snake, Super Mario, etc. with reinforcement learning.

How does machine learning work?

People tend to assume that only professional data scientists can apply machine learning to different areas in the modern world and assume that the topic is extremely deep-reaching and complex. This is false. For example, I am not a professional data scientist, yet I understand various techniques in machine learning relatively well. Machine learning is just a series of steps that a computer takes that turn any problem into one that has numbers. As you may know, computers are not good at evaluating *anything* other than numbers. All information is stored as 0s and 1s, which is why people state that "computers will never have emotions". This series of steps consist of:

1. Gathering Data
    - Gather relevant data that can be used to predict an output with a set of inputs.
2. Preparing Data
    - Make sure data is relevant and in the same format so that when the computer learns with the data, it doesn't mistake certain characteristics as a feature. For example, make sure that images of digits are all centered so that the computer doesn't determine a digit by location on the image.
    - Resizing images to be proportional
3. Modeling

    - Choosing an appropriate architecture/model for the particular problem
4. Training
    - Forward propagation
    - Backward propagation/learning
    - Error calculation
5. Evaluation
    - Validation/testing
        - Prevent overfitting or underfitting, where features are too narrow(starting to be biased to the training dataset) or too broad/abstract(not determining key features, but small characteristics), more will be said of this topic.
6. Hyperparameter adjustment
    - Adjust hyperparameters to get the best result possible
        - Learning rate
        - Activation function
        - Optimization
7. Prediction/Deployment
    - Interpret new inputs using the trained model!

This may all sound like gibberish to you, it's fine! By the end of this book, you should have a solid fundamental understanding of each of these unique peculiar terms!

Chapter Summary

1. Machine learning is simply a set of algorithms a computer runs to learn characteristics from a given set of inputs, which enables them to classify sets of unseen input data.

2. Machine learning consists of three categories: supervised learning, unsupervised learning, and reinforcement learning.

3. Supervised learning requires data to be labeled. Common uses for supervised learning are for classification (recognizing) or regression (estimating/predicting a value).

4. Unsupervised learning does not require data to be labeled. Uses for unsupervised learning are to identify features that appear in data, often finding a correlation between two seemingly unrelated factors.

5. Reinforcement learning embraces a reward-based learning system, where the computer is "rewarded" when it achieves something and "penalized" when failing to achieve a goal. Common uses for reinforcement learning are in game-based AI, for example, Chess, Go, Snake, etc.

6. Labeling is the action of putting the correct answer onto data so that a computer can learn from its mistakes.

7. Machine learning consists of seven main steps:

a. Gathering Data
b. Preparing Data
c. Modeling
d. Training
e. Evaluation
f. Hyperparameter adjustment
g. Prediction/Deployment

Chapter 2 - The Perceptron

> *"I think the brain is essentially a computer and consciousness is like a computer program. It will cease to run when the computer is turned off. Theoretically, it could be re-created on a neural network, but that would be very difficult, as it would require all one's memories."*
>
> *— Stephen Hawking*

Now that we had an introduction, it is time to introduce a basic computational model called the perceptron, which is a model in a collection of models called *artificial neurons*, in general. The perceptron was invented by Frank Rosenblatt in 1958. Rosenblatt defined the perceptron as a unit of computation that had a threshold Θ, and received inputs $i_1, i_2, \ldots, i_n$, which are multiplied by $w_1, w_2, \ldots, w_n$ respectively. He then proposed that all of the products be summed together and another constant called the *bias* was added, for which the perceptron would output 1 if the sum was greater than or equal to the threshold, and 0 if it was smaller. In mathematical terms, $t = \sum_{k=1}^{n} w_k i_k$, output

$\begin{cases} 1, & b+t>\Theta \\ 0, & b+t\leq\Theta \end{cases}$. In other words, a perceptron is a unit consisting of something called a *weight*, which was multiplied by the *input* (sometimes called the *activation*), and if the sum of the products of the two was bigger than a number called the threshold, the perceptron would output a 1, or 0 if not.

Another way to think about this is, the threshold is a value for which classifies all events a as similar if the summation of the product of their inputs and a set of weights with another constant called the *bias* is higher than the threshold. This also could be thought of as an event with a point (x, y). The threshold would therefore be a line $y = \Theta$, which separated two different sets of points. Here is a graphical representation:

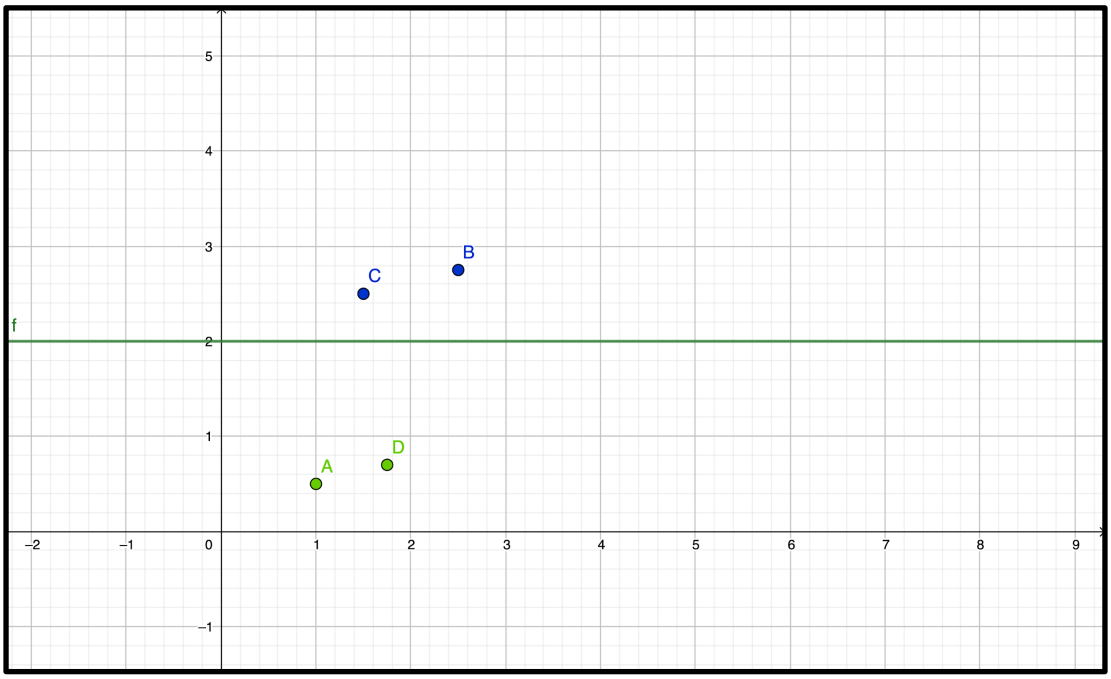


Figure 2-1: *A graphical representation of a threshold separating two points*

Like the diagram, the perceptron can similarly "separate" two different inputs and classify or predict the output for that input. So, now that we have these two basic properties, we can start to understand why machine learning and artificial intelligence are important in the field of data science. For example, say you want to classify whether a 3 by 3 grid of inputs is a 0 or 1. This requires a method to separate different sets of inputs and identify common features. In the perceptron model, since the weights, biases, and threshold are the only unknown variables, these are the ones we need to figure out to have the desired classification result. Machine learning, therefore, can be defined as the method that achieves this task of finding ideal values for the weights, bias, and threshold. Do note that other models may not have a threshold, but require an *activation function* which we will discuss in later chapters.

This was a ground-breaking invention that inspired others after Rosenblatt and caused the first AI winter. An AI winter is a period when no or little progress was made in the field of artificial intelligence. After the invention of the perceptron, professor Marvin Minsky in his book *Perceptrons* proved that a single layer of these perceptrons wasn't able to predict anything. This led to a misunderstanding which resulted in a decrease in research on the area. Minsky actually also wrote in the same book that a network of several layers would have the ability to approximate any function, which resulted in the Universal Approximation Theorem, stating that a neural network with multiple layers could approximate any continuous (and predictable) function to any degree of accuracy.

Given this information, neural networks are just a collection or array of these perceptrons (although they may have a different type of neuron instead), connected through multiple layers. Usually, the first layer is called the *input layer*, the middle layers are called the *hidden layers* and the last layer is called the *output layer*. Below is a diagram of a neural network.

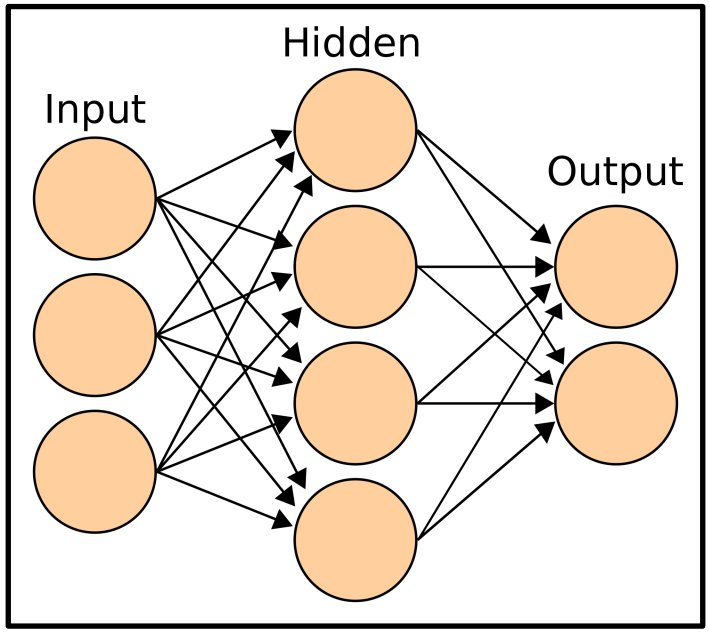


Figure 2-2: *Neural Network Architecture*

This network can be called a neural network with structure 3-4-2 (appears in the layer order from left to right). The lines on the diagram show the different connections between neurons in the network, which are represented by orange circles. The connections can be referred to as weights, which (if you remember) are multiplied by the *input* or *activation*. It is important to note that there is only one bias in the weighted sum to each neuron in the next layer. Usually, there are no connections *towards* a bias, but biases are connected to the

next layer's neurons. Networks can be called "single-layered networks" when there are only two layers in it(input and output). We will talk more about neural networks in Chapter 4.

Chapter Summary

1. A perceptron is a type of artificial neuron(I will be calling it a 'neuron' instead, throughout most of the book).

2. The perceptron is a computational model consisting of *weights*, a set of *inputs* or *activations*, *biases*, and a *threshold*.

3. The *input* to the perceptron is multiplied by the *weight* and a *bias* is added. If the result is bigger than another value the *threshold*, then the perceptron outputs a 1, otherwise, it outputs 0.

4. The perceptron model was invented by Frank Rosenblatt in 1958.

5. A perceptron in mathematical terms is: $t = \sum_{k=1}^{n} w_k i_k$, where the output of the perceptron is $\begin{cases} 1, & b+t>\Theta \\ 0, & b+t\leq\Theta \end{cases}$, where w_k and i_k are the weights and inputs at index k respectively. n is the total number of weights connected to the perceptron and Θ is the threshold value. The output is also alternatively expressed as: $\begin{cases} 1, & b+t\geq\Theta \\ 0, & b+t<\Theta \end{cases}$, which rarely makes a difference to the overall output of the perceptron and the accuracy of the neural network.

6. Machine learning is a set of algorithms that finds ideal values for the *weights*, *biases*, and *threshold*.

7. A threshold is like a line that separates two different sets of labeled inputs.

8. Marvin Minsky in his book *Perceptrons* proved that a layer of perceptrons could not predict anything, he also stated that multiple layers of perceptrons were able to approximate any function. This led to an AI winter.

9. The Universal Approximation Theorem states that any continuous function can be approximated to any degree of accuracy using a multilayer neural network.

10. An AI winter is a period when there is little or no further advancement in the field of Artificial Intelligence. This usually is caused by a lack of sponsoring or money that motivates the research in the field of AI.

11. Perceptrons may not have a *threshold* or *bias*, but (in that case) will require an *activation function* (Chapter 3).

12. A network consists of several layers of neurons. Single-layer networks have two layers, the first called the *input layer* and the last called the

output layer. Multi-layer networks also have *hidden layers*, which are the layers in between the *input layer* and the *output layer*.

Chapter 3 - Activation Functions and Derivatives

> *"Hoping for the best, prepared for the worst, and unsurprised by anything in between."*
> *— I Know Why the Caged Bird Sings by Maya Angelou*

Soon after the invention of the perceptron, people started to create variations of it. A widely used structure is a perceptron that does not have a threshold but instead uses an activation function (we will call this structure a *neuron*). The key inspiring question or idea that sparked this structure was: how about we keep every component but the threshold, and instead, treat the classification problem as one that is probabilistic? So, there would be a layer of neurons in the output layer, but instead of only outputting 0 or 1, all values in between are covered as well. This is essentially the main purpose of the activation function, a function that takes an input and outputs a corresponding value in a given range (usually -1 to 1, or 0 to 1). We will talk about the name's origin and various widely-used activation functions in this chapter.

In previous chapters, the *input* of the perceptron was also called the alternative name *activation*. This seems to coincidentally match the name of this chapter, doesn't it? Well, it wasn't *coincidental* when people at the time came up with the names, because activation functions basically adjust the input that goes into a

perceptron. To elaborate further, activation functions *fit* the weighted sum of the inputs and weights of a perceptron and the bias into a range, usually between 0 and 1, but sometimes between -1 and 1 or other ranges, and make the perceptron output this. Since the output of a perceptron is the input of the next layer's perceptrons, this is where the name *activation function* comes from. The *activation* of a perceptron also has another meaning of whether the perceptron fires up(1) or not(0) because neurons in the brain "activate" or light up when the output is 1.

So, what are some commonly used activation functions? Below we will list a few widely used activation functions:

Identity Activation Function

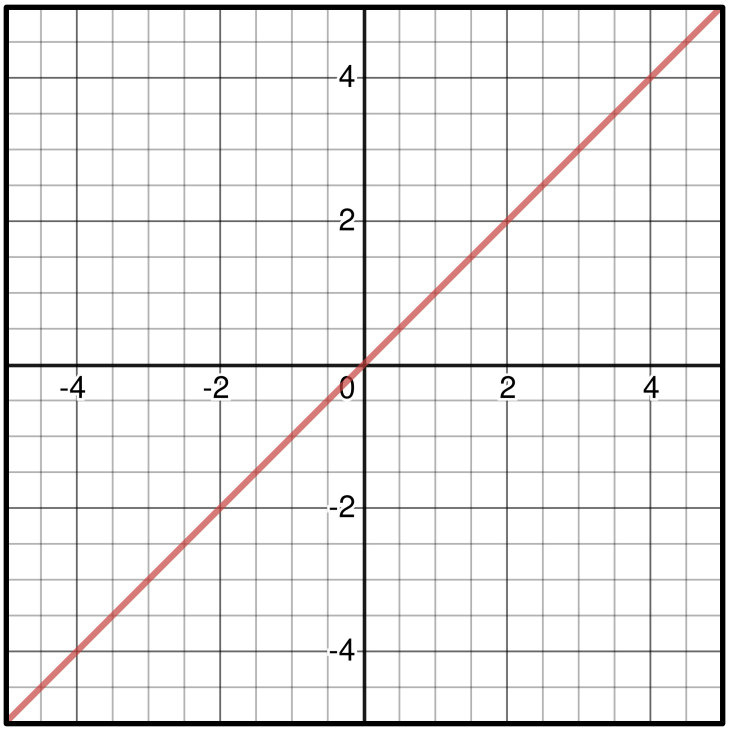


Figure 3-1: *Graph of Identity Activation Function*

The identity activation function takes the form: $\sigma(x) = x$, ranging from $-\infty$ to ∞. The function is linear and its derivative takes the form: $\sigma'(x) = 1$, which makes calculations faster due to its simplicity.

Sigmoid Activation Function

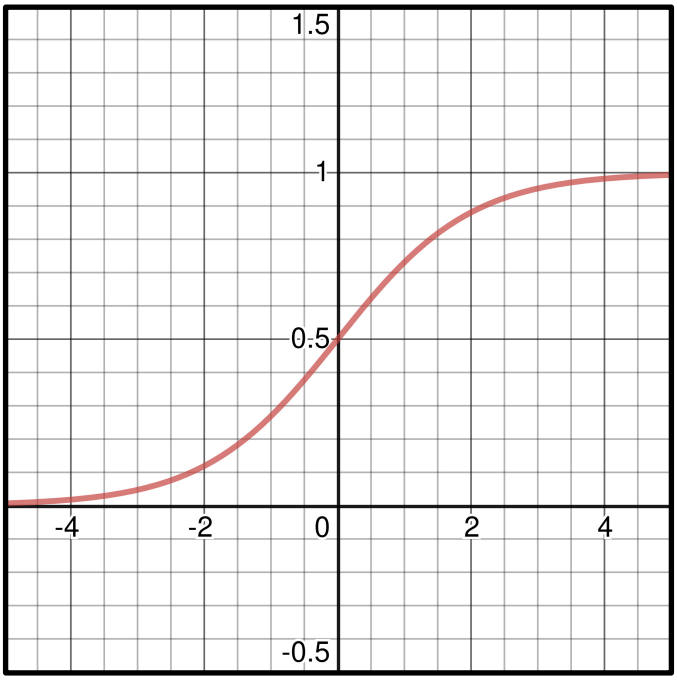


Figure 3-2: *Graph of Sigmoid Activation Function*

The sigmoid activation function takes the form: $\sigma(x) = \frac{1}{1 + e^{-x}}$. The range of the function is between 0 and 1, reaching 0.9 and 0.1 at approximately 2.125 and -2.125 respectively. The sigmoid function is nonlinear and its derivative takes the form: $\sigma'(x) = \sigma(x)(1 - \sigma(x))$, which is important when using various other learning algorithms that we will talk about like backpropagation. The

Sigmoid Function is commonly used to convert a network's output into a probabilistic output, since its values range from 0 to 1, from never uncertain to definite.

TanH Activation Function

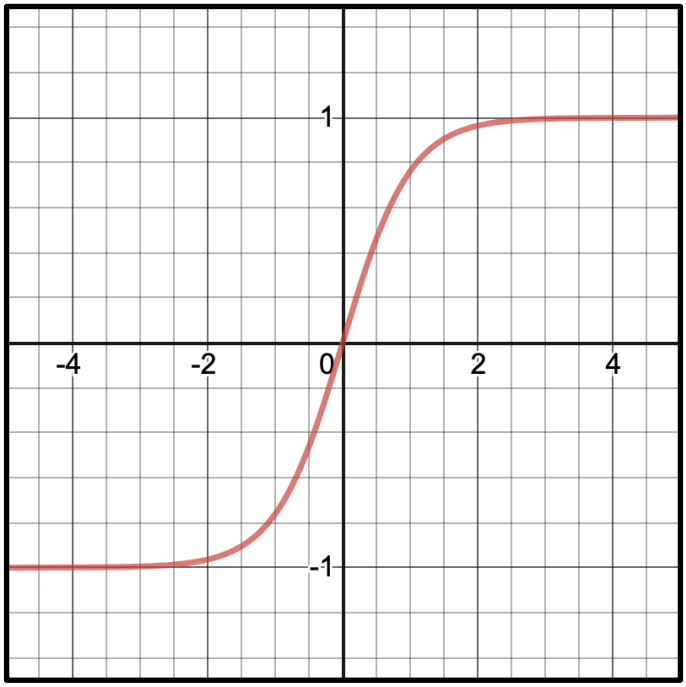


Figure 3-3: Graph of Tanh Activation Function

The tanh activation function takes the form: $\sigma(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$. You may notice that it looks quite similar to the sigmoid activation function because it is actually just a squished version of it: another way you can write the tanh activation is: $\sigma(x) = 2S(2x) - 1$, where $S(x)$ is the sigmoid function. The derivative of the tanh activation function is: $\sigma'(x) = 1 - \sigma(x)^2$. In case you didn't already know, e refers to Euler's number, and is mathematically

expressed as $\sum_{n=0}^{\infty} \frac{1}{n!}$, and $\sigma(x)^2$ refers to the square of the result of the tanh activation function.

ReLU Activation Function

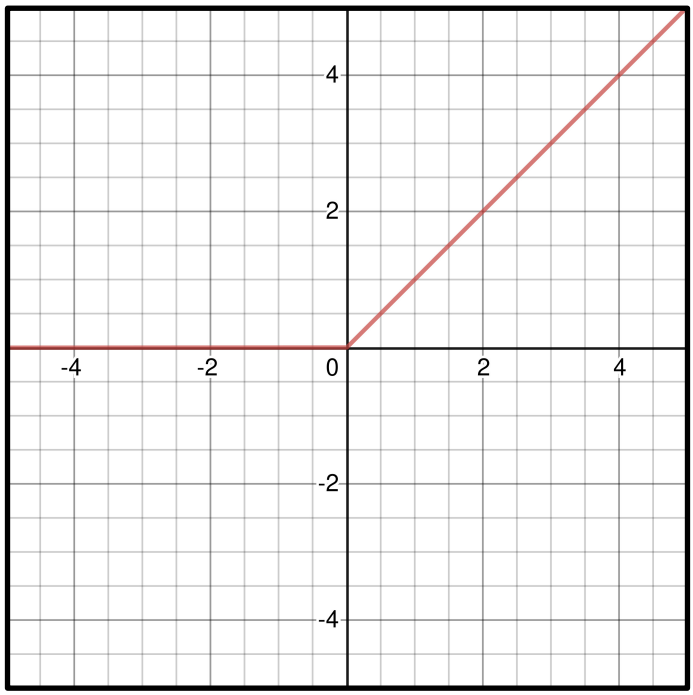


Figure 3-4: Graph of ReLU Activation Function

By far the least computationally expensive, the ReLU activation function takes the form: $\sigma(x) = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases}$. What the equation means is, that if the input x is less than or equal to 0, the function returns 0, otherwise, it returns x itself. This is arguably the simplest activation function by far and is linear. The derivative for the ReLU activation function is really similar to the function itself: $\begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases}$ (the only difference is in $1,\ x > 0$), and probably is the simplest derivative of all as well.

Leaky ReLU Activation Function

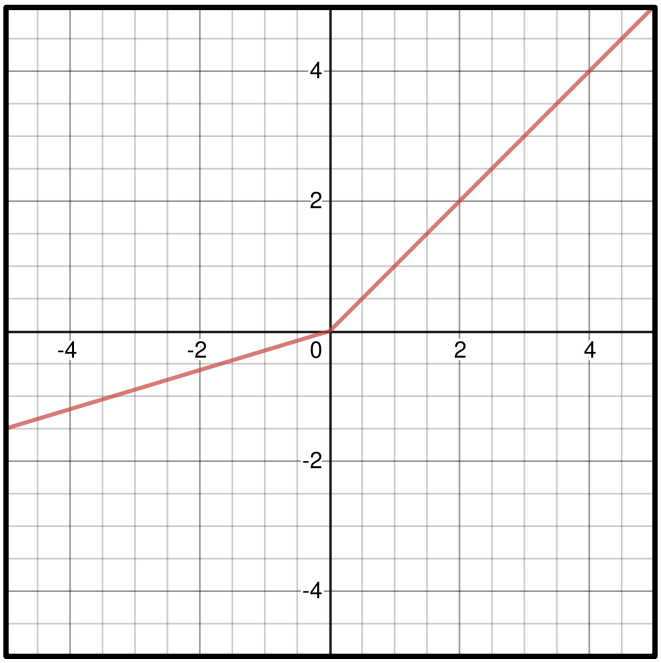


Figure 3-5: *Graph of Leaky ReLU Activation Function with* $\begin{cases} x, & x > 0 \\ 0.3x, & x \leq 0 \end{cases}$

Another alternative to the ReLU activation function is the Leaky ReLU. In the graph above, I used $\begin{cases} x, & x > 0 \\ 0.3x, & x \leq 0 \end{cases}$ to better illustrate its difference from the ReLU activation function. Usually, any input less than or equal to 0 is not "leaked" as much as the version I chose above. A popular form of the Leaky ReLU takes the form $\sigma(x) = \begin{cases} x, & x > 0 \\ 0.01x, & x \leq 0 \end{cases}$, and has a derivative of $\sigma'(x) = \begin{cases} 1, & x > 0 \\ 0.01, & x \leq 0 \end{cases}$. If you do not know what the derivative of something means, it will be explained later. The complex mathematical symbols will also be explained later in this chapter.

SoftPlus Activation Function

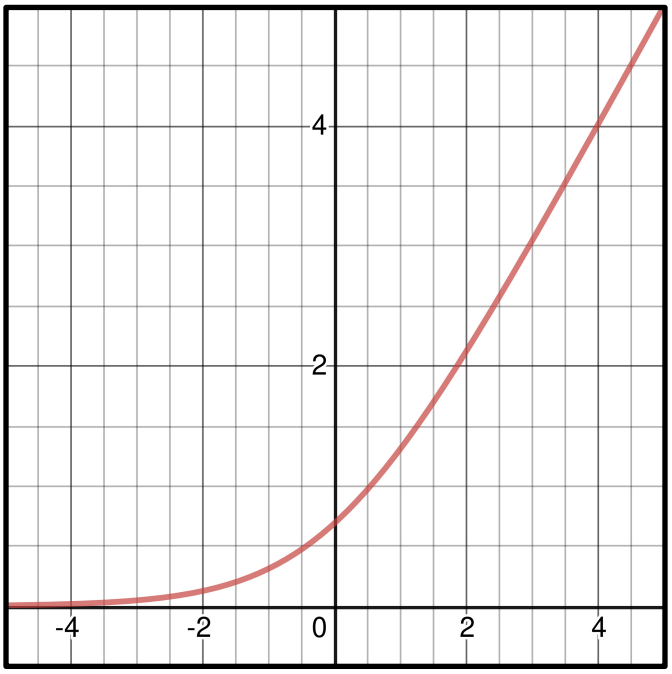


Figure 3-6: Graph of SoftPlus Activation Function

The SoftPlus activation function is another commonly used activation function. Usually, it takes the form: $\sigma(x) = \ln(1 + e^x)$ where $\ln(x)$ refers to the natural logarithm(logarithm base e) of input x. The derivative of the SoftPlus activation function is actually just the sigmoid activation function: $\sigma'(x) = \frac{1}{1 + e^{-x}}$. This activation function once again is nonlinear(particularly exponential growth, not linear).

Notation

In my explanations, you may have started noticing an *excessive* use of the σ (lowercase sigma), to express an activation function. The first activation

function that you may have heard of is the sigmoid function, which is named *sigmoid* because of its S-shaped appearance. Since sigmoid was derived from the character sigma, sigma is used to express the Sigmoid activation function and was conventionally used to express *any* activation function.

Derivatives

I also talk a lot about *derivatives* of functions, which is basically a measure of how much a function changes at any given point with respect to any other point on the function, slowly stepping closer and closer to a single point until the distance between the points is 0. In more formal terms, a *derivative* of a function measures the sensitivity of a given function to change. Basically, how does the tweaking of an input to a function change its output? The process of finding the *derivative* of a function is called *differentiation*. The derivative of a function can also be interpreted geometrically as the slope of a line tangent to a single point on the function. Derivatives of an entire function can also be found, where functions' derivatives are expressed in terms of variables like the ones given above. Another important thing to note about derivatives is that the derivative of a linear function is *always* equal to the slope of the function. You probably may be thinking: "how do we find a derivative of a single point?" because I haven't really explained that yet. Well, derivatives first take two points, one being the original point, and the other being one that slowly gets closer and closer to the original point. Call these points (x, y) and $(x + \Delta x, y + \Delta y)$. Then, we keep on decreasing the changes Δx and Δy until the two points are the same, this will therefore give the slope of a function at the single point. By connecting the two points, we are given a line with a slope

of $\frac{\Delta y}{\Delta x}$. y can also be expressed as $f(x)$(function output with input of x), therefore we can rewrite the points as $(x,\ f(x))$ and $(x + \Delta x,\ f(x + \Delta x))$ (note that $y + \Delta y = f(x + \Delta x)$).

To find the slope, we can use the slope formula (traditionally, rise over run):

$$\frac{\Delta y}{\Delta x} = \frac{f(x + \Delta x) - f(x)}{x + \Delta x - x} = \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

Or alternatively:

$$\frac{\Delta y}{\Delta x} = \frac{f(a) - f(b)}{a - b}$$

We use the slope formula because *differentiation* requires finding the limit of the slope formula, as Δx approaches 0. Expressed mathematically, find:

$$\lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

We cannot directly substitute 0 as Δx because this gives: $\frac{f(x + 0) - f(x)}{0} = \frac{0}{0}$, which is undefined (divide by 0). Instead, we need to simplify the limit such that substituting Δx with 0 works.

Let's try an example:

Given $f(x) = x^2$, find $f'(x)$. Here, $f'(x)$ represents the derivative of the function and $f(x)$ represents the function. This type of notation is usually referred to as *Lagrange's Notation* or *Prime Notation*, because of the use of prime marks ($'$) to represent derivatives. Other notations commonly used are *Leibniz's Notation*, where derivatives are expressed as $\frac{dy}{dx}$ or $\frac{d}{dx}f$; and *Newton's Notation* or *Dot Notation*, where derivatives are expressed as $\dot{y}$. Anyway, carrying on, we can substitute the function into limit equation, giving:

$$\lim_{\Delta x \to 0} \frac{x^2 + (\Delta x)^2 + 2x\Delta x - x^2}{\Delta x}$$

$$= \lim_{\Delta x \to 0} \frac{(\Delta x)^2 + 2x\Delta x}{\Delta x}$$

$$= \lim_{\Delta x \to 0} \Delta x + 2x$$

$$= \underline{2x}$$

Therefore, for $f(x) = x^2$, $f'(x) = 2x$, and $f''(x) = 2$. $f''(x)$ means taking the second derivative of the function, or taking the derivative of the derivative of function $f(x)$. It is not exactly necessary for you to have a very deep understanding of derivatives until we get to the topic of *backpropagation*, but I have explained it now in case you were curious. If you are a middle school or high school student, you probably won't have been in contact with derivatives or calculus as a whole, and it is not necessary for you to learn these topics before being taught in school. Instead, just follow my explanations and see if you can understand it. If not, please submit a feedback form or ask/learn about the topics online, on platforms like Khan Academy if you are curious.

It is almost time to start training our own neural network, so head on to the next chapter!

Chapter Summary

1. Activation functions replace the threshold of a perceptron, therefore treating classification problems like one that is probabilistic.

2. The name *activation function* is used because *activation functions* fit all inputs in between a range, which is used as the *input* or *activation* of the next layer's neurons. Also, neurons "activate" or light up(biological neuron) when the output of it is 1.

3. Some commonly used activation functions:
    a. Identity(nothing is changed)
        i. Function: $\sigma(x) = x$
        ii. Derivative: $\sigma'(x) = 1$
    b. Sigmoid
        i. Function: $\sigma(x) = \dfrac{1}{1 + e^{-x}}$
        ii. Derivative: $\sigma'(x) = \sigma(x)(1 - \sigma(x))$
    c. TanH
        i. Function: $\sigma(x) = \dfrac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$, or $\sigma(x) = 2S(2x) - 1$, where $S(x)$ is the Sigmoid Activation Function.
        ii. Derivative: $\sigma'(x) = 1 - \sigma(x)^2$
    d. ReLU
        i. Function: $\sigma(x) = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases}$
        ii. Derivative: $\sigma'(x) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases}$
    e. Leaky ReLU
        i. Function: $\sigma(x) = \begin{cases} x, & x > 0 \\ 0.01x, & x \leq 0 \end{cases}$
        ii. Derivative: $\sigma'(x) = \begin{cases} 1, & x > 0 \\ 0.01, & x \leq 0 \end{cases}$
    f. SoftPlus

i. Function: $\sigma(x) = \ln(1 + e^x)$ where $\ln(x)$ refers to the natural logarithm $\left(\text{logarithm base e}\right)$ of input x.

ii. Derivative: $\sigma'(x) = \dfrac{1}{1 + e^{-x}}$

4. The lowercase sigma is commonly used to symbolize an activation function. It is used because *sigmoid* was used in the Sigmoid Activation Function's name because of its S-shape, and because *sigmoid* came from *sigma*, and the Sigmoid Activation Function wasn't the only activation function, and the notation was widely used for it, so all activation functions conventionally were expressed with the lowercase sigma as well.

