Practical MATLAB Deep Learning

A Project-Based Approach

Michael Paluszek
Stephanie Thomas

Apress®

Practical MATLAB Deep Learning

A Project-Based Approach

Michael Paluszek
Stephanie Thomas

Apress®

Practical MATLAB Deep Learning: A Project-Based Approach

Michael Paluszek
Plainsboro, NJ
USA

Stephanie Thomas
Plainsboro, NJ
USA

ISBN-13 (pbk): 978-1-4842-5123-2
https://doi.org/10.1007/978-1-4842-5124-9

ISBN-13 (electronic): 978-1-4842-5124-9

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Coordinating Editor: MarkPowers

Cover designed by eStudioCalamar

Cover image designed by Freepik (http://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 editorial@apress.com; for reprint, paperback, or audio rights, please email bookpermissions@springernature.com.

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/9781484251232. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Contents

About the Authors

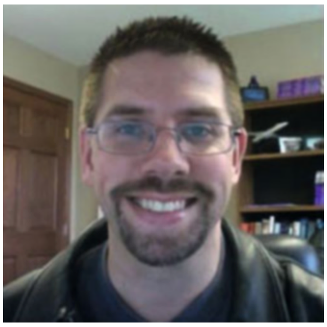
Michael Paluszek is President of Princeton Satellite Systems, Inc. (PSS) in Plainsboro, New Jersey. Mr. Michael founded PSS in 1992 to provide aerospace consulting services. He used MATLAB to develop the control system and simulations for the IndoStar-1 geosynchronous communications satellite. This led to the launch of Princeton Satellite Systems' first commercial MATLAB toolbox, the Spacecraft Control Toolbox, in 1995. Since then he has developed toolboxes and software packages for aircraft, submarines, robotics, and nuclear fusion propulsion, resulting in Princeton Satellite Systems' current extensive product line. He is working with the Princeton Plasma Physics Laboratory on a compact nuclear fusion reactor for energy generation and space propulsion.

Prior to founding PSS, Mr. Michael was an engineer at GE, Astro Space in East Windsor, NJ. At GE he designed the Global Geospace Science Polar despun platform control system and led the design of the GPS IIR attitude control system, the Inmarsat-3 attitude control systems, and the Mars Observer delta-V control system, leveraging MATLAB for control design. Mr. Michael also worked on the attitude determination system for the DMSP meteorological satellites. He flew communication satellites on over 12 satellite launches, including the GSTAR III recovery, the first transfer of a satellite to an operational orbit using electric thrusters. At Draper Laboratory, Mr. Michael worked on the Space Shuttle, Space Station, and submarine navigation. His Space Station work included designing of Control Moment Gyro-based control systems for attitude control.

Mr. Michael received his bachelor's degree in Electrical Engineering and master's and engineers' degrees in Aeronautics and Astronautics from the Massachusetts Institute of Technology. He is author of numerous papers and has over a dozen US patents. Mr. Michael is the author of *MATLAB Recipes*, *MATLAB Machine Learning,* and *MATLAB Machine Learning Recipes: A Problem-Solution Approach,* all published by Apress.

Stephanie Thomas is Vice President of Princeton Satellite Systems, Inc. in Plainsboro, New Jersey. She received her bachelor's and master's degrees in Aeronautics and Astronautics from the Massachusetts Institute of Technology in 1999 and 2001. Ms. Stephanie was introduced to the PSS Spacecraft Control Toolbox for MATLAB during a summer internship in 1996 and has been using MATLAB for aerospace analysis ever since. In her nearly 20 years of MATLAB experience, she has developed many software tools including the Solar Sail Module for the Spacecraft Control Toolbox, a proximity satellite operations toolbox for the Air Force, collision monitoring Simulink blocks for the Prisma satellite mission, and launch vehicle analysis tools in MATLAB and Java. She has developed novel methods for space situation assessment such as a numeric approach to assessing the general rendezvous problem between any two satellites implemented in both MATLAB and C++. Ms. Stephanie has contributed to PSS' *Spacecraft Attitude and Orbit Control* textbook, featuring examples using the Spacecraft Control Toolbox, and written many software user guides. She has conducted SCT training for engineers from diverse locales such as Australia, Canada, Brazil, and Thailand and has performed MATLAB consulting for NASA, the Air Force, and the European Space Agency. Ms. Stephanie is the author of *MATLAB Recipes*, *MATLAB Machine Learning,* and *MATLAB Machine Learning Recipes: A Problem-Solution Approach,* published by Apress. In 2016, Ms. Stephanie was named a NASA NIAC Fellow for the project ''Fusion-Enabled Pluto Orbiter and Lander.''

