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.

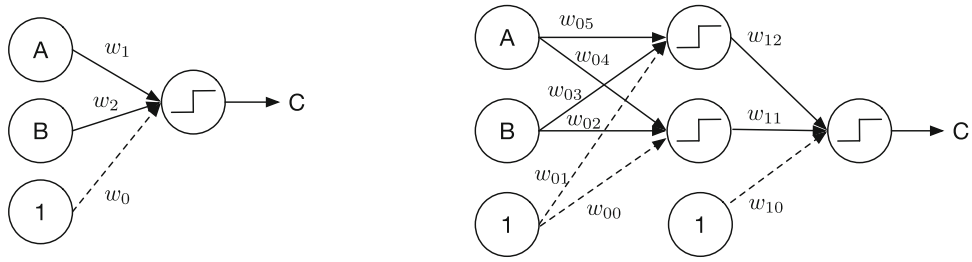
The first computer programs, and the vast majority now, have knowledge built into the code by the programmer. The programmer may make use of vast databases. For example, a model of an aircraft may use multidimensional tables of aerodynamic coefficients. The resulting software therefore knows a lot about 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:

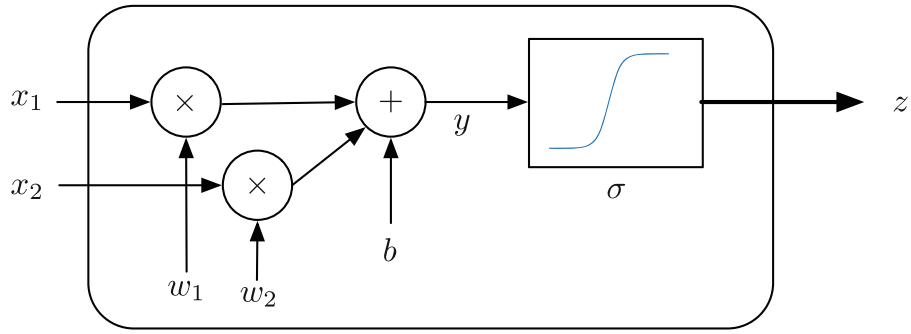
$$\sigma(y) = \tanh(y) \tag{1.2}$$

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

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

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.



Neuron

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

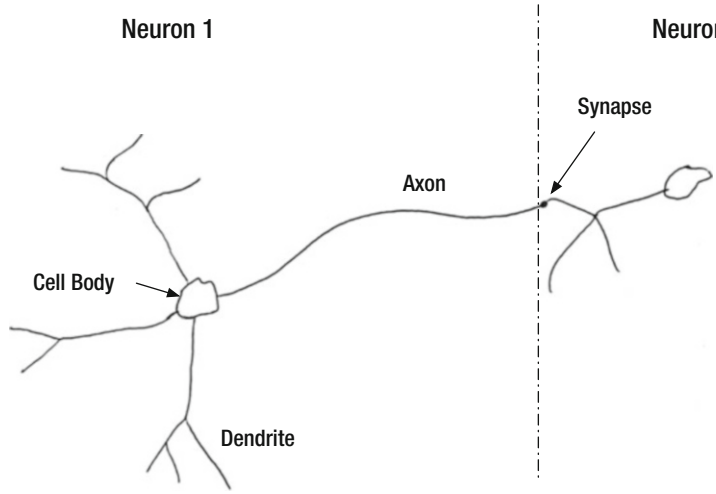
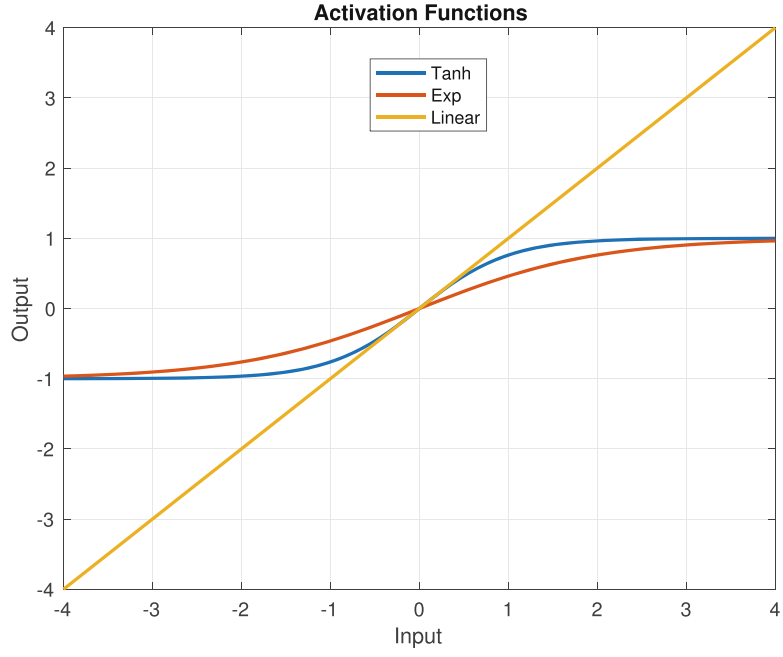


Figure 1.4: The three activation functions from `OneNeuron`.



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.

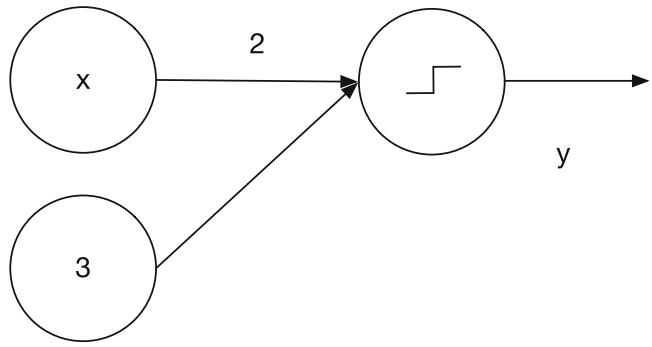
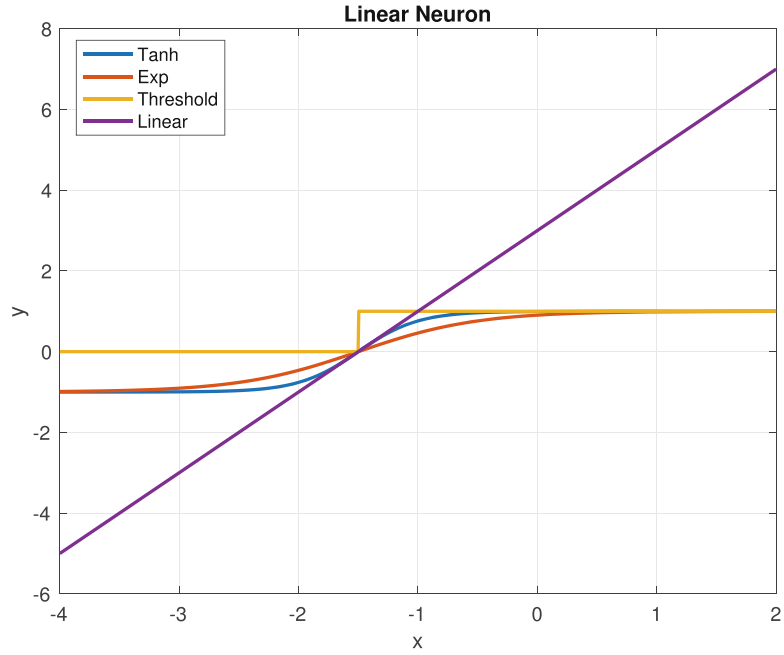


Figure 1.6: The ‘‘linear” neuron compared to other activation functions from `LinearNeuron`.



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.

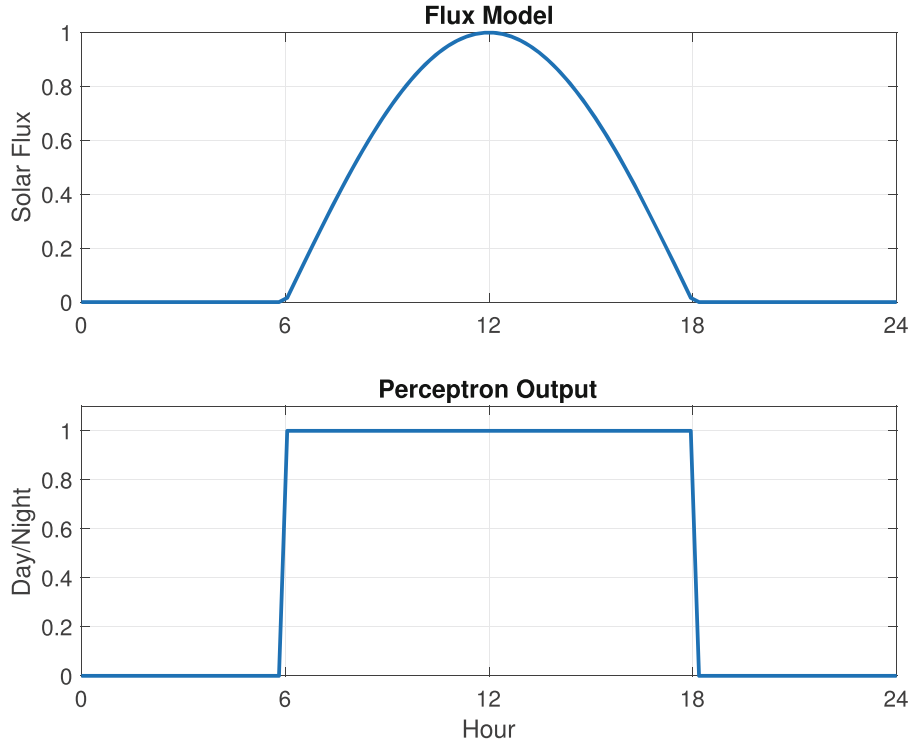


Figure 1.7 shows the detector results. The `set(gca,...)` code sets the x-axis ticks to end at exactly 24 hours. This is a really trivial example but does show how classification works. If we had multiple neurons with thresholds set to detect sunlight levels within bands of solar flux, we would have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

1.3.2 XOR Neural Net

Problem

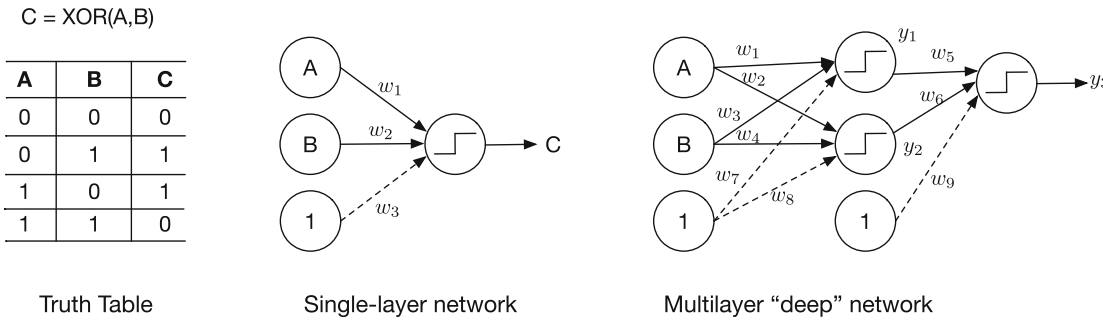
We want to implement the exclusive-or (XOR) problem with a neural network.

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 = 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

```
1  %% XOR Implement an 'Exclusive Or' neural net
2  %  c = XOR(a,b,w)
3  %
4  %% Description
5  % Implements an XOR function in a neural net. It accepts vector inputs.
6  %
7  %% Inputs
8  %  a  (1,:)  Input 1
9  %  b  (1,:)  Input 2
10 %  w  (9,1)  Weights and biases
11 %% Outputs
12 %  c  (1,:)  Output
13 %
```

```
14 function [y3,y1,y2] = XOR(a,b,w)
15
16 if( nargin < 1 )
17   Demo
18   return
19 end
20
21 y1 = tanh(w(1)*a  + w(2)*b  + w(7));
22 y2 = tanh(w(3)*a  + w(4)*b  + w(8));
23 y3 = w(5)*y1 + w(6)*y2 + w(9);
24 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.

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

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.

XORTraining.m

```
%% XORTRAINING Implements an XOR training function.
%% Inputs
%  a     (1,4) Input 1
%  b     (1,4) Input 2
%  c     (1,4) Output
%  w     (9,1) Weights and biases
%  n     (1,1) Number of iterations through all 4 inputs
%  eta   (1,1) Training weight
%
%% Outputs
%  w     (9,1)  Weights and biases
%% See also
% XOR

function w = XORTraining(a,b,c,w,n,eta)

if( nargin < 1 )
  Demo;
  return
end

e       = zeros(4,1);
y3      = XOR(a,b,w);
e(:,1)  = y3 - c;
wP      = zeros(10,n+1); % For plotting the weights
for k = 1:n
  wP(:,k) = [w;mean(abs(e))];
  for j = 1:4
    [y3,y1,y2]  = XOR(a(j),b(j),w);
    psi1        = 1 - y1^2;
    psi2        = 1 - y2^2;
    e(j)        = y3 - c(j);
    psi3        = e(j); % Linear activation function
    dW          = psi3*[psi1*a(j);psi1*b(j);psi2*a(j);psi2*b(j);y1;y2;
                  psi1;psi2;1];
    w           = w - eta*dW;
  end
end
wP(:,k+1) = [w;mean(abs(e))];

% For legend entries
wName = string;
for k = 1:length(w)
  wName(k) = sprintf('W_%d',k);
end
leg{1} = wName;
leg{2} = '';

PlotSet(0:n,wP,'x label','step','y label',{'Weight' 'Error'},...
  'figure title','Learning','legend',leg,'plot set',{1:9 10});
```

The first two arguments to `PlotSet` are the data and are the minimum required. The remainder are parameter pairs. The `leg` value has legends for the two plots, as defined by `'plot set'`. The first plot uses the first nine data points, in this case, the weights. The second plot uses the last data point, the mean of the error. `leg` is a cell array with two strings or string arrays. The `'plot set'` is two arrays in a cell. A plot with only one value will not generate a legend.

The demo script `XORDemo.m` starts with the training data, which is actually the complete truth data for this simple function, and randomly generated weights. It iterates through the inputs 25000 times, with a training weight of 0.001.

XORDemo.m

```
% Training data - also the truth data
a       = [0 1 0 1];
b       = [0 0 1 1];
c       = [0 1 1 0];

% First try implementing random weights
w0      = [ 0.1892; 0.2482; -0.0495; -0.4162; -0.2710;...
             0.4133; -0.3476; 0.3258; 0.0383];
cR      = XOR(a,b,w0);

fprintf('\nRandom Weights\n')
fprintf('    a     b   c\n');
for k = 1:4
  fprintf('%5.0f %5.0f %5.2f\n',a(k),b(k),cR(k));
end

% Now execute the training
w       = XORTraining(a,b,c,w0,25000,0.001);
cT      = XOR(a,b,w);
```

The results of the neural network with random weights and biases, as expected, are not good. After training, the neural network reproduces the XOR problem very well, as shown in the following demo output. Now, if you change the initial weights and biases, you may find that you get bad results. This is because the simple gradient method implemented here can fall into local minima from which it can't escape. This is an important point about finding the best answer. There may be many good answers, which are local optimals, but there will be only one best answer. There is a vast body of research on how to guarantee that a solution is a global optimal.

```
>> XORDemo

Random Weights
    a     b   c
    0     0  0.26
    1     0  0.19
    0     1  0.03
```

```
     1     1 -0.04

Weights and Biases
   Initial   Final
    0.1892  1.7933
    0.2482  1.8155
   -0.0495 -0.8535
   -0.4162 -0.8591
   -0.2710  1.3744
    0.4133  1.4893
   -0.3476 -0.4974
    0.3258  1.1124
    0.0383 -0.5634

Trained
     a     b   c
     0     0  0.00
     1     0  1.00
     0     1  1.00
     1     1  0.01
```

Figure 1.9 shows the weights and biases converging and also shows the mean output error over all four inputs in the truth table going to zero. If you try other starting weights and biases, this may not be the case. Other solution methods, such as Genetic Algorithms [13], Electromagnetism based [4], and Simulated Annealing [23], are less susceptible to falling into local minima but can be slow. A good overview into optimization specifically for machine learning is given by Bottou [6].

In the next chapter, we will use the deep learning toolbox to solve this problem.

You might wonder how this compares to a set of linear equations. If we remove the activation functions, we get

$$y_3 = w_9 + w_6 w_8 + w_5 w_7 + a(w_1 w_5 + w_3 w_6) + b(w_2 w_5 + 2_r w_6) \tag{1.32}$$

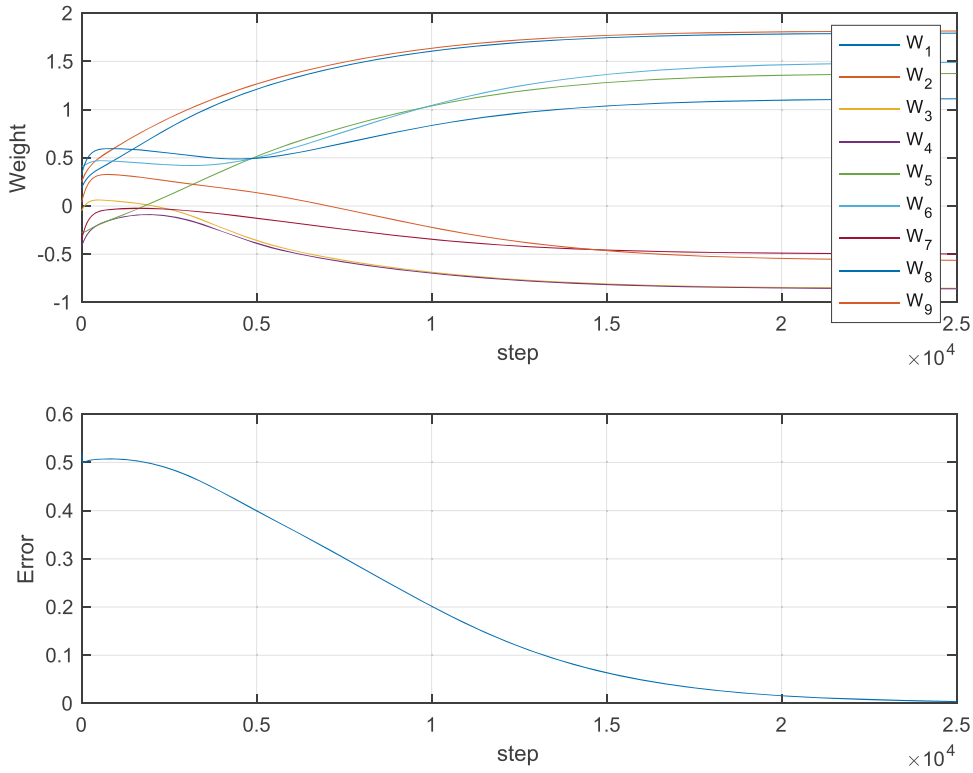
This reduces to just three independent coefficients.

$$y_3 = k_1 + k_2 a + k_3 b \tag{1.33}$$

One is a constant, and the other two multiply the inputs. Writing the four possible cases in matrix notation, we get

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = k_1 + \begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} k_2 \\ k_3 \end{bmatrix} \tag{1.34}$$

Figure 1.9: Evolution of weights during exclusive-or (XOR) training.



We can get close to a working XOR if we choose

$$\begin{bmatrix} k_1 \\ k_2 \\ k_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} \tag{1.35}$$

This makes three out of four equations correct. There is no way to make all four correct with just three coefficients. The activation functions separate the coefficients and allow us to reproduce the XOR. This is not surprising because the XOR is not a linear problem.

1.4 Deep Learning and Data

Deep learning systems operate on data. Data may be organized in many ways. For example, we may want a deep learning system to identify an image. A color image that is 2 pixels by 2 pixels by 3 colors could be represented with random data using `rand`.

```
>> x = rand(2,2,3)

x(:,:,1) =

```

```
    0.9572    0.8003
    0.4854    0.1419

x(:,:,2) =

    0.4218    0.7922
    0.9157    0.9595

x(:,:,3) =

    0.6557    0.8491
    0.0357    0.9340
```

The array form implies a structure for the data. The same number of points could be organized into a single vector using `reshape`.

```
>> reshape(x,12,1)

ans =

    0.9572
    0.4854
    0.8003
    0.1419
    0.4218
    0.9157
    0.7922
    0.9595
    0.6557
    0.0357
    0.8491
    0.9340
```

The numbers are the same, they are just organized differently. Convolutional neural networks, described in the next section, are often used for image structured data. We might also have a vector:

```
>> s = rand(2,1)

s =

    0.6787
    0.7577
```

for which we wish to learn a temporal or time sequence. In this case, if each column is a time sample, we might have

```
>> rand(2,4)