5. Derivatives measure the sensitivity of a function to the change of its input.

6. The process of finding derivatives is called *differentiation*, which is in the field of *differential calculus*.

7. The derivative of a linear function is *always* equal to its slope.

8. Finding the derivative of a single point uses limits, where the distance between two points, one of which was the original, approaches 0.

9. The slope formula: $\frac{\Delta y}{\Delta x} = \frac{f(x + \Delta x) - f(x)}{\Delta x}$, where Δx and Δy represent the change in x and y respectively, given two points (x, y) and $(x + \Delta x, y + \Delta y)$.

10. To find the derivative with the slope formula, simply simplify the limit: $\lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$, plugging in the point values or function into the equation.

11. Backpropagation requires knowledge on derivatives.

Chapter 4 - Neural Networks and Learning

> *"The worst enemy to creativity is self-doubt."*
> *— The Unabridged Journals of Sylvia Plath by Sylvia Plath*

We have introduced most of the components in a neural network including the neuron, weights or connections, biases, layer terminology and activation functions, so now it is time to introduce learning rates and algorithms.

Neurons consist of several parameters: the weights and biases. We need to know how to initialize them, and how to find the most suitable values for them. To do this, we need to first pass through the network once, the process of which is called *forward propagation* (also known as stepping and forward-passing). Afterward, we need to choose a learning algorithm that works well with the current neural network structure and use it to find the most suitable set of values for the variables.

In this chapter, we will cover basic initialization of parameters and a single-layer neural network learning algorithm.

Single-layer Neural Network Initialization and Learning

Usually, the weights and biases of a single-layer neural network are initialized using Gaussian distribution(sometimes called normal distribution) or assigned a value of 0. Here is a diagram of what gaussian distribution does:

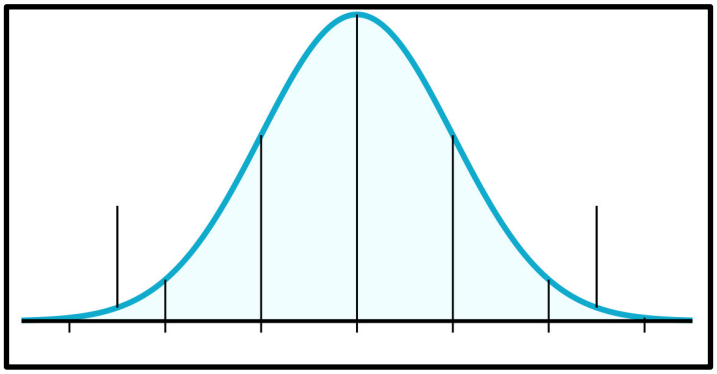


Figure 4-1: *Gaussian Distribution Chart*

It distributes a series of values with the most common values being the ones closest to a middle value, and having values further away from the middle value appears less. Usually, the middle value is 0 for initializing weights and biases. The reason that this method is used is that it randomly initializes all weights and biases with distributed values so that you can start at a different location each time, perhaps altering the accuracy of the network. People tend to stick to the initialization with 0s when trying to compare different models with each other.

These values are then adjusted using a learning algorithm called the delta rule. Now, before I show the entire formula for the delta rule, let's try derive it!

Neural networks are structures that perform *generalization* well, meaning that these networks learn by finding *patterns* in a given dataset and applying these patterns onto a set of input data. At first, parameters like *weights* and *biases* are initialized randomly, so the network isn't able to perform the desired task well. To *improve*, networks then *compare* their initial outputs after *forward propagation* and see how much it *differs* from the target (or desired) output. This notion of *differing* is usually referred to as *error* or *loss*. It can be as simple as calculating the difference between a desired output and the network's predicted output, although we usually use a function called the *mean squared error*, which simply just squares the difference between a target and predicted output pair:

$$C(t, y) = (t - y)^2$$

Where y refers to the predicted output (result of forward propagating through the network) and t refers to the target or desired output of the network for a specific input pair that was inputted into the network. MSE (mean squared error) usually takes the form:

$$\frac{1}{n}\sum_{i=1}^{n}(t_i - y_i)^2$$

Where we calculate the average *error* of the network when given n input samples and their target output samples. MSE is usually referred to as a *cost function* (and thus C) because it performs an averaging of n *error values* instead of just calculating the error of a single input sample. Note now that given an error function able to measure the difference between an expected output and an actual output, we can adjust our parameters based on this calculated error such that in the next iteration, the network has a *lower* error value than before on the same input sample(s). This process is called *minimization* or more generally, *optimization*. Furthermore, to lower an error value by adjusting

parameters, we need to know how adjusting each parameter affects that error value. Sounds familiar? Well, it is! Remember that derivatives are the measure of how *sensitive* a function is to changes to its input! So we can use derivatives to find how *sensitive* our error function is to changes to particular parameters. Then, adjusting our parameters is just as simple as increasing or decreasing their values such that the error can be decreased. For example, if the predicted output is greater than the expected output, then say, all weights (and their respective neurons) that are activated by the given input sample should be decreased, and vice versa (assuming inputs are positive values). But before we try to apply derivatives, there are a few rules that will come in handy:

1. Given $z = xy$, $\frac{dz}{dx} = y$. This makes sense as the value of z would change by y for every unit of change to x. For example:

   $z = xy$

   $z_n = (x + 1)y = xy + y$

   $z_n - z = xy + y - xy = y$

   See that by increasing x by 1, we increased z by y. Likewise, $\frac{dz}{dy} = x$.
2. Given $y = x^a$, $\frac{dy}{dx} = ax^{a-1}$. This one is harder to prove and involves concepts like the Binomial Theorem, so I won't go explicitly into its proof. Here is a good resource that proves this rule (known as the power rule).
3. $\frac{de^x}{dx} = e^x$
4. The chain rule: $\frac{\partial f(g(x))}{\partial x} = f'(g(x)) \cdot g'(x)$ or $\frac{\partial f(g(x))}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$.

Note that we will deal with some partial derivatives, which are quite similar to derivatives. Partial derivatives is deriving how a multivariable function changes

with respect to a single variable. In this case, you can just leave other variables as-is, and only derive the targeted variable. That's pretty much all you'll need for now, let's get started!

A Simple Network Example

Let's assume that our network as 1 input neuron and 1 output neuron. The task is for the network to be able to double an input value and return it in the output neuron. Here is a diagram of the network:

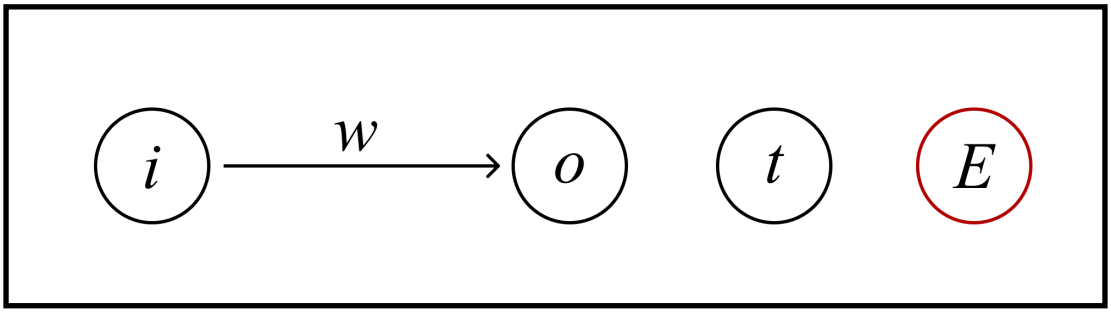


Figure 4-2: A diagram of a 1-1 neural network.

Here, i refers to the input, w a weight value, o the output of the network, t the target output of the network, and E the error of the network (calculated based on the target and actual output). We won't include a bias in the network for more simplicity and also because it isn't necessary (doubling values doesn't have a correlation with an added constant). We'll use the identity activation function for simplicity and also because we aren't fitting our values into a certain range (but we'll assume there might be a different activation function). Let's write the relationships between these variables.

$$o = \sigma(wi)$$
$$C(t, o) = \frac{1}{2}(t - o)^2$$

Note that the cost function C is a slightly modified version of the mean squared error such that the error is halved after being squared.

Also, let's have (i, t) be either $(1, 2)$, $(2.5, 5)$, or $(2, 4)$ and have w be initialized as 0.

Note that:

$$\frac{\partial C(t,o)}{\partial w} = \frac{\partial C(t,o)}{\partial o}\frac{\partial o}{\partial wi}\frac{\partial wi}{\partial w}$$

$$\frac{\partial C(t,o)}{\partial o} = \frac{\partial \frac{1}{2}(t-o)^2}{\partial o} = \frac{1}{2}[2(t-o)^{2-1}\cdot(-1)] = -(t-o)$$

$$\frac{\partial o}{\partial wi} = \frac{\partial \sigma(wi)}{\partial wi}$$

In our case, $\sigma(wi) = wi$, so: $\frac{dwi}{dwi} = 1$. And $\frac{\partial wi}{\partial w} = i$, so all in all, we get:

$$\frac{\partial C(t,o)}{\partial w} = -(t-o)\cdot i$$

Note that this derivative tells us the gradient $\left(\text{amount of change}\right)$ of C based on changes to w. It also tells us which direction to adjust w $\left(\text{i.e., increase or decrease}\right)$: a negative slope indicates that there is a lower cost at some bigger w and a positive slope tells us there is a lower cost C at some smaller w. Notice that to lower our cost, we need to move in the direction *opposite* that of the gradient value. Thus:

$$\Delta w = -(o-t)i$$

$$w_n = w + \Delta w$$

We can rewrite this slightly as:

$$\Delta w = (o-t)i$$

$$w_n = w - \Delta w$$

From now, I'll use this updated version (where $w_n = w - \Delta w$ and $\Delta w = (o - t)i$ instead of $w_n = w + \Delta w$ and $\Delta w = -(o - t)i$). This Δw value is usually called the *gradient.* For the general case (where the activation function may not be the identity activation function, and there may be multiple inputs):

$$\Delta w = \sigma'(wi)(o - t)i$$

$$w_n = w - \Delta w$$

Usually another constant (often denoted as α or η) is multiplied to Δw in order to make weight updates smaller and prevent "overstepping", so let's just set this constant at 0.1 for now. More on this constant value and "overstepping" will be discussed later. For now, let's try manually calculate a few iterations with our delta rule for the two-neuron example! We'll iterate through each of the three examples once and then test it on $i = 1$ to see whether there is any improvement.

1. $(i, t) = (1, 2)$, $w = 0$ and $\alpha = 0.1$

   $o = iw = 1 \cdot 0 = 0$

   $C(t, o) = \frac{1}{2}(t - o)^2 = \frac{1}{2}(2 - 0)^2 = 2$

   $\Delta w = \alpha(o - t)i = 0.1(0 - 2)1 = -0.2$

   $w_n = w - \Delta w = 0 - (-0.2) = 0.2$

2. $(i, t) = (2.5, 5)$, $w = 0.2$ and $\alpha = 0.1$

   $o = iw = 2.5 \cdot 0.2 = 0.5$

   $C(t, o) = \frac{1}{2}(t - o)^2 = \frac{1}{2}(5 - 0.5)^2 = 10.125$

   $\Delta w = \alpha(o - t)i = 0.1(0.5 - 5)2.5 = -1.125$

$$w_n = w - \Delta w = 0.2 - (-1.125) = 1.325$$

3. $(i, t) = (2, 4)$, $w = 1.325$ and $\alpha = 0.1$

$$o = iw = 2 \cdot 1.325 = 2.65$$
$$C(t, o) = \frac{1}{2}(t - o)^2 = \frac{1}{2}(4 - 2.65)^2 = 0.91125$$
$$\Delta w = \alpha(o - t)i = 0.1(2.65 - 4)2 = -0.27$$
$$w_n = w - \Delta w = 1.325 - (-0.27) = 1.595$$

To achieve better results, more iterations of training would be needed (we only cycled through the data once), but you can see that the weight value increased from 0 to 1.595, which is closer to the most optimal value of 2. To actually be able to say our network has improved, we need to compare the error value of our updated network with that of our original network on the same input (ideally, you should compare networks by their average error per sample, but it is simpler to just compare one particular sample's error for this demonstration):

$$(i, t) = (1, 2), w = 0$$
$$o = iw = 1 \cdot 0 = 0$$
$$C(t, o) = \frac{1}{2}(t - o)^2 = \frac{1}{2}(2 - 0)^2 = 2$$

$$(i, t) = (1, 2), w = 1.595$$
$$o = iw = 1 \cdot 1.595 = 1.595$$
$$C(t, o) = \frac{1}{2}(t - o)^2 = \frac{1}{2}(2 - 1.595)^2 = 0.0820125$$

As you can see, the error of our updated network is lower than our original network, so our network improved! Note that the entire notion of updating parameters in this manner is called *gradient descent*. Normally in gradient

descent, an entire set of inputs are used to calculate an average error, which is then used to perform one single weight update. Our version of gradient descent (where we perform updates after calculating the error for each example) is called *stochastic gradient descent*, whereas there is also *mini-batch gradient descent* (where updates are performed after an averaged error over a batch of inputs).

Now, let's review the delta rule and interpret it in another perspective. As a reminder, the general delta rule is expressed mathematically as:

$$\Delta w = \alpha(y - t)\,\sigma'(h)\,x.$$

$$w_t = w_{t-1} - \Delta w$$

In the first equation above, Δw signifies the change calculated for each of the individual weights or biases in the network. α refers to the *learning rate*, which is a constant that is multiplied with the other values. The learning rate helps to increase or decrease the change in weights or biases depending on whether it is bigger than 1 or smaller. This can be thought of as the length of step you take as you try to run down a valley. Your goal is to get to the bottom, so you cannot take massive steps, or else you will never reach that minimum point (you'll keep overrunning and end up on the opposite sloping side). You cannot take tiny steps either because it will take you an unimaginably long time to get to the lowest point in the valley. The reason why I use a valley as an analogy for the learning rate instead of a mountain is that data scientists like to think of machine learning as a method that leads you to the lowest point on a function ("valley") or the minima of the graph, which best fits the appropriate case, giving the best result. Some functions may also have several "peaks" (also called maxima) and "valleys" (minima), so the highest "peak" is usually called the

global maximum, and the deepest "valley" is called the global minimum. Here is an example of what I mean:

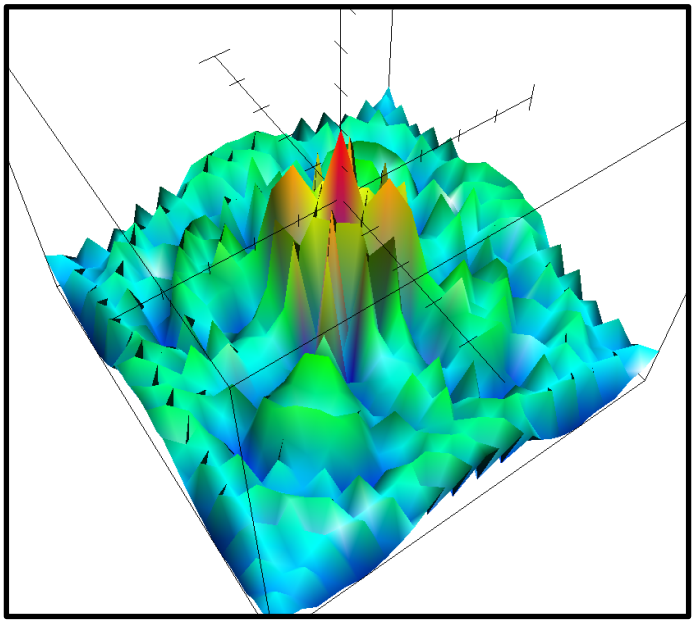


Figure 4-3: *Multivariable Graph Result for the equation*:

$$z = 0.5 + \frac{\cos(\sin(|y^2 - x^2|))^2 - 0.5}{[1 + 0.001(y^2 + x^2)]^2} + 0.2 \times \cos(xy)^2$$

Any other "peaks" or "valleys" that are not the highest and lowest respectively are called local maximums and minimums respectively. The t and y variables in the equation stand for the target value or expected value and predicted value $\left(\text{the output of the network}\right)$ and their difference is usually referred to as the *training error*. $\sigma'(h)$ stands for the derivative of the activation function on a particular value h which is the weighted sum of inputs. Finally, x stands for the input to the neuron that multiplied it with the weight. The second equation shows that the next generation or iterations' weight should be the difference between the original weight and the change in the weight calculated in the first

equation. The delta rule is sometimes simplified when the activation function is the identity activation function (linear) as:

$$\Delta w = \alpha(t - y)\, x.$$

So, here are the steps in machine learning:

1. *Initialization*: The step where all weights, biases are initialized using Gaussian distribution or assigned a value of 0.
2. *Forward-propagation*: The step where the input is passed through the network once to get the output of the network at the current stage.
3. *Backpropagation* or *learning*: The step where parameters are adjusted to find the most optimal in the current input's case. May take the average change required for multiple training data inputs. For single-layer neural networks, the *delta rule* is commonly used, which takes the product of the training error of the current input set, a learning rate, and input at the current index.

Note that the learning process should be repeated a certain amount of times (called *epoch*), for which all input sets are used for learning a certain number of times.

Chapter Summary

1. The goal of learning algorithms is to find the most suitable values for multiple different variables or unknown values in a perceptron, including weights, biases, and threshold value.

2. The first pass through a network before learning is called *forward propagation*.

3. Unknown values need to be initialized before *forward propagating*.

4. Usually weights, biases and thresholds are assigned a value of 0 or randomly distributed values using Gaussian distribution. Usually, people assign all variables a value of 0 to compare different models against each other in terms of training and learning efficiency.

5. The delta rule is used for single-layer neural networks to learn. Equation: $\Delta w = \alpha(t - y)\,\sigma'(h)\,x$. Δw signifies the change for the current weight, α represents the learning rate, t and y represent the target and predicted value respectively, $\sigma'(h)$ represents the derivative of the function at h, and x is the input value to the neuron.

6. The learning rate helps to increase or decrease the change in weights or biases depending on whether it is bigger than 1 or smaller. This can be thought of as the length of step you take as you try to run down a valley. Your goal is to get to the bottom, so you cannot take massive steps, or else you will never reach that minimum point. You cannot take tiny steps either because it will take you an unimaginably long time to get to the lowest point in the valley.

7. Data scientists like to think of machine learning as a method that leads you to the lowest point on a function ("valley") or the minima of the graph, which gives the best overall result. Some functions may also have

several "peaks" (also called maxima) and "valleys" (minima), so the highest "peak" is usually called the global maximum, and the deepest "valley" is called the global minimum. Any other "peaks" or "valleys" that are not the highest and lowest respectively are called local maximums and minimums respectively.

8. *Epoch* refers to the number of cycles a learning network runs through the *entire* dataset.

9. Steps to machine learning:
    a. *Initialization*: The step where all weights, biases are initialized using Gaussian distribution or assigned a value of 0.
    b. *Forward-propagation*: The step where the input is passed through the network once to get the output of the network at the current stage.
    c. *Backpropagation* or *learning*: The step where parameters are adjusted to find the most optimal in the current input's case. May take the average change required for multiple training data inputs. For single-layer neural networks, the *delta rule* is commonly used, which takes the product of the training error of the current input set, a learning rate, and input at the current index.
10. In gradient descent, an entire set of inputs are used to calculate an average error, which is then used to perform one single weight update. Another version of gradient descent (where we perform updates after

calculating the error for each example) is called stochastic gradient descent. There is also mini-batch gradient descent (where updates are performed after an averaged error over a batch of inputs).

Task 1 - Recognizing Digits

> *"And, now that you don't have to be perfect you can be good."*
>
> *— East of Eden by John Steinbeck*

That was a *lot* of theoretical information. Now, it is just about time to start on our first project! I strongly recommend you try to solve the problem before I provide a solution. Once again, all solutions are in Java, and pseudocode will be used otherwise. Usually, I will state the task requirements and solution steps before providing the solution so you can try and make the program yourselves before looking at my solution. My solution may not be the best, most accurate, or efficient, but should give you an understanding of how all the previous concepts you learned work together. Just note that this task won't implement biases. Let's start!

Task Description

Goal

Recognize digits 0-9 that are represented by a 5x3 (row * column) grid.

Background

This task requires knowledge of activation functions, the delta learning rule and the perceptron model.

Description

All digits from 0-9 can be represented as a matrix that is 5 x 3, so digits should look roughly like the following:

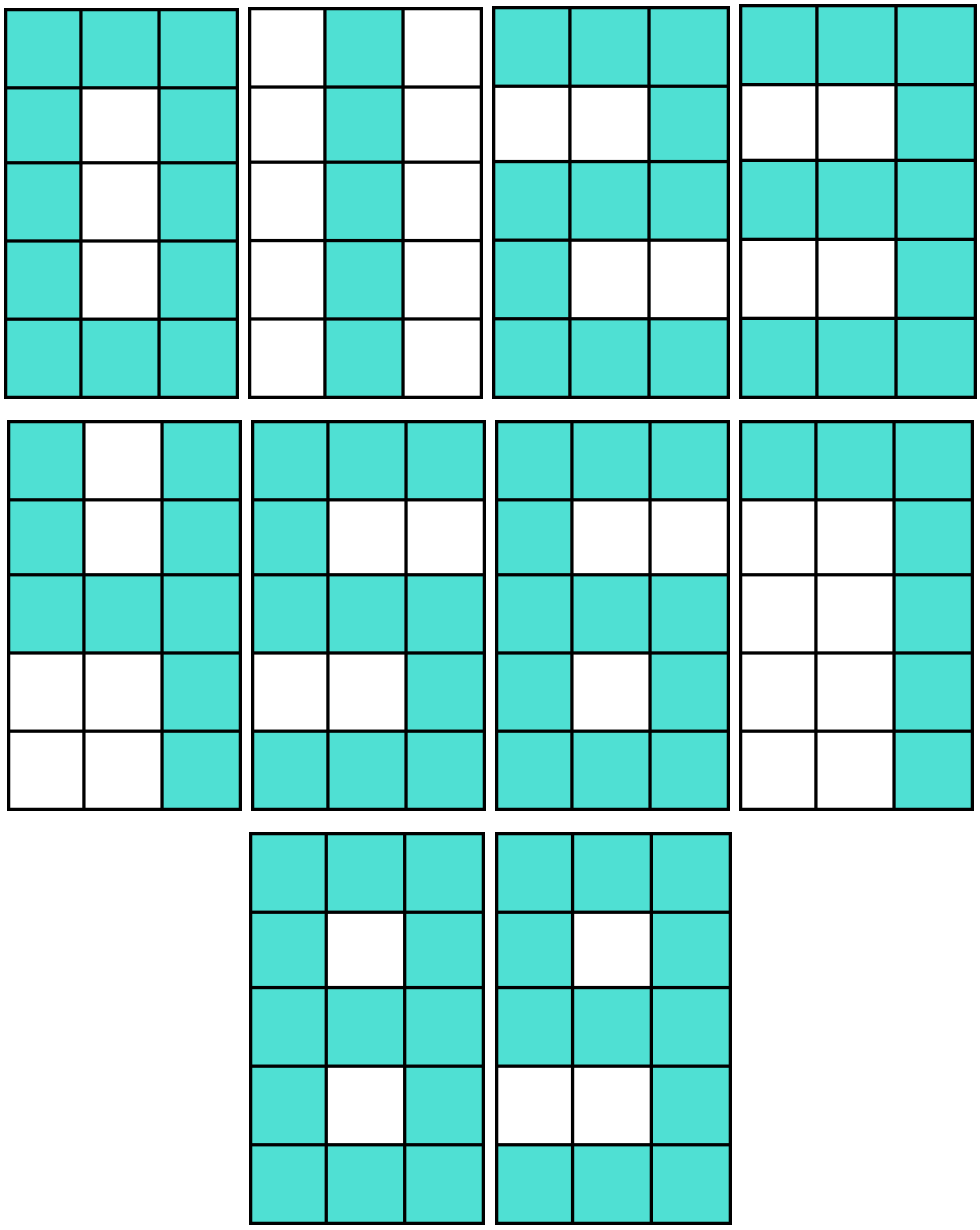


Figure T1-1: A visual representation of 5x3 grid digits.

In the pictures, the grids colored blue represent 1s and white represent 0s. The network should learn from the above samples and be able to classify unseen data. Something that may be classified as unseen data looks like:

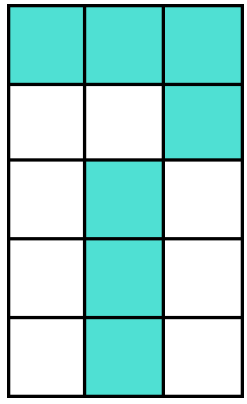


Figure T1-2: *Another digit 7 that should be used to test the network*

You may now be wondering: how do we represent images as numbers? Well, given that each blue square is a 1 and each white square is a 0, the 7 above looks like: [1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0]. Likewise, we can represent all other digits in the same way:

0: [1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1],

1: [0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0],

2: [1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1],

3: [1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1],

4: [1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1],

5: [1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1],

6: [1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1],

7: [1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1],

8: [1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1],

9: [1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1].

So, the above set of arrays should be our *input* array. Next, we need to define a *learning rate, input neuron number, output neuron number, target array* and *epoch*. For our example (after I experimented around a while), here are the optimal values(that work as expected):

Learning rate (α or η): 0.5

Epoch: 1000

Input Neuron Number: 15(5 * 3 = 15, one for each square)

Output Neuron Number: 10(0-9, one for each digit)

And the *target array* should be a set of arrays that contain the correct prediction:

0: [1, 0, 0, 0, 0, 0, 0, 0, 0, 0],

1: [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],

2: [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],

3: [0, 0, 0, 1, 0, 0, 0, 0, 0, 0],

4: [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],

5: [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],

6: [0, 0, 0, 0, 0, 0, 1, 0, 0, 0],

7: [0, 0, 0, 0, 0, 0, 0, 1, 0, 0],

8: [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],

9: [0, 0, 0, 0, 0, 0, 0, 0, 0, 1].

Our neuron model will not have a *bias* or *threshold,* but instead use an *activation function*.

Now you should be able to program the perceptron! You may still feel unconfident, so I will list a few steps below to do it all in a systematic manner so as to not forget any of the components for our digit recognizing network. Try to make your network adaptable, meaning that it can be easily converted to train another model for something else.

Steps to Take

1. First, design an object called `Layer`, with a constructor that takes in a *learning rate*, the number of neurons in the current and next layer and the next layer as an object. Make the `Layer` class keep track of the number of neurons in the next and current layer, neurons in the layer (as objects of `Neuron` class), the learning rate, and the next layer. Call the `initialise()` function in the Neuron class once the values are kept. An `initialise()` method in the Layer class should call an `initialise()` method in the Neuron class, and use `connect()` (in the Neuron class which we define later) to connect the neuron to all neurons in the next layer.

2. Design another constructor for Layer for output layers, which should only require the number of neurons in the current layer. Call a different initializing method called `outputInit()`.

3. Create another class called Neuron. Make the Neuron class have a constructor that requires the number of neurons the current neuron is

connected to. This constructor is for hidden layer and input layer neurons.

4. Then create another constructor in class Neuron, which accepts nothing at all. This is for the output layer neurons.

5. Create two initialization methods in the Neuron class. One initializes an array of weights using Gaussian distribution(for input and hidden layer neurons), and one that initializes output neurons, which should have a weight of 1 and the result of the neuron should only have an array length of 1.

6. The `outputInit()` function in the Layer class should initialise an array of output neurons using `new Neuron()`, the constructor that we created for output neurons.

7. Make the Neuron class keep track of the neurons that the current neuron connects to using an array of Neurons. Then create a function called `connect(Neuron neuron)`, which accepts a neuron as input and adds this to the array of neurons the neuron connects to.

8. Create a function called `setInput(double input)`, that takes in a number as input. Then, add this to a number stored locally because it will be used later. Along with it, create a `getInput()` function that returns the input to the caller.

9. Create a function called `finalise()`, which runs the sigmoid activation function on the input.

10. Create a function called `clear()`, which will reset the input to the neuron.

11. Create a function called `step()`, which takes the input to the neuron and multiply it with the array of weights. If the neuron connects to another layer, call the `setInput()` function of that neuron and set its input with the output of the neuron *to* the neuron.

12. To allow the Layer class to adjust the weights in the current Neuron, create another function called `adjustWeight(double[] adjustment)`, which takes an array of numbers and adds it to each weight in the neuron.

13. Finally for the Neuron class, create three functions: `getResult(int index)`, `getResult()`, and `toString()`. The first function takes an index and gets the result(from `step()`) at the index, the second function returns the entire result array, and the third prints the result array as a string(for debugging purposes).

14. (Back to Layer class): create `inputFired()`, `setInput(double[] input)`, `setTarget(double[]`

`target)`, and `clear()` functions. `inputFired()` calls the `finalise()` method in Neuron class for each neuron in the layer, `setInput(double[] input)` sets the input for the Layer, and `setTarget(double[] target)` sets the target for the neurons in the output layer. Lastly, `clear()` calls the `clear()` function for each neuron in the next and current layer.

15. Forward propagation: create a method `step()`. In the method, set the input for each neuron in the current layer and call the `step()` function for the neuron. Then, add the result of the neuron to a list and repeat until all neurons have fired.

16. Learning: create a method `learn()`. First call the `step()` method. Then, initialize a multidimensional array to contain the change in weights for each neuron in the current layer. Apply the delta rule to each input value and weight, then use the `adjustWeight(double[] adjustment)` function in the Neuron class to adjust the neuron's weights accordingly. Finally, call the `clear()` function to prepare the network for learning another set of data.

17. Testing: create a function `test(double[] input)` that tests a certain input set and returns the result as an array of values.

In here, I have used two peculiar terms: `double` and `int`. These are basically just two data types that Java uses, somewhat like what we call (mathematically) *rational numbers* and *whole numbers*. I also use the symbol `[]`, which basically just means an array or matrix of the data type in front of the brackets.