About the Technical Reviewer



Dr. Joseph Mueller specializes in control systems and trajectory optimization. For his doctoral thesis, he developed optimal ascent trajectories for stratospheric airships. His active research interests include robust optimal control, adaptive control, applied optimization and planning for decision support systems, and intelligent systems to enable autonomous operations of robotic vehicles. Prior to joining SIFT in early 2014, Dr. Joseph worked at Princeton Satellite Systems for 13 years. In that time, he served as the principal investigator for eight Small Business Innovation Research contracts for NASA, Air Force, Navy, and MDA. He has developed algorithms for optimal guidance and control of both formation flying spacecraft and high-altitude airships, and developed a course of action planning tool for DoD communication satellites. In support of a research study for NASA Goddard Space Flight Center in 2005, Dr. Joseph developed the Formation Flying Toolbox for MATLAB, a commercial product that is now used at NASA, ESA, and several universities and aerospace companies around the world. In 2006, he developed the safe orbit guidance mode algorithms and software for the Swedish Prisma mission, which has successfully flown a two-spacecraft formation flying mission since its launch in 2010. Dr. Joseph also serves as an adjunct professor in the Aerospace Engineering and Mechanics Department at the University of Minnesota, Twin Cities campus.

Acknowledgments

The authors would like to thank Eric Ham for suggesting LSTMs and also the idea for Chapter 7. Mr. Eric's concept was to use deep learning to identify specific flaws in a pirouette. Chapter 7 is a simpler version of the problem. Thanks to Shannen Prindle for helping with the Chapter 7 experiment and doing all of the photography for Chapter 7. Shannen is a Princeton University student who worked as an intern at Princeton Satellite Systems in the summer of 2019. We would also like to thank Dr. Charles Swanson for reviewing Chapter 6 on Tokamak control. Thanks to Kestras Subacius of the MathWorks for tech support on the bluetooth device. We would also like to thank Matt Halpin for reading the book from front to end.

We would like to thank dancers Shaye Firer, Emily Parker,田中稜子(Ryoko Tanaka) and Matanya Solomon for being our experimental subjects in this book. We would also like to thank the American Repertory Ballet and Executive Director Julie Hench for hosting our Chapter 7 experiment.

CHAPTER 1

■ ■ ■

What Is Deep Learning?

1.1 Deep Learning

Deep learning is a subset of machine learning which is itself a subset of artificial intelligence and statistics. Artificial intelligence research began shortly after World War II [24]. Early work was based on the knowledge of the structure of the brain, propositional logic, and Turing's theory of computation. Warren McCulloch and Walter Pitts created a mathematical formulation for neural networks based on threshold logic. This allowed neural network research to split into two approaches: one centered on biological processes in the brain and the other on the application of neural networks to artificial intelligence. It was demonstrated that any function could be implemented through a set of such neurons and that a neural net could learn. In 1948, Norbert Wiener's book, *Cybernetics*, was published which described concepts in control, communications, and statistical signal processing. The next major step in neural networks was Donald Hebb's book in 1949, *The Organization of Behavior,* connecting connectivity with learning in the brain. His book became a source of learning and adaptive systems. Marvin Minsky and Dean Edmonds built the first neural computer at Harvard in 1950.

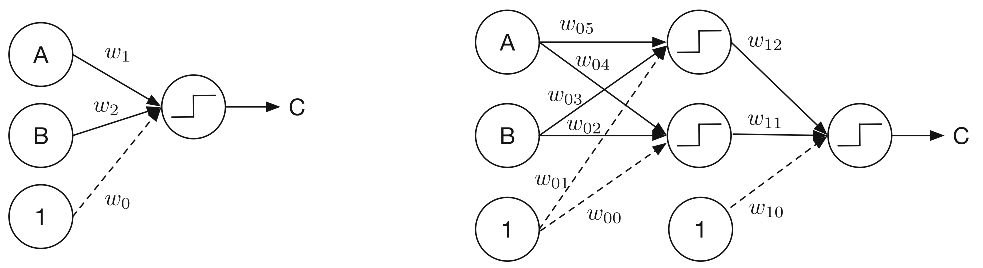
The first computer programs, and the vast majority now, have knowledge built into the code by the programmer. The programmer may make use of vast databases. For example, a model of an aircraft may use multidimensional tables of aerodynamic coefficients. The resulting software therefore knows a lot about aircraft, and running simulations of the models may present surprises to the programmer and the users. Nonetheless, the programmatic relationships between data and algorithms are predetermined by the code.