ans =
```

```
4
5       0.7431    0.6555    0.7060    0.2769
6       0.3922    0.1712    0.0318    0.0462
```

For example, we might want to look at an ongoing sequence of samples and determine if a set of k samples matches a predetermined sequence. For this simple problem, the neural net would learn the sequence and then be fed sets of four samples to match.

We also need to distinguish what we mean by matching. If all of our numbers are exact, the problem is relatively straightforward. In real systems, measurements are often noisy. In those cases, we want to match with a certain probability. This leads to the concept of statistical neural nets.

1.5 Types of Deep Learning

There are many types of deep learning networks. New types are under development as you read this book. One deep learning researcher joked that if you randomly put together four letters, you will have the name for an existing deep learning algorithm.

The following sections briefly describe some of the major types.

1.5.1 Multilayer Neural Network

Multilayer neural networks have

1. Input neurons
2. Multiple layers of hidden neurons
3. Output neurons

The different layers may have different activation functions. They may also be functionally different such as being a convolution or pooling layer. In a later chapter, we will introduce the idea of an algorithmic layer.

1.5.2 Convolutional Neural Networks (CNN)

A CNN has convolutional layers (hence the name). It convolves a feature with the input matrix so that the output emphasizes that feature. This effectively finds patterns. For example, you might convolve an L pattern with the incoming data to find corners. The human eye has edge detectors, making the human vision system a convolutional neural network of sorts.

1.5.3 Recurrent Neural Network (RNN)

Recurrent neural networks are a type of recursive neural network. Recurrent neural networks are used for time-dependent problems. They combine the last time step's data with the data from the hidden or intermediate layer, to produce a representation of the current time step. A recurrent neural net has a loop. An input vector at time k is used to create an output which is then passed to the next element of the network. This is done recursively in that each stage is identical with external inputs and inputs from the previous stage. Recurrent neural nets are used in speech recognition, language translation, and many other applications. One can see

how a recurrent network would be useful in translation in that the meaning of the latter part of an English sentence can be dependent of the beginning. Now this presents a problem. Suppose we are translating a paragraph. Is the output of the first stage necessarily relevant to the 100th stage? Maybe not. In standard estimation, old data is forgotten using a forgetting factor. In neural nets, we use long short-term memory networks, or LSTM networks.

1.5.4 Long Short-Term Memory Networks (LSTMs)

LSTMs are designed to avoid the dependency on old information. A standard RNN has a repeating structure. An LSTM also has a repeating structure, but each element has four layers. The LSTM layers decide what old information to pass on to the next layer. It may be all, or it may be none. There are many variants on LSTM, but they all include the fundamental ability to forget things.

1.5.5 Recursive Neural Network

This is often confused with recurrent neural networks (RNNs), which are a type of recursive neural network. Recursive neural networks operate on structured data. They've been used successfully on language processing as language is structured (as opposed to images which are not).

1.5.6 Temporal Convolutional Machines (TCMs)

The TCM is a convolutional architecture designed to learn temporal sequences [19]. TCMs are particularly useful for statistical modeling of temporal sequences. Statistical modeling is appropriate when incoming data is noisy.

1.5.7 Stacked Autoencoders

A stacked autoencoder is a neural net made up of a series of sparse autoencoders. An autoencoder is a type of neural network that is an unsupervised learning algorithm using back propagation. Sparsity is a measure of how many neurons are activated, that is, have inputs that cause it to produce an output for a given activation function. The outputs of one layer feed into the next. The number of nodes tends to decrease as you move from input to output.

1.5.8 Extreme Learning Machine (ELM)

ELMs were invented by Guang-Bin Huang [15]. ELMs are a single hidden layer feedforward network. It randomly chooses the weights of the hidden nodes and analytically computes the weights of the output nodes. ELMs provide good performance and learn quickly.

1.5.9 Recursive Deep Learning

Recursive deep learning [28] is a variation and extension of RNNs. The same set of neural node weight is applied recursively over a structured input. That is, not all of the inputs are processed in batch. Recursion is a standard method used in general estimation when data is coming in at different times and you want the best estimate at the current time without having to process all available data at once.

1.5.10 Generative Deep Learning

Generative deep learning allows a neural network to learn patterns [12] and then create completely new material. A generative deep learning network can create articles, paintings, photographs, and many other types of material.

1.6 Applications of Deep Learning

Deep learning is used in many applications today. Here are a few.

Image recognition —This is arguably the best known and most controversial use of deep learning. A deep learning system is trained with pictures of people. Cameras are distributed everywhere and images captured. The system then identifies individual faces and matches them against its trained database. Even with variations of lighting, weather conditions, and clothing, the system can identify the people in the images.

Speech recognition —You hardly ever get a human being on the phone anymore. You are first presented with a robotic listener that can identify what you are saying, at least within the limited context of what it expects. When a human listens to another human, the listener is not just recording the speech, he or she is guessing what the person is going to say and filling in gaps of garbled words and confusing grammar. Robotic listeners have some of the same abilities. A robotic listener is an embodiment of the ‘‘Turing test.’’ Did you ever get one that you thought was a human being? Or for that matter, did you ever reach a human who you thought was a robot?

Handwriting analysis —A long time ago, you would get forms in which you had boxes in which to write numbers and letters. At first they had to be block capitals! A robotic handwriting system could figure out the letters in those boxes reliably. Years later, though many years ago, the US Post Office introduced zip code reading systems. At first you had to put the zip code on a specific part of the envelope. That system has evolved so that it can find zip codes anywhere. This made the zip + 4 system really valuable and a big productivity boost.

Machine translation —Google translate does a pretty good job considering it can translate almost any language in the world. It is an example of a system with online training. You see that when you type in a phrase and the translation has a check mark next to it because a human being has indicated that it is correct. Figure 1.10 gives an example. Google harnesses the services of free human translators to improve its product!

Targeting —By targeting we mean figuring out what you want. This may be a movie, a clothing item, or a book. Deep learning systems collect information on what you like and decide what you would be most interested in buying. Figure 1.11 gives an example. This is from a couple of years ago. Perhaps ballet dancers like Star Wars!

Other applications include game playing, autonomous driving, medicine, and many others. Just about any human activity can be an application of deep learning.

Figure 1.10: Translation from Japanese into English.

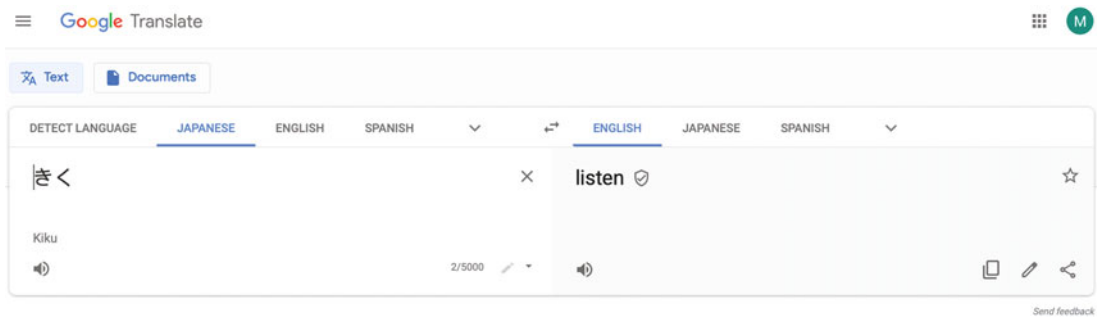
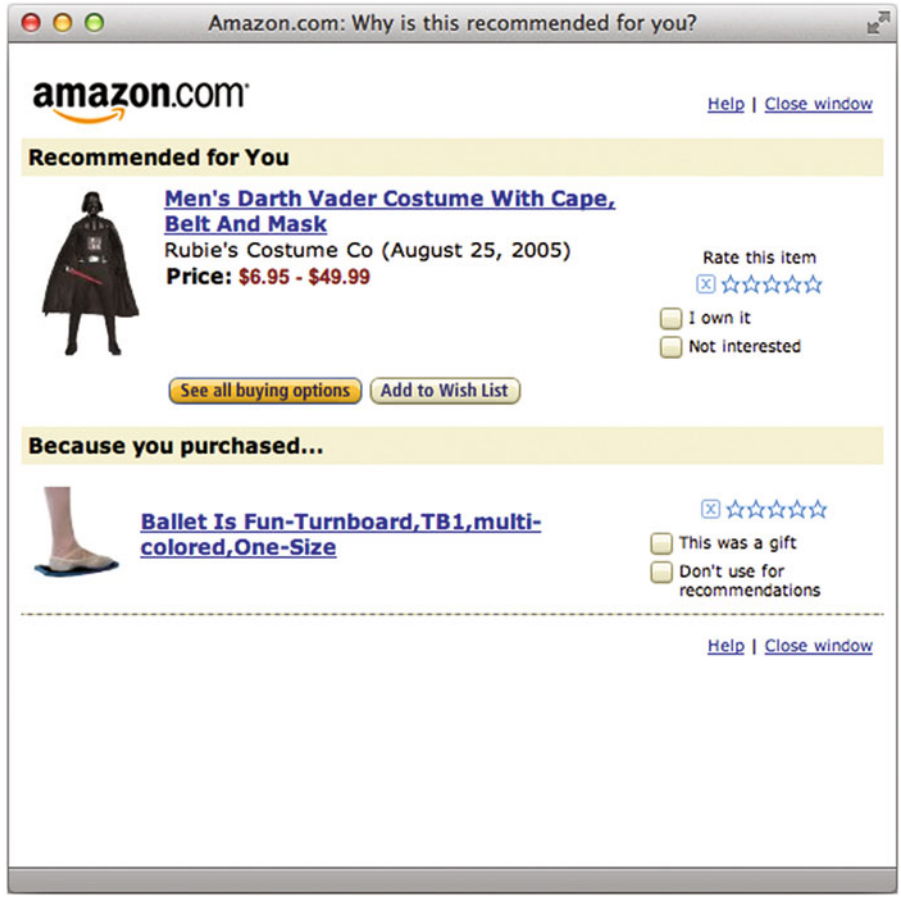


Figure 1.11: Prediction of your buying patterns.



1.7 Organization of the Book

This book is organized around specific deep learning examples. You can jump into any chapter as they are pretty much independent. We've tried to present a wide range of topics, some of which, hopefully, align with your work or interests. The next chapter gives an overview of MATLAB products for deep learning. We only use three of their toolboxes in this book, besides the core MATLAB development environment.

Each chapter except for the first and second is organized in the following order:

1. Modeling
2. Building the system
3. Training the system
4. Testing the system

Training and testing are often in the same script. Modeling varies with each chapter. For physical problems, we derive numerical models, usually sets of differential equations, and build simulations of the processes.

The chapters in this book present a range of relatively simple examples to help you learn more about deep learning and its applications. It will also help you learn the limitations of deep learning and areas for future research. All use the MATLAB deep learning toolbox.

1. **What Is Deep Learning?** (this chapter).
2. **MATLAB Machine and Deep Learning Toolboxes**—This chapter gives you an introduction to MATLAB machine intelligence toolboxes. We'll be using three of the toolboxes in this book.
3. **Finding Circles with Deep Learning**—This is an elementary example. The system will try to figure out if a figure is a circle. It will be presented with circles, ellipses, and other objects and trained to determine which are circles.
4. **Classifying Movies**—All movie databases try to guess what movies will be of most interest to their viewers to speed movie selection and reduce the number of disgruntled customers. This example creates a movie rating system and attempts to classify movies in the movie database as good or bad.
5. **Algorithmic Deep Learning**—This is an example of fault detection using a detection filter as an element of the deep learning system. It uses a custom deep learning algorithm, the only example that does not use the MATLAB deep learning toolbox.
6. **Tokamak Disruption Detection**—Disruptions are a major problem with a nuclear fusion device known as a Tokamak. Researchers are using neural nets to detect disruptions before they happen so that they can be stopped. In this example, we use a simplified dynamical model, to demonstrate deep learning.

7. **Classifying a Ballet Dancer's Pirouette**—This example demonstrates how to use real-time data in a deep learning system. IT uses IMU data via Bluetooth, and a camera input. The data is combined to classify a dancer's pirouette. This example will also cover data acquisition and use real-time data as part of a deep learning system.

8. **Completing Sentences**—Writing systems sometimes attempt to predict what word or sentence fragment you are trying to use. We create a database of sentences and try to predict the remainder as soon as possible.

9. **Terrain-Based Navigation**—The first cruise missiles used terrain mapping to reach their targets. This has been largely replaced by GPS. This system will identify where an aircraft is on a map and use past positions to predict future positions.

10. **Stock Prediction**—Who wouldn't want a system that could create portfolios that would beat an index fund? Perhaps a stock prediction system that would find the next Apple or Microsoft! In this example, we create an artificial stock market and train the system to identify the best stocks.

11. **Image Classification**—Training deep learning networks can take weeks. This chapter gives you an example of using a pretrained network.

12. **Orbit Determination**—Orbits can be determined using only angle measurements. This chapter shows how `fitnet` can produce estimates of semi-major axis and eccentricity from angles.

These are all very different problems. We give a brief summary of the theory behind each which is hopefully enough for you to understand the problem. There are hundreds of papers on each topic, and even textbooks on these subjects. The references provide more information. There are two broad methods for applying deep learning in MATLAB. One is using `trainNetwork` and the other is using the various feedforward functions. Table 1.1 summarizes which methods are used in each chapter.

Chapter 11 uses pretrained networks, but these are similar to those produced by `trainNetwork`. Chapter 12 applies four types of network training to the same problem.

For each problem, we are creating a world in which to work. For example, in the classifying movies problem, we create a world of movies and viewers based on a particular model that we create. This is akin to the famous ‘‘Blocks World" in which a world of colored blocks was created. The artificial intelligence engine could reason and solve problems of stacking blocks within the context of this world. Much like ‘‘Blocks World" did not map into general reasoning, we do not claim that our code can be applied directly to real-world problems.

In each chapter, we will present a problem and give code that creates a deep learning network to solve the problem. We will show you the performance of the code and where it doesn't work as well as we would like. Deep learning is a work in progress, and it is important to understand what works and what doesn't work. We encourage our readers to go beyond the code in the book and see if they can improve on its performance.

Table 1.1: Deep learning methods. The specific form of the network is shown. The last column shows the application.

Chapter	Feed Forward	`trainNetwork`	Type
2	`feedforwardnet`		Regression
3		Convolutional	Image classification
4	`patternnet`		Classification
5	`feedforwardnet`		Regression
6		Bidirectional LSTM	Classification
7		Bidirectional LSTM	Classification
8		Bidirectional LSTM	Classification
9		Convolutional	Image classification
10		LSTM	Regression
11		**	Image classification
12	`feedforwardnet`, `fitnet`, `cascadeforwardnet`	Bidirectional LSTM	Regression

We present much of the code in segments. Unless specified, you cannot cut and paste the code into the MATLAB command window and get a result. You should run the demos from the code base that is included with the book. Remember also that you will need the Deep Learning Toolbox and Instrument Control Toolbox for Chapter 7. The other chapters only require core MATLAB and the Deep Learning Toolbox.

The code in this book was developing using MATLAB 2019a on a Macintosh MacBook Pro under MacOS 10.14.4. The code should work on all other operating systems though processing time may vary.

CHAPTER 2

MATLAB Machine Learning Toolboxes

2.1 Commercial MATLAB Software

2.1.1 MathWorks Products

The MathWorks sells several packages for machine learning. Their toolboxes work directly with MATLAB and Simulink. The MathWorks products provide high-quality algorithms for data analysis along with graphics tools to visualize the data. Visualization tools are a critical part of any machine learning system. They can be used for data acquisition, for example, for image recognition or as part of systems for autonomous control of vehicles, or for diagnosis and debugging during development. All of these packages can be integrated with each other and with other MATLAB functions to produce powerful systems for machine learning. The most applicable toolboxes that we will discuss are listed in the following; we will use only the deep learning and the Instrument Control toolboxes in this book.

- **Deep Learning Toolbox**
- **Instrument Control Toolbox**
- Statistics and Machine Learning Toolbox
- Neural Network Toolbox
- Computer Vision System Toolbox
- Image Acquisition Toolbox
- Parallel Computing Toolbox
- Text Analytics Toolbox

The breadth of MATLAB and Simulink products allow you to explore every facet of machine learning and to connect with other areas of data science including controls, estimation,

M. Paluszek and S. Thomas, *Practical MATLAB Deep Learning*,
https://doi.org/10.1007/978-1-4842-5124-9_2

and simulation. There are also many domain-specific toolboxes, such as the Automated Driving Toolbox and Sensor Fusion and Tracking Toolbox that can be used with the learning products.

Deep Learning Toolbox

The deep learning toolbox allows you to design, build, and visualize convolutional neural networks. You can implement existing, pretrained neural networks available on the Web, such as GoogLeNet, VGG-16, VGG_19, AlexNet, and ResNet-59. GoogLeNet and AlexNet are image classification networks and are discussed in Chapter 11. The deep learning toolbox has extensive capabilities for visualization and debugging of neural networks. The debugging tools are important to ensure that your system is behaving properly and help you to understand what is going on inside your neural network. It includes a number of pretrained models. We will use this toolbox in all of our examples.

Instrument Control Toolbox

The MATLAB Instrument Control Toolbox is designed to directly connect instruments. This simplifies the use of MATLAB with hardware. Examples include oscilloscopes, function generators, and power supplies. The toolbox provides support for TCP/IP, UDP, I2C, SPI, and Bluetooth. With the Instrument Control Toolbox, you can integrate MATLAB directly into your laboratory workflow without the need for writing drivers or creating specialized MEX files. We use the Bluetooth functionality with an IMU in this book.

Statistics and Machine Learning Toolbox

The statistics and machine learning toolbox provides data analytics methods for gathering trends and patterns from massive amounts of data. These methods do not require a model for analyzing the data. The toolbox functions can be broadly divided into classification tools, regression tools, and clustering tools. Statistics are the foundation for much of deep learning.

Classification methods are used to place data into different categories. For example, data, in the form of an image, might be used to classify an image of an organ as having a tumor. Classification is used for handwriting recognition, credit scoring, and face identification. Classification methods include support vector machines (SVM), decision trees, and neural networks.

Regression methods let you build models from current data to predict future data. The models can then be updated as new data becomes available. If the data is only used once to create the model, then it is a batch method. A regression method that incorporates data as it becomes available is a recursive method.

Clustering finds natural groupings in data. Object recognition is an application of clustering methods. For example, if you want to find a car in an image, you look for data that is associated with the part of an image that is a car. While cars are of different shapes and sizes, they have many features in common. Clustering can also deal with different orientations and scalings.

The toolbox has many functions to support these areas and many that do not fit neatly into these categories. The statistics and machine learning toolbox provides professional tools that are seamlessly integrated into the MATLAB environment.

Computer Vision System Toolbox

The MATLAB Computer Vision System Toolbox provides functions for developing computer vision systems. The toolbox provides extensive support for video processing. It includes functions for feature detection and extraction. Prior to the extensive use of deep learning, feature detection was the approach for image identification. It also supports 3D vision and can process information from stereo cameras. 3D motion detection is supported.

Image Acquisition Toolbox

The MATLAB Image Acquisition Toolbox provides functions for connecting cameras directly into MATLAB without the need for intermediary software or using the apps that come with many cameras. You can use the package to interact with the sensors directly. Foreground and background acquisition is supported. The toolbox supports all major standards and hardware vendors. It makes it easier to design deep learning image processing software using real data. It allows control of cameras as is shown in the chapter on using images as part of deep learning.

The Image Acquisition Toolbox supports USB3 Vision, GigE Vision, and GenICam GenTL. You can connect to Velodyne LiDAR®sensors, machine vision cameras, and frame grabbers, as well as high-end scientific and industrial devices. USB3 gives you considerable control over the camera and is used in Chapter 7.

Parallel Computing Toolbox

The Parallel Computing Toolbox allows you to use multicore processors, graphical processing units (GPUs), and computer clusters with your MATLAB software. It allows you to easily parallelize algorithms using high-level programming constructs like parallel `for` loops. Some functions in the deep learning toolbox can take advantage of GPUs and parallel processing. There is an example of potential GPU use in Chapter 10. As almost every personal computer has a GPU, this can be a worthwhile addition to your MATLAB software.

Text Analytics Toolbox

Text Analytics Toolbox provides algorithms and visualizations for working with text data. Models created with the toolbox can be used in applications such as sentiment analysis, predictive maintenance, and topic modeling. The toolbox includes tools for processing raw text from many sources. You can extract individual words, convert text into numerical representations, and build statistical models. This is a useful adjunct to deep learning.

2.2 MATLAB Open Source

MATLAB open source tools are a great resource for implementing machine learning. Machine learning and convex optimization packages are available. Universities are constantly producing new neural network toolsets. Much work is done in Python, but MATLAB is a very popular base for software development and AI work.

2.2.1 Deep Learn Toolbox

The Deep Learn Toolbox by Rasmus Berg Palm is a MATLAB toolbox for deep learning. It includes Deep Belief Nets, Stacked Autoencoders, Convolutional neural nets, and other neural net functions. It is available through the MathWorks File Exchange.

2.2.2 Deep Neural Network

The Deep Neural Network by Masayuki Tanaka provides deep learning tools of deep belief networks of stacked restricted Boltzmann machines. It has functionality for both unsupervised and supervised learning. It is available through the MathWorks File Exchange.

2.2.3 MatConvNet

MatConvNet implements Convolutional Neural Networks for image processing. It includes a range of pretrained networks for image processing functions. You can find it by searching on the name or at the time of printing at www.vlfeat.org/matconvnet/. This package is open source and is open to contributors.

2.2.4 Pattern Recognition and Machine Learning Toolbox (PRMLT)

This toolbox implements the functionality of the book *Pattern Recognition and Machine Learning,* by Christopher Bishop [5]. The book is an excellent reference, and the code makes it easy to use the algorithms discussed in the book.

2.3 XOR Example

We'll give many examples of the deep learning toolbox in subsequent chapters. We'll do one example just to get you going. This example doesn't even unlock a fraction of the power in the deep learning toolbox. We will implement the XOR example which we also did in Chapter 1. The `DLXOR.m` script is shown in the following, using the MATLAB functions `feedforwardnet`, `configure`, `train`, and `sim`.

DLXOR.m

```
%% Use the Deep Learning Toolbox to create the XOR neural net

%% Create the network
% 2 layers
% 2 inputs
% 1 output

net = feedforwardnet(2);

% XOR Truth table
a   = [1 0 1 0];
b   = [1 0 0 1];
c   = [0 0 1 1];

% How many sets of inputs
```

```
n   = 600;

% This determines the number of inputs and outputs
x   = zeros(2,n);
y   = zeros(1,n);

% Create training pairs
for k = 1:n
  j       = randi([1,4]);
  x(:,k)  = [a(j); b(j)];
  y(k)    = c(j);
end

net       = configure(net, x, y);
net.name  = 'XOR';
net       = train(net,x,y);
c         = sim(net,[a;b]);

fprintf('\n    a     b   c\n');
for k = 1:4
  fprintf('%5.0f %5.0f %5.2f\n',a(k),b(k),c(k));
end

% This only works for feedforwardnet(2);
fprintf('\nHidden layer biases %6.3f %6.3f\n',net.b{1});
fprintf('Output layer bias   %6.3f\n',net.b{2});
fprintf('Input layer weights  %6.2f %6.2f\n',net.IW{1}(1,:));
fprintf('                     %6.2f %6.2f\n',net.IW{1}(2,:));
fprintf('Output layer weights %6.2f %6.2f\n',net.LW{2,1}(1,:));

