Neural Networks for Electronics Hobbyists

A Non-Technical Project-Based Introduction

Richard McKeon

Apress®

Neural Networks for Electronics Hobbyists

A Non-Technical Project-Based Introduction

Richard McKeon

Apress®

Neural Networks for Electronics Hobbyists: A Non-Technical Project-Based Introduction

Richard McKeon
Prescott, Arizona, USA

ISBN-13 (pbk): 978-1-4842-3506-5 ISBN-13 (electronic): 978-1-4842-3507-2
https://doi.org/10.1007/978-1-4842-3507-2

Library of Congress Control Number: 2018940254

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Natalie Pao
Development Editor: James Markham
Coordinating Editor: Jessica Vakili

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-3506-5. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author

Hi, I'm **Rick McKeon**. I am currently living in beautiful Prescott, Arizona. Since retiring, I have been spending time pursuing my passion for writing, playing music, and teaching. I am currently producing a series of books on music, nature, and science. Some of my other interests include hiking, treasure hunting, recreational mathematics, photography, and experimenting with microcontrollers. Visit my website at www.rickmckeon.com.

About the Technical Reviewer

Chaim Krause is first and foremost a #Geek. Other hashtags used to define him are (in no particular order) #autodidact, #maker, #gamer, #raver, #teacher, #adhd, #edm, #wargamer, #privacy, #liberty, #civilrights, #computers, #developer, #software, #dogs, #cats, #opensource, #technicaleditor, #author, #polymath, #polyglot, #american, #unity3d, #javascript, #smartwatch, #linux, #energydrinks, #midwesterner, #webmaster, #robots, #sciencefiction, #sciencefact, #universityofchicago, #politicalscience, and #bipolar. He can always be contacted at chaim@chaim.com and goes by the Nom de Net of Tinjaw.

Preface

This book is for the layman and the electronics hobbyist who wants to know a little more about neural networks. We start off with an interesting nontechnical introduction to neural networks, and then we construct an electronics project to give you some hands-on experience training a network.

If you have ever tried to read a book about neural networks or even just tried to watch a video on this topic, you know things can get really technical really fast!

Almost immediately, you start to see strange mathematical symbols and computer code that looks like gibberish to most of us! I have degrees in mathematics and electrical engineering. I have taught math, and spent a career designing electronic products. But most of the articles that I start to read in this field just blow me away! Well, that's what we hope to avoid here. My goal is to give you an interesting and fun introduction to this fascinating topic in an easy-to-understand, nontechnical way. If you want to understand neural networks without calculus or differential equations, this is the book for you!

There are no prerequisites. You don't need an engineering degree, and you don't even need to understand high school math in order to understand everything we are going to discuss. In this book, you won't see a single line of computer code.

For this project, we are going to take a hardware-based approach using very simple electronic components. The project we are going to build isn't complicated, but it illustrates how back propagation can be used to adjust connection strengths or "weights" and train a network. We do this manually by adjusting potentiometers in the hidden layer.

This network doesn't learn automatically. We have to intervene with a screwdriver. This is a tutorial for us to learn how adjusting connection strengths between neurons results in a trained network. Now, how much fun is that?

If you like to tinker around with components and build circuits on a breadboard, you're going to love this project! Who knows, if you enjoy this brief introduction, you may want to pursue this amazing subject further!

Neural networks are modeled after biological computers like the human brain. Instead of following a step-by-step set of instructions, a neural network consists of a bunch of "neurons" that act together in parallel—all at once—to produce an output!

So, instead of writing a program like you would for a conventional computer with

1. Fetch an instruction.
2. Execute it.
3. Fetch the next instruction.
4. Execute it . . .

You "train" a neural network by showing it the input data and the correct answer. You do this over and over again (maybe hundreds of times), and leave it up to the network to figure out how to come up with the right answer. During this training process, the network figures out how to solve the problem, even when we don't know how to solve it ourselves. In fact, some of our best algorithms have come from figuring out how the neural network did it.

That may seem impossible, but think about it: many of our best products have been developed by observing nature. The natural world has been around for millions of years, adapting and discovering solutions to all kinds of problems. Modern computers operate so fast that they can simulate thousands of generations of natural adaptation in just a few minutes! As long as we don't make this planet uninhabitable for humans,

we are on the verge of amazing technological discoveries! And the application of neural networks is one of them.

I know it sounds crazy that we could build a machine that does stuff we don't know how to do ourselves, but the fact is that we work really hard to go back and try to figure out how the network did it. It's called "feature extraction." We delve deep into the "hidden layers" for hidden secrets.

This exciting field of study reminds me of the early days of exploration when adventurers traveled in sailing ships to strange and exotic lands to discover hidden mysteries. The age of discovery is not over. With today's technology, it's really just beginning!

Are you getting excited to learn more?

Neural networks are great at pattern recognition and finding answers even when the input data isn't all that great. They can reliably find patterns even when some of the input data is missing or damaged. Also, a neural network can produce amazingly accurate results based on data it has never seen before. In other words, it can "generalize."

That may be hard to believe, but that's what you and I do every day.

How do we do it?

We have a brain!

Your brain is a huge collection of very simple processing elements called "neurons." These neurons are interconnected like crazy. It's hard to imagine how many connections there are! But, no worries, we will see how some of this stuff works even with just a few neurons.

Tips and Tricks When you see text set off like this, I am offering some interesting tips to make your life easier or just a silly comment to lighten things up.

I'm hoping this book will be an interesting and informative introduction to neural networks, but it certainly is not comprehensive.

I'm also hoping this brief introduction will be enjoyable enough that you will want to go on and learn more about this amazing technology!

I am always right here to help you along and answer questions. No question is to simple or too basic, so shoot me an email at rmckeon5@gmail.com.

OK, enough talk. Let's get started!

CHAPTER 1

Biological Neural Networks

"Is there intelligent life in outer space?" OK, that may be a little bit tongue in cheek, but, think about it, maybe it is a valid question. How much biological intelligence is there on earth? Where does it come from? And how much greater can it be? Is it just a matter of "bigger brains" or more complex neural networks inside our skulls?

Are there intelligent networks other than the human brain? How about animal intelligence or even plant intelligence? Many of these nonhuman networks share a surprising amount of DNA with humans. In fact, scientists have sequenced the genome of the chimpanzee and found that we share with them about 94% of the same DNA. Is that amazing, or what?

Think about the following:

1. Dogs can learn to follow voice commands.
2. Gorillas and chimps can learn sign language and use it to communicate.
3. Many birds use tools and can figure out complex ways to get food without being taught.

R. McKeon, *Neural Networks for Electronics Hobbyists*,
https://doi.org/10.1007/978-1-4842-3507-2_1

Amazing Fact Scientists estimate that there are about 44 billion neurons in the human brain and each one of them is connected to thousands of other neurons! See Figure 1-1.

Here are some estimates of the number of neurons for other species:

- Fruit Fly: 100 thousand neurons
- Cockroach: One million neurons
- Mouse: 33 million neurons
- Cat: One billion neurons
- Chimpanzee: Three billion neurons
- Elephant: 23 billion neurons

So, is a neural network all it takes to develop intelligence? Many people say yes.

Modern electronic neural networks are already performing amazing feats of pattern recognition. The near future will almost certainly bring huge advances in this area!

Biological Computing: The Neuron

Here's where it starts. Figure 1-1 is a graphical representation of a nerve cell or "neuron." Your brain has billions of these things—all interconnected! Just because of the sheer number of neurons and how they are interconnected, the study of the human brain gets complicated really fast!

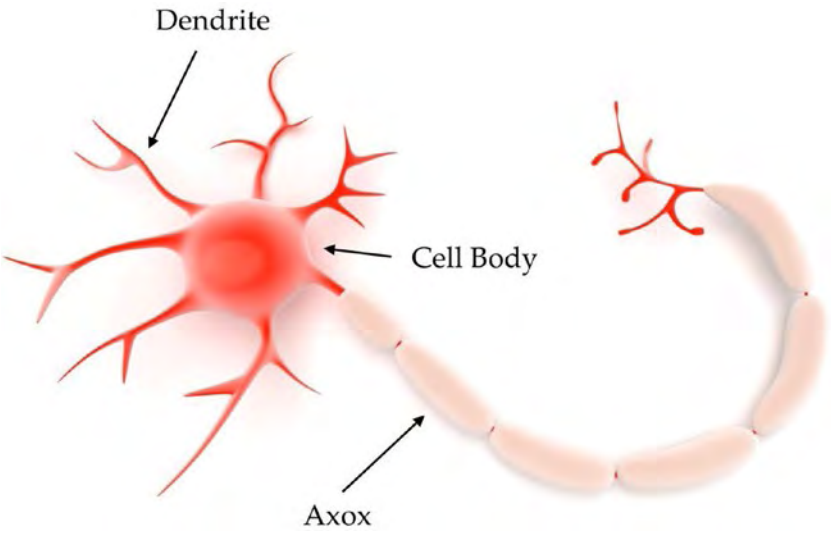


Figure 1-1. *An individual neuron*

The doctors studying the brain scan in Figure 1-2 are probably looking for a specific, identifiable problem like a brain tumor. Even these specialists don't have all the answers about detailed brain function.

Figure 1-2. *Doctors study brain scan*

In recent years, we have made great strides in understanding the structure and electrical activity of the brain, but we still have a long way to go! This is especially so when it comes to concepts like self-awareness and consciousness. Where does that come from? Fortunately, for us to build functioning networks that can accomplish practical tasks, we don't need to have all the answers.

Of course we want to understand every detail of how the brain works, but even if simple and incomplete, today's neural network simulations can do amazing things! Just like you and me, neural networks can perform very well in terms of pattern recognition and prediction even when given partial or corrupted data. You are probably thinking, "This is more like science fiction than science!" Believe me, I'm not making this stuff up.

So, how does a neuron work? Figure 1-3 gives us a few hints. A neuron is a very complex cell, but basically they all operate the same way.

1. The dendrites receive electrical impulses from several other neurons.
2. The cell body adds up all these signals and determines what to do next. If there is enough stimulation, it decides to fire a pulse down its axon.
3. The axon has connections to several other neurons.
4. And "the beat goes on," so to speak.

Of course, I have left out some details, but that's basically how it works. We will talk about "weights," "activation potentials," "transfer functions," and stuff like that later (without getting too technical).

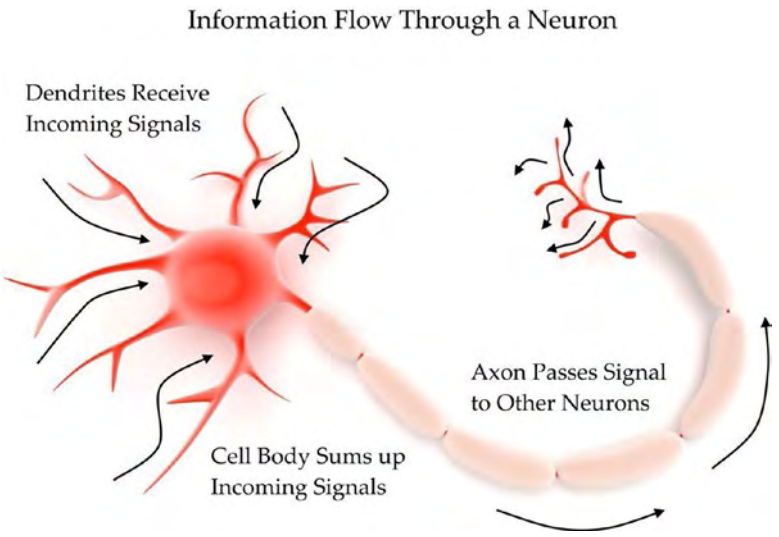


Figure 1-3. *Information flow*

So, if you connect up all these neurons, what does it look like? Well, not exactly like Figure 1-4, but it kind of gives you the idea. Biological computers are highly interconnected.

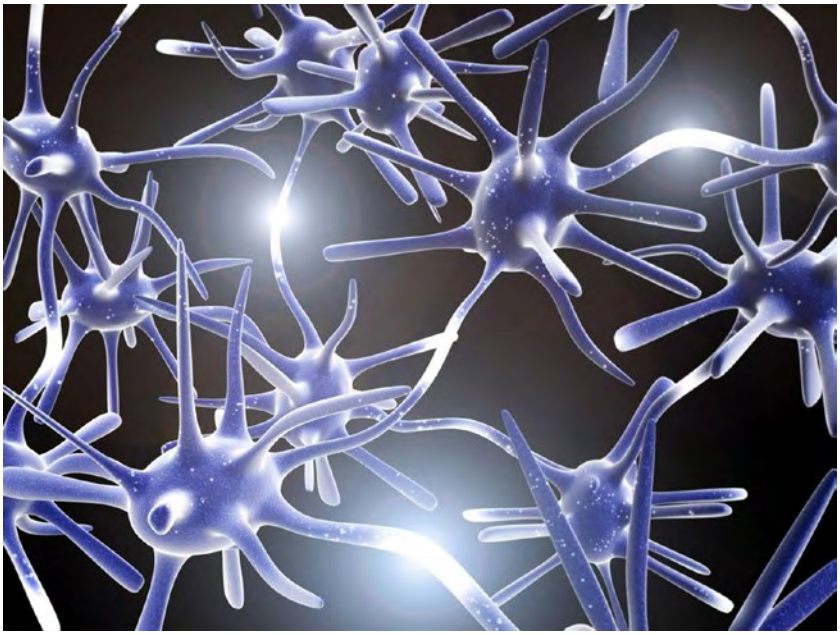


Figure 1-4. *Interconnected neurons*

An interesting thing that is not shown in Figure 1-4 is that the neurons are not directly connected or "hard-wired" to the others. Where these connections take place, there is a little gap called a "synapse." When a neuron fires, it secretes a chemical into the synapse. These chemical messengers are called "neurotransmitters." Depending on the mix of neurotransmitters in the synapse, the target cell will "get the message" either pretty strongly or pretty weakly. What will the target neuron do? It will sum up all these signals and decide whether or not to fire a pulse down its axon.

You can see that we are not exactly talking about electrons flowing along a piece of copper wire. The equivalent thing in neurons is a chemical exchange of ions. Is the concept of "ion exchange" more complicated than electrons flowing in a wire? Not really. We actually don't understand electron flow all that well either; it's just that we're more used to hearing about things that use electricity.

When a person drinks alcohol or takes certain types of drugs, guess what they are affecting. You guessed it! They are affecting the neurotransmitters—the chemistry within the synapse.

When you go to the dentist and get a shot of Novocain to block the pain, what is that Novocain doing? It's preventing neurons from firing by interfering with the chemical processes taking place. So, understanding that brain activity is electrochemical makes this whole discussion a lot more understandable.

Remember when I said we weren't going to get too technical? Well, that's it for this discussion.

Congratulations! You just graduated from the "Rick McKeon School of Brain Chemistry."

Figure 1-5 may not be scientifically accurate, but it is a pretty picture, and it graphically represents what happens in a synapse.

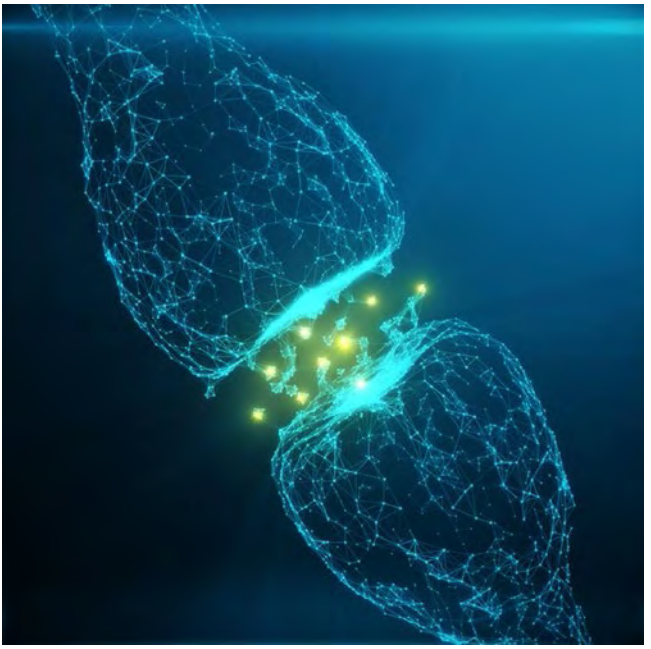


Figure 1-5. *The synapse*

Given all these complex interconnections, something good is bound to emerge, right? Well, it does, and that's what makes this new field of study so exciting!

How can behavior "emerge"? Well, this is another fascinating topic that we are just beginning to understand. Without getting too technical, let me just say that when many individuals interact, an overall behavior can emerge that is more complex than any of the individuals are capable of. How crazy is that? Think about the behavior of an ant colony, a flock of birds, or a school of fish. The individuals use pretty simple rules to interact with each other, but the overall behavior can be pretty complex.

Let me relate an interesting story from my music teacher days.

What Did You Do to Me?

I have taught guitar and banjo students for many years, and I am continually amazed when learning takes place. I'm not amazed that learning takes place, but I am at a loss to explain exactly what has happened. When learning occurs some physical changes have taken place in the brain. By this, I mean actual rewiring of neurons or chemical changes in the synapses! And it happens quickly. That is why the brain is called "plastic."

Teaching banjo is so much fun because stuff like this happens all the time! We may be working on a certain lick or musical phrase and the student just isn't getting it. His timing and accent are way off, and he just can't make it sound musical. We will go over it several times and I'll say, "Make it sound like this." I'll have him sing it rhythmically and say, "Now, make the banjo sing it like that." All of a sudden he can play it perfectly! What the heck?

One time when this happened, the student was just as surprised as I was and asked, "What did you do to me?" Amazing question, and a hard one to answer! Learning has taken place. He couldn't play the lick no matter how hard he tried, but then he could, and he recognized the difference. What happened? Something changed.

I don't know exactly how it works, but learning has taken place, and new neural connections have been formed or synaptic weights have changed. Our brains are changing all the time. Even as we get older, we can still learn new things. The saying that you "can't teach an old dog new tricks" is a folly. We are capable of learning new things until we draw our last breath!

We'll get more into the details of how (we think) learning takes place when we talk about training neural networks.

Wetware, Software, and Hardware

Artificial neural networks represent our attempt to mimic the amazing capabilities of biological computers. Many of our existing technologies have been inspired by nature. This is sometimes called "biomimicry" because the solution we eventually come up with mimics the structure or function of nature's solution.

We recognize that there are lessons to be learned from nature. After all, nature has been changing and adapting for millions of years. Why not learn a few things from all that time and effort?

Think of things as diverse as the airplane, Velcro, distribution networks resembling leaf veins, and antibacterial surfaces inspired by sharkskin. Engineers often look to the natural world to see if nature has already figured out a workable solution to the problem.

Also (kind of a philosophical question perhaps), think about the huge advantage our ability to write things down and build a library of shared knowledge gives us. Each person doesn't have to learn everything all over again from scratch. We draw on a shared database of knowledge that doesn't have to be rediscovered! That may seem like a trivial thing at first, but it moves our species forward in a huge way!

We can't actually create living biological computers (yet), but we are learning to emulate them in hardware and software. And we are starting to get good at it! Are you getting excited to see where this thing is going? Let's just do a quick comparison between nature's neural networks and how we try to simulate them in hardware and software. This will be just a quick overview. In later chapters we will get more specific.

Wetware: The Biological Computer

"Wetware" is what we call biological computers. How cool is that?

Are neurons really wet? Well, believe it or not, the human body is about 50% water! The numbers vary quite a bit depending on age and sex, but that's a pretty significant percentage. If you poke yourself with a sharp object (not recommended) out will come blood. Blood is about 92% water by volume.

The problem with actual living biological neurons is that we can't manufacture them. Maybe one day, but not today. We are learning quite a bit by studying animal brains—even at the individual neuron level, but we can't build living biological networks at this time. Is this getting to sound a little bit like Star Trek bio-neural gel packs? Well, yesterday's science fiction is today's science, and (you know what I'm going to say) today's science fiction is tomorrow's science!

In wetware, how is the feedback and actual weight adjustment accomplished? We don't know. Will we ever know? At the current rate of discovery in brain research, it is pretty likely, and indeed may not even be too far off.

So, we do the best we can. We use biological networks (at least our current limited understanding of them) to build hardware and software systems that can perform similar functions. I mean, if you base your simulation on a model that is known to be successful, your chances of success should be pretty good, right?

Our options at this time are pretty limited. We can

1. Write software that runs on a conventional processor and try to emulate the way we think neurons actually work.
2. Produce hardware chips that contain electronic circuits that mimic actual biological neurons.
3. Try to combine these two approaches in a way that makes economic sense.

If you know anything about semiconductor manufacturing, I'm sure you realize that designing and setting up to manufacture a new chip takes a huge investment. Would Intel or Motorola make this kind of investment if the prospects for sales and profits were minimal? No way!

Software development for a product running on a PC can be very cost-effective. So, who wins? Software emulation.

But, if the goal is to implement a product using an embedded neural network, who wins? Hardware!

In real life, the program will probably be written in software and then compiled and downloaded to an embedded microcontroller. So what am I saying? It's probably still going to be a software simulation. You can search "till the cows come home" but today you probably won't find much for actual neural network "chips."

Real-life technology advancement and product development depend on several factors, the most important one being profit.

In this book, we will be building a neural network out of simple electronic components, but the options available to us today are amazing. Let me just mention a few:

1. Large, general purpose computers and PCs are hardware platforms capable of running a variety of different applications on the same machine. To accomplish a different task, you don't need a different machine, you just run a different program.

2. Recently, small, inexpensive computers like the Arduino and Raspberry Pi have become readily available. These machines are easy to program and are well supported by enthusiastic user groups. Also, there are plenty of add-on peripherals available to expand their functionality. Even young kids are already doing amazing projects using these simple machines.

3. Special-purpose machines and products with embedded controllers are more limited in scope, but can be produced fairly inexpensively.

4. As far as embedded systems go, we may spend a lot of time and money writing software and getting it working properly, and then we need to reduce the hardware to a minimum so we can build it into your toothbrush!

Software: Programs Running on a Computer

As shown in Figure 1-6, we can emulate a neural network with software running on a conventional sequential processor.

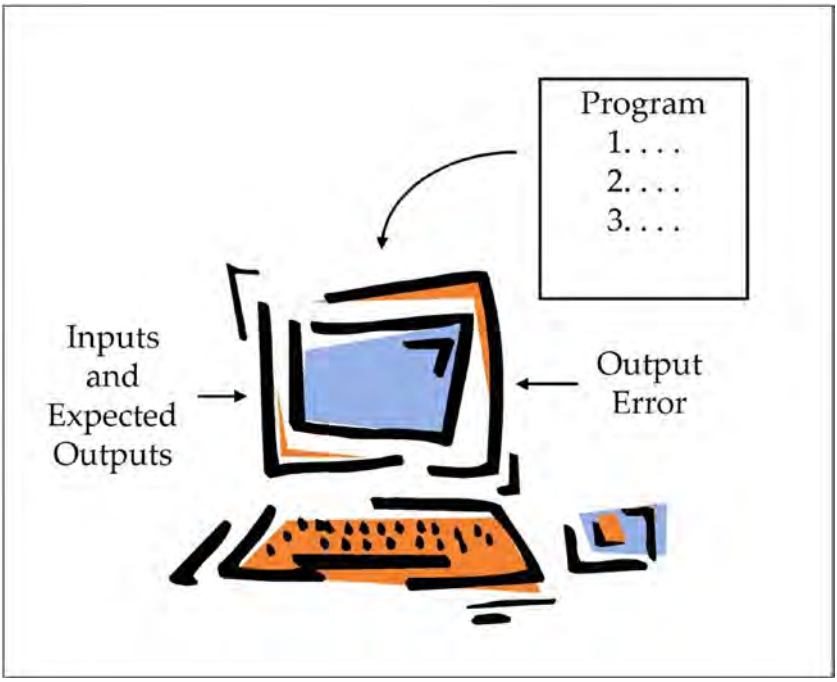


Figure 1-6. *Software implementation*

You might be thinking, "If we just write software running on a PC to emulate a neural network, how is that different from any other program?" Good question! These programs do not use step-by-step instructions that tell the processor exactly how to get the right answer. Why not? Because (many times) we don't actually know how to solve the problem, but we do know the desired results for a bunch of different inputs, and we know how the program can change a few things to get better at producing the desired results.

I know how strange that sounds, but hopefully these concepts will become clear as we go along.

Note Neural networks can learn to do things that we don't know how to do!

That's a theme that will run throughout this book—we have always built tools that outperform us. Think about that great old blues tune "John Henry," about a man who beat the steam-powered hammer but died in the process, or simple things like a pair of pliers. With a handheld calculator, you can easily do computations that would be difficult using just pencil and paper.

The programs we write to "train" a network to get better and better at a task involve the following steps:

1. Let the program produce a result based on the inputs.
2. Have it check its result against the correct answer that we have provided (all the inputs and desired results comprise the "training set").
3. Have it adjust the connection strengths between neurons to improve its results.
4. Have it repeat this process over and over until the error gets really small for all possible inputs.

Because computers can do things really fast and don't get tired, this process can be repeated millions of times if necessary.

So, instead of us knowing everything up front, we write code that will "learn" how to find the solution instead of writing code that we know will produce the correct solution.

You may be wondering, "What type of a problem can't we write a straightforward program for?" Well, how about voice recognition in a noisy environment or pattern recognition where some of the information is missing? To write step-by-step programs for tasks like these can be very difficult, but we can train a neural network to figure out how to solve the problem.

Hardware: Electronic Circuits

Figure 1-7 is a graphical representation of a hardware-based approach to implementing neural networks. There are no actual components shown. It's just meant to get you thinking about a different approach to implementing neural networks.