In machine learning, the relationships between the data are formed by the learning system. Data is input along with the results related to the data. This is the system training. The machine learning system relates the data to the results and comes up with rules that become part of the system. When new data is introduced, it can come up with new results that were not part of the training set.

Deep learning refers to neural networks with more than one layer of neurons. The name ‘‘deep learning'' implies something more profound, and in the popular literature, it is taken to imply that the learning system is a ‘‘deep thinker.'' Figure 1.1 shows a single-layer and multilayer network. It turns out that multilayer networks can learn things that single-layer

M. Paluszek and S. Thomas, *Practical MATLAB Deep Learning*,
https://doi.org/10.1007/978-1-4842-5124-9_1

Figure 1.1: Two neural networks. The one on the right is a deep learning network.



networks cannot. The elements of a network are nodes, where signals are combined, weights and biases. Biases are added at nodes. In a single layer, the inputs are multiplied by weights, then added together at the end, after passing through a threshold function. In a multilayer or deep learning network, the inputs are combined in the second layer before being output. There are more weights, and the added connections allow the network to learn and solve more complex problems.

There are many types of machine learning. Any computer algorithm that can adapt based on inputs from the environment is a learning system. Here is a partial list:

1. Neural nets (deep learning or otherwise)
2. Support vector machines
3. Adaptive control
4. System identification
5. Parameter identification (may be the same as the previous one)
6. Adaptive expert systems
7. Control algorithms (a proportional integral derivative control stores information about constant inputs in its integrator)

Some systems use a predefined algorithm and learn by fitting parameters of the algorithm. Others create a model entirely from data. Deep learning systems are usually in the latter category.

We'll give a brief history of deep learning and then move on to two examples.

1.2 History of Deep Learning

Minsky wrote the book *Perceptrons* with Seymour Papert in 1969, which was an early analysis of artificial neural networks. The book contributed to the movement toward symbolic processing in AI. The book noted that single neurons could not implement some logical functions such as exclusive-or (XOR) and erroneously implied that multilayer networks would have the same issue. It was later found that three-layer networks could implement such functions. We give the XOR solution in this book.

Multilayer neural networks were discovered in the 1960s but not really studied until the 1980s. In the 1970s, self-organizing maps using competitive learning were introduced [14]. A resurgence in neural networks happened in the 1980's. Knowledge-based, or ''expert,'' systems were also introduced in the 1980s. From Jackson [16],

> *An expert system is a computer program that represents and reasons with knowledge of some specialized subject with a view to solving problems or giving advice.*
>
> —Peter Jackson, *Introduction to Expert Systems*

Back propagation for neural networks, a learning method using gradient descent, was reinvented in the 1980s, leading to renewed progress in this field. Studies began both of human neural networks (i.e., the human brain) and the creation of algorithms for effective computational neural networks. This eventually led to deep learning networks in machine learning applications.

Advances were made in the 1980s as AI researchers began to apply rigorous mathematical and statistical analysis to develop algorithms. Hidden Markov Models were applied to speech. A Hidden Markov Model is a model with unobserved (i.e., hidden) states. Combined with massive databases, they have resulted in vastly more robust speech recognition. Machine translation has also improved. Data mining, the first form of machine learning as it is known today, was developed.

In the early 1990s, Vladimir Vapnik and coworkers invented a computationally powerful class of supervised learning networks known as Support Vector Machines (SVM). These networks could solve problems of pattern recognition, regression, and other machine learning problems.

There has been an explosion in deep learning in the past few years. New tools have been developed that make deep learning easier to implement. TensorFlow is available from Amazon AWS. It makes it easy to deploy deep learning on the cloud. It includes powerful visualization tools. TensorFlow allows you to deploy deep learning on machines that are only intermittently connected to the Web. IBM Watson is another. It allows you to use TensorFlow, Keras, PyTorch, Caffe, and other frameworks. Keras is a popular deep learning framework that can be used in Python. All of these frameworks have allowed deep learning to be deployed just about everywhere.

In this book, we will present MATLAB-based deep learning tools. These powerful tools let you create deep learning systems to solve many different problems. In our book, we will apply MATLAB deep learning to a wide range of problems ranging from nuclear fusion to classical ballet.