fprintf('Hidden layer activation function %s\n',net.layers{1}.
    transferFcn);
fprintf('Output layer activation function %s\n',net.layers{2}.
    transferFcn);
```

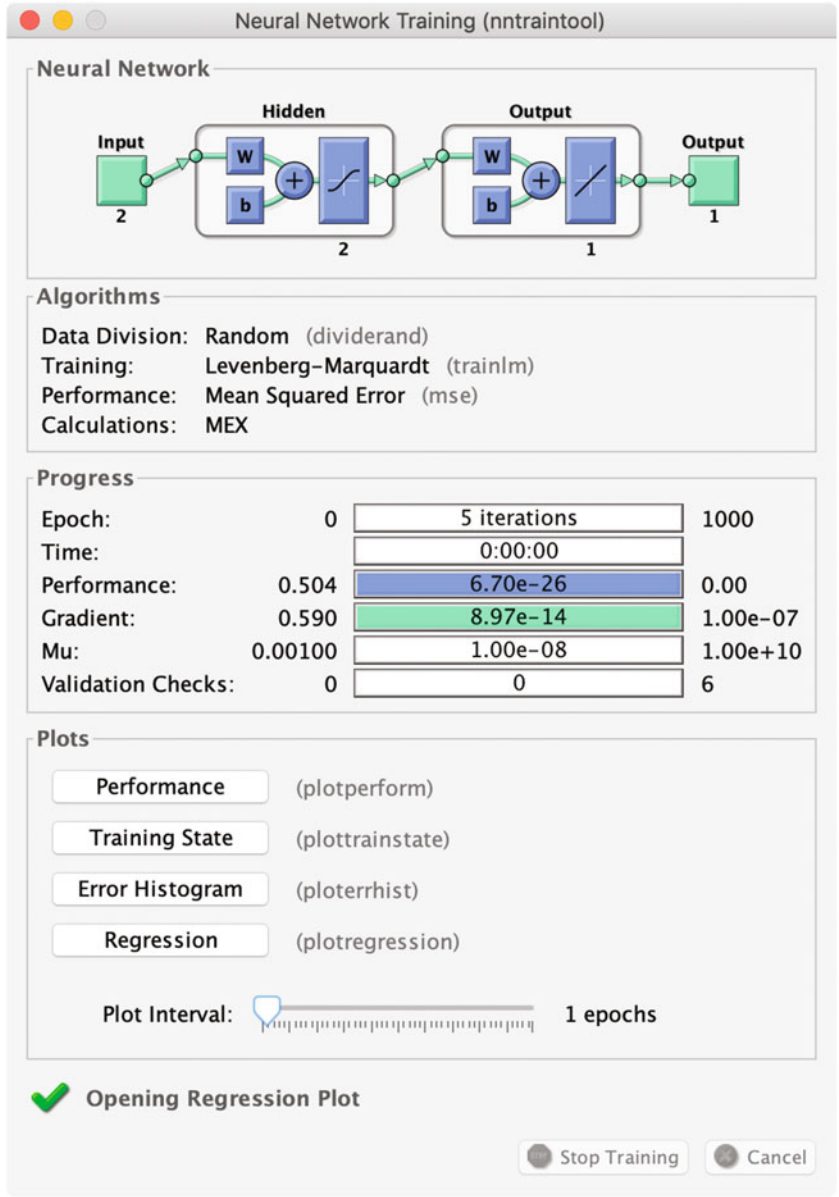
Running the script produces the MATLAB GUI shown in Figure 2.1.

As you can see, we have two inputs, one hidden layer and one output layer. The diagram indicates that our hidden layer activation function is nonlinear, while the output layer is linear. The GUI is interactive, and you can study the learning process by clicking the buttons. For example, if you click the performance button, you get Figure 2.2. Just about everything in the network development is customizable. The GUI is a real-time display. You can watch the training in progress. If you just want to look at the layout, type `view(net)`.

The three major boxes in the GUI are Algorithms, Progress, and Plots. Under **Algorithms** we have

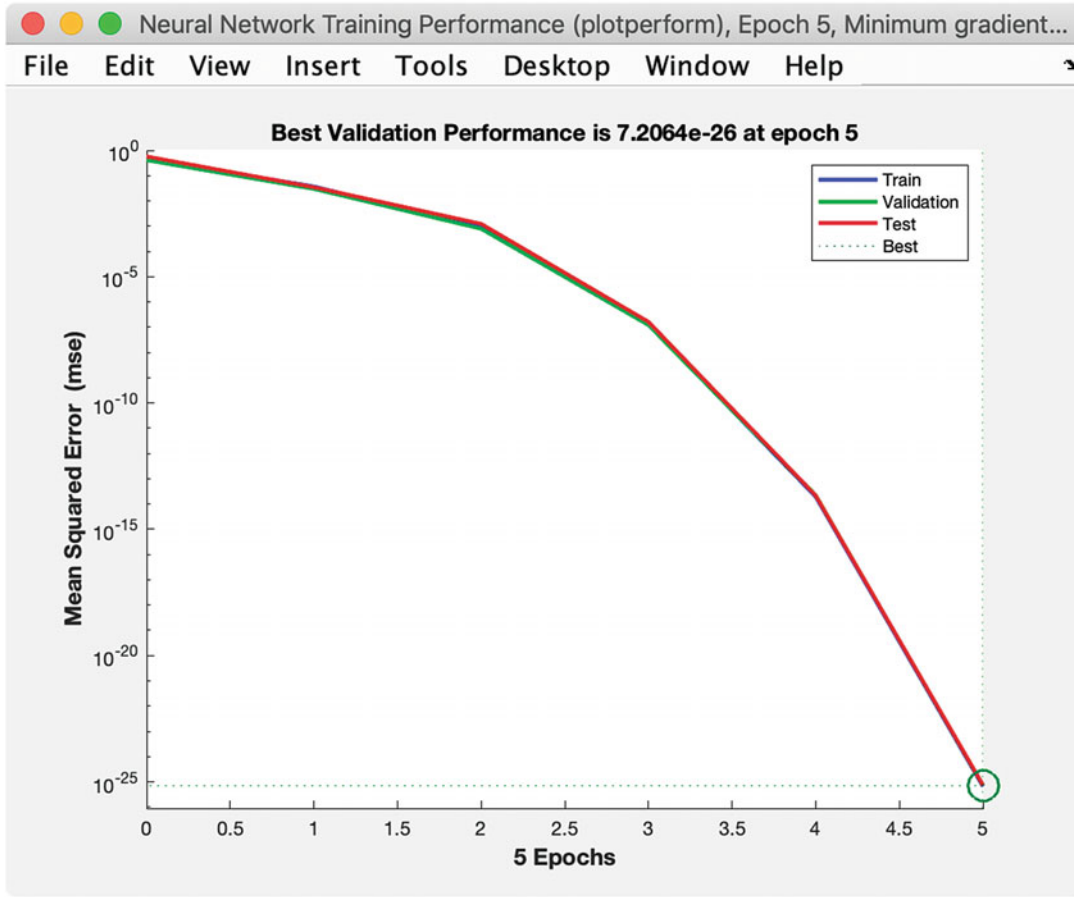
- **Data division**—Data division divides the data into training, testing, and validation sets. ‘‘Random” says that the division between the three categories is done randomly.
- **Training**—This shows the training method to be used.

Figure 2.1: Deep learning network GUI.



- **Performance**—This says that mean squared error is used to determine how well the network works. Other methods, such as maximum absolute error, could be used. Mean squared is useful because the error grows as the square of the deviation meaning that large errors are more heavily weighted.
- **Calculations**—This shows that the calculations are done via a MEX file, that is, in a C or C++ program.

Figure 2.2: Network training performance.



The **Progress** of the GUI is useful to watch during long training sessions. We are seeing it at the end.

- **Epoch**—Says five epochs were used. The range is 0 to 1000 epochs.
- **Time**—Gives you the clock time during training.
- **Performance**—Shows you the MSE performance during training.
- **Gradient**—This shows the gradient that shows the speed of training as discussed earlier.
- **Mu**—Mu is the control parameter for training the neural network.
- **Validation checks**—Shows that no validation checks failed.

The last section is **Plots**. There are four figures we can study to understand the process.

Figure 2.3: Network training state.

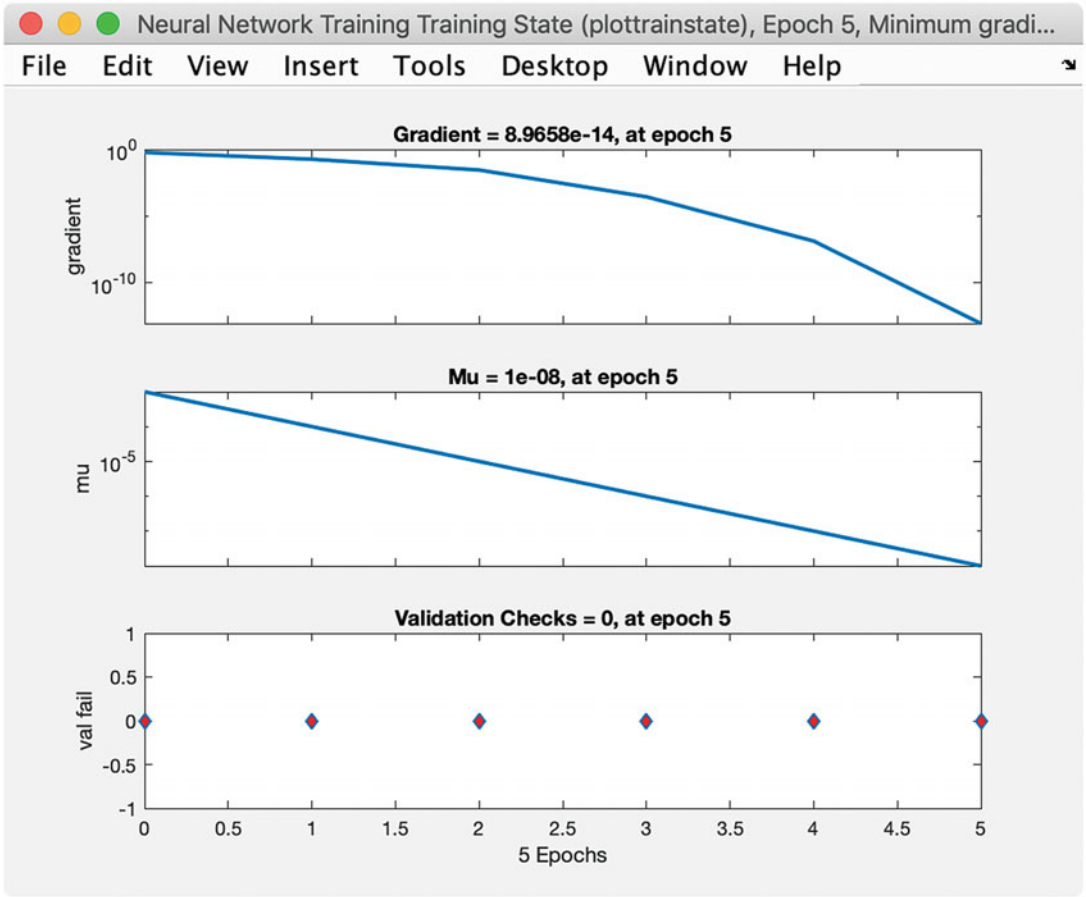


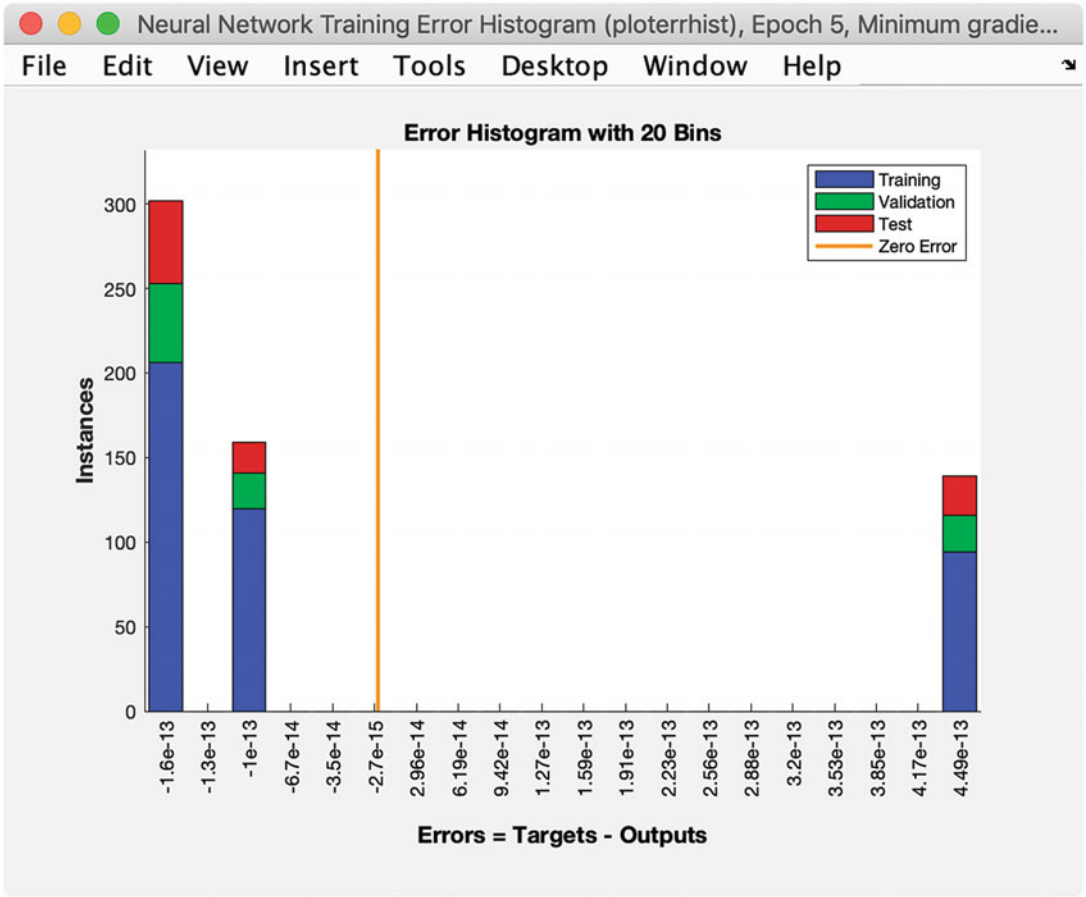
Figure 2.2 shows the training performance as a function of epoch. Mean squared error is the criteria. The test, validation, and training sets have their own lines. In this training, all have the same values.

Figure 2.3 shows the training state as a function of epoch. Five epochs are used. The titles show the final values in each plot. The top plot shows the progression of the gradient. It decreases with each epoch. The next shows mu decreasing linearly with epoch. The bottom plot shows that there were no validation failures during the training.

Figure 2.4 gives a training histogram. This shows the number of instances when one of the sets shows the error value on the x-axis. The bars are divided into training, validation, and test sets. Each number on the x-axis is a bin. Only three bins are occupied, in this case. The histogram shows that the training sets are more numerous than the testing or validation sets.

Figure 2.5 gives a training regression. There are four subplots: one for training sets, one for validation sets, one for test sets, and one for all sets. There are only two targets, 0 and 1.

Figure 2.4: Network training histogram.



The linear fit doesn't give much information in this case since we can only have a linear fit with two points. The plot title says we reached the minimum gradient after 5 epochs, that is, after passing all the cases through the training five times. The legend shows the data, the fit, and the Y=T plot which is the same as the linear in this system.

Typing

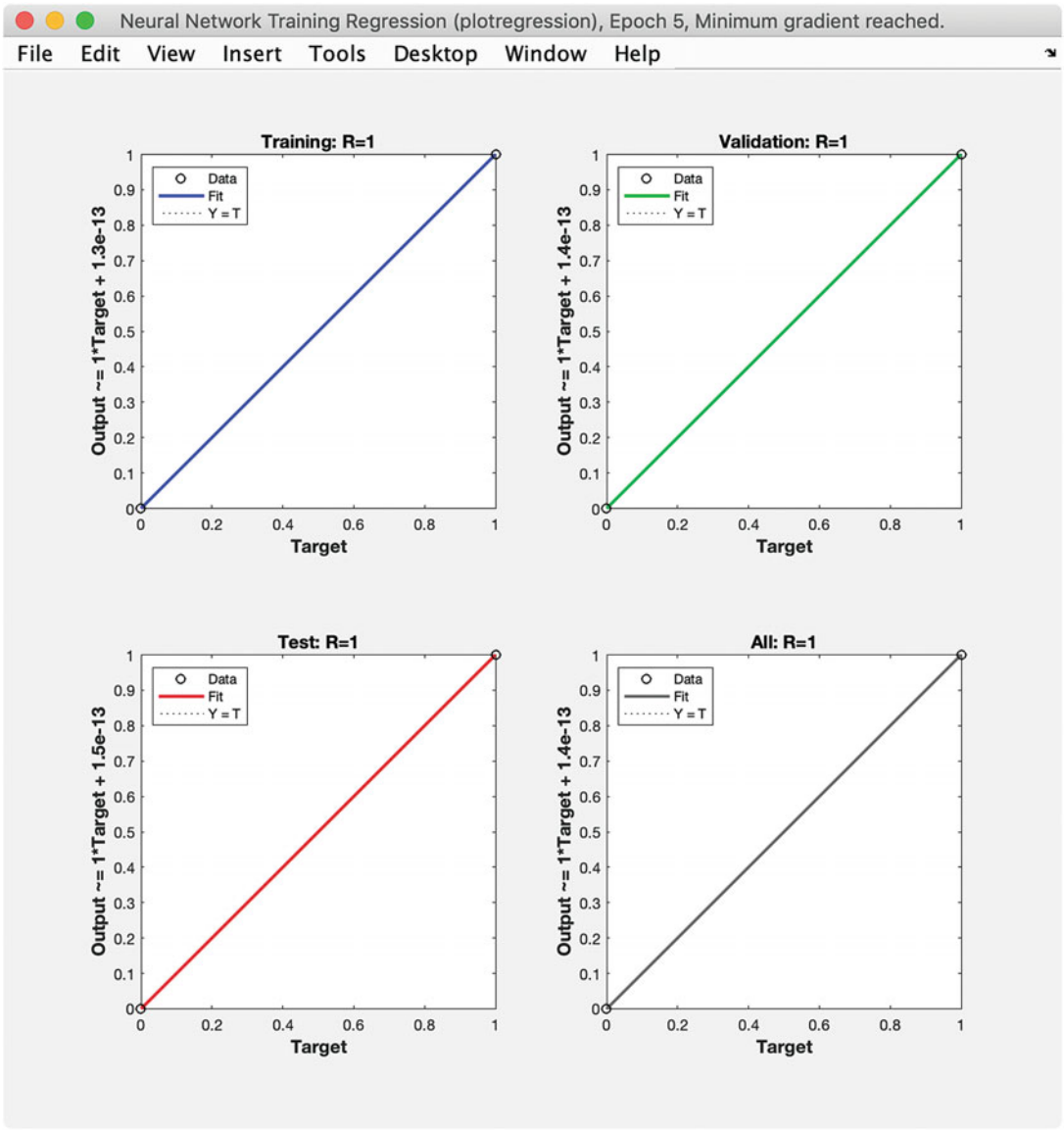
```
>> net = feedforwardnet(2);
```

creates the neural network data structure which is quite flexible and complex. The ‘‘2” means two neurons in one layer. If we wanted two layers with two neurons each, we would type

```
>>  net = feedforwardnet([2 2]);
```

We create 600 training sets. `net = configure(net, x, y);` configures the network. The configure function determines the number of inputs and outputs from the `x` and

Figure 2.5: Regression.



`y` arrays. The network is trained with `net = train(net,x,y);` and simulated with `c = sim(net,[a;b]);`. We extract the weights and biases from the cell arrays `net.IW`, `net.LW`, and `net.b`. ‘‘I” stands for input and ‘‘IW” for layer. Input is from the single input node to the two hidden nodes, and layer is from the two hidden nodes to the one output node.

Now the training sets are created randomly from the truth table. You can run this script many times, and usually you will get the right result, but not always. This is an example where it worked well.

```
>> DLXOR

    a     b    c
    1     1  0.00
    0     0  0.00
    1     0  1.00
    0     1  1.00

Hidden layer biases  1.735 -1.906
Output layer bias    1.193
Input layer weights   -2.15   1.45
                      -1.83   1.04
Output layer weights  -1.16   1.30
Hidden layer activation function tansig
Hidden layer activation function purelin
```

`tansig` is hyperbolic tangent.

Each run will result in different weights, even when the network gives the correct results. For example:

```
1  > DLXOR
2
3      a     b    c
4      1     1  0.00
5      0     0 -0.00
6      1     0  1.00
7      0     1  1.00
8
9  Hidden layer biases  4.178  0.075
10 Output layer bias   -1.087
11 Input layer weights   -4.49  -1.36
12                       -3.79  -3.67
13 Output layer weights   2.55  -2.46
```

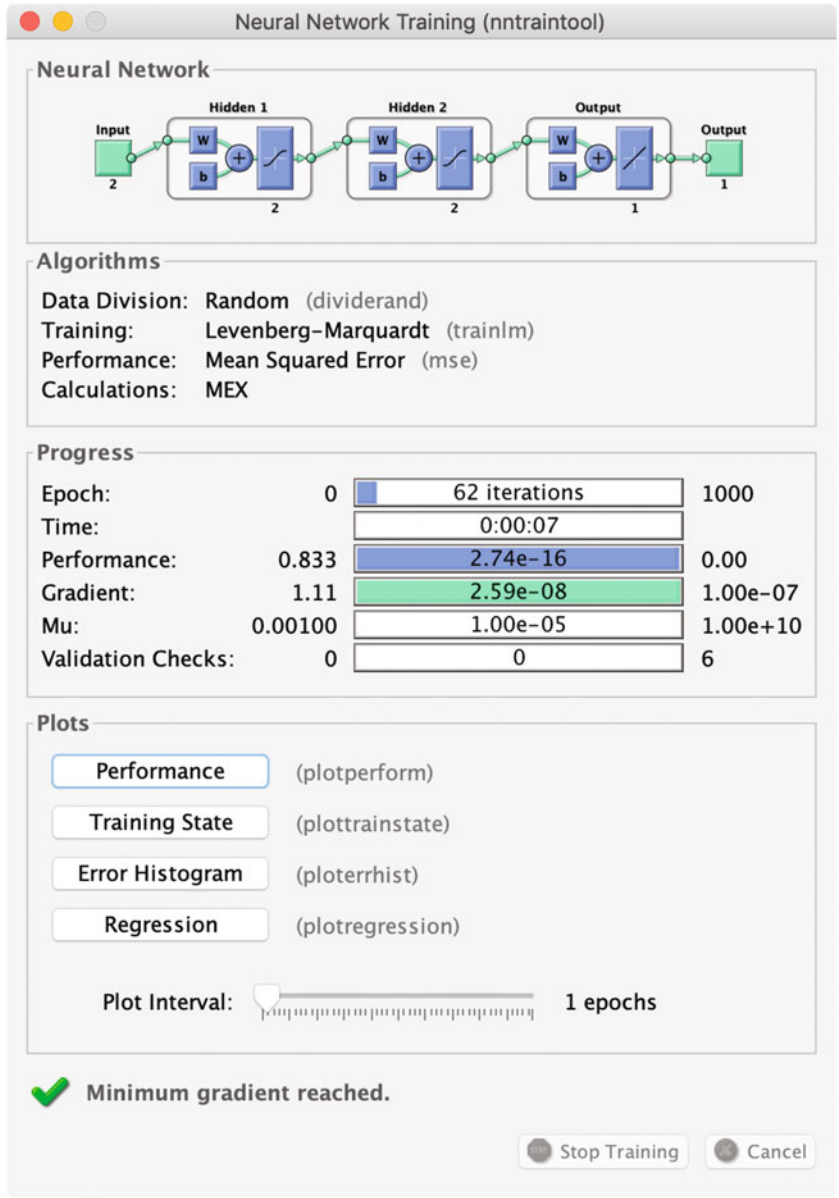
There are many possible sets of weights and biases because the weight/bias sets are not unique. Note that the 0.00 are really not 0. This means that used operationally, we would need to set a threshold such as

```
1  if( abs(c) < tol )
2    c = 0;
3  end
```

You might be interested in what happens if we add another layer to the network, by creating it with `net = feedforwardnet([2 2])`. Figure 2.6 shows the network in the GUI.

The additional hidden layer makes it easier for the neural net to fit the data from which it is learning. On the left are the two inputs, `a` and `b`. In each hidden layer, there is a weight `w` and bias `b`. Weights are always needed, but biases are not always used. Both hidden layers have nonlinear activation functions. The output layer produces the one output using a linear activation function.

Figure 2.6: Deep learning network GUI with two hidden layers.




```
>> DLXOR
    a      b    c
    1      1   0.00
    0      0   0.00
    1      0   1.00
    0      1   1.00
```

This produces good results too. We haven't explored all the diagnostic tools available when using `feedforwardnet`. There is a lot of flexibility in the software. You can change activation functions, change the number of hidden layers, and customize it in many different ways. This particular example is very simple as the input sets are limited to four possibilities.

We can explore what happens when the inputs are noisy, not necessarily all ones or zeros. We do this in `DLXORNoisy.m`, and the only difference from the original script is in lines 33--35 where we add Gaussian noise to the inputs.

DLXORNoisy.m

```
1  a          = a + 0.01*randn(1,4);
2  b          = b + 0.01*randn(1,4);
3  c          = sim(net,[a;b]);
```

The output from running this script is shown as follows.

```
>> DLXORNoisy

    a     b   c
 0.991  1.019 -0.003
 0.001 -0.005 -0.002
 0.996  0.009  0.999
-0.001  1.000  1.000

Hidden layer biases -1.793  2.135
Output layer bias   -1.158
Input layer weights    1.70   1.54
                       1.80   1.52
Output layer weights  -1.11   1.15
Hidden layer activation function tansig
Output layer activation function purelin
```

As one might expect, the outputs are not exactly one or zero.

2.4 Training

The neural net is a nonlinear system due to the nonlinear activation functions. The Levenberg-Marquardt training algorithm is one way of solving a nonlinear least squares problem. This algorithm only finds a local minimum which may or may not be a global minimum. Other algorithms, such as genetic algorithms, downhill simplex, simulated annealing, and so on. could also be used for finding the weights and biases. To achieve second-order training speeds, one has to compute the Hessian matrix. The Hessian matrix is a square matrix of second-order partial derivative of a scalar-valued function. Suppose we have a nonlinear function

$$f(x_1, x_2) \tag{2.1}$$

then the Hessian is

$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} \end{bmatrix} \tag{2.2}$$

x_k are weights and biases. This can be very expensive to compute. In the Levenberg-Marquardt algorithm, we make an approximation

$$H = J^T J \tag{2.3}$$

where

$$J = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} \end{bmatrix} \tag{2.4}$$

The approximate Hessian is

$$H = \begin{bmatrix} (\frac{\partial f}{\partial x_1})^2 & \frac{\partial f}{\partial x_1}\frac{\partial f}{\partial x_2} \\ \frac{\partial f}{\partial x_1}\frac{\partial f}{\partial x_2} & (\frac{\partial f}{\partial x_2})^2 \end{bmatrix} \tag{2.5}$$

This is an approximation of the second derivative. The gradient is

$$g = J^T e \tag{2.6}$$

where e is a vector of errors. The Levenberg-Marquardt uses the following algorithm to update the weights and biases:

$$x_{k+1} = x_k - \left[J^T J + \mu I\right]^{-1} J^T e \tag{2.7}$$

I is the identity matrix (a matrix with all diagonal elements equal to 1). If the parameter μ is zero, this is Newton's method. With a large μ, this becomes gradient descent which is faster. Thus μ is a control parameter. After a successful step, we decrease μ since we are in less need of the advantages of the faster gradient descent.

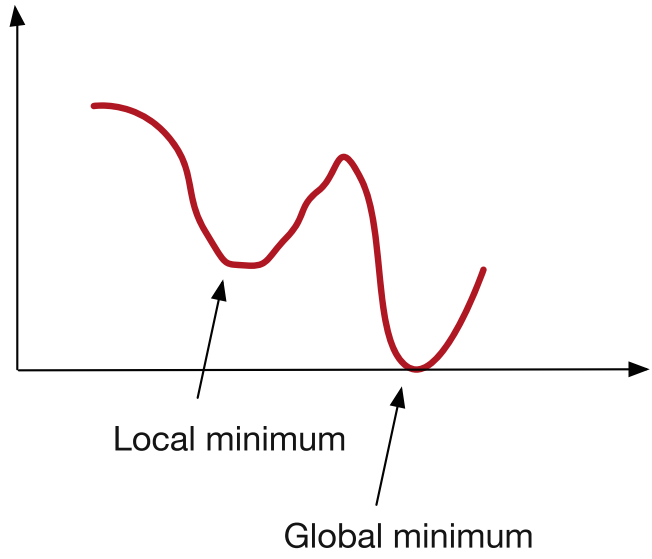
Why are gradients so important and why can they get us into trouble? Figure 2.7 shows a curve with a local and global minimum. If our search first enters the local minimum, the gradient is steep and will drive us to the bottom from which we might not get out. Thus would not have found the best solution.

The cost can be very complex even for simple problems. A famous problem that can give insight into the problem is Zermelo's problem which is discussed in Section 2.5.

2.5 Zermelo's Problem

Insight into issues of global optimization can be obtained by examining Zermelo's problem [7]. Zermelo's problem is a 2D trajectory problem of a vehicle at constant speed in a velocity field in which the velocity is a function of position, for example, a ship with a given maximum speed moving through strong currents. The magnitude and direction of the currents in each axis (u,v) are functions of the position: $u(x, y)$ and $v(x, y)$. The goal is to steer the ship to find a minimum time trajectory between two points. An analytical solution is possible for the case

Figure 2.7: Local and global minimums.



where only the current in the x direction is nonzero, and it is a linear function of the y position of the vehicle. The equations of motion f are

$$\begin{aligned} \dot{x} &= V\cos\theta + u(x,y) = V\cos\theta - V\frac{y}{h} \\ \dot{y} &= V\sin\theta + v(x,y) = V\sin\theta \end{aligned} \tag{2.8}$$

V is the velocity of the vehicle relative to the current which is constant, and θ is the angle of the vehicle relative to the x-axis and is the control in the problem. The problem has a characteristic dimension of h. The Hamiltonian of the system is

$$H = \lambda_x(V\cos\theta + u) + \lambda_y(V\sin\theta + v) + 1 \tag{2.9}$$

The solution of the optimal control problem requires the solution of the following equations:

$$\dot{x} = f(x,u,t) \tag{2.10}$$

$$\dot{\lambda}(t)^T = -\frac{\partial H}{\partial x} \tag{2.11}$$

$$\frac{\partial H}{\partial u} = 0 \tag{2.12}$$

If the final time is not constrained and the Hamiltonian is not an explicit function of time, then

$$H(t) = 0 \tag{2.13}$$

The costate equations are

$$\dot{\lambda} = -\frac{\partial f}{\partial x}^T \lambda \tag{2.14}$$

where the boundary conditions of the partials with respect to the vector x are unknown. The optimality condition from (2.12) becomes

$$0 = \frac{\partial f}{\partial u}^T \lambda \tag{2.15}$$

where the subscript denotes the partial with respect to the control vector u which provides a relationship between the controls and the costates.

Applying the costate equations ((2.14)), we first compute the partials of the state equations

$$\frac{\partial f}{\partial x} = \begin{bmatrix} 0 & -V/h \\ 0 & 0 \end{bmatrix} \tag{2.16}$$

so applying the transpose to the partials matrix and substituting in, we then have the costate derivatives.

$$\dot{\lambda} = -\begin{bmatrix} 0 & 0 \\ -V/h & 0 \end{bmatrix} \begin{bmatrix} \lambda_x \\ \lambda_y \end{bmatrix} = \begin{bmatrix} 0 \\ \lambda_x \frac{V}{h} \end{bmatrix} \tag{2.17}$$

We then compute the partials of the state equations with respect to the control

$$\frac{\partial f}{\partial u} = \begin{bmatrix} -V \sin\theta \\ V \cos\theta \end{bmatrix} \tag{2.18}$$

so that the control angle θ can then be computed from the optimality condition in (2.15).

$$\begin{bmatrix} -V \sin\theta & V \cos\theta \end{bmatrix} \begin{bmatrix} \lambda_x \\ \lambda_y \end{bmatrix} = 0 \tag{2.19}$$

$$\tan\theta = \frac{\lambda_y}{\lambda_x}$$

The preceding equations are used for the indirect optimization method. We can also compute the analytical solution for this problem when the final position is the origin (0,0). The optimal control angle as a function of the current position is expressed as

$$\frac{y}{h} = \sec\theta - \sec\theta_f \tag{2.20}$$

$$\frac{x}{h} = -\frac{1}{2}\left[\sec\theta_f(\tan\theta_f - \tan\theta) - \tan\theta_f(\sec\theta_f - \sec\theta) + \log\frac{\tan\theta_f + \sec\theta_f}{\tan\theta + \sec\theta}\right] \tag{2.21}$$

where θ_f is the final control angle and log is $\log_e$. These equations enable us to solve for the initial and final control angles θ_0 and θ_f given an initial position. Note that this corrects a sign error in the text.

Figure 2.8: Zermelo's problem cost.

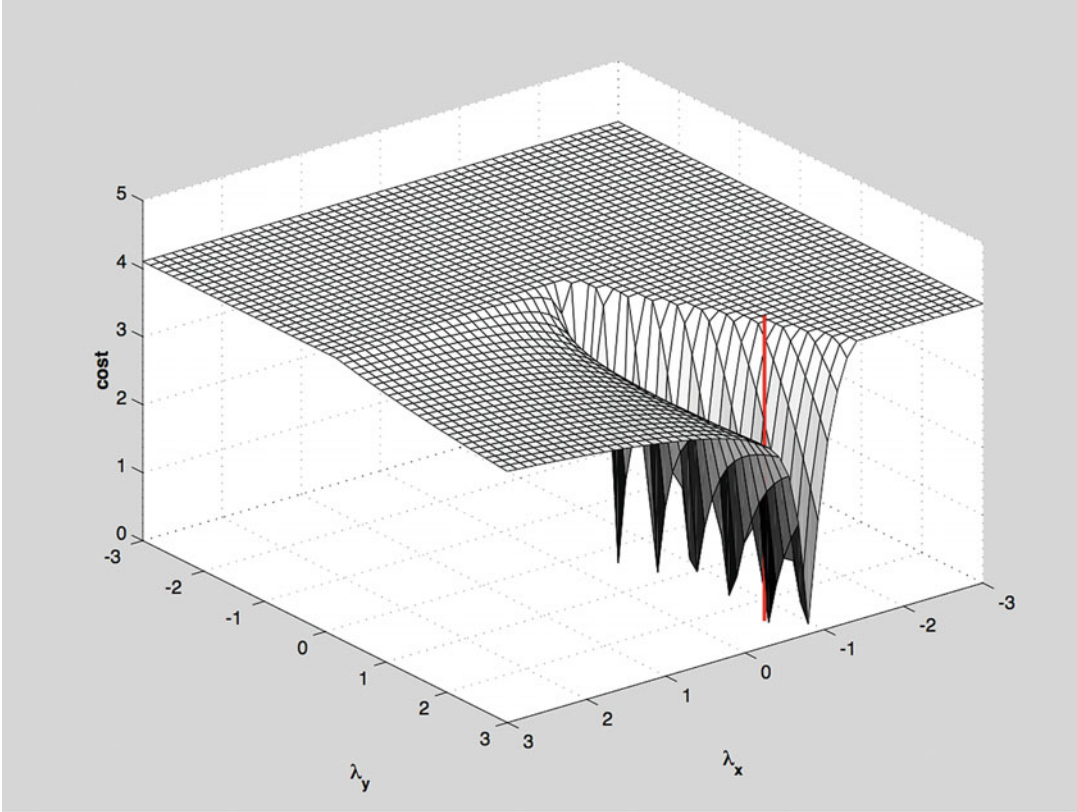


Table 2.1: Solutions.

Test	λ_x	λ_y	**Ratio**
Analytical	-0.5	1.866025	-0.26795
Downhill simplex	-0.65946	2.9404	-0.22428
Simulated annealing	-0.68652	2.4593	-0.27915
Genetic algorithm	-0.78899	2.9404	-0.26833

The cost, while appearing simple, produces a very complex surface as shown in Figure 2.8. There are very flat regions and then a series of deep valleys.

For each method being tested, the optimization parameters have been chosen, by a certain amount of trial and error, to get the best results. The final λ vector is different in each case. However, the control is determined by the ratio so the magnitudes are not important. Table 2.1 gives the analytical and numerical solutions for the problem. The initial conditions are $[3.66; -1.86]$ and the final conditions are $[0; 0]$.

CHAPTER 3

Finding Circles with Deep Learning

3.1 Introduction

Finding circles is a classification problem. Given a bunch of geometric shapes, we want the deep learning system to classify a shape as either a circle or something else. This is much simpler than classifying faces or digits. It is a good way to determine how well your classification system works. We will apply a convolutional network to the problem as it is the most appropriate for classifying image data.

In this chapter, we will first generate a set of image data. This will be a set of ellipses, a subset of which will be circles. Then we will build the neural net, using convolution, and train it to identify the circles. Finally, we will test the net and try some different options for training options and layer architecture.

3.2 Structure

The convolutional network consists of multiple layers. Each layer has a specific purpose. The layers may be repeated with different parameters as part of the convolutional network. The layer types we will use are

1. imageInputLayer
2. convolution2dLayer
3. batchNormalizationLayer
4. reluLayer
5. maxPooling2dLayer
6. fullyConnectedLayer
7. softmaxLayer
8. classificationLayer

M. Paluszek and S. Thomas, *Practical MATLAB Deep Learning*,
https://doi.org/10.1007/978-1-4842-5124-9_3

You can have multiple layers of each type of layer. Some convolutional nets have hundreds of layers. Krizhevsky [1] and Bai [3] give guidelines for organizing the layers. Studying the loss in the training and validation can guide you to improving your neural network.

3.2.1 imageInputLayer

This tells the network the size of the images. For example:

```
layer = imageInputLayer([28 28 3]);
```

says the image is RGB and 28 by 28 pixels.

3.2.2 convolution2dLayer

Convolution is the process of highlighting expected features in an image. This layer applies sliding convolutional filters to an image to extract features. You can specify the filters and the stride. Convolution is a matrix multiplication operation. You define the size of the matrices and their contents. For most images, like images of faces, you need multiple filters. Some types of filters are

1. Blurring filter `ones(3,3)/9`
2. Sharpening filter `[0 -1 0;-1 5 -1;0 -1 0]`
3. Horizontal Sobel filter for edge detection `[-1 -2 -1; 0 0 0; 1 2 1]`
4. Vertical Sobel filter for edge detection `[-1 0 1;-2 0 2;-1 0 1]`

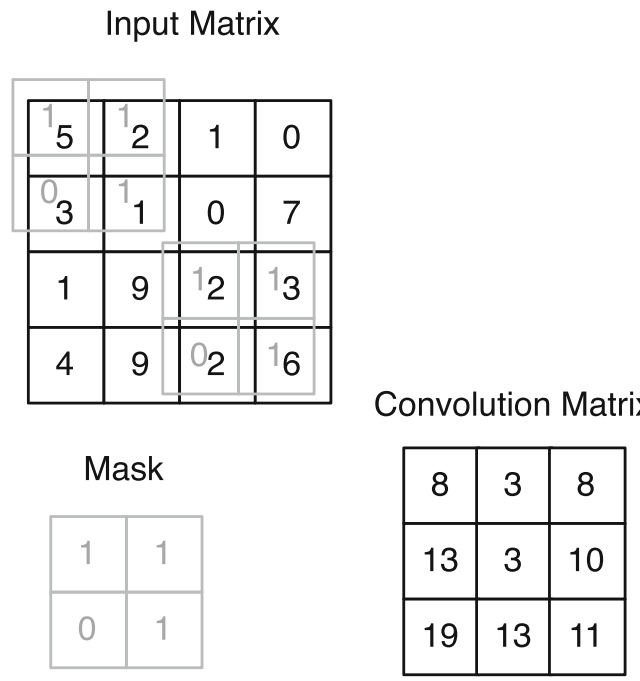
We create an n-by-n mask that we apply to an m-by-m matrix of data where m is greater than n. We start in the upper left corner of the matrix, as shown in Figure 3.1. We multiply the mask times the corresponding elements in the input matrix and do a double sum. That is the first element of the convolved output. We then move it column by column until the highest column of the mask is aligned with the highest column of the input matrix. We then return it to the first column and increment the row. We continue until we have traversed the entire input matrix and our mask is aligned with the maximum row and maximum column.

The mask represents a feature. In effect we are seeing if the feature appears in different areas of the image. Here is an example. We have a 2 by 2 mask with an L. `Convolution` demonstrates convolution.

Convolution.m

```
%% Demonstrate convolution

filter = [1 0;1 1]
image  = [0 0 0 0 0 0;...
          0 0 0 0 0 0;...
          0 0 1 0 0 0;...
          0 0 1 1 0 0;...
          0 0 0 0 0 0]
```

Figure 3.1: Convolution process showing the mask at the beginning and end of the process.




```
9
10  out = zeros(3,3);
11
12  for k = 1:4
13    for j = 1:4
14      g = k:k+1;
15      f = j:j+1;
16      out(k,j) = sum(sum(filter.*image(g,f)));
17    end
18  end
19
20  out
```

The 3 appears where the ‘‘L” is in the image.

```
>> Convolution
filter =
     1     0
     1     1
image =
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     1     0     0     0
     0     0     1     1     0     0
```

```
     0     0     0     0     0     0
out =
     0     0     0     0
     0     1     1     0
     0     1     3     1
     0     0     1     1
```

We can have multiple masks. There is one bias and one weight for each element of the mask for each feature. In this case, the convolution works on the image itself. Convolutions can also be applied to the output of other convolutional layers or pooling layers. Pooling layers further condense the data. In deep learning, the masks are determined as part of the learning process. Each pixel in a mask has a weight and may have a bias; these are computed from the learning data. Convolution should be highlighting important features in the data. Subsequent convolution layers narrow down features. The MATLAB function has two inputs: the `filterSize`, specifying the height and width of the filters as either a scalar or an array of `[h w]`, and `numFilters`, the number of filters.

3.2.3 batchNormalizationLayer

A batch normalization layer normalizes each input channel across a mini-batch. It automatically divides up the input channel into batches. This reduces the sensitivity to the initialization.

3.2.4 reluLayer

`reluLayer` is a layer that uses the rectified linear unit activation function.

$$f(x) = \begin{cases} x & x >= 0 \\ 0 & x < 0 \end{cases} \tag{3.1}$$

Its derivative is

$$\frac{df}{dx} = \begin{cases} 1 & x >= 0 \\ 0 & x < 0 \end{cases} \tag{3.2}$$

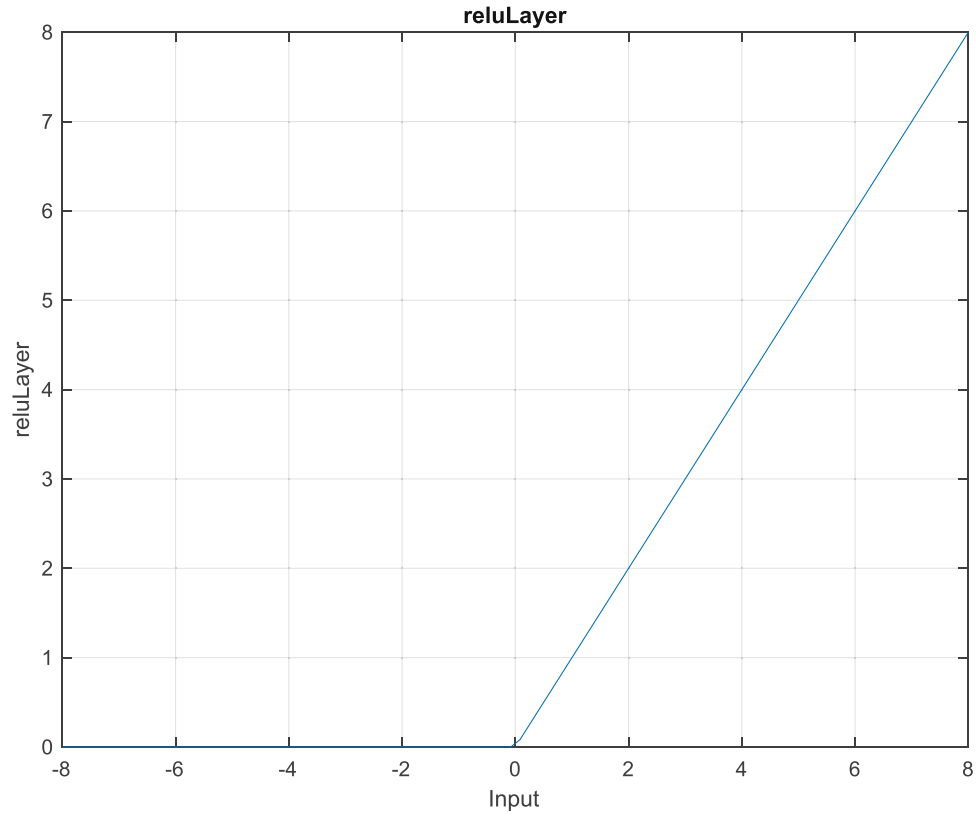
This is very fast to compute. It says that the neuron is only activated for positive values, and the activation is linear for any value greater than zero. You can adjust the activation point with a bias. This code snippet generates a plot of `reluLayer`:

```
x = linspace(-8,8);
y = x;
y(y<0) = 0;
PlotSet(x,y,'x label','Input','y label','reluLayer','plot title','
    reluLayer')
```

Figure 3.2 shows the activation function. An alternative is a leaky reluLayer where the value is not zero below zero. Now the difference in the `y` computation in the snippet:

```
x = linspace(-8,8);
y = x;
```

Figure 3.2: reluLayer.




```
y(y<0) = 0.01*x(y<0);
PlotSet(x,y,'x label','Input','y label','reluLayer','plot title','leaky
    reluLayer')
```

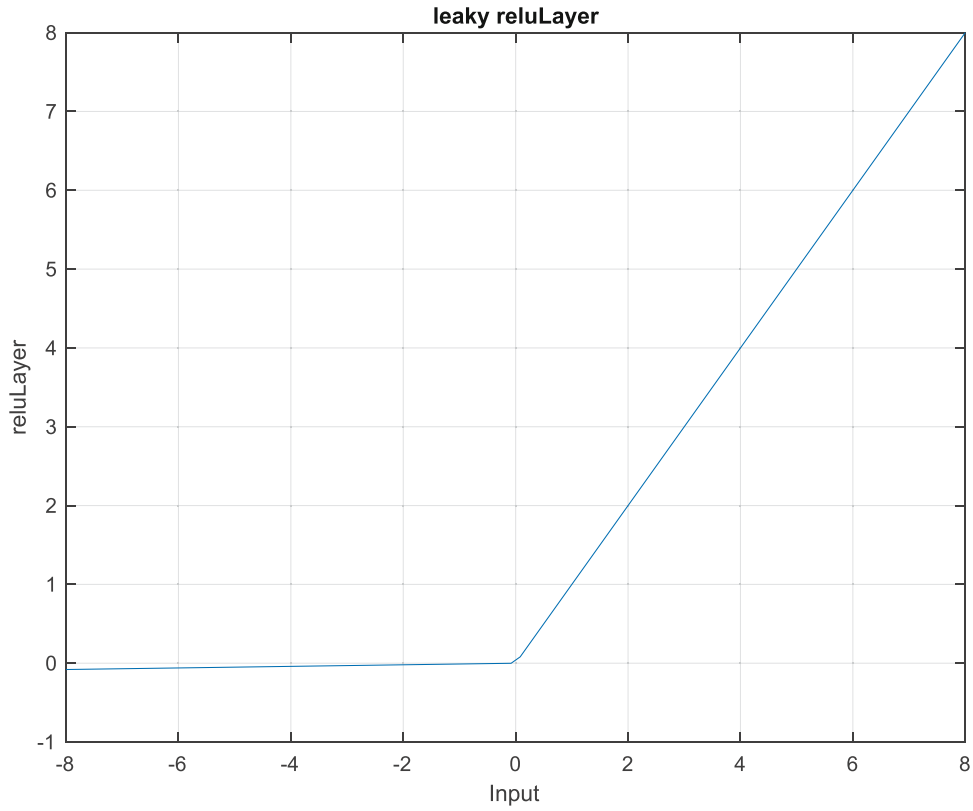
Figure 3.3 shows the leaky function. Below zero it has a slight slope.

A leaky Relu layer solves the dead Relu problem where the network stops learning because the inputs to the activation problem are below zero, or whatever the threshold might be. It should let you worry a bit less on how you initialize the network.

3.2.5 maxPooling2dLayer

`maxPooling2dLayer` creates a layer that breaks the 2D input into rectangular pooling regions and outputs the maximum value of each region. The input `poolSize` specifies the width and height of a pooling region. `poolSize` can have one element (for square regions) or two for rectangular regions. This is a way to reduce the number of inputs that need to be evaluated. Typical images have to be more than a mega-pixel in size, and it is not practical to use all as inputs. Furthermore, most images, or two-dimensional entities of any sort, don't really have enough information to require finely divided regions. You can experiment with pooling and see how it works for your application. An alternative is `averagePooling2dLayer`.

Figure 3.3: Leaky reluLayer.



3.2.6 fullyConnectedLayer

The fully connected layer connects all of the inputs to the outputs with weights and biases. For example:

```
layer = fullyConnectedLayer(10);
```

creates ten outputs from any number of inputs. You don't have to specify the inputs. Effectively, this is the equation:

$$y = ax + b \tag{3.3}$$

If there are m inputs and n outputs, b is a column bias matrix of length n and a is n by m.

3.2.7 softmaxLayer

softmax finds a maximum of a set of values using the logistic function. The softmax is the maximum value of the set

$$p_k = e^{\frac{q_k}{\sum e^{q_k}}} \tag{3.4}$$

```
>> q = [1,2,3,4,1,2,3]

q =

     1     2     3     4     1     2     3

>> d = sum(exp(q));
>> p = exp(q)/d

p =

    0.0236    0.0643    0.1747    0.4748    0.0236    0.0643    0.1747
```

In this case, the maximum is element 4 in both cases. This is just a method of smoothing the inputs. Softmax is used for multiclass classification because it guarantees a well-behaved probability distribution. Well behaved means that the sum of the probabilities is 1.

3.2.8 classificationLayer

A classification layer computes the cross-entropy loss for multiclass classification problems with mutually exclusive classes. Let us define loss. Loss is the sum of the errors in training the neural net. It is not a percentage. For classification the loss is usually the negative log likelihood, which is

$$L(y) = -\log(y) \tag{3.5}$$

where y is the output of the softmax layer.

For regression it is the residual sum of squares. A high loss means a bad fit.

Cross-entropy loss means that an item being classified can only be in one class. The number of classes is inferred from the output of the previous layer. In this problem, we have only two classes, circle or ellipse, so the number of outputs of the previous layer must be 2. Cross-entropy is the distance between the original probability distribution and what the model believes it should be. It is defined as

$$H(y, p) = -\sum_i y_i \log p_i \tag{3.6}$$

where i is the index for the class. It is a widely used replacement for mean squared error. It is used in neural nets where softmax activations are in the output layer.

3.2.9 Structuring the Layers

For our first net to identify circles, we will use the following set of layers. The first layer is the input layer, for the 32x32 images. These are relatively low-resolution images. You can visually determine which are ellipses or circles so we would expect the neural network to be able to do the same. Nonetheless, the size of the input images is an important consideration. In our case, our images are tightly cropped around the shape. In a more general problem, the subject of interest, a cat, for example, might be in a general setting.

We use a `convolution2dLayer`, `batchNormalizationLayer`, and `reluLayer` in sequence, with a pool layer in between. There are three sets of convolution layers, each with an increasing number of filters. The output set of layers consists of a `fullyConnectedLayer`, `softmaxLayer`, and finally, the `classificationLayer`.

EllipsesNeuralNet.m

```
%% Define the layers for the net
% This gives the structure of the convolutional neural net
layers = [
    imageInputLayer(size(img))

    convolution2dLayer(3,8,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,16,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,32,'Padding','same')
    batchNormalizationLayer
    reluLayer

    fullyConnectedLayer(2)
    softmaxLayer
    classificationLayer
        ];
```

3.3 Generating Data: Ellipses and Circles

3.3.1 Problem

We want to generate images of ellipses and circles of arbitrary sizes and with different thicknesses in MATLAB.

3.3.2 Solution

Write a MATLAB function to draw circles and ellipses and extract image data from the figure window. Our function will create a set of ellipses and a fixed number of perfect circles as specified by the user. The actual plot and the resulting downsized image will both be shown in a figure window so you can track progress and verify that the images look as expected.

3.3.3 How It Works

This is implemented in `GenerateEllipses.m`. The output of the function is a cell array with both the ellipse data and a set of image data obtained from a MATLAB figure using `getframe`. The function also outputs the type of image, that is, the ''truth'' data.

GenerateEllipses.m

```
%% GENERATEELLIPSES Generate random ellipses
%% Form
%  [d, v] = GenerateEllipses(a,b,phi,t,n,nC,nP)
%% Description
% Generates random ellipses given a range of sizes and max rotation.
    The number
% of ellipses and circles must be specified; the total number generated
    is their
% sum. Opens a figure which displays the ellipse images in an animation
    after
% they are generated.
%% Inputs
%  a    (1,2) Range of a sizes of ellipse
%  b    (1,2) Range of b sizes of ellipse
%  phi  (1,1) Max rotation angle of ellipse
%  t    (1,1) Max line thickness in the plot of the circle
%  n    (1,1) Number of ellipses
%  nC   (1,1) Number of circles
%  nP   (1,1) Number of  pixels, image is nP by nP
%
%% Outputs
%  d      {:,2} Ellipse data and image frames
%  v      (1,:) Boolean for circles, 1 (circle) or 0 (ellipse)
```

The first section of the code generates random ellipses and circles. They are all centered in the image.

GenerateEllipses.m

```
nE    = n+nC;
d     = cell(nE,2);
r     = 0.5*(mean(a) + mean(b))*rand(1,nC)+a(1);
a     = (a(2)-a(1))*rand(1,n) + a(1);
b     = (b(2)-b(1))*rand(1,n) + b(1);
phi   = phi*rand(1,n);
cP    = cos(phi);
sP    = sin(phi);
theta = linspace(0,2*pi);
c     = cos(theta);
s     = sin(theta);
m     = length(c);
t     = 0.5+(t-0.5)*rand(1,nE);
aMax  = max([a(:);b(:);r(:)]);

% Generate circles
for k = 1:nC
  d{k,1} = r(k)*[c;s];
end

% Generate ellipses
for k = 1:n
  d{k+nC,1} = [cP(k) sP(k);-sP(k) cP(k)]*[a(k)*c;b(k)*s];
end

% True if the object is a circle
v       = zeros(1,nE);
v(1:nC) = 1;
```

The next section produces a 3D plot showing all the ellipses and circles. This is just to show you what you have produced. The code puts all the ellipses between z $\pm$ 1. You might want to adjust this when generating larger numbers of ellipses.

```
% 3D Plot
NewFigure('Ellipses');
z  = -1;
dZ = 2*abs(z)/nE;
o  = ones(1,m);
for k = 1:length(d)
  z   = z + dZ;
  zA  = z*o;
  plot3(d{k}(1,:),d{k}(2,:),zA,'linewidth',t(k));
  hold on
end
grid on
rotate3d on
```

The next section converts the frames to `nP` by `nP` sized images in grayscale. We set the figure and the axis to be square, and set the axis to `'equal'`, so that the circles will have the

correct aspect ratio and in fact be circular in the images. Otherwise, they would also appear as ellipses, and our neural net would not be able to categorize them. This code block also draws the resulting resized image on the right-hand side of the window, with a title showing the current step. There is a brief pause between each step. In effect, it is an animation that serves to inform you of the script's progress.

```
% Create images - this might take a while for a lot of images
f = figure('Name','Images','visible','on','color',[1 1 1]);
ax1 = subplot(1,2,1,'Parent', f, 'box', 'off','color',[1 1 1] );
ax2 = subplot(1,2,2,'Parent',f); grid on;
for k = 1:length(d)
  % Plot the ellipse and get the image from the frame
  plot(ax1,d{k}(1,:),d{k}(2,:),'linewidth',t(k),'color','k');
  axis(ax1,'off'); axis(ax1,'equal');
  axis(ax1,aMax*[-1 1 -1 1])
  frame  = getframe(ax1); % this call is what takes time
  imSmall = rgb2gray(imresize(frame2im(frame),[nP nP]));
  d{k,2} = imSmall;
  % plot the resulting scaled image in the second axes
        imagesc(ax2,d{k,2});
  axis(ax2,'equal')
        colormap(ax2,'gray');
  title(ax2,['Image ' num2str(k)])
        set(ax2,'xtick',1:nP)
        set(ax2,'ytick',1:nP)
        colorbar(ax2)
        pause(0.2)
end
close(f)
```

The conversion is `rgb2gray(imresize(frame2im(frame),[nP nP]))`, which performs these steps:

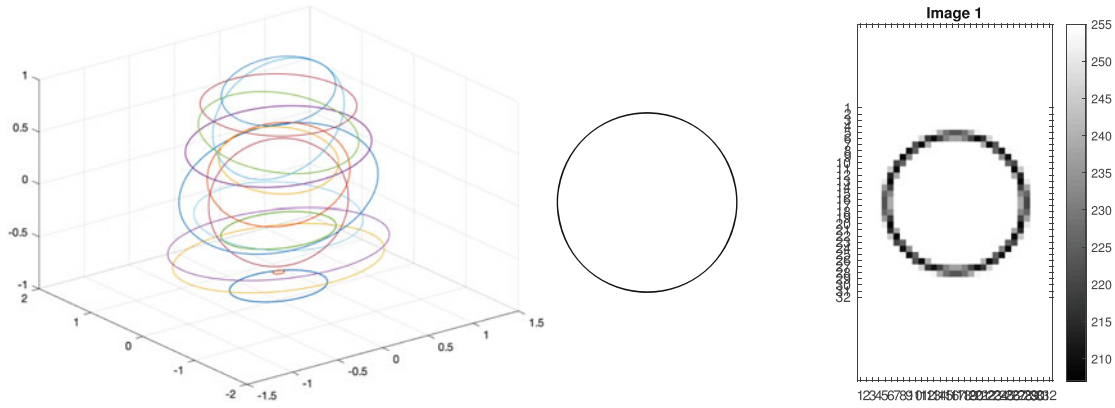
1. Get the frame with `frame2im`
2. Resize to `nP` by `nP` using `imresize`
3. Convert to grayscale using `rgb2gray`

Note that the image data originally ranges from 0 (black) to 255 (white), but is averaged to ligher gray pixels during the resize operation. The colorbar in the progress window shows you the span of the output image. The image looks black as before since it is plotted with `imagesc`, which automatically scales the image to use the entire colormap—in this case, the gray colormap.

The built-in demo generates ten ellipses and five circles.