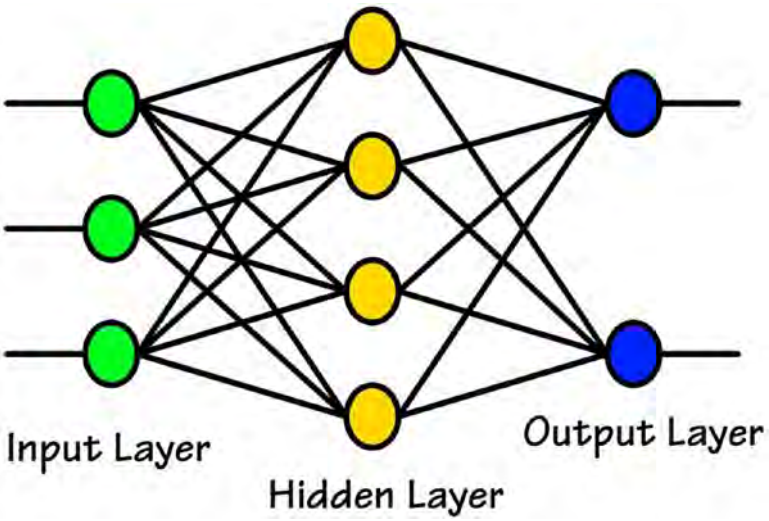


Figure 1-7. *Hardware implementation*

When we take a hardware-based or "components-based" approach we are trying to build electronic circuits that actually function as neurons. We build voltage summing circuits and transistor switches that can decide whether or not to fire. This is amazing stuff! In Chapters 3 and 4, we'll do an interesting electronics project, and then in Chapter 5 we will try to understand what we built.

Applications

This technology is advancing so rapidly that we are seeing new applications every day in fields as diverse as voice recognition, financial forecasting, machine control, and medical diagnoses. Any activity that requires pattern recognition is a prime target for the application of neural networks—especially pattern recognition in noisy environments or where some of the data is missing or corrupted. Tough problems like these can be solved using neural networks.

Whatever you can imagine can probably be done by a trained neural network. How about a machine that listens to the bearings in a commuter train and can predict bearing failure before it occurs. How about a machine that can predict earthquakes. Once you understand the typical characteristics of these networks, you start to realize that the possibilities are limitless!

As the technology matures and becomes more cost-effective, we will see many more applications running on large standalone machines and as embedded systems.

The amazing processing powers of neural networks running on large machines is already mindboggling, but to me, it is even more exciting to see neural networks becoming embedded in a wide variety of products.

When I say "embedded," I mean the neural network performs a specific function and is part of the product. It is small enough with low enough power consumption and inexpensive enough to simply be one of the

components of the product. Years ago, who would have imagined that there would be a computer in your car, your TV, or even in your watch! Can you say "smart phone"?

Just keep up with the news or do a search on the Internet and you will see new neural network and AI (artificial intelligence) applications cropping up every day.

Just Around the Corner

Maybe you have seen the movie *Alpha Go*. For the first time in history a neural network–based computer has consistently beaten the best human players in this complex game! For near-term advances I would especially watch companies like Intel, Nvidia, and Google. Only our imaginations will limit the possibilities!

OK, that's a brief introduction to neural networks. I hope you are excited to learn more.

CHAPTER 2

Implementing Neural Networks

OK, so now that we have had an introduction to neural networks in Chapter 1—how can we actually build one and make it do something?

One of the ways to make a complex task more manageable is to take a "top-down" approach. First we'll look at the big picture in general terms, and then we will be able to start implementing the details "from the bottom up."

To make sense of all this "top-down" and "bottom-up" stuff, let's start with the concept of architecture.

Architecture?

The word "architecture" may seem pretty technical and confusing, but it simply means how the different components are connected to each other and how they interact. No big deal. We need some kind of a word to describe it.

There is a big difference between the architecture of conventional computers and neural networks. Even if the neural network consists of a program running on a PC or a Mac, the approach is very different because we are trying to model the architecture of the brain.

R. McKeon, *Neural Networks for Electronics Hobbyists*,
https://doi.org/10.1007/978-1-4842-3507-2_2

When writing a program for a conventional computer, we tell it step-by-step exactly what to do. In other words, we need to know exactly what has to be done before we can write the program. The computer just follows our instructions, but it can execute those instructions millions of times faster than we could by hand, and it never gets tired or has to have a coffee break.

But for complicated, real-life problems with messy or missing data, we may not even know how to solve the problem! Holy smokes! Nobody can write a straightforward program for stuff like that.

When writing a program to simulate a neural network we don't need to know exactly how to solve the problem. We "train" the network by giving it various inputs and the correct outputs. At first, the network's performance will be pretty awful—full of mistakes and wrong answers. But we build in ways for it to make adjustments and "learn" to solve the problem.

So, you can see that these two approaches are very different. Of course, when the neural network is "component based" or running in hardware, the physical architecture is even more different than software running on a general purpose machine.

Figure 2-1 is just a symbolic representation of a neuron, but it helps us to visualize "connection weights," "summation," and "transfer functions." I know all this sounds pretty strange, but we'll take it one little step at a time and have fun with it. No complicated formulas or "techspeak"; just simple arithmetic like addition and subtraction. You can do this!

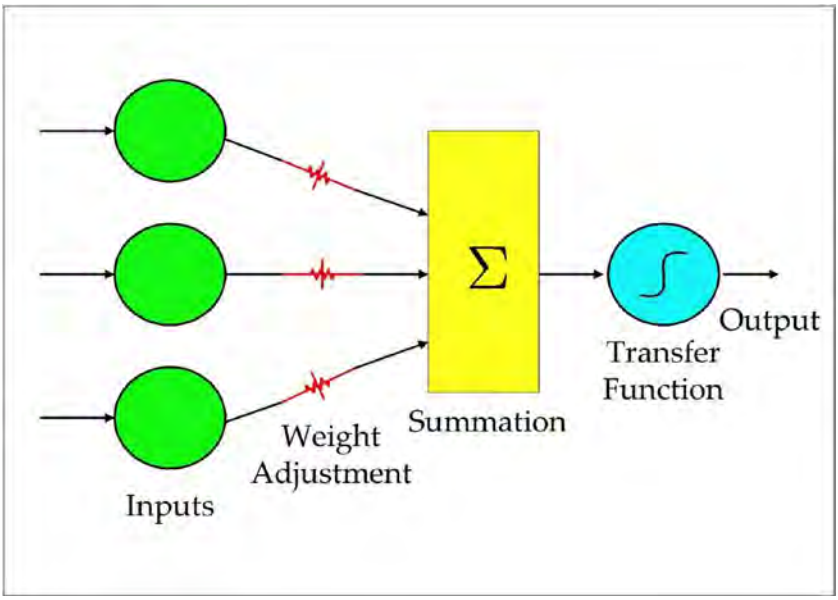


Figure 2-1. *The artificial neuron*

So, what Figure 2-1 is showing is that a neuron can receive several different signals of varying strengths (weights) and sum them up. Then it will decide whether to fire an outgoing pulse or not.

The "transfer function" can be complex or simple. For our purposes, it will be a simple Yes/No decision.

A Variety of Models

During the short history of neural network development, there has been a huge—I mean HUGE—number of models proposed. It's almost certain that none of them actually represent exactly how the human brain operates, but many of these models have shown remarkable success at pattern recognition! It's like the old saying, "Nothing succeeds like success." Does that apply here? Well yeah, I think so. Even though we don't have all the answers, we can build neural networks that perform amazing tasks!

This is a nontechnical introduction, so we are going to limit our discussion to one simple "feed-forward" approach using "back propagation" as the training algorithm. Believe me, this will be enough to keep you going for a while!

I love the concept of "back propagation of errors" because it makes so much sense. I mean, if someone is contributing to a wrong answer (feeding you bad information), he needs to have his input reduced, and if someone is contributing to the right answer (giving you good information), we want to hear more from him, right?

Our Sample Network

For this project, we are going to build a network to solve the XOR problem. It is a simple problem, but complex enough to require a three-layer network. We'll talk a lot more about XOR and the actual network in Chapter 3, but for now Figure 2-2 represents our three-layer network.

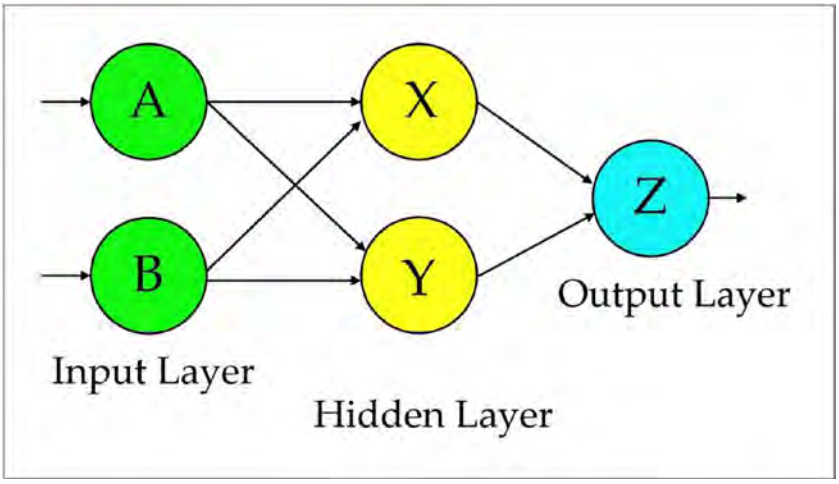


Figure 2-2. *Our sample three-layer network*

There are two inputs, two neurons in the hidden layer, and one output.

It's called a "feed-forward" network because the signals are sent only in the forward direction. There are some models that feed signals back to previous layers, but we are going to keep it simple for this one. We will use "back propagation of errors" to train the network, but that's just for training. In operation, all signals are only fed to the next layer.

The Input Layer

The input layer receives signals from the outside world kind of like our senses. Our eyes receive light from outside our bodies and convert it to signals that get sent from the retina along the optic nerve to the visual cortex in the back of our brain. It's interesting to note that the signals coming from our eyes aren't actually light. In fact, we construct our perception of the world entirely within our brain. All of the things we see, hear, feel, taste, or smell are really just electrical activity in our brains. I find this amazing! There is no light inside your head. It is completely dark in there! Is that spooky or what!

Our perception of the world is just electrical activity in our brain. Everything that seems so real to us is just activity based on signals coming from various sensors. Could we have receptors that are sensitive to other kinds of things? Would they seem just as real? Of course they would! Think about the possibilities!

The Hidden Layer

The hidden layer gets its name from the fact that it only has connections to other layers within the network. It has no connections to the outside world. So it is "hidden" from the outside world. Depending on the complexity of the task, a neural network may have several hidden layers. The network for our project will have only one hidden layer with two neurons.

The Output Layer

The output layer presents the results that the network has come up with based on the inputs and the functioning of the previous layers. For this project, there is just a single output that will be either ON or OFF. Many networks have several outputs that present text or graphic information, or produce signals for machine control.

Training the Network

Believe it or not, the connections between neurons in your brain can change. What? I mean the neurons can actually hook up differently. How else can I say it? Some connections can actually be terminated and new ones can be formed. Your brain can change! Did you ever think you had this kind of stuff going on inside your head? It's called "plasticity." If you want to get really technical, it's called "neural plasticity."

Not only can the actual connections change, there is a process that adjusts the amount of influence that a neuron has on other neurons. This may sound like science fiction, and you're probably thinking, "You gotta be kidding me."

Wherever neurons are connected to each other, they have a "synapse." That's a little gap or junction that the electrical signals have to cross over before they can affect the next neuron (kind of like how lightning strikes travel from clouds to ground).

So, are you ready for this? It might be really easy for the signal to jump across this gap, or it might be hard. Networks get trained by adjusting the "weight" or how easy it is to jump the gap. When a neuron has contributed to the wrong answer, the algorithm says, "We don't want to hear from you so much." Pretty harsh, I know, but that's the way it works. It's all part of the learning process called "back propagation of errors." It's like, "Those of you who did good get rewarded and those of you who did bad get sent to the back of the room."

Now, those neurons that contributed most to the correct answer have their connections reinforced or strengthened. It's like saying, "You did good. We want to hear more from you."

The networks we build today are called "artificial neural networks" because they merely "simulate" or "imitate" the workings of actual biological networks.

During the 1980s, neural networks were a hot topic and several companies started producing neural chips. They were usually 1024 × 1024 arrays of neurons that could be trained using a development system. That effort dropped off rapidly and everything reverted back to software that emulated neural networks and ran on conventional processors. OK, so it's got to have the potential to make money before anyone will invest in it.

In the next three chapters, we are going to build a network on a solderless breadboard and train it to perform the XOR logic function. The completed network will look something like Figure 2-3. This figure is not an actual schematic, just a graphical representation. We'll get into the actual components in Chapter 3.

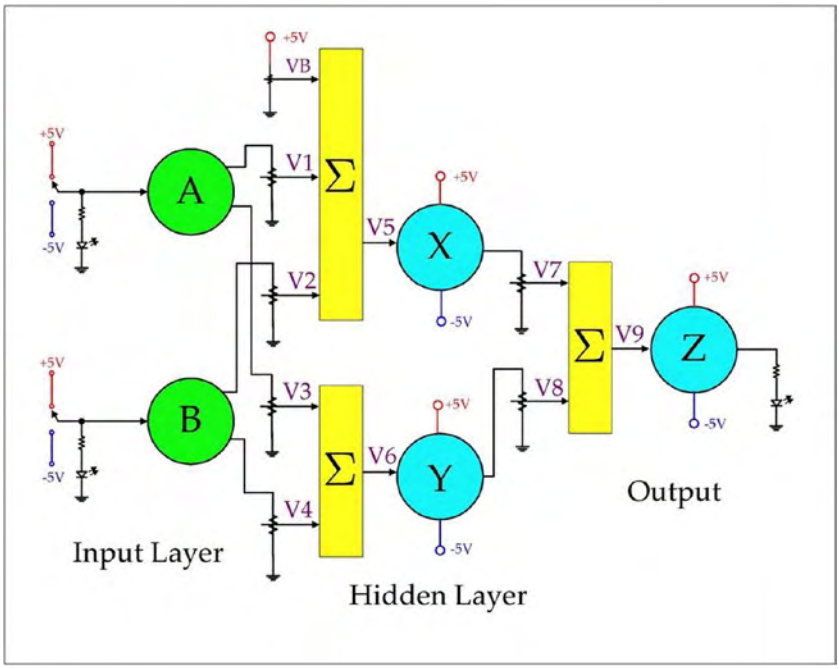


Figure 2-3. *Diagram of our completed project*

Just to whet your appetite, Figure 2-4 is a sneak preview of the completed project:

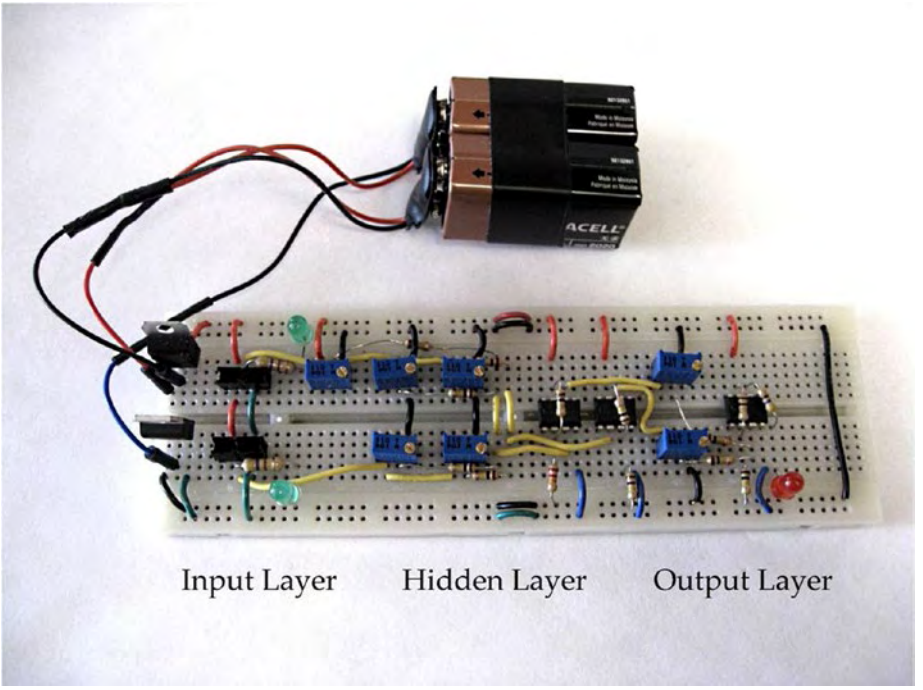


Figure 2-4. *Sneak preview of our completed project*

Summary

These first two chapters have been a high-level introduction to this exciting field. Now it's time to get down to the "nuts and bolts." OK, not really nuts and bolts—more like "wires and components." But in any case, I hope you are excited and ready to get out a breadboard, gather up some components, and start building a neural network!

CHAPTER 3

Electronic Components

In this chapter, we are going to introduce the electronic components that we'll be using to build our hardware-based neural network to solve the XOR problem (not that it's a "problem" it's just a logic function). If you like to dabble with electronic projects, I think you will have some fun with this one.

But, even if you have never done a project like this before, don't be afraid to give it a try! It's all pretty simple, and we'll take it one little step at a time. There are no high voltages, so don't worry about getting shocked or burning the house down. We will be powering the entire project with just a couple of 9-volt batteries.

Who knows, you might get interested in this kind of stuff and discover a new hobby!

What Is XOR?

The XOR function is used in many electronic applications. Figure 3-1 compares the OR and XOR functions. On the left-hand side of each truth table A and B are the inputs, and on the right-hand side is the output. Let's say that "0" means false, and "1" means true.

R. McKeon, *Neural Networks for Electronics Hobbyists*,
https://doi.org/10.1007/978-1-4842-3507-2_3

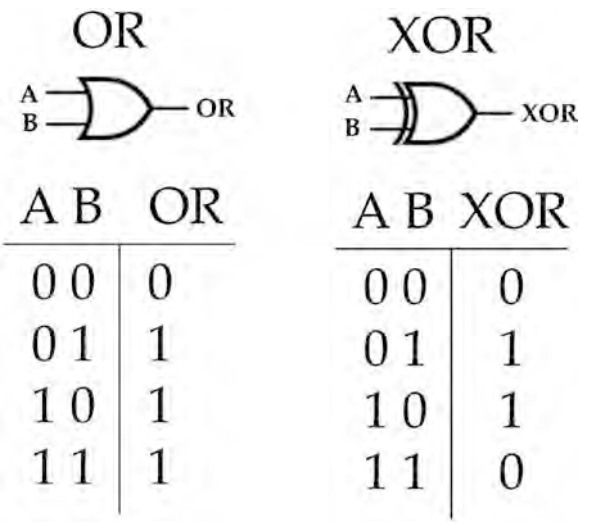


Figure 3-1. *OR and XOR functions*

For the OR function, if either input is true or both inputs are true, then the output is true. This is called "Inclusive OR" because it includes the possibility of both inputs being true.

The XOR function excludes the possibility of both inputs being true, so it is called "Exclusive OR."

In everyday language, we use the word "or" quite a bit. We might say, "Next summer I'm either going to the beach or to the mountains." In that statement it's understood that I'll be going to one place or the other, but not both. This is an example of the exclusive OR.

If the store clerk says, "We take cash or credit" he is using the inclusive OR because either cash or credit will work.

Now, here's the interesting thing about this simple function. It is "nonlinear" and if a neural network is going to solve it, there needs to be a hidden layer. A problem is nonlinear if a given input value can result in different output values depending on what the other inputs are doing.

For example:

If input A is TRUE and input B is FALSE, then the output is TRUE, but
If input A is TRUE and input B is also TRUE,
Then the output is FALSE.

So, here's a brief description of the components that we'll be using to implement this network in hardware.

ONLY BUY QUALITY PARTS!

We buy parts from all over the world and there is a huge difference in quality! The voltage regulators I originally purchased for this project were inexpensive, but dropped out of regulation with the least amount of load. You want everything in this project to be solid. The whole concept of training a network by adjusting weights (voltages) depends on those weights being reliable and consistent. I buy almost everything from `www.amazon.com` because they make it so easy. The regulators I found to be reliable came from STMicroelectronics.

Also, make sure the parts you purchase have leads with the size and spacing that are compatible with a breadboard. Don't buy switches with big solder lugs.

The Protoboard

Building our circuit on a "protoboard" will allow us to make electrical connections (and change them if necessary) quickly and easily without having to make permanent solder connections. It's called a "protoboard" because we use it as a "prototyping board." It is often called a "solderless breadboard."

It's amazing that such a useful component with hundreds of built-in connection points can be available so inexpensively. That's what mass production does for us!

Figure 3-2 shows the protoboard we will use in this project.

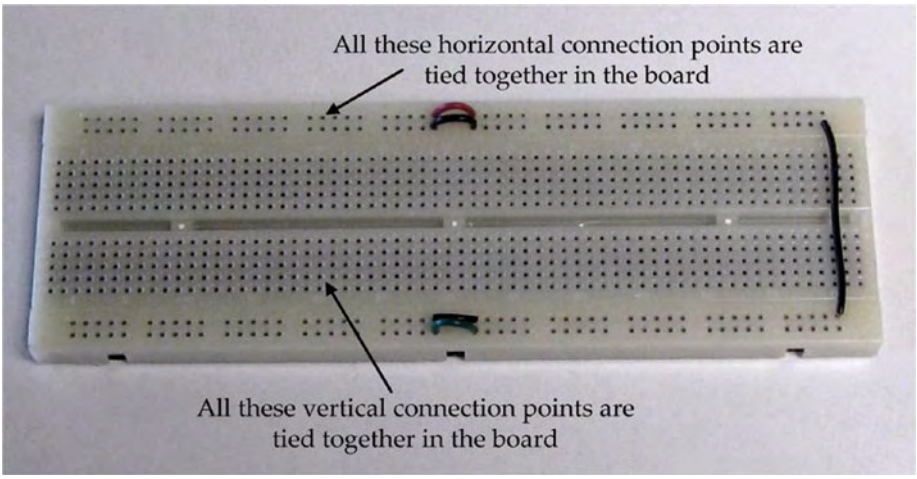


Figure 3-2. *The protoboard*

All of the components that we will use in this project have leads that are the right size to plug directly into the protoboard and make good contact. You can purchase jumper wires for making interconnections or simply make your own using 22 AWG solid (not stranded but solid) wire. Premade jumper wires are convenient, but I like to make my own so they will be the right length and keep the project nice and clean (OK, so I'm a clean freak).

We will use the long horizontal buses (four of them) as our power rails and the shorter vertical buses to mount components. We'll talk more about the protoboard as we start to build the circuit.

You will notice that this particular breadboard has the power buses split in half. This could add some versatility, but I jumpered them so they would run the entire length of the board. Also, I jumpered the two inside busses that we will be using as ground rails. So, for this project we will have power rails for +5V, -5V, and ground.

The Power Supply

As shown in Figure 3-3, the power supply for this project uses two 9V batteries. The reason for using two instead of just one is that we need both positive and negative voltages. This way we can have both "excitatory" and "inhibitory" neurons. In biological neural networks, excitatory neurons secrete neurotransmitters that tend to encourage the target neuron to fire and inhibitory neurons do the opposite. So, we have a "bipolar" (+5V and -5V) power supply. If a person exhibits bipolar behavior, that can be a bad thing, but for our power supply that's a good thing.

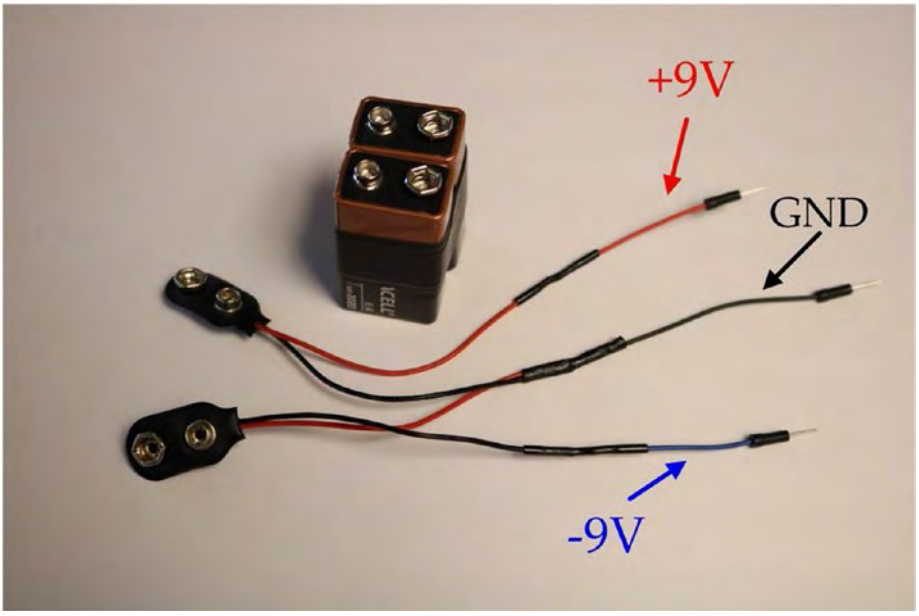


Figure 3-3. *Battery clips*

Notice how the battery clips are wired. I spliced on some jumper wires to the battery clip leads by stripping the insulation off and soldering them together. Then I put a short piece of "heat-shrink" tubing over each joint just to keep the exposed wires from touching anything. If you don't have heat-shrink tubing, you can simply wrap some tape around them.

To ensure the supply voltages are stable even under varying load conditions, we will use a +5V regulator (7805) and a -5V regulator (7905) instead of just running straight off the battery terminals. This project doesn't draw much current, so we can run directly off the voltage regulators.