Before getting into our examples, we will give some fundamentals on neural nets. We will first give backgrounds on neurons and how an artificial neuron represents a real neuron. We will then design a daylight detector. We will follow this with the famous XOR problem that stopped neural net development for some time. Finally, we will discuss the examples in this book.

1.3 Neural Nets

Neural networks, or neural nets, are a popular way of implementing machine ‘‘intelligence.” The idea is that they behave like the neurons in a brain. In this section, we will explore how neural nets work, starting with the most fundamental idea with a single neuron and working our way up to a multilayer neural net. Our example for this will be a pendulum. We will show how a neural net can be used to solve the prediction problem. This is one of the two uses of a neural net, prediction and classification. We’ll start with a simple classification example.

Let’s first look at a single neuron with two inputs. This is shown in Figure 1.2. This neuron has inputs x_1 and x_2, a bias b, weights w_1 and w_2, and a single output z. The activation function σ takes the weighted input and produces the output. In this diagram, we explicitly add icons for the multiplication and addition steps within the neuron, but in typical neural net diagrams such as Figure 1.1, they are omitted.

$$z = \sigma(y) = \sigma(w_1 x_1 + w_2 x_2 + b) \tag{1.1}$$

Let’s compare this with a real neuron as shown in Figure 1.3. A real neuron has multiple inputs via the dendrites. Some of these branch which means that multiple inputs can connect to the cell body through the same dendrite. The output is via the axon. Each neuron has one output. The axon connects to a dendrite through the synapse. Signals pass from the axon to the dendrite via a synapse.

There are numerous commonly used activation functions. We show three:

$$\sigma(y) = \tanh(y) \tag{1.2}$$

$$\sigma(y) = \frac{2}{1 - e^{-y}} - 1 \tag{1.3}$$

$$\sigma(y) = y \tag{1.4}$$

The exponential one is normalized and offset from zero so it ranges from -1 to 1. The last one, which simply passes through the value of y, is called the *linear* activation function. The

Figure 1.2: A two-input neuron.

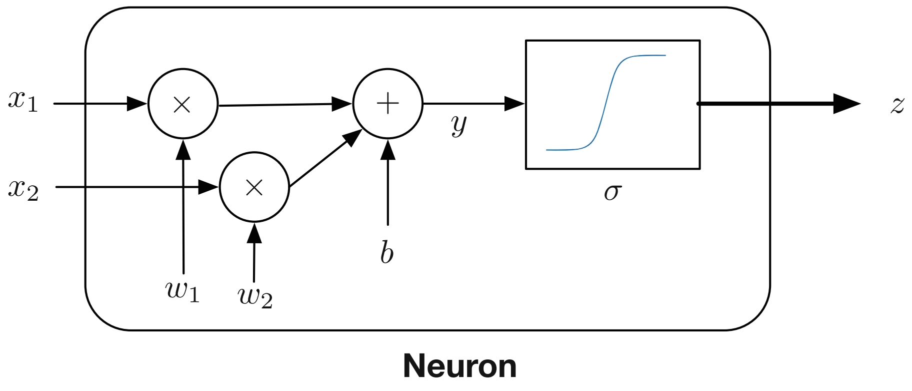


Figure 1.3: A neuron connected to a second neuron. A real neuron can have 10,000 inputs!