```
function Demo

a   = [0.5 1];
```

Figure 3.4: Ellipses and a resulting image.




```
4  b    = [1 2];
5  phi  = pi/4;
6  t    = 3;
7  n    = 10;
8  nC   = 5;
9  nP   = 32;
10
11 GenerateEllipses(a,b,phi,t,n,nC,nP);
```

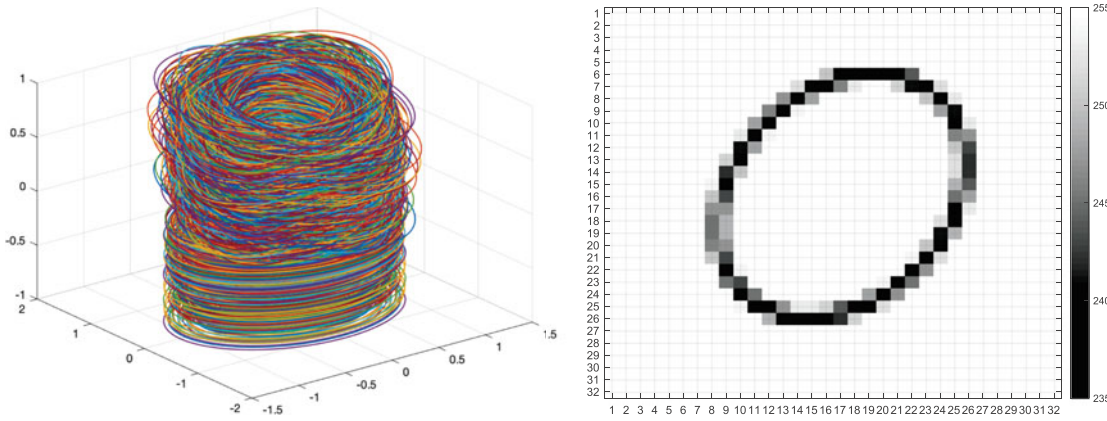
Figure 3.4 shows the generated ellipses and the first image displayed.

The script `CreateEllipses.m` generates 100 ellipses and 100 circles and stores them in the `Ellipses` folder along with the type of each image. Note that we have to do a small trick with the filename. If we simply append the image number to the filename, 1, 2, 3, ... 200, the images will not be in this order in the datastore; in alphabetical order, the images would be sorted as 1, 10, 100, 101, and so on. In order to have the filenames in alphabetical order match the order we are storing with the type, we generate a number a factor of 10 higher than the number of images and add it to the image index before appending it to the file. Now we have image 1001, 1001, and so on.

CreateEllipses.m

```
1  %% Create ellipses to train and test the deep learning algorithm
2  % The ellipse images are saved  as jpegs in the folder Ellipses.
3
4  % Parameters
5  nEllipses = 1000;
6  nCircles  = 1000;
7  nBits     = 32;
8  maxAngle  = pi/4;
9  rangeA    = [0.5 1];
10 rangeB    = [1 2];
11 maxThick  = 3.0;
```

Figure 3.5: Ellipses and a resulting image. 100 circles and 100 ellipse images are stored.



```
tic
[s, t] = GenerateEllipses(rangeA,rangeB,maxAngle,maxThick,nEllipses,
    nCircles,nBits);
toc
cd Ellipses
kAdd = 10^ceil(log10(nEllipses+nCircles)); % to make a serial number
for k = 1:length(s)
  imwrite(s{k,2},sprintf('Ellipse%d.jpg',k+kAdd));
end

% Save the labels
save('Type','t');
cd ..
```

The graphical output is shown in Figure 3.5. It first displays the 100 circles and then the 100 ellipses. It takes some time for the script to generate all the images.

If you open the resulting jpegs, you will see that they are in fact 32x32 images with gray circles and ellipses.

This recipe provides the data that will be used for the deep learning examples in the following sections. You must run `CreateEllipses.m` before you can run the neural net examples.

3.4 Training and Testing

3.4.1 Problem

We want to train and test our deep learning algorithm on a wide range of ellipses and circles.

3.4.2 Solution

The script that creates, trains, and tests the net is `EllipsesNeuralNet.m`.

3.4.3 How It Works

First we need to load in the generated images. The script in Recipe 3.3 generates 200 files. Half are circles and half ellipses. We will load them into an image datastore. We display a few images from the set to make sure we have the correct data and it is tagged correctly—that is, that the files have been correctly matched to their type, circle (1) or ellipse (0).

EllipsesNeuralNet.m

```
%% Get the images
cd Ellipses
type = load('Type');
cd ..
t    = categorical(type.t);
imds = imageDatastore('Ellipses','labels',t);

labelCount = countEachLabel(imds);

% Display a few ellipses
NewFigure('Ellipses')
n = 4;
m = 5;
ks = sort(randi(length(type.t),1,n*m)); % random selection
for i = 1:n*m
        subplot(n,m,i);
        imshow(imds.Files{ks(i)});
  title(sprintf('Image %d: %d',ks(i),type.t(ks(i))))
end

% We need the size of the images for the input layer
img = readimage(imds,1);
```

Once we have the data, we need to create the training and testing sets. We have 100 files with each label (0 or 1, for an ellipse or circle). We create a training set of 80% of the files and reserve the remaining as a test set using `splitEachLabel`. Labels could be names, like ‘‘circle’’ and ‘‘ellipse.’’ You are generally better off with descriptive ‘‘labels.’’ After all, a 0 or 1 could mean anything. The MATLAB software handles many types of labels.

EllipsesNeuralNet.m

```
% Split the data into training and testing sets
fracTrain             = 0.8;
[imdsTrain,imdsTest]  = splitEachLabel(imds,fracTrain,'randomize');
```

The layers of the net are defined as in the previous recipe. The next step is training. The `trainNetwork` function takes the data, set of layers, and options, runs the specified training

algorithm, and returns the trained network. This network is then invoked with the `classify` function, as shown later in this recipe. This network is a series network. The network has other methods which you can read about in the MATLAB documentation.

EllipsesNeuralNet.m

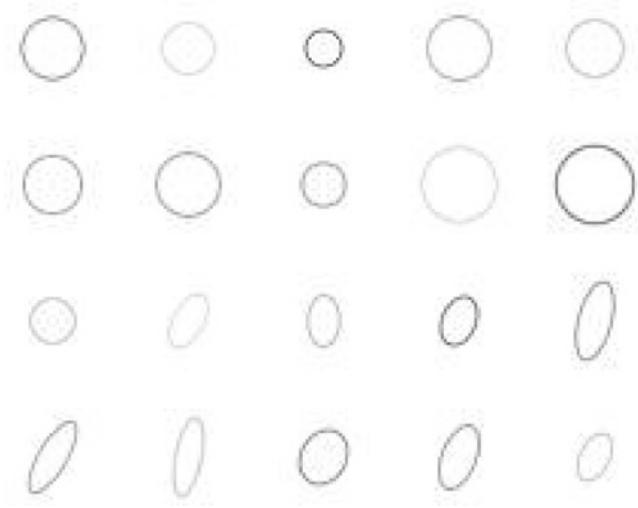
```
%% Training
% The mini-batch size should be less than the data set size; the mini-
    batch is
% used at each training iteration to evaluate gradients and update the
    weights.
options = trainingOptions('sgdm', ...
    'InitialLearnRate',0.01, ...
    'MiniBatchSize',16, ...
    'MaxEpochs',5, ...
    'Shuffle','every-epoch', ...
    'ValidationData',imdsTest, ...
    'ValidationFrequency',2, ...
    'Verbose',false, ...
    'Plots','training-progress');

net = trainNetwork(imdsTrain,layers,options);
```

Figure 3.6 shows some of the ellipses used in the testing and training. They were obtained randomly from the set using `randi`.

The training options need explanation. This is a subset of the parameter pairs available for `trainingOptions`. The first input to the function, `'sgdm'`, specifies the training method. There are three to choose from:

1. `'sgdm'` —Stochastic gradient descent with momentum

Figure 3.6: A subset of the ellipses used in the training and testing.



2. `'adam'` —Adaptive moment estimation (ADAM)

3. `'rmsprop'` —Root mean square propagation (RMSProp)

The `'InitialLearnRate'` is the initial speed of learning. Higher learn rates mean faster learning, but the training may get stuck in a suboptimal point. The default rate for `sgdm` is 0.01. `'MaxEpochs'` is the maximum number of epochs to be used in the training. In each epoch, the training sees the entire training set, in batches of `MiniBatchSize`. The number of iterations in each epoch is therefore determined by the amount of data in the set and the `MiniBatchSize`. We are using a smaller data set so we reduce the `MiniBatchSize` from the default of 128 to 16, which will give us 10 iterations per epoch. `'Shuffle'` tells the training how often to shuffle the training data. If you don't shuffle, the data will always be used in the same order. Shuffling should improve the accuracy of the trained neural network. `'ValidationFrequency'` is how often, in number of iterations, `'ValidationData'` is used to test the training. This validation data will be the data we reserved for testing when using `splitEachLabel`. The default frequency is every 30 iterations. We can use a validation frequency for our small problem of one, two, or five iterations. `'Verbose'` means print out status information to the command window. `'Plots'` only has the option `'training-progress'` (besides `'none'`). This is the plot you see in this chapter.

"Padding" in the `convolution2dLayer` means that the output size is `ceil(inputSize/stride)`, where `inputSize` is the height and width of the input.

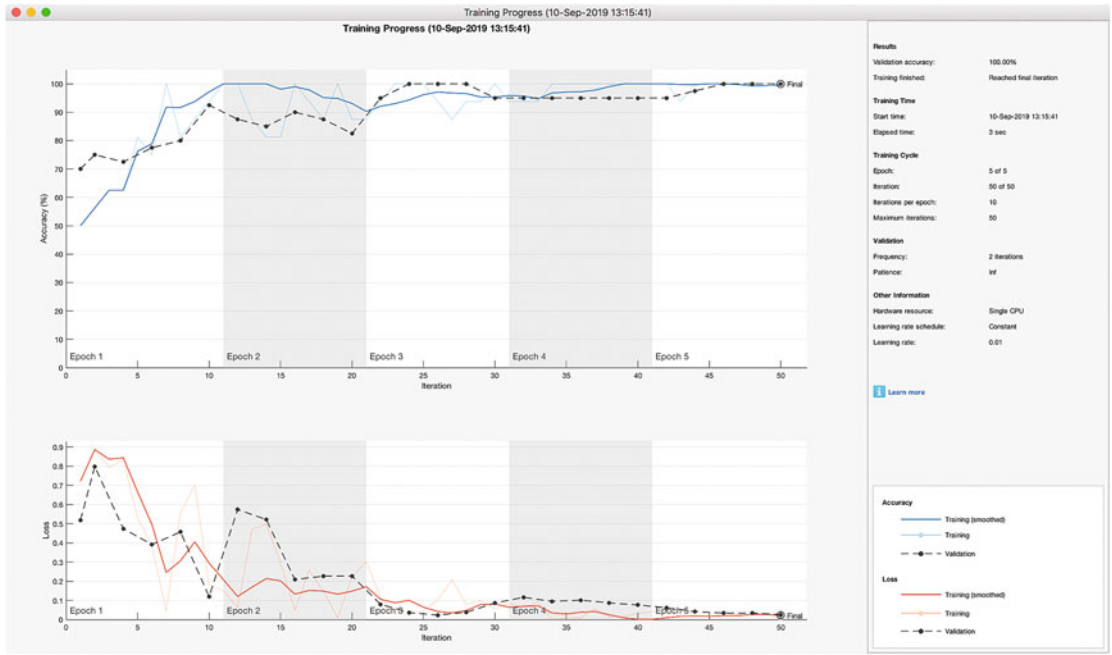
The training window runs in real time with the training process. The window is shown in Figure 3.7. Our network starts with a 50% accuracy since we only have two classes, circles and ellipses. Our accuracy approaches 100% in just five epochs, indicating that our classes of images are readily distinguishable. The loss plot shows how well we are doing. The lower the loss, the better the neural net. The loss plot approaches zero as the accuracy approaches 100%. In this case the validation data loss and the training data loss are about the same. This indicates good fitting of the neural net with the data. If the validation data loss is greater than the training data loss, the neural net is overfitting the data. Overfitting happens when you have an overly complex neural network. You can fit the training data, but it may not perform very well with new data, such as the validation data. For example, if you have a system which really is linear, and you fit it to a cubic equation, it might fit the data well but doesn't really model the real system. If the loss is greater than the validation data loss, your neural net is underfitting. Underfitting happens when your neural net is too simple. The goal is to make both zero.

Finally, we test the net. Remember that this is a classification problem. An image is either an ellipse or a circle. We therefore use `classify` to implement the network. `predLabels` is the output of the net, that is, the predicted labels for the test data. This is compared to the truth labels from the datastore to compute an accuracy.

EllipsesNeuralNet.m

```
1
2   %% Test the neural net
3  predLabels  = classify(net,imdsTest);
```

Figure 3.7: The training window with a learn rate of 0.01. The top plot is the accuracy expressed as a percentage.

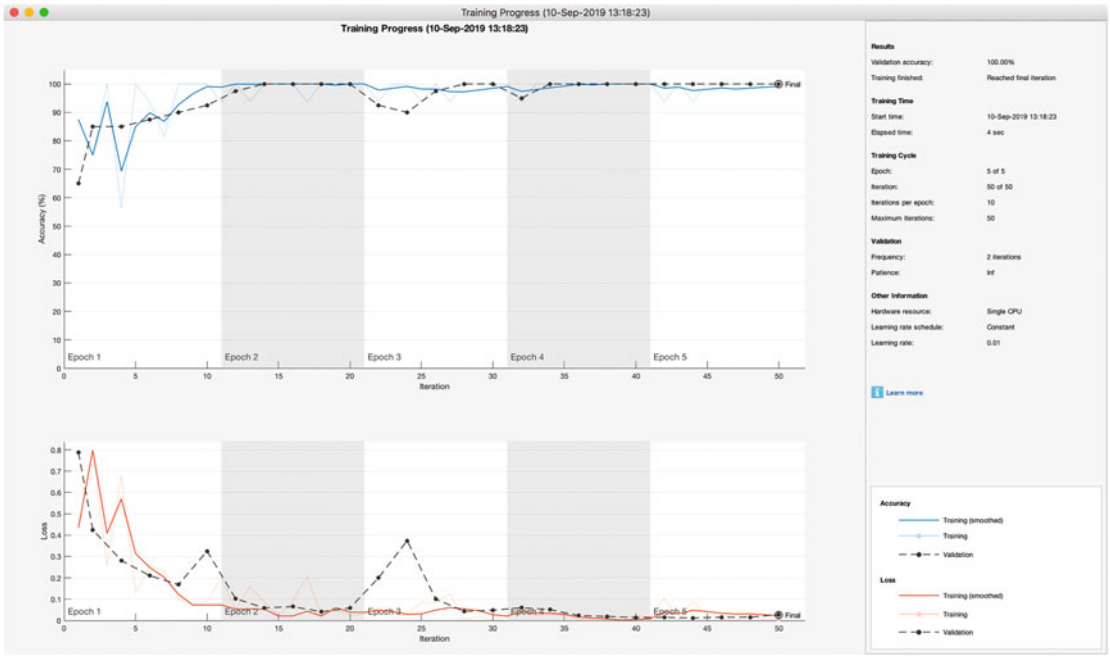



```
4  testLabels  = imdsTest.Labels;
5
6  accuracy = sum(predLabels == testLabels)/numel(testLabels);
```

The output of the testing is shown in the following. The accuracy of this run was 97.50%. On some runs, the net reaches 100%.

```
>> EllipsesNeuralNet

ans =

  Figure (1: Ellipses) with properties:

      Number: 1
        Name: 'Ellipses'
       Color: [0.9400 0.9400 0.9400]
    Position: [560 528 560 420]
       Units: 'pixels'

  Show all properties

Accuracy is    97.50%
```

Figure 3.8: The training window with a learn rate of 0.01 and a leaky reluLayer.



We can try different activation functions. `EllipsesNeuralNetLeaky` shows a leaky reluLayer. We replaced `reluLayer` with `leakyReluLayer`. The output is similar, but in this case, learning was achieved even faster than before. See Figure 3.8 for a training run.

EllipsesNeuralNetLeaky.m

```
% This gives the structure of the convolutional neural net
layers = [
    imageInputLayer(size(img))

    convolution2dLayer(3,8,'Padding','same')
    batchNormalizationLayer
    leakyReluLayer

    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,16,'Padding','same')
    batchNormalizationLayer
    leakyReluLayer

    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,32,'Padding','same')
    batchNormalizationLayer
    leakyReluLayer
```

```
20
21      fullyConnectedLayer(2)
22      softmaxLayer
23      classificationLayer
24          ];
```

The output with the leaky layer is shown as follows.

```
>> EllipsesNeuralNetLeaky

ans =

  Figure (1: Ellipses) with properties:

      Number: 1
        Name: 'Ellipses'
       Color: [0.9400 0.9400 0.9400]
    Position: [560 528 560 420]
       Units: 'pixels'

  Show all properties

Accuracy is    84.25%
```

We can try fewer layers. `EllipsesNeuralNetOneLayer` has only one set of layers.

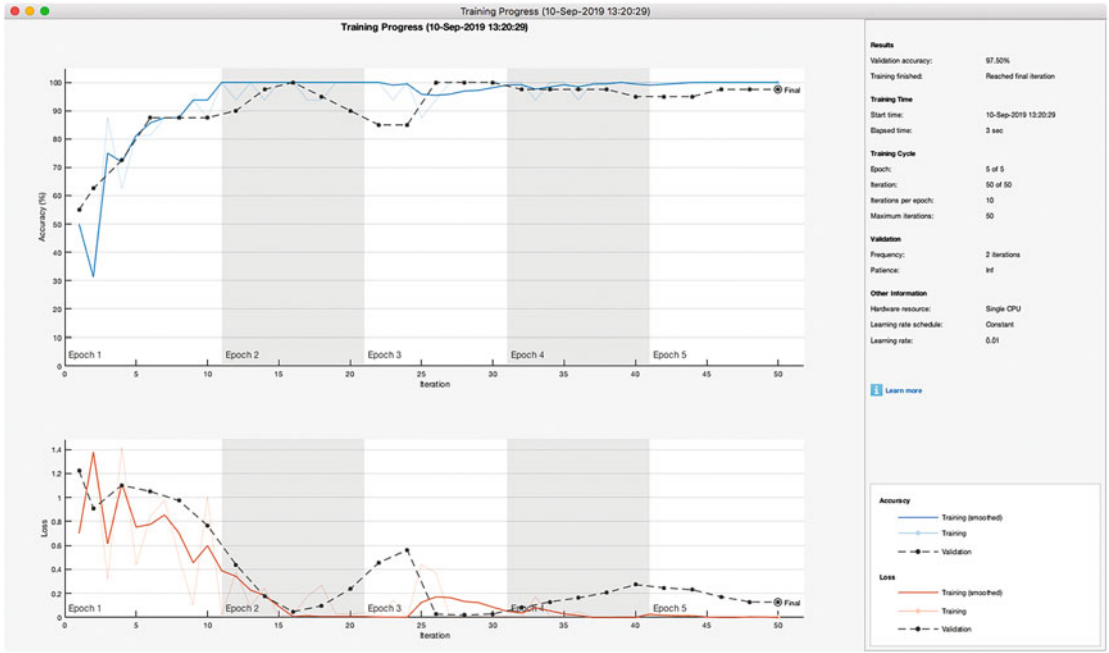
EllipsesNeuralNetOneLayer.m

```
1  %% Define the layers for the net
2  % This gives the structure of the convolutional neural net
3  layers = [
4      imageInputLayer(size(img))
5
6      convolution2dLayer(3,8,'Padding','same')
7      batchNormalizationLayer
8      reluLayer
9
10     fullyConnectedLayer(2)
11     softmaxLayer
12     classificationLayer
13         ];
14
15 analyzeNetwork(layers)
```

The results shown in Figure 3.9 with only one set of layers is still pretty good. This shows that you need to try different options with your net architecture as well. With this size of a problem, multiple layers are not buying very much.

```
>> EllipsesNeuralNetOneLayer
ans =
  Figure (2: Ellipses) with properties:
```

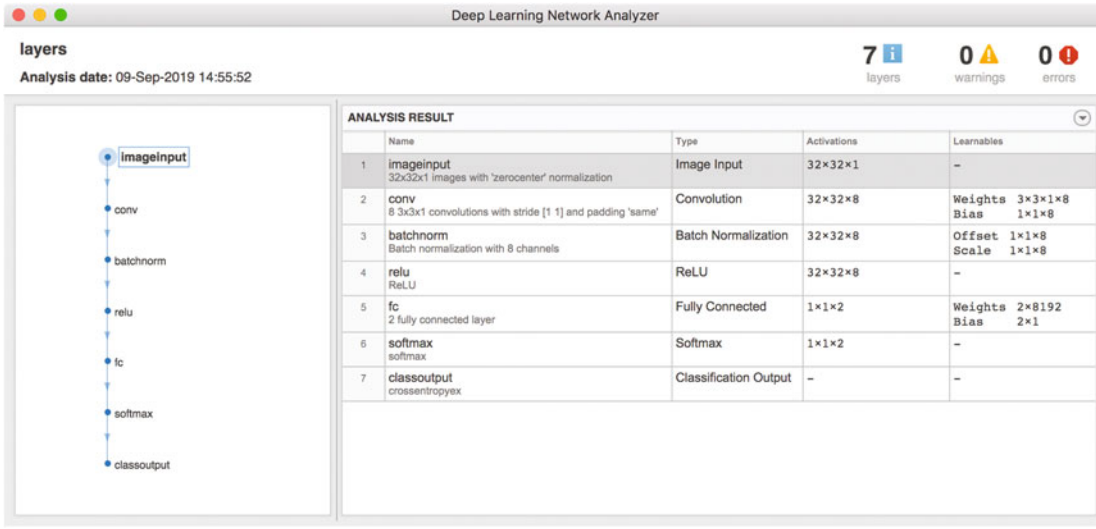
Figure 3.9: The training window for a net with one set of layers.




```
      Number: 2
        Name: 'Ellipses'
       Color: [0.9400 0.9400 0.9400]
    Position: [560 528 560 420]
       Units: 'pixels'

  Show all properties
Accuracy is    87.25%
```

The one-set network is short enough that the whole thing can be visualized inside the window of `analyzeNetwork`, as in Figure 3.10. This function will check your layer architecture before you start training and alert you to any errors. The size of the activations and ‘‘Learnables’’ is displayed explicitly.

Figure 3.10: The analyze window for the one-set convolutional network.



That concludes this chapter. We both generated our own image data and trained a neural net to classify features in our images! In this example, we were able to achieve 100% accuracy, but not after some debugging was required with creating and naming the images. It is critical to carefully examine your training and test data to ensure it contains the features you wish to identify. You should be prepared to experiment with your layers and training parameters as you develop nets for different problems.

CHAPTER 4

Classifying Movies

4.1 Introduction

Netflix, Hulu, and Amazon Prime all attempt to help you pick movies. In this chapter, we will create a database of movies, with fictional ratings. We will then create a set of viewers. We will then try to predict if a viewer would choose to watch a particular movie. We will use Deep Learning with MATLAB's pattern recognition network, `patternnet`. You will see that we can achieve accuracies of up to 100% over our small set of movies. Guessing what a customer would like to buy is something that all manufacturers and retailers want to do as it lets them focus their efforts on products that are of the greatest interest to their customers. As we show in this chapter, deep learning can be a valuable tool.

4.2 Generating a Movie Database

4.2.1 Problem

We first need to generate a database of movies.

4.2.2 Solution

Write a MATLAB function, `CreateMovieDatabase.m`, to create a database of movies. The movies will have fields for genre, reviewer ratings (like IMDb), and the viewer ratings.

4.2.3 How It Works

We first need to come up with a method for characterizing movies. Table 4.1 gives our system. MPAA stands for Motion Picture Association of America. It is an organization that rates the movies. Other systems are possible, but this will be sufficient to test out our deep learning system. Three are strings and two are numbers. One number, length, is a continuum, while rating has discrete values. The second number, quality, is based on the ‘‘stars” in the rating. Some movie databases, like IMDb, have fractional values because they average over all their users. We created our own MPAA ratings and genres based on our opinions. The real MPAA ratings may be different.

M. Paluszek and S. Thomas, *Practical MATLAB Deep Learning*,
https://doi.org/10.1007/978-1-4842-5124-9_4

Table 4.1: The movie database will have five characteristics.

Characteristic	Value
Name	String
Genre	Type of movie (Animated, Comedy, Dance, Drama, Fantasy, Romance, SciFi, War) in a string
Rating	Average viewer rating string, one to four stars, such as from IMDb
Quality	Number of stars
Length	Minutes duration
MPAA rating	MPAA rating (PG, PG-13, R) in a string

Length can be any duration. We'll use `randn` to generate the lengths around a mean of 1.8 hours and a standard deviation of 0.15 hours. Length is a floating point number. Stars are one to five and must be integers.

We created an Excel file with the names of 100 real movies, which is included with the book's software. We assigned genres and MPAA ratings (PG, R, etc.) to them. These were made up by the authors. The length and rating were left blank. We then saved the Excel file as tab-delimited text and search for tabs in each line. (There are other ways to import data from Excel and text files in MATLAB, this is just one example.) We then assign the data to the fields. The function will check to see if the maximum length or rating is zero, which it is for all the movies in this case, and then create random values. You can create a spreadsheet with rating values as an extension of this recipe! We use `str2double` since it is faster than `str2num` when you know that the value is a single number. `fgetl` reads in one line and ignores the end of line characters.

You'll notice that we check for `NaN` in the length and rating fields since

```
>> str2double('')

ans =

   NaN
```

CreateMovieDatabase.m

```
function d = CreateMovieDatabase( file )

if( nargin < 1 )
  Demo
  return
end

f = fopen(file,'r');

d.name    = {};
d.rating  = [];
d.length  = [];
```

```
d.genre   = {};
d.mPAA    = {};
t         = sprintf('\t');  % a tab character
k         = 0;

while(~feof(f))
  k             = k + 1;
  q             = fgetl(f);      % one line of the file
  j             = strfind(q,t);  % find the tabs in the line
  d.name{k}     = q(1:j(1)-1);   % the name is the first token
  d.rating(1,k) = str2double(q(j(1)+1:j(2)-1));
  d.genre{k}    = q(j(2)+1:j(3)-1);
        d.length(1,k)   = str2double(q(j(3)+1:j(4)-1));
  d.mPAA{k}     = q(j(4)+1:end);
end % end of the file

if( max(d.rating) == 0 || isnan(d.rating(1)) )
  d.rating = randi(5,1,k);
end

if( max(d.length) == 0 || isnan(d.length(1)))
  d.length = 1.8 + 0.15*randn(1,k);
end

fclose(f);
```

Running the function demo, shown in the following, creates a database of movies in a data structure.