Solution

First, I will show the entire code solution, which can be found with this book. There are two versions: one with extensive comments in a folder with path `Project1/Code/Task/Commented`, and one without comments in a folder with path `Project1/Code/Task/Clean`. I will show you a clean version and walk you through it by using the comments along with a further explanation. The entire project only takes up 280 lines of code, which is considerably small for a program that is doing something so *seemingly* complex. You can see the code below according to file name:

`Layer.java`

```
001    public class Layer {
002      private int neuronNumber;
003      private int nextNeuronNumber;
004      private Neuron[] neurons;
005      private Layer nextLayer;
006      private double[] input;
007      private double[] result;
008      private double[] target;
009      private double learningRate;
010
011      public Layer(int neuronNumber) {
012        this.neuronNumber = neuronNumber;
013        neurons = new Neuron[neuronNumber];
014        result = new double[nextNeuronNumber];
015        outputInit();
```

```
016       }
017
018       public Layer(int neuronNumber, int next,
                  Layer nextLayer, double learningRate) {
019         this.neuronNumber = neuronNumber;
020         nextNeuronNumber = next;
021         neurons = new Neuron[neuronNumber];
022         this.learningRate = learningRate;
023         result = new double[nextNeuronNumber];
024         this.nextLayer = nextLayer;
025         initialise();
026       }
027
028       private void inputFired() {
029         for (Neuron x : neurons) {
030           x.finalise();
031         }
032       }
033
034       public void setInput(double[] input) {
035         this.input = input;
036       }
037
038       public void setTarget(double[] target) {
039         this.target = target;
040       }
041
042       private void outputInit() {
043         for (int i = 0; i < neuronNumber; i++) {
044           neurons[i] = new Neuron();
045           neurons[i].initialise();
046         }
047       }
048
049       private void initialise() {
050         for (int i = 0; i < neuronNumber; i++) {
051           neurons[i] = new Neuron(nextNeuronNumber);
052           neurons[i].initialise();
053         }
054
055         if (nextLayer != null) {
056           for (int j = 0; j < neuronNumber; j++) {
```

```
057             for (int k = 0; k < nextNeuronNumber; k++) {
058              neurons[j].connect(nextLayer.getNeurons()[k]);
059             }
060           }
061         }
062       }
063
064       public void learn() {
065         step();
066         double[][] changeWeights
              = new double[neurons.length][nextNeuronNumber];
067         for (int i = 0; i < nextNeuronNumber; i++) {
068           for (int j = 0; j < neurons.length; j++) {
069             changeWeights[j][i]
                = learningRate * input[j] *
                  (target[i] - result[i]) *
                  derivedSigmoid(neurons[j].getInput());
070           }
071         }
072
073         for (int index = 0;
                    index < neurons.length; index++) {
074          neurons[index].adjustWeight(changeWeights[index]);
075         }
076
077         clear();
078       }
079
080       private void step() {
081         for (int i = 0; i < input.length; i++) {
082           neurons[i].setInput(input[i]);
083           neurons[i].step();
084           for (int j = 0; j < result.length; j++) {
085             result[j] += neurons[i].getResult()[j];
086           }
087         }
088         result = sigmoid(result);
089         nextLayer.inputFired();
090       }
091
092       public double[] test(double[] input) {
093         for (int i = 0; i < input.length; i++) {
```

```
094           neurons[i].setInput(input[i]);
095           neurons[i].step();
096           for (int j = 0; j < result.length; j++) {
097             result[j] += neurons[i].getResult()[j];
098           }
099         }
100         result = sigmoid(result);
101         clear();
102         return result;
103       }
104
105       private void clear() {
106         for (Neuron neuron : neurons) {
107           neuron.clear();
108         }
109
110         for (Neuron neuron : nextLayer.getNeurons()) {
111           neuron.clear();
112         }
113       }
114
115       private double[] sigmoid(double[] result) {
116         for (int i = 0; i < result.length; i++) {
117           result[i] = sigmoid(result[i]);
118         }
119         return result;
120       }
121
122       private double sigmoid(double result) {
123         return 1 / (1 + Math.exp(-result));
124       }
125
126       private double derivedSigmoid(double x) {
127         return sigmoid(x) * (1 - sigmoid(x));
128       }
129
130       private Neuron[] getNeurons() {
131         return neurons;
132       }
133     }
```

Whoa! Let's first start with an explanation of what is happening before we continue.

Line 1: This will be our layer class, which does all the work for machine learning.

Line 2: Store the number of neurons in this layer. Used during for-loops.

Line 3: Store the number of neurons in the next layer.

Line 4: The neurons in this layer as an array of `Neuron` objects.

Line 5: Keep the next layer's instance/object reference point.

Line 6: This stores the input, or current dataset that is being used to train or test the neural network.

Line 7: Stores the result from the output neurons from `step()` method/function.

Line 8: Stores the best result for each output neuron for the current input set.

Line 9: Stores the learning rate or constant that is multiplied with the rest of the calculations to increase/decrease the step taken/adjustment made to prevent missing a minima.

Line 11: First constructor option for output layer. Output layers are not a separate class but are initialized differently. Output layers only require the neuron number in the layer.

Line 18: Second constructor option for hidden and input layers. Requires neuron number in current layer, next layer, the next layer's reference.

Line 28: After stepping through the network, each neuron will have a sum of neuron outputs as input, but sigmoid function is not taken yet, so we finalise all the neurons' inputs before stepping through the layer.

Line 34: Set the input to the layer for current iteration of learning. The input is given as an array, which should have the same length as the number of neurons in the current layer.

Line 38: Set the target, which is an array with zeros and ones. The ones indicate the neuron that should fire if the prediction is correct. See the 'test' class for the target array.

Line 42: Initialise the neurons for an output neuron, which is constructed differently from the hidden and input neurons.

Line 49: Initialise the neurons normally(for input and hidden layer neurons), we connect the neurons to the next layer's neurons here as well.

Line 64: Learning method. First, step through the layer with inputs using the `step()` method. Then, we initialise a multidimensional array that contains the changes for each neuron. Then, we use the delta rule to calculate the change in weights for each neuron's weights. I use the general delta rule formula, which multiplies the product with an

additional number which is the derived sigmoid given an input of the weighted sum of inputs to a neuron. Then, we give each neuron a list of changes that are added to the weight for the next epoch. Then, we perform the `clear()` function which uses the `clear()` in the `Neuron` class for each neuron in the layer. This makes it ready for another epoch of learning.

Line 80: Step function named the same as one in the `Neuron` class. Sets the input for the neuron using the input array initialised with the constructor and calls the `step()` function on the neuron. Then, the result array gets the value of the output of the neuron, and is run through the Sigmoid activation function. Then call the `inputFired()` function for the next layer.

Line 92: After learning, provide a method to test the network on a given input set/array. Basically, it performs a `step()` on the network, and prevents it from getting ready to learn the sample. Then, it calls `clear()` to get the network ready for another round of recognizing if needed. Then, it returns the result as an array to the caller.

Line 105: Clear the inputs and indexes kept in the current and next layer.

Line 115: Take in an array of values and run the sigmoid activation function calculations on each of them, then return the array to the caller.

Line 122: Method overriding, used in the `sigmoid()` method above for calculating sigmoid of single value. Returns the value back to the caller.

Line 126: This method returns the value of the derived sigmoid function calculated on an input.

Line 130: Return the neurons in the current layer to the caller. Used for the layer to access the next layer's neurons.

Neuron.java

```
01    import java.util.Arrays;
02    import java.util.Random;
03
04    public class Neuron {
05      Random rand = new Random();
06      Neuron[] connectTo;
07      double[] weight;
08      double input;
09      double[] result;
10      int index = 0;
11
12      public Neuron() {
13        weight = new double[]{1};
14        result = new double[1];
15      }
16
17      public Neuron(int connectCount) {
18        connectTo = new Neuron[connectCount];
19        weight = new double[connectCount];
20        result = new double[connectCount];
21        initialise();
22      }
23
24      public void connect(Neuron neuron) {
25        connectTo[index] = neuron;
```

```
26          index++;
27        }
28
29        public void setInput(double input) {
30          this.input += input;
31        }
32
33        public double getInput() {
34          return input;
35        }
36
37        public void finalise() {
38          input = sigmoid(this.input);
39        }
40
41        public double sigmoid(double input) {
42          return 1 / (1 + Math.exp(-input));
43        }
44
45        public void initialise() {
46          for (int i = 0; i < weight.length; i++) {
47            weight[i] = rand.nextGaussian();
48          }
49        }
50
51        public void step() {
52          for (int i = 0; i < weight.length; i++) {
53            result[i] = input * weight[i];
54          }
55
56          if (connectTo != null) {
57            for (int index = 0;
                       index < connectTo.length; index++) {
58              connectTo[index]
                       .setInput(this.getResult(index));
59            }
60          }
61        }
62
63        public void adjustWeight(double[] adjustment) {
64          for (int i = 0; i < weight.length; i++) {
65            weight[i] += adjustment[i];
```

```
66          }
67        }
68
69        public double getResult(int index) {
70          return this.result[index];
71        }
72
73        public double[] getResult() {
74          return this.result;
75        }
76
77        public void clear() {
78          input = 0;
79          index = 0;
80        }
81
82        @Override
83        public String toString() {
84          return Arrays.toString(result);
85        }
86      }
```

Explanation:

Line 1: Java Utilities Array import: used 1 time in `@Override toString()` method below `clear()`.

Line 2: Gaussian distribution for weights is done in `initialise()`, requires the Random package.

Line 4: This will be our `Neuron` class, based on the perceptron. Note that I have not included the bias component or threshold component. They are not necessary for the network to classify digits extremely accurately.

Line 5: Random instance for entire class. This makes the class more efficient because a new Random instance is not initialized/created for every new weight when finding a Gaussian-distributed value for it.

Line 6: This array stores the `Neuron` objects that the current neuron connects to in the next layer. This is used to directly transfer the result to the neuron instead of implementing it in the `Layer` class. The connections are made in the `initialise()` function in the `Layer` class.

Line 7: This stores the weight values that connect the current neuron to a neuron in the next layer according to index. The weights are initialized using Gaussian distribution in this program.

Line 8: I store all inputs as a single variable. If this neuron is the input neuron, a single value is passed to it. If it is from a hidden or output layer, the input is added to it. Then, the `Layer` class calls the `finalise()` method, when all neurons have fired their inputs to the next layer's neurons which runs the Sigmoid activation function on the overall input.

Line 9: The result from firing the input into the `Neuron` is stored as an array to each of the neurons this neuron is connected to. Then, each of them is given to the next neuron using the `setInput()` method.

Line 10: I put a global 'index' variable to store the connections the current neuron has with the next layer's neurons. It is incremented, and just in case there is some `ArrayIndexOutOfBoundsException`, I use a `clear()` method, which is called after stepping through the neuron. The `clear()` method also clears the input to the neuron so the input doesn't accumulate throughout the epochs for training the network.

Line 12: Initialise `Neuron`, providing nothing. This is how the `Layer` class initializes the output layer's neurons. Other neurons are initialized using the other constructor.

Line 17: This is how hidden and input layers' neurons are initialized. The arrays are given lengths and initialized using the `initialise()` function.

Line 24: The layer class uses this method to connect neurons to other neurons. Another alternative is to get a `Neuron` array (`Neuron[]`), and set the array here instead. In that case, the 'index' global variable can be removed.

Line 29: The input layer neurons are given a single input, but the output and hidden layer neurons are given multiple. These have to be added, and later finalized(take the sigmoid of the number, using the activation function in `finalise()`).

Line 33: This method isn't used in the class, but the `Layer` class needs it to perform the learning(delta rule formula) on the neuron's weights.

Line 37: The function simply makes the input fit between 0 and 1, by passing it through the Sigmoid activation function.

Line 41: Sigmoid activation function, you can find the formula in Chapter 3. `Math.exp(-x)` is used to express e to the power of $-x$, which is Euler's number to the power of negative x.

Line 45: Here, weights are initialized using Gaussian distribution. The Random import class provides the `nextGaussian()` method which generates Gaussian-distributed numbers.

Line 51: Probably the most complicated function in the entire class, it is quite self-explanatory. The input and weight variables have been initialized, so we calculate the result array's values, then, we pass these values to each of the neuron's connected neurons in the next layer if there is a set of connections for the neuron. The if condition checks this because output neurons are not a separate class, but initialized differently(I provided two constructor methods) without the `connectTo` array with connections(output is network's output, not connected to another layer of neurons).

Line 63: The `adjustWeight()` function is put here to reduce the amount of work the `Layer` class needs to do. An array of adjustments are given after the `Layer` class runs through the delta rule in the `learn()` function. The values are added to the weight. (not subtracted)

Line 69: The current class uses it to assign input values together to each of the neurons the neuron is connected to in the `step()` method. Note that it gets the result at a single index in the result array.

Line 73: The `Layer` class needs the result of a neuron to compute changes for the neuron's weights using the delta learning rule in the `learn()` function. Note that the entire array is returned, so that the for-loop in the `Layer` class can use the values.

Line 77: Simply clears the 'input' variable for a new epoch of learning. The 'index' variable is cleared in case of future adjustments that may require it.

Line 82: Overrides the `Object.toString()` method to return something more meaningful. Used for debugging and testing/ implementing like in the 'test2.java' class.

test2.java

```
01    public class test2 {
02      public static void main(String[] args) {
03        final double[][] target = {
04          {1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
05          {0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
06          {0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
07          {0, 0, 0, 1, 0, 0, 0, 0, 0, 0},
08          {0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
09          {0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
10          {0, 0, 0, 0, 0, 0, 1, 0, 0, 0},
11          {0, 0, 0, 0, 0, 0, 0, 1, 0, 0},
12          {0, 0, 0, 0, 0, 0, 0, 0, 1, 0},
13          {0, 0, 0, 0, 0, 0, 0, 0, 0, 1}
14        };
15
16        final double[][] inputs = {
17          {1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1},
18          {0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0},
19          {1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1},
20          {1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1},
21          {1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1},
22          {1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1},
23          {1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1},
24          {1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1},
25          {1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1},
26          {1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1}
27        };
28
29        Layer nextLayer = new Layer(10);
30        Layer layer = new Layer(15, 10, nextLayer, 0.5);
31
32        for (int j = 0; j < 1000; j++) {
33          for (int i = 0; i < inputs.length; i++) {
34            layer.setInput(inputs[i]);
```

```
35            layer.setTarget(target[i]);
36            layer.learn();
37          }
38        }
39
40        for (double[] input : inputs) {
41          System.out.println(
                getBestGuess(layer.test(input)));
42        }
43      }
44
45      public static int getBestGuess(double[] result) {
46        double k = Integer.MIN_VALUE;
47        double index = 0;
48        int current = 0;
49        for (double a : result) {
50          if (k < a) {
51            k = a;
52            index = current;
53          }
54          current++;
55        }
56
57        return (int) index;
58      }
59    }
```

Explanation:

Line 1:

In this class, we will test our neural network.

Goal: Classify digits 0-9.

Requires: learning rate, number of neurons, target values, input values.

Line 3: Here, we store the desired result, where all neurons should ideally output 0, while the correct neuron should output a 1.

Line 16: Store our training dataset as a multidimensional array, where the numbers should form a 5x3(row * column) grid that looks like a digit respectively. Stored from 0-9 to make it easier to retrieve the output of the network as a result, not array in the `getBestGuess()` method.

Line 29: Initialise the layers from output layer to input layer, output layer doesn't require a layer to be connected with it. The hidden and input layers should be connected to one another.

Line 32:

> Epoch: 1000.
>
> Objective: make the network learn the dataset with 10 sets of inputs. Roughly 1000 epochs are needed to make the desired output neuron output 0.9 or higher. The input is set using `setInput()` and target using `setTarget()`. The layer learns when the `learn()` function is called.

Line 40: View our result! Run through each of the input sets and see our network's classification result. Should look like:

```
0
1
2
3
4
5
6
7
8
9
Process finished with exit code 0
```

Line 45: Get the answer. E.g., 5, 3, 4... instead of an array of values between 0 and 1. This method is not stored in the `Layer` class because it is more convenient for the programmer to define the output of the network given a set of values that each output neuron outputs.

Well, now we have officially completed our *first* challenge of making a digit classifier! You may have failed to create the network on your own…, but I hope you now understand where the issue was! Now, I wager that some of you may feel slightly disappointed: what does this have to do with digit recognition in the *practical* world? We rarely will *ever* see 5x3 *digits* again, so how does our code do anything useful? Well, I provided a solution that is slightly configurable, meaning you can change it slightly to *solve* a different problem, so therefore, you can just tweak the code slightly, and you will be able to use it to train a model on real-world handwritten digits! At this point, several questions pop up: how do we save our model for use? How can we feed our network data that has a different number of inputs(image size is different)? Where can we *get* data to train our network? How can we visualize the weights in our network? I will try to answer these as well as I can later in the book.

Common Questions

Saving Models

One of the prominent reasons why I chose Java to write all the code in this book is because Java provides a method for saving objects called *serialization*

and one called *deserialization* to extract objects from the saved files. Basically, it saves an object into a file with extension name (usually) *.ser*, which can be exported to other computers and locations for deployment. Below you can find the code to do just that, in five lines:

```
1     try (ObjectOutputStream oos =
            new ObjectOutputStream(
            new BufferedOutputStream(
            new FileOutputStream("weights.ser")))) {
2       oos.writeObject(layer);
3     } catch (IOException ex) {
4       ex.printStackTrace();
5     }
```

Apart from that, you will need to change the line of code in each class that defines the class to:

```
1     public class Layer implements Serializable {
```

And:

```
1     public class Neuron implements Serializable {
```

To extract the object from the file, you use:

```
1     ObjectInputStream ois =
       new ObjectInputStream(
       new BufferedInputStream(
       new FileInputStream("weights.ser")));
2     Layer lay = (Layer) ois.readObject();
```

For those, who are implementing this program perhaps in another language, or are just curious how objects can be saved to files using other methods, you can simply write each index in the array on a new line, or write each array in the multidimensional array onto the same line, and read the file similarly. If you aren't satisfied with these methods… experiment! See what strategies or methods you find most convenient or fastest.

Feeding Networks Different Input Lengths

Now comes a harder question: how do we feed our network inputs with different lengths? As seen before in the previous example, there were strictly only to be 15 inputs to the network, with 10 outputs. So, what if we wanted to ask a 15-10 network the classification of an input set with length 20? This problem is very practical. For example, with our previous example, we trained a network to classify digits of length 15, but in reality, no pictures will be of length 15. Pictures can come in all sizes, if the user resizes them, our network should still give a prediction for the input. So how do we solve this problem? Turns out, people around the world figured a simple yet effective way around the problem: resizing or cropping. Basically, you resize the image using various algorithms or built-in classes or crop the image and center the digit to be recognized. In Java, there is a class with deals with resizing operations and can be used like this:

```
01    File input = new File(filepath);
02    image = ImageIO.read(input);
03    Image newImage =
           image.getScaledInstance(28, 28,
                                   Image.SCALE_SMOOTH);
04    BufferedImage bimage =
        new BufferedImage(newImage.getWidth(null),
                          newImage.getHeight(null),
                          BufferedImage.TYPE_INT_RGB);
05    Graphics2D bGr = bimage.createGraphics();
06    bGr.drawImage(newImage, 0, 0, null);
07    bGr.dispose();
08    ImageIO.write(bimage, "png", new File("result.png"));
09    double[] a = new double[INPUT];
10    int counter = 0;
11    for(int i = 0; i < 28; i++) {
12        for(int j = 0; j < 28; j++) {
```

```
13              Color c = new Color(bimage.getRGB(j, i));
14              a[counter] =
                        255 - (c.getRed() * 0.30) -
                        (c.getGreen() * 0.59) -
                        (c.getBlue() * 0.11);
15              a[counter] /= 255;
16              a[counter] = Double.parseDouble(
                                    String.format("%.1f",
                                                a[counter]));
17              counter++;
18          }
19      }
```

What is happening is, an image's file path is given to the program. Then, the program uses `ImageIO.getScaledInstance()` and scales or resizes the image to the dimension 28 x 28. Then, the image is converted to a grayscale value between 0 and 1 and stored into `double[] a`, for use later on in the program. I actually took this from the code we will be creating to deploy our network in the real world, and slightly modified it for display on its own.

Finding Datasets For Model Training

As we have seen previously, we need data to be able to train a neural network, so where can we get manually labelled (100% accurately labelled) digit recognition data? Well, if you have tried to make models previously to recognize digits, you will have heard of the MNIST dataset, which is a dataset with a curation of 70000 images, 60000 of which are used to train the model, and 10000 of them are used to test the model's accuracy. Each input set consists of a 28 x 28 dimension array of numbers, or 784 numbers, along with a 785th, being the correct prediction of the example. You can access the website at http://yann.lecun.com/exdb/mnist/. Here is how the page should look like:

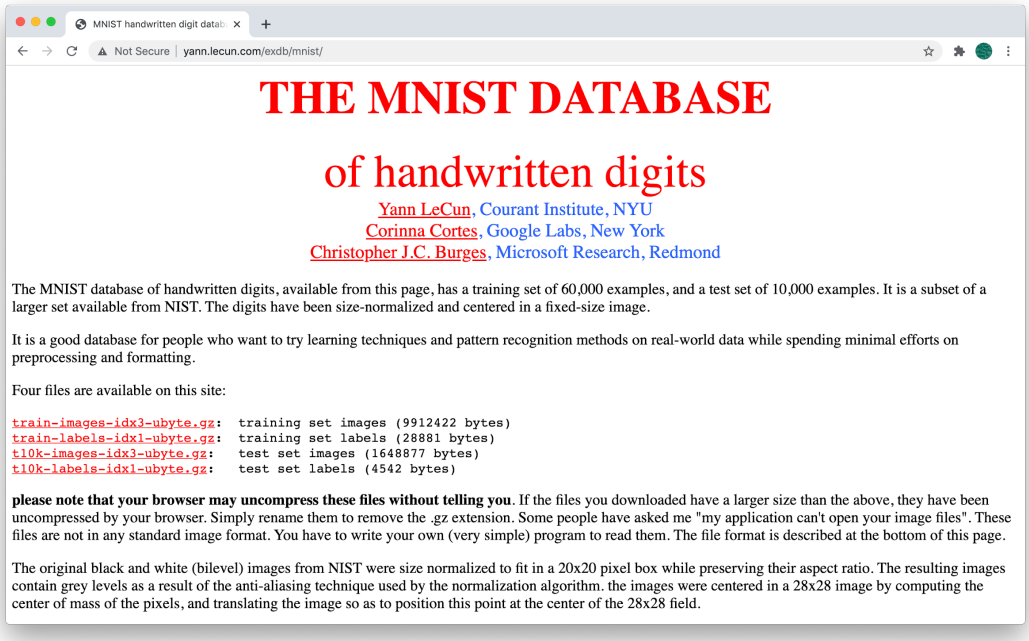


Figure T1-3: MNIST Dataset Website

You can directly download the dataset from their site, or you can go to https://github.com/JC-ProgJava/Building-Neural-Networks-From-Scratch/releases/download/v1.0/MNIST.zip, where I have uploaded the dataset as a series of 70000 files containing the numbers. Notice that the dataset of 70000 files are not images but greyscale-converted numbers. You can look online on websites like kaggle.com, to find the MNIST dataset as the images themselves, but the number version saves a little time in the computing process. This will be the dataset we use in the Challenge section.

You can also find datasets through https://datasetsearch.research.google.com/, which Google provides as a search engine for datasets. Here is what it looks like:

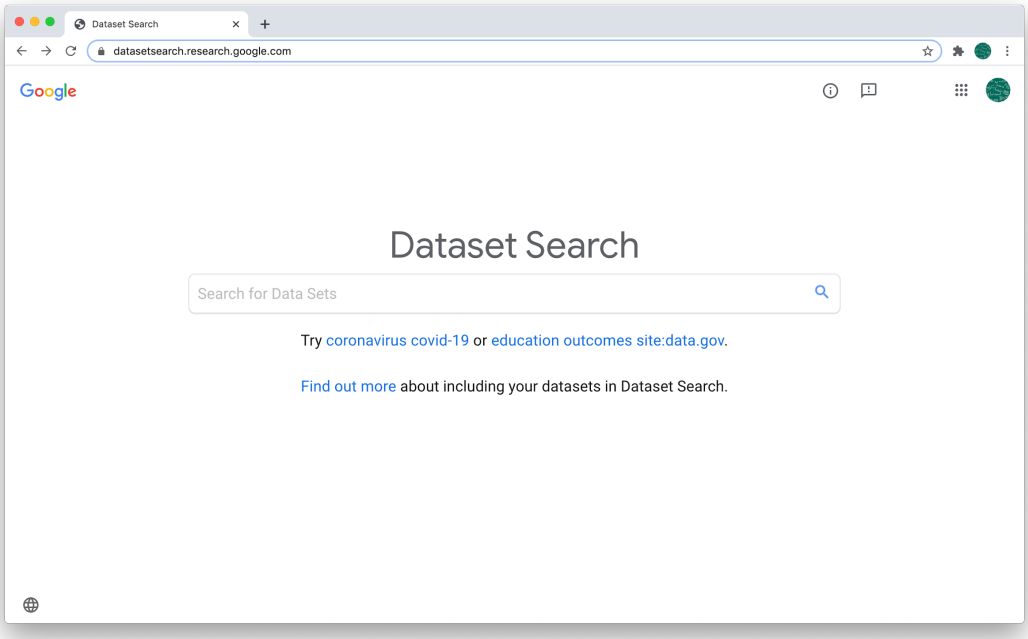


Figure T1-4: Google Dataset Search Homepage

You can search anything on there, and usually several relevant search results will appear.

I won't answer the last question yet, because I think it will be a little easier to explain after we finish with the challenge task. For now, head on to the next section, where we will be classifying digits of any input length and training with the MNIST dataset!

Challenge: A Real-World Application of Digit Recognition

Task Description

Goal

Step 1: Recognize digits 0-9 that are represented in a 28 x 28 (row x column) grid.

Step 2: Recognize digits 0-9 of any input length accurately.

Background

This task requires knowledge of activation functions, the delta learning rule and the perceptron model. Also, knowledge of Java (or C-like) programming may help you get through this part of the book more smoothly.

Description

The MNIST dataset represents digits in 28x28 grids. Here is what each digit's file may look like:

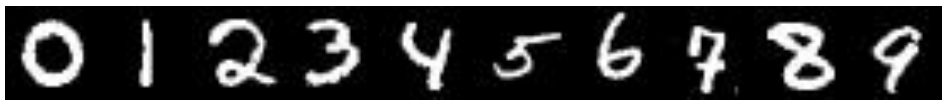


Figure T1-5: *Digits represented in 28x28 grids.*

Here, the white squares represent 1s and black represent 0s. Since the MNIST dataset consists of digits written by hundreds of people and *labelled manually* by people, and unrecognizable digits are not being removed, it is *impossible* to

recognize all testing data accurately. The highest accuracy achieved by data scientists was 99.67%(incorrect classification of 23 out of 10000 images). If you see models that get "100% accuracy on MNIST dataset", the model usually *overfits*, which is simply where the model starts making too detailed or sophisticated assumptions on how to separate different objects. For example, a model trying to classify apples and oranges is *overfitted* if it classifies oranges as "an object that is orange and perfectly round" just because the training dataset consists of hundreds of examples that look like this:

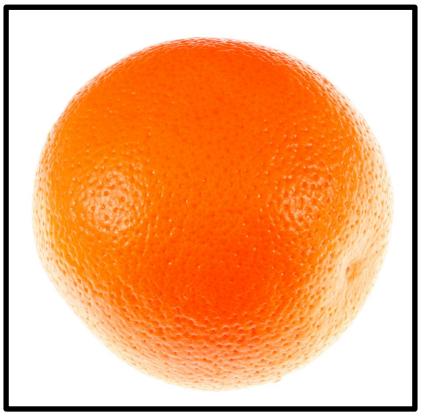


Figure T1-6: *A perfectly round orange that is orange in color*

The result to the network may have looked like this if the model overfitted: [1, 0], where the first index is the output for classifying oranges and the other index classifies apples. The following may be an image that the network can't classify correctly if *overfitted*:

Figure T1-7: *A pile of oval-like oranges that have a hint of green*

The reason why it can't classify the picture above is because:

1. The oranges are not round.
2. There are multiple oranges.
3. The orange color is different.
4. The oranges are slightly green.
5. The lighting of the picture is different from the training images.

Generally, you can tell the model *overfits*, if the following conditions are mostly satisfied:

- The training accuracy(accuracy of learning the training dataset) is much higher than the testing accuracy(accuracy of testing dataset, which the network has never seen before)
- The output of the network is too certain(seen above: [1, 0]).

More about *overfitting* and *underfitting* will be introduced in another chapter.

Therefore, we should be relatively happy with a model being able to recognize digits with 90%-98% accuracy. In reality, this should achieve a lower accuracy because images used to test aren't oriented the same way, centered, or have the same thickness(pens may be extra thick or thin). Finally, I should give you the values for some of the constants in the network:

Learning rate (α or η): 0.5

Epoch: 10

Input Neuron Number: 784(28 * 28 = 784, one for each input value in 28x28 grid input)

Output Neuron Number: 10(0-9, one for each digit)

Expected output accuracy: 50% or above (should be around 80%-90%, but 50% should really be the minimum by far, if you get around 10%, something is wrong with your training or testing class).

Steps to Take

1. Download the MNIST dataset from https://github.com/JC-ProgJava/Building-Neural-Networks-From-Scratch/releases/download/v1.0/MNIST.zip.

2. Unzip the zip file 'data.zip'. I would not recommend you open it because it consists of 70000 files with grayscale image values inside, which would probably crash your computer.

3. Put this folder in your project directory or somewhere where your program can reach.

4. Create a training class with any name, I will use '`application`'. In here, modify the `test2.java` class so that the program trains a model on 28x28 digits. As an extension, you can track the number of milliseconds or seconds the program takes to train the model, how can you decrease the time required?

5. To train the model, all the data needs to be imported into an array(from file 00001.txt to 60000.txt, leave the other 10000 files for testing the model later on). Do this, and then shuffle the array after each epoch of training(training for all samples in the *entire* dataset). In each input file, there are 785 inputs, with the last being the correct predicted answer. Fill an array with 0s and make the index of the correct answer be 1. This should be used in the `setTarget()` function that sets the target output value for the network.

6. Serialize the model and save it to a file with any name, I will use "`app.ser`".

7. Create a *testing* class, called "`test3.java`"(or anything else), where you take all files from 60001.txt to 70000.txt and test your trained model. Output this result(should only take a few seconds before the program terminates).

Estimating Computational Time

In the steps I listed, I added an additional task/challenge to measure the time it takes for the program to train the model. The reason why I do this is so you can get a rough understanding on how long it takes a computer to train a model on 60000 images for any given number of epochs. The task required 10 epochs at a learning rate of 0.5, which took `187015 milliseconds` (or roughly 3 minutes) for my computer to train, which runs with four 2.4 GHz Intel Core i5 cores and got an accuracy of 89.93%, slightly less than our goal of 90% or above. It took the program roughly 80 seconds to store all the inputs in an array, so `187015 - 80000` should give the time taken to train 10 epochs. $10e = 107015$, $e \approx 10700$. Each epoch took roughly 10.7 seconds (10700 milliseconds) to train. Therefore, if we tried to train 1000 epochs, it would take $10700 \times 1000 + 80000 = 10{,}780{,}000$ (we get the training time and then add on the time it takes to load the input sets into the array), or around 3 hours. That is a *lot* of computational time, requiring a *lot* of memory and power. I trained this on an IDE, which probably made the running time slightly slower than optimal, but results should be similar to the ones given above if you have a similar level of computational power. We could optimize our program slightly by serializing the array of inputs for later runs/executions of the program, which should cut the computational time down by around 77 seconds (serialization and deserialization processes both take several seconds as well) on average.

Solution

Okay! That was *quite* a racket. I will again show the entire code solution in Java, which can be found with this book on Github. The entire project only takes up 381 lines of code.