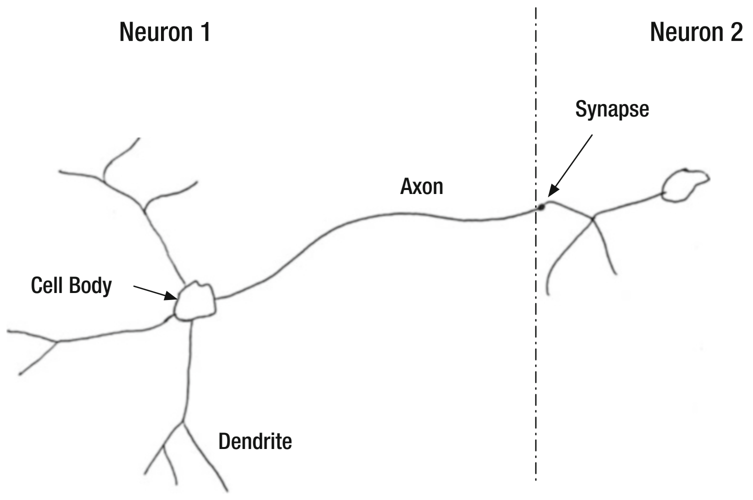
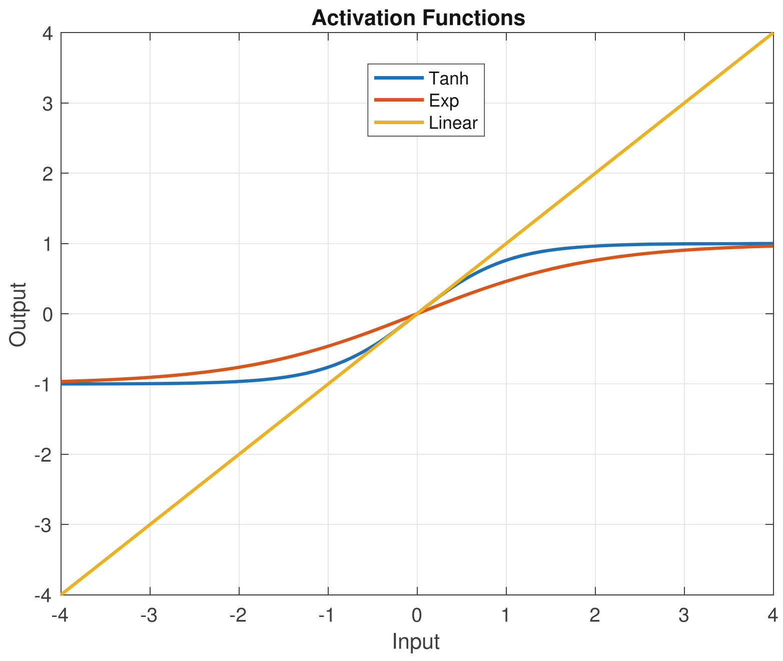


Figure 1.4: The three activation functions from `OneNeuron`.



following code in the script `OneNeuron.m` computes and plots these three activation functions for an input `q`. Figure 1.4 shows the three activation functions on one plot.

OneNeuron.m

```
%% Single neuron demonstration.
%% Look at the activation functions
y       = linspace(-4,4);
z1      = tanh(y);
z2      = 2./(1+exp(-y)) - 1;

PlotSet(y,[z1;z2;y],'x label','Input', 'y label',...
  'Output', 'figure title','Activation Functions','plot title', '
      Activation Functions',...
  'plot set',{[1 2 3]},'legend',{{'Tanh','Exp','Linear'}});
```

Activation functions that saturate, or reach a value of input after which the output is constant or changes very slowly, model a biological neuron that has a maximum firing rate. These particular functions also have good numerical properties that are helpful in learning.

Let's look at a single input neural net shown in Figure 1.5. This neuron is

$$z = \sigma(2x + 3) \tag{1.5}$$

where the weight w on the single input x is 2 and the bias b is 3. If the activation function is linear, the neuron is just a linear function of x,

$$z = y = 2x + 3 \tag{1.6}$$

Neural nets do make use of linear activation functions, often in the output layer. It is the nonlinear activation functions that give neural nets their unique capabilities.

Let's look at the output with the preceding activation functions plus the threshold function from the script `LinearNeuron.m`. The results are in Figure 1.6.

Figure 1.5: A one-input neural net. The weight w is 2 and the bias b is 3.

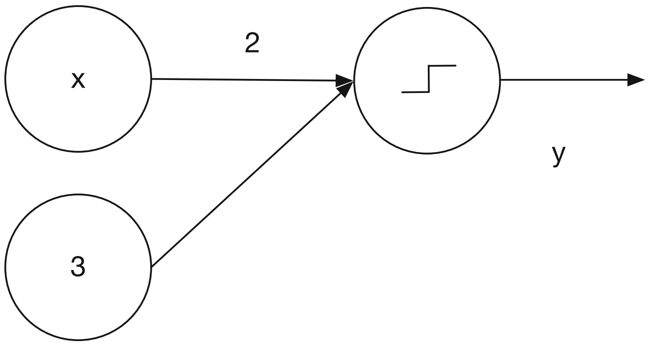
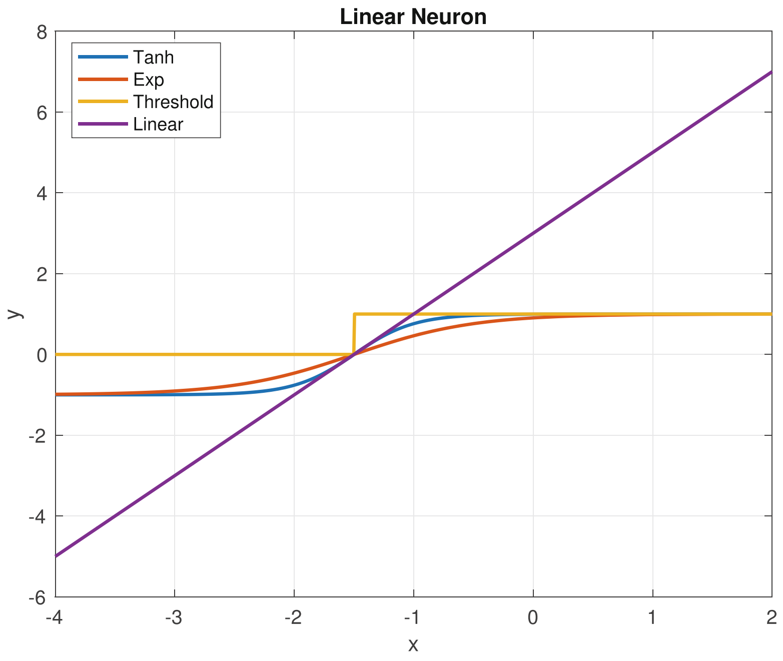