```
function Demo

file = 'Movies.txt';
d    = CreateMovieDatabase( file )
```

The output is the following.

```
>> CreateMovieDatabase

d =

  struct with fields:

      name: {1 x 100 cell}
    rating: [1 x 100 double]
    length: [1 x 100 double]
     genre: {1 x 100 cell}
      mPAA: {1 x 100 cell}
```

Here are the first few movies:

```
>> d.name'
ans =
  100x1 cell array
    {'2001: A Space Odyssey'            }
    {'A Star is Born'                   }
    {'Alien'                            }
    {'Aliens'                           }
    {'Amadeus'                          }
    {'Apocalypse Now'                   }
    {'Apollo 13'                        }
    {'Back to the Future'               }
```

4.3 Generating a Movie Watcher Database

4.3.1 Problem

We next need to generate a database of movie watchers for training and testing.

4.3.2 Solution

Write a MATLAB function, `CreateMovieViewers.m`, to create a series of watchers. We will use a probability model to select which of the movies each viewer has watched based on the movie's genre, length, and ratings.

4.3.3 How It Works

Each watcher will have seen a fraction of the 100 movies in our database. This will be a random integer between 20 and 60. Each movie watcher will have a probability for each characteristic: the probability that they would watch a movie rated 1 or 5 stars, the probability that they would watch a movie in a given genre, and so on. (Some viewers enjoy watching so-called ''turkeys''!) We will combine the probabilities to determine the movies the viewer has watched. For `mPAA`, `genre`, and `rating`, the probabilities will be discrete. For the length, it will be a continuous distribution. You could argue that a watcher would always want the highest rated movie, but remember this rating is based on an aggregate of other people's opinions so may not directly map onto the particular viewer. The only output of this function is a list of movie numbers for each user. The list is in a cell array.

We start by creating cell arrays of the categories. We then loop through the viewers and compute probabilities for each movie characteristic. We then loop through the movies and compute the combined probabilities. This results in a list of movies watched by each viewer.

CreateMovieViewer.m

```
1  function [mvr,pWatched] = CreateMovieViewers( nViewers, d )
2
3  if( nargin < 1 )
4    Demo
```

```
  return
end

mvr   = cell(1,nViewers);
nMov  = length(d.name);
genre = { 'Animated', 'Comedy', 'Dance', 'Drama', 'Fantasy', 'Romance'
    ,...
          'SciFi', 'War', 'Horror', 'Music', 'Crime'};
mPAA  = {'PG-13','R','PG'};

% Loop through viewers. The inner loop is movies.
for j = 1:nViewers
  % Probability of watching each MPAA
  rMPAA = rand(1,length(mPAA));
  rMPAA = rMPAA/sum(rMPAA);

  % Probability of watching each Rating (1 to 5 stars)
  r = rand(1,5);
  r = r/sum(r);

  % Probability of watching a given Length
  mu    = 1.5 + 0.5*rand; % preferred movie length, between 1.5 and 2 hrs
  sigma = 0.5*rand;       % variance, up to 1/2 hour

  % Probability of watching by Genre
  rGenre = rand(1,length(genre));
  rGenre = rGenre/sum(rGenre);

  % Compute the likelihood the viewer watched each movie
  pWatched = zeros(1,nMov);
  for k = 1:nMov
    pRating   = r(d.rating(k));           % probability for this rating
    i         = strcmp(d.mPAA{k},mPAA);   % logical array with one match
    pMPAA     = rMPAA(i);                 % probability for this MPAA
    i         = strcmp(d.genre{k},genre); % logical array
    pGenre      = rGenre(i);                % probability for this genre
    pLength   = Gaussian(d.length(k),sigma,mu);  % probability for this
        length
    pWatched(k) = 1 - (1-pRating)*(1-pMPAA)*(1-pGenre)*(1-pLength);
  end

  % Sort the movies and pick the most likely to have been watched
  nInterval = floor( [0.2 0.6]*nMov );
  nMovies = randi(nInterval);
  [~,i]   = sort(pWatched);
  mvr{j}  = i(1:nMovies);
end
```

This code computes the Gaussian or normal probability. The inputs include a standard deviation `sigma` and a mean `mu`.

```
%% CreateMovieViewers>Gaussian
% The probability is 1 when x==mu and declines for shorter or longer
    movies
function p = Gaussian(x,sigma,mu)

p = exp(-(x-mu)^2/(2*sigma^2));
```

The built-in function demo follows the Gaussian function. It is run automatically if the function is called with no inputs.

```
%% CreateMovieViewers>Demo
function Demo

s = load('Movies.mat');
```

The output from the demo is shown next. This shows how many of the movies in the database each viewer has watched; the most is 57 movies and the fewest is 26.

```
>> CreateMovieViewers

mvr =

  1x4 cell array

    {1x33 double}    {1x57 double}    {1x51 double}    {1x26 double}
```

4.4 Training and Testing

4.4.1 Problem

We want to test a deep learning algorithm to select new movies for the viewer, based on what the algorithm thinks a viewer would choose to watch.

4.4.2 Solution

Create a viewer database and train a pattern recognition neural net on the viewer's movie selections. This is done in the script `MovieNN`. We will train a neural net for each viewer in the database.

4.4.3 How It Works

First, the movie data is loaded and displayed.

MovieNN.m

```
%% Data
genre  = { 'Animated', 'Comedy', 'Dance', 'Drama', 'Fantasy', ...
           'Romance', 'SciFi', 'War', 'Horror', 'Music', 'Crime'};
mPAA   = {'PG-13','R','PG'};
rating = {'*' '**' '***' '****' '*****'};

%% The movies
s = load('Movies.mat');
NewFigure('Movie Data')
subplot(2,2,1)
histogram(s.d.length)
xlabel('Movie Length')
ylabel('# Movies')
subplot(2,2,2)
histogram(s.d.rating)
xlabel('Stars')
ylabel('# Movies')
subplot(2,1,2)
histogram(categorical(s.d.genre))
ylabel('# Movies')
set(gca,'xticklabelrotation',90)
```

The viewer database is then created from the movie database:

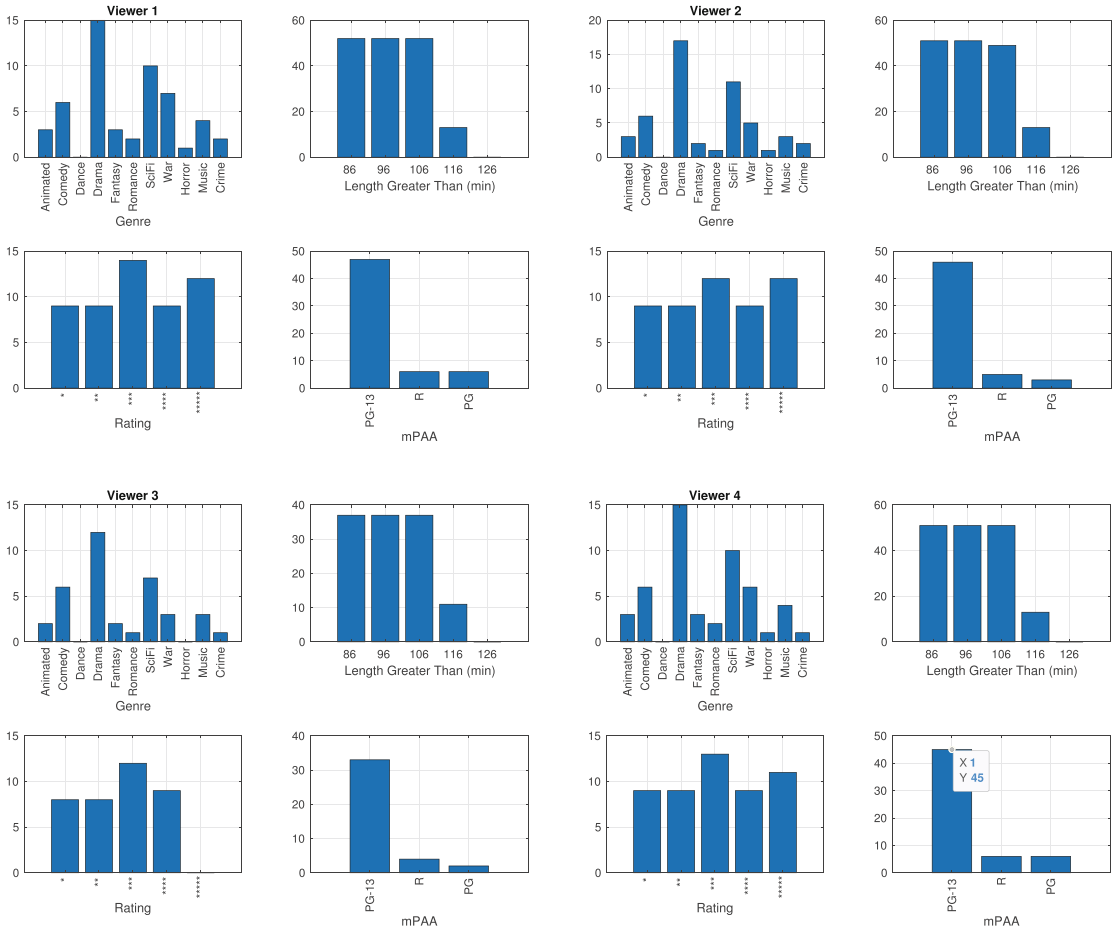
MovieNN.m

```
%% The movie viewers
nViewers  = 4;
mvr       = CreateMovieViewers( nViewers, s.d );
```

The next block displays the characteristics of the movies each viewer has watched. This is shown graphically in Figure 4.1. For the moment, there are only four viewers.

```
% Display the movie viewer's data
lX     = linspace(min(s.d.length),max(s.d.length),5);

for k = 1:nViewers
  NewFigure(sprintf('Viewer %d',k));

  subplot(2,2,1);
  g = zeros(1,11);
  for j = 1:length(mvr{k})
    i     = mvr{k}(j);
    l     = strmatch(s.d.genre{i},genre); %#ok<MATCH2>
    g(l)  = g(l) + 1;
  end
```

Figure 4.1: The watched movie data for four viewers.




```
    bar(1:11,g);
    set(gca,'xticklabel',genre,'xticklabelrotation',90,'xtick',1:11)
    xlabel('Genre')
    title(sprintf('Viewer %d',k))
    grid on

    subplot(2,2,2);
    g = zeros(1,5);
    for j = 1:5
      for i = 1:length(mvr{k})
        if( s.d.length(mvr{k}(i)) > lX(j) )
          g(j)  = g(j) + 1;
        end
      end
    end
    bar(1:5,g);
```

```
    set(gca,'xticklabel',floor(lX*60),'xtick',1:5)
    xlabel('Length Greater Than (min)')
    grid on

    subplot(2,2,3);
    g = zeros(1,5);
    for j = 1:length(mvr{k})
      i     = mvr{k}(j);
      l     = s.d.rating(i);
      g(l)  = g(l) + 1;
    end
    bar(1:5,g);
    set(gca,'xticklabel',rating,'xticklabelrotation',90,'xtick',1:5)
    xlabel('Rating')
    grid on

    subplot(2,2,4);
    g = zeros(1,3);
    for j = 1:length(mvr{k})
      i     = mvr{k}(j);
      l     = strmatch(s.d.mPAA{i},mPAA); %#ok<MATCH2>
      g(l)  = g(l) + 1;
    end
    bar(1:3,g);
    set(gca,'xticklabel',mPAA,'xticklabelrotation',90,'xtick',1:3)
    xlabel('mPAA')
    grid on
end
```

We use bar charts throughout. Notice how we make the x labels strings for the genre and so on. We also rotate them 90 degrees for clarity. The length is the number of movies longer than that number.

This data is based on our viewer model from the previous recipe which is based on joint probabilities. We will train the neural net on a subset of the movies. This is a classification problem. We just want to know if a given movie would be picked or not picked by the viewer.

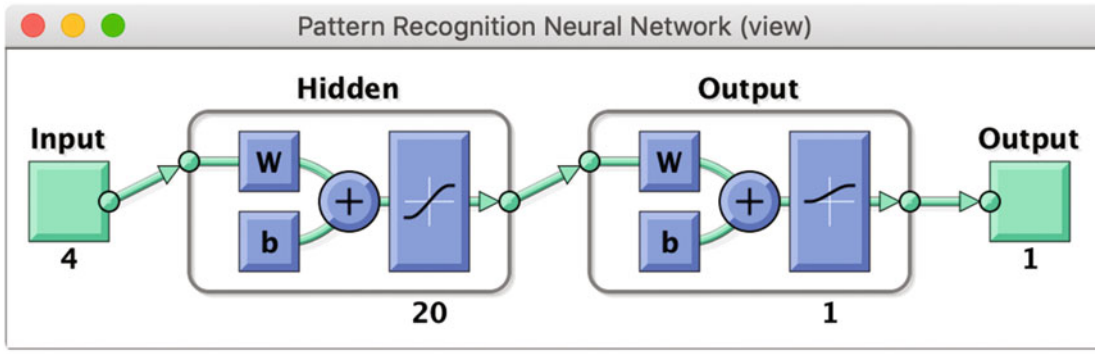
We use `patternnet` to predict the movies watched. This is shown in the next code block. The input to `patternnet` is the sizes of the hidden layers, in this case a single layer of size 40. We convert everything into integers. Note that you need to round the results since `patternnet` does not return integers, despite the label being an integer. `patternnet` has methods `train` and `view`.

```
%% Train and test the neural net for each viewer
for k = 1:nViewers
  % Create the training arrays
  x = zeros(4,100); % the input data
  y = zeros(1,100); % the target - did the viewer watch the movie?

  nMov = length(mvr{k}); % number of watched movies
  for j = 1:nMov
```

```
    i      = mvr{k}(j); % index of the jth movie watched by the kth
        viewer
    x(1,j) = s.d.rating(i);
    x(2,j) = s.d.length(i);
    x(3,j) = strmatch(s.d.mPAA{i},mPAA,'exact'); %#ok<*MATCH3>
    x(4,j) = strmatch(s.d.genre{i},genre,'exact');
    y(1,j) = 1; % movie watched
  end

  i = setdiff(1:100,mvr{k}); % unwatched movies
  for j = 1:length(i)
    x(1,nMov+j) = s.d.rating(i(j));
    x(2,nMov+j) = s.d.length(i(j));
    x(3,nMov+j) = strmatch(s.d.mPAA{i(j)},mPAA,'exact');
    x(4,nMov+j) = strmatch(s.d.genre{i(j)},genre,'exact');
    y(1,nMov+j) = 0; % movie not watched
  end

  % Create the training and testing data
  j       = randperm(100);
  j       = j(1:70);  % train using 70% of the available data
  xTrain  = x(:,j);
  yTrain  = y(1,j);
  j       = setdiff(1:100,j);
        xTest   = x(:,j);
  yTest   = y(1,j);

  net     = patternnet(40); % input a scalar or row of layer sizes
  net     = train(net,xTrain,yTrain);
  view(net);
  yPred = round(net(xTest));

  %% Test the neural net
  accuracy = sum(yPred == yTest)/length(yTest);
  fprintf('Accuracy for viewer %d (%d movies watched) is %8.2f%%\n',...
    k,nMov,accuracy*100)
end
```

The training window is shown in Figure 4.3. When we view the net, MATLAB opens the display in Figure 4.2. Each net has four inputs, for the movie's rating, length, genre, and MPAA classification. The net's single output is the classification for whether the viewer has watched the movie or not. The training window provides access to additional plots of the training and performance data.

Figure 4.2: The patternnet network with four inputs and one output.

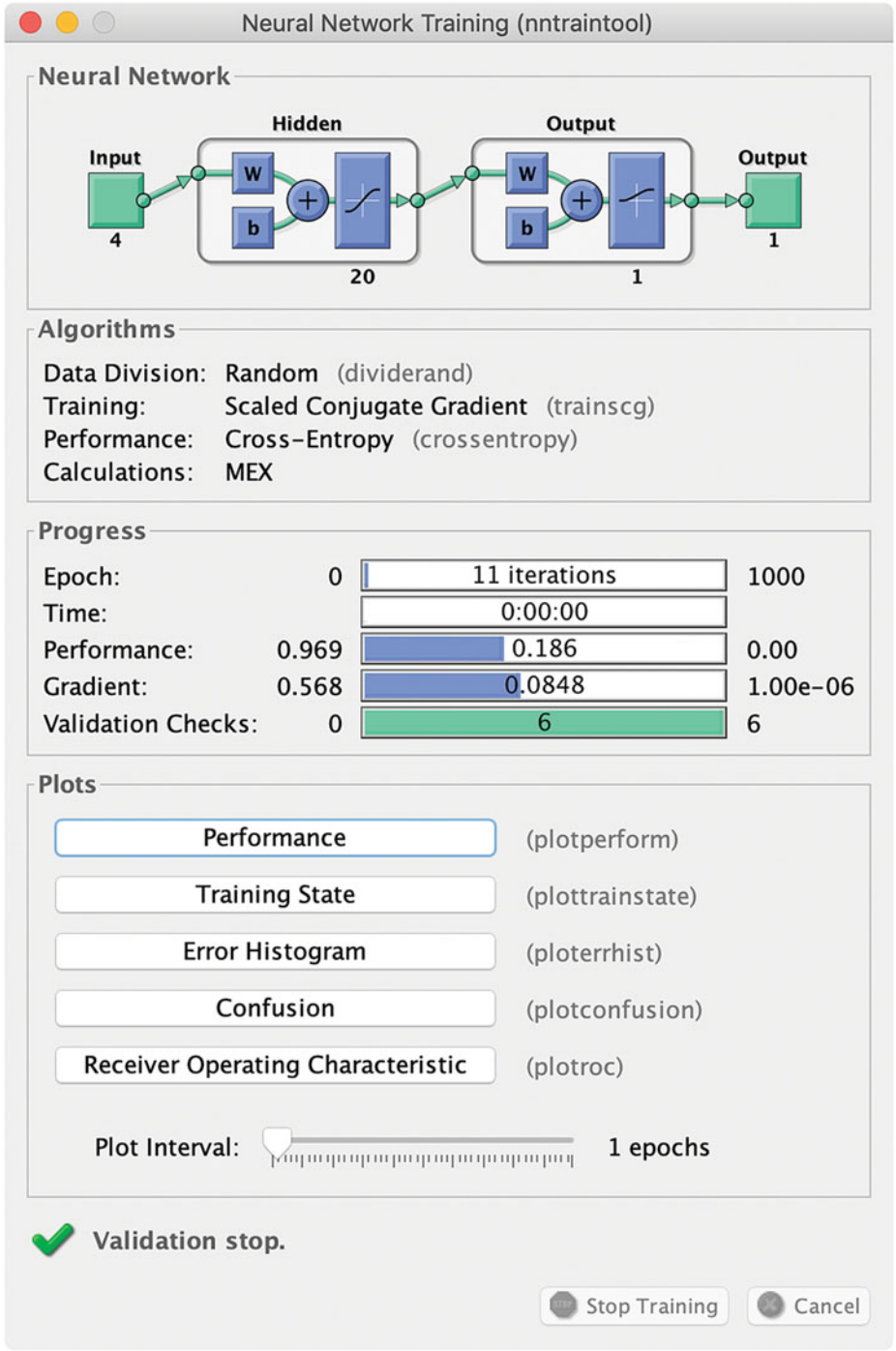


The output of the script is shown later for `patternnet(40)`. The accuracy is the percentage of the movies in the test set (30% of the data available) that the net correctly predicted the viewer to have watched. The accuracy is usually between 65% and 90% for this size hidden layer.

```
>> MovieNN
Accuracy for viewer 1 (58 movies watched) is    83.33%
Accuracy for viewer 2 (51 movies watched) is    76.67%
Accuracy for viewer 3 (23 movies watched) is    83.33%
Accuracy for viewer 4 (54 movies watched) is   100.00%
```

`patternnet(40)` returns good results, but we also tried greater and smaller numbers of layers and multiple hidden layers. For example, with a layer size of just 5, the accuracy ranges from 50 to 70%. With a size of 50, we reached over 90% for all viewers! Granted, this is a small number of movies. The results will vary with each run due to the random nature of the variables in the test. The predictions are probably as good as Netflix! It is important to note that the neural network did not know anything about the viewer model. Nonetheless, it does a good job of predicting movies that the viewer might like. This is one of the advantages of neural nets.

Figure 4.3: Patternnet training window.



CHAPTER 5

Algorithmic Deep Learning

In this chapter, we introduce the Algorithmic Deep Learning Neural Network (ADLNN), a deep learning system that incorporates algorithmic descriptions of the processes as part of the deep learning neural network. The dynamical models provide domain knowledge. These are in the form of differential equations. The outputs of the network are both indications of failures and updates to the parameters of the models. Training can be done using simulations, prior to operations, or through operator interaction during operations.

The system is shown in Figure 5.1. This is based on work from the books by Paluszek and Thomas [30, 29]. These books show the relationships between machine learning, Adaptive Control, and Estimation. This model can be encapsulated in a set of differential equations. We will limit ourselves to sensor failures in this example.

The output indicates what kind of failures have occurred. It indicates that either one or both of the sensors have failed.

Figure 5.2 shows an air turbine.[1] This air turbine has a constant pressure air supply. The pressurized air causes the turbine to spin. It is a way to produce rotary motion for a drill or other purpose.

We can control the valve from the air supply, the pressure regulator, to control the speed of the turbine. The air flows past the turbine blades causing it to turn. The control needs to adjust the air pressure to handle variations in the load. The load is the resistance to turning. For example, a drill might hit a harder material while in use. We measure the air pressure p downstream from the valve, and we also measure the rotational speed of the turbine ω with a tachometer.

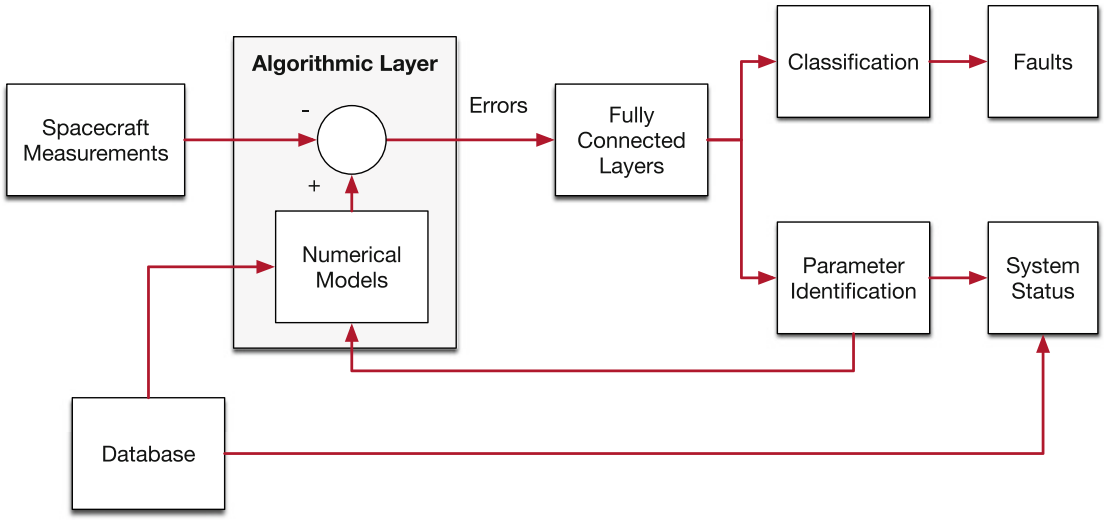
The dynamical model for the air turbine is

$$\begin{bmatrix} \dot{p} \\ \dot{\omega} \end{bmatrix} = \begin{bmatrix} -\frac{1}{\tau_p} & 0 \\ \frac{K_t}{\tau_t} & -\frac{1}{\tau_t} \end{bmatrix} \begin{bmatrix} p \\ \omega \end{bmatrix} + \begin{bmatrix} \frac{K_p}{\tau_p} \\ 0 \end{bmatrix} u \tag{5.1}$$

[1] PhD thesis of Jere Schenck Meserole, ''Detection Filters for Fault-Tolerant Control of Turbofan Engines,'' Massachusetts Institute of Technology, Department of Aeronautics and Astronautics, 1981.

M. Paluszek and S. Thomas, *Practical MATLAB Deep Learning*,
https://doi.org/10.1007/978-1-4842-5124-9_5

Figure 5.1: Algorithmic Deep Learning Neural Network (ADLNN). The network uses numerical models as a filtering layer. The numerical models are a set of differential equations configured as a detection filter.



Algorithmic Deep Learning Neural Network

This is a state space system

$$\dot{x} = ax + bu \tag{5.2}$$

where

$$a = \begin{bmatrix} -\frac{1}{\tau_p} & 0 \\ \frac{K_t}{\tau_t} & -\frac{1}{\tau_t} \end{bmatrix} \tag{5.3}$$

$$b = \begin{bmatrix} \frac{K_p}{\tau_p} \\ 0 \end{bmatrix} \tag{5.4}$$

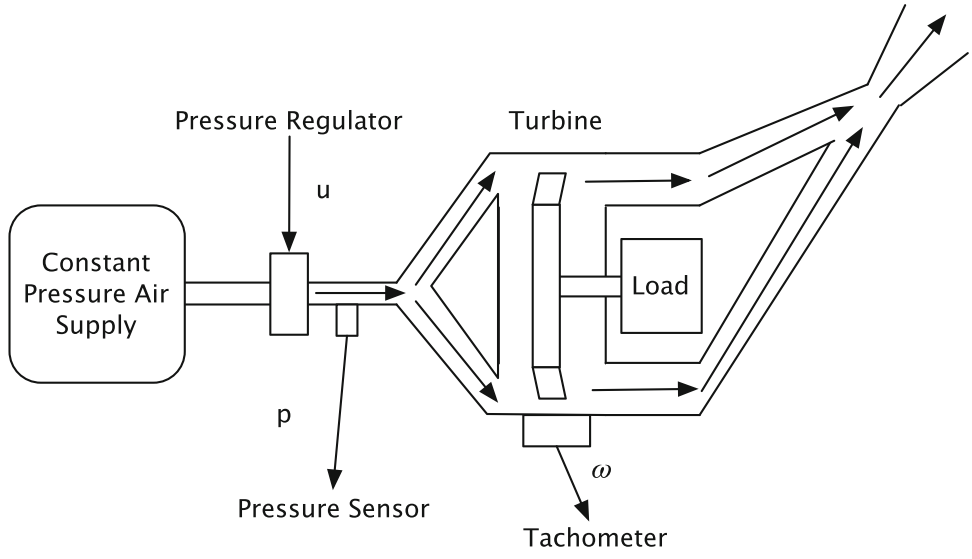
The state vector is

$$\begin{bmatrix} p \\ \omega \end{bmatrix} \tag{5.5}$$

The pressure downstream from the regulator is equal to $K_p u$ when the system is in equilibrium. τ_p is the regulator time constant, and τ_t is the turbine time constant. The turbine speed is $K_t p$ when the system is in equilibrium. The tachometer measures ω and the pressure sensor measures p. The load is folded into the time constant for the turbine.

The code for the right-hand side of the dynamical equations is shown in the following. Only one line of code is the right-hand side. The rest returns the default data structure. The simplicity of the model is due to its being a state space model. The number of states could be large, yet the code would not change. As you can see, the dynamical equations are just one line of code.

Figure 5.2: Air turbine. The arrows show the airflow. The air flows through the turbine blade tips causing it to turn.



RHSAirTurbine.m