Layer.java

```
001    import java.io.Serializable;
002
003    public class Layer implements Serializable {
004      private final int neuronNumber;
005      private final Neuron[] neurons;
006      private int nextNeuronNumber;
007      private Layer nextLayer;
008      private double[] input;
009      private double[] result;
010      private double[] target;
011      private double learningRate;
012
013      Layer(int neuronNumber) {
014        this.neuronNumber = neuronNumber;
015        neurons = new Neuron[neuronNumber];
016        result = new double[nextNeuronNumber];
017        outputInit();
018      }
019
020      Layer(int neuronNumber, int next,
                 Layer nextLayer, double learningRate) {
021        this.neuronNumber = neuronNumber;
022        nextNeuronNumber = next;
023        neurons = new Neuron[neuronNumber];
024        this.learningRate = learningRate;
025        result = new double[nextNeuronNumber];
026        this.nextLayer = nextLayer;
027        initialise();
028      }
029
030      private void inputFired() {
```

```
031         for (Neuron x : neurons) {
032           x.finalise();
033         }
034       }
035
036       void setInput(double[] input) {
037         this.input = input;
038       }
039
040       void setTarget(double[] target) {
041         this.target = target;
042       }
043
044       public double getLearningRate() {
045         return learningRate;
046       }
047
048       public void setLearningRate(double learningRate) {
049         this.learningRate = learningRate;
050       }
051
052       private void outputInit() {
053         for (int i = 0; i < neuronNumber; i++) {
054           neurons[i] = new Neuron();
055           neurons[i].initialise();
056         }
057       }
058
059       private void initialise() {
060         for (int i = 0; i < neuronNumber; i++) {
061           neurons[i] = new Neuron(nextNeuronNumber);
062           neurons[i].initialise();
063         }
064
065         if (nextLayer != null) {
066           for (int j = 0; j < neuronNumber; j++) {
067             for (int k = 0; k < nextNeuronNumber; k++) {
068              neurons[j].connect(nextLayer.getNeurons()[k]);
069             }
070           }
071         }
072       }
```

```
073
074       void learn() {
075         step();
076         double[][] changeWeights =
                 new double[neurons.length][nextNeuronNumber];
077         for (int i = 0; i < nextNeuronNumber; i++) {
078           for (int j = 0; j < neurons.length; j++) {
079             changeWeights[j][i] =
                  learningRate * input[j] *
                  (target[i] - result[i]) *
                  derivedSigmoid(neurons[j].getInput());
080           }
081         }
082
083         for (int index = 0;
                  index < neurons.length; index++) {
084          neurons[index].adjustWeight(changeWeights[index]);
085         }
086
087         clear();
088       }
089
090       private void step() {
091         for (int i = 0; i < input.length; i++) {
092           neurons[i].setInput(input[i]);
093           neurons[i].step();
094           for (int j = 0; j < result.length; j++) {
095             result[j] += neurons[i].getResult()[j];
096           }
097         }
098         result = sigmoid(result);
099         nextLayer.inputFired();
100       }
101
102       double[] test(double[] input) {
103         for (int i = 0; i < input.length; i++) {
104           neurons[i].setInput(input[i]);
105           neurons[i].step();
106           for (int j = 0; j < result.length; j++) {
107             result[j] += neurons[i].getResult()[j];
108           }
109         }
```

```
110         result = sigmoid(result);
111         clear();
112         return result;
113       }
114
115       private void clear() {
116         for (Neuron neuron : neurons) {
117           neuron.clear();
118         }
119
120         for (Neuron neuron : nextLayer.getNeurons()) {
121           neuron.clear();
122         }
123       }
124
125       private double[] sigmoid(double[] result) {
126         for (int i = 0; i < result.length; i++) {
127           result[i] = sigmoid(result[i]);
128         }
129         return result;
130       }
131
132       private double sigmoid(double result) {
133         return 1 / (1 + Math.exp(-result));
134       }
135
136       private double derivedSigmoid(double x) {
137         return sigmoid(x) * (1 - sigmoid(x));
138       }
139
140       Neuron[] getNeurons() {
141         return neurons;
142       }
143     }
```

Neuron.java

```
01    import java.io.Serializable;
02    import java.util.Arrays;
03    import java.util.Random;
```

```
04
05    public class Neuron implements Serializable {
06      Random rand = new Random();
07      Neuron[] connectTo;
08      double[] weight;
09      double input;
10      double[] result;
11      int index = 0;
12
13      public Neuron() {
14        weight = new double[]{1};
15        result = new double[1];
16      }
17
18      public Neuron(int connectCount) {
19        connectTo = new Neuron[connectCount];
20        weight = new double[connectCount];
21        result = new double[connectCount];
22        initialise();
23      }
24
25      public void connect(Neuron neuron) {
26        connectTo[index] = neuron;
27        index++;
28      }
29
30      public void setInput(double input) {
31        this.input += input;
32      }
33
34      public double getInput() {
35        return input;
36      }
37
38      public void finalise() {
39        input = sigmoid(this.input);
40      }
41
42      public double sigmoid(double input) {
43        return 1 / (1 + Math.exp(-input));
44      }
45


88
```

```
46      public void initialise() {
47        for (int i = 0; i < weight.length; i++) {
48          weight[i] = rand.nextGaussian();
49        }
50      }
51
52      public void step() {
53        for (int i = 0; i < weight.length; i++) {
54          result[i] = input * weight[i];
55        }
56
57        if (connectTo != null) {
58          for (int index = 0;
                    index < connectTo.length; index++) {
59           connectTo[index].setInput(this.getResult(index));
60          }
61        }
62      }
63
64      public double[] getWeight() {
65        return weight;
66      }
67
68      public void adjustWeight(double[] adjustment) {
69        for (int i = 0; i < weight.length; i++) {
70          weight[i] += adjustment[i];
71        }
72      }
73
74      public double getResult(int index) {
75        return this.result[index];
76      }
77
78      public double[] getResult() {
79        return this.result;
80      }
81
82      public void clear() {
83        input = 0;
84        index = 0;
85      }
86
```

```
87      @Override
88     public String toString() {
89        return Arrays.toString(result);
90      }
91    }
```

initArray.java

```
01    import java.io.*;
02    import java.util.*;
03
04    public class initArray {
05      public static void main(String[] args)
                                  throws IOException {
06        long start = System.currentTimeMillis();
07        ArrayList<Integer> arr = new ArrayList<>();
08        double[][] inputValues = new double[60000][784];
09        double[][] targetValues = new double[60000][10];
10
11        for(int i = 0; i < 60000; i++) {
12          arr.add(i);
13        }
14        Collections.shuffle(arr);
15
16        for(int index = 0; index < 60000; index++) {
17          File file =
                new File("data/" +
                      String.format("%05d",
                            arr.get(index) + 1) + ".txt");
18
19          FileInputStream fis = new FileInputStream(file);
20          byte[] data = new byte[(int) file.length()];
21          fis.read(data);
22          fis.close();
23
24          Scanner in = new Scanner(file);
25          double[] x = new double[784];
26          for(int i = 0; i < 784; i++){
27            x[i] = in.nextDouble() / 255;
28          }
```

```
29
30          double[] target = new double[10];
31          target[(int) in.nextDouble()] = 1;
32
33          inputValues[index] = x;
34          targetValues[index] = target;
35        }
36
37        try (ObjectOutputStream oos =
                new ObjectOutputStream(
                new BufferedOutputStream(
                new FileOutputStream("input.ser")))) {
38          oos.writeObject(inputValues);
39        } catch (IOException ex) {
40          ex.printStackTrace();
41        }
42
43        try (ObjectOutputStream oos =
                new ObjectOutputStream(
                new BufferedOutputStream(
                new FileOutputStream("target.ser")))) {
44          oos.writeObject(targetValues);
45        } catch (IOException ex) {
46          ex.printStackTrace();
47        }
48        long stop = System.currentTimeMillis();
49        System.out.println(stop - start + " milliseconds.");
50      }
51    }
```

Explanation:

Line 1 & 2: Import necessary libraries.

Line 4: Start of `initArray` class.

Line 6: Keep track of the current time in milliseconds, then subtract it with the ending time to find the amount of milliseconds elapsed.

Line 7: Keep a list of all file name numbers and shuffle using the built-in class `Collections.shuffle()`. This is used to randomize the

order of training each sample so that the network doesn't have a bias to the ending samples because they appeared last.

Line 8 & 9: Initialize the two arrays that we will use to store the input values and target values for the sample. These arrays are later *serialized* using `ObjectOutputStream`.

Line 11: Add all file name numbers(0-60000) to the `ArrayList`.

Line 14: Shuffle the `ArrayList`.

Line 16: Open the file at the current index in the `ArrayList` and read all 784 inputs and the target. Note the values are converted to grayscale in the range 0-255, so we need to divide by 255 to get a value between 0 and 1.

Line 30: The target array doesn't just contain a single value, but an array of values with a single '1' for the correct prediction's index and '0' for all other indexes.

Line 37: Serialize the `input` array and store it as a file.

Line 43: Serialize the `target` array and store it as a file.

Line 48 & 49: Get the time elapsed and print to the console.

application.java

```
01    import java.io.*;
02    import java.util.*;
03
04    public class application {
05      public static void main(String[] args)
                throws IOException, ClassNotFoundException {
06        System.out.println("Gathering resources...");
07        long start = System.currentTimeMillis();
08        Layer nextLayer = new Layer(10);
09        Layer layer = new Layer(784, 10, nextLayer, 0.5);
10        Random rand = new Random();
```

```
11
12        ArrayList<String> randomIndex = new ArrayList<>();
13        for(int index = 0; index < 60000; index++) {
14          randomIndex.add(
                String.valueOf(rand.nextInt(60000) + 1));
15        }
16
17        ObjectInputStream ois =
                    new ObjectInputStream(
                    new BufferedInputStream(
                    new FileInputStream("input.ser")));
18        ObjectInputStream oiss =
                    new ObjectInputStream(
                    new BufferedInputStream(
                    new FileInputStream("target.ser")));
19        double[][] inputs = (double[][]) ois.readObject();
20        double[][] targets = (double[][]) oiss.readObject();
21
22        long stop = System.currentTimeMillis();
23        System.out.println(stop - start + " milliseconds.");
24
25        System.out.println("Training...");
26        start = System.currentTimeMillis();
27
28        for (int j = 0; j < 10; j++) {
29          System.out.println("Epoch: " + j);
30          for (int i = 0; i < inputs.length; i++) {
31            layer.setInput(
                inputs[Integer.parseInt(
                            randomIndex.get(i)) - 1]);
32            layer.setTarget(
                targets[Integer.parseInt(
                            randomIndex.get(i)) - 1]);
33            layer.learn();
34          }
35
36          Collections.shuffle(randomIndex);
37        }
38
39        try (ObjectOutputStream oos =
                    new ObjectOutputStream(
                    new BufferedOutputStream(
```

```
                    new FileOutputStream("app.ser")))) {
40          oos.writeObject(layer);
41        } catch (IOException ex) {
42          ex.printStackTrace();
43        }
44        stop = System.currentTimeMillis();
45
46        System.out.println(stop - start + " milliseconds.");
47        System.out.println("Done!");
48      }
49    }
```

Explanation:

Line 1 & 2: Import necessary libraries.

Line 7: Start timing the amount of time it takes to get resources.

Line 8 & 9: Create two layer objects for the learning stage.

Line 13: Create a list of random indexes to choose a set of inputs and targets from.

Line 17-20: Read the input and target arrays from their respective files which were generated after executing `initArray.java`.

Line 28: Start the learning stage by setting the input and target for the network and use the `learn()` function in the `Layer` class to adjust weights in the network.

Line 36: Shuffle the list of random indexes for the next epoch.

Line 39: Write our new neural network to a file named `app.ser`.

Line 46: Print the time (in milliseconds) that elapsed.

test3.java

```
01    import java.io.*;
02    import java.util.ArrayList;
03    import java.util.Scanner;
```

```
04
05    public class test3 {
06      public static double actual = 0;
07
08      public static void main(String[] args)
                throws IOException, ClassNotFoundException {
09        ArrayList<String> filenames = new ArrayList<>();
10        for(int x = 60001; x <= 70000; x++){
11          filenames.add(
                "data/" + String.format("%05d", x) + ".txt");
12        }
13
14        ObjectInputStream oiss =
                new ObjectInputStream(
                new BufferedInputStream(
                new FileInputStream("app.ser")));
15        Layer lays = (Layer) oiss.readObject();
16        int corrects = 0;
17        for (String z : filenames) {
18          double[] a = scan(z);
19          int x = getBestGuess(lays.test(a));
20          corrects += x == actual ? 1 : 0;
21        }
22        System.out.println("Testing: " + corrects +
                " / " + filenames.size() + " correct.");
23      }
24
25      public static double[] scan(String filename)
                            throws FileNotFoundException {
26        Scanner in = new Scanner(new File(filename));
27        double[] a = new double[784];
28        for(int i = 0; i < 784; i++){
29          a[i] = in.nextDouble() / 255;
30        }
31        actual = in.nextDouble();
32        return a;
33      }
34
35      public static int getBestGuess(double[] result) {
36        double k = Integer.MIN_VALUE;
37        double index = 0;
38        int current = 0;
```

```
39          for (double a : result) {
40            if (k < a) {
41              k = a;
42              index = current;
43            }
44            current++;
45          }
46
47          return (int) index;
48        }
49      }
```

Explanation:

Line 1-3: Import necessary libraries.

Line 10: Prepare a list of file paths from 60001.txt to 70000.txt, these will be used to test the network accuracy, not train the network.

Line 14: Read the layer object from the file we created in `application.java`.

Line 17-21: Read the file at the current index and step it through the network. See if the result is the same as the answer given in the file.

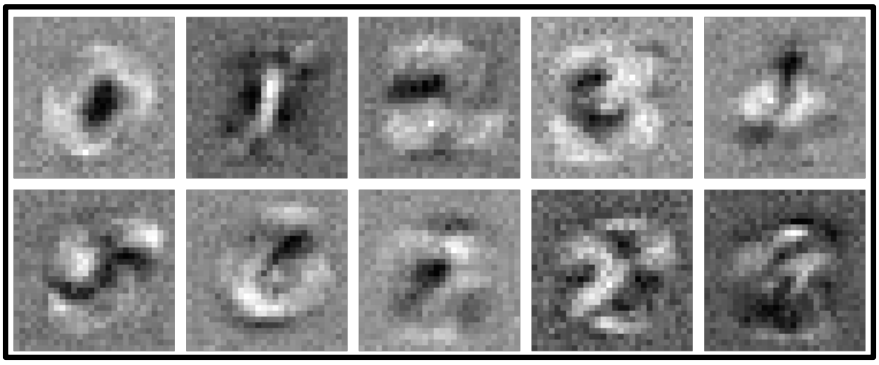
Line 22: Print out the accuracy of the network by printing the amount of answers the network was correct on out of 10000 (there were 10000 files).

Line 25-33: This is a method used to read all number values in the file path given, returning the result as an array. Note that I assigned the correct value to a global variable named *`actual`*.

Line 35-48: Get the most probable answer from the network output. Since we organized our predictions from 0-9, we can take the index at which the prediction is closest to 1 and return that as the network's prediction output.

Visualizing Weights

Now that we have been able to get through the solution and explanations of the code going with our solution, I can now answer the last question that I proposed before the Challenge Task. How can we visualize weights? Simply, we can just map out a 28x28 image with the weights of each neuron to each output neuron. Here is what people have been getting by doing this:



T1-8 - A visualization of weights from the Machine Learning for Artists website.

Here, whiter squares represent more important weights (the biggest weights), and the darker squares represent less important weights. If you look closely, you may notice that they look like the average of each digit from 0 to 9! This is not a mistake because you fitted your network given a set of data, so the network took an average or *best fit* for each weight in the network so that it could classify as many examples as possible. You may want to see how our network output looks compared to the example above! Here is a program that does just that!

visualize.java

```
01    import javax.imageio.ImageIO;
02    import java.awt.*;
03    import java.awt.image.BufferedImage;
```

```
04    import java.io.*;
05
06    public class visualize {
07     public static void main(String[] args)
              throws IOException, ClassNotFoundException {
08       int row = 28;
09       int column = 28;
10       int layNum = row * column;
11       int outNum = 10;
12
13       ObjectInputStream oiss =
                            new ObjectInputStream(
                            new BufferedInputStream(
                            new FileInputStream("app.ser")));
14       Layer layer = (Layer) oiss.readObject();
15
16       Neuron[] neurons = layer.getNeurons();
17       double[][] weights = new double[outNum][layNum];
18       double[][] transposeWeights =
                               new double[layNum][outNum];
19
20       for(int index = 0; index < neurons.length; index++) {
21         transposeWeights[index] =
                            neurons[index].getWeight();
22       }
23
24       for(int i = 0; i < transposeWeights.length; i++) {
25         for(int j = 0;
                   j < transposeWeights[i].length; j++) {
26           weights[j][i] = transposeWeights[i][j];
27         }
28       }
29
30       int index = 0;
31       for(double[] z : weights) {
32         File file = new File("images/" + index + ".png");
33         BufferedImage bufferedImage =
                        new BufferedImage(column, row,
                            BufferedImage.TYPE_INT_RGB);
34         Graphics2D g2d = bufferedImage.createGraphics();
35         double max = getMax(z);
36         double min = getMin(z);
```

```
37          double diff = max - min;
38          int ind = 0;
39          for (int x = 0; x < column; x++) {
40            for (int y = 0; y < row; y++) {
41              int col = (int) (255 - ((z[ind] - min)
                                    / diff * 255));
42              g2d.setColor(new Color(col, col, col));
43              g2d.fillRect(x, y, 1, 1);
44              ind++;
45            }
46          }
47          g2d.dispose();
48          ImageIO.write(bufferedImage, "png", file);
49          index++;
50        }
51      }
52
53      public static double getMax(double[] x) {
54        double y = Double.MIN_VALUE;
55        for(double z : x) {
56          if(z > y) {
57            y = z;
58          }
59        }
60        return y;
61      }
62
63      public static double getMin(double[] x) {
64        double y = Double.MAX_VALUE;
65        for(double z : x) {
66          if(z < y) {
67            y = z;
68          }
69        }
70        return y;
71      }
72    }
```

Explanation:

Line 1: Import necessary libraries.

Line 8-11: Define constants used later on in the program so that it is easily changeable.

Line 13: Read the `Layer` class so that weights can be obtained.

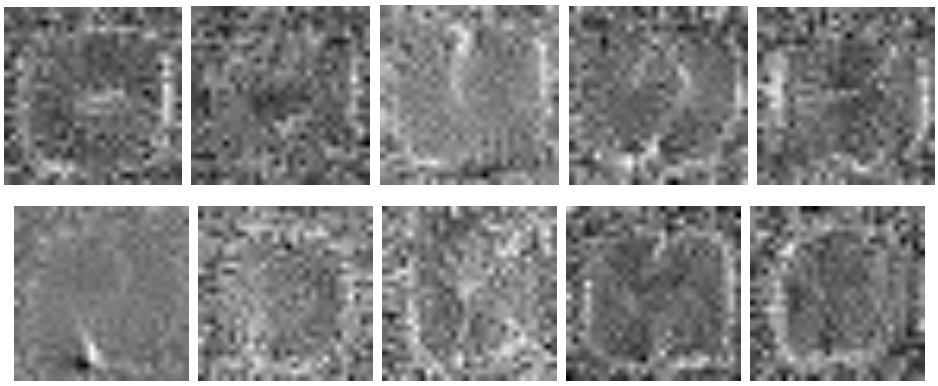
Line 20-28: First collect all weights as a multidimensional array, then transpose the array so that it is more usable and easier to loop through later on.

Line 31-50: Write each image by finding the range of the weights and judging the weight's importance relatively, and then filling a pixel with the same RGB values to get a grayscale image.

Line 53-61: Get the maximum weight value from an array.

Line 63-71: Get the minimum weight value from an array.

After running the above program, here's what I got on a model with roughly 89% accuracy:



T1-9 - A visualization of weights from our trained network

These don't quite look like the ones we saw previously, but you still can see some patterns in them. For now, head on to the next chapter!

Chapter Summary

1. To start training a neural network, we need a *learning rate*, *input neuron number*, *output neuron number*, *target array*, and *epoch number*.

2. Networks that don't use a *bias* or *threshold*, will instead need to use an *activation function*.

3. Making adaptable networks is an ideal practice.

4. Java includes methods to *serialize* and *deserialize* objects, which can be used to save models. If you are using another programming language, you may have to manually create a file including all weight (and bias if necessary) values.

5. Images can be resized before being given to a network in order to fit all input values into the number of input neurons in the network.

6. The MNIST dataset is a collection of 70,000 handwritten digits that are labeled manually to ensure accuracy. You can find it at http://yann.lecun.com/exdb/mnist/. Here is the one I used in the Challenge section: https://github.com/JC-ProgJava/Building-Neural-Networks-From-Scratch/releases/download/v1.0/MNIST.zip.

7. The Google Dataset Search tool is also another handy tool that can be used to search for datasets. Link: https://datasetsearch.research.google.com/.

8. 100% accuracy on datasets is impossible without *overfitting* a network. A model overfits when:
    - The training accuracy(accuracy of learning the training dataset) is much higher than the testing accuracy(accuracy of testing dataset, which the network has never seen before)
    - The output of the network is too certain(seen above: [1, 0]).

9. You can visualize weights by color-coding them from having the smallest impact to having the biggest.

Chapter 5 - Overfitting and Underfitting

> *"Those who don't believe in magic will never find it."*
> *— The Minpins by Roald Dahl*

In the previous chapter, I mentioned briefly the issue of *overfitting* and *underfitting*, which I want to go over quickly in this chapter.

Overfitting

First of all, what is *overfitting*? In its simplest terms, *overfitting* refers to a model that has been *overly fitted*. This sounds kind of pointless, somewhat like saying an *overreaction* is an *over*[exaggerated] *reaction*, so let me further explain what I mean. *Fitting* a model can be thought of as finding ideal values for different variables in the network, like the *weight* and *bias* through a learning process because variables are *fitted* with values that help them *generalize* a given set of training data as well as possible to achieve the highest accuracy on a testing dataset. This is somewhat analogous to fitting a new shoe: you try out a few different sizes and see which one you like best/find most comfortable. Therefore, *overly fitting* a model makes the model have *too ideal* values. What this means is that the model is starting to generate a "bias" on the training dataset, making it classify the training dataset with *unexpectedly high* accuracy. Referring back to the previous analogy, this is like making a shoe that will only

fit *one* person in the *whole world*. This is bad because it's not a good marketing decision… So if we go right back to my definition, an *overly fitted* model will be too specific when classifying the testing dataset, which causes the model to have an *unexpectedly low* model accuracy. To see why, I used an example in the previous chapter which I want to make as a generalized definition below:

==========

Overfitting

A model can overfit, which is simply where the model starts making too detailed or sophisticated assumptions on how to separate different objects.

For example, a model trying to classify apples and oranges is *overfitted* if it classifies oranges as "an object that is orange and perfectly round" just because the training dataset consists of hundreds of examples of perfectly-round oranges.

==========

We can tell when a model *overfits* if a few of the following are true:

- The training accuracy(accuracy of learning the training dataset) is much higher than the testing accuracy(accuracy on testing dataset, which the network has never seen before).
- The output of the network is too certain when classifying training examples (e.g., [1, 0]).

For example, let's say we are given a set of points on a graph, and we are to use a neural network to predict a value y for a new input value x. You may instantly think of the equation $f(x) = y$ when I assign the values to their

respective variables, which is good! Neural networks are somewhat like a function given an input, that gives you an output in this sense! But, they shouldn't exactly predict values like functions, if they did, the network must be overfitted.

This must sound vague, so let's go ahead with an example. Let's say we wanted to predict values for a given dataset-table below:

x Values	*y* Values
2	2
3	4
5	3
6	4.5
7	6
8	5
9	7.6
10	6

Figure 5-1: *A dataset consisting of various inputs and outputs.*

You may also like to see a visualization of it:

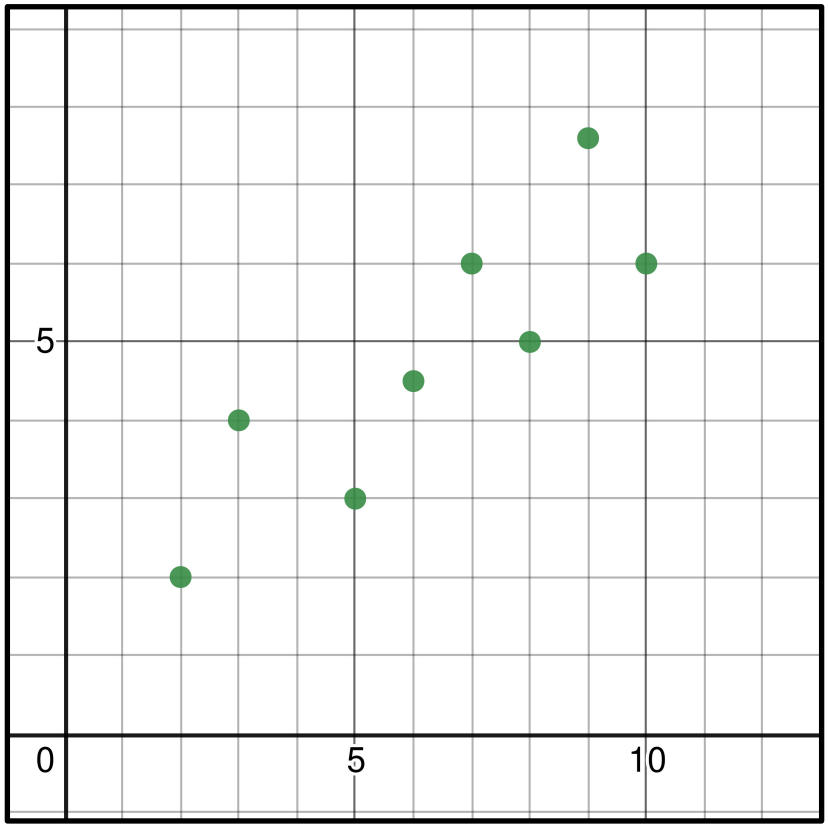


Figure 5-2: A visualization of the dataset given above.

Here, we see that the values we are trying to approximate show a weak positive correlation between the input and output. Therefore, we can conclude the function *can* be approximated. Without the use of neural networks, datasets like the above are usually approximated using a *line of best fit*. Wikipedia has a whole *list* of various algorithms to fit lines, so let's go over one of them and see if we compare and contrast the algorithm with a neural network's learning algorithm.

Least Squares Method

The least squares method is defined as below:

Given (x_1, y_1), (x_2, y_2), $\ldots$, (x_i, y_i), find:

1. Let $X = \sum_{n=1}^{i} x_n$, and $Y = \sum_{n=1}^{i} y_n$.
2. Let $\bar{x} = X \div i$, and $\bar{y} = Y \div i$.
3. Find $\sum_{n=1}^{i} (x_n - \bar{x})(y_n - \bar{y})$, and divide it by $\sum_{n=1}^{i} (x_n - \bar{x})^2$.
4. Find y-intercept by evaluating $\bar{y} - k\bar{x}$, where k is the value computed in step 3.
5. The line of best fit should be: $y = kx + (\bar{y} - k\bar{x})$, or $y = kx + c$, where $c = \bar{y} - k\bar{x}$.

Given this five-step algorithm to computing a line of best fit, we can try it on our dataset above and see what it outputs!

1. $X = 2 + 3 + 5 + 6 + 7 + 8 + 9 + 10$,
   $Y = 2 + 4 + 3 + 4.5 + 6 + 5 + 7.6 + 6$
   $X = 50$
   $Y = 38.1$
2. $\bar{x} = X \div 8$, $\bar{y} = Y \div 8$
   $\bar{x} = 50 \div 8$, $\bar{y} = 38.1 \div 8$
   $\bar{x} = 6.25$
   $\bar{y} = 4.7625$
3. $\sum_{n=1}^{i} (x_n - \bar{x})(y_n - \bar{y}) = (2 - 6.25)(2 - 4.7625) + \ldots = 30.275$
   $\sum_{n=1}^{i} (x_n - \bar{x})^2 = (2 - 6.25)^2 + \ldots = 55.5$

$k = 30.275 \div 55.5 \approx 0.545495495$ (this is the slope value)

4. $\bar{y} - k\bar{x} = 4.7625 - 0.545495495 \times 6.25 \approx 1.353$
5. $y \approx 0.545x + 1.353$

Graphing this equation, we get:

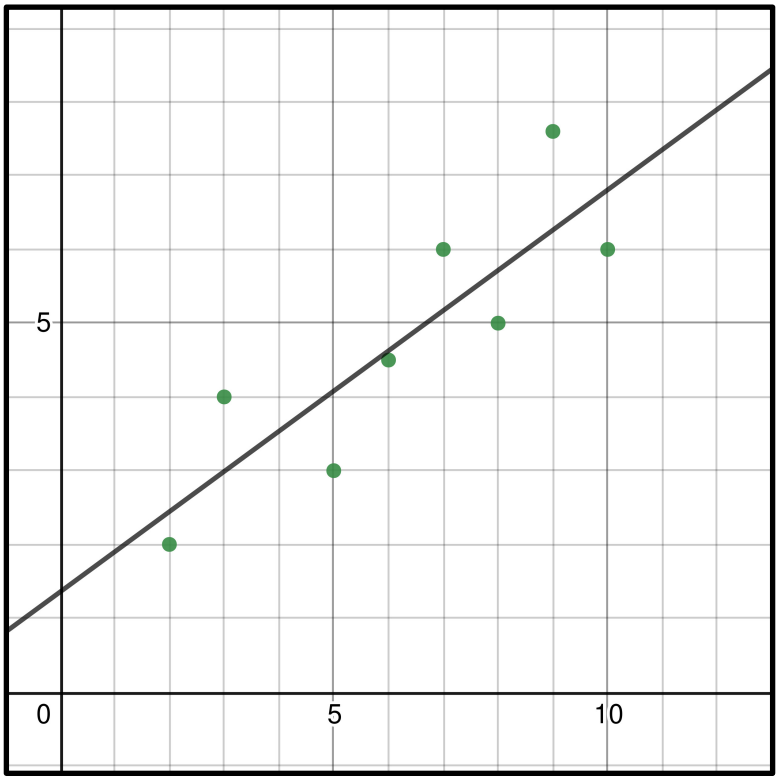


Figure 5-3: *Approximation of example dataset using the least squares method.*

You might feel that this algorithm is quite complex, but after we split it into 5 steps, you'll find it a lot easier to understand! Some readers thought that I *manually* calculated these values with an old-fashioned calculator, but do realize that this can be automated using programming languages! Below, I provide an example to do this in Python and Java, along with some explanations.

lsm.py

```
01    x = [2.0, 3.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]
02    y = [2.0, 4.0, 3.0, 4.5, 6.0, 5.0, 7.6, 6.0]
03
04    xAverage = 0
05    yAverage = 0
06
07    sum1 = 0
08    sum2 = 0
09
10    for i in x:
11        xAverage += i
12    for i in y:
13        yAverage += i
14
15    xAverage /= len(x)
16    yAverage /= len(y)
17
18    for i in range(len(x)):
19        sum1 += (x[i] - xAverage) * (y[i] - yAverage)
20        sum2 += (x[i] - xAverage) * (x[i] - xAverage)
21
22    slope = sum1 / sum2
23    inter = yAverage - slope * xAverage
24
25    print(sum1)
26    print(sum2)
27    print(slope)
28    print(inter)
29    print(xAverage)
30    print(yAverage)
```

Output:

```
30.275
55.5
0.5454954954954955
1.3531531531531535
6.25
4.7625

Process finished with exit code 0
```

Explanation:

Line 1-2: Note that I put '.0' behind every whole number in the arrays to make sure the Python interpreter understands that I want floating-point precision (the `x` array are all whole numbers).

Line 4-5: Initialize the variables that store the average of all input and output values as two different arrays. These values are to be used later on in the program.

Line 7-8: Store values for the summation equations in Step 3 of the algorithm.

Line 10-13: Accomplish Step 1 in the algorithm: add all input and output values into two separate arrays.

Line 15-16: Do Step 2 of the algorithm: find the average of the input and output values by dividing the summation of all values in the two arrays by the number of items in the array.

Line 18-22: Do Step 3 of the algorithm and find the slope of the *line of best fit* by dividing the values from the two summations.

Line 23: Find the y-intercept by enforcing Step 4 of the algorithm.

Line 25-30: Print out the values that we are interested in.