ABOUT VOLTAGE REGULATORS

The purpose of a voltage regulator is to provide a solid, stable voltage that will not change under varying load conditions. A quality voltage regulator will give you power rails that don't change. But each regulator has a "tolerance" or accuracy relating to the "nominal" or stated value. The main thing is stability. The voltage regulators I am using for this project give a positive rail of 5.04V (That's extremely accurate) and a negative rail of -5.14V (not as accurate, but still within 5% of the nominal voltage). So, if your exact voltage readings differ a little bit from mine, no big deal.

When wiring up these regulators, be careful to observe their "pin out." The function of each pin is different between the two of them, as shown in Figure 3-4.

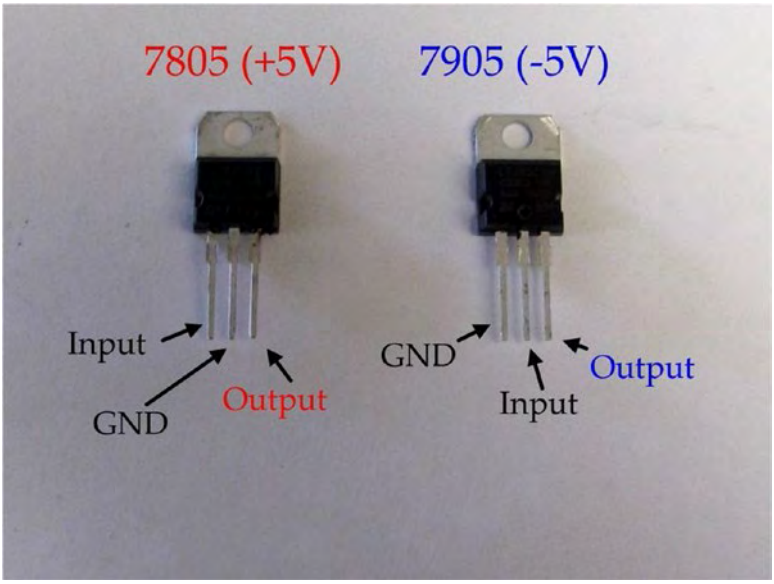


Figure 3-4. *Voltage regulators*

Figure 3-5 is a schematic of the power supply. See how we get both +5V and -5V?

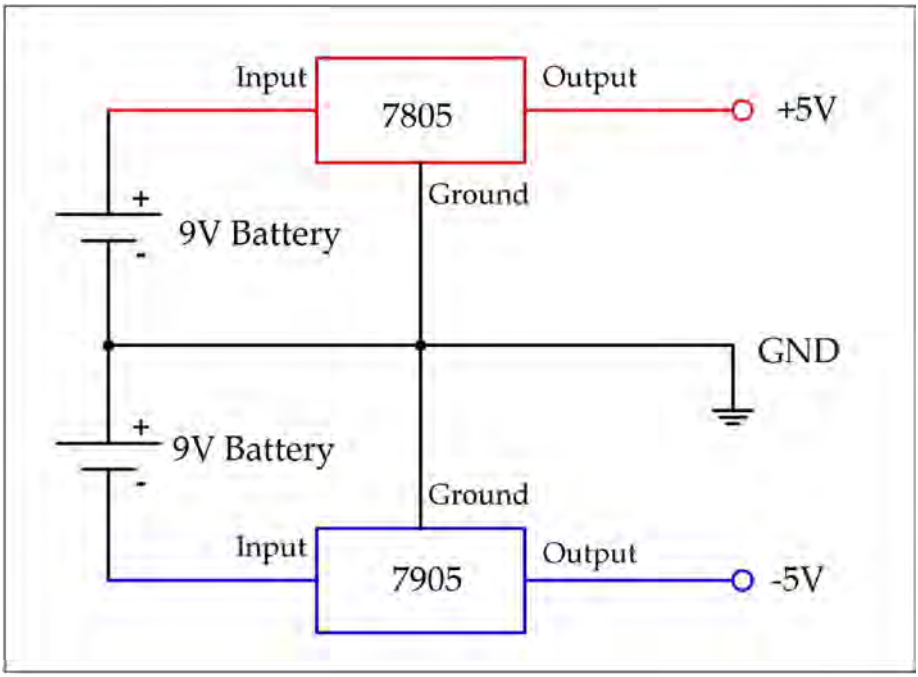


Figure 3-5. *The power supply*

Figure 3-6 shows the completed power supply connected to the prototyping board.

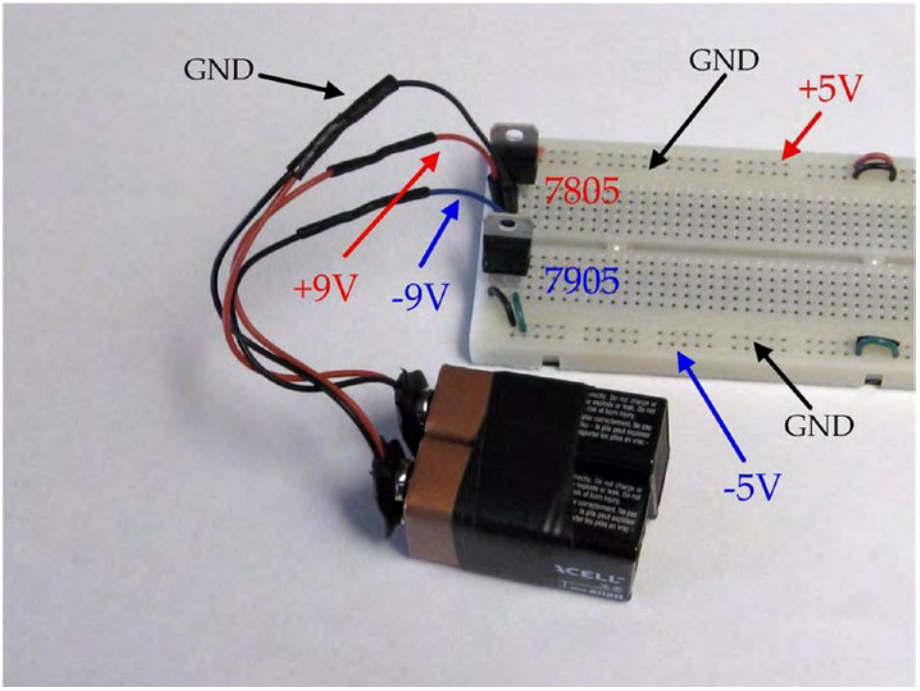


Figure 3-6. *Completed power supply*

Inputs

For the XOR function, there are two inputs called "A" and "B." They can have logic values of 0 or 1. In this circuit, the logic value 0 (False) is -5V and the logic value 1 (True) is +5V.

SPDT Switches

To present these inputs to the network we use a couple of SPDT switches. SPDT means "Single Pole Double Throw." The center contact of the switch can be connected to one side or the other, but not both at the same time. These switches are available in many different packages. For this project, we are using a pair of slide switches. As shown in Figure 3-7, when the slider is to the left the center contact is connected to the left contact, and when the slider is to the right the center contact is connected to the right contact. The contacts on these switches have the right size and spacing to plug directly into the protoboard.

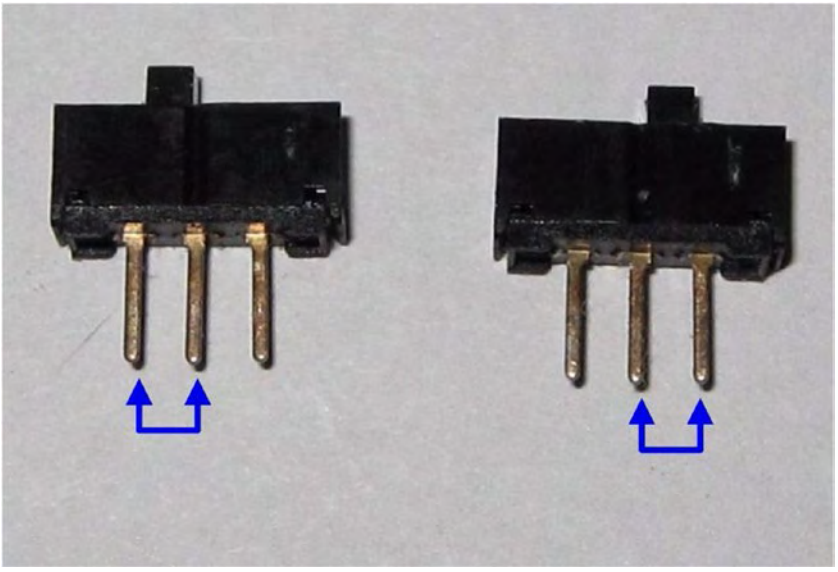


Figure 3-7. *SPDT switches*

When we present a logic value of 1 (+5V), we want to light an LED (light-emitting diode) as a visual indicator. When we present a logic value of 0 (-5V), we want the LED to be turned off. Does that seem like a challenging task? Well, actually it's pretty easy. Figure 3-8 shows how it works.

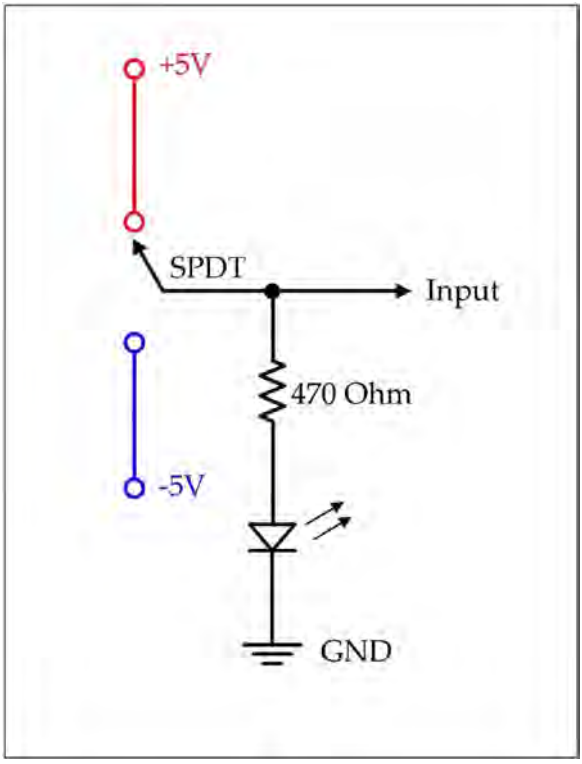


Figure 3-8. *Inputs*

When the switch is in the up position, we are presenting +5V to the input and also to the LED and its current-limiting resistor. When the switch is in the down position, we are presenting -5V to the Input and the LED will be off because it is "reverse biased." See how that works?

We use a 470-ohm resistor to limit current through the LED. Where does that value come from? This particular LED will be bright enough with around 6mA of current, and its forward voltage drop is about 2V. This means the resistor is dropping the rest of the supply voltage or about 3V. Applying Ohm's Law we see

$$R = V/I = 3V/6mA = 500 \text{ ohms}$$

So we are pretty close at 470 ohms.

Resistor Color Code

Resistors usually don't have their values written on them. Instead, they use a system of colored bands to tell the value and the tolerance. The tolerance means how accurate the stated value really is. For this project, we don't need precision resistors; 5% tolerance will do. This means that the actual value of the resistor is not more than 5% away from the stated value. For example, our 470-ohm resistor with a 5% tolerance will be somewhere between 446 ohms and 494 ohms. The tolerance values are usually pretty conservative. As you can see from Figure 3-9, the actual measured value was 461 ohms.

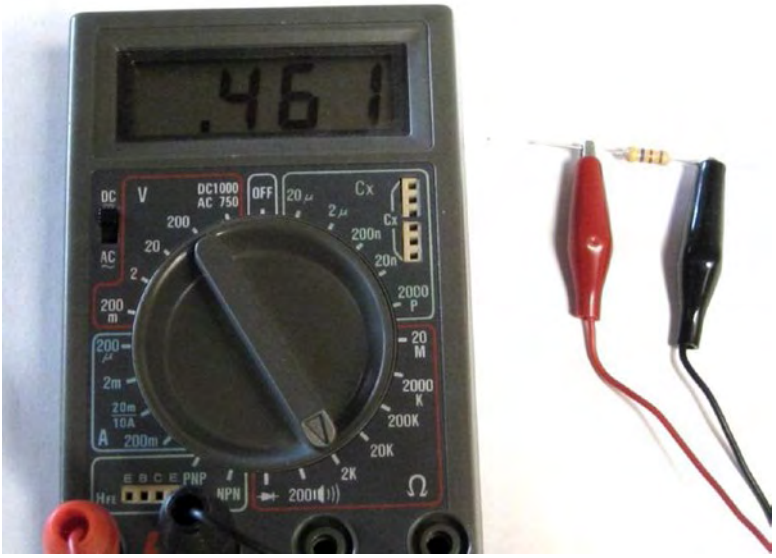


Figure 3-9. *Actual value of 470-ohm resistor*

So, how does this color-coding system work? There are four bands on the resistor. The first three tell its value and the last one tells the tolerance or accuracy. The first two bands you read like regular numbers and the third one tells how many zeros to add on. For example, our 470-ohm resistor has yellow, violet, and brown bands. So, that means 4 then 7 with one zero added on (Figure 3-10).

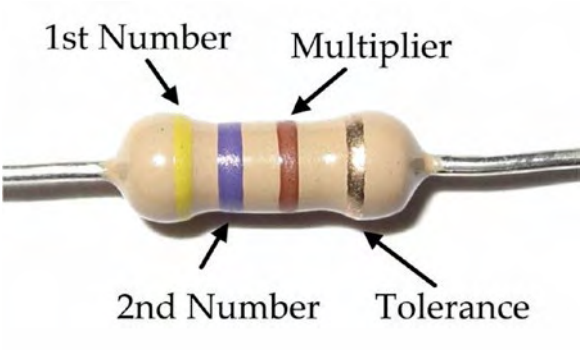


Figure 3-10. *Color bands for 470-ohm resistor*

Table 3-1 summarizes what the colors represent.

Table 3-1. *Color Values*

Color Code		Tolerance	
Black	0	Gold	5%
Brown	1	Silver	10%
Red	2	None	20%
Orange	3		
Yellow	4		
Green	5		
Blue	6		
Violet	7		
Grey	8		
White	9		

LEDs

The LED has two connections called “anode” and “cathode.” The anode is the positive side, and the cathode is the negative side. It’s easy to tell the cathode because it is the shorter leg and there is a little flat spot on the rim on that side. You can use Figure 3-11 to help you identify the anode and cathode.

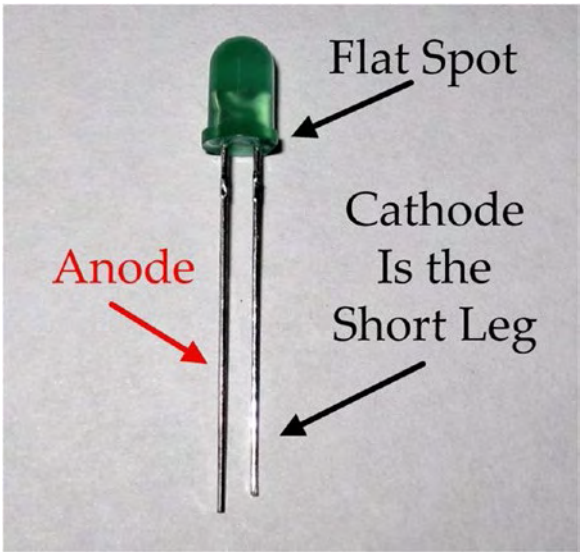


Figure 3-11. *LED*

What Is a Voltage Divider?

“Voltage divider” is another one of those terms that seems mysterious, but is really very simple.

When current flows through a resistor it is going to drop some voltage from one side to the other. That’s just how Ohm’s Law works.

So, how much voltage is dropped? Figure 3-12 shows three different arrangements of resistors placed across the terminals of a power supply.

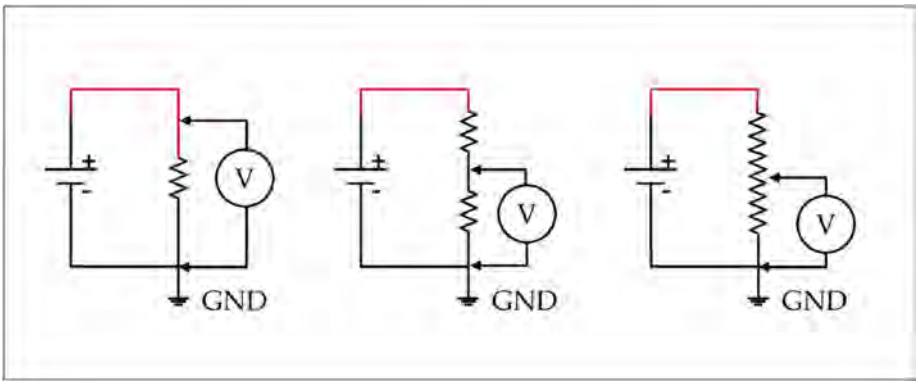


Figure 3-12. *Voltage divider*

The arrangement on the left-hand side shows a single resistor placed between the positive and negative terminals of the supply. Obviously it will drop the entire voltage. Measuring with a voltmeter across the resistor is the same as measuring directly across the supply.

The middle diagram shows two resistors in series across the supply. One of them will drop some of the supply voltage and the other one will drop the rest. How much each resistor drops depends on its value. With the same current flowing through both resistors (it has nowhere else to go), the one with the higher resistance will drop the most. Again, according to Ohm's Law the voltage will be equal to the current times the resistance. So for any given current, the voltage measured across it will get bigger as the resistance gets bigger.

$$V = I \times R$$

What would you measure if the two resistors were the same? You guessed it—half the supply voltage!

The diagram on the right shows a potentiometer across the supply. Now, a pot (like in potentiometer, not as in marijuana) is a special kind of resistor. It has a "wiper" as one of its connections that can contact the resistor anywhere along its length. In this way, we can measure the voltage potential all along the resistor. Depending on where the wiper is, we would measure a voltage equal to the entire power supply voltage all the way down to 0V.

Adjusting Connection Weights

Using a potentiometer as shown on the right-hand side of Figure 3-12, we can adjust the voltage or "weight" of an input. Figure 3-13 shows how it is done.

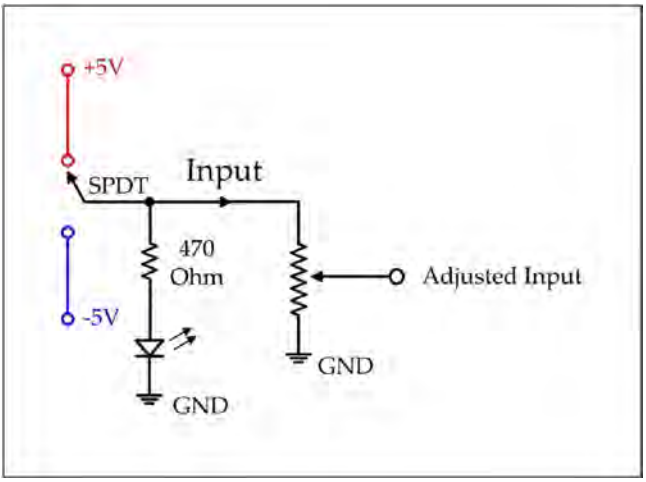


Figure 3-13. *Adjusting weights*

When the input switch is in the up position, we will be presenting +5V to the potentiometer, and the adjusted input will be between +5V and ground (0V).

When the switch is down, we will be presenting -5V to the potentiometer, and the adjusted input will be between -5V and ground (0V).

Potentiometers come in many different physical sizes and shapes. I chose the potentiometer shown in Figure 3-14 because its small footprint on the breadboard gives more room for wiring. Also, it is a "10-turn pot," which means that it takes ten complete revolutions for the wiper to go from one end to the other. This gives us loads of accuracy and stability!

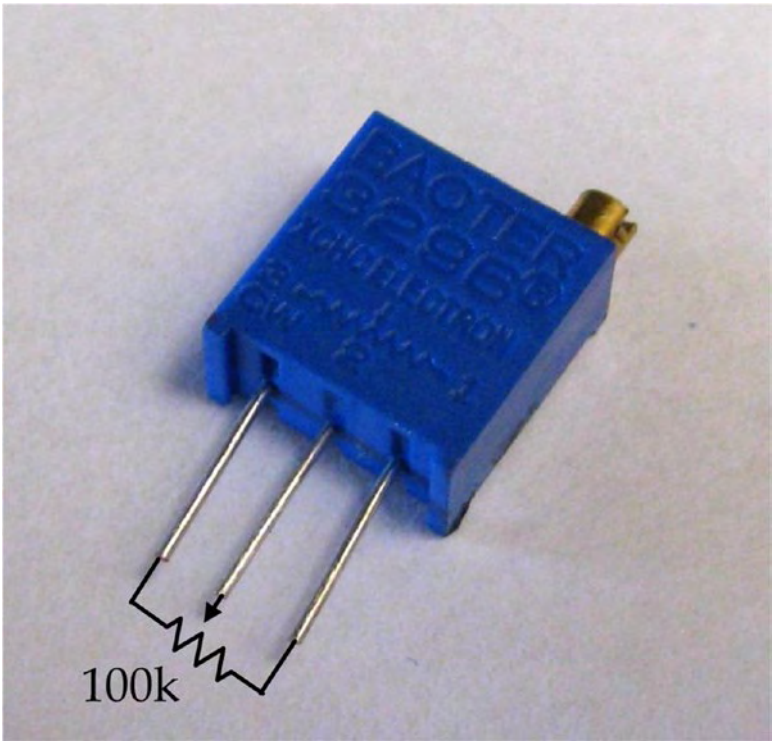


Figure 3-14. *Potentiometer*

Summing Voltages

Each neuron needs to sum the input signals it is receiving, and then make a decision what to do next based on their combined value. For this project, the neurons are receiving either two or three inputs.

1. Hidden layer neuron X receives signals from inputs A, B, and the bias voltage (VB).
2. Hidden layer neuron Y receives signals from inputs A and B.
3. Output neuron Z receives signals from hidden layer neurons X and Y.

To simulate how a biological cell would sum these signals, we will use a simple voltage divider network as shown in Figure 3-15. This kind of circuit is sometimes called a "passive averager." If all of the resistors are the same value, the average will be

Vout with two resistors = (V1+V2)/2 or
Vout with three resistors = (V1+V2+V3)/3

In other words, the average is simply the sum of the voltages divided by how many there are.

We'll talk more about the neuron's "threshold value" and "transfer function" in the next section.

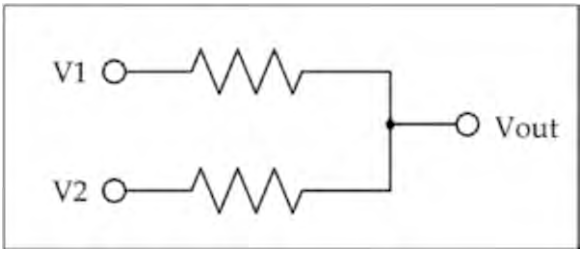


Figure 3-15. *Voltage summing circuit*

Op Amp Comparator

For this project, the op amp comparators are the components that simulate the neurons. We will be using CA3130 op amps. The circuits are really basic with just a few external components. Here's what we do:

1. We tie two of its pins to the positive and negative power rails,
2. We make a two-resistor voltage divider for the "activation threshold," and
3. We present the voltage from our summing circuit as the input.
4. That's it!

HERE'S HOW THE OP AMP COMPARATORS WORK:

If pin 3 is higher than pin 2, the output will be HIGH.

If pin 3 is lower than pin 2, the output will be LOW

So, just like actual neurons in the brain, these op amp comparators determine if the incoming signals warrant them firing. They only produce two different outputs because they swing from rail to rail.

Neuron Y is a little different from the other two. I'm calling it an "inhibitory" neuron because we are feeding the input signal to the inverting (-) input on pin 2 and presenting the threshold voltage on the non-inverting (+) input on pin 3.

So what's the thing about "threshold value" and "transfer function"? This could be pretty complex, but the way we are implementing them it's quite simple. As shown in Figure 3-16:

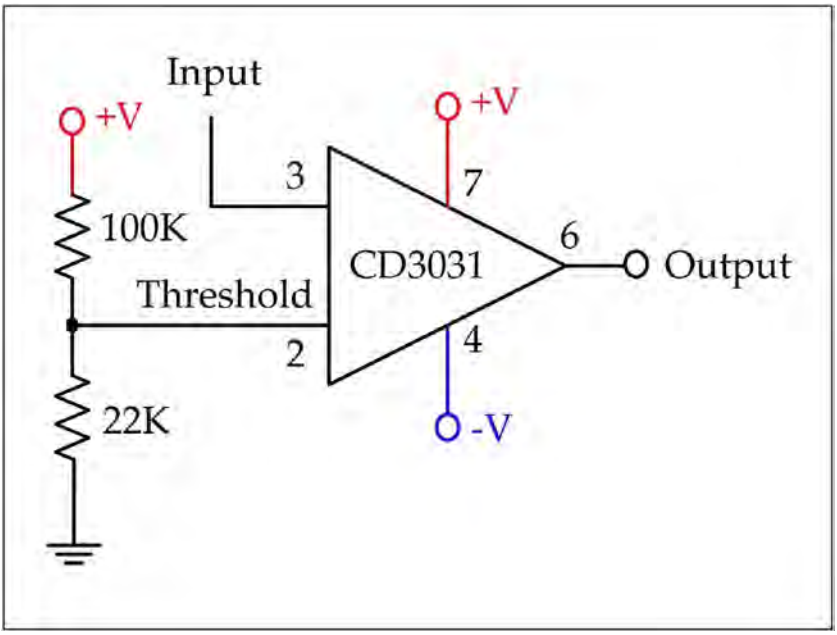


Figure 3-16. *Op amp comparator*

1. For the threshold value, we will use a simple two-resistor voltage divider. To get a threshold of +0.9V, we will make the bottom resistor 22K and the top resistor 100K.
2. The transfer function is just the op amp deciding whether to turn ON or OFF. That's why in this configuration it is called a "comparator." It's just comparing two inputs.