```
if( nargin < 1 )
  kP   = 1;
  kT   = 2;
  tauP = 10;
  tauT = 40;
  c    = eye(2);
  b    = [kP/tauP;0];
  a    = [-1/tauP 0; kT/tauT -1/tauT];

  xDot = struct('a',a,'b',b,'c',c,'u',0);
  if( nargout == 0)
    disp('RHSAirTurbine struct:');
  end
  return
end

% Derivative
xDot = d.a*x + d.b*d.u;
```

The simulation, `AirTurbineSim.m`, is shown in the following. The control is a constant, also known as a step input. `TimeLabel` converts the time vector into units (minutes, hours, etc.) that are easier to read. It also returns a label for the time units that you can use in the plots.

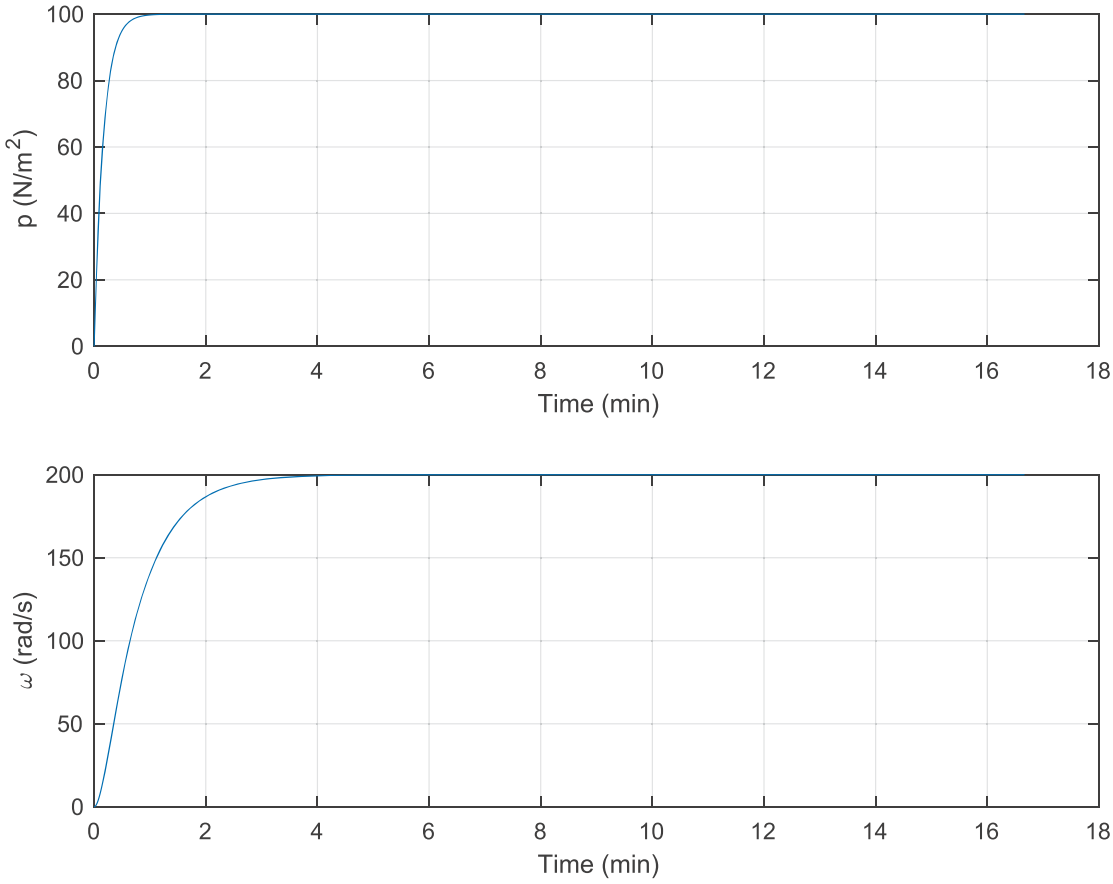
AirTurbineSim.m

```
%% Initialization
tEnd  = 1000; % sec

% State space system
d     = RHSAirTurbine;

% This is the regulator input.
d.u   = 100;

dT    = 0.02; % sec
n     = ceil(tEnd/dT);

% Initial state
x     = [0;0];

%% Run the simulation

% Plotting array
xP    = zeros(2,n);
t     = (0:n-1)*dT;

for k = 1:n
  xP(:,k) = x;
  x       = RungeKutta( @RHSAirTurbine, t(k), x, dT, d );
end

%% Plot the states and residuals
[t,tL] = TimeLabel(t);
yL     = {'p (N/m^2)' '\omega (rad/s)' };
tTL    = 'Air Turbine Simulation';
PlotSet( t, xP,'x label',tL,'y label',yL,'figure title',tTL)
```

The response to a step input for u is shown in Figure 5.3. The pressure settles faster than the turbine angular velocity. This is due to the turbine time constant and the lag in the pressure change.

Now that we understand better how an air turbine works, we can build the filter to detect its sensor failures. It is always a good idea to understand your dynamical system. When building an algorithmic filter or estimator, this is a necessity. For a neural net, it is not necessary just to get something working, but really helps when interpreting the neural net performance.

Figure 5.3: Air turbine response to a step pressure regulator input.



5.1 Building a Detection Filter

5.1.1 Problem

We want to build a system to detect failures in our air turbine using the linear model developed in the previous recipe.

5.1.2 Solution

We will build a detection filter that detects pressure regulator failures and tachometer failures. Our plant model (continuous a, b, and c state space matrices) will be an input to the filter building function.

5.1.3 How It Works

The detection filter is an estimator with a specific gain matrix that multiplies the residuals.

$$\begin{bmatrix} \dot{\hat{p}} \\ \dot{\hat{\omega}} \end{bmatrix} = \begin{bmatrix} -\frac{1}{\tau_p} & 0 \\ \frac{K_t}{\tau_t} & -\frac{1}{\tau_t} \end{bmatrix} \begin{bmatrix} \hat{p} \\ \hat{\omega} \end{bmatrix} + \begin{bmatrix} \frac{K_p}{\tau_p} \\ 0 \end{bmatrix} u + \begin{bmatrix} d_{11} & d_{12} \\ d_{21} & d_{22} \end{bmatrix} \begin{bmatrix} p - \hat{p} \\ \omega - \hat{\omega} \end{bmatrix} \tag{5.6}$$

where $\hat{p}$ is the estimated pressure and $\hat{\omega}$ is the estimated angular rate of the turbine. The D matrix is the matrix of detection filter gains. This matrix multiplies the residuals, the difference between the measured and estimated states, into the detection filter. The residual vector is

$$r = \begin{bmatrix} p - \hat{p} \\ \omega - \hat{\omega} \end{bmatrix} \tag{5.7}$$

The D matrix needs to be selected so that this vector tells us the nature of the failure. The gains should be selected so that

1. The filter is stable.
2. If the pressure regulator fails, the first residual $p - \hat{p}$ is nonzero, but the second remains zero.
3. If the turbine fails, the second residual $\omega - \hat{\omega}$ is nonzero, but the first remains zero.

The gain matrix is

$$D = a + \begin{bmatrix} \frac{1}{\tau_1} & 0 \\ 0 & \frac{1}{\tau_2} \end{bmatrix} \tag{5.8}$$

We can see this by substituting this D into Equation 5.6.

$$\begin{bmatrix} \dot{\hat{p}} \\ \dot{\hat{\omega}} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} + \frac{K_t}{\tau_t} & a_{22} \end{bmatrix} \begin{bmatrix} \hat{p} \\ \hat{\omega} \end{bmatrix} + \begin{bmatrix} \frac{K_p}{\tau_p} \\ 0 \end{bmatrix} u + D \begin{bmatrix} p \\ \omega \end{bmatrix} \tag{5.9}$$

The time constant τ_1 is the pressure residual time constant. The time constant τ_2 is the tachometer residual time constant. In effect, we cancel out the dynamics of the plant and replace them with decoupled detection filter dynamics. These time constants should be shorter than the time constants in the dynamical model so that we detect failures quickly. However, they need to be at least twice as long as the sampling period to prevent numerical instabilities.

We will write a function, `DetectionFilter.m`, with three actions: an initialize case, an update case, and a reset case. `varargin` is used to allow the three cases to have different input lists. The function signature is

```
1  function d = DetectionFilter( action, varargin )
```

It can be called in three ways:

```
>> d = DetectionFilter( 'initialize', d, tau, dT )
>> d = DetectionFilter( 'update', u, y, d )
>> d = DetectionFilter( 'reset', d )
```

The first initializes the function, the second is called in each time step for an update, and the last resets the filter. All data is stored in the data structure `d`.

The function simulates detecting failures of an air turbine. An air turbine has a constant pressure air source that sends air through a duct that drives the turbine blades. The turbine is attached to a load. The air turbine model is linear. Failures are modeled by multiplying the regulator input and tachometer output by a constant. A constant of 0 is a total failure and 1 is perfect operation.

The filter is built and initialized in the following code in `DetectionFilter`. The continuous state space model of the plant, in this case our linear air turbine model, is an input. The selected time constants τ are also an input, and they are added to the plant model as in Equation 5.8. The function discretizes the plant `a` and `b` matrices and the computed detection filter gain matrix `d`.

DetectionFilter.m

```
1    case 'initialize'
2      d   = varargin{1};
3      tau = varargin{2};
4      dT  = varargin{3};
5
6      % Design the detection filter
7      d.d = d.a + diag(1./tau);
8
9      % Discretize both
10     d.d        = CToDZOH( d.d, d.b, dT );
11     [d.a, d.b] = CToDZOH( d.a, d.b, dT );
12
13     % Initialize the state
14     m   = size(d.a,1);
15     d.x = zeros(m,1);
16     d.r = zeros(m,1);
```

The update for the detection filter is in the same function. Note the equations implemented as described in the header.

```
1    case 'update'
2      u   = varargin{1};
3      y   = varargin{2};
4      d   = varargin{3};
5      r   = y - d.c*d.x;
6      d.x = d.a*d.x + d.b*u + d.d*r;
7      d.r = r;
```

Finally, we create a reset action to allow us to reset the residual and state values for the filter in between simulations.

```
  case 'reset'
    d   = varargin{1};
    m   = size(d.a,1);
    d.x = zeros(m,1);
    d.r = zeros(m,1);
```

5.2 Simulating Fault Detection

5.2.1 Problem

We want to simulate a failure in the plant and demonstrate the performance of the failure detection.

5.2.2 Solution

We will build a MATLAB script that designs the detection filter using the function from the previous recipe and then simulates it with a user selectable pressure regulator or tachometer failure. The failure can be total or partial.

5.2.3 How It Works

`DetectionFilterSim` designs a detection filter using `DetectionFilter` from the previous recipe and implements it in a loop. A Runge Kutta numerical integration algorithm propagates the continuous domain in the right-hand side of the air turbine, `RHSAirTurbine`. The detection filter is discrete time.

The script has two scale factors `uF` and `tachF` that multiply the regulator input and the tachometer output to simulate failures. Setting a scale factor to zero is a total failure, and setting it to one indicates that the device is working perfectly. If we fail one, we expect the associated residual to be nonzero and the other to stay at zero.

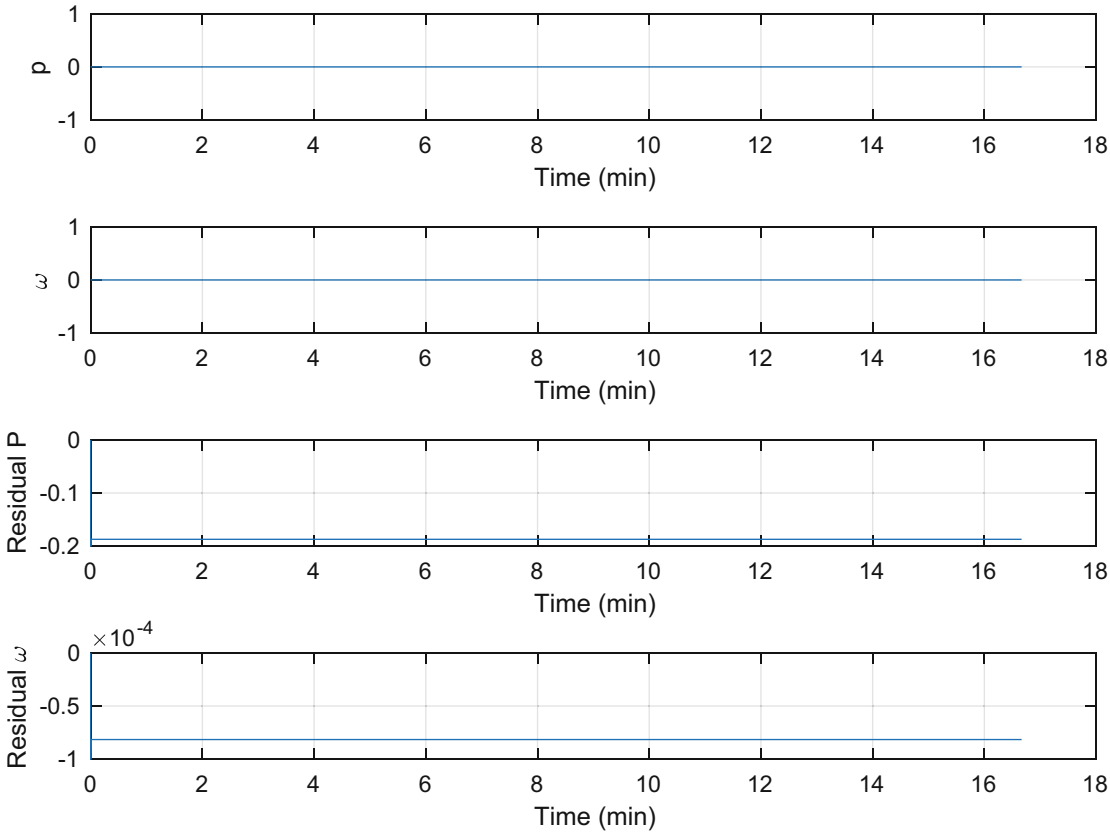
DetectionFilterSim.m

```
%% Script to simulate a detection filter
% Simulates detecting failures of an air turbine. An air turbine has a
    constant
% pressure air source that sends air through a duct that drives the
    turbine
% blades. The turbine is attached to a load.
%
% The air turbine model is linear. Failures are modeled by multiplying
    the
% regulator input and tachometer output by a constant. A constant of 0
    is a
% total failure and 1 is perfect operation.
%% See also:
```

```
10
11 % Time constants for failure detection
12 tau1 = 0.3; % sec
13 tau2 = 0.3; % sec
14
15 % End time
16 tEnd = 1000; % sec
17
18 % State space system
19 d = RHSAirTurbine;
20
21 %% Initialization
22 dT = 0.02; % sec
23 n  = ceil(tEnd/dT);
24
25 % Initial state
26 x = [0;0];
27
28 %% Detection Filter design
29 dF = DetectionFilter('initialize',d,[tau1;tau2],dT);
30
31 %% Run the simulation
32
33 % Control. This is the regulator input.
34 u = 100;
35
36 % Plotting array
37 xP = zeros(4,n);
38 t  = (0:n-1)*dT;
39
40 for k = 1:n
41   % Measurement vector including measurement failure
42   y       = [x(1);tachF*x(2)]; % Sensor failure
43   xP(:,k) = [x;dF.r];
44
45   % Update the detection filter
46   dF      = DetectionFilter('update',u,y,dF);
47
48   % Integrate one step
49   d.u     = uF*u; % Actuator failure
50   x       = RungeKutta( @RHSAirTurbine, t(k), x, dT, d );
51 end
52
53 %% Plot the states and residuals
54 [t,tL] = TimeLabel(t);
55 yL     = {'p' '\omega' 'Residual P' 'Residual \omega' };
56 tTL    = 'Detection Filter Simulation';
57 PlotSet( t, xP,'x label',tL,'y label',yL,'figure title',tTL)
```

In Figure 5.4 the regulator fails and its residual is nonzero. In Figure 5.5 the tachometer fails and its residual is nonzero. The residuals show what has failed clearly. Simple boolean

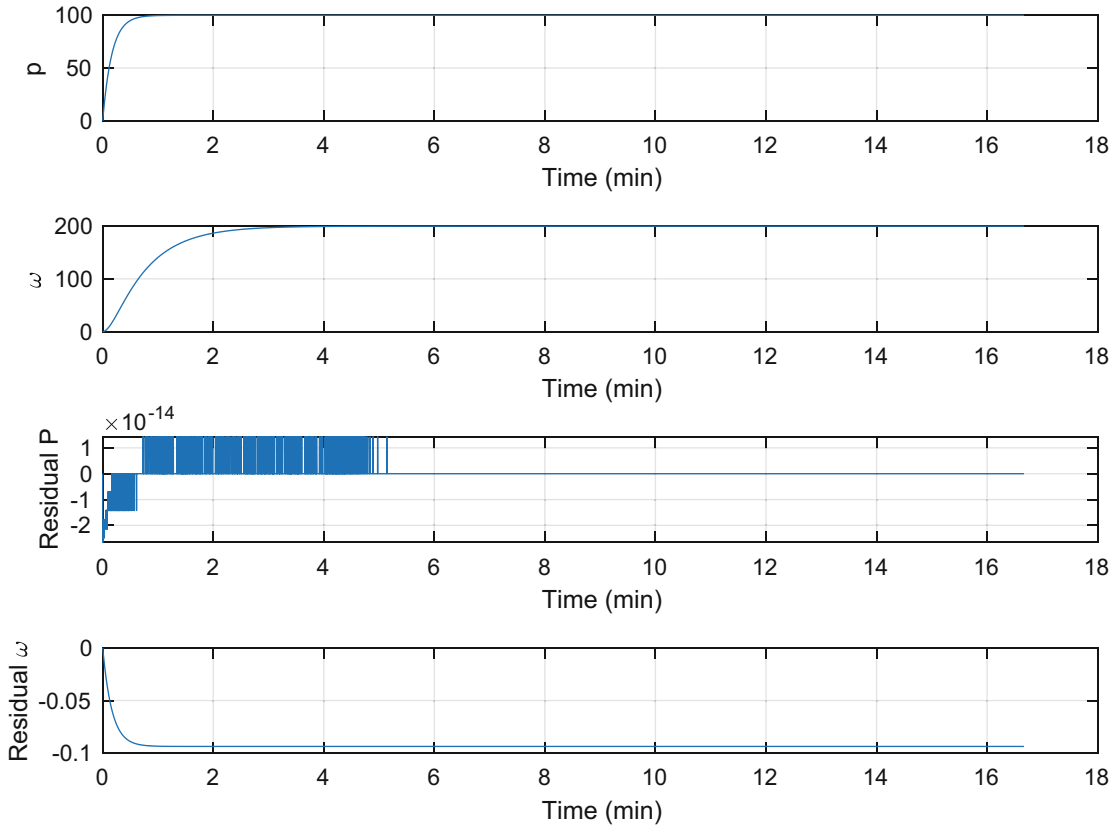
Figure 5.4: Air turbine response to a failed regulator.



logic (i.e., if end statements) are all that is needed. Now, this indicates we don't need machine learning for this problem. The goal here is to show that machine learning can recognize the faults. This will demonstrate that it has the potential to be coupled with more complex systems.

A detection filter is a type of filter. It filters out nonfailures, much like a low-pass filter filters out noise. Adding any type of filter stage to a deep learning system can enhance its performance. Of course, as with any filter, one needs to be careful not to filter out information needed by the learning system. For example, suppose you had an oscillator that was your dynamical. If a low-pass filter cutoff were below the oscillation frequency, you would not be able to learn anything about the oscillation.

Figure 5.5: Air turbine response to a failed tachometer. The residuals immediately reach the outputs indicating the failure because the filter is fast.



5.3 Testing and Training

5.3.1 Problem

We want a neural network to characterize faults. Both tachometer and regulator failures will be characterized.

5.3.2 Solution

We use the same approach as we did for the XOR problem in Chapter 2. The outputs from the detection filter are classified. This could be done by simple boolean logic. The point is to show that a neural net can solve the same problem.

5.3.3 How It Works

We run the simulation to get all possible residuals and combine them in a residual 2 by 4 array. Our outputs are strings for the four cases. You can use strings instead of numbers as the classifier labels, which is better than using integers and then converting them to strings. This just makes the code cleaner and easier to understand. Someone else working on your code is less likely to misinterpret the outputs. We use `feedforwardnet` to implement the neural net. It has two layers, two inputs, and one output. The output is the status of the system. We first train the system with 600 randomly selected test cases. We then simulate the network.

The net is a `feedforwardnet` with two layers. There is one output, the failure, and two inputs from the detection filter. We measure the residuals that we expect to see for each possible failure case. There are four, ‘‘none,’’ ‘‘both,’’ ‘‘tach,’’ and ‘‘regulator.’’ The training pairs are a random selection from the four possible sets using `randi`.

DetectionFilterNN.m

```
% Train the neural net
% Cases
% 2 layers
% 2 inputs
% 1 output

net       = feedforwardnet(2);

%           [none both       tach         regulator]
residual = [0     0.18693851  0          -0.18693851;...
             0    -0.00008143 -0.09353033 -0.00008143];

% labels is a strings array
label     = ["none" "both"  "tach" "regulator"];

% How many sets of inputs
n   = 600;

% This determines the number of inputs and outputs
x   = zeros(2,n);
y   = zeros(1,n);

% Create training pairs
for k = 1:n
  j       = randi([1,4]);
  x(:,k)  = residual(:,j);
  y(k)    = label(j);
end

net       = configure(net, x, y);
net.name  = 'DetectionFilter';
```

```
net       = train(net,x,y);
c         = sim(net,residual);

fprintf('\nRegulator  Tachometer    Failed\n');
for k = 1:4
  fprintf('%9.2e %9.2e      %s\n',residual(1,k),residual(2,k),label(k));
end