Figure 1.6: The "linear" neuron compared to other activation functions from `LinearNeuron`.



LinearNeuron.m

```
%% Linear neuron demo
x        = linspace(-4,2,1000);
y        = 2*x + 3;
z1       = tanh(y);
z2       = 2./(1+exp(-y)) - 1;
z3       = zeros(1,length(x));

% Apply a threshold
k        = y >=0;
z3(k)    = 1;

PlotSet(x,[z1;z2;z3;y],'x label','x', 'y label',...
  'y', 'figure title','Linear Neuron','plot title', 'Linear Neuron',...
  'plot set',{[1 2 3 4]},'legend',{{'Tanh','Exp','Threshold','Linear'}});
```

The `tanh` and `exp` are very similar. They put bounds on the output. Within the range $-3 \leq x < 1$, they return the function of the input. Outside those bounds, they return the sign of the input, that is, they saturate. The threshold function returns zero if the value is less than 0 and 1 if it is greater than -1.5. The threshold is saying the output is only important, thus *activated*, if the input exceeds a given value. The other nonlinear activation functions are saying that we care about the value of the linear equation only within the bounds. The nonlinear functions (but not step) make it easier for the learning algorithms since the functions have derivatives. The binary step has a discontinuity at an input of zero so that its derivative is infinite at that point. Aside from the linear function (which is usually used on output neurons), the neurons are just telling us that the sign of the linear equation is all we care about. The activation function is what makes a neuron a neuron.

We now show two brief examples of neural nets: first, a daylight detector, and second, the exclusive-or problem.

1.3.1 Daylight Detector

Problem

We want to use a simple neural net to detect daylight. This will provide an example of using a neural net for classification.

Solution

Historically, the first neuron was the perceptron. This is a neuron with an activation function that is a threshold. Its output is either 0 or 1. This is not really useful for man real-world problems. However, it is well suited for simple classification problems. We will use a single perceptron in this example.

How It Works

Suppose our input is a light level measured by a photo cell. If you weight the input so that 1 is the value defining the brightness level at twilight, you get a sunny day detector.

This is shown in the following script, `SunnyDay`. The script is named after the famous neural net that was supposed to detect tanks but instead detected sunny days; this was due to all the training photos of tanks being taken, unknowingly, on a sunny day, while all the photos without tanks were taken on a cloudy day. The solar flux is modeled using a cosine and scaled so that it is 1 at noon. Any value greater than 0 is daylight.

SunnyDay.m

```
%% The data
t = linspace(0,24);         % time, in hours
d = zeros(1,length(t));
s = cos((2*pi/24)*(t-12)); % solar flux model

%% The activation function
% The nonlinear activation function which is a threshold detector
j    = s < 0;
s(j) = 0;
j    = s > 0;
d(j) = 1;

%% Plot the results
PlotSet(t,[s;d],'x label','Hour', 'y label',...
  {'Solar Flux', 'Day/Night'}, 'figure title','Daylight Detector',...
  'plot title', {'Flux Model','Perceptron Output'});
set([subplot(2,1,1) subplot(2,1,2)],'xlim',[0 24],'xtick',[0 6 12 18 24]);
```

Figure 1.7: The daylight detector. The top plot shows the input data, and the bottom plot shows the perceptron output detecting daylight.