SO HOW DOES THAT 22K OHM/100K OHM COMBINATION GIVE US A +0.9V THRESHOLD?

Divide 22K by the total resistance of 122K to get what percentage of the total resistance the 22K resistor is.

That turns out to be 0.18 or 18%.

Then multiply 0.18 times 5V and you get +0.9V.

Putting It All Together

OK, we've discussed all the individual components. Now it's time to hook them all up and build a neural network!

Figure 3-17 shows the complete schematic for this project. Don't be overwhelmed by this schematic. I know it looks pretty busy, but in the next chapter we will build this thing one step at a time on the breadboard, and then we'll walk through the back propagation training algorithm one "training cycle" at a time. With each training cycle, we will reduce the error in the system, and eventually our network will perform the XOR function perfectly!

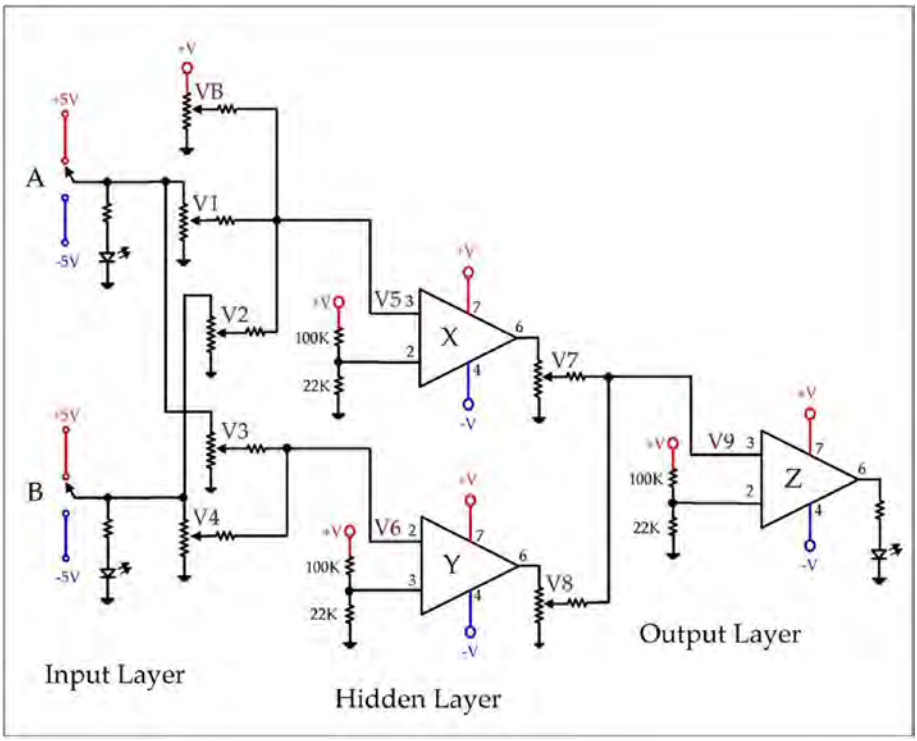


Figure 3-17. *Schematic for the XOR project*

This approach is manual and crude, but the goal is to get in there at the nitty-gritty level and experience how adjusting weights makes a difference.

When you finish this project, you will be "the first kid on your block" to have built a neural network in hardware! Give yourself a pat on the back! (Unless you have arthritis in your shoulders. Then just take another Aleve).

Parts List

Here's a summary of the parts required for this project.

1. Solderless breadboard (protoboard). Make sure you get a high-quality protoboard. There is a big difference in the quality and reliability of these breadboards! I'm telling you this from experience (bad experiences). Protoboards are great, and they allow for quick and easy connections, BUT if the reliability of those connections is questionable the whole project will be flaky and unreliable. After all, the breadboard is the backbone of the entire project. For this project, I used a Velleman SD12N.
2. Jumpers. These are convenient, but not really necessary. I prefer to make my own jumpers using a pair of wire strippers to cut and strip 22 AWG solid (not stranded, but solid) wire. This way you can make them the right length. That makes things clean and neat on the protoboard. My motto is, "Simple is good." It's nice to use different color jumpers for different functions. If you have wire in red, blue, yellow, and black, that would be great.
3. 9-volt batteries (2×)
4. 9-volt battery clips (2×)
5. 7805 +5V voltage regulator (1×)
6. 7905 -5V voltage regulator (1×)
7. 100K-ohm potentiometers (with leads that can plug into the protoboard) for adjusting weights (7×)

8. 100K-ohm resistors for the summing networks and threshold voltage dividers (10×)
9. CA3130 op amp (3×)
10. LEDs (3×)
11. 470-ohm current-limiting resistors for the LEDs (3×)
12. 22K-ohm resistor for the threshold voltage divider (3×).
13. Voltmeter. You will need an inexpensive voltmeter to measure and adjust voltages as part of the back propagation training cycles. It will be convenient if you make up leads of 22-gauge solid wire that you can stick in the breadboard connections. This way you can be "hands-free" as you adjust weights with a small screwdriver.
14. Wire strippers. Only necessary if you are going to make your own jumper wires. I highly recommend that you do!
15. Small long-nose pliers. These are not necessary, but believe me, they make it a lot easier to get into tight spots on the protoboard. You want to make this project easy and fun.
16. Small screwdriver for adjusting the pots. I got so frustrated with my little flat-bladed screwdriver slipping off the adjustment screw! So I slipped a little piece of heat shrink tubing over the end. It sticks out only a fraction of an inch, but voila! Problem solved.

Summary

In this chapter, we have discussed the various components that will go into our neural network. Each component is pretty simple by itself, but when we get the entire network assembled and trained it will be able to solve the XOR problem. In Chapter 4 we will take it step-by-step and assemble the network, and in Chapter 5 we will train it using the back propagation algorithm.

CHAPTER 4

Building the Network

Now we are going to build a three-layered neural network using simple electronic components. This network will have two inputs, two neurons in the hidden layer, and a single output. That might sound a little intimidating, but I have included plenty of step-by-step instructions together with photographs and drawings, so hopefully everything will go smoothly. We'll take it one layer at a time, including

1. Power supply
2. Input layer
3. Hidden layer and
4. Output layer

Note This circuit probably doesn't resemble how the brain actually functions at all. You can't get inside your head with a screwdriver and adjust things, but it is based on a model of what we think might be happening in the brain. Also, it makes for a good discussion of training using the back propagation algorithm. And it works!

R. McKeon, *Neural Networks for Electronics Hobbyists*,
https://doi.org/10.1007/978-1-4842-3507-2_4

Do We Need a Neural Network?

Certainly we don't need to build a neural network just to perform the XOR function. We can buy a 14-pin chip that has four XOR gates on it (CD4070) for $0.21.

So, the purpose of this project is not just to build an XOR gate, but to demonstrate how adjusting the weights in the hidden layer of a neural network can train it to perform this function.

OK, let's put this thing together and see if we can make it work.

Just to make it easier to see what's going on, I suggest using different color jumper wires for different functions as follows:

1. Red for connecting to +5V.
2. Black for connecting to GND.
3. Blue for connecting to -5V.
4. Yellow for signal wires.

The color choices are pretty arbitrary. Those are just the colors of 22-gauge solid wire I had on hand.

The Power Supply

Chapter 3 gives a fairly detailed explanation of how to wire up the power supply, so I won't elaborate on it too much here. When finished, the completed power supply should look something like Figure 4-1.

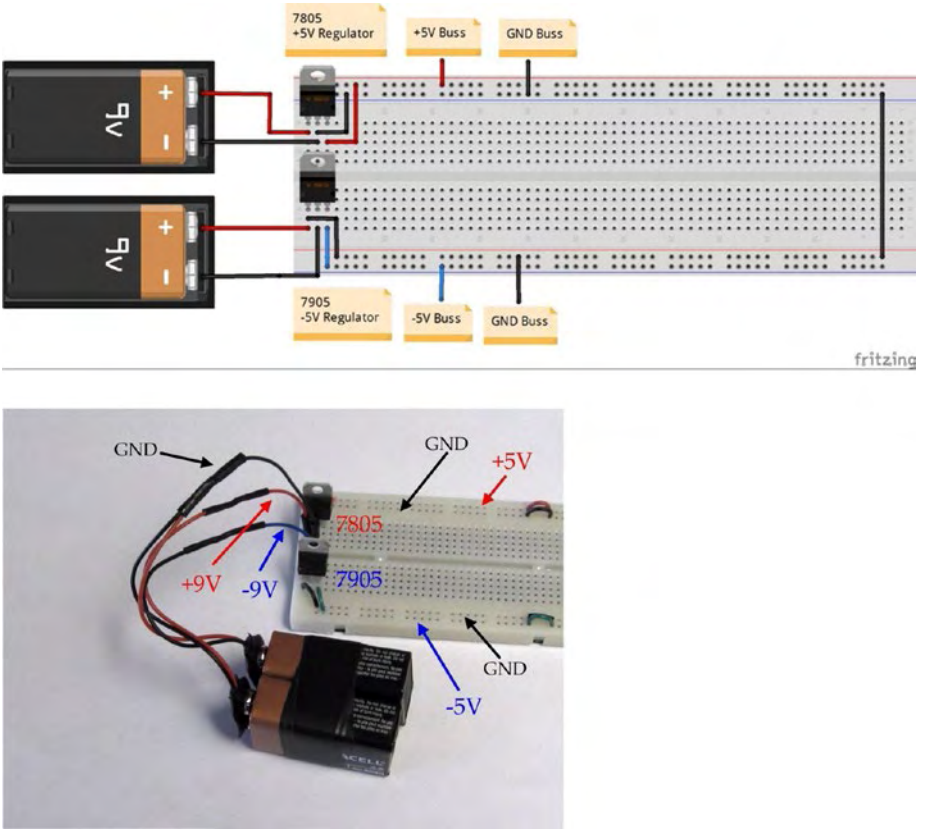


Figure 4-1. *Completed power supply*

Notice that the +9V and -9V wires from the batteries go directly to the voltage regulator inputs, and the outputs from the regulators go to the power rails. For this project, I chose to use the top rail as the +5V bus and the bottom rail as the -5V bus. We use the two inside rails as ground

buses. One thing that doesn't show in the photograph is a jumper wire connecting the ground buses at the other end of the breadboard. It does show in the drawing.

Once you have the power supply wired up, go ahead and check it with a voltmeter. On the power rails, you should measure something close to +5V and -5V with respect to ground.

If everything checks out, congratulations! You're good to go! If not, double-check your wiring. There's no sense in going forward until you have a solid power source.

The Input Layer

Now it's time to install the input switches and indicator LEDs.

A COUPLE OF SUGGESTIONS FOR YOUR CONVENIENCE:

1. Many times you can use the component leads to connect without using jumper wires. For example, the cathodes of the LEDs can plug directly onto the ground bus and the 470-ohm resistors can connect directly to the center post of the SPDT switches and to the LED anodes. No jumpers required!
2. To keep the space around the switches clear, I trimmed the leads on the 470-ohm resistors and laid them flat.
3. If you make up a set of leads for your voltmeter out of 22-Ga solid wire, it will make getting at the connection points a lot easier.

Use Figure 4-2 as a reference to wire the SPDT switches as follows:

1. Jumper the left side to the +5V rail.
2. Jumper the right side to the -5V rail.
3. Connect the LED anode (long lead) to the center connection via a 470-ohm resistor.
4. Connect the LED cathode (short lead) to the ground rail.

I trimmed the leads of the LEDs and resistors just to keep them from sticking up too far. You can still tell the LED cathode by the flat spot on the rim. Many times I don't trim leads because I plan to reuse the components in another project, but this breadboard is going to be on the front cover of the book, so I wanted it to look good.

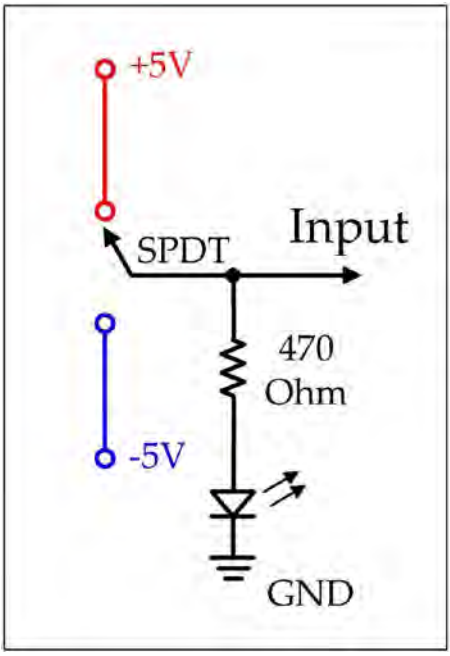


Figure 4-2. *Schematic for input switches*

Notice in Figure 4-3 that input switch A is on the top and input B is on the bottom.

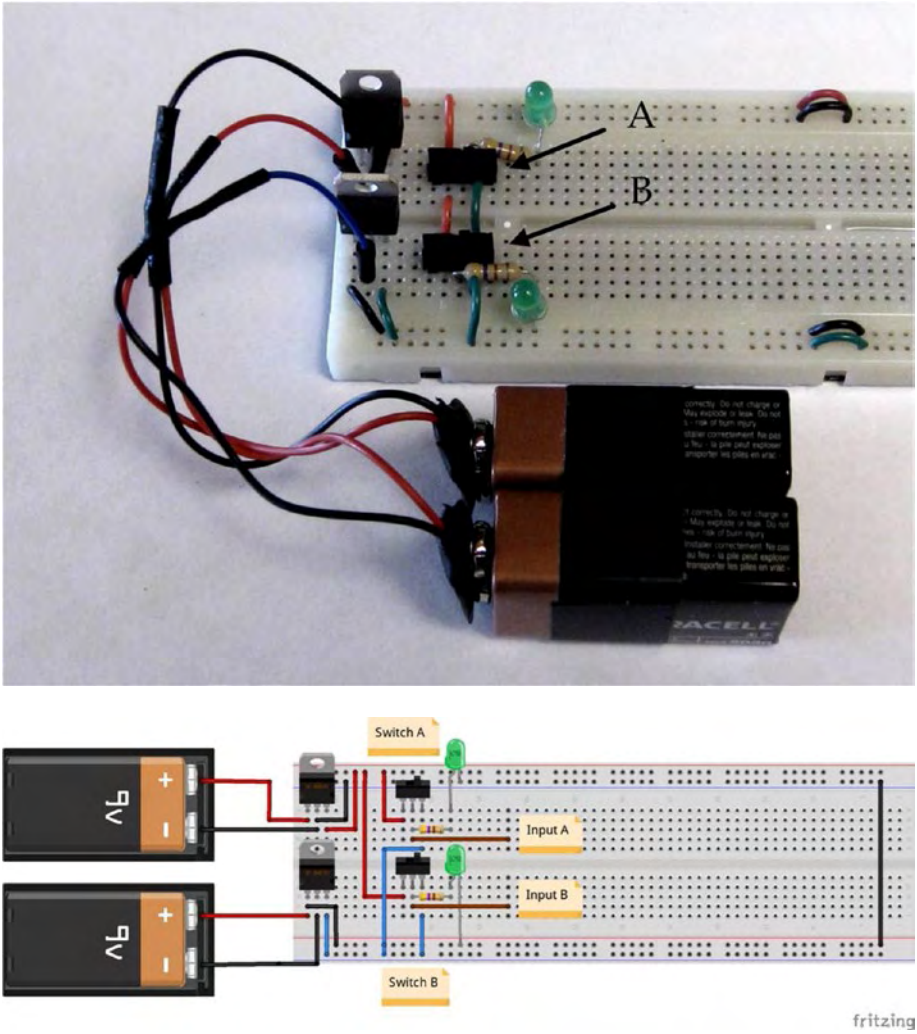


Figure 4-3. *Input switches and indicator LEDs*

Test your input layer to see if the LEDs turn ON and OFF. Also, measure the input voltage at the center contact of both switches in each position. You should read +5V for ON and -5V for OFF.

The Hidden Layer

All three layers are necessary, but the hidden layer is where the intelligence and the mystery lie! Figure 4-4 shows how we can

1. Adjust the bias voltage and weights from inputs A and B by using potentiometers RB, R1, R2, R3, and R4.
2. Sum those adjusted weights and present them as V5 and V6 to the neurons (op amps) X and Y.
3. Produce threshold voltages.
4. Wire up the remaining pins of the op amps.

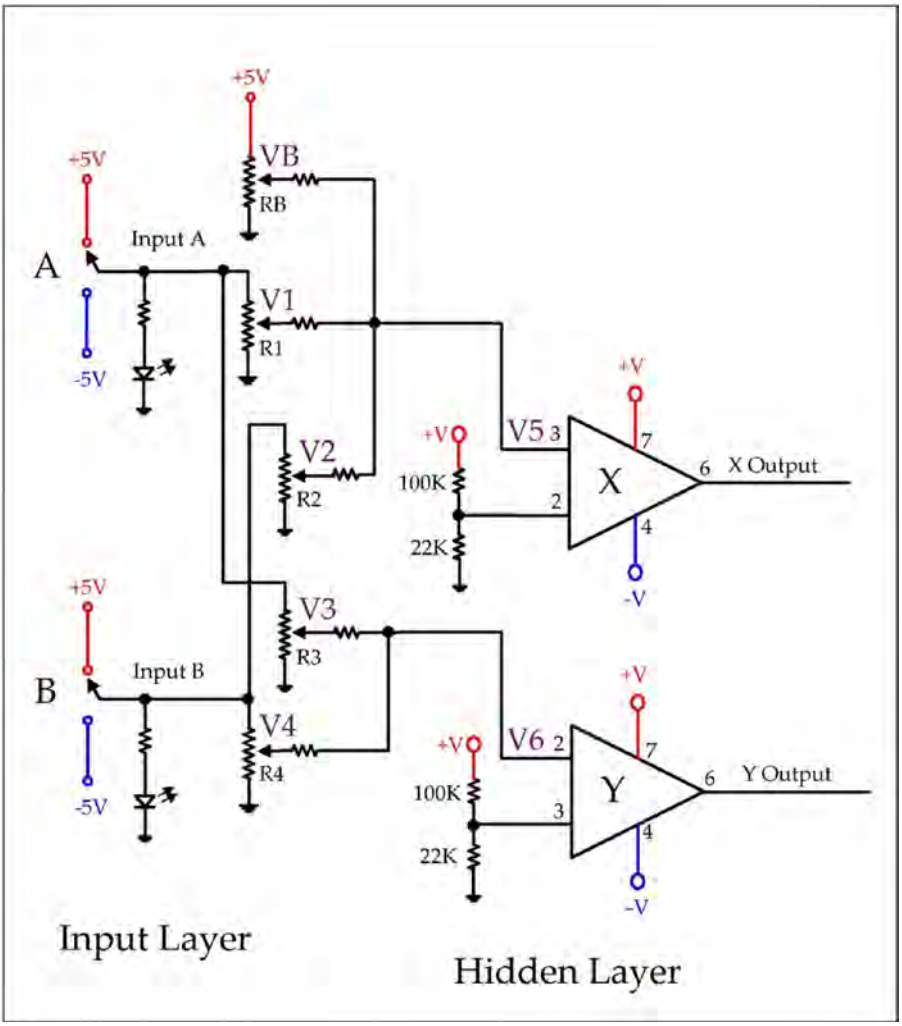


Figure 4-4. *Hidden layer schematic*

As you wire up each component, keep in mind that the vertical connections on the breadboard are all common, so you can use really short jumpers to connect to the power buses. I always like to use the shortest possible jumper. This keeps the board cleaned up and easy to follow.

Your actual component placement and wire routing may differ somewhat from what you see in these diagrams. That's OK. The purpose for the Fritzing diagrams is to help make things clear and easily identifiable. Therefore, I don't have wires running on top of each other or resistors plugged in at oddball angles, but on your breadboard you can use lots of little tricks to simplify the wiring.

Here's a step-by-step plan for wiring the hidden layer: Figure 4-5 shows how to install the potentiometers and hidden layer op amps, and Figure 4-6 completes the hidden layer.

Installing potentiometers and Op Amps

1. Install potentiometers RB, R1, R2, R3, and R4 on the protoboard.
2. Install op amps X and Y (CA3130) on the protoboard. Install them with the notch facing to the left, which means that pin 1 will be on the bottom.
3. Connect Input A (center contact of switch A) to one side of R1 and R3.
4. Connect the other side of R1 and R3 to ground.
5. Connect Input B (center contact of switch B) to one side of R2 and R4.
6. Connect the other side of R2 and R4 to ground.
7. Connect one side of RB to the +5V rail and the other side to ground.

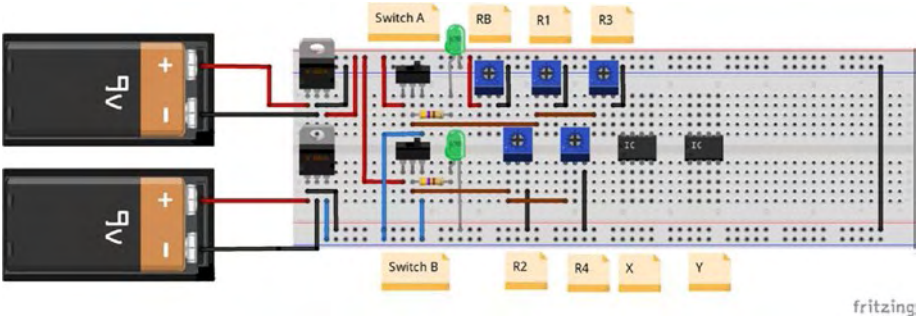


Figure 4-5. *Install potentiometers and op amps*

Installing Input Signals to the Op Amps

1. Connect 100K resistors to the center contact of RB, R1, and R2 and connect their other lead to a common point. This will be V5, which is the input signal to neuron X. Jumper that common point to pin 3 of neuron X.
2. Connect 100K resistors to the center contact of R3 and R4 and connect their other lead to a common point. This will be V6, which is the input signal to neuron Y. Jumper that common point to pin 2 of neuron Y. (You'll notice that we are connecting to pin 2 instead of pin 3 because neuron Y will be inhibitory.)
3. Jumper pin 7 of both op amps to 5V.

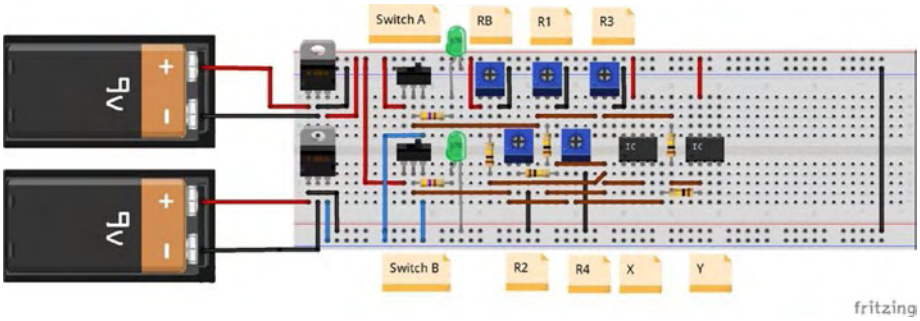


Figure 4-6. *Input signals to hidden layer completing the hidden layer*

4. To create a +0.9V offset for the threshold on pin 2 of the op amp X, tie a 22K resistor from ground to pin 2 and a 100K resistor from pin 2 to +5V.

5. To create a +0.9V offset for the threshold on pin 3 of the op amp Y, tie a 22K resistor from ground to pin 3 and a 100K resistor from pin 3 to +5V.

6. Jumper pin 4 of both op amps to -5V.4. The output of each op amp will be on pin 6.

Figure 4-7 shows the completed hidden layer.

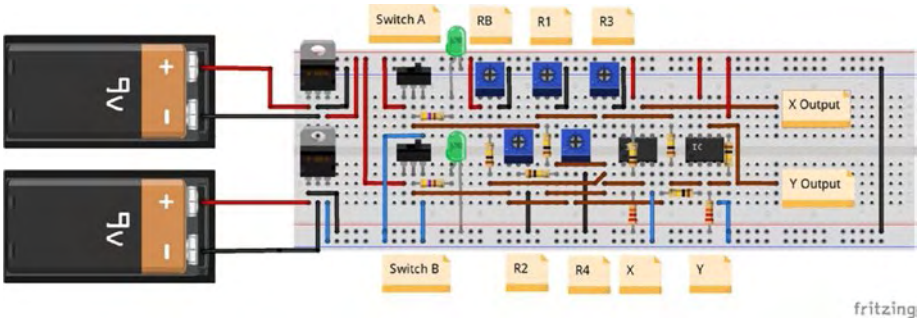


Figure 4-7. *Completed hidden layer*

Let's test our circuit to see if we have the hidden layer wired up correctly. We are going to measure the range of voltages possible by adjusting the potentiometer wipers from end to end. These ranges are shown in Tables 4-1 through 4-4.

It is important to perform this test. The first time I did it the voltage ranges were wrong. Turns out I had made a mistake in the wiring!

To understand the following ranges, keep in mind that our summation circuit adds the voltages and divides by how many there are.

Because we are running ten-turn pots from one end to the other, an electric screwdriver comes in really handy!