The Java example follows the same thought and step process, so I won't go over another almost-duplicate explanation this time.

lsm.java

```
01    public class lsm {
02      public static void main(String[] args) {
03        double[] x = {
04                2.0, 3.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0
05       };
```

```
06
07          double[] y = {
08                  2.0, 4.0, 3.0, 4.5, 6.0, 5.0, 7.6, 6.0
09          };
10
11          double xAverage = 0;
12          double yAverage = 0;
13          double sum1 = 0;
14          double sum2 = 0;
15
16          for (double val : x) {
17            xAverage += val;
18          }
19
20          for (double val : y) {
21            yAverage += val;
22          }
23
24          xAverage /= x.length;
25          yAverage /= y.length;
26
27          for (int i = 0; i < x.length; i++) {
28            sum1 += (x[i] - xAverage) * (y[i] - yAverage);
29            sum2 += (x[i] - xAverage) * (x[i] - xAverage);
30          }
31
32          double slope = sum1 / sum2;
33          double inter = yAverage - slope * xAverage;
34
35          System.out.println(sum1);
36          System.out.println(sum2);
37          System.out.println(slope);
38          System.out.println(inter);
39          System.out.println(xAverage);
40          System.out.println(yAverage);
41        }
42    }
```

So now that we have found another way to approximate output pairs in this example, how is it related to a neural network? Firstly, a neural network with a

single layer has the ability to approximate or classify in linear form, which means that training a network helps the network create a *line of best fit*. Therefore, you can analogically imply that *multiple* layers of a network would approximate functions in higher *dimensions*, or with a higher degree/index. This means that a multi-layer neural network is able to create more sophisticated approximations, and are therefore almost always better than single-layer networks. We *want* our network to create a *line of best fit*, because this is the most ideal way to approximate a set of data without having *bias*. Sometimes, you may hear news that tells you of biases in artificial intelligence and network models. This is caused by a *lack of general data*, which is where there isn't enough data across all categories the network was intended to classify. For example, this is like training a network with images of cats and expecting it to recognize a dog with a high degree of accuracy. You've *never shown* the network a picture of a dog: it doesn't *know* what a dog looks like, so it can't classify them, which makes networks produce a "guessed" result, or provides one that is uncertain(perhaps [0.57, 0.43] instead of [0.97, 0.03]). Other times, "biased" networks are created when data is improperly shuffled or the error rate of the network just peaked on the last epoch of training. When data is improperly shuffled, the network isn't given an adequate amount of examples of some of the things it is meant to classify or approximate. For example, this is like giving a short section of a function to the network that doesn't entirely *define* the function. Or, giving more examples of cats than dogs in the training dataset, which makes it harder for the network to classify dogs. To prevent or avoid these problems, people give a more general training dataset: instead of presenting a small part of a function to the network, cover *most* of the function,

and leave gaps in between data points; instead of giving 6000 images of cats and 4000 images of dogs in the training set, give 5000 images of each and shuffle their order. If we wanted our network to make a *line of best fit*, then overfitting would make the *line of best fit* "*fit too much*" on the training data.

For example, this is what would happen when a multi-layer network *overfits* when training to approximate the below dataset:

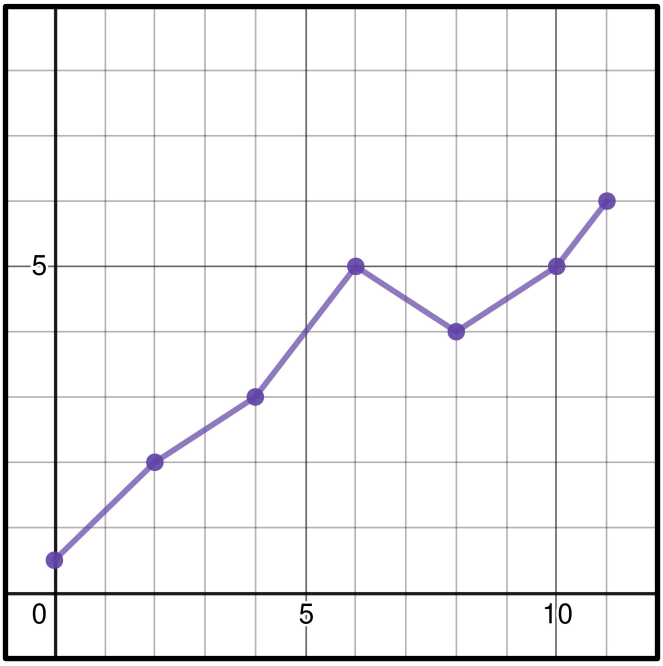


Figure 5-4: *Graphical representation of overfitted approximations from networks.*

First of all, why is this considered as overfitting? All data points are connected to each other, which means that the network is starting to "assign" values for each data point. This means that if you were to ask for an output value when $x = 12$, you'll almost certainly obtain an inaccurate result. It is because of this, that *overfitted* neural networks have a low testing and high training accuracy. Instead, we want our network to approximate somewhat like this:

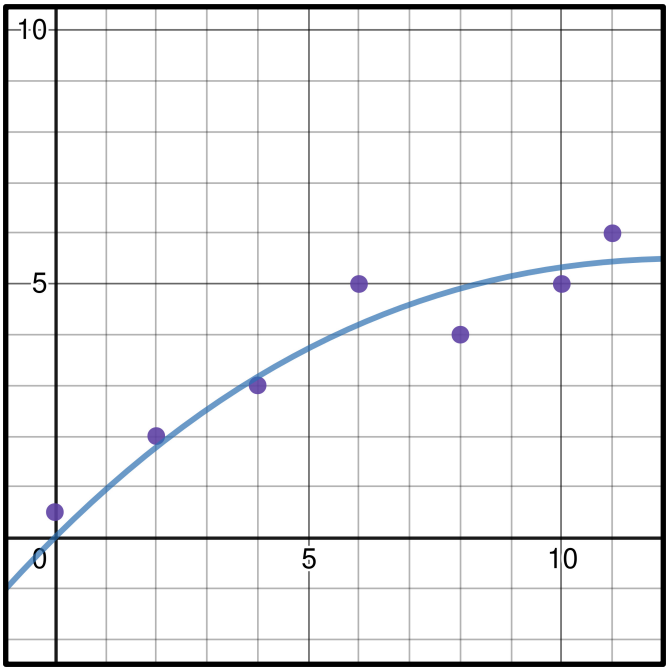


Figure 5-5: A better approximation of a function by a neural network.

If you wonder how I created these graphs, visit the website Desmos. There, you can actually play around with parameters and see how they affect the graphical representation.

Above, you can see that the line doesn't perfectly match the data points on the graph, but instead loop through them. This is the result that we want to achieve with our neural network: we don't want our network to be too specific or too abstract. For a neural network achieving the above result, the testing accuracy should be relatively higher because the network has *suggestions* on what the function will look like to the left and right of the range of the training dataset.

Now, you might as well wonder: why do networks sometimes *overfit*? What is the cause of *overfitting*? The answer is actually quite simple: in Chapter 4 (Neural Networks and Learning), we talked about the various parameters like

the *learning rate* and in the first Task, and introduced *epochs*. Networks are usually *overfitted* if they have been training over an *excessive* amount of *epochs* under a *very small* learning rate. Why is this? In Chapter 4, I mentioned that finding an ideal learning rate should get you more efficiently to the *minimal loss* or *error*. If we plot the error of the network over the training and testing dataset, we should find the training accuracy of the network is higher than the testing accuracy. Overfitting, on the other hand, would be like achieving the *global minimum loss* on the training dataset. This is not good, because the network starts being *biased* towards the examples in the training dataset, somewhat like *familiarizing* the network with training examples and telling it that all examples are the same as the ones given. Instead, we want to achieve an *error rate* close to but not *at* the *global minimum loss*. But quite often, that *global minimum* for the *loss* or *error* of a network cannot be achieved, which is why not every network *overfits*. Next, an *excessive amount* of *epochs* affects the accuracy a network *obtains*. For example, in Task 1, if you tried various other epoch numbers for the challenge, you'll realize that you can actually get a higher accuracy by increasing the number of epochs and also decreasing the learning rate, until you get to the point where the difference is noticeable. This increase in accuracy is because the network starts taking more *careful small* steps, towards a *minimum loss*, so it is less likely to accidentally reach a *higher loss*. You'll notice that the smaller your *learning rate* is, and the bigger your *epoch number* is, the closer you get to that global minimum for your training dataset, which *increases* the likelihood of overfitting.

Underfitting

So what is *underfitting*? It is somewhat the *opposite* of *overfitting*! Here, I'll give a definition like I did for *overfitting*.

==========

Underfitting

A model can underfit, which is simply where the model makes too general or abstract assumptions on how to separate different categories or approximate functions.

For example, a model trying to classify apples and oranges is *underfitted* if it gives incorrect general assumptions, like saying oranges are "usually green".

==========

Well… I'm sure that example confused you quite a lot! So, let's think about it: first of all, why would a model *make too general or abstract assumptions to classify or regress*? Second, what does this imply? To answer these questions, let's turn to our definition of overfitting and define underfitting alongside it. Here, I provide a comparison between the two, and further define underfitting:

Overfitting	Underfitting
Model makes too specific assumptions to regress or classify.	Model makes too abstract and often incorrect assumptions to regress or classify.
Training accuracy is much higher than testing accuracy.	Neither training accuracy or testing accuracy is high.

Because of a very low learning rate and a large number of epochs.	Because of an inadequate amount of epochs and perhaps a high learning rate value.
Occurs more than underfitting.	Occurs less than overfitting.

So what does this imply? First of all, this implies that underfitting is somewhat like a model learning too less from the training dataset and overfitting is somewhat like a model learning "too much" from the training dataset. Since underfitting is caused by a high learning rate and/or inadequate amount of epochs, then this implies that the model's *weights* and *biases* are closer to their initialized and random value than to their ideal values. Usually, models can *overfit*, which often results in slightly lower accuracy. But sometimes, models can *underfit*, which can result in even lower accuracy levels. But, why does *underfitting* not occur as much as *overfitting*? Models *overfit* more than *underfit* because operators who train models understand that the learning rate of a model has to be smaller (usually) than 0.5 or 1, and epoch numbers need to be greater than 5 at least. This means, there are certain benchmarks out there that work well to prevent underfitting in all types of network models. But, there aren't a lot of guidelines on how to choose a learning rate that is not too small and an epoch number that is not too big. There aren't a lot of these guidelines (for avoiding overfitting) because results aren't consistent between different model use cases, so people end up not venturing into the unknown.

This tradeoff between accuracy and generalization ability has a sophisticated name stuck to it: *bias-variance tradeoff*, and is one of the many problems

practitioners of data science and artificial intelligence are facing in the real world.

That's just about all I wanted to say! Take a look at the Chapter Summary and head on to the next chapter!

Chapter Summary

1. A model can overfit, which is simply where the model starts making too detailed or sophisticated assumptions on how to regress or classify.

2. We can tell when a model overfits if a few of the following are true:
   - The training accuracy(accuracy of learning the training dataset) is much higher than the testing accuracy(accuracy on testing dataset, which the network has never seen before).
   - The output of the network is too certain when classifying training examples (e.g., [1, 0]).

3. The Least Squares Method is one of many linear regression algorithms used to determine a line of best fit. Defined below:

   Given (x_1, y_1), (x_2, y_2), $\ldots$, (x_i, y_i), find:

   1. Let $X = \sum_{n=1}^{i} x_n$, and $Y = \sum_{n=1}^{i} y_n$.
   2. Let $\bar{x} = X \div i$, and $\bar{y} = Y \div i$.
   3. Find $\sum_{n=1}^{i} (x_n - \bar{x})(y_n - \bar{y})$, and divide it by $\sum_{n=1}^{i} (x_n - \bar{x})^2$.

    4. Find y-intercept by evaluating $\bar{y} - k\bar{x}$, where k is the value computed in step 3.
    5. The line of best fit should be: $y = kx + (\bar{y} - k\bar{x})$, or $y = kx + c$, where $c = \bar{y} - k\bar{x}$.

4. Single-layer neural networks have the ability to perform *linear regression*.

5. Multiple-layered networks can approximate functions in higher *dimensions*, or degree/index, therefore creating more sophisticated and accurate approximations.

6. *Bias* in networks is caused by a *lack of data* generalizing the use case for the model, an improper shuffling of data, or just by chance when the network stopped training (error rate peaks on last epoch).

7. Overfitting is caused by an *excessive* amount of training *epochs* and a (too) *small learning rate*.

8. A model can *underfit*, which is simply where the model makes too general or abstract assumptions on how to regress or classify a given dataset.

9. *Underfitting* is caused by an inadequate amount of epochs and a high learning rate.

10. The tradeoff between accuracy and generalization ability is known as *bias-variance tradeoff*, and is a problem that the artificial intelligence community is trying to solve.

Chapter 6 - Validation Datasets

> *"The moment you doubt whether you can fly,*
> *you cease forever to be able to do it."*
> *— Peter Pan by J.M. Barrie*

In this chapter, I aim to answer several questions about testing, training, and validation datasets:

1. What are testing and training datasets for?
2. What are validation datasets?
3. Why are validation datasets needed?
4. Usually, how much of the entire dataset is dedicated to a validation sub-dataset?
5. What is the difference between the validation and testing dataset?
6. How are datasets separated into three sub-datasets(training, testing, validation)?
7. Are validation datasets necessary?

First of all, what *are* datasets? According to the Cambridge Advanced Learner's Dictionary (4th Edition), a dataset is "*a collection of separate sets of information that is treated as a single unit by a computer*". This somewhat defines the term "dataset" with general terms and can be used to describe *any* dataset. So, why do we *need* a dataset? If you take a look at Google Books' Ngram Viewer of the word, you find that it only started being used widely in the past few decades.

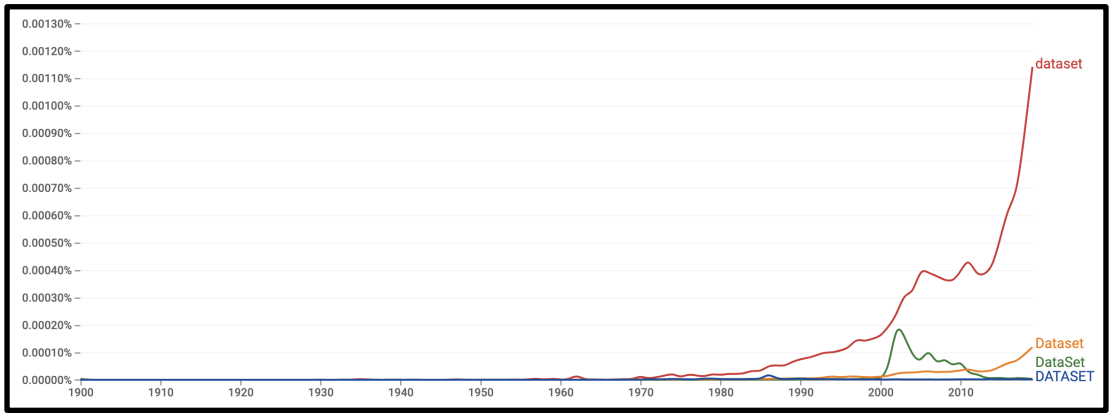


Figure 6-1: A graph showing the usage of the word 'dataset' over time (1900-2019).

This shows that *datasets* have only become popular recently, probably due to renewed interest in empirical scientific research (research from observation and measurements) and the development of new fields like data and computer science. With this, we can conclude that datasets are created for scientific research, and often consist of similar data grouped in such a way to show correlations or patterns.

Training neural networks require data. This means we need to find a collection of examples to *give* to our network in order to achieve an approximation *of* the examples that we like. In order to see how well our network performs, we need to provide *another* set of data that *wasn't* used in the training process and run our network through the inputs and find its *accuracy* or *error rate/loss*. The dataset used for training is called the *training dataset*, and the one used to evaluate a network's performance is called the *testing dataset*. Note these definitions and purposes of the dataset types above are important when in consideration of what *validation datasets* are.

An important question comes up after a network is trained and a testing dataset is used to evaluate its performance is: *what if I'm not satisfied with my network prediction accuracy on the testing dataset?* Your first reaction should be to adjust various *hyperparameters* of the network like the *learning rate*, *epoch number,* or *network structure* (consisting of *layer number*, *neuron number*, *activation function*, and various other decisions like whether *biases* should be used), which should undoubtedly either *improve* or *worsen* your network accuracy. If this doesn't work, then you should realize that neural networks probably aren't the best solution for what you have in mind.

In the *Overfitting and Underfitting* chapter, I stated that *biased networks* are caused by a lack of data generalizing the use case, an improper shuffling of data, or just chance. Apart from this, networks can actually be biased because of the *operator's choices*. This means, choices like hyperparameter tuning, dataset source, etc. directly affect whether a network becomes *biased*, *overfitted*, or *underfitted*. Why? In the previous paragraph, the decision to adjust *hyperparameters* in order to improve network accuracy leads to an unavoidable problem: the network is being fitted *for* the testing dataset, meaning it might have a *bias* on the testing dataset. This is because the *operator* of the network *evaluates* the network's performance on the testing dataset, which means that the *aim* of *hyperparameter adjustment* isn't just to improve training dataset accuracy, but ultimately the testing dataset accuracy. This means that the operator is *hoping* for a *higher testing accuracy*, which is why the operator is making adjustments to the network parameters that may *bias* towards the testing dataset. This undoubtedly isn't intended, and will most definitely affect the real-world performance of the neural network, which is why network

operators invented something to fix this issue: *validation datasets*. Validation datasets are *another* separate dataset that is used *also* to evaluate the network's performance. But instead of evaluating the performance of the network after every *iteration* of hyperparameter adjustment, evaluate it at the *last minute*, meaning you should use the validation dataset to evaluate the performance of the network *just before* the network is exported or ported off for use in applications. If the *testing accuracy* is a lot higher than the *validation accuracy* (accuracy of the network on the *validation dataset*), the network is definitely becoming *biased* to the testing dataset. This way of splitting an entire dataset also has other benefits, e.g., you can better detect overfitting because you have a second "layer" of testing before the network is deployed for use in the real world. Below, I provide a comparison between the three types of datasets or sub-datasets:

<table>
<tr><th>Training Sub-Dataset</th><th>Testing Sub-Dataset</th><th>Validation Sub-Dataset</th></tr>
<tr><td>Usually a substantial portion of the entire dataset.</td><td colspan="2">Both training and validation sub-datasets are usually around the same size and each covers roughly 10-30% of the entire dataset.</td></tr>
<tr><td>Used to train the neural network by the adjustment of weights and biases through a learning process.</td><td>Used to evaluate a network's performance after every iteration of hyperparameter adjustment.</td><td>Used to evaluate the network's performance just before deployment.</td></tr>
</table>

Necessary/compulsory in order to train a neural network.	Optional, but usually preferred because it can prevent overfitting and underfitting as well as make sure the network isn't biased and performs as expected in the real world.
Randomized.	Not randomized (there is no need).

Figure 6-2: *A comparison of testing, training, and validation datasets.*

Note that I wrote "[validation datasets are] *optional but usually preferred because it can prevent overfitting and underfitting as well as make sure the network isn't biased and performs as expected in the real world*". This means, that it is up to the operator to choose whether to use validation datasets, so when should you not use a validation dataset? You shouldn't use a validation dataset when your dataset is small. This is because it limits the number of training and/or testing samples that can be used when training your model. Since networks learn better when given more examples that *generalize* the use case, this inevitably will reduce the network's training accuracy, so operators of networks tend to opt-out of creating a validation dataset when data is limited.

Separation of Testing, Training, and Validation Datasets

So how should a single dataset be separated into these three separate sub-datasets? Is there any optimal way of splitting the data? Usually, the best way to

split a dataset into the three datasets is to spread examples evenly into each dataset with an equal (or relatively close) amount of each type of example in each dataset. For example, putting 1000 images of different 0-9 digits into each of the three datasets (training, testing, and validation). If your use case isn't classification, but regression, you should find a wide range of inputs that cover most of the regressing problem you want to solve, and pick various points in between these training samples to be your testing and validation samples. This ensures your network has a somewhat whole viewpoint/outlook on the function being approximated so the network will be able to predict outputs for new input values.

That's just about it, head on to the next chapter!

Chapter Summary

1. A dataset is "*a collection of separate sets of information that is treated as a single unit by a computer*", and are created for scientific research, and often consist of similar data grouped in such a way to show correlations or patterns.

2. Training neural networks require data. This means we need to find a collection of examples to *give* to our network in order to achieve an approximation *of* the examples that we like. In order to see how well our network performs, we need to provide *another* set of data that *wasn't* used in the training process and run our network through the inputs and find its *accuracy* or *error rate/loss*.

3. The dataset used for training is called the *training dataset*, and the one used to evaluate a network's performance is called the *testing dataset*.

4. Networks can be biased because of an *operator's choices*. This means, choices like hyperparameter tuning, dataset source, etc. directly affect whether a network becomes *biased*, *overfitted*, or *underfitted*.

   Reasoning: The operator of the network is *hoping* for a *higher testing accuracy*, which is why the operator makes adjustments to the network; thus, parameters may make the network be *biased* towards the testing dataset.

5. Validation datasets are *another* separate dataset that is *also* used to evaluate the network's performance. But instead of evaluating the performance of the network after every *iteration* of hyperparameter adjustment, evaluate it at the *last minute*, meaning the validation dataset is used to evaluate the performance of the network *just before* the network is exported or ported off for use in applications. If the *testing accuracy* is a lot higher than the *validation accuracy* (accuracy of the network on the *validation dataset*), the network is definitely becoming *biased* to the testing dataset.

6. A comparison between testing, training, and validation datasets:

Training Sub-Dataset	Testing Sub-Dataset	Validation Sub-Dataset

<table>
<tr><td>Usually a substantial portion of the entire dataset.</td><td colspan="2">Both training and validation sub-datasets are usually around the same size and each covers roughly 10-30% of the entire dataset.</td></tr>
<tr><td>Used to train the neural network by the adjustment of weights and biases through a learning process.</td><td>Used to evaluate a network's performance after every iteration of hyperparameter adjustment.</td><td>Used to evaluate the network's performance just before deployment.</td></tr>
<tr><td colspan="2">Necessary/compulsory in order to train a neural network.</td><td>Optional, but usually preferred because it can prevent overfitting and underfitting as well as make sure the network isn't biased and performs as expected in the real world.</td></tr>
<tr><td>Randomized.</td><td colspan="2">Not randomized (there is no need).</td></tr>
</table>

7. You shouldn't use a validation dataset when your dataset is small. This is because it limits the number of training and/or testing samples that can be used when training your model. Since networks learn better when given more examples that *generalize* the use case, this inevitably will reduce the network's training accuracy, which is why operators of

networks tend to opt-out of creating a validation dataset when data is limited.

8. The best way to split a dataset into the three datasets is to spread examples evenly into each dataset with an equal (or relatively close) amount of each type of example in each dataset. For example, putting 1000 images of different 0-9 digits into each of the three datasets (training, testing, and validation). If your use case isn't classification, but regression, you should find a wide range of inputs that cover most of the regressing problem you want to solve, and pick various points in between these training samples to be your testing and validation samples. This ensures your network has a somewhat whole viewpoint/outlook on the function being approximated so the network will be able to predict outputs for new input values.

Chapter 7 - Data Preprocessing & Augmentation

"Time you enjoy wasting is not wasted time."

— Phrynette Married by Marthe Troly-Curtin

Image Preprocessing

This section will be dedicated to image preprocessing mainly, while another section will talk about data (numerical and text-based) preprocessing; and image and data augmentation. I mainly focus on image-based preprocessing and augmentation because preprocessing and augmentation is most commonly applied to images. Afterwards, I will outline various techniques used and provide code examples. Let's start!

First of all, *image* ***preprocessing*** defines the process to which an image is processed and modified before use in the future. On the other hand, *image* ***processing*** instead just refers to the broader subject of processing/modifying an image. Image processing is separated into two categories: digital image processing and analogue image processing. Analog image processing refers to processing applied to analog signals (i.e., light or sound. Analog signals are continuous and appear in the real world). This processing is usually done through a camera lens, which focuses light onto a sensor that converts the light (a continuous analog signal) into an image (a non-continuous [images are encoded in a finite number of pixels, which approximates the scene which was

taken] digital signal). Notice that at this stage, various processing techniques are also available (and thus called *analog image processing*): you can adjust multiple different parts of the camera like the aperture size or focal length, which results in changes to the image (i.e., perspective, distortion). We won't focus on analog image processing techniques in this book, but rather on digital image processing techniques. Digital processing techniques are performed through the manipulation of images, which is done using various algorithms. The aim of this process is to enhance images (or particular features in an image) through the removal/lightening of image noise and distortions. It is performed before feeding a model the input data, and can be performed on both training and testing data. It is useful because it can help artificial intelligence models learn better (reach a higher accuracy in a smaller number of training epochs) as data is better represented and *noise* or irrelevant parts in data are removed. Preprocessing images can therefore also reduce the likelihood of overfitting and consequences from underfitting (i.e., accuracy may be affected less by underfitting). Models can also be trained more quickly with the help of image preprocessing techniques. For example, by resizing all data to a smaller length, computational time would decrease as a whole, and the network itself will have a smaller input size, which makes patterns in data more obvious. Digital image processing is also useful for training with neural networks as neural networks require input data to be of the same size. E.g., images cannot have different sizes, so a preprocessing technique (resizing) would have be applied to make the input images of identical sizes. Some examples of digital image processing techniques include: resizing/dilation, orientation, coloring

(i.e., through contrast, filters, greyscale adjustments), and deskewing (opposite of skewing, which is a technique used in data augmentation).

Image Data Augmentation

Image data augmentation is a process undertaken to increase the size of image datasets through the use of various augmentation techniques. These techniques often manipulate various properties present in an image to create similar images which can be used to train networks. Augmenting datasets is useful as models learn better with more variance in data (example), which can help prevent model underfitting and overfitting. Data augmentation is performed on data before training and is usually only applied to the training dataset.

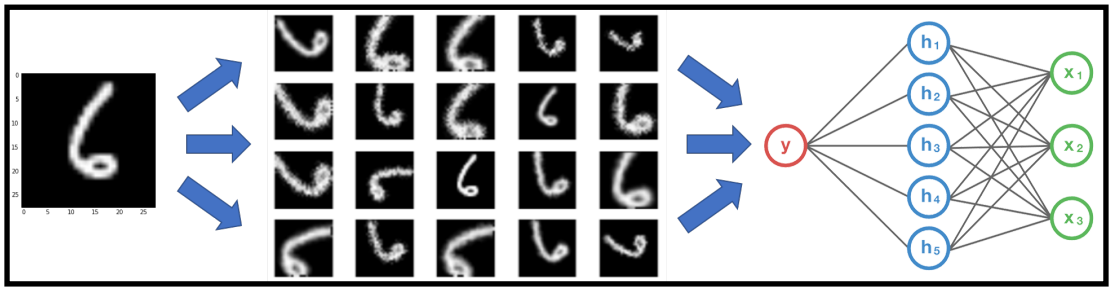


Figure 7-1: An example of how image data can be augmented.

Some common augmentation techniques include rotating, skewing (slanting), translating and resizing images. Sometimes, datasets can also be augmented through the adjustment of the brightness (or even opacity/transparency) of an image.

If we can augment data, what is the point of caching or collecting large quantities of data? Why can't we just augment a small subset of data to create practical and usable datasets? The main reason why data augmentation cannot

completely substitute original (or 'real') data is because augmented data cannot possibly attempt to account for every possible scenario (or set of data) that a model may encounter.

I also want to emphasize the difference between preprocessing and augmentation. Augmentation is applied *before* the training stage with a purpose to increase dataset size. Preprocessing is applied *during* the training stage with a purpose of removing irrelevant data (or features) that may interfere with the network's ability to *generalize*.

There are many libraries out there that can assist in creating augmented datasets. For example, Augmentor, a library for data augmentation in Python. Some other popular ML frameworks like Tensorflow also support various augmentation techniques, and Java has the `Graphics2D` library to perform transformations to images. You can see some examples in later sections that demonstrate how these libraries can be utilized for data augmentation.

Furthermore, I have created a Kaggle dataset (that is also available on Github) of handwritten images (not in MNIST) that uses image data augmentation. The techniques I used and the complete source code will also be disclosed in the code examples section.

Image Preprocessing and Image Data Augmentation Techniques

Both image preprocessing and image data augmentation (for images) use similar techniques. Below is a list of techniques, when to use them, and precautions:

1. Resizing

   Usually this refers to the stretching or shrinking of an image.

   This technique seems surprisingly simple, but there still are different options within this single processing technique. You can resize image data directly or add blank pixels around an image to resize the overall image. Also, images can be 'cropped', which is also a form of resizing. Shrinking large images is often better than stretching small images when training networks on image data. Image resizing can occur during both preprocessing and data augmenting stages.

2. Rotation

   Rotation refers to a transformation where an object is rotated in a certain direction (clockwise or anti-clockwise/counter-clockwise) by a certain number of degrees. In data augmentation, rotation is used to make slight changes to an image in order to expand a dataset. This requires that the image data is *allowed* to be rotated. For example, handwritten digits.

3. Greyscale conversion

   A greyscale conversion converts an image from a range of colors into a range of blacks and whites. This is usually done to reduce the *dimensionality* of data features (by reducing information about the image such as color). Greyscaling images is usually done as a data preprocessing technique as it helps improve model performance (through the removal of possible noise and irrelevant features). Note that it may sometimes not be a good idea to remove color from images as it may help models identify various features in a dataset. For example, in classification tasks like object classification or animal classification (e.g., cats and tigers look similar but can be

distinguished through the color of their fur [usually, although sometimes cat fur may be orange]).

4. Reflection

   Reflection is a transformation where an object is reflected across an axis such that its mirror image and the original image is perfectly equidistant to the axis of reflection. This technique is useful in data augmentation as it can help increase dataset size whilst still creating relevant new data examples for a machine learning model to train on.

5. Exposure

   This technique adjusts the brightness of images. It is mainly used in data augmentation to enlarge a dataset, although it can be used as a data preprocessing technique (e.g., identifying the brightness of an image and adjusting it so that model performance is improved).

6. Adding Noise

   This technique adds noise into images in the form of black or white pixels. Although this may seem to be an odd processing technique, it can help prevent the model identifies patterns in irrelevant features in images, and thus overfitting as a whole.

7. Skewing

   This is a technique used to 'center' images that are slanted. This process can be done on multiple occasions (e.g., turning a parallelogram into a rectangle involves skewing) and is useful during the image preprocessing stage as it can help make images more similar to ones a model is trained to recognize (e.g., slanted handwritten digits can be made vertically straight and

centered, which may improve the model's confidence on its prediction/classification result). Skewing can also be used to augment datasets as well.

Other Mediums of Data Augmentation

Datasets that don't contain images can also be augmented before use. For example, in numerical datasets, input-output pairs can be slightly adjusted to create different data points (i.e., adding or subtracting minuscule values from input values or output values). I won't cover these extensively as usually numerical datasets are rather abundant and contain extensive amounts of data, and don't need to be augmented.

Other Mediums of Data Preprocessing

Once again, data preprocessing may be needed for other forms of data, like numerical data. A common approach to data preprocessing is something called *feature removal*. Primarily, feature removal aims to remove features in datasets that may not be relevant to the use case. For example, given that the aim of a program is to predict the weather tomorrow given today's weather data, information like the name of the weather station that recorded the data would be unnecessary. It might even *hinder* the network's generalization abilities as the network might associate certain beliefs with certain weather station names, which is clearly unrelated to how the weather might be.

In the case that the name of a weather station has a correlation to the weather, what would happen if a weather station that wasn't in the dataset was used in application?

Another common approach is called *outlier removal*, which basically removes various data points that don't have any outlying correlation with the use case itself, and may affect the accuracy of a network.