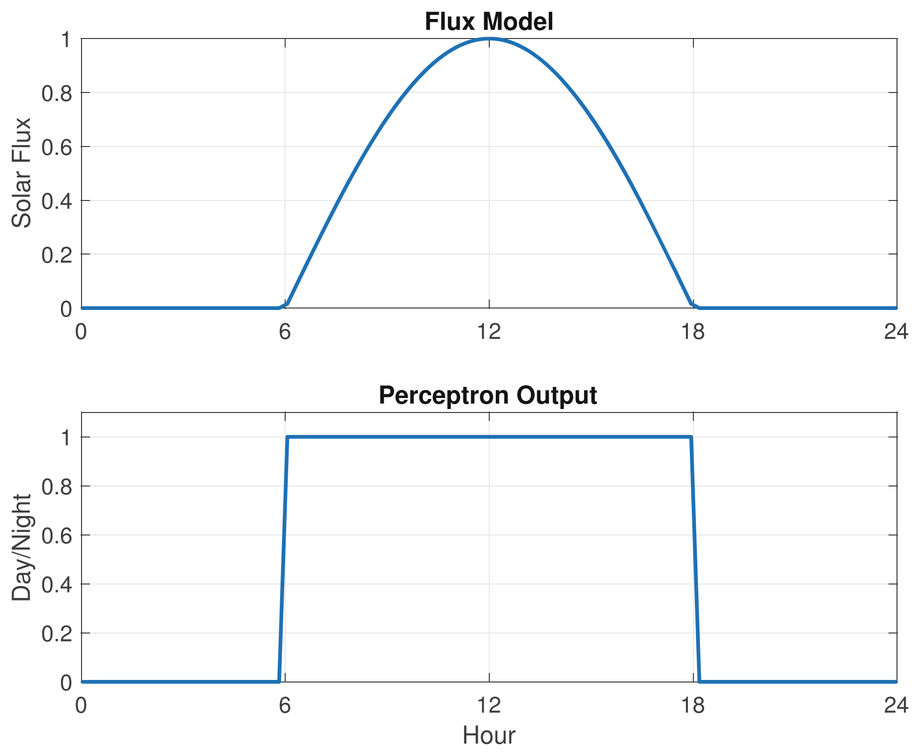


Figure 1.7 shows the detector results. The `set(gca,...)` code sets the x-axis ticks to end at exactly 24 hours. This is a really trivial example but does show how classification works. If we had multiple neurons with thresholds set to detect sunlight levels within bands of solar flux, we would have a neural net sun clock.

1.3.2 XOR Neural Net

Problem

We want to implement the exclusive-or (XOR) problem with a neural network.

Solution

The XOR problem impeded the development of neural networks for a long time before ‘‘deep learning’’ was developed. Look at Figure 1.8. The table on the left gives all possible inputs A and B and the desired outputs C. ‘‘Exclusive-or’’ just means that if the inputs A and B are different, the output C is 1. The figure shows a single-layer network and a multilayer network, as in Figure 1.1, but with the weights labeled as they will be in the code. You can implement this in MATLAB easily, in just seven lines:

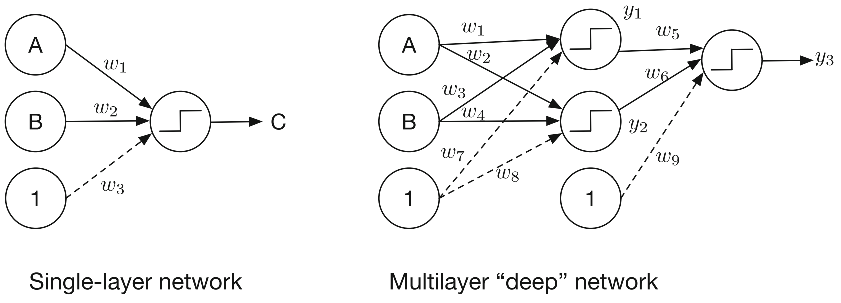
```
>> a = 1;
>> b = 0;
>> if( a == b )
```

Figure 1.8: Exclusive-or (XOR) truth table and possible solution networks.

C = XOR(A,B)

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

Truth Table



```
>>   c = 1
>> else
>>   c = 0
>> end

c =
     0
```

This type of logic was embodied in medium-scale integrated circuits in the early days of digital systems and in tube-based computers even earlier than that. Try as you might, you cannot pick two weights and a bias on the single-layer network to reproduce the XOR. Minsky created a proof that it was impossible.

The second neural net, the deep neural net, can reproduce the XOR. We will implement and train this network.