Table 4-1. *Testing Hidden Layer with A=0 and B=0*

A=0, B=0			
VB	**V1**	**V2**	**V5**
+5V to GND	-5.V to GND	-5V to GND	-3.33V to +1.67V

1. When VB, V1, and V2 are at their most positive, we have 5V divided by 3 = 1.67V2.
2. When VB, V1, and V2 are at their most negative, we have -10V divided by 3 = -3.33V

	V3	**V4**	**V6**
	-5V to GND	-5V to GND	-5V to GND

1. When V3 and V4 are at their most positive, we have 0V divided by 2 = 0V
2. When V3 and V4 are at their most negative, we have -10V divided by 2 = -5V

Table 4-2. *Testing Hidden Layer with A=1 and B=0*

A=1, B=0			
VB	**V1**	**V2**	**V5**
+5V to GND	+5V to GND	-5V to GND	-1.67V to +3.33V

1. When VB, V1, and V2 are at their most positive, we have 10V divided by 3 = 3.33V
2. When VB, V1, and V2 are at their most negative, we have -5V divided by 3 = -1.67V

	V3	**V4**	**V6**
	+5V to GND	-5V to GND	-2.5V to +2.5V

1. When V3 and V4 are at their most positive, we have 5V divided by 2 = 2.5V
2. When V3 and V4 are at their most negative, we have -5V divided by 2 = -2.5V

Table 4-3. *Testing Hidden Layer with A=0 and B=1*

A=0, B=1			
VB	**V1**	**V2**	**V5**
+5V to GND	-5V to GND	+5V to GND	-1.67V to +3.33V

1. When VB, V1, and V2 are at their most positive, we have 10V divided by 3 = 3.33V
2. When VB, V1, and V2 are at their most negative, we have -5V divided by 3 = -1.67V

	V3	**V4**	**V6**
	-5V to GND	+5V to GND	-2.5V to +2.5V

1. When V3 and V4 are at their most positive, we have 5V divided by 2 = 2.5V
2. When V3 and V4 are at their most negative, we have -5V divided by 2 = -2.5V

Table 4-4. *Testing Hidden Layer with A=1 and B=1*

A=1, B=1			
VB	**V1**	**V2**	**V5**
+5V to GND	+5V to GND	+5V to GND	+5V to GND

1. When VB, V1, and V2 are at their most positive, we have 15V divided by 3 = 5V
2. When VB, V1, and V2 are at their most negative, we have 0V divided by 3 = 0V

	V3	**V4**	**V6**
	+5V to GND	+5V to GND	+5V to GND

1. When V3 and V4 are at their most positive, we have 10V divided by 2 = 5V
2. When V3 and V4 are at their most negative, we have 0V divided by 2 = 0V

Now, once you have gone through this exercise, look at the schematic and figure out where these numbers come from.

The Output Layer

Now we'll finish building our network by wiring up the output layer. Refer to Figure 4-8 as we add the output layer to the breadboard.

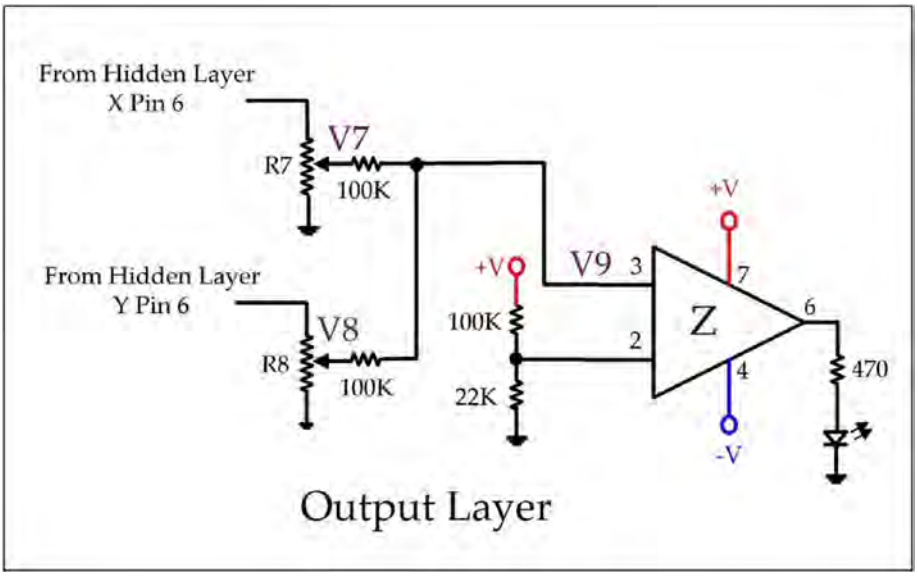


Figure 4-8. *The output layer*

Here's a step-by-step plan for completing the output layer: Figure 4-9 shows the potentiometers and op amp Z, and Figure 4-10 shows how to wire the inputs to op amp Z.

Installing Potentiometers and Op Amp Z

1. Install potentiometers R7 and R8 on the protoboard.
2. Install op amp Z (CA3130) on the protoboard. Install it with the notch facing to the left, which means that pin 1 will be on the bottom.
3. Connect the Output from X (on pin 6) to one side of R7.
4. Connect the other side of R7 to ground.
5. Connect the Output from Y (on pin 6) to one side of R8.
6. Connect the other side of R8 to ground.

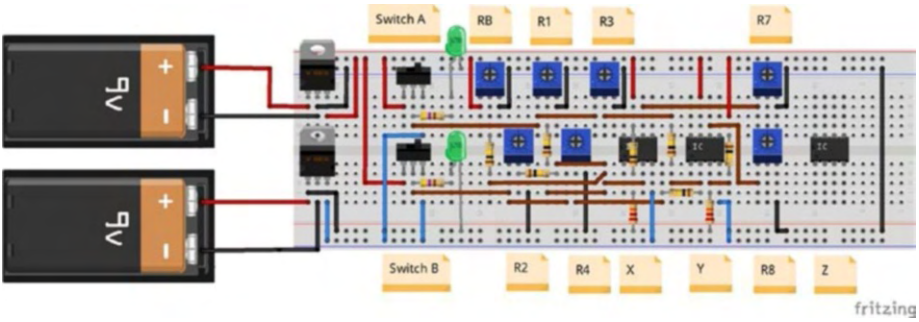


Figure 4-9. *Installing potentiometers and op amp Z*

Installing Inputs to Op Amp Z

1. Connect 100K resistors to the center contact of R7 and R8 and connect their other lead to pin 3 of op amp Z. This will be the input V9 to neuron Z. You can do this using just the resistor leads themselves. No jumpers required.
2. Jumper pin 7 of op amp Z to 5V.

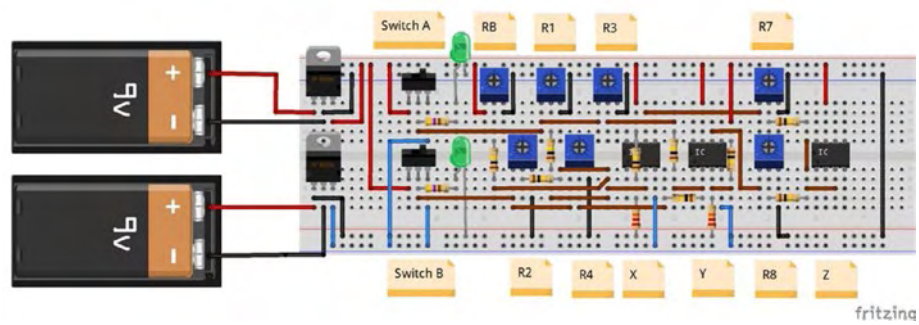


Figure 4-10. *Installing inputs to op amp Z*

Finishing the Output Layer

1. To create a +0.9V offset for the threshold on pin 2, tie a 22K resistor from ground to pin 2 and a 100K resistor from pin 2 to +5V. You can do this on the protoboard using just the resistors themselves. No jumper wires required.
2. Jumper pin 4 of op amp Z to -5V.
3. Connect the LED cathode to the ground bus and its anode to pin 6 via a 470-ohm resistor.

Figure 4-11 shows the completed circuit installed on the breadboard.

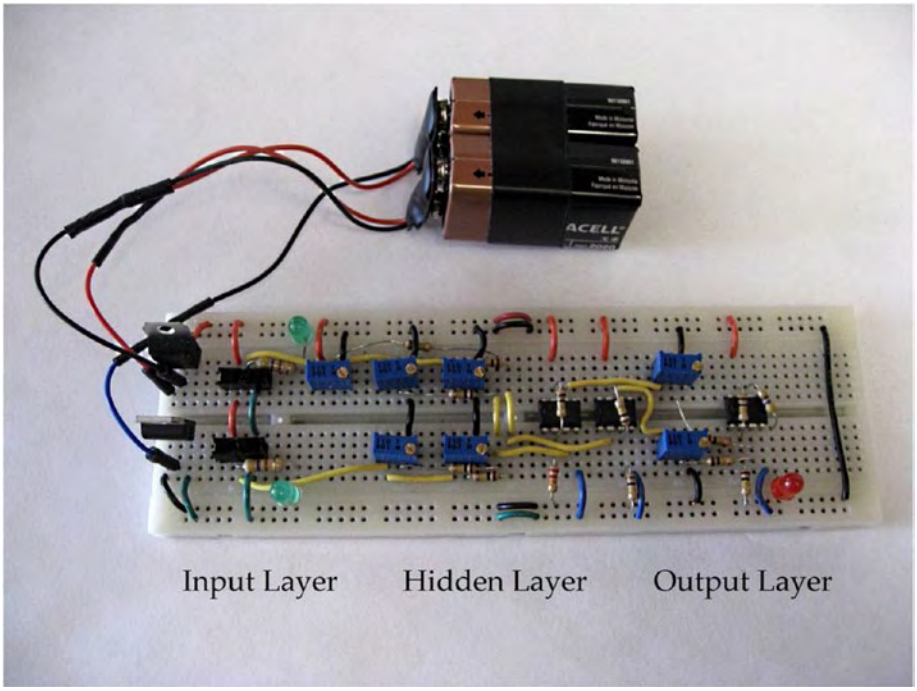
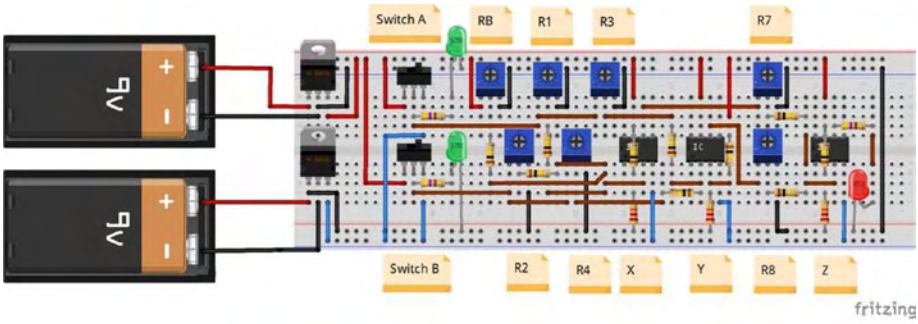


Figure 4-11. *Completed circuit*



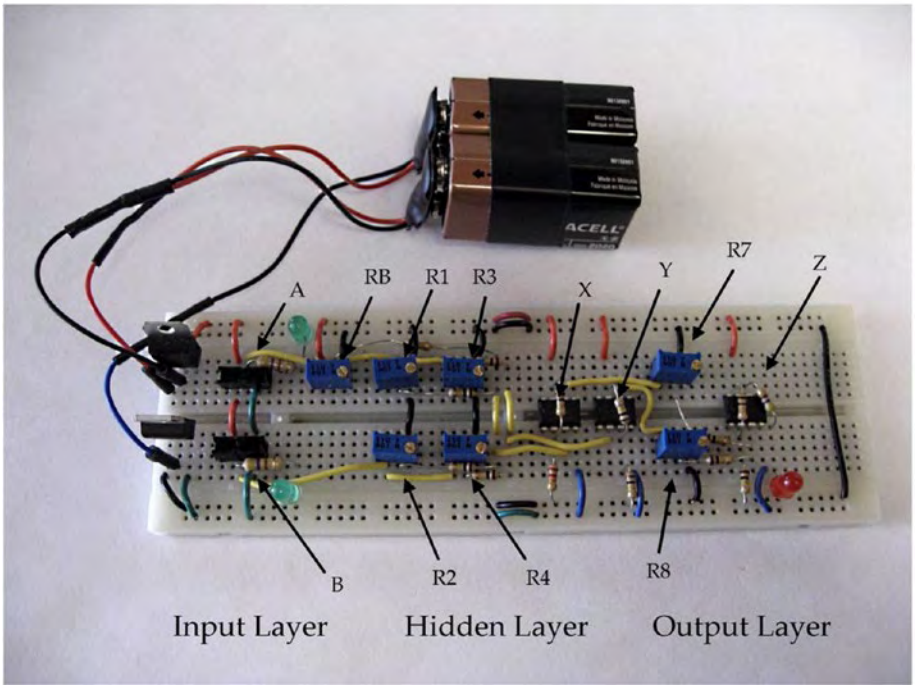


Figure 4-11. (*continued*)

Testing the circuit

Now that we have the network built we can test it on the XOR function. Guess what—it probably won't work! It doesn't know how to produce the right output for each of the four different combinations of inputs. Why not? It hasn't been trained.

Now the fun begins! In Chapter 5 we are going to train the network to recognize the XOR function, and we are going to make some interesting observations regarding weight values.

Summary

Congratulations! You have built a neural network out of electronic components! In Chapter 5, we are going to train this network to solve the XOR function. The approach that we are going to take in training the network is pretty "manual" in that we actually adjust the weights by tweaking potentiometers with a screwdriver. It doesn't get more manual than that!

A real biological network, like your brain, can make its own adjustments automatically—no screwdriver required. But we will adhere to the back propagation algorithm, and the network does eventually get trained. Without following the algorithm, I wouldn't know how to adjust the weights to make it perform properly, so the network really does learn to do something that I don't know how to do!

I hope you are excited to see what happens as we train our network.

CHAPTER 5

Training with Back Propagation

What does it mean to "train" a neural network? The network model we are using "learns" or gets better at its task by adjusting its connection strengths. We will strengthen those signals that tend to contribute to a right answer, and weaken those signals that tend toward a wrong answer. We do this by adjusting the potentiometer that lies in the path of each signal. It's kind of like changing the conductive environment in a neuron's synapse. To accomplish this, we will simply increase or decrease the voltage in question by 0.2V. That's our method of implementing back propagation of errors. Pretty crude, I know, but it makes for a good simulation of the process.

Note The only way our training will be valid is if we stick strictly to the training algorithm. What this implies is that we are not going to intervene in the training process; we are simply going to follow the training algorithm as defined!

So, get out your voltmeter and your screwdriver, and let's work through these training cycles together.

The "back propagation of errors" algorithm is very well known, and everyone implements it in software. In this project, we will implement it in hardware. At first, the weights are just random, so it's no wonder the output

R. McKeon, *Neural Networks for Electronics Hobbyists*,
https://doi.org/10.1007/978-1-4842-3507-2_5

neuron behaves randomly. As the network learns, the analog weights of the hidden layer have to somehow start to represent the function, right?

Later in the chapter, we will talk a little bit about the concept of "feature extraction." Can we learn something about the network by looking at the hidden layer weights and their changes as we go through several training cycles? We'll look a little closer at this interesting question in the "Attractors and Trends" section.

Figure 5-1 is not a real schematic; it's a simplified diagram to help us visualize what's happening as we walk through the manual steps of training our network. It shows the important voltages (weights) that will be modified during the training process.

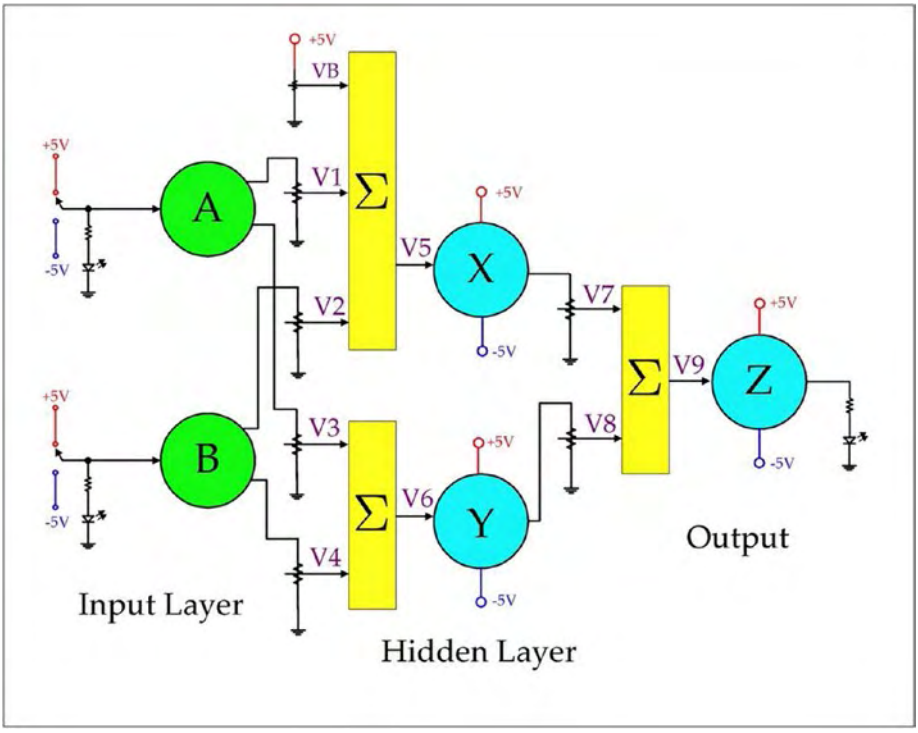


Figure 5-1. *Training with back propagation*

Things to notice about Figure 5-1:

1. The outputs from A and B only have two states. They are binary as in the XOR truth table. If we measure them with a voltmeter, 1 = true = +5V and 0 = false = -5V.
2. It's easy to see if an input is a "1" or a "0" because the LED will be lit or not.
3. The bias voltage VB is analog and can vary between +5V and GND.
4. The output from Z is binary as in the truth table. The LED will be lit for "true" and not lit for "false."
5. V5 is the sum of the bias voltage VB, V1, and V2. All three of these voltages can be adjusted using potentiometers.
6. A similar thing can be said for V6 and V9.
7. The summation voltages (V5, V6, and V9) to neurons X, Y, and Z are analog. We call these "activation voltages" because if they are greater than the "threshold voltage" for that neuron, the neuron will turn ON (+5V). If they are less than the thresholds, the neurons will turn OFF (-5V).
8. Don't be concerned if the op amps don't exactly swing from rail to rail. Under load, their internal resistance may cause them to be less than the rail voltage. The main thing is if they are ON or OFF. So if their output voltages are not exactly +5V or -5V, don't sweat it.

The Back Propagation Algorithm

When we talk about neural networks, we speak in terms of physical layers like input layer, hidden layers, and output layer, but I see this project as consisting of conceptual layers too (OK, maybe a little philosophical).

1. At the application layer, we want to get the LEDs to light up like in the truth table.
2. At the logical layer, we design a schematic based on a neural network model.
3. At the physical layer, we build the circuit out of electronic components and train it by tweaking its weights.
4. At the intellectual layer, we sit back and think, "Wow, this is interesting stuff!"

It may seem a little bit counterintuitive that either switch A or B in the ON position could turn the output ON, but both switches ON could turn the output OFF. So how does that happen? I think the answer lies in the fact that Y is an inhibitory neuron.

DON'T WORRY: IT WILL WORK!

After several unsuccessful training cycles, I started to get concerned that the network wasn't actually training, and started to wonder, "Can a network with this specific architecture actually solve the XOR problem? If it can't, I'm just wasting my time tweaking the weights!" So, just to set your mind at ease, do the following. Set both inputs A and B OFF and set your weights up like this:

```
VB = 4.06V, V1 = -1.11V, V2 = -1.71V, V3 = -2.72V,
V4 = -1.42V, V7 = -1.03V, and V8 = 2.18V.
```

My negative rail (-5.14V) might be a little bit lower than yours, but these settings should be pretty close. It may take a little tweaking, but it won't take you long to get the weights dialed in so that the four different input combinations yield correct outputs. Once you realize that the network actually can work, you can proceed with confidence, knowing that you are not wasting your time.

As I was just messing around adjusting weights I started to realize that I was actually using back propagation, only without the formal algorithm. I would think, "OK, we're almost there. How can I get X to turn ON? What do I have to adjust? I want to change this weight without affecting others too much." And, "You have to get this voltage just a little bit higher. Which pots affect it, and which way do they have to go?" Eventually I found the right combination of weights and bias.

Just to bolster your confidence, Figures 5-2 through 5-5 show the trained network solving XOR for the four possible input combinations.

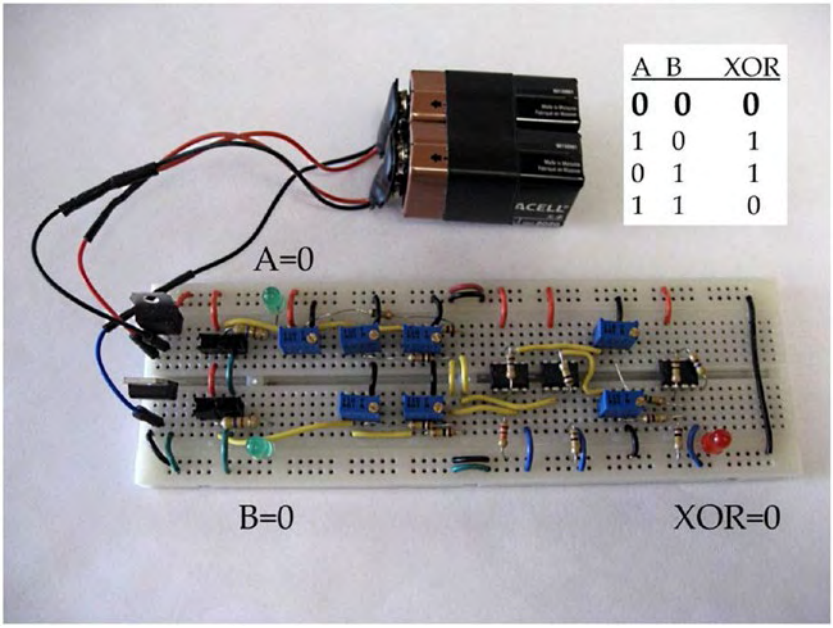


Figure 5-2. *Trained XOR network: 000*

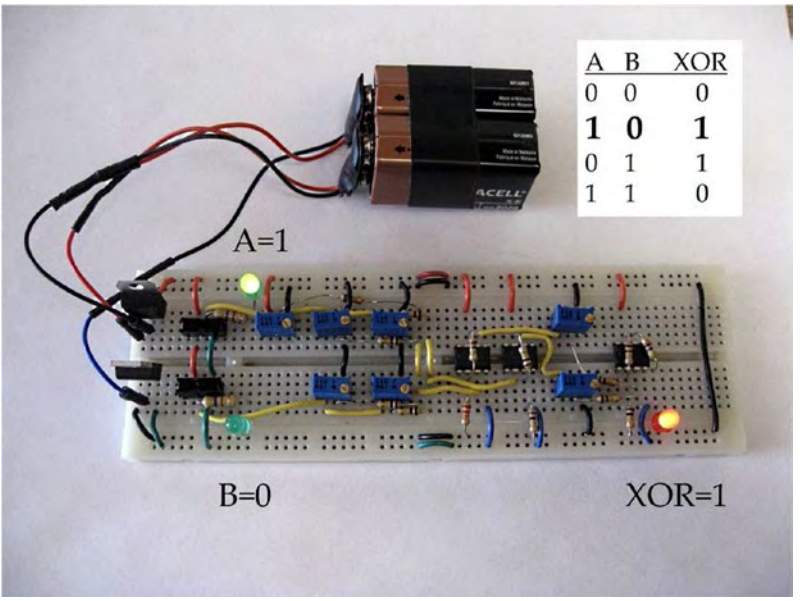


Figure 5-3. *Trained XOR network: 101*

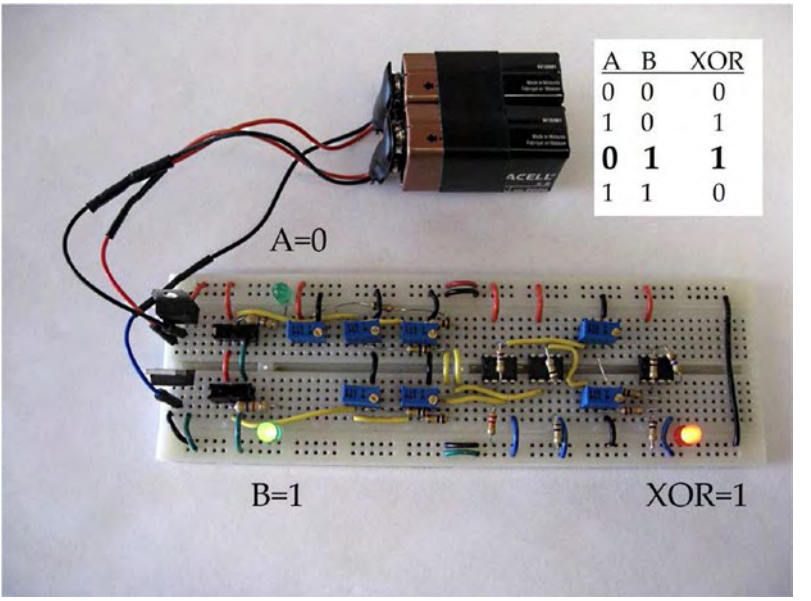


Figure 5-4. *Trained XOR network: 011*

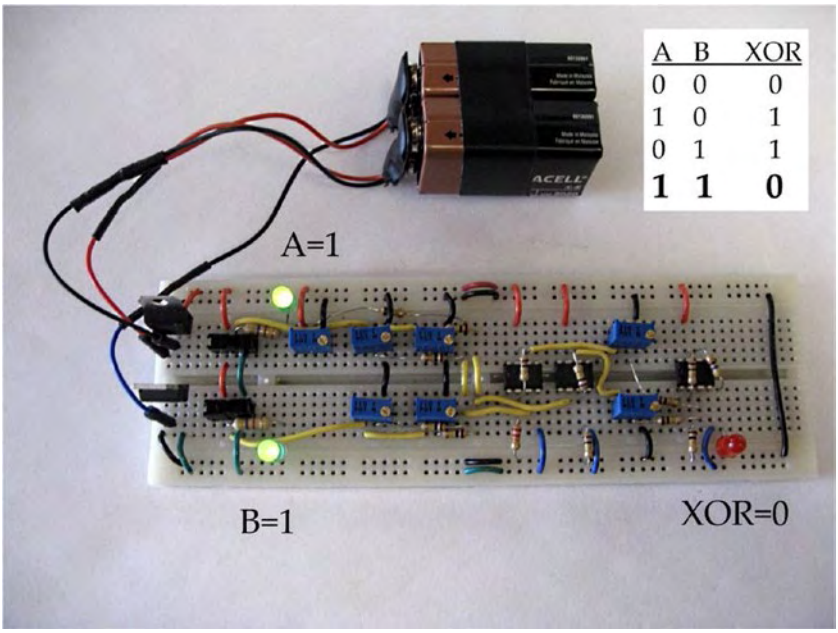


Figure 5-5. *Trained XOR network: 110*

Implementing the Back Propagation Algorithm

Here's a summary of back propagation:

1. If it works, don't mess with it.
2. If it doesn't work, make some adjustments.

So, the algorithm we are going to use says the following:

1. If a signal contributes to a correct answer, it should be strengthened.
2. If a signal tends to cause a wrong answer, it should be weakened.