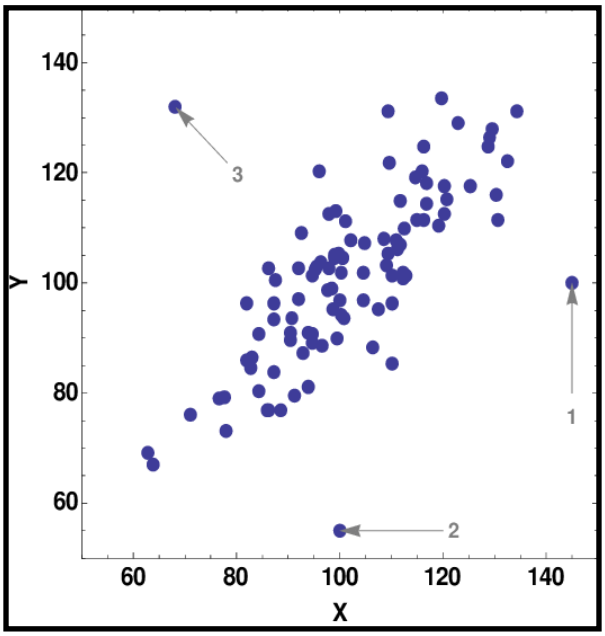


Figure 7-2: *Outliers in numerical data. Notice how points marked '1', '2', and '3' have no underlying correlation with the general group of points.*

Outliers are often recorded due to real-world measurement errors. For example, the weather station may have recorded temperatures 10°C higher than normal because of some reported fault in the measuring instrument.

You might think that there only exists two forms of data: numerical and image-based, but that isn't true! For example, audio is also a form of data which can be used by networks. A common example of where audio data is used is in voice and speech recognition. A form of preprocessing for audio would be to reduce sampling range (the frequency of samples in a unit length of time) to increase network training speed. We can also make a *spectrogram*, which

visualizes audio as images plotted based on frequencies recorded in the audio file.

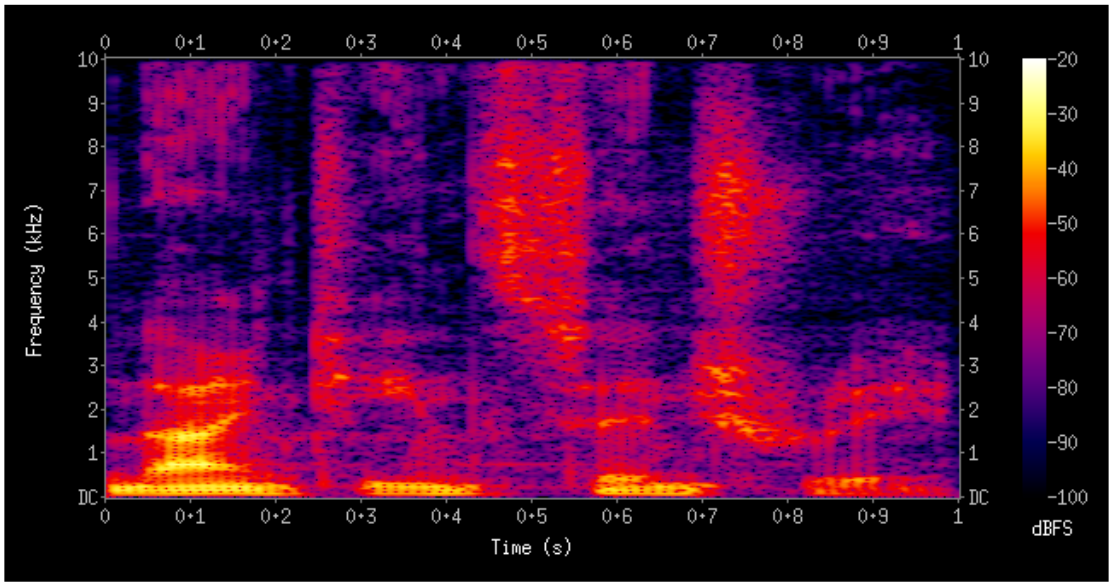


Figure 7-3: *A spectrogram.*

We can then use this image as an input to the network instead of all the values from the audio file.

Code Examples

Now, I'll show you some programming examples that perform image data augmentation and image preprocessing.

Task 1

> *Given a user inputed handwritten digit, output a series of images that are similar to it. Achieve this by applying various data augmentation techniques like skewing and adjusting exposure.*

I'll use Java again for these programming tasks. Do note that these techniques can also be applied in other programming languages.

```
01    import javax.imageio.ImageIO;
02    import java.awt.*;
03    import java.awt.image.BufferedImage;
04    import java.io.File;
05    import java.io.IOException;
06
07    public class DataAugment {
08      public static void main(String[] args)
                                throws IOException {
09        int counter = 1;
10        BufferedImage image = ImageIO.read(
                                 new File("src/digit.png"));
11        for (double i = -0.2; i <= 0.2; i += 0.05) {
12          for (double j = -0.2; j <= 0.2; j += 0.05) {
13            BufferedImage output = new BufferedImage(28, 28,
                                   BufferedImage.TYPE_INT_ARGB);
14            Graphics2D graphics2D = output.createGraphics();
15            graphics2D.shear(i, j);
16            graphics2D.drawImage(image, 0, 0, null);
17
18            File outFile = new File("output-images/output"
                                       + counter + ".png");
19            ImageIO.write(output, "png", outFile);
20            counter++;
21          }
22        }
23      }
24    }
```

Notice how short and simple this program is! This shows that automating data augmentation isn't that hard given the general availability and ubiquity of image processing libraries out there. Also, note that *"src/digit.png"* on Line 10 is in italic, meaning you should replace this absolute path with the absolute path of the actual input test image.

Code explanation:

Line 1-5: Import necessary libraries.

Line 8: Note that throwing exceptions aren't the best/standard, but this program is meant for demonstration purposes of data augmentation, not writing perfect code that handles every error.

Line 9: Store a counter that helps create files. Names: 'output1', 'output2', …

Line 10: Create an object that stores the input image.

Line 11-22: Perform shearing/skewing operations. Note that `i` and `j` adjust the amount of x-shearing and y-shearing. Shearing/skewing basically tilts an image (that is rectangular) by a certain angle, to create another image (which now is technically a parallelogram).

Line 13: Create another object to store the output image, which is later written into an output file.

Line 14: Create a graphics object. It is used to perform the shearing transformation.

Line 15: Set the transformation to shear by a certain amount. Note that this function call appears before we draw an image, because we are technically setting a configuration such that painting operations (like `drawImage()`) that follow have the transformation applied to it.

Line 16: Draw the original image, but perform the previously set shearing transformation on it.

Line 18: Create a file object that stores the output file's path. We use the counter here so that each file is named 'output1', 'output2', and so forth. Note that we store these files in another folder called 'output-images'. **The output folder must exist before running the program**.

Line 19: Write the image into the file in the PNG format.

Line 20: Increment the counter (which stores which file is currently being written into).

I've actually created a dataset of handwritten images that accomplishes this task as well. You can find the repository here. I decided to use Javascript so that users could make their own dataset online via a webpage, although the fundamental mechanics of the above program and the JS version are almost identical. In fact, I basically just translated the augmentation loop into Java and made it fit for this book.

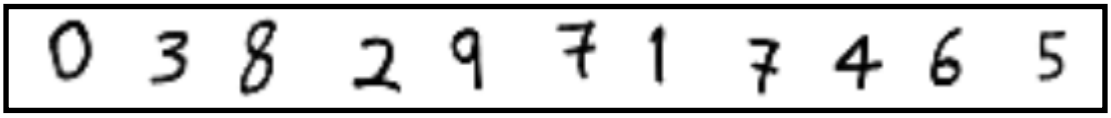


Figure 7-4: Some sample outputs of the JS version of the program above.

The only difference between the program above and my original dataset creator is that the original JS version took image data augmentation even further, being able to provide approximately 3,000 different images given a single input image through the use of skewing techniques.

Task 2

> *Adjust the brightness of images to augment a dataset. Note that this also applies for preprocessing images.*

Note that this task can be accomplished in many ways, so I'll just show one that I find most efficient and readable. I'll just give a short explanation on the program this time, as it uses quite a lot of native libraries for programming the Graphical User Interface.

```
01  import javax.imageio.ImageIO;
02  import javax.swing.*;
03  import java.awt.*;
04  import java.awt.image.BufferedImage;
05  import java.awt.image.RescaleOp;
06  import java.io.File;
07  import java.io.IOException;
08
09  public class Brighten {
10    public static void main(String[] args)
                                throws IOException {
11      JLabel[] images = new JLabel[8];
12      BufferedImage image = ImageIO.read(
                                new File("src/digit.png"));
13      images[0] = new JLabel();
14      images[0].setIcon(new ImageIcon(image));
15      for (int i = 1; i < 8; i++) {
16        RescaleOp op = new RescaleOp(1.0f + i * 0.1f,
                                       0.0f, null);
17        BufferedImage temp = op.filter(image, null);
18        images[i] = new JLabel();
19        images[i].setIcon(new ImageIcon(temp));
20      }
21      JFrame frame = new JFrame();
22      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23      JPanel panel = new JPanel(new GridLayout(4, 2));
24      for (JLabel img : images) {
25        panel.add(img);
26      }
27      frame.setSize(400, 400);
28      frame.add(panel);
29      frame.setVisible(true);
30    }
31  }
```

Instead of just modifying an image's brightness once and outputting its result to a file, this program modifies the image 7 times (not including the original image) with each image varying in brightness by 10%. The resulting 8 images

(including the original) are then added to a panel and displayed visually. Below is a comparison of an example input and output of the program:

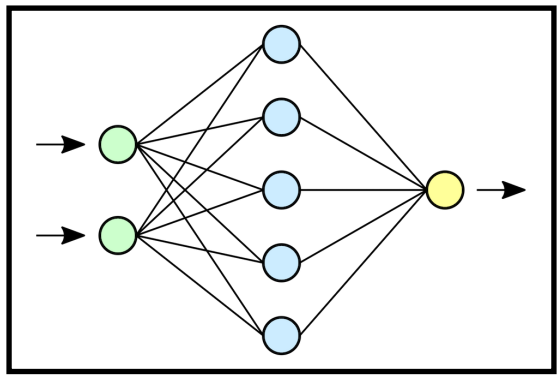


Figure 7-5: An example input to a program that modifies image brightness.

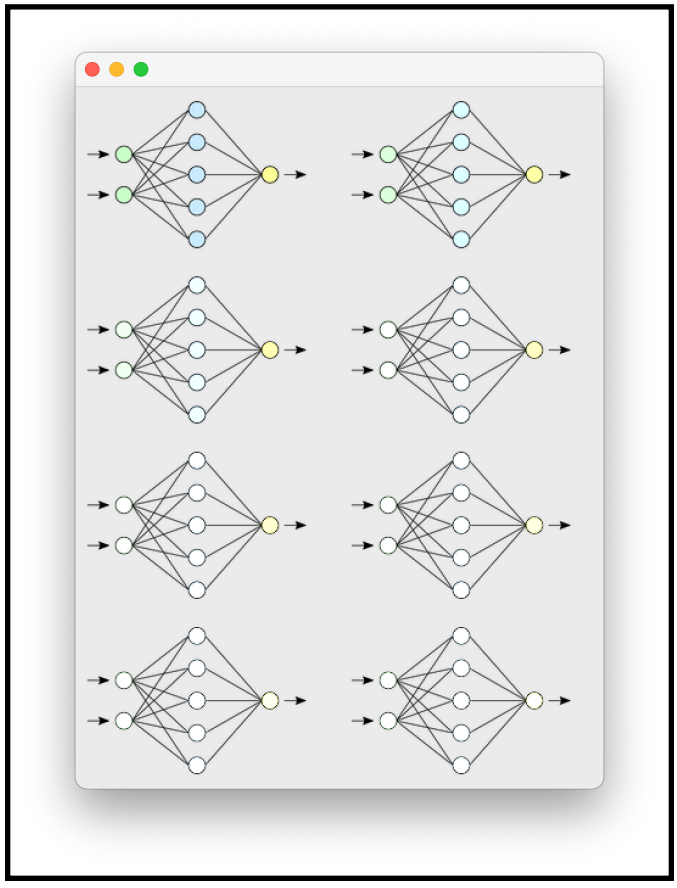


Figure 7-6: An example output to a program that modifies image brightness.

Notice how each image varies slightly from each other, but are all still discernible as a neural network diagram. Adjusting brightness/exposure changes how bright colors are, so pixels that are black will remain black and pixels that are white will remain white. As you increase the brightness of a picture, the pixels' colors turn lighter and end up becoming white, and as you decrease the brightness of a picture, the pixel's colors turn darker and may end up becoming black. Likewise, a dataset consisting of images that aren't black and white can be augmented by adjusting their brightness levels.

Task 3

Add noise to an image.

```
01  import javax.imageio.ImageIO;
02  import java.awt.*;
03  import java.awt.image.BufferedImage;
04  import java.io.File;
05  import java.io.IOException;
06  import java.util.Random;
07
08  public class NoiseImage {
09    public static void main(String[] args)
                                throws IOException {
10      BufferedImage image = ImageIO.read(
                               new File("src/digit.png"));
11      BufferedImage output = new BufferedImage(28, 28,
                                BufferedImage.TYPE_INT_ARGB);
12      Random random = new Random();
13      for (int i = 0; i < output.getHeight(); i++) {
14        for (int j = 0; j < output.getWidth(); j++) {
15          Color color = new Color(
                                     image.getRGB(j, i), true);
16          if (isTransparent(image.getRGB(j, i))) {
17            color = new Color(255, 255, 255);
18          }
19          int choice = random.nextInt(2);
```

```
20            int r = color.getRed();
21            int g = color.getGreen();
22            int b = color.getBlue();
23            int a = color.getAlpha();
24
25            int rand = random.nextInt(50);
26            if (choice == 0) {
27              r += rand;
28              g += rand;
29              b += rand;
30              a += rand;
31            } else {
32              r -= rand;
33              g -= rand;
34              b -= rand;
35              a -= rand;
36            }
37
38            r = isOutOfBounds(r);
39            g = isOutOfBounds(g);
40            b = isOutOfBounds(b);
41            a = isOutOfBounds(a);
42            output.setRGB(j, i,
                    new Color(r, g, b, a).getRGB());
43          }
44        }
45        File outFile = new File("output.png");
46        ImageIO.write(output, "png", outFile);
47      }
48
49      private static int isOutOfBounds(int val) {
50        if (val > 255) {
51          return 255;
52        }
53        return Math.max(val, 0);
54      }
55
56      private static boolean isTransparent(int val) {
57        return val >> 24 == 0;
58      }
59    }
```

Before I start giving an extensive explanation, let's take a look at an example input and output to the program!

Figure 7-7: *An example input to our image noise adding program.*

This is an image of a 0 from the MNIST dataset. Now, let's see its output:

Figure 7-8: *An example output to our image noise adding program.*

The image is slightly small, so I modified the program to perform the noise addition to larger images:

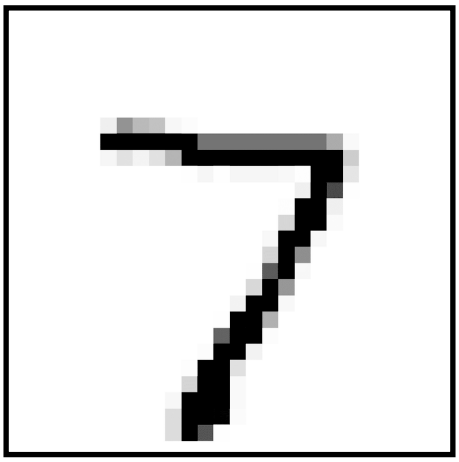


Figure 7-9: *Another example input to our image noise adding program.*

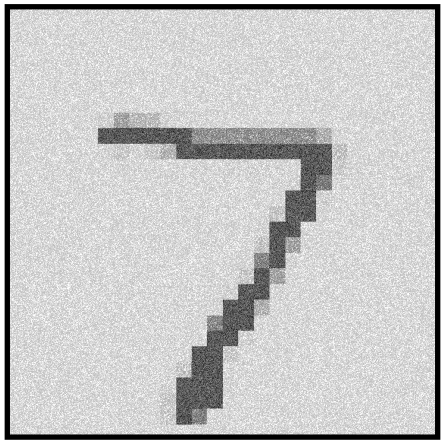


Figure 7-10: *Another example output to our image noise adding program.*

By combining these three programs, you would be able to augment datasets monumentally, which will help improve the accuracy of your networks.

Code explanation:

Line 1-6: Import necessary libraries

Line 10: Read an image from a filepath.

Line 11: Initialize an output image of the same width and height encoded in ARGB format.

Line 12: Initialize a Random object to generate noise values.

Line 13-44: For each pixel in output, get its corresponding color on the input image (which has an alpha filter) and check whether it is transparent. If it is, then set the color to white, otherwise, set the R, G, B, A values to those found on the original pixel. Have a random choice between 0 and 1 decide whether a random value between 0-49 is subtracted from or added to the original pixel's color. Then, check whether the values for RGBA are smaller than 0 or greater than 255, in which case set the value to 0 and 255 respectively. Finally, set the

corresponding pixel on the output image to the new color using the RGBA values.

Line 45: Initialize the file to which the new image is written to.

Line 46: Write the output image to the file.

Line 49-54: Check if a value is greater than 255 or less than 0. If it is greater than 255, set value to 255. Otherwise, if value less than 0, set value to 0. Note that the value is returned, not directly changed. Also, if the pixel is in normal range (0-255), the `Math.max(val, 0)` will return the normal pixel's value.

Line 56-58: Check if a pixel is transparent. This function uses the `>>` operator, which shifts the binary representation of `val` to the right 24 times. If that value is equal to 0, then output `true`. For example, RGBA color `-1023410176` (`rgba: 0, 0, 0, 195`) can be represented in binary as `-1111010000000000000000000000000`, which when shifted 24 times to the right produces `-111101`, which isn't 0, therefore the pixel must be non-transparent.

That's about it for this chapter! Read the chapter summary and head on!

Chapter Summary

1. I*mage **preprocessing*** defines the process to which an image is processed and modified before use in the future. On the other hand, *image **processing*** instead just refers to the broader subject of processing/modifying an image.

2. Image processing is separated into two categories: *digital image processing* and *analogue image processing*. Analog image processing refers to processing applied to analog signals. This processing is usually done through a camera lens. Notice that at this stage, various processing techniques are also available (and thus called *analog image processing*): you can adjust multiple different parts of the camera like the aperture size or focal length, which results in changes to the image (i.e., perspective, distortion).

3. Digital processing techniques are performed through the manipulation of images, which is done using various algorithms.

4. The aim of digital image processing is to enhance images (or particular features in an image) through the removal/lightening of image noise and distortions. Image preprocessing is performed before feeding a model the input data, and can be performed on both training and testing data. It is also useful because it can help artificial intelligence models learn better and more quickly. Processing images can also reduce the likelihood of overfitting and underfitting.

5. Some examples of digital image processing techniques include: resizing/dilation, orientation, coloring (i.e., through contrast, filters, greyscale adjustments), and deskewing (opposite of skewing, which is a technique used in data augmentation).

6. Image data augmentation is a process undertaken to increase the size of image datasets through the use of various augmentation techniques.

7. Augmenting datasets is useful as models learn better with more variance in data, which can help prevent model underfitting and overfitting. Data augmentation is performed on data before training and is usually only applied to the training dataset.

8. Some common augmentation techniques include rotating, skewing (slanting), translating and resizing images. Sometimes, datasets can also be augmented through the adjustment of the brightness (or even opacity/transparency) of an image.

9. Data augmentation is not better than having a dataset with original data as augmentation cannot possibly account for (and help a network *generalize*) all possible inputs that the network may be faced with.

10. Data augmentation and preprocessing can be performed on various other forms of data, not just image data. For example, it can be performed on audio and numerical data.

11. Combining various augmentation techniques can increase the amount of augmented data created from original data.

Chapter 8 - Backpropagation

> *"It is only with the heart that one can see rightly; what is essential is invisible to the eye."*
> *— The Little Prince by Antoine de Saint-Exupéry*

I've discussed a lot about aspects of the issues that may occur during the learning stage in machine learning, but I haven't yet introduced the most generalized learning algorithm for neural networks: *backpropagation* (a generalized version of the delta rule). We've learned about the *delta rule*, but it cannot be used to completely train a neural network. So now a good time to introduce backpropagation! In this chapter, you will learn about the mathematics behind backpropagation and also head into some practical exercises. If you can recall, we created a neural network architecture in Task 1 (MNIST/Digit recognition) consisting of a *network* object that stores *layers* (which store *neurons*). That architecture worked quite well for starters, but isn't very efficient. In this chapter, we'll explore another path to implementing neural networks with two fundamental data structures: *vectors* and *matrices*. I'll also discuss the implementation of *thread parallelism*, which can give a substantial performance speedup to our programs.

Derivation

Recall that there are several components in a neural network that are used during training: activation functions, an error/loss function, weights, inputs, and biases. Note that the notion of learning is adjusting parameters of a network to minimize the error function, thus we need to model a relationship between each specific parameter and its effect on the loss function. This can also be represented by the sensitivity of the error function to changes in a specific parameter, which again involves derivatives. To derive backpropagation, we'll need the chain rule, which as a reminder is expressed as:

$$\frac{\partial f(g(x))}{\partial x} = f'(g(x)) \cdot g'(x) \text{ or } \frac{\partial f(g(x))}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$$

Let's first define notation and variables for our parameters:

- Weights are represented by: $w_{i,j}$ where the weight represents a connection between the *i*th neuron of the current layer and *j*th neuron of the next layer. A variable L will represent the current layer.

- Biases are represented by: b_j where the bias is added to the weighted sum of w and x for the *j*th neuron in the next layer.

- Inputs are represented by: x_i, for an input of the *i*th neuron in the current layer.

- Activation function: $\sigma(x)$

- Cost function: $C(t, y)$, for t being the target output and y being the network output.

Then, we find that a forward pass through the network can be represented as:

$$z_{j,L} = \sum_{k=1}^{i} (w_{k,j,L} \cdot x_{k,L}) + b_{j,L}$$
$$x_{j,L} = \sigma(z_{j,L})$$

The cost function (assuming mean-squared error) is:

$$C(t, y) = \sum_{k=1}^{i} (t_k - y_k)^2$$

To adjust our weights, we need to find:

$$\frac{\partial C(t,y)}{\partial w_{i,j,L}} = \frac{\partial C(t,y)}{\partial o_{j,L}} \frac{\partial o_{j,L}}{\partial z_{j,L}} \frac{\partial z_{j,L}}{\partial w_{i,j,L}}$$

Notice that:

$$\frac{\partial z_L}{\partial w_{i,j,L}} = \frac{\partial [\sum_{k=1}^{i} (w_{k,j,L} \cdot x_{k,L}) + b_{j,L}]}{\partial w_{i,j,L}} = x_{i,L}$$

Because the only term in the summation that includes $w_{i,j,L}$ is $w_{i,j,L} \cdot x_{i,L}$. Also:

$$\frac{\partial o_L}{\partial z_L} = \sigma'(z_L)$$

As $o_L = \sigma(z_L)$.

The only term that we cannot directly take a derivative of is $\frac{\partial C(t,y)}{\partial o_L}$, as o_L could refer to the output of any layer in a multi-layer network. What can be done, then?

Let's use an example to help us solve this problem. Following this paragraph is an image depicting a neural network architecture with structure 5-4-2. Biases are the nodes in green and are distinguishable because there are no connections leading *towards* them.

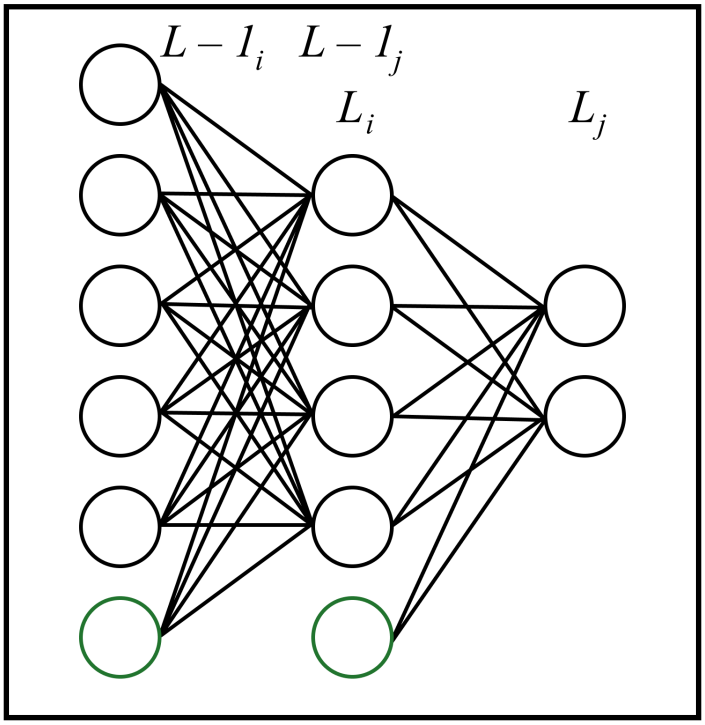


Figure 8-1: *A multilayer network, where green nodes represent biases. How should the first-second layer weight connections learn?*

First, notice that we can use the delta rule to adjust the weights connecting the second last layer and last layer of the network as:

$$\begin{aligned}\frac{\partial C(t,y)}{\partial o_L} &= \frac{\partial \sum_{k=1}^{j}(t_k - o_{k,L})^2}{\partial o_L} \\ &= \frac{\partial (t - o_L)^2}{\partial o_L} \\ &= 2(o_L - t)\end{aligned}$$

The network output is always equivalent to o_L for the weights connecting to the last layer. Combining this with all previous terms, we find that the gradient (amount to change, denoted with Δ) for weights in the last layer is:

$$\Delta w_{i,j,L} = \alpha \cdot 2(o_j - t_j)x_i\sigma'(z_{j,L})$$

With the usual learning rate α.

Notice that a weight from the first layer will directly affect the input to the last layer, which in turn directly affects the output of the network.

So, let us find how the input to the middle layer changes with respect to each weight in the first layer (of the example):

$$\begin{aligned}\frac{\partial x_{i,L}}{\partial z_{j,L-1}}\frac{\partial z_{j,L-1}}{\partial w_{i,j,L-1}} &= \frac{\partial \sigma(z_{j,L-1})}{\partial z_{j,L-1}}\frac{\partial z_{j,L-1}}{\partial w_{i,j,L-1}} \\ &= \sigma'(z_{j,L-1}) \cdot x_{i,L-1}\end{aligned}$$

Notice that this is identical to the derivative of the last two terms for the update formula for the last layer in a network. Then, to find how the weight affects the final output, we *multiply* the above term with how each input to the middle layer changes the cost of the network. Since the cost function is the sum of the losses of both the output neurons, the derivative must be the sum of how the input changes each respective output neuron's loss:

$$\begin{aligned}\sum_{k=1}^{j}\frac{\partial C(t_k, y_k)}{\partial x_{i,L}} &= \sum_{k=1}^{j}\frac{\partial C(t_k, y_k)}{\partial o_{k,L}}\frac{\partial o_{k,L}}{\partial z_{k,L}}\frac{\partial z_{k,L}}{\partial x_{i,L}}\end{aligned}$$

(Note that $x_{i,L}$ is the input to the last layer as i refers to the ith neuron in the current layer [the next layer is $L-1+1=L$])

Simplifying, we get:

$$\begin{aligned}\sum_{k=1}^{j}\frac{\partial C(t_k, y_k)}{\partial o_{k,L}}\frac{\partial o_{k,L}}{\partial z_{k,L}}\frac{\partial z_{k,L}}{\partial x_{i,L}} &= \sum_{k=1}^{j} 2(o_{k,L} - t_k) \cdot \sigma'(z_{k,L}) \cdot w_{i,k,L}\end{aligned}$$

Now, by multiplying this term with the derivative of how the input to the middle layer changes with respect to weights in the first layer, we get the gradient for updates to the weights in the first layer:

$$\Delta w_{i,j,L-1} = \alpha \cdot \sigma'(z_{j,L-1}) \cdot x_{i,L-1} \cdot [\sum_{k=1}^{j} 2(o_{k,L} - t_k) \cdot \sigma'(z_{k,L}) \cdot w_{i,k,L}]$$

Well… we're done now! Notice that we can recursively back-propagate through our network, starting from the last layer and finishing on the first. Also, the terms in the rightmost summation will have been previously computed when we back-propagated the through the layer (as weight updates in the next layer are computed before the current layer). So, all in all, we have:

$$\Delta w_{i,j,L} = \alpha \cdot 2(o_j - t_j) x_i \sigma'(z)$$

$$\Delta w_{i,j,L-1} = \alpha \cdot \sigma'(z_{j,L-1}) \cdot x_{i,L-1} \cdot [\sum_{k=1}^{j} 2(o_{k,L} - t_k) \cdot \sigma'(z_{k,L}) \cdot w_{i,k,L}]$$

Where we have $\Delta w_{i,j,L}$ being the weight update for the last layer, and $\Delta w_{i,j,L-1}$ being the weight update for preceding layers. We additionally multiplied the terms by a learning rate α. Now, weight updates look like:

$$w_{i,j,l} = w_{i,j,l} - \Delta w_{i,j,l}$$

But, wait… what about our biases? We haven't yet derived their update formulas! But notice that biases are just weights multiplied by an always-constant input value 1 (so x is always 1 for biases), so we only need to modify our equations a little bit to reflect this:

$$\Delta b_{j,L} = \alpha \cdot 2(o_j - t_j) \sigma'(z_{j,L})$$

$$\Delta b_{j,L-1} = \alpha \cdot \sigma'(z_{j,L-1}) \cdot [\sum_{k=1}^{j} 2(o_{k,L} - t_k) \cdot \sigma'(z_{k,L}) \cdot w_{i,k,L}]$$

$$b_{j,l} = b_{j,l} - \Delta b_{j,l}$$

That was a lot of information in just a few pages! If you weren't able to grasp the underlying mathematical notation or how derivatives work in general, try

search around for some short and simple explanations for them online. If you're still stuck, I recommend you reread this chapter, or just continue onwards to the practical implementation section.

Application

Now that we have the essential ingredients for using backpropagation for parameters updates, it is time to introduce two new structures that we will use to help store our parameters more efficiently: vectors and matrices. Afterwards, I'll just head straight on to a step-by-step implementation of these structures. We'll attempt to make a full-on neural network architecture like the production-level standard ones (e.g., PyTorch, Tensorflow).

Vectors are essentially what programmers call *arrays* or *lists* (although the two differ slightly). Usually, they are expressed with brackets ('[]'), and store elements delimited by commas. In our implementation, we'll create vector objects from double arrays. A fundamental operation that will be required in our implementation is an operation called the *Hadamard product*, or element-wise multiplication. As the latter name suggests, a Hadamard product takes the product of corresponding elements in a pair of vectors and returns a vector of the same size. Of course, this requires that the vectors be of the same length.

A **matrix** is essentially *a vector of vectors*. In programming context, you could either represent matrices as a two-dimensional array (e.g., `matrix = double[][]`) or (as suggested) an array/vector of vectors (e.g., `matrix = Vector[]`). You can also perform a Hadamard product on matrices, but in

our implementation, we'll just have them store vectors and perform operations on vector values (i.e., for our neural network to learn).