How It Works

What we will do is explicitly write out the back propagation algorithm that trains the neural net from the four training sets given in Figure 1.8, that is, (0,0), (1,0), (0,1), (1,1). We'll write it in the script `XORDemo`. The point is to show you explicitly how back propagation works. We will use the `tanh` as the activation function in this example. The XOR function is given in `XOR.m` shown as follows.

XOR.m

```
%% XOR Implement an 'Exclusive Or' neural net
%   c = XOR(a,b,w)
%
%% Description
% Implements an XOR function in a neural net. It accepts vector inputs.
%
%% Inputs
%   a   (1,:)   Input 1
%   b   (1,:)   Input 2
%   w   (9,1)   Weights and biases
%% Outputs
%   c   (1,:)   Output
%
```

```
function [y3,y1,y2] = XOR(a,b,w)

if( nargin < 1 )
  Demo
  return
end

y1 = tanh(w(1)*a  + w(2)*b  + w(7));
y2 = tanh(w(3)*a  + w(4)*b  + w(8));
y3 = w(5)*y1 + w(6)*y2 + w(9);
c  = y3;
```

There are three neurons. The activation function for the hidden layer is the hyperbolic tangent. The activation function for the output layer is linear.

$$\begin{aligned} y_1 &= \tanh(w_1 a + w_2 b + w_7) && (1.7)\\ y_2 &= \tanh(w_3 a + w_4 b + w_8) && (1.8)\\ y_3 &= w_5 y_1 + w_6 y_2 + w_9 && (1.9)\end{aligned}$$

Now we will derive the back propagation routine. The hyperbolic activation function is

$$f(z) = \tanh(z) \tag{1.10}$$

Its derivative is

$$\frac{df(z)}{dz} = 1 - f^2(z) \tag{1.11}$$

In this derivation, we are going to use the chain rule. Assume that F is a function of y which is a function of x. Then

$$\frac{dF(y(x))}{dx} = \frac{dF}{dy}\frac{dy}{dx} \tag{1.12}$$

The error is the square of the difference between the desired output and the output. This is known as a quadratic error. It is easy to use because the derivative is simple and the error is always positive, making the lowest error the one closest to zero.

$$E = \frac{1}{2}(c - y_3)^2 \tag{1.13}$$

The derivative of the error for w_j for the output node

$$\frac{\partial E}{\partial w_j} = (y_3 - c)\frac{\partial y_3}{\partial w_j} \tag{1.14}$$

For the hidden nodes, it is

$$\frac{\partial E}{\partial w_j} = \psi_3 \frac{\partial n_3}{\partial w_j} \tag{1.15}$$

Expanding for all the weights

$$\frac{\partial E}{\partial w_1} = \psi_3\psi_1 a \tag{1.16}$$
$$\frac{\partial E}{\partial w_2} = \psi_3\psi_1 b \tag{1.17}$$
$$\frac{\partial E}{\partial w_3} = \psi_3\psi_2 a \tag{1.18}$$
$$\frac{\partial E}{\partial w_4} = \psi_3\psi_2 b \tag{1.19}$$
$$\frac{\partial E}{\partial w_5} = \psi_3 y_1 \tag{1.20}$$
$$\frac{\partial E}{\partial w_6} = \psi_3 y_2 \tag{1.21}$$
$$\frac{\partial E}{\partial w_7} = \psi_3\psi_1 \tag{1.22}$$
$$\frac{\partial E}{\partial w_8} = \psi_3\psi_2 \tag{1.23}$$
$$\frac{\partial E}{\partial w_9} = \psi_3 \tag{1.24}$$

where

$$\psi_1 = 1 - f^2(n_1) \tag{1.25}$$
$$\psi_2 = 1 - f^2(n_2) \tag{1.26}$$
$$\psi_3 = y_3 - c \tag{1.27}$$
$$n_1 = w_1 a + w_2 b + w_7 \tag{1.28}$$
$$n_2 = w_3 a + w_4 b + w_8 \tag{1.29}$$
$$n_3 = w_5 y_1 + w_6 y_2 + w_9 \tag{1.30}$$

You can see from the derivation how this could be made recursive and apply to any number of outputs or layers. Our weight adjustment at each step will be

$$\Delta w_j = -\eta \frac{\partial E}{\partial w_j} \tag{1.31}$$

where η is the update gain. It should be a small number. We only have four sets of inputs. We will apply them multiple times to get the XOR weights.

Our back propagation trainer needs to find the nine elements of `w`. The training function `XORTraining.m` is shown as follows.