3. If the output is correct, leave it alone. It's kind of like the old saying, "If it works, don't fix it."
4. If we want to turn Z ON, we want X and Y ON. If we want to turn Z OFF, we want X and Y OFF. (Keep in mind that we are using the inverting input for Y.)

KEEP IT SIMPLE!

Technically, we could go crazy computing all kinds of equations and using the "C word" (calculus), but forget all that! We are going to use a simple process based on the back propagation algorithm.

We are going to use a 0.2V correction factor when the output is wrong and just leave it alone if it is working. If some of the weights are quite far from the ideal solution, a 0.2V correction may take awhile, but let's start with a conservative learning rate.

How do we strengthen or weaken a signal? We adjust the potentiometers to include more or less resistance in its path. Looking back at Figure 5-1, we can see that moving the wiper of a potentiometer toward the top (toward the signal driving it) will reduce the amount of resistance in the path, therefore strengthening it. Moving the wiper down will introduce more resistance and weaken the effect of the signal. As we adjust the weight, the voltmeter will show the voltage increasing or decreasing.

So that we are not too aggressive and overcorrect, we will use an adjustment of 0.2V. If we used a larger correction factor, we would have a faster learning rate, but if we overcorrect, the network might not train at all! In other words, we might not be able to find the happy mixture that satisfies all four combinations of input values for A and B.

This will make a lot more sense as you start working your way through the training cycles and see the effect you are having on the numbers.

Training Cycles

Tables 5-1 through 5-6 summarize the training cycles or "epochs" required to train the network starting with randomized weights. Each training cycle consists of presenting the network with the four different combinations of inputs A and B, and then using a 0.2V correction factor to adjust the weights in the following order:

1. VB, V1, and V2
2. V3 and V4
3. V7 and V8

Each weight will affect the overall performance of the network, so it may take a few training cycles before the network performs satisfactorily. During each training cycle, I will make some observations to help us understand what is happening as the network learns. Make sure you do the same. Making some comments adds a lot to the experience and your understanding.

During training, here are a few things to keep in mind:

1. The threshold voltage for all neurons is 0.9V. In order to turn a neuron ON, the activation voltage must be greater than 0.9V. This will tell you which direction to adjust the signals feeding that neuron.
2. The values in the table for the neurons X, Y, and Z are logic values, not the actual voltage on their outputs (pin 6). 0 = OFF and 1 = ON.
3. The cells in the table that are shaded are the weights that we can adjust. All of the others are recorded just to help us understand how the training is coming along.

Note For your convenience, I have included a blank Training Table as a Microsoft Word document on my web site at rickmckeon.com/neural.html. Also, I have included a blank table in this book. Feel free to photocopy it.

OK, let's do some training. For this first training session, I am going to start with random values but will keep them fairly close to what I know works. In the next session we will get a little more adventurous. Keep in mind that my voltage values might be a little bit different from yours depending on the exact value of your power supply rails and resistor tolerances.

Table 5-1. *Details for Network #1, Training Cycle #1*

Network #1, Training Cycle #1												
A=0, B=0												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
1	1.23	-0.50	3.00	0	0.41	3.75	-0.50	-2.00	1	-1.86	-1.50	-2.25
Comments:												
1. A quick check of all four input combinations reveals that A=0, B=0 is the only input combination that gives an incorrect output. So we will adjust weights by 0.2V to move the network closer to turning Z OFF. 2. First, we adjust VB, V1, and V2 (VB goes to 3.55V, V1 goes to -0.70V, and V2 goes to -2.20V). 3. Next, we adjust V3 and V4 to get closer to turning Y OFF. This means increasing their voltage. (V3 goes to -1.30V and V4 goes to -2.05V). 4. Then, we adjust V7 and V8 (V7 goes to -0.70V and V8 goes to 2.80V). Z is still ON but V9 had dropped to 1.03V. That's getting pretty close to the threshold of 0.9V. 5. A check of the other input combinations reveals that they are all still working properly. So for the next training cycle, we will start with A=0, B=0.												
Adjusted to:												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
1	1.03	-0.70	2.80	0	0.20	3.55	-0.70	-2.20	1	-1.67	-1.30	-2.05

Table 5-2. *Details for Network #1, Training Cycle #2*

Network #1, Training Cycle #2

A=0, B=0

Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
1	1.03	-0.70	2.80	0	0.20	3.75	-0.70	-2.20	1	-1.67	-1.30	-2.05

Comments:

1. Z is ON so we are going to adjust weights to move us closer to the 0.9V threshold that will turn Z OFF.
2. We start by adjusting VB, V1, and V2 (VB goes to 3.55V, V1 goes to -0.90V, and V2 goes to -2.40V).
3. Next, we adjust V3 and V4 (V3 goes to -1.10V and V4 goes to -1.85V).
4. Finally, we adjust V7 and V8 (V7 goes to -0.90V and V8 goes to 2.60V). Z turned off while adjusting V8.
5. A check of the other input combinations reveals that now A=1, B=0 is not producing the right output, so we will go next to A=1, B=0 to continue training.

Adjusted to:

Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
0	0.84	-0.90	2.60	0	0.07	3.55	-0.90	-2.40	1	-1.46	-1.10	-1.85

A=1, B=0

Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
0	0.84	-0.90	2.60	0	0.78	3.63	1.01	-2.25	1	-0.42	-1.10	-1.85

Comments:

1. First, we adjust VB, V1, and V2 (VB goes to 3.83V, V1 goes to 1.21V, and V2 goes to -2.05V). While adjusting V1, Z turned ON.
2. A quick check of the other input combinations reveals that the network is trained!
3. Next follows a summary of all parameters for the four input combinations.

(*continued*)

Table 5-2. (*continued*)

Network #1, Training Cycle #2

Adjusted to:

Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
						3.83	1.21					

Summary of Parameters for Trained Network #1

A=0, B=0

Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
0	0.83	-0.91	2.60	0	0.08	3.74	-1.06	-2.40	1	-1.46	-1.08	-1.85

A=1, B=0

Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
1	1.98	1.15	2.83	1	0.92	3.84	1.21	-2.24	1	-0.43	0.80	-1.66

A=0, B=1

Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
1	2.09	1.34	2.84	1	2.00	3.97	-0.74	2.77	1	0.39	-0.83	1.61

A=1, B=1

Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
0	-0.88	0.88	-2.66	1	2.85	4.07	1.53	2.95	0	1.43	1.07	1.81

OK, this is pretty strange. I started with the values shown in Table 5-3 thinking they were random and different from any I had used before. Well, a quick check revealed the network performed as if it had already been trained on the XOR function! Just a lucky guess!

Table 5-3. *Random Weights Perform Like a Trained Network*

Network #2 Initial Values for A=0, B=0												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
		-1.50	2.50			3.00	-1.50	-1.00			-3.00	-2.00

Table 5-4 is a summary of parameters for "trained" network #2.

Table 5-4. *Summary of Parameters for "Trained" Network #2*

Summary of Parameters for Trained Network #2												
A=0, B=0												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
0	0.50	-1.50	2.50	0	0.15	3.00	-1.50	-1.00	1	-2.49	-3.00	-2.00
A=1, B=0												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
1	2.45	2.03	2.89	1	1.38	3.21	1.77	-0.81	1	0.57	2.57	-1.42
A=0, B=1												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
1	2.45	2.03	2.89	1	0.99	3.14	-1.33	1.18	1	-0.63	-2.64	1.37
A=1, B=1												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
0	-0.55	1.45	-2.56	1	2.22	3.35	1.94	1.38	0	2.44	2.94	1.96

Tables 5-5 and 5-6 summarize the adjustments necessary to train network #3.

Table 5-5. *Details for Network #3, Training Cycle #1*

Network #3, Training Cycle #1												
A=0, B=0												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
0	0.61	-2.00	3.25	0	0.64	2.50	-1.50	-2.50	1	-0,16	-2.00	-2.25
Comments:												
1. A quick check reveals that A=1, B=0 gives an incorrect result, but the other input combinations work. So we will start training with A=1, B=0.												
A=1, B=0												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
0	0.61	-2.00	3.25	0	0.64	2.73	1.50	-2.27	1	-0.16	1.53	-1.86
Comments:												
1. Z is OFF but V9 is at 0.61V. That's not too far away from the threshold of 0.9V. We will start by adjusting VB, V1, and V2 (VB goes to 2.93V, V1 goes to 1.70V, and V2 goes to -2.07V). 2. Even though Y is already ON, we will stick with the rule and strengthen lower V3 and V4 (V3 goes to 1.33V and V4 goes to -2.06V). 3. Next we adjust V7 and V8 (V7 goes to -1.80V and V8 goes to 3.45V). Z did not turn ON with these adjustments, so we will need at least one more training cycle. 4. A check of the other input combinations reveals that they work fine, so we will go directly to A=1, B=0 for the next pass.												
Adjusted to:												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
0	0.83	-1.80	3.45	0	0.85	2.93	1.70	-2.07	1	-0.37	1.33	-2.06

Table 5-6. *Details for Network #3, Training Cycle #2*

Network #3, Training Cycle #2

A=1, B=0

Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
0	0.83	-1.80	3.45	0	0.85	2.93	1.70	-2.07	1	-0.37	1.33	-2.06

Comments:

1. Z is OFF, so we start by adjusting VB, V1, and V2 (VB goes to 3.13V, V1 goes to 1.90V, and V2 goes to -1.87V).
2. While adjusting VB, neuron Z came ON. A quick check of all input combinations revealed that the network is trained!

Adjusted to:

Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
						3.13						

Summary of Parameters for Trained Network #3

A=0, B=0

Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
0	0.83	-1.77	3.46	0	-0.35	2.92	-1.64	-2.32	1	-2.09	-1.80	-2.39

A=1, B=0

Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
1	2.94	2.12	3.79	1	0.91	3.14	1.73	-2.08	1	-0.38	1.31	-2.08

A=0, B=1

Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
1	3.17	2.54	3.82	1	1.49	3.24	-1.28	2.51	1	0.31	-1.35	2.00

A=1, B=1

Z	V0	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
0	-0.91	1.72	-3.56	1	2.77	3.46	2.10	2.78	0	2.04	1.79	2.34

Table 5-7 is a blank training sheet for your use.

Table 5-7. *Blank Training Table*

Training Cycle #												
A=0, B=0												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
Comments:												
Adjusted to:												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
A=1, B=0												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
Comments:												
Adjusted to:												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
A=0, B=1												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
Comments:												

(*continued*)

Table 5-7. (*continued*)

Training Cycle #												
Adjusted to:												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
A=1, B=1												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
Comments:												

Adjusted to:												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4

Convergence

What is convergence? It's when all of the different weights converge or get to a place where the network starts to give the right answer every time for all possible inputs. This is the happy place!

As you work your way through the training cycles, you can see this happening. This is why it's good to make lots of comments in the training table. You can write down things that you think might be happening, especially when you think the next training cycle might get you there. This is part of becoming familiar with the process! It's your table and it's your network, so have some fun and write lots of comments. You can always clean it up later. This is a learning process: for the network and for you!

Attractors and Trends

Now let's look at the activation voltages for our trained networks and see if we can notice any trends.

What Is an Attractor?

Without getting too technical, let me just say that an attractor is a set of values that a system seems to be attracted to. It's like there is some unseen force that is pulling the parameters toward those certain values. If you drop a glass of water, what happens? Probably it will crash to the floor, making a big mess. In this case, gravity is the attractor and the big mess is the destination.

If you look at a topo map, you will observe watershed areas where the raindrops flow down into different streams and lakes. Here again, gravity is the attracting force and the lakes are the destinations. This discussion could get a lot more complex with "local minimums," "global minimums," and so on, but let's leave it at that for now.

Here's a little more technical example. Sometimes we solve equations by guessing at the answer and then working an algorithm like the Newton Method over and over again to give us successive approximations that get closer and closer to the actual answer. If you were to see this process displayed as a colored graph, you would see the plot attracted toward the answer (or answers). Figure 5-6 is a plot using this method to solve a third-degree equation. OK, I said this was a little more technical.

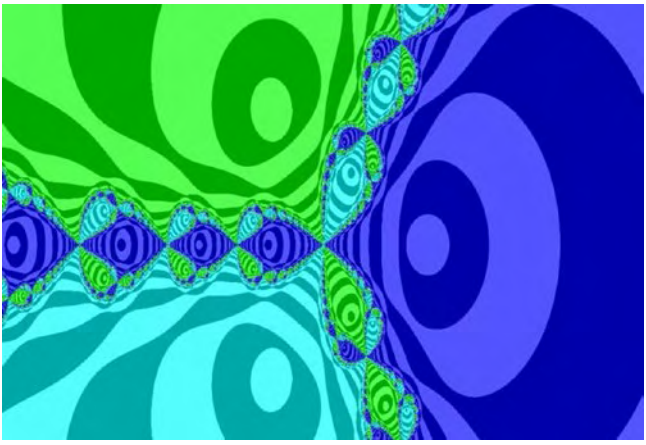


Figure 5-6. *Attractors for a third-degree equation*

To solve the XOR problem, each of our activation voltages appear to be attracted to a small range of values. When starting with different initial values, the activation voltages may be somewhat different from each other, but we can spot a trend. What produces this strange force? I think it has to do with the architecture of our network and the problem we are asking it to solve. If we were training on a different logic function, we would see different attractors.

By the way, that would be another great project! Train this same network on the OR function or the AND function and see where the activation voltages go. In Appendix A, we will investigate other common logic functions, and we'll even train on some arbitrary patterns. You may be surprised what the activation values have to show!

WARNING!

If you get interested in dynamical systems and attractors, you will end up spending an inordinate amount of time and energy pursuing that passion!

Attractors in Our Trained Networks

With just a quick look at Figures 5-7 through 5-9 you will see some overall patterns. It doesn't take much to do a little bit of "feature extraction."

Each figure shows the activation voltages for neurons X, Y, and Z for the four different combinations of input values.

V9 is the "biggie" because it is the signal that turns neuron Z ON or OFF. The activation threshold for each neuron is 0.9V. In the charts, it is just below the 1V line. Notice that V9 is below the threshold for AB=00 and AB=11. In each case for AB=00, it is just barely below the threshold, and for AB=11 it is quite a bit below. Where does that come from? How does it happen? Now, I was the one operating the screwdriver, but I was just blindly following the back propagation algorithm.

From the graphs (and also from the tables above them) you can see whether X and Y are turned ON or OFF.

In all three figures, you will notice that V5 and V6 rise from AB=00 to AB=11. This is a distinct trend, again based on the training algorithm. For input AB=11, X is ON and Y is OFF (we are using the inverting input for Y). It appears that Y has much more influence than X. Where does that come from? I would say from the ratio of V7 and V8. If you look at V7 and V8 in the training tables, you will see that V8 is always quite a bit larger on the negative side than V7 is on the positive side for this input.

Now that you have trained the network several times, you may have some ideas to share. I would love to hear your insights. Send me an email at `rmckeon5@gmail.com`.

	AB=00	AB=10	AB=01	AB=11
V5	0.08	0.92	2.00	2.85
V6	-1.46	-0.43	0.39	1.43
V9	0.83	1.98	2.09	-0.88

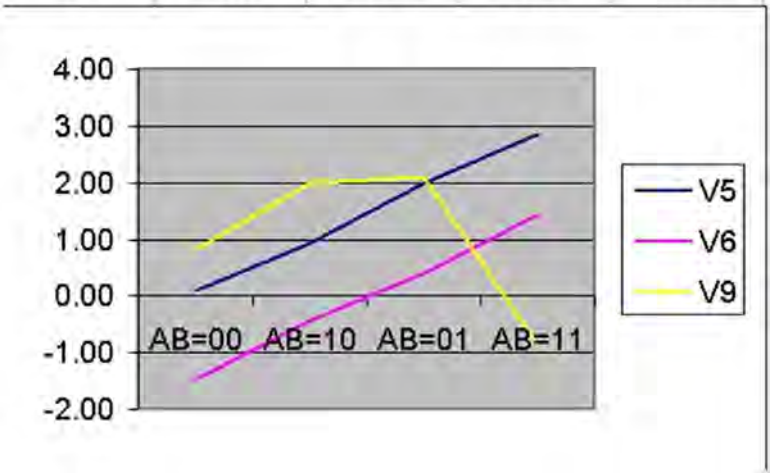


Figure 5-7. *Activation values for network #1*

	AB=00	AB=10	AB=01	AB=11
V5	0.15	1.38	0.99	2.22
V6	-2.49	0.57	-0.63	2.44
V9	0.50	2.45	2.45	-0.55

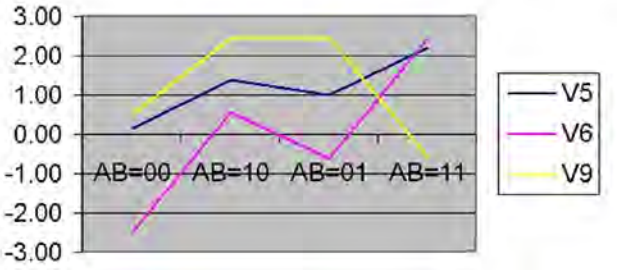


Figure 5-8. *Activation values for network #2*

	AB=00	AB=10	AB=01	AB=11
V5	-0.35	0.91	1.49	2.77
V6	-2.09	-0.38	0.31	2.04
V9	0.83	2.94	3.17	-0.91

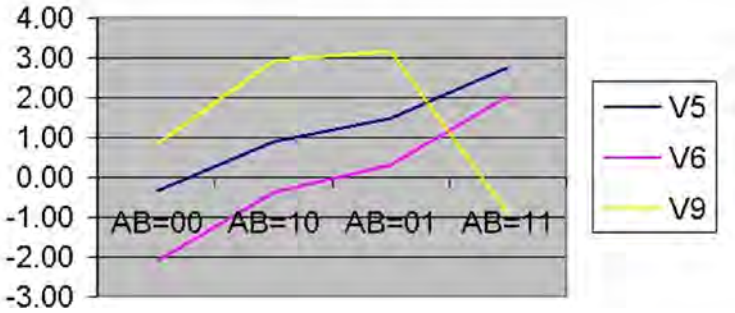


Figure 5-9. *Activation values for network #3*

Implementation

When you implement a neural network, you take a trained network and do one of the following:

1. Compile the program into machine language and download it to a microcontroller unit (MCU) that can reside in a product.
2. Burn the trained network into a neural network chip that is just one component of the product.
3. Run the trained network on a large computer system.

In any case, the training is over and now it doesn't need to learn any more. It just performs the function it has been trained to do.

The next step in the evolution of neural networks is for them to become more like human beings. Think about it. We perform the functions that we already know how to do but we also continue to learn new things at the same time.

Of course we would not take the trained network that we build for this project and implement it in hardware. The CA4070 integrated circuit has four XOR gates on board and it only cost about $0.21. But many other sophisticated functions have been implemented as neural networks in real products.

Here's the interesting thing about implementation. There are thousands of neural networks out there performing important tasks of pattern recognition in every environment imaginable! They are being used in medical applications, security systems, trend forecasting, and even in stock market predictions. These networks need to be accurate and cost-effective.

Accurate means well trained, and cost-effective means that we need to reduce the support costs. In other words, once a network has been trained to perform a specific function, the hardware and software costs can be minimized.

The neural network may have been trained by running software on a PC or larger computer, but once trained, the runtime program can be

compiled and downloaded to a small, inexpensive microcontroller unit. The computer that the neural network was developed on does not need to be part of the product.

Hardware-based neural networks are slow in coming, but we are seeing some progress in that arena. Hopefully, we will start to see neural network chips and development systems become commonplace soon. When that happens, the same argument will apply. The trained neural network chip will become just another component in the product. The development system can be reused to develop other applications but will not need to be part of the product itself. How exciting is that? And right now in the year 2018, we are witnessing an explosion of concepts and products in this exciting field. Now, that's something to be excited about!

Summary

What an adventure!

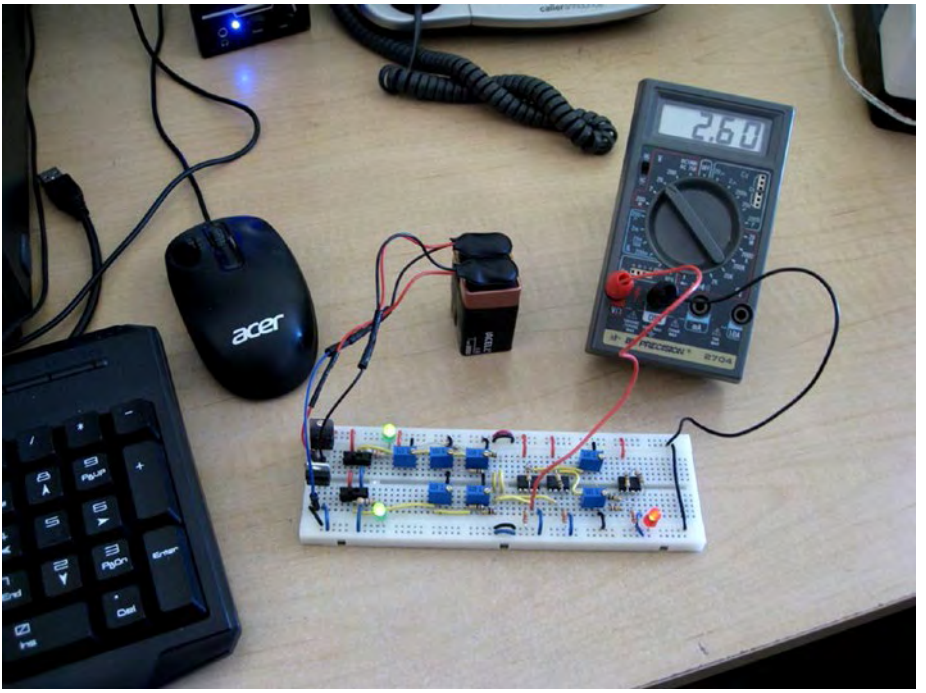
I hope you have found the training process for your network to be interesting and even fascinating. Maybe some things worked out differently than you expected.

I know all the details of the process might have seemed tedious at first, but we have immersed ourselves in the training a network. You have been through the process at the nitty-gritty level. I'm telling you, when we start to see neural network chips become commonplace, you will have a jumpstart in understanding.

In Chapter 6 we will reflect on where we have come from, and some of the exciting possibilities for future projects. In fact, we will demonstrate that this network is really a general purpose machine by training on some other patterns. What's the difference between a known and valued logic function like AND or NAND and some arbitrary set of input and output values? Well, really not all that much. Maybe you will discover an application for some arbitrary transformation, who knows! If so, I would love to hear about your discoveries.

CHAPTER 6

Training on Other Functions



R. McKeon, *Neural Networks for Electronics Hobbyists*,
https://doi.org/10.1007/978-1-4842-3507-2_6

We have seen how our three-layer network can perform the XOR function. In this chapter, we will train the same network to perform a few other tasks of pattern recognition by simply adjusting its connection weights. Figure 6-1 shows the truth tables for several important logic functions. Let's try training our network on a few of them.