Now that you know the basics of vectors and matrices, it's time to implement them! Try to follow along with a text editor (or most preferably, an IDE like IntelliJ CE [free] or Ultimate). Let's start with vectors. First, let's define a general vector class with three constructors:

```
class Vector {
  private double[] vector;

  Vector(int length) {
    vector = new double[length];
  }

  Vector(double[] values) {
    vector = values;
  }

  public Vector(int[] values) {
    vector = new double[values.length];
    for (int i = 0; i < values.length; i++) {
      vector[i] = values[i];
    }
  }
}
```

You can initialize a vector by specifying values directly or by supplying a length and filling an array of that length with values later on. Let's also define a Hadamard product method that performs the Hadamard product on the current vector and another vector (and return a *new* vector with the result). Alongside this, we'll also define element-wise division:

```
int length() {
  return this.vector.length;
}

Vector mult(Vector anotherVector) {
```

```
  if (anotherVector.length() != this.length()) {
    throw new IllegalArgumentException(
          "mult(): Vectors not of same size: {" +
           this.length() + ", " + anotherVector.length() +
          "}.");
  }

  Vector output = new Vector(this.length());
  for (int i = 0; i < output.length(); i++) {
    output.vector[i] =
           this.vector[i] * anotherVector.vector[i];
  }

  return output;
}

Vector div(Vector anotherVector) {
  if (anotherVector.length() != this.length()) {
    throw new IllegalArgumentException(
              "div(): Vectors not of same size: {" +
               this.length() + ", " + anotherVector.length() +
              "}.");
  }

  Vector output = new Vector(this.length());
  for (int i = 0; i < output.length(); i++) {
    output.vector[i] =
           this.vector[i] / anotherVector.vector[i];
  }

  return output;
}
```

Let's also define an element-size summation and a way to total all values in the vector:

```
double total() {
  double sum = 0;
  for (double val : this.vector) {
    sum += val;
  }
  return sum;
```

```
}

Vector add(Vector anotherVector) {
  if (anotherVector.length() != this.length()) {
    throw new IllegalArgumentException(
          "add(): Vectors not of same size: {" +
           this.length() + ", " + anotherVector.length() +
          "}.");
  }

  Vector output = new Vector(this.length());
  for (int i = 0; i < output.length(); i++) {
    output.vector[i] =
           this.vector[i] + anotherVector.vector[i];
  }

  return output;
}
```

We'll also define some other methods for convenience later on:

```
Vector mult(Vector anotherVector) {
  if (anotherVector.length() != this.length()) {
    throw new IllegalArgumentException(
          "mult(): Vectors not of same size: {" +
           this.length() + ", " + anotherVector.length() +
          "}.");
  }

  Vector output = new Vector(this.length());
  for (int i = 0; i < output.length(); i++) {
    output.vector[i] =
           this.vector[I] * anotherVector.vector[i];
  }

  return output;
}

Vector mult(double value) {
  Vector output = new Vector(this.length());
  for (int i = 0; i < output.length(); i++) {
    output.vector[i] = this.vector[i] * value;
  }
```

```
  return output;
}

Vector add(double value) {
  Vector output = new Vector(this.length());
  for (int i = 0; i < output.length(); i++) {
    output.vector[i] = this.vector[i] + value;
  }
  return output;
}

Vector subtract(Vector anotherVector) {
  if (anotherVector.length() != this.length()) {
    throw new IllegalArgumentException(
          "subtract(): Vectors not of same size: {" +
           this.length() + ", " + anotherVector.length() +
          "}.");
  }

  Vector output = new Vector(this.length());
  for (int i = 0; i < output.length(); i++) {
    output.vector[i] =
           this.vector[i] - anotherVector.vector[i];
  }

  return output;
}
```

Then, let's define a few methods to initialize our vector, and also a handy way to print it to the console. Additionally, we'll have getters and setters:

```
import java.util.Arrays;
import java.util.Random;

...
private Random rand = new Random();

Vector fill(double value) {
  for (int i = 0; i < this.length(); i++) {
    this.vector[i] = value;
  }
```

```
  return this;
}

Vector fillGaussian() {
  for (int i = 0; i < length(); i++) {
    this.vector[i] = rand.nextGaussian();
  }
  return this;
}

Vector fillGaussian(double average, double deviation) {
  for (int i = 0; i < length(); i++) {
    this.vector[i] =
          rand.nextGaussian() * deviation + average;
  }
  return this;
}

Vector fillRandom() {
  for (int i = 0; i < length(); i++) {
    this.vector[i] = rand.nextDouble();
  }
  return this;
}

double get(int index) {
  return vector[index];
}

void set(int index, double value) {
  vector[index] = value;
}

double[] values() {
  return vector;
}

@Override
public String toString() {
  return Arrays.toString(vector);
}
```

That's it for our Vector class! You can see the full source code at the end of this chapter and also on Github with this book. In continuation, let's define a Layer class that is a wrapper around an array of vectors. But before doing so, let's also define enumerations for activation and error functions:

```
public enum ActivationFunction {
  ELU,
  IDENTITY,
  LEAKY_RELU,
  RELU,
  SIGMOID,
  SOFTMAX,
  SOFTPLUS,
  TANH
}

public enum Error {
  MEAN_SQUARED,
  BINARY_CROSS_ENTROPY,
  CATEGORICAL_CROSS_ENTROPY
}
```

You might have noticed that I've introduced two new activation functions. ELU stands for Exponential Linear Unit, and is a variation of RELU (and Leaky-RELU), taking the form: $\sigma(x) = \begin{cases} m(e^x - 1), & x \leq 0 \\ x, & x > 0 \end{cases}$, where m is a constant (I'll use 1). The derivative of ELU is: $\sigma'(x) = \begin{cases} e^x, & x < 0 \\ 1, & x \geq 0 \end{cases}$ (for $m = 1$).

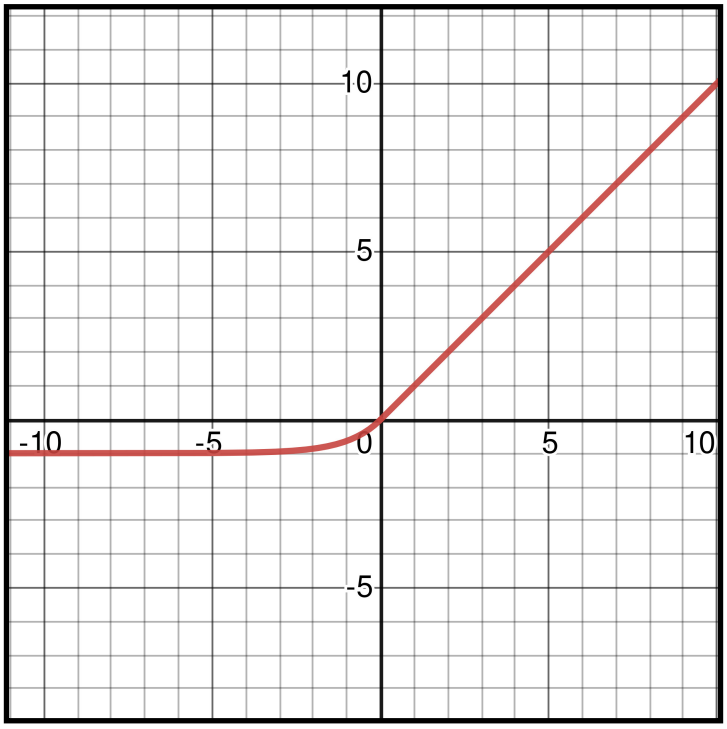


Figure 8-2: Graph of ELU activation function.

The Softmax activation function takes the form: $\sigma(x)_i = \frac{x_i}{\sum_{j=1} x_j}$. The sum of its outputs adds to 1, so it can be used to express network predictions as probabilities. Softmax is usually used with another type of error function called *cross entropy*, which we will also implement in this chapter. Softmax's derivative is: $\sigma'(x)_i = \sigma(x_i)(\delta_{i,j} - \sigma(x_j))$ for i being each index in the output vector and j being the index of the neuron whose output is currently being used for training. Here, $\delta_{i,j}$ is the Kronecker delta, and represents: $\delta_{i,j} = \begin{cases} 0, & i \neq j \\ 1, & i = j \end{cases}$.

The categorical cross-entropy loss function takes the form: $E(t, o) = -t \cdot \ln(o)$ (with derivative $(o - t)$ when paired with Softmax), and is designed for use in categorical classification tasks (tasks where networks classify data that belongs to multiple classes). Binary cross-entropy takes the form:

$E(t, o) = -(t \cdot \ln(o) + (1 - t) \cdot \ln(1 - o))$ (with derivative $\frac{o - t}{o(1 - o)}$), and is used for binary classification tasks (tasks where networks classify data into one of two classes/labels). I'll explain why cross-entropy is used later. Note that the logarithm specified in cross-entropy is the natural logarithm, although some implementations use the base-2 logarithm. There isn't much difference between which implementation you choose. Since cross-entropy requires the natural logarithm, let's implement that function in the vector class:

```
Vector log() {
  Vector out = new Vector(length());
  for (int i = 0; i < length(); i++) {
    out.set(i, Math.log(vector[i]));
  }
  return out;
}
```

Let's start on the Layer class. First, we'll define two constructors:

```
class Layer {
  private final boolean isOutputLayer;
  private final ActivationFunction activationFunction;
  private final Error errorType;
  Vector[] vectors;
  Vector bias;

  Layer(boolean isOutputLayer,
        ActivationFunction activationFunction, Error error) {
    this.isOutputLayer = isOutputLayer;
    this.activationFunction = activationFunction;
    this.errorType = error;
  }

  Layer(Vector[] vectors, Vector bias, boolean isOutputLayer,
        ActivationFunction activationFunction, Error error) {
    this.vectors = vectors;
    this.bias = bias;
    this.isOutputLayer = isOutputLayer;
    this.activationFunction = activationFunction;
```

```
    this.errorType = error;
    initDeltaArray();
  }
}
```

The user of the class can initialize a layer with several values:

- A boolean value indicating whether the layer is the output layer of a network.
- An ActivationFunction object indicating what activation function is used by the layer.
- An error function denoting the error function used during training.
- A Vector array (in the second constructor option) that is passed on to the layer. The given array should be the weights of the layer. You'll see why this constructor may be useful later on.
- A Vector object (in the second constructor option) that is passed to the layer. This object should store the biases in the layer.

In our architecture, each Vector object storing weights stores them as illustrated below:

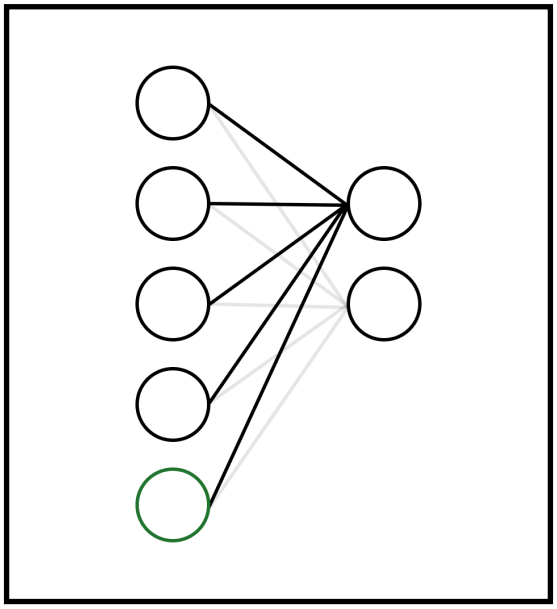


Figure 8-3: How our vector class stores weights.

This means, each vector object will have *i* weights, and our `vectors` variable will store *j* vectors. Here, *i* is the number of neurons in the current layer and *j* the number of neurons in the next layer.

Note that in the second constructor, we initialize our delta arrays, which are arrays that store the amount to change each parameter (weight or bias) after each learning example. Additionally, we'll also initialize our display error variable (more on this variable later).

```
...

  private Vector[] deltas;
  private Vector deltaBias;
  Vector displayError;

  private void initDeltaArray() {
    deltas = new Vector[vectors.length];
    for (int i = 0; i < vectors.length; i++) {
      deltas[i] = new Vector(vectors[i].length());
    }
```

```
    deltaBias = new Vector(vectors.length);

    if (isOutputLayer) {
      displayError = new Vector(vectors.length);
    }
  }
```

Next, we'll allow users to choose between mini-batch, stochastic and normal gradient descent by specifying a batch size parameter:

```
  private int BATCH_SIZE = 1;

  void setBatchSize(int BATCH_SIZE) {
    this.BATCH_SIZE = BATCH_SIZE;
  }
```

We'll also have some convenience methods that get data from the layer.

```
  int length() {
    return this.vectors.length;
  }

  Vector error;

  Vector getError() {
    return error;
  }

  Vector getDisplayError() {
    return displayError;
  }

  void resetDisplayError() {
    if (displayError == null || !isOutputLayer) {
      return;
    }
    displayError = displayError.mult(0);
  }

  private Vector output;

  Vector getOutput() {
```

```
    return output;
  }

  Vector getBias() {
    return bias;
  }

  ActivationFunction getActivationFunction() {
    return activationFunction;
  }

  Error getErrorType() {
    return errorType;
  }
```

Above, the output variable is quite self-explanatory (the output of the layer), but you may be confused by the two error variables `displayError` and `error`. The former refers to the error we display in the console, and the latter refers to the error value we use during training. Note that the difference between these two variables and how they are calculated is that the `error` variable stores the result of performing the derivative of the error function (e.g., $2(o - t)$ for MSE) whilst the `displayError` variable stores the error calculated with the error function (e.g., $(t - o)^2$ for MSE). Note also that `error` is replaced by a new value for each training sample the network learns from, whereas `displayError` sums up all the errors and only resets after each epoch. To reset it, we'll provide a method called `resetDisplayError()`. Next, let's define a method that gives the layer its input and makes it feed the input through the network. We store the weighted sum of the product of our weights and the input; and output (which is the weighted sum passed through an activation function) into separate variables:

```
...
  private Vector weightedSum;
```

```
  Vector test(Vector input) {
    if (input.length() != vectors[0].length()) {
      throw new IllegalArgumentException("test(Vector): Input"
            + " vector not the same size as weight vectors.");
    }

    Vector output = new Vector(vectors.length);
    for (int i = 0; i < vectors.length; i++) {
      output.set(i, vectors[i].mult(input).total());
    }

    output = output.add(bias);
    this.weightedSum = output;

    output = activation(weightedSum);

    return output;
  }

  private Vector input;

  void feed(Vector input) {
    this.input = input;
    this.output = test(input);
  }
```

Note that `feed()` is just a wrapper around `test()` that stores the input and output values (we can use `test()` to test the network). We use a function `activation()`, which looks like this:

```
...
  private Vector activation(Vector val) {
    int valSize = val.length();
    Vector out = new Vector(valSize);
    for (int index = 0; index < valSize; index++) {
      switch (activationFunction) {
        case IDENTITY:
          return val;
        case TANH:
          out.set(index, tanh(val.get(index)));
          break;
```

```
      case RELU:
        out.set(index, relu(val.get(index)));
        break;
      case LEAKY_RELU:
        out.set(index, leaky_relu(val.get(index)));
        break;
      case SIGMOID:
        out.set(index, sigmoid(val.get(index)));
        break;
      case SOFTPLUS:
        out.set(index, softplus(val.get(index)));
        break;
      case ELU:
        out.set(index, elu(val.get(index)));
        break;
      case SOFTMAX:
        return softmax(val);
      default:
        throw new IllegalArgumentException(
              "activation(): No such ActivationFunction '" +
               activationFunction + "'.");
    }
  }

  return out;
}

private Vector softmax(Vector val) {
  Vector out = new Vector(val.length());
  double total = 0.0;

  for (int i = 0; i < val.length(); i++) {
    total += Math.exp(val.get(i));
  }

  if (total == 0.0) {
    System.out.println(val);
    throw new ArithmeticException(
    "softmax(Vector val): total is 0, cannot divide by 0.");
  }
  for (int i = 0; i < val.length(); i++) {
    out.set(i, (Math.exp(val.get(i)) / total));
```

```
    }

    return out;
  }

  private double elu(double val) {
    return val > 0 ? val : Math.exp(val) - 1.0;
  }

  private double sigmoid(double val) {
    return 1.0 / (1.0 + Math.exp(-val));
  }

  private double relu(double val) {
    return val <= 0 ? 0 : val;
  }

  private double leaky_relu(double val) {
    return val <= 0 ? 0.01 * val : val;
  }

  private double softplus(double val) {
    return Math.log(1.0 + Math.exp(val));
  }

  private double tanh(double val) {
    return (Math.exp(val) - Math.exp(-val)) /
           (Math.exp(val) + Math.exp(-val));
  }
```

These are just literal translations of the activation functions you've seen in this chapter and the main activation functions chapter. We'll also define a `derivative()` method that takes the index of the currently being trained set of weights (which is used for the Softmax derivative) and outputs the derivative of the activation function being used (with respect to the network output). I've made references to the index bold so that they're easier to spot, although note that it's only used in the Softmax derivative (to perform the Kronecker delta).

```
...
  Vector derivative(int index) {
    int outLength = output.length();
    Vector returnVector = new Vector(outLength);
    switch (activationFunction) {
      case IDENTITY:
        return new Vector(output.length()).fill(1.0);
      case TANH:
        for (int i = 0; i < outLength; i++) {
          returnVector.set(i, 1.0 - output.get(i) *
                                    output.get(i));
        }
        return returnVector;
      case RELU: {
        for (int i = 0; i < outLength; i++) {
          returnVector.set(i, output.get(i) < 0.0 ? 0.0 :
                                                    1.0);
        }
        return returnVector;
      }
      case LEAKY_RELU: {
        for (int i = 0; i < outLength; i++) {
          returnVector.set(i, output.get(i) < 0.0 ? 0.01 :
                                                    1.0);
        }
        return returnVector;
      }
      case SIGMOID:
        return output.mult(output.mult(-1.0).add(1.0));
      case SOFTMAX: {
        for (int i = 0; i < outLength; i++) {
          if (i == index) {
            returnVector.set(i, output.get(i) *
                          (1.0 - output.get(i)));
          } else {
            returnVector.set(i, -1.0 * output.get(i) *
                                      output.get(index));
          }
        }

        return returnVector;
      }
```

```
        case SOFTPLUS: {
          for (int i = 0; i < outLength; i++) {
            returnVector.set(i, sigmoid(weightedSum.get(i)));
          }
          return returnVector;
        }
        case ELU: {
          for (int i = 0; i < outLength; i++) {
            if (output.get(i) >= 0) {
              returnVector.set(i, 1.0);
            } else {
              returnVector.set(i, Math.exp(weightedSum.get(i)));
            }
          }

          return returnVector;
        }
        default:
          throw new IllegalArgumentException(activationFunction
                  + " is not a valid ActivationFunction type.");
      }
    }
```

That was a lot of code! If you've been blindly copying it into a local text editor or IDE, take some time to read through it and understand what each part means. Then continue onwards for one of the most essential methods in the layer class: `learn()`. We first define a variable that keeps track of the currently being trained example. This way, once we reach the batch size, we know that it is time to update our weights and biases.

```
  private int currentInputBatchId = 0;
```

In our learning method, we'll accept a vector consisting of the target output (if the layer is the output layer, otherwise `null` is passed), next layer (used for training in hidden layers, `null` for the output layer), and a double-precision floating-point value referring to the learning rate. If the user of the class

accidentally sets the learning rate to 0, we'll ping an error message notifying the mistake instead of training a network with $\alpha = 0$.

```
void learn(Vector targetOutput,
           Layer nextLayer, double alpha) {
  if (alpha == 0) {
    throw new NullPointerException(
        "learn(): learning rate can't be 0 (no learning).");
  }
```

Then comes the main backpropagation part. Recall that the formula for backpropagation is:

$$\Delta w_{i,j,L} = \alpha \cdot 2(o_j - t_j)x_i\sigma'(z)$$

$$\Delta w_{i,j,L-1} = \alpha \cdot \sigma'(z_{j,L-1}) \cdot x_{i,L-1} \cdot [\sum_{k=1}^{j} 2(o_{k,L} - t_k) \cdot \sigma'(z_{k,L}) \cdot w_{i,k,L}]$$

$$w_{i,j,l} = w_{i,j,l} - \Delta w_{i,j,l}$$

$$\Delta b_{j,L} = \alpha \cdot 2(o_j - t_j)\sigma'(z_{j,L})$$

$$\Delta b_{j,L-1} = \alpha \cdot \sigma'(z_{j,L-1}) \cdot [\sum_{k=1}^{j} 2(o_{k,L} - t_k) \cdot \sigma'(z_{k,L}) \cdot w_{i,k,L}]$$

$$b_{j,l} = b_{j,l} - \Delta b_{j,l}$$

If the layer is the output layer, we calculate the derivative of the error function with respect to the ideal output and network output. We then multiply this by the derivative of the layer's activation function. Then, we cache this value for use in training layers that aren't the final layer of the network. Then, we multiply the error by the input and learning rate, and add the result to our delta array, which stores the changes that should be added to the weights after each batch of examples. If the layer is a hidden layer, we multiply the weights in the next layer that are connected to the output the current weight contributes to with the activation function derivative (and cache the result), input and learning rate. Below is the implementation:

```
learn()
(import java.util.stream.IntStream;)
...
    IntStream.range(0, vectors.length)
              .parallel().forEach(index -> {
      Vector error;
      if (isOutputLayer) {
        Vector[] errorCalc = calcError(output, targetOutput);
        error = errorCalc[0];
        displayError = displayError.add(errorCalc[1]);
      } else {
        error = new Vector(vectors.length);
        for (int indice = 0;
             indice < nextLayer.vectors.length; indice++) {
         error.set(index, error.get(index) +
                          nextLayer.vectors[indice].get(index)
                        * nextLayer.getError().get(indice));
        }
      }
      error = error.mult(derivative(index));
      this.error = error;

      Vector delta = input.mult(error.get(index));
      delta = delta.mult(alpha);

      deltas[index] = deltas[index].add(delta);
      deltaBias.set(index, deltaBias.get(index) +
                           error.get(index) * alpha);
    });
```

Note that `IntStream.range().parallel().forEach` simply iterates through our vector array in parallel. This helps speed up our program, but will create a higher load on your computer's CPU (or GPU). Finally, we increment the BatchID of the current input sample, and check whether it is equal to our batch size. If it is, then we update our weights and biases, reset the delta cached values and BatchID.

```
learn()
...
    currentInputBatchId++;
```

```
    if (currentInputBatchId == BATCH_SIZE) {
      currentInputBatchId = 0;
      for (int index = 0; index < vectors.length; index++) {
        vectors[index] = vectors[index]
                                .subtract(deltas[index]);
        deltas[index] = deltas[index].mult(0.0);
      }
      bias = bias.subtract(deltaBias);
      deltaBias = deltaBias.mult(0.0);
    }
  }
```

We used another helper called `calcError()` in our learning method, which calculates the network error as an array, with the 0th index being the error derivative with respect to layer output, and the 1st index being the display error:

```
  Vector[] calcError(Vector output,
                             Vector targetOutput) {
    switch (errorType) {
      case MEAN_SQUARED: {
        Vector err = (output.subtract(targetOutput));
        return new Vector[]{err.mult(2), err.mult(err)};
      }

      case BINARY_CROSS_ENTROPY: {
        Vector display = (targetOutput.mult(output.log())
                      .add((targetOutput.mult(-1.0).add(1.0)))
                      .mult(output.mult(-1.0).add(1.0).log()))
                      .mult(-1.0);
        return new Vector[]{
               (output.subtract(targetOutput))
                      .div(output.mult(output.mult(-1.0)
                                             .add(1.0)))
               , display
        };
      }

      case CATEGORICAL_CROSS_ENTROPY: {
        Vector display = targetOutput.mult(output.log())
```

```
                                        .mult(-1.0);
        return new Vector[]{
                output.subtract(targetOutput),
                display
        };
      }

      default: {
        throw new IllegalArgumentException(
        "calcError(): Error type " + errorType + " unknown.");
      }
    }
  }
```

That was a lot! So I figured we should end this class with something more simple. Recall that we had two constructors for the class. The first constructor didn't specify the weights and biases of the layer, so they're still `null`! We'll supply the user of our class with three methods to initialize a network layer:

1. Not-custom Gaussian distribution.
2. Custom Gaussian distribution (given an average and deviation parameter).
3. Random/uniform distribution.

```
  Layer fillGaussian(int size, int numVectors) {
    vectors = new Vector[numVectors];
    for (int i = 0; i < numVectors; i++) {
      vectors[i] = new Vector(size).fillGaussian();
    }

    bias = new Vector(numVectors).fillGaussian();
    initDeltaArray();

    return this;
  }

  void fillGaussian(double average, double deviation) {
    Vector[] replaceVect = new Vector[vectors.length];

    for (int i = 0; i < replaceVect.length; i++) {
```

```
      replaceVect[i] = new Vector(this.vectors[I].length())
                           .fillGaussian(average, deviation);
    }

    bias = new Vector(vectors.length)
               .fillGaussian(average, deviation);

    this.vectors = replaceVect;
    initDeltaArray();
  }

  void fillRandom(int size, int numVectors) {
    this.vectors = new Vector[numVectors];
    for (int i = 0; i < numVectors; i++) {
      vectors[i] = new Vector(size).fillRandom();
    }

    bias = new Vector(numVectors).fillRandom();
    initDeltaArray();
  }
```

That's it for our Layer class! You may notice that some methods or fields aren't strongly encapsulated, which is because they will be used by other classes in other chapters. Now, we'll define a final class called Network, which has two constructors:

```
import java.io.*;
import java.nio.file.Files;
import java.nio.file.Path;

public class Network {
  private Layer[] layers;

  public Network(Vector config,
                 ActivationFunction[] activationFunctions,
                 Error errorType) {
    if ((config.length() - 1) != activationFunctions.length) {
      throw new IllegalArgumentException("Network(Vector," +
       " ActivationFunction[], Error errorType): Vector" +
       " length must be one more than ActivationFunction[]" +
       " length.");
```

```
    }

    layers = new Layer[activationFunctions.length];

    for (int i = 0; i < config.length() - 1; i++) {
      if (i == layers.length - 1) {
        layers[i] = new Layer(true, activationFunctions[i],
                              errorType)
                    .fillGaussian((int) config.get(i),
                                  (int) config.get(i + 1));
      } else {
        layers[i] = new Layer(false, activationFunctions[i],
                              errorType)
                    .fillGaussian((int) config.get(i),
                                  (int) config.get(i + 1));
      }
    }
  }
```

This constructor initializes a neural network that is of the form $x - x - \ldots - x$, where these values are stored in a vector object. Each layer's activation function and error function (one for the entire network) are also specified. If the number of layers isn't equal to the number of specified activation functions, an error is thrown. The default initialization is the non-custom gaussian distribution, although the user can specify another initialization method with:

```
  public Network fromCustomGaussianDistribution(
                           double average, double deviation) {
    for (Layer layer : layers) {
      layer.fillGaussian(average, deviation);
    }

    return this;
  }

  public Network fromUniformDistribution() {
    for (Layer layer : layers) {
      layer.fillRandom(layer.vectors[0].length(),
```

```
                        layer.length());
    }

    return this;
  }
```

The second constructor allows users to initialize a network with a pre-trained network's parameter values that are stored in a file. Here is the method for storing networks in files:

```
  public void export(String name) {
    System.out.printf("Exporting to %s.\n", name);

    double[][][] weights = new double[layers.length][][];
    double[][] bias = new double[layers.length][];
    ActivationFunction[] activationFunctions = new
ActivationFunction[layers.length];
    Error type = layers[layers.length - 1].getErrorType();
    for (int i = 0; i < layers.length; i++) {
      double[][] layerWeights = new double
                                    [layers[i].length()][];
      for (int j = 0; j < layerWeights.length; j++) {
        layerWeights[j] = layers[i].vectors[j].values();
      }
      activationFunctions[i] = layers[i]
                               .getActivationFunction();
      weights[i] = layerWeights;
      bias[i] = layers[i].getBias().values();
    }

    Object[] objects = {
                weights, bias, activationFunctions, type
    };

    try (ObjectOutputStream oos = new ObjectOutputStream(new
BufferedOutputStream(new FileOutputStream(name)))) {
      oos.writeObject(objects);
    } catch (IOException ex) {
      System.err.printf("Cannot export to %s.\n", name);
    }
  }
```

Basically, the method stores each layer's weights, biases, error function and activation function in an object array, and then uses Java's serialization framework to write the object into a space-efficient file. The constructor basically just performs the opposite of the export function. We also warn users that the network they are using is imported from a file:

```
public Network(String filepath) {
  if (!Files.exists(Path.of(filepath))) {
    throw new IllegalArgumentException("Network(file)" +
     ": Please input a valid filepath to the " +
     "exported network.");
  }

  try (ObjectInputStream ois =
       new ObjectInputStream(
       new BufferedInputStream(
       new FileInputStream(filepath)))) {
    Object[] objects = (Object[]) ois.readObject();

    if (objects.length < 4) {
      throw new IllegalArgumentException(
             "Network(file): Cannot import saved network.");
    }

    double[][][] weights = (double[][][]) objects[0];
    double[][] bias = (double[][]) objects[1];
    ActivationFunction[] activationFunctions =
                         (ActivationFunction[]) objects[2];
    Error type = (Error) objects[3];

    layers = new Layer[weights.length];

    boolean isOutputLayer = false;
    for (int i = 0; i < weights.length; i++) {
      Vector[] vectors = new Vector[weights[i].length];
      for (int j = 0; j < weights[i].length; j++) {
        vectors[j] = new Vector(weights[i][j]);
      }
      if (i == weights.length - 1) {
        isOutputLayer = true;
```

```
        }

        Layer matrix = new Layer(vectors, new Vector(bias[i]),
                                 isOutputLayer,
                                 activationFunctions[I],
                                 type);
        layers[i] = matrix;
      }
    } catch (IOException | ClassNotFoundException |
             ClassCastException e) {
      System.err.println(
     "An error occurred when importing an exported network.");
      System.err.println(e);
    }
    System.out.println(
                  "Note: You are using an exported network.");
  }
```

Some exceptions are thrown if the file import was unsuccessful. The more important part of the Network class is its training method. The method receives an array of all training examples' data and their target outputs, the number of training epochs, learning rate, and batch size as parameters, and helps the user train a network with the configurations. If the number of input examples and the number of targeted/desired outputs aren't the same, then we throw an exception. We also ensure that the batch size is positive and can evenly divide the training data. Then, we set the batch size for each layer.

```
  public void train(double[][] input, double[][] target,
                    int epoch, double alpha, int BATCH_SIZE) {
    if (input.length != target.length) {
      throw new IllegalArgumentException("train(): Input" +
       " and target (expected) indices must have the same" +
       " number of examples.");
    }

    if (BATCH_SIZE <= 0) {
      throw new IllegalArgumentException(
                 "train(): Batch size must be positive.");
```

```
    } else if (input.length % BATCH_SIZE != 0) {
      throw new IllegalArgumentException(
              "train(): Batch size must evenly divide data.");
    }

    for (Layer layer : layers) {
      layer.setBatchSize(BATCH_SIZE);
    }
```

In the next block, we estimate the amount of time that training will take by making our network learn 1000 examples and getting their average time. We multiply this average time by the total number of examples and epochs to get the predicted time of computation. To make our network learn, we simply forward-pass through our network by feeding the input to the first layer, getting the first layer's output to be the input to the second layer… and so on. Then, we reverse the order and start with the last layer of the network and make it calculate the error with respect to the ideal output. Afterwards, we back-propagate through our layers until we finish updating our first layer.

```
train()
...
    {
      long start = System.currentTimeMillis();
      for (int i = 0; i < 1000; i++) {
        layers[0].feed(new Vector(input[i]));
        int size = layers.length;
        for (int indice = 1; indice < size; indice++) {
          layers[indice].feed(layers[indice - 1].getOutput());
        }

        layers[layers.length - 1].learn(
                          new Vector(target[i]), null, alpha);

        for (int indice = 1; indice < size; indice++) {
          layers[layers.length - indice - 1].learn(
                  null, layers[layers.length - indice], alpha);
        }
      }
```

```
      double time = (System.currentTimeMillis() - start);
      time *= input.length * epoch;
      time /= 1000.0;
      time /= 1000.0;
      if ((time / 60.0) > 60.0) {
        System.out.println("Training will take approximately "
                    + Math.round(time / 3600.0) + " hours.");
      } else {
        System.out.println("Training will take approximately "
                    + Math.round(time / 60.0) + " minutes.");
      }
    }
```

Although calculating this predicted time of training does update our neural network, it won't really impact the performance of our network, so it is fine. If such performance risks are apparent, then you can comment out or remove the block of code. The next block is the actual training:

```
train()
...
    for (int iter = 1; iter <= epoch; iter++) {
      for (Layer layer : layers) {
        layer.resetDisplayError();
      }
      long start = System.currentTimeMillis();
      for (int index = 0; index < input.length; index++) {
        layers[0].feed(new Vector(input[index]));
        for (int indice = 1; indice < layers.length; indice++)
        {
          layers[indice].feed(layers[indice - 1].getOutput());
        }

        layers[layers.length - 1].learn(
                      new Vector(target[index]), null, alpha);

        for (int indice = 1; indice < layers.length; indice++)
        {
          layers[layers.length - indice - 1].learn(
                  null, layers[layers.length - indice], alpha);
        }
      }
```

```
      if (Double.isNaN(layers[layers.length - 1]
                .getDisplayError().total())) {
        System.err.println("Error exploded, producing NaN" +
        " (not a number). Program terminated automatically.");
        System.exit(-1);
      }

      if (epoch < 100 || iter % 50 == 0) {
        System.out.println("Epoch: " + iter + " Error: " +
         layers[layers.length - 1].getDisplayError().total() /
         input.length + " Time: " +
         ((System.currentTimeMillis() - start) / 1000.0) +
         " seconds.");
      }
    }
  }
```

Notice that if the error explodes (i.e., gets too big), we report the issue and exit (as there is no point training with an error of value 'NaN'). We'll also define a convenience function in Network for users to test the network on a set of inputs and get the output:

```
  public Vector test(double[] input) {
    Vector out = layers[0].test(new Vector(input));
    for (int index = 1; index < layers.length; index++) {
      out = layers[index].test(out);
    }
    return out;
  }
```

Finally, for debugging and general information logging, we'll provide an overridden `toString()` method that prints information like layer configuration and the number of trainable parameters (i.e., weights and biases).

```
  @Override
  public String toString() {
    int trainableParams = 0;
    for (Layer layer : layers) {
      trainableParams += (layer.vectors[0].length() + 1) *
```

```
                        (layer.length());
    }
    StringBuilder out = new StringBuilder();
    out.append("Network with ").append(layers.length)
       .append(" layers and ")
       .append(trainableParams)
       .append(" trainable parameters.\n");
    out.append("Size: {");
    out.append(layers[0].vectors[0].length()).append(", ");

    for (int i = 0; i < layers.length; i++) {
      out.append(layers[i].length());
      if (i != layers.length - 1) {
        out.append(", ");
      } else {
        out.append("}");
      }
    }
    for (int i = 0; i < layers.length; i++) {
      out.append("\n\tLayer ").append(i + 1).append(":\n");

      out.append("\t\tSize: ")
         .append(layers[I].vectors[0].length())
         .append("-").append(layers[i].length());
      out.append("\n\t\tActivation Function: ")
         .append(layers[i].getActivationFunction());

      out.append("\n\t\tError: ")
         .append(layers[i].getErrorType());
    }
    return out.toString();
  }
}
```

That's it! See if you can find a way to use our network object to learn on the MNIST training dataset! If you're stuck, here is a possible implementation. In this implementation a 784-30-10 network with sigmoid activations and mean-squared error is created. Then, the network is trained for 5 epochs with a batch size of 1 and learning rate of 0.01:

```
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.util.Arrays;

public class Driver {
  public static void main(String[] args)
                throws IOException, ClassNotFoundException {
    Vector config = new Vector(new int[] {
            784, 30, 10
    });

    ActivationFunction[] activationFunctions =
                        new ActivationFunction[] {
            ActivationFunction.SIGMOID,
            ActivationFunction.SIGMOID,
    };

    Network network = new Network(config, activationFunctions,
                                  Error.MEAN_SQUARED);

    ObjectInputStream ois = new ObjectInputStream(
                            new FileInputStream("input.ser"));
    ObjectInputStream oiss = new ObjectInputStream(
                           new FileInputStream("target.ser"));
    double[][] inputs = (double[][]) ois.readObject();
    inputs = Arrays.copyOfRange(inputs, 0, 60000);
    double[][] targets = (double[][]) oiss.readObject();
    targets = Arrays.copyOfRange(targets, 0, 60000);

    network.train(inputs, targets, 5, 0.01, 1);
    network.export("network.ser");
    NetworkTest.main(null);
  }
}
```

We also run another program `NetworkTest`, which tests our network on the training and testing datasets and prints the accuracies:

```
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
```

```
public class NetworkTest {
  public static void main(String[] args) throws IOException,
                                        ClassNotFoundException {
    Network network = new Network("network.ser");

    ObjectInputStream ois = new ObjectInputStream(
                            new FileInputStream("input.ser"));
    ObjectInputStream oiss = new ObjectInputStream(
                        new FileInputStream("target.ser"));

    double[][] inputs = (double[][]) ois.readObject();
    double[][] targets = (double[][]) oiss.readObject();

    int corrects = 0;
    for (int i = 0; i < 60000; i++) {
      int x = getMax(network.test(inputs[i]));
      int actual = -1;
      for (int j = 0; j < targets[i].length; j++) {
        if (targets[i][j] == 1) {
          actual = j;
          break;
        }
      }
      corrects += x == actual ? 1 : 0;
    }
    System.out.println("Training: " + corrects + " / " +
                        60000 + " correct.");
    double accuracy = 100.0 * (double) corrects / 60000.0;
    System.out.println("Accuracy: " +
                       String.format("%.2f", accuracy) + "%");

    corrects = 0;
    for (int i = 60000; i < 70000; i++) {
      int x = getMax(network.test(inputs[i]));
      int actual = -1;
      for (int j = 0; j < targets[i].length; j++) {
        if (targets[i][j] == 1) {
          actual = j;
          break;
        }
      }
```

```
      corrects += x == actual ? 1 : 0;
    }
    System.out.println("Testing: " + corrects + " / " +
                       10000 + " correct.");
    accuracy = 100.0 * (double) corrects / 10000.0;
    System.out.println("Accuracy: " +
                       String.format("%.2f", accuracy) + "%");
  }

  private static int getMax(Vector out) {
    int index = 0;
    double val = Double.NEGATIVE_INFINITY;
    for (int i = 0; i < out.length(); i++) {
      if (out.get(i) > val) {
        val = out.get(i);
        index = i;
      }
    }
    return index;
  }
}
```

If you had any issue with programming these snippets of code, ping me an email or file an issue on Github, and I'll try to help resolve issues. In any case, here is the output of an experimental run of the `Driver` class:

```
Training will take approximately 2 minutes.
Epoch: 1 Error: 5.583719879382241 Time: 10.155 seconds.
Epoch: 2 Error: 3.50310763800316 Time: 10.3 seconds.
Epoch: 3 Error: 2.8034668058664343 Time: 10.131 seconds.
Epoch: 4 Error: 2.552442266053705 Time: 10.404 seconds.
Epoch: 5 Error: 2.3540135904076043 Time: 9.807 seconds.
Exporting to network.ser.
Note: You are using an exported network.
Training: 48529 / 60000 correct.
Accuracy: 80.88%
Testing: 8125 / 10000 correct.
Accuracy: 81.25%
```

As a sort of "food for thought" extension, here is another run, but with one slight modification:

```
Network network = new Network(config, activationFunctions,
                                Error.MEAN_SQUARED)
                    .fromCustomGaussianDistribution(0.0, 0.1);
```

In this snippet, we create our network initialized from a custom gaussian distribution of average 0 and deviation 0.1. The output after running `Driver` again is:

```
Training will take approximately 2 minutes.
Epoch: 1 Error: 3.061087367359798 Time: 10.318 seconds.
Epoch: 2 Error: 1.3980115498301675 Time: 9.911 seconds.
Epoch: 3 Error: 1.1846242010837236 Time: 10.779 seconds.
Epoch: 4 Error: 1.0450447302536288 Time: 10.273 seconds.
Epoch: 5 Error: 0.9521872192447736 Time: 9.833 seconds.
Exporting to network.ser.
Note: You are using an exported network.
Training: 56551 / 60000 correct.
Accuracy: 94.25%
Testing: 9407 / 10000 correct.
Accuracy: 94.07%
```

Make no mistake, this is not something that happened by chance! The testing and training accuracy of the second run was approximately 14% more accurate than the first run, but why? Let's first understand how our network was originally initialized. Originally, our network was initialized with normal gaussian distribution (which in Java's implementation, is gaussian distribution with average 0 and deviation 1) that ranges from -1 to 1. By modifying our distribution range to between -0.1 and 0.1, we reduced the overall range of our weight and bias initialization. Below is a plot of several different gaussian distributions:

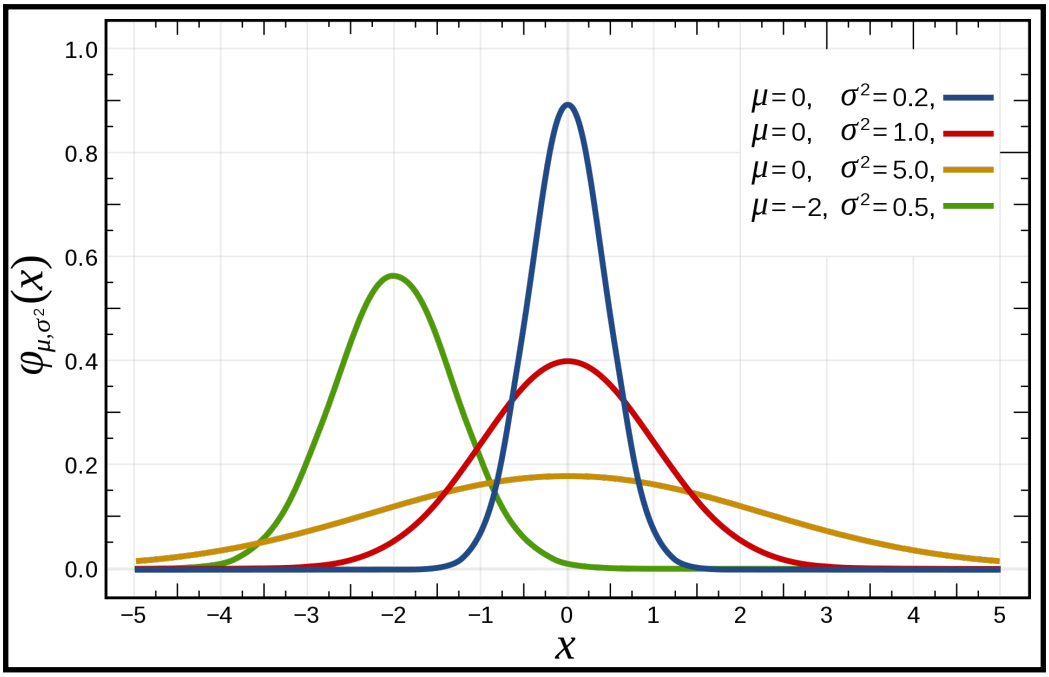


Figure 8-4: Several different gaussian distributions.

Notice that the y-axis represents the probability of getting the x-axis number for the given gaussian distribution. In the graph, the blue curve represents gaussian distribution with average 0 and deviation 0.2, and the red curve represents gaussian distribution with average 0 and deviation 1.0. Notice that drawing a 0 in the red curve is less likely than drawing a 0 from the blue curve. Also, notice that if more weights are initialized at 0, the input to a neuron in the next layer would be $\sigma(0) = 0.5$. To adjust our weights, recall that we multiply by the derivative of the sigmoid activation function with respect to the weighted sum:

$$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$$
$$\sigma'(0) = 0.5 \cdot (1 - 0.5) = 0.25$$

Then notice that the derivative of the sigmoid activation function peaks at 0. This means that if the weighted sum of the product of weights and inputs is

closer to 0, the weights will be adjusted more. Now, let's assume that we have a single layer network that gets 100 inputs that are approximately 50% zeros and 50% ones. Notice that we'll then only have 50 weight and input products (the rest are 0). Therefore, the sum of the product of the weights with the inputs will have an average of 0, but deviation of $\sqrt{50}$ (as summing gaussian distributions is equal to summing their averages and deviations respectively). Notice then that there is a strong likelihood that the summed result be between the range -5 to 5. If we applied the derivative of sigmoid to -5 and 5, we get: $\sigma'(\pm 5) \approx 0.0066$. Notice then that these weights will be adjusted more slowly, and therefore affect our training performance. On the other hand, our gaussian distribution with deviation of 0.1 would achieve better results as the weight and input products' sums will have deviation of $\sqrt{5}$, which concentrates the weighted sum between a much smaller range of approximately -2 to 2. This helps improve our training as the derivative of these two bounds is most obviously larger than that of ± 5 (and will result in larger weight updates).

You may also want to know a little more about our cross-entropy cost function. The reason why it is sometimes better for training networks is because both cross-entropy cost functions (for categorical and binary classification) remove the effects of the derivative of the activation function. For example, the derivative of binary cross-entropy is $\frac{o - t}{o(1 - o)}$, so when paired with sigmoid, the $\sigma(z)(1 - \sigma(z))$ term cancels out $o(1 - o)$ (as $o = \sigma(z)$). As for categorical cross-entropy, it is only suitable in companion with the Softmax activation function, which has fewer terms whose derivative is $o(1 - o)$, so its convergence (how quickly a network can reach an optimal classification/regression accuracy) is usually faster than MSE.

For now, that's all! Try to reread this chapter or review the chapter summary below.

Chapter Summary

1. *Backpropagation* is a generalized version of the delta rule.

2. Backpropagation rule updates for weights and biases:

$$\Delta w_{i,j,L} = \alpha \cdot 2(o_j - t_j)x_i\sigma'(z)$$

$$\Delta w_{i,j,L-1} = \alpha \cdot \sigma'(z_{j,L-1}) \cdot x_{i,L-1} \cdot [\sum_{k=1}^{j} 2(o_{k,L} - t_k) \cdot \sigma'(z_{k,L}) \cdot w_{i,k,L}]$$

$$w_{i,j,l} = w_{i,j,l} - \Delta w_{i,j,l}$$

$$\Delta b_{j,L} = \alpha \cdot 2(o_j - t_j)\sigma'(z_{j,L})$$

$$\Delta b_{j,L-1} = \alpha \cdot \sigma'(z_{j,L-1}) \cdot [\sum_{k=1}^{j} 2(o_{k,L} - t_k) \cdot \sigma'(z_{k,L}) \cdot w_{i,k,L}]$$

$$b_{j,l} = b_{j,l} - \Delta b_{j,l}$$

Note that $2(o_j - t_j)$ is the derivative of the Mean Squared Error. To use a different error function, simply replace the derivative with that of the desired error function's derivative.

3. ELU stands for Exponential Linear Unit, and is a variation of RELU (and Leaky-RELU), taking the form: $\sigma(x) = \begin{cases} m(e^x - 1), & x \leq 0 \\ x, & x > 0 \end{cases}$, where m is a constant (usually 1). The derivative of ELU is: $\sigma'(x) = \begin{cases} e^x, & x < 0 \\ 1, & x \geq 0 \end{cases}$ (for $m = 1$).

4. The Softmax activation function takes the form: $\sigma(x)_i = \frac{x_i}{\sum_{j=1} x_j}$. The sum of its outputs adds to 1, so it can be used to express network predictions as probabilities. Softmax is usually used with another type of error function called *cross entropy*. Softmax's derivative is: $\sigma'(x)_i = \sigma(x_i)(\delta_{i,j} - \sigma(x_j))$ for i being each index in the output vector and j being the index of the neuron whose output is currently being used for training. Here, $\delta_{i,j}$ is the Kronecker delta, and represents: $\delta_{i,j} = \{^{0,\ i \neq j}_{1,\ i = j}$.

5. The categorical cross-entropy loss function takes the form: $E(t, o) = -t \cdot \ln(o)$ (with derivative $(o - t)$ when paired with Softmax), and is designed for use in categorical classification tasks (tasks where networks classify data that belongs to multiple classes).

6. Binary cross-entropy takes the form: $E(t, o) = -(t \cdot \ln(o) + (1 - t) \cdot \ln(1 - o))$ (with derivative $\frac{o - t}{o(1 - o)}$), and is used for binary classification tasks (tasks where networks classify data into one of two classes/labels).

7. *Convergence* is the notion of network training *converging* (reaching) to a *minimal loss*.

8. Initializing weights and biases with gaussian distribution average 0 and deviation close to zero can improve convergence rates.

Chapter 9 - Optimizers

"A friend may be waiting behind a stranger's face."

— Letter to My Daughter by Maya Angelou

START HERE

Chapter 10 - Generative Adversarial Networks

"We all require devotion to something more than ourselves for our lives to be endurable."
— Being Mortal by Atul Gawande

Task 2 - Object Recognition

"A goal without a plan is just a wish."

— Antoine de Saint-Exupéry

https://www.kaggle.com/c/cifar-10

Chapter 11 - L1 & L2 Regularization

"We all require devotion to something more than ourselves for our lives to be endurable."
— Being Mortal by Atul Gawande

START HERE

Chapter 12 - Convolutional Neural Networks

"Even the darkest night will end and the sun will rise."

— Les Misérables by Victor Hugo

START HERE

Chapter 13 - TITLE

"QUOTE"

— TITLE, AUTHOR

START HERE

Chapter 14 - TITLE

“QUOTE”

— TITLE, AUTHOR

START HERE

Chapter 20 - Conclusion

"Each of us is more than the worst thing we've ever done."

— Just Mercy by Bryan Stevenson

START HERE

Chapter Summaries

Further Reading & Resources

Fast AI fast.ai, a comprehensive course on machine learning

Neural Networks and Deep Learning, by Michael Nielsen

Neural Networks, a playlist of videos by 3Blue1Brown

Deep Learning, a book by Ian Goodfellow, Yoshua Bengio, and Aaron Courville

Machine Learning Glossary, for more accurate and precise definitions of Machine Learning terms.

Wikipedia, although not suitable for formal writings like this one, you can use this as a source of knowledge for almost all topics!

Khan Academy, for a more intuitive understanding of various mathematical terms, symbols, and topics used in this book including:

- Derivatives
- Chain Rule
- Summations
- Piecewise Functions
- Maxima and minima of a function

- Euler's number *e*
- Slope formula
- Limits

You can directly search on the webpage, and learning with the website is free, requiring no sign-ups, although the website does recommend signing up to save progress.

Fundamentals of Calculus, an excellent book on calculus by Benjamin Crowell, Joel Robbin and Sigurd Angenent.

MNIST Dataset, if you haven't visited it yet :), you can find the papers published by the group there as well.

Google Dataset Search, search for a dataset to start your own neural network project!

CSS Gradient, a tool I use to create gradients.

Desmos Calculator, another tool I used throughout the book to make original graphs and convert them to pictures.

Machine Learning News, a collection of links to various websites. It's constantly updated!

Machine Learning From Scratch, a collection of articles on machine learning.

Mathematics for Machine Learning, a book about machine learning by Marc Peter Deisenroth, A. Aldo Faisal, and Cheng Soon Ong.

Augmentor, a library for data augmentation in Python.

Machine Learning, a series of articles by Tobias Hill that walks through the creation of a digit recognizer in Java.

Grokking Deep Learning, a book by Andrew Trask that introduces deep learning techniques very simply and comprehensibly.

Data Preprocessing Collection on Roboflow's blog.

Data Augmentation Collection on Roboflow's blog.

CS224n Stanford Course

CS229n Stanford Course

CS231n Stanford Course

The Nature of Code, a book by Daniel Shiffman that has a chapter on neural networks. The book also explores natural simulation and is an excellent resource.

Neptune AI's blog, which provides some handy articles on machine learning.

Reinforcement Learning: An Introduction, a book introducing reinforcement learning.

Speech and Language Processing, a book about NLP (Natural Language Processing).

The Learning Machine, a website with articles on machine-learning related topics.

DeepLearning.AI Notes, a website providing informational articles on machine-learning topics.

Bayesian Optimization, an article outlining a technique to initializing hyperparameters.

Practical Recommendations for Gradient-Based Training of Deep Architectures, a document containing insightful information and tips on training deep neural networks (and other architectures).

The effects of weight initialization on neural nets, an article describing how different initialization methods may affect the performance of a network.

"Build Your Own Automatic Differentiation Program", an article implementing auto-differentiation from scratch.

References

Al-Masri, Anas. "What Are Overfitting and Underfitting in Machine Learning?" Towards Data Science, Medium, 22 June 2019, https://towardsdatascience.com/what-are-overfitting-and-underfitting-in-machine-learning-a96b30864690. Accessed 8 December 2020.

"Analog Image Processing vs. Digital Image Processing." JavaTpoint, JavaTpoint, www.javatpoint.com/analog-image-processing-vs-digital-image-processing.

Antipov, Grigory, et al. "Face Aging with Conditional Generative Adversarial Networks." ArXiv.org, 30 May 2017, arxiv.org/abs/1702.01983.

Aquegg. "Spectrogram." Wikimedia Commons, Wikipedia, 21 Dec. 2008, commons.wikimedia.org/wiki/File:Spectrogram-19thC.png.

Brownlee, Jason. "How to Develop a Conditional Gan (CGAN) from Scratch." Machine Learning Mastery, 1 Sept. 2020, machinelearningmastery.com/how-to-develop-a-conditional-generative-adversarial-network-from-scratch/.

Brownlee, Jason. "How to Develop a GAN for Generating MNIST Handwritten Digits." Machine Learning Mastery, 1 Sept. 2020, machinelearningmastery.com/how-to-develop-a-generative-adversarial-network-for-an-mnist-handwritten-digits-from-scratch-in-keras/.

Brownlee, Jason. "What is the Difference Between Test and Validation Datasets?" Machine Learning Mastery, 14 July 2017, https://machinelearningmastery.com/difference-test-validation-datasets/. Accessed 27 December 2020.

Cab02. "A Pile of oranges." Wikipedia, 10 July 2006, https://en.wikipedia.org/wiki/File:Pile_of_Oranges.jpg. Accessed 14 November 2020.

Canuma, Prince. "Image Pre-Processing." Medium, Towards Data Science, 28 Aug. 2020, towardsdatascience.com/image-pre-processing-c1aec0be3edf.

Cousineau, Denis. "Outliers Detection and Treatment: A Review - Scientific Figure on ResearchGate." ResearchGate, June 2010, www.researchgate.net/figure/Examples-of-various-outliers-found-in-regression-analysis-Case-1-is-an-outlier-with_fig2_50946372.

Crowell, Benjamin, et al. Fundamentals of Calculus. Lulu Press, Inc., 2015.

Dake, and Mysid. "A Simplified View of an Artifical Neural Network." Wikimedia Commons, Wikipedia, 28 Nov. 2006, commons.wikimedia.org/wiki/Artificial_neural_network#/media/File:Neural_network.svg.

"Dataset." Cambridge Advanced Learner's Dictionary & Thesaurus, Cambridge University Press, 20 April 2013, https://dictionary.cambridge.org/dictionary/english/dataset. Accessed 27 December 2020.

Dawkins, Paul. Paul's Online Math Notes, 2003, tutorial.math.lamar.edu/.

The Editors of Encyclopaedia Britannica. "Derivative." Encyclopædia Britannica, 22 September 2017, https://www.britannica.com/science/derivative-mathematics. Accessed 7 November 2020.

Elgendy, Mohamed. Deep Learning for Vision Systems. Manning Publications, 2020. Manning Publications, https://www.manning.com/books/deep-learning-for-vision-systems#toc. Accessed 24 January 2021.

Fast AI. "Lesson 1: Practical Deep Learning for Coders." Fast AI, 22 August 2020, https://course.fast.ai/videos/?lesson=1. Accessed 26 October 2020.

"File:Artificial neural network.svg." Wikimedia Commons, the free media repository., 19 October 2020, https://commons.wikimedia.org/w/index.php?title=File:Artificial_neural_network.svg&oldid=494529927. Accessed 26 October 2020.

"Generative Adversarial Network." Wikipedia, Wikimedia Foundation, 8 Aug. 2021, en.m.wikipedia.org/wiki/Generative_adversarial_network.

Géron, Aurélien. Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow. 2nd ed., Canada, O'Reilly, 2019.

Glosser.ca. "Normal Distribution PDF." Wikimedia Commons, 29 Apr. 2016, commons.wikimedia.org/wiki/File:Normal_Distribution_PDF.svg.

Hill, Tobias. "Extra 1 - Data Augmentation." Machine Learning, 2019, machinelearning.tobiashill.se/.

Hu, Evan. "Using Conditional Deep Convolutional Gans to Generate Custom Faces from Text Descriptions." Medium, Towards Data Science, 18 June 2021, towardsdatascience.com/using-conditional-deep-convolutional-gans-to-generate-custom-faces-from-text-descriptions-e18cc7b8821.

Jenkner, Patrycja. "The Essential Guide to Data Augmentation in NLP." Hacker Noon, Hackernoon, 5 Oct. 2020, hackernoon.com/the-essential-guide-to-data-augmentation-in-nlp-9n3l3tbt.

Kamps, Haje Jan. "Everything You Need to Know about Camera Lenses." Medium, Photography Secrets, 22 Aug. 2019, medium.com/photography-secrets/lenses-e033d2f77548.

Kernes, Jonathan. "Build Your Own Automatic Differentiation Program." Medium, Towards Data Science, 15 Feb. 2021, towardsdatascience.com/build-your-own-automatic-differentiation-program-6ecd585eec2a.

Khan Academy. "Normal Distributions Review." Khan Academy, https://www.khanacademy.org/math/statistics-probability/modeling-distributions-of-data/normal-distributions-library/a/normal-distributions-review. Accessed 31 October 2020.

Kurbiel, Thomas. "Derivative of the Softmax Function and the Categorical Cross-Entropy Loss." Medium, Towards Data Science, 22 Apr. 2021, towardsdatascience.com/derivative-of-the-softmax-function-and-the-categorical-cross-entropy-loss-ffceefc081d1.

Lao, Randy. "A Beginner's Guide to Machine Learning." Medium, 23 January 2018, https://medium.com/@randylaosat/a-beginners-guide-to-machine-learning-dfadc19f6caf. Accessed 26 October 2020.

"Loss Functions." Loss Functions - ML Glossary Documentation, 2017, ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html.

Loy, James. "Fundamentals of Generative Adversarial Networks." Medium, Towards Data Science, 3 Nov. 2020, towardsdatascience.com/fundamentals-of-generative-adversarial-networks-b7ca8c34f0bc.

"Machine Learning Glossary." Machine Learning Glossary - ML Glossary Documentation, 2017, ml-cheatsheet.readthedocs.io/en/latest/.

Martinez, Juan Cruz. "7 Steps of Machine Learning." Live Code Stream, 2 June 2020, https://livecodestream.dev/post/2020-06-02-7-steps-of-machine-learning/. Accessed 26 October 2020.

Mirza, Mehdi, and Simon Osindero. "Conditional Generative Adversarial Nets." ArXiv.org, 6 Nov. 2014, arxiv.org/abs/1411.1784.

ML4A. "Looking inside neural nets." Machine Learning for Artists, https://ml4a.github.io/ml4a/looking_inside_neural_nets/. Accessed 14 November 2020.

Nelson, Joseph. "When Should I Use Grayscale as a Pre-Processing Step?" Roboflow Blog, Roboflow Blog, 20 July 2020, blog.roboflow.com/when-to-use-grayscale-as-a-preprocessing-step/.

Nelson, Joseph. "Why Should I Do Pre-Processing and Augmentation on My Computer Vision Datasets?" Roboflow Blog, Roboflow Blog, 28 Dec. 2020, blog.roboflow.com/why-preprocess-augment/.

Nelson, Joseph. "Why to Add Noise to Images for Machine Learning." Roboflow Blog, Roboflow Blog, 5 Oct. 2020, blog.roboflow.com/why-to-add-noise-to-images-for-machine-learning/.

Nicolas, Sarah. "30 Short Inspirational Quotes From Books." Book Riot, 18 July 2020, https://bookriot.com/short-inspirational-quotes/. Accessed 8 December 2020.

Pierce, Rod. "Introduction to Derivatives." Math Is Fun, 18 December 2017, http://www.mathsisfun.com/calculus/derivatives-introduction.html. Accessed 7 November 2020.

Pokhrel, Sabina. "Feature Preprocessing for Numerical Data - The Most Important Step." Medium, Analytics Vidhya, 14 Sept. 2019, medium.com/analytics-vidhya/feature-preprocessing-for-numerical-data-the-most-important-step-e9ed76151298.

Raschka, Sebastian. "Is the logistic sigmoid function just a rescaled version of the hyberbolic tangent (tanh) function?" Sebastian Raschka, https://sebastianraschka.com/faq/docs/tanh-sigmoid-

relationship.html#:~:text=The%20hyperbolic%20tangent%20(tanh)%20and,11%2Be%E2%88%92x.&text=Since%20the%20logistic%20sigmoid%20function,can%20write%20the%20following%20relationship%3A&text=1%E2%88%9211%2Be%. Accessed 4 November 2020.

Ratner, Alex, et al. "Image Data Augmentation Illustration." Learning to Compose Domain-Specific Transformations for Data Augmentation, Stanford, 30 Aug. 2017, dawn.cs.stanford.edu/2017/08/30/tanda/.

Rojas, Raúl. Neural Networks - A Systematic Introduction. Berlin, New York, Springer-Verlag, 1996. Neural Networks - A Systematic Introduction, http://page.mi.fu-berlin.de/rojas/neural/. Accessed 31 October 2020.

Sadowski, Peter. "Notes on Backpropagation." University of California, www.ics.uci.edu/~pjsadows/notes.pdf.

Sharma, Sagar. "Activation Functions in Neural Networks." Medium, towards data science, 6 September 2017, https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6. Accessed 6 November 2020.

Shiffman, Daniel. The Nature of Code, 2012, natureofcode.com/.

Suran, Abhishek. "Conditional and Controllable Generative Adversarial Networks." Medium, Towards Data Science, 16 Oct. 2020, towardsdatascience.com/conditional-and-controllable-generative-adversarial-networks-a149691ddae6.

"The Backpropagation Algorithm." Neural Networks: A Systematic Introduction, by Rojas Raúl, Springer, 1996, pp. 151–184.

Trask, Andrew W. Grokking Deep Learning. Manning, 2019, www.manning.com/books/grokking-deep-learning.

Varsity Tutors. "Line of Best Fit (Least Square Method)." Varsity Tutors, https://www.varsitytutors.com/hotmath/hotmath_help/topics/line-of-best-fit. Accessed 18 December 2020.

Vibrant Orange. "Orange (fruit) isolated on a white background." Wikipedia, 4 November 2007, https://commons.wikimedia.org/wiki/File:Orange_Fruit_Close-up.jpg. Accessed 14 November 2020.

Wikipedia Contributors. "Analog Signal." Wikipedia, Wikimedia Foundation, 13 July 2021, en.wikipedia.org/wiki/Analog_signal.

Wikipedia Contributors. "Digital Image Processing." Wikipedia, Wikimedia Foundation, 18 July 2021, en.wikipedia.org/wiki/Digital_image_processing.

A Readable Bibliography

I realize that a bibliography is useless to normal readers as it is simply just *unreadable*, therefore, I created a *readable* version of it here:

1. "What Are Overfitting and Underfitting in Machine Learning?", an article written by Anas Al-Masri that was published on the Medium publication Towards Data Science. The article defines overfitting and underfitting in an easily comprehendible way, and also outlines the importance of addressing overfitting and underfitting issues in neural networks (data structures that generalize datasets). At the end, the author also mentions an important issue in neural network architecture called *bias-variance tradeoff*, which is the notion that there is a tradeoff between the *bias* and *variance* of a network.
2. "What is the Difference Between Test and Validation Datasets?", an article by Jason Brownlee that outlines the main differences between test and validation datasets, and defines them respectively. The importance of having validation datasets is also explained thoroughly in the article.
3. Fundamentals of Calculus, a book by Benjamin Crowell, Joel Robbin and Sigurd Angenent that approaches calculus from a more mathematical perspective.