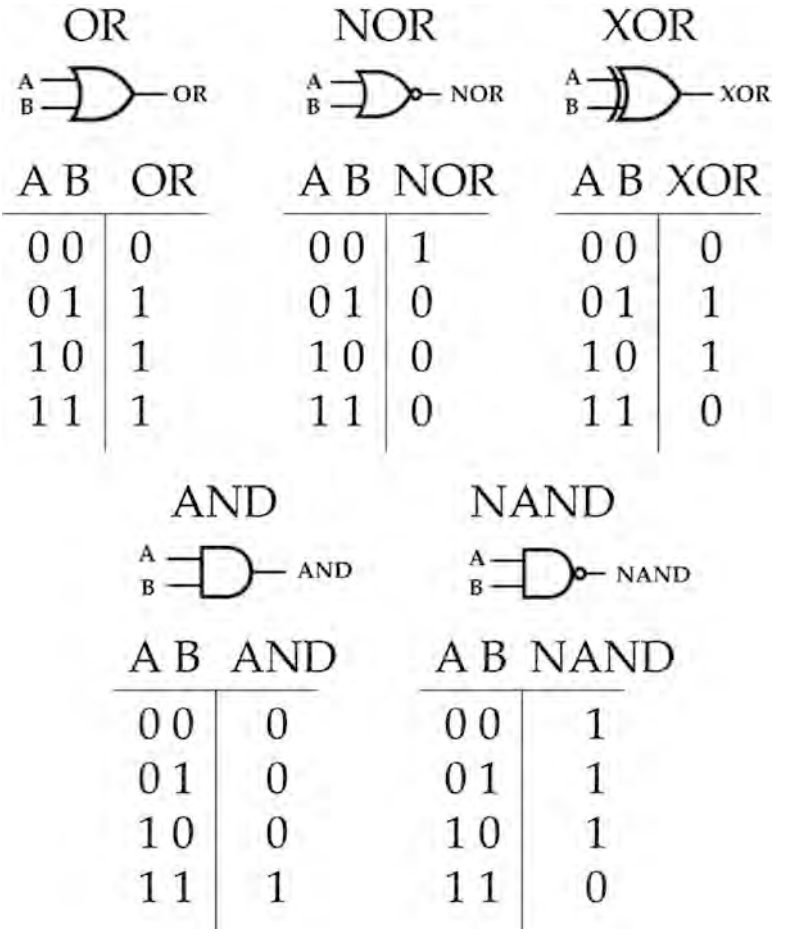


Figure 6-1. *Common logic functions*

The OR Function

The OR function is similar to the XOR function except that it produces a logic 1 or true in the case where both inputs are true. Because it includes this possibility, we call it the "inclusive OR." The XOR function excludes this possibility, so it is called the "exclusive OR."

Since the network is already trained on XOR, let's start there. We will follow the back propagation algorithm and train it to recognize the OR function. We'll use a 0.2V correction factor and do the following:

1. If a signal contributes to a correct answer, it should be strengthened.
2. If a signal tends to cause a wrong answer, it should be weakened.
3. If the output is correct, leave it alone. It's kind of like the old saying, "If it works, don't fix it."

Each training cycle consists of presenting the network with the four different combinations of inputs A and B, and then using a 0.2V correction factor to adjust the weights.

Table 6-1 summarizes the details of the training process for the OR function. Remember, it's good to make lots of notes and try to understand how the network is learning, because we are starting with the network already trained on XOR we will go directly to A=1, B=1 to start the training.

Table 6-1. *Training on the OR Function*

Training on the OR Function												
A=1, B=1												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
0	-0.88	0.87	-2.66	1	2.84	4.07	1.53	2.94	0	1.42	1.06	1.81
Comments:												
1. X is already ON, so we will strengthen its output by increasing V7 by 0.2V to 1.07V. 2. Y is LOW (keep in mind that Y is an inhibitory neuron), so we want to get its input signal V6 closer to the threshold of 0.9V. Therefore, we will adjust V3 down to 0.86V and V4 down to 1.61V. This brought V6 down to 1.22V. 3. Y is still LOW, so we will bring V8 up to -2.46. Adjusting V7 and V8 brought V9 up to -0.68V. 4. A check of the other input switch combinations reveals that they all still produce correct outputs for the OR function, so we will continue training with A=1, B=1.												
Adjusted to:												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
0	-0.68	1.07	-2.46	1	2.84	4.07	1.53	2.94	0	1.22	0.86	1.61
A=1, B=1												
1. X is ON, so we will strengthen its output by increasing V7 to 1.27V. 2. Y is LOW, so we want to get his input signal closer to the threshold of 0.9V. Therefore we will adjust V3 down to 0.66V and V4 down to 1.41V. This brought V6 down to 1.02V. (We're getting closer to the 0.9V threshold that will cause Y to go HIGH.) 3. Z is still LOW, so we will bring V8 up to -2.26V. Adjusting V7 and V8 brought V9 up to -0.48V. 4. A check of the other input switch combinations reveals that they are still producing the correct outputs, so we will continue with A=1, B=1.												

(*continued*)

Table 6-1. (*continued*)

Training on the OR Function

Adjusted to:

Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
0	-0.48	1.27	-2.26	1	2.84	4.07	1.53	2.94	0	1.02	0.66	1.41

A=1, B=1

1. X is ON, so we will strengthen its output by increasing V7 to 1.47V.
2. Y is LOW, so we want to get its input signal closer to the threshold of 0.9V. Therefore we will adjust V3 down to 0.46V and V4 down to 1.21V. This brought V6 down to 0.83V (below the threshold) and neuron Y went HIGH, causing V9 to jump to 2.27V, turning the output LED on.
3. A check of the other input switch combinations reveals that the network is trained on the OR function!

Adjusted to:

Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
1	2.27	1.47	2.59	1	2.84	4.07	1.53	2.94	1	0.83	0.46	1.21

It's interesting to note that throughout the training process, we never adjusted RB, R1, or R2. Therefore, the inputs to neuron X remained the same and it stayed ON throughout the training. Following the back propagation protocol, we did strengthen the output from X on each pass, but the major trigger event happened as we were adjusting R4. This brought V6 down below the threshold for neuron Y (inhibitory neuron) and allowed its output to go HIGH.

At that point, a quick check of all the input switch combinations revealed that the network was trained. With a correction value of 0.2V, it only took three passes to achieve convergence.

Table 6-2 summarizes the parameters for the network trained on the OR function.

Table 6-2. *Summary of Parameters for the Network Trained on the OR Function*

Summary of Parameters for the OR Function												
A=0, B=0												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
0	0.33	-1.51	2.18	0	0.09	3.74	-1.06	-2.40	1	-0.84	-0.47	-1.23
A=0, B=1												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
1	2.27	1.98	2.59	1	1.99	3.95	-0.74	2.77	1	0.38	-0.37	1.13
A=1, B=0												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
1	2.27	1.98	2.55	1	0.91	3.82	1.21	-2.24	1	-0.40	0.36	-1.17
A=1, B=1												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
1	2.27	1.98	2.59	1	2.84	4.07	1.53	2.94	1	0.83	0.46	1.21

Figure 6-2 shows how the activation values V5, V6, and V9 changed during training.

	AB=00	AB=01	AB=10	AB=11
V5	0.09	1.99	0.91	2.84
V6	-0.84	0.38	-0.40	0.83
V9	0.33	2.27	2.27	2.27

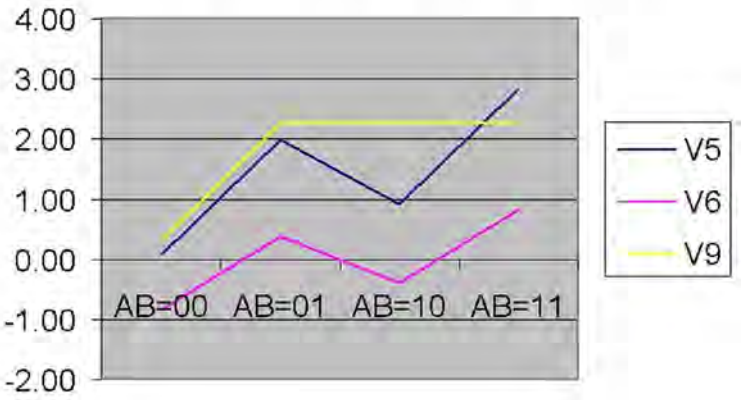


Figure 6-2. *Activation values during training on OR*

Let's try the AND function next.

The AND Function

The AND function requires that both inputs be true in order for the output to be true. In the truth table, you will notice that this occurs only once for the four combinations of inputs. Just to randomize things a little bit, we will start with all of the potentiometers somewhere near their center position. That is what gives us the initial conditions shown in Table 6-3. Also, just by accident, starting with those initial conditions the network performs like it was trained on the XOR function. Therefore, we will start training with A=0, B=1.

Table 6-3. *Training on the AND Function*

Training on the AND Function												
A=0, B=1												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
1	3.02	3.02	3.03	1	1.04	2.70	-1.88	2.22	1	-0.18	-2.15	1.99
Comments:												
1. Z is ON, but for this input we need it to be OFF. Outputs for both X and Y are high, so we will start by bringing VB down to 2.50V, V1 down to -2.08V, and V2 down to 2.02V. Adjusting these three weights brought V5 down to 0.80V and the output LED turned OFF. 2. A=1, B=0 still gives a 1 on the output, so we'll go there next.												
Adjusted to:												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
0	-0.04	-2.50	2.42	0	0.81	2.50	-2.08	2.02	1	-0.07	-2.15	1.99
A=1, B=0												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
1	3.02	3.03	3.04	1	1.11	2.55	2.44	-1.66	1	0.09	2.23	-2.05
Comments:												
1. Z is ON but we need it to be OFF, Outputs for both X and Y are high, so we will start by bringing VB down to 2.35V, V1 down to 2.24V, and V2 down to -1.86V. Adjusting these three weights brought V5 down to 0.89V, which turned X OFF, and the output LED turned OFF. 2. Now A=0, B=0, A=0, B=1, and A=1, B=0 all give correct outputs. Next we will go to A=1, B=1.												

(*continued*)

Table 6-3. (*continued*)

Training on the AND Function

Adjusted to:

Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
0	-0.03	-2.50	2,42	0	0.89	2.35	2.24	-1.89	1	0.01	2.07	-2.05

A=1, B=1

Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
0	-0.03	2.43	-2.49	1	2.61	2.71	2.59	2.56	0	2.53	2.57	2.51

Comments:

1. Z is OFF, but we want it ON. V9=-0.03V, which is not too far from the 0.9V threshold required to turn it ON.
2. X is already ON, so we will strengthen its output by raising V7 to 2.63V. This brought V9 up to 0.08V (closer to the required threshold of 0.9V).
3. Y is LOW, so we will adjust V3 to 2.37V and V4 to 2.31V. This won't change V6 all that much, but we are just following the back propagation algorithm. These adjustments brought V6 down to 2.32V. Also, we reduced V8 to -2.29V, and that brought V9 up to 0.18V.
4. The other input combinations still give correct outputs, so we will continue training on A=1, B=1.
5. Training is going slowly, so let's try a correction factor of 0.4V.

Adjusted to:

Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
0	0.18	2.63	-2.29	1	2.61	2.69	2.59	2.56	0	2.32	2.37	2.31

A=1, B=1

Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
0	0.18	2.63	-2.29	1	2.61	2.69	2.59	2.56	0	2.32	2.37	2.31

(*continued*)

Table 6-3. (*continued*)

Training on the AND Function												
Comments:												
1. Z is still OFF, but we want it ON. V9=0.18V, which is not very far from the 0.9V threshold required to turn it ON. 2. X is already ON so we will strengthen its output by raising V7 to 3.03V. This brought V9 up to 0.48V (closer to the required threshold of 0.9V.0). 3. Y is LOW, so we will adjust V3 to 1.97V and V4 to 1.91V. This won't change V6 all that much, but we are just following the back propagation algorithm. These adjustments brought V6 down to 1.93V. Also, we reduced V8 to -1.89V, and that brought V9 up to 0.58V. 4. A quick check of the other input combinations reveals that they are still producing the correct OFF output, so we will continue training A=1, B=1 with a 0.4V correction factor.												
Adjusted to:												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
0	0.57	3.03	-1.89	1	2.61	3.44	-1.01	2.34	0	1.93	1.97	1.91
A=1, B=1												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
0	0.57	3.03	-1.89	1	2.61	3.44	-1.01	2.34	0	1.93	1.97	1.91

(*continued*)

Table 6-3. (*continued*)

Training on the AND Function

Comments:

1. Z is still OFF, but we want it ON. V9=0.57V, which is pretty close to the 0.9V threshold required to turn it ON.
2. X is already ON, so we will strengthen its output by raising V7 to 3.43V. This brought V9 up to 0.78V (closer to the required threshold of 0.9V).
3. Y is LOW, so we will adjust V3 to 1.57V and V4 to 1.51V. These adjustments brought V6 down to 1.51V. Also, we reduced V8 to -1.45V. When we reached -1.61V on V8, the output LED came ON. A quick check of all input switch combinations revealed that the network was trained on the AND function! Wow! It took several training cycles, but the network did eventually learn to produce the AND function!

Adjusted to:

Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
1	0.90	3.43	-1.45	1	2.61	2.69	2.59	2.56	0	1.51	1.57	1.51

Note After a few passes it looked like we were training pretty slowly, so I got frustrated (impatient) and bumped the correction factor up to 0.4V. After that, it only took two more passes for the network to become trained.

An interesting point is that once the network became trained on the AND function, neuron Y was still producing a LOW output. (One of my secret goals was to get it to produce a HIGH output so it would turn on neuron Z, but I was wrong! The network knew how to solve the problem when I didn't). So one of the amazing things about neural networks is that the knowledge is distributed throughout the network. This is also one of the properties that makes them so robust!

Table 6-4 summarizes the parameters for the network trained on the AND function.

Table 6-4. *Summary of Parameters for the Network Trained on the AND Function*

Summary of Parameters for the AND Function												
A=0, B=0												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
0	-0.99	-3.54	1.54	0	-0.83	1.98	-2.25	-2.23	1	-1.54	-1.56	-1.54
A=0, B=1												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
0	-0.99	-3.54	1.54	0	0.88	2.33	-1.90	2.19	1	-0.03	-1.30	1.23
A=1, B=0												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
0	-0.99	-3.54	1.54	0	0.88	2.33	2.22	-1.87	1	0.00	1.26	-1.28
A=1, B=1												
Z	V9	V7	V8	X	V5	VB	V1	V2	Y	V6	V3	V4
1	0.89	3.45	-1.61	1	2.60	2.68	2.59	2.55	0	1.51	1.53	1.51

Figure 6-3 shows how the activation values V5, V6, and V9 changed during training.

	AB=00	AB=01	AB=10	AB=11
V5	-0.83	0.88	0.88	2.60
V6	-1.54	-0.03	0.00	1.51
V9	-0.99	-0.99	-0.99	0.89

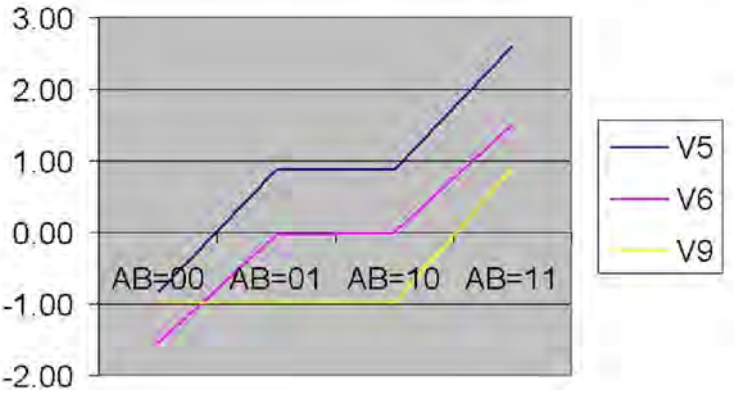


Figure 6-3. *Activation values during training on AND*

The General Purpose Machine

So far, we have seen that our network can be trained on XOR, OR, and AND. There are many more combinations of inputs and outputs that this network can associate. How many? Well, if you think about the different outputs as just a combination of four binary bits (1's and 0's), then the possible number of combinations is 16. Where does that come from? Table 6-5 shows the possible number of combinations of four binary bits together with their decimal equivalents.

Table 6-5. *Number of Possible Combinations of Four Binary Digits*

Binary Number	Decimal Equivalent	Number of Combinations
0000	0	1
0001	1	2
0010	2	3
0011	3	4
0100	4	5
0101	5	6
0110	6	7
0111	7	8
1000	8	9
1001	9	10
1010	10	11
1011	11	12
1100	12	13
1101	13	14
1110	14	15
1111	15	16

So what am I getting at? If this simple three-layer network really is a general purpose machine, it should be able to associate (solve) any of the 16 possible combinations. A few of them are well-known logic functions, but any one of them could turn out to be a useful association in real-life commercial products.

Summary

In this chapter, we have discovered that our simple three-layer network can perform many different functions depending on how it is trained. What we have built is a pretty simple network. How can we extend its functionality? By adding more inputs, outputs, and hidden layers, and also by implementing more sophisticated architectures (there are many!). I suspect that we will soon see an explosion of neural network hardware and development systems! Even as a hobbyist, you can take part in this revolution.

In Appendix A, we are going to have a look at a free, open source neural network software simulator called Simbrain. And then in Appendix B, I'll point you to some resources that will keep you motivated as you travel along this exciting path.

CHAPTER 7

Where Do We Go from Here?

I hope you have enjoyed this brief introduction to neural networks. If this book has sparked your interest, the journey is not over. It's just beginning! There are countless free online resources, and a quick search will reveal plenty of beginning-level (and advanced) books to purchase. In Appendix B, I list a few of my favorite resources. This exciting field could easily turn into a lifelong study. You will never run out of interesting books, articles, and videos because new advances are being made every day!

In this chapter, I am going to suggest a few more projects to experiment with using the network we have already constructed. In Appendix A, we'll have a look at a free software neural network simulator. In this book, we have focused on a hardware-based approach, but there are plenty of free, open source software programs available. The one we will look at is called Simbrain.

So, let's have a look at some of the other experiments you can perform with our current project.

Varying the Learning Rate

We used a 0.2V correction factor that resulted in trained networks with just a few training cycles. Could we do even better with a larger correction factor? (Actually, in Chapter 6, when I got impatient with our progress, I bumped the correction factor to 0.4V with good effect.)

R. McKeon, *Neural Networks for Electronics Hobbyists*,
https://doi.org/10.1007/978-1-4842-3507-2_7

You would think that a higher learning rate would train the network faster, but there is a chance of overcorrecting. In that case, the network might not converge at all.

Also, you would think that a lower learning rate would cause the network to be more stable and learn more reliably, but it might require a lot more training cycles to reach convergence.

Give some different correction factors a try and record your results. I would love to hear your observations. What did you learn about training?

Crazy Starting Values

We were pretty conservative in our choice of initial conditions. Can this network eventually learn to solve the XOR function no matter what the initial weights are?

An interesting experiment would to choose an "outlier" or a value that is really extreme and see if the process eventually brings it under control and trains properly. Maybe it would take a lot more training cycles, or maybe the attractors in the system aren't strong enough to overcome this value. The error in the system might get stuck in a "local minimum" instead of being eliminated altogether. Put it to a test and see what happens.

Apply the Back Propagation Rule Differently

Our application of back propagation used a pretty straightforward approach. Perhaps you may have some ideas about how to apply it differently or more effectively. Here are some things you might try:

1. Change the learning rate for each training cycle. Start with a large correction factor and reduce it with each training cycle.

2. Use a different correction factor for the hidden layer than you used for the output layer.
3. Try eliminating the bias voltage on neuron X and see if you can still get the network to learn.
4. Don't strengthen good signals; just reduce bad signals that contributed to a wrong answer.
5. Don't weaken bad signals; just strengthen good signals or signals that contributed to a correct output.
6. Does this sound a little bit like trying different approaches to child-rearing?

Feature Extraction

Can we learn something by looking at the weights in the hidden layer of a trained network? How about the path or "orbit" that the weights took while the network was being trained?

When we look for patterns in the hidden layer, we are doing "feature extraction." In other words, we are asking: "What do these features mean?"

Determining the Range of Values

As we saw in Figures 5-7 through 5-9, the range of weights and activation signals in a trained network can vary quite a bit depending on initial conditions. It appears that the important factor is their relationship to each other, not their actual value. Will these values always fall within a certain range? A neuron (op amp comparator) won't turn ON unless its activation signal is above the threshold, so that creates a built-in restriction.

Training on Different Logic Functions

There are many logic functions to investigate. Figure 7-1 shows some common ones. Use this same network to train on different functions. Can it learn any arbitrary combination of inputs vs. outputs?

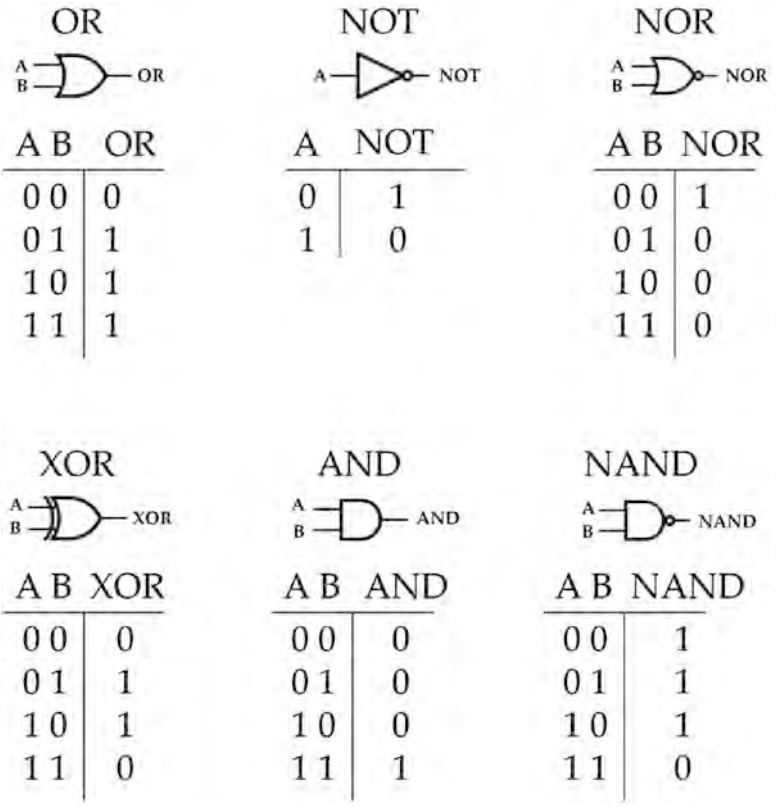


Figure 7-1. *Common logic functions*

As you train on these different functions, observe what is happening with the weights. Plot the activation signals for the four possible input combinations as we did for the XOR function.

Try Using a Different Model

There have been other models proposed for solving the XOR problem. One of them uses a single neuron in the hidden layer. The inputs are connected to the hidden layer and also to the output layer as shown in Figure 7-2. Can a three-layer network always solve the XOR function?

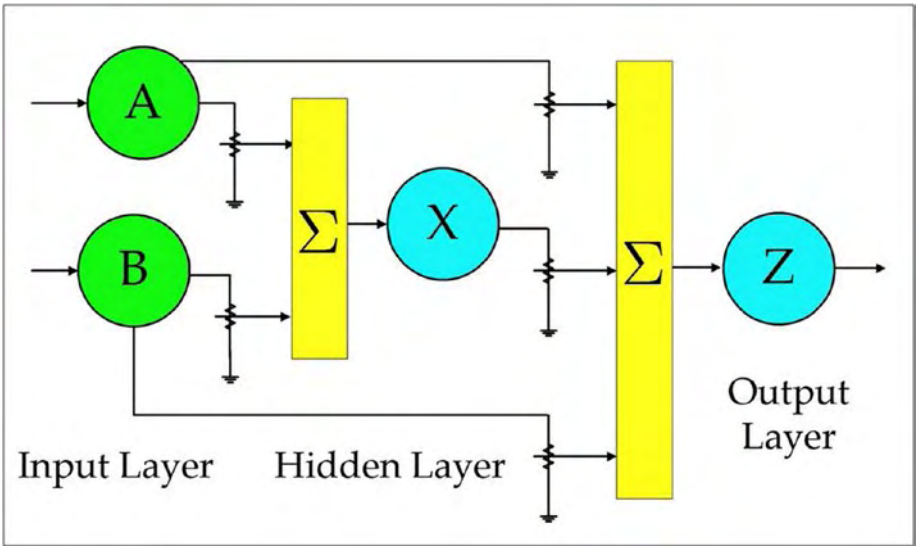


Figure 7-2. *Single neuron in the hidden layer*

Build a Neural Network to Do Other Things

If you have some experience with electronic projects, you may want to expand on this project. The possibilities are endless! Here are a few ideas:

1. Replace slide switches A and B with photo diodes and put a power transistor on the output to control a motor for clockwise and counterclockwise rotation.

2. Use the principles discussed here to build a network with a completely different architecture (different number of layers, different number of neurons in each layer, a more sophisticated task to solve).

Postscript

I'm sure you realize that the approach we have taken here is not the most efficient for simply designing a circuit to do something. This has been a tutorial and a project just to get you thinking about hardware-based approaches to neural networks. When neural network chips become readily available (which I'm sure they will), you will already have a lot of experience and some knowledge about how they work at a low level.

Note Low-level is not bad. It's good! Investigating things at the machine level gives you a deeper understanding.

It's like in the old days when we used to write programs by hand on coding sheets using op codes or assembler language. When higher-level languages and compilers came along, they were vastly more efficient, but I'm still glad for having learned about machine language (1's and 0's) written in hex, having to build address decoders with external hardware, understanding timing diagrams, using external EEPROMs to store the program, and stuff like that.

TALK TO ME

I would love to hear about your projects and what you are doing as "The Adventure Continues."

Send me an email at rmckeon5@gmail.com.

Summary

We started out discussing neural networks in general terms, and then we got pretty specific by building a three-layer network from simple parts. I hope our construction project helped to solidify some of the general concepts.

The adventure is not over. It can continue for as long as you want! This has been a brief introduction, but the journey has just begun. The adventure of advancing technology belongs to everyone, not just to a few select engineers. Who knows what interesting questions (or answers) you may come up with! My adventures in science and music have convinced me that "The Joy Is in the Journey." I took the photograph in Figure 7-3 on one of my hiking adventures in the Sierra Nevada Mountains of California. It depicts my view of life as an ongoing adventure.

Figure 7-3. *The joy is in the journey*

APPENDIX A

Neural Network Software, Simbrain

The focus throughout this book has been on a hardware (components)-based approach to neural networks, but there are many colorful and intuitive software packages available that you might find interesting and fun to experiment with. Also, unless you are looking for a high-end professional program, they are usually free or inexpensive.

Just learning to use one of these programs is an education in itself! You will become familiar with

1. Neural network terminology
2. Network models
3. Learning methods
4. Collecting and preparing data for analysis
5. Possible applications you might not have even thought of yet

A quick search on the Internet will yield lots of neural network simulators. Download a few of them (especially the free ones or ones that have a free trial period) and play around with them. If you find one that you don't like or the learning curve seems too steep, simply delete it from your computer. One that I think is worth spending some time with is called Simbrain.

R. McKeon, *Neural Networks for Electronics Hobbyists*,
https://doi.org/10.1007/978-1-4842-3507-2

Simbrain is a free, open source, neural network simulator. It is written in Java and runs on Windows, Mac OS X, and Linux. The interface is colorful, easy to use, and intuitive. Simbrain is supported by a large user community, and there are lots of tutorials available on the Simbrain web site and on YouTube. Their overarching design goal is to make neural networks available to as wide an audience as possible, and to educate the layperson about neural networks. Visit the Simbrain home page at `www.simbrain.net` to see what it's all about and download the program.

Note Simbrain is not perfect and it is still evolving, but it is a lot of fun to experiment with, and there is an active user community contributing to its development. I know you will learn a lot about neural networks simply by experimenting with its available network models and implementation strategies. Figure A-1 shows the Simbrain home page.

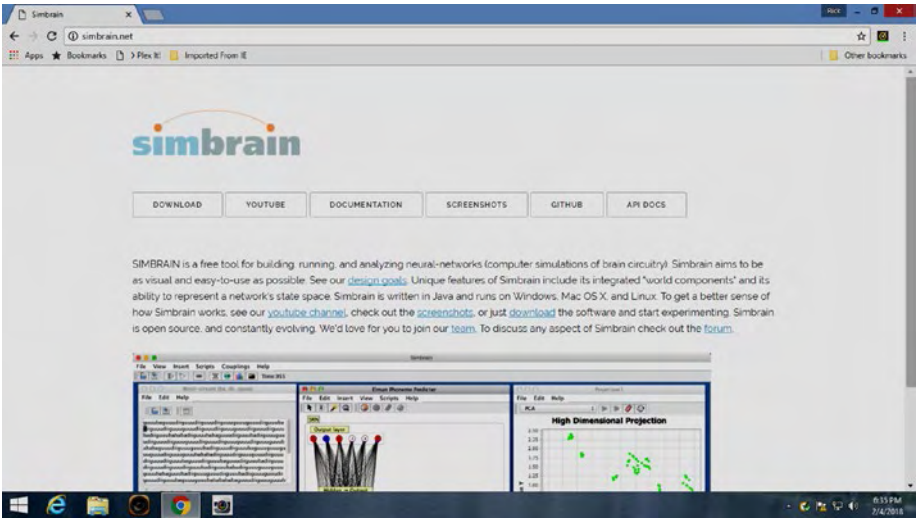


Figure A-1. *Simbrain home page*

OK, once you have downloaded and installed the program, let's build a simple network. We'll build a three-layer pattern-matching network with four inputs, five neurons in the hidden layer, and four output neurons. To get this project up and trained quickly, we will pretty much use the defaults for everything.

Table A-1 shows the training set. In each case the input pattern has only one neuron turned ON and it will be matched to an output pattern with a different neuron turned ON.

Table A-1. *Training Set*

Input Pattern	Output Pattern
1000	0001
0100	0010
0010	0100
0001	1000

Open Simbrain and click "New Network." Then click Insert/Insert Network/Backprop, as shown in Figure A-2.

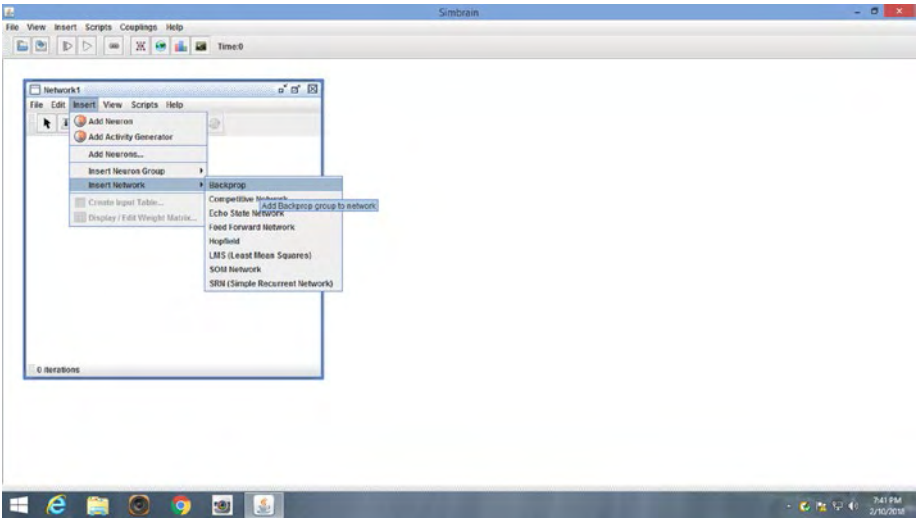


Figure A-2. *Creating a new back propagation network*

Once the network opens, change the number of inputs and outputs to four and just accept all other defaults (Figure A-3).

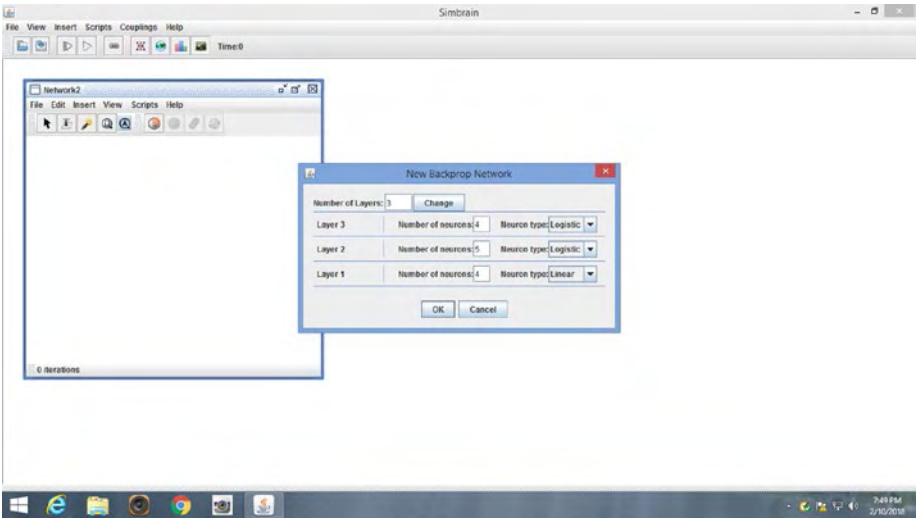


Figure A-3. *Specify the number of neurons in each layer*

Then, I like to drag and drop the neurons to show the network horizontally (just my preference). Don't worry about moving the neurons; they will stay in the appropriate layer and their connections will remain intact. I also like to double-click each neuron and give it a label. In this case, the label is simply the neuron number. Once you do that, your network should look like Figure A-4.

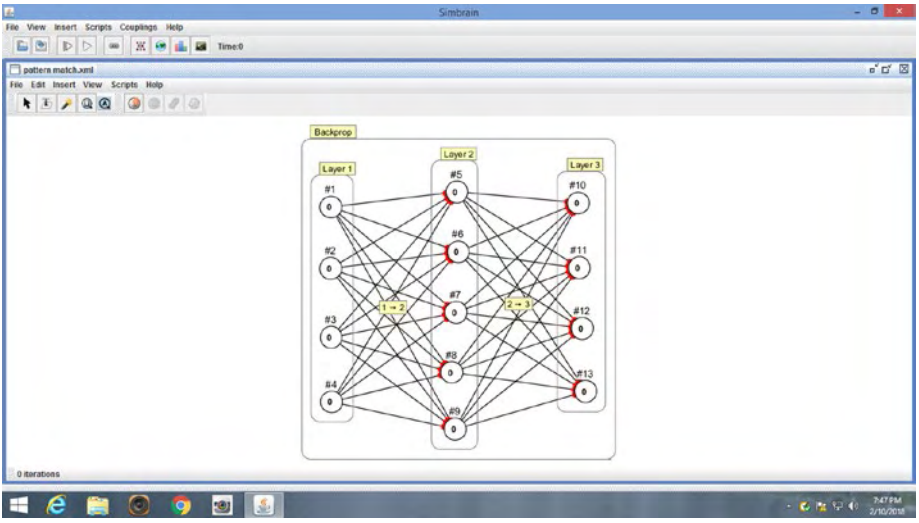


Figure A-4. *Network displayed horizontally from left to right*

Double-click the Backprop tab in the upper left-hand corner of the network to open the training dialog box. Under the Input data tab, enter the input data (Figure A-5).

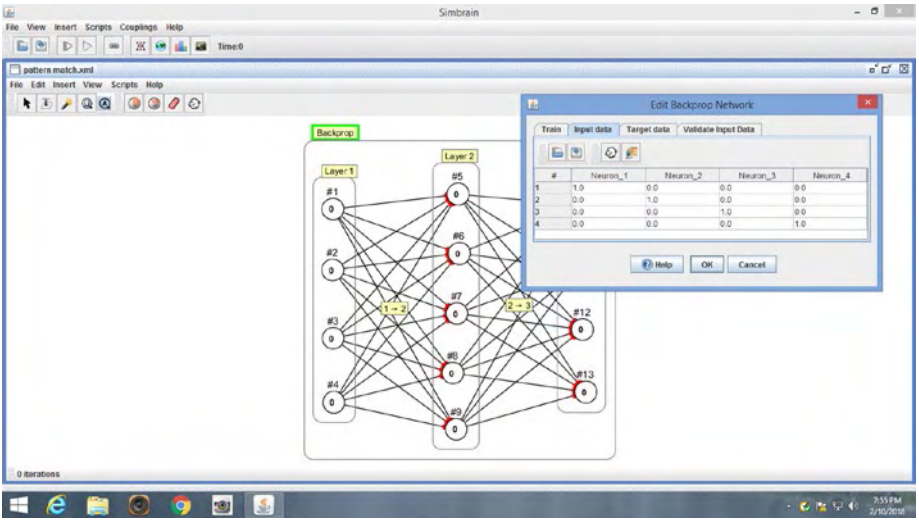


Figure A-5. *Input data tab*

Then, under the Target data tab, enter the desired target pattern for each output (Figure A-6).

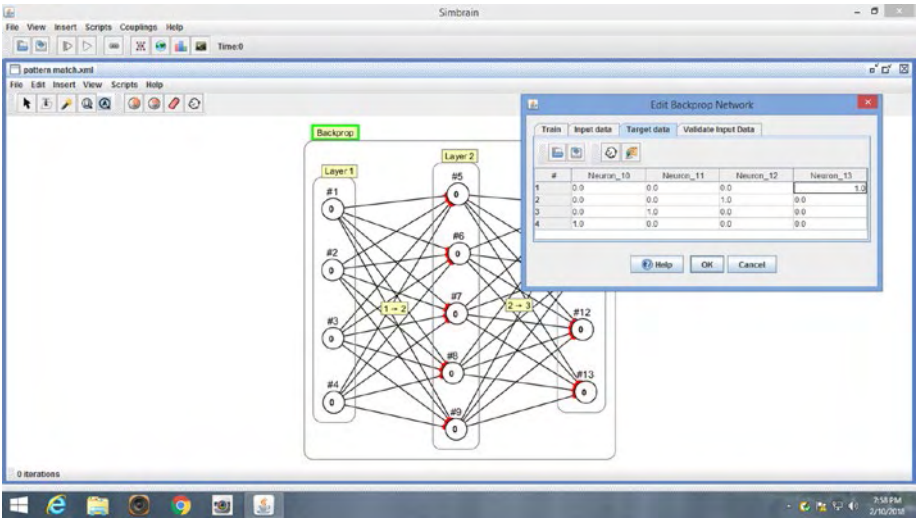


Figure A-6. *Target data tab*

Next, click the play button under the Train tab. The error should drop to zero almost immediately (Figure A-7). If it doesn't, hit the randomize button. Once the error goes to zero (or almost zero), hit the stop button to stop the training.

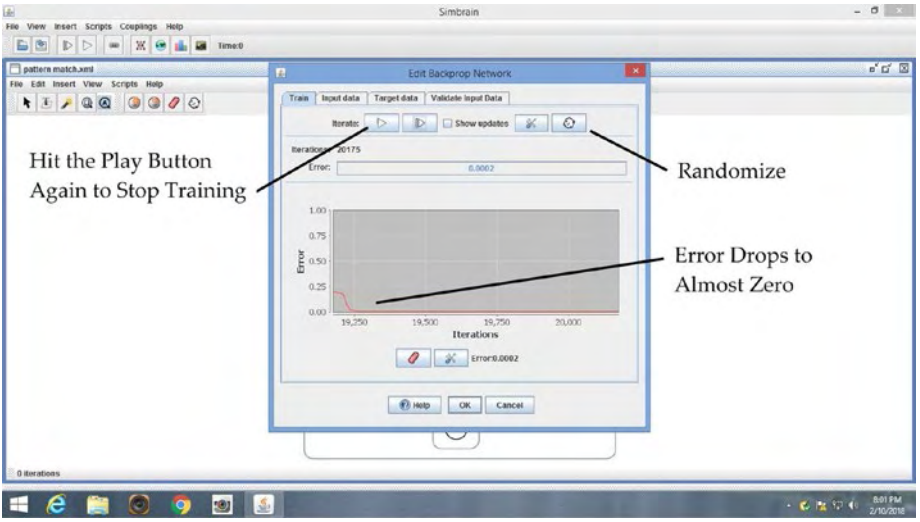


Figure A-7. *The network should train very quickly*

Then open the Validate Input Data tab and verify one row at a time that the network is, in fact, trained. Click the single step button to move to the next row. Figures A-8 through A-11 show the trained network one row at a time.

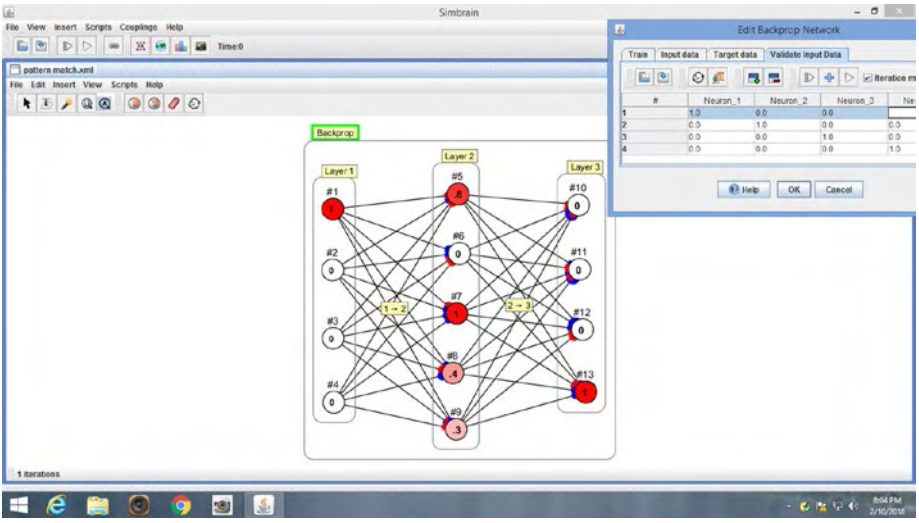


Figure A-8. *First row*

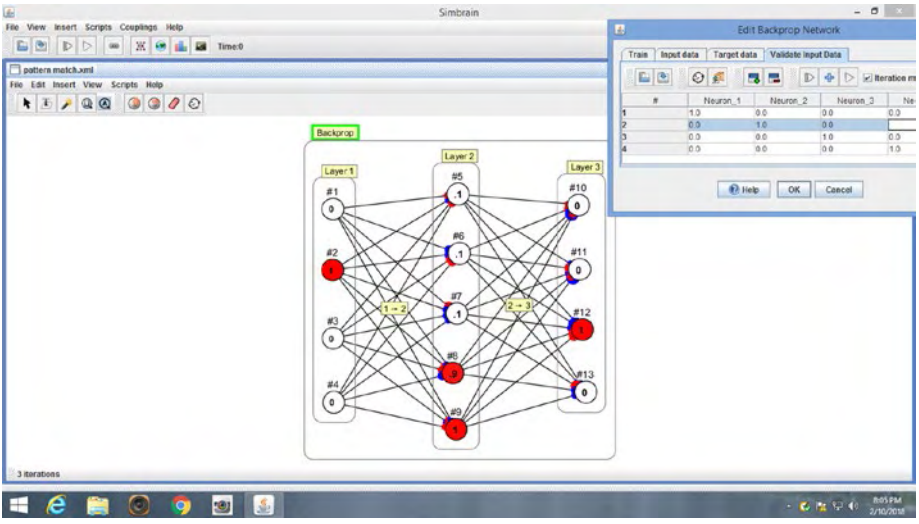


Figure A-9. *Second row*

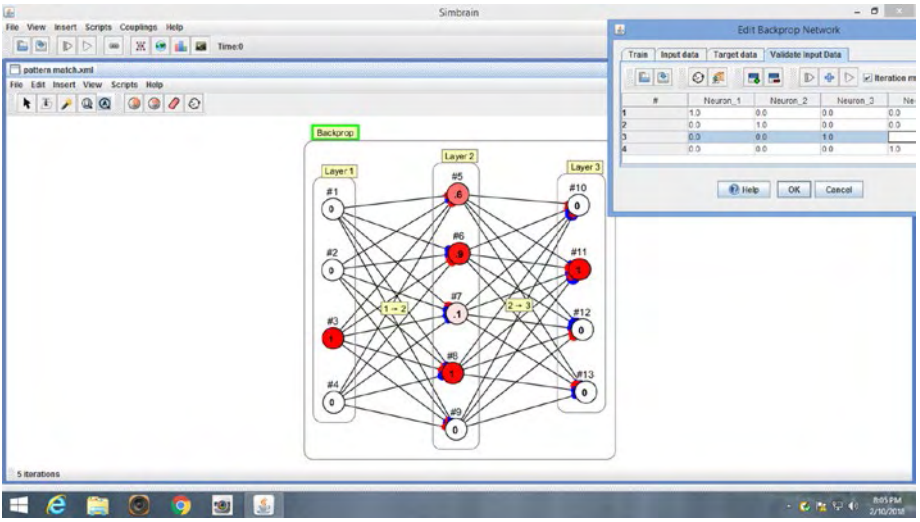


Figure A-10. *Third row*

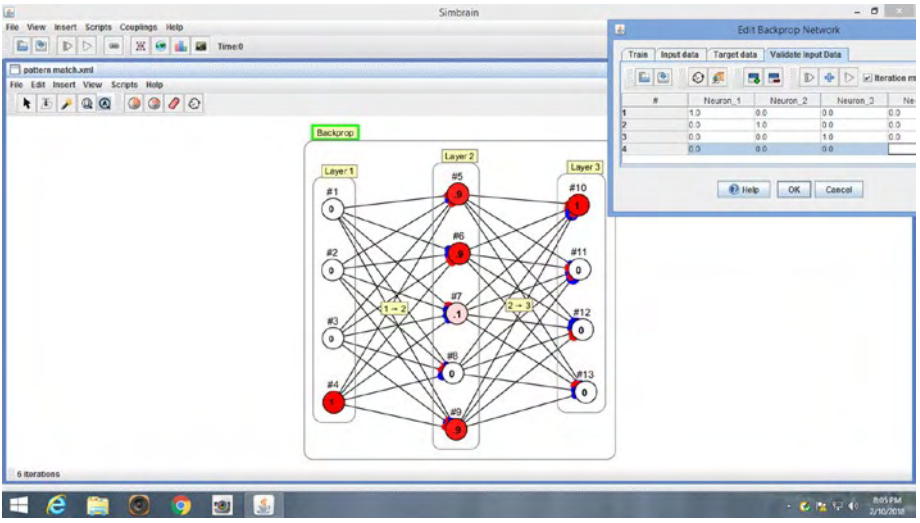
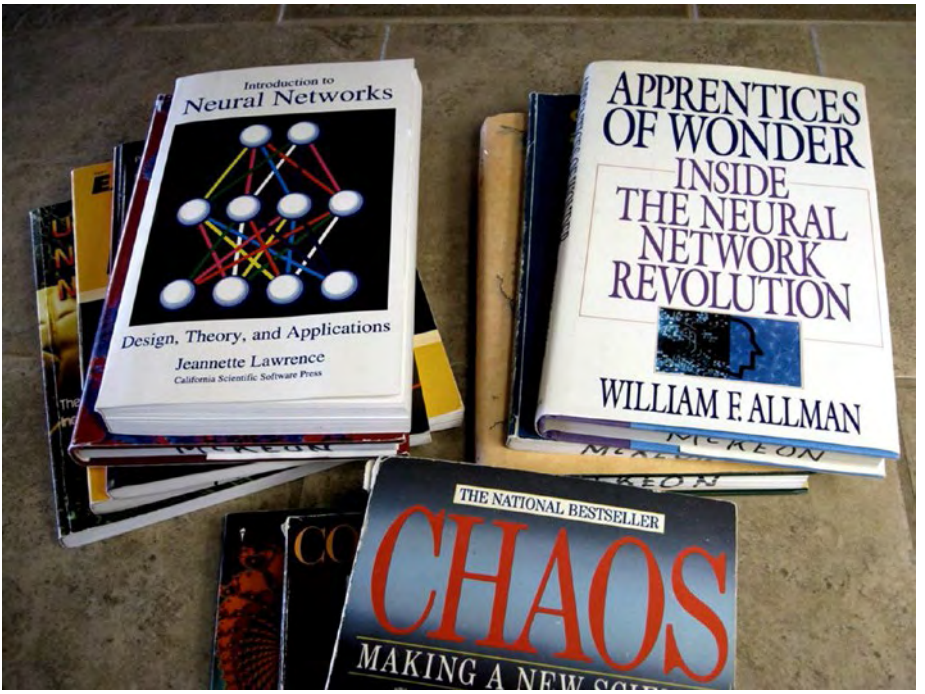


Figure A-11. *Fourth row*

APPENDIX B

Resources



There is a wealth of information available for learning more about neural networks. A quick search on the Internet will bring you hundreds of tutorial articles and web site addresses. A search on www.amazon.com for "neural networks" will bring up over 6,000 books! Many of them (probably most) are quite technical, but a quick look at their covers and descriptions will tell you if they are right for you.

R. McKeon, *Neural Networks for Electronics Hobbyists*,
https://doi.org/10.1007/978-1-4842-3507-2

Also, a search on YouTube for "neural networks" will bring up loads of excellent video tutorials. There are even online courses from recognized universities.

Some of my favorite books are ones that I bought several years ago. Back in the 1990s, a lot of new and exciting things were happening, and a lot of people were writing introductory-level books. Below I list a few of my favorites. All of these books are written with the layman and the hobbyist in mind.

Note regarding online reviews: You will see both good and bad reviews for each of these books. Keep in mind that everyone has an opinion, and sometimes those with the strongest opinions are the least qualified to judge. For the most part I can see where the reviewers are coming from, but I personally like each of these recommended books. None of them will answer all of your questions, but they all contain some valuable insights.

Neural Network Books

These books deal specifically with neural networks.

Lawrence, Jeannette. *Introduction to Neural Networks: Design, Theory, and Applications*. Nevada City, CA: California Scientific Software Press, 1994.

Caudill, Maureen, and Charles Butler. *Naturally Intelligent Systems.* Cambridge, MA: MIT Press, 2000.

Lovine, John. *Understanding Neural Networks: The Experimenter's Guide.* Copenhagen: SI, 2012.

De Wilde, Philippe. *Neural Network Models: Theory and Projects.* Berlin: Springer, 1997.

Rietman, Ed. *Experiments in Artificial Neural Networks*. Blue Ridge Summit, PA: TAB Books Inc., 1988.

Allman, William F. *Apprentices of Wonder: Inside the Neural Network Revolution.* New York: Bantam Books, 1989.

Chaos and Dynamic Systems

These books are not specifically about neural networks, but if you are interested in neural networks, I bet you will enjoy the following.

Gleick, James. *Chaos: Making a New Science*. New York: Penguin Books, 2008.

Peterson, Ivars. *The Mathematical Tourist: Snapshots of Modern Mathematics.* New York: W.H. Freeman and Company, 2001.

Devaney, Robert L. *Chaos, Fractals, and Dynamics: Computer Experiments in Mathematics*. Menlo Park, CA: Addison-Wesley Publishing Company, 2000.

Johnson, Steven. *Emergence: The Connected Lives of Ants, Brains, Cities, and Software*. New York: Scribner, 2012.

Waldrop, M. Mitchell. *Complexity: The Emerging Science at the Edge of Order and Chaos*. New York: Simon & Schuster, 1992.

Briggs, John, and F. David Peat. *Turbulent Mirror: An Illustrated Guide to Chaos Theory and the Science of Wholeness*. New York: Harper & Row, 2000.

McKeon, Rick. *Underlying Patterns*. CreateSpace.

Johnson, Neil. *Simply Complexity: A Clear Guide to Complexity Theory*. Oxford: Oneworld Publications, 2012.

Strogatz, Steven. *SYNC: How Order Emerges from Chaos in the Universe, Nature, and Daily Life.* New York: Hyperion, 2012.

Index

R. McKeon, *Neural Networks for Electronics Hobbyists*,
https://doi.org/10.1007/978-1-4842-3507-2

F

G

H, I

J, K

L, M

N

O

P, Q

R

S

T, U

V

W

X, Y, Z