% This only works for feedforwardnet(2);
fprintf('\nHidden layer biases %6.3f %6.3f\n',net.b{1});
fprintf('Output layer bias   %6.3f\n',net.b{2});
fprintf('Input layer weights  %6.2f %6.2f\n',net.IW{1}(1,:));
fprintf('                     %6.2f %6.2f\n',net.IW{1}(2,:));
fprintf('Output layer weights %6.2f %6.2f\n',net.LW{2,1}(1,:));
```

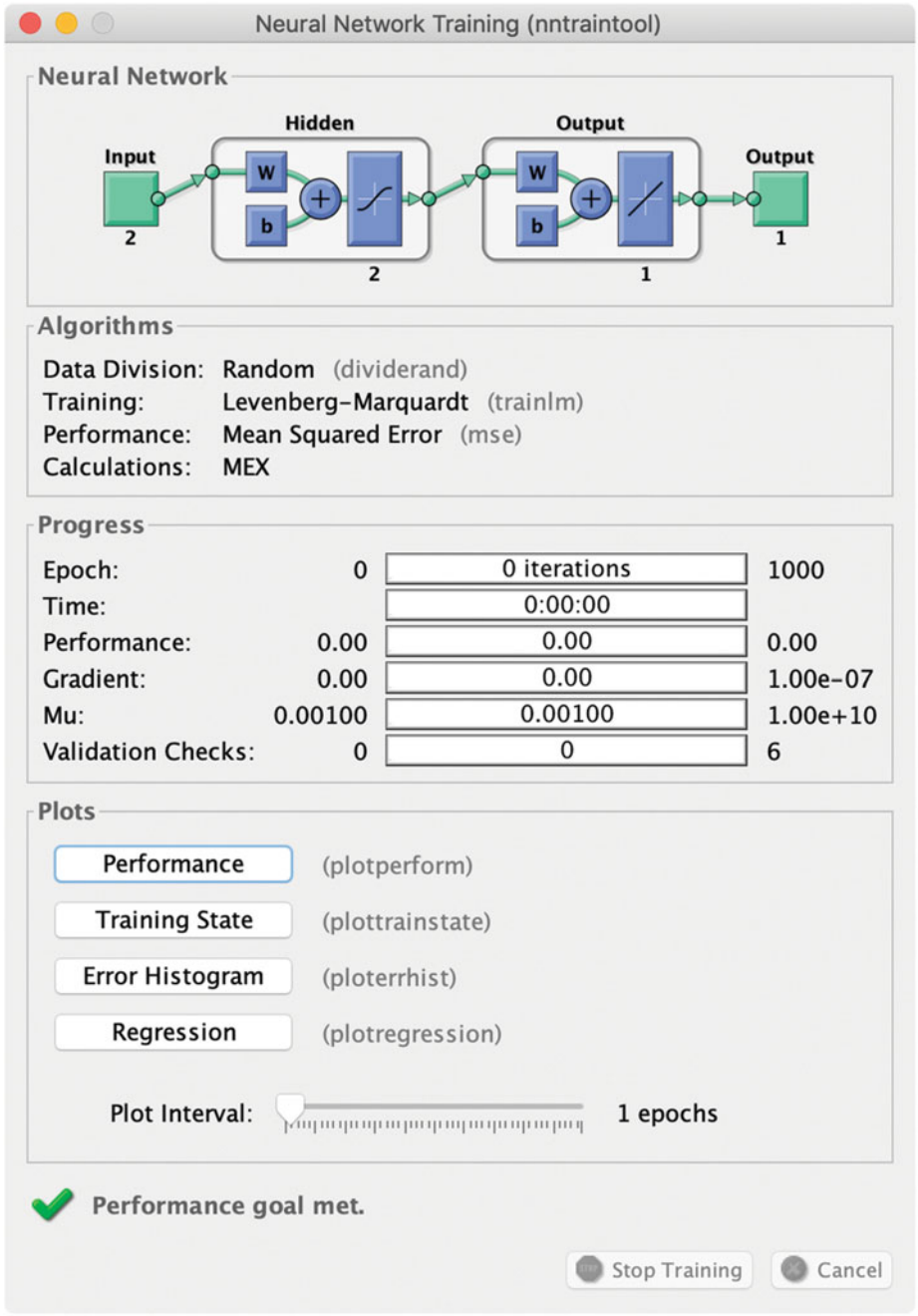
The training GUI is shown in Figure 5.6.

The GUI buttons are described in detail in Chapter 2 in the XOR problem.

The results are shown in the following. The neural net works quite well. The printout in the command window shows it uses two types of activation functions. The output layers use a linear activation function.

```
>> DetectionFilterNN

Regulator  Tachometer    Failed
 0.00e+00  0.00e+00      none
 1.87e-01 -8.14e-05      both
 0.00e+00 -9.35e-02      tach
-1.87e-01 -8.14e-05      regulator

Hidden layer biases -1.980  1.980
Output layer bias    0.159
Input layer weights    0.43  -1.93
                       0.65  -1.87
Output layer weights  -0.65   0.71
Hidden layer activation function tansig
Output layer activation function purelin
```

Figure 5.6: The training GUI.



CHAPTER 6

■ ■ ■

Tokamak Disruption Detection

6.1 Introduction

Tokamaks are fusion machines that are under development to produce baseload power. Baseload power is power that is produced 24/7 and provides the base for powering the electric grid. The International Tokamak Experimental Reactor (ITER) is an international project that will produce net power from a Tokamak. Net power means the Tokamak produces more energy than it consumes. Consumption includes heating the plasma, controlling it, and powering all the auxiliary systems needed to maintain the plasma. It will allow researchers to study the physics of the Tokamak which will hopefully lead the way toward operational machines. A Tokamak is shown in Figure 6.1. The inner poloidal field coils act like a transformer to initiate a plasma current. The outer poloidal and toroidal coils maintain the plasma. The plasma current itself produces its own magnetic field and induces currents in the other coils.

The image in Figure 6.1 was produced by the function `DrawTokamak` which calls `DCoil` and `SquareHoop`. We aren't going to discuss those three functions here. You should feel free to look through the functions as they show how easy it is to do 3D models using MATLAB.

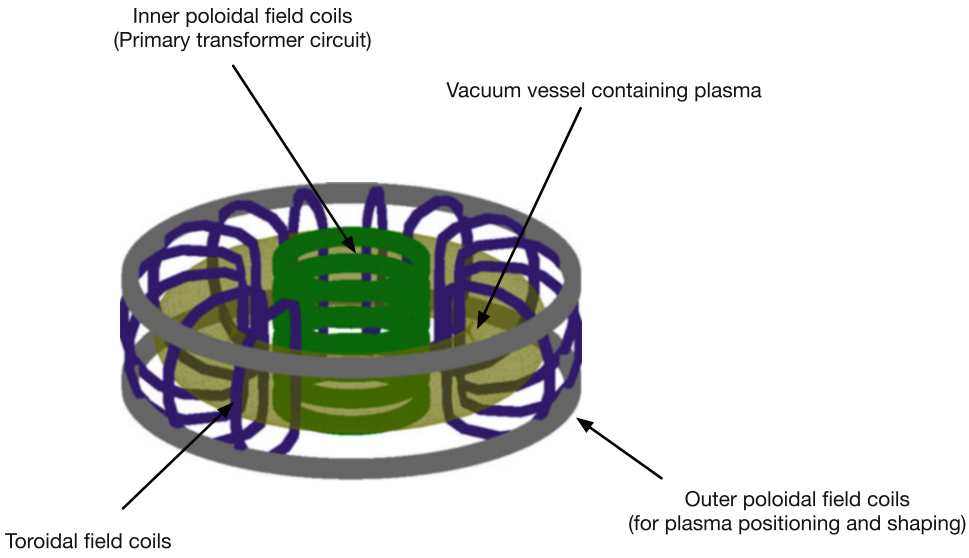
One problem with Tokamaks is disruptions. A disruption is a massive loss of plasma control that extinguishes the plasma and results in large thermal and structural loads on the Tokamak wall. This can lead to catastrophic wall damage. This would be bad in an experimental machine and unacceptable in a power plant as it could lead to months of repairs.

The factors that can be used to predict a disruption [22] are

1. The poloidal beta (beta is the ratio of plasma pressure to magnetic pressure).
2. The line-integrated plasma density.
3. The plasma elongation.
4. The plasma volume divided by the device minor radius.
5. The plasma current.
6. The plasma internal inductance.

M. Paluszek and S. Thomas, *Practical MATLAB Deep Learning*,
https://doi.org/10.1007/978-1-4842-5124-9_6

Figure 6.1: A Tokamak. There are three sets of coils. The inner poloidal field coil initializes the plasma, and the poloidal and toroidal coils maintain the plasma. Some of the toroidal coils are left out to make it easier to see the Tokamak center.

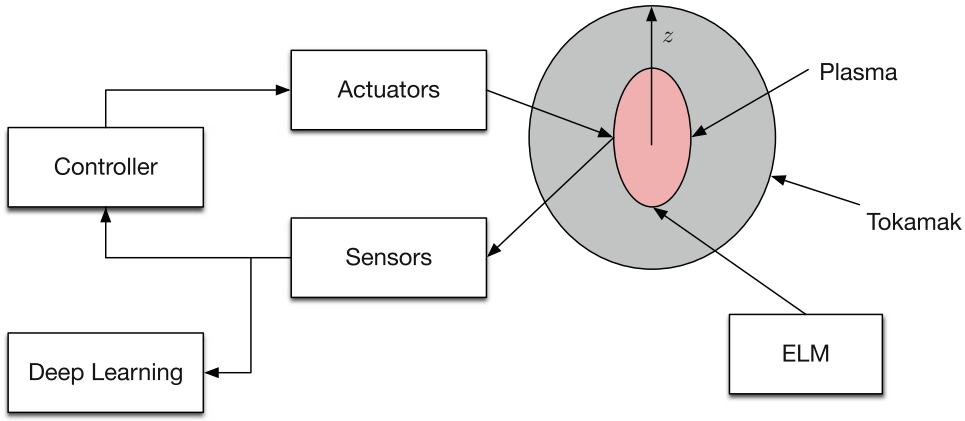


7. The locked mode amplitude.
8. The plasma vertical centroid position.
9. The total input power.
10. The safety factor reaching 95%. Safety factor is the ratio of the times a magnetic field line travels toroidally (the long way around the doughnut) vs. poloidally (short way). We want the safety factor to be greater than 1.
11. The total radiated power.
12. The time derivative of the stored diamagnetic energy.

Locked modes are magnetohydrodynamic (MHD) instabilities that are locked in phase and in the laboratory frame. They can be precursors to disruptions. The plasma internal inductance is the inductance measured by integrating the inductance over the entire plasma. In a Tokamak, the poloidal direction is along the minor radius circumference. The toroidal direction is along the major radius circumference. In a plasma, the dipole moment due to the circulating current is in the opposite direction of the magnetic field which makes it diamagnetic. Diamagnetic energy is the energy stored in a magnetized plasma. Diamagnetic measurements measured this energy.

Our system is shown in Figure 6.2. We will just be looking at the plasma vertical position and the coil currents. We are only going to look at the plasma vertical position in this example. We'll find it to be more than complex enough! We will start with the dynamics of the vertical

Figure 6.2: Tokamak and control system. ELM means ‘‘Edge Localized Mode” and is a disturbance. The plasma is shown in a poloidal cut through the torus.



motion of a plasma. We'll then learn about plasma disturbances. After that, we will design a vertical position controller. Finally, we'll get to the deep learning.

6.2 Numerical Model

6.2.1 Dynamics

For our example, we need a numerical model of disruptions [9], [8], [26]. Ideally our model would include all of the effects in the list given earlier. We use the model in Scibile [27]. We will only consider vertical movement.

The equilibrium force on the plasma is induced by the magnetic field and current density in the plasma.

$$J \times B = \nabla p \tag{6.1}$$

where J is current density, B the magnetic field, and p the pressure. Pressure is force per unit area on the plasma. The momentum balance is [2]

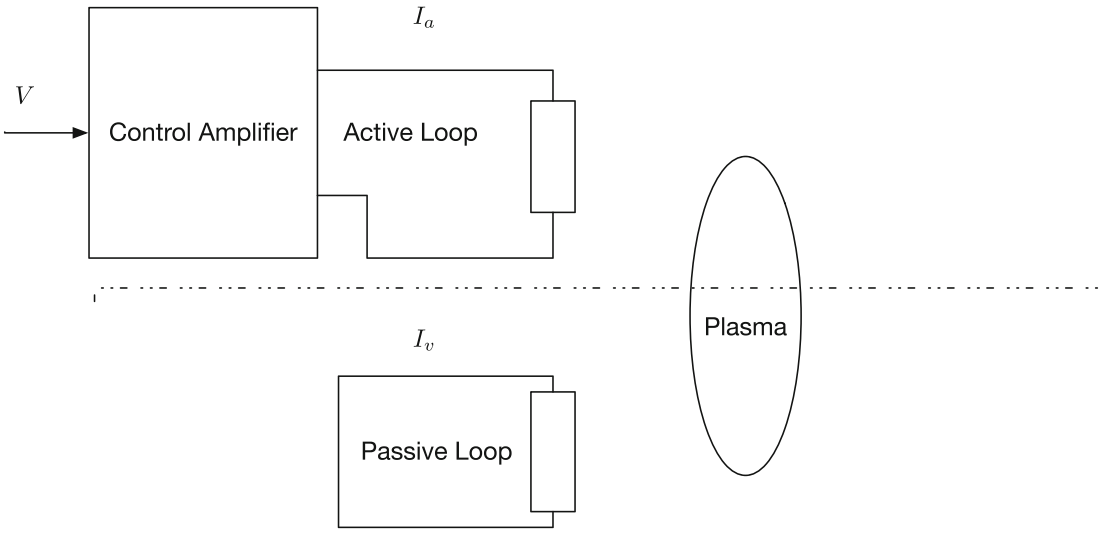
$$\rho \frac{dv}{dt} = J \times B - \nabla p \tag{6.2}$$

where v is the plasma velocity and ρ is the plasma density. The imbalance causes plasma motion, that is, when $J \times B \neq \nabla p$. If we neglect the plasma mass, we get

$$L_P^T I + A_{PP} z I_p = F_p \tag{6.3}$$

L_p is the mutual change inductance matrix of the coils. I is the vector of currents in the Tokamak coils and in the conducting shell around the plasma. F_p is the external force normalized to the plasma current I_p. If we lump currents into active currents, driven by an external voltage,

Figure 6.3: Lumped parameter model.



and passive currents, we get a simplified model of the plasma. We need to add Kirchhoff's voltage law to get a dynamical model.

$$L\dot{I} + RI + L_P\dot{z}I_p = \Gamma V \tag{6.4}$$

Γ couples to voltages to the currents, L is the coil inductance matrix, and R is the coil resistance. If we combine these, we get the state space matrices shown in the following. The lumped model is shown in Figure 6.3.

The dynamical equations are

$$\begin{bmatrix} \dot{I}_a \\ \dot{I}_v \\ \dot{V}_a \end{bmatrix} = A^s \begin{bmatrix} I_a \\ I_v \\ V_a \end{bmatrix} + B^s \begin{bmatrix} V_c \\ \dot{F}_p \\ F_p \end{bmatrix} \tag{6.5}$$

$$z = C^s \begin{bmatrix} I_a \\ I_v \\ V_a \end{bmatrix} + D^s \begin{bmatrix} V_c \\ \dot{F}_p \\ F_p \end{bmatrix} \tag{6.6}$$

$$A^s = \frac{1}{1-k_{av}} \begin{bmatrix} -\frac{R_{aa}}{L_{aa}} & k_{av}\frac{R_{vv}}{L_{av}} & L_{aa} \\ k_{av}\frac{R_{aa}}{L_{av}} & -\frac{R_{vv}}{L_{vv}}\frac{k_{av}-M_{vp}}{1-M_{vp}} & -L_{av} \\ 0 & 0 & -\frac{1-k_{av}}{\tau_t} \end{bmatrix} \tag{6.7}$$

$$B^s = \begin{bmatrix} 0 & 0 & 0 \\ 0 & \frac{1}{L'_{vp}(1-M_{vp})} & 0 \\ \frac{1}{\tau_t} & 0 & 0 \end{bmatrix} \tag{6.8}$$

$$C^s = \frac{1}{A''_{pp}I_p} \begin{bmatrix} -L'_{ap} & L'_{vp} & 0 \end{bmatrix} \tag{6.9}$$

Table 6.1: Model parameters from the Joint European Torus (JET).

Parameter	Description	JET	Units
L_{AA}	Active coil self-inductance	42.5×10^{-3}	H
$L_{AV} = L_{VA}$	Passive coil self-inductance	0.432×10^{-3}	H
L_{VV}	Active-passive coil mutual inductance	0.012×10^{-3}	H
R_{AA}	Active coil resistance	35.0×10^{-3}	Ω
R_{VV}	Passive coil resistance	2.56×10^{-3}	Ω
L'_{AP}	Mutual change inductance between the active coils and plasma displacement	115.2×10^{-6}	H/m
L'_{VP}	Mutual change inductance between the passive coils and plasma displacement	3.2×10^{-6}	H/m
A_{PP}	Normalized destabilizing force	0.5×10^{-6}	H/m^2
F_P	Disturbance force normalized to the plasma current I_p	See ELM	N/A
τ_t	Controller lag	310×10^{-6}	s
I_p	Plasma current	1.5×10^6	A

N/A is Newton (unit of force) per Amp, in this case. Ω is Ohm, the unit of resistance; H is Henry, the unit of inductance; A is amps

$$D^s = \begin{bmatrix} 0 & 0 & \frac{1}{A''_{pp} I_p} \end{bmatrix} \tag{6.10}$$

$$k_{av} = \frac{L_{av}^2}{L_{aa} L_{vv}} \tag{6.11}$$

$$M_{vp} = \frac{A''_{pp} L_{vv}}{L'^2_{vp}} \tag{6.12}$$

$$M_{ap} = \frac{A''_{pp} L_{aa}}{-L'^2_{ap}} \tag{6.13}$$

This includes a first-order lag to replace the pure delay in Scibile. The parameters used in the simulation are given in Table 6.1. The preceding plasma dynamical equations may look very mysterious, but they are really just a variation on a circuit with an inductor and a resistor, shown in Figure 6.4.

The major addition is that the currents produce forces that move the plasma in z.

The equation for this circuit is

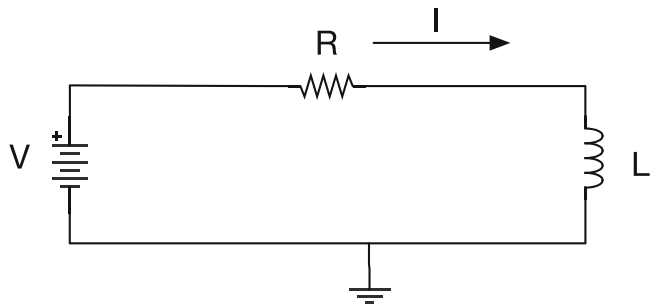
$$L\frac{dI}{dt} + RI = V \tag{6.14}$$

which looks the same as our first-order lag. The first term is the voltage drop across the inductor and the second the voltage drop across the resistor. If we suddenly apply a constant V, we get the analytical solution

$$I = \frac{V}{R}\left(1 - e^{-\frac{R}{L}t}\right) \tag{6.15}$$

L/R is the circuit's time constant τ. As $t \to \infty$, we get the equation for a resistor $V = IR$. You will notice a lot of R/L's in Equation 6.7.

Figure 6.4: Series resistor and inductor circuit, an RL circuit.



6.2.2 Sensors

We are going to assume that we can measure the vertical position and the two currents directly. This is not entirely the case in a real machine. The vertical position is measured indirectly in a real machine. We also assume we have available control voltage.

6.2.3 Disturbances

The disturbances are due to Edge Localized Modes (ELM). An Edge Localized Mode is a disruptive magnetohydrodynamic instability that occurs along the edges of a Tokamak plasma due to steep plasma pressure gradients [18]. The strong pressure gradient is called the edge pedestal. The edge pedestal improves plasma confinement time by a factor of 2 over the low-confinement mode. This is now the preferred mode of operation for Tokamaks. A simple model for an ELM is

$$d = k\left(e^{-\frac{t}{\tau_1}} - e^{-\frac{t}{\tau_2}}\right) \tag{6.16}$$

d is the output of the ELM. It can be scaled by k based on the usage. For example, in Figure 6.5 it is scaled to show the output of a sensor. In our simulation, it is scaled to produce a driving force on the plasma.

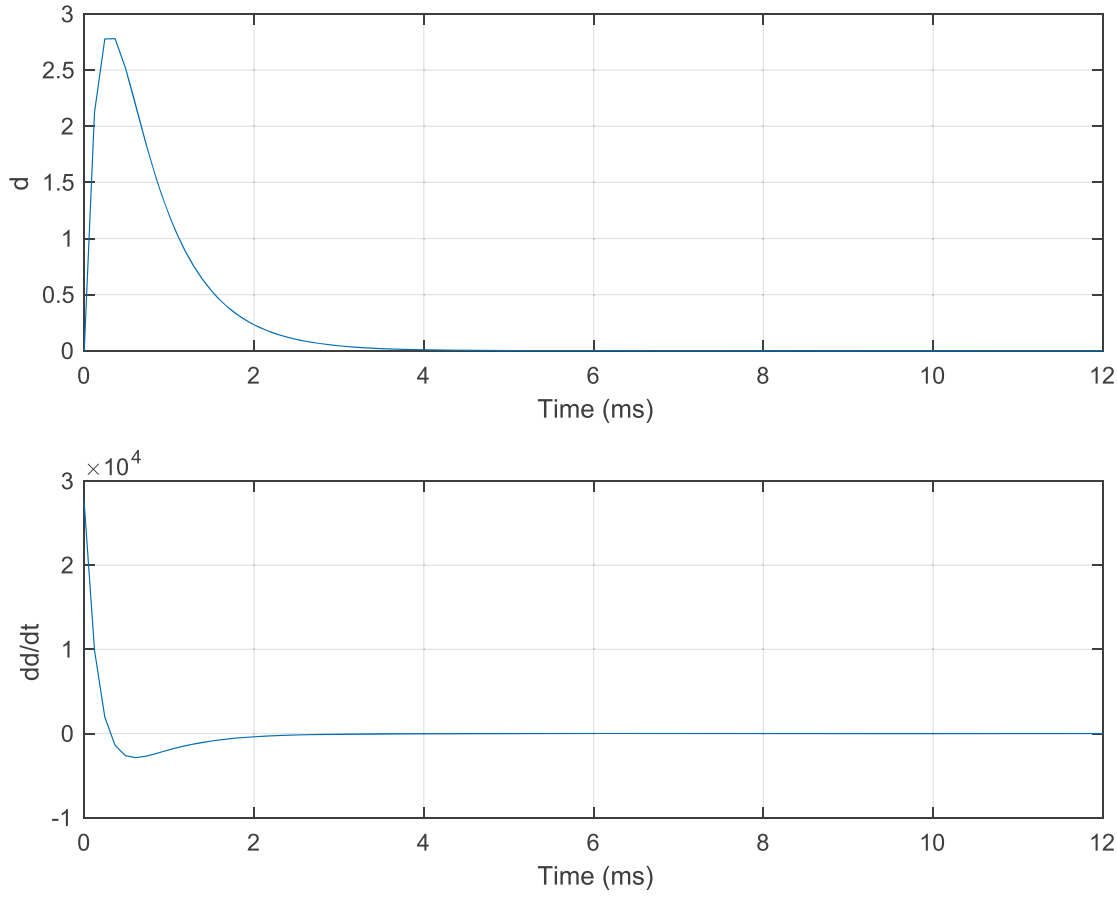
$\tau_1 > \tau_2$ with the ELMs appearing randomly. The function `ELM` produces one ELM. The simulation must call it with a new sequence of times to get a new ELM. Figure 6.5 shows the results of the built-in demo in `ELM`. The function also computes the derivative since the derivative of the disturbance is also an input.

ELM.m

```
1  function eLM = ELM( tau1, tau2, k, t )
2
3  % Constants from the reference
4  if( nargin < 3 )
5    tau1 = 6.0e-4;
6    tau2 = 1.7e-4;
7    k    = 6.5;
8  end
```

```
9
10  % Reproduce the reference results
11  if( nargin < 4 )
12    t = linspace(0,12e-3);
13  end
14
15  d = k*[ exp( -t/tau1 ) - exp( -t/tau2 );...
16          exp( -t/tau2 )/tau2 - exp( -t/tau1 )/tau1 ];
17
18  if( nargout == 0 )
19    PlotSet( t*1000, d, 'x label', 'Time (ms)', 'y label', {'d' 'dd/dt'},
          'figure title', 'ELM' )
20  else
21    eLM = d;
22  end
```

Figure 6.5: Edge Localized Mode.



6.2.4 Controller

We will use a controller to control the vertical position of the plasma, which otherwise is unstable as shown earlier. The controller will be a state space system using full state feedback. The states are the two currents. Position, z, is controlled indirectly. We will use a quadratic regulator. We will use a continuous version. This just means that we need to sample much faster than the range of frequencies for the control.

QCR.m

```
1
2  if( nargin < 1 )
3    Demo
4    return
5  end
6
7  bor = b/r;
8
9  [sinf,rr] = Riccati( [a,-bor*b';-q',-a'] );
10
11 if( rr == 1 )
12   disp('Repeated roots. Adjust q or r');
13 end
14
15  k = r\(b'*sinf);
```

If you get repeated roots, you must manually adjust `q` or `r`, the state and control weights, respectively. The matrix Riccati equation is solved in the subfunction `Riccati`. Notice the use of `unique` to find repeated roots.

QCR.m

```
1  function [sinf, rr] = Riccati( g )
2
3  [w, e] = eig(g);
4
5  [rg,~] = size(g);
6
7  es = sort(diag(e));
8
9  % Look for repeated roots
10 if ( length(unique(es)) < length(es) )
11   rr = 1;
12 else
13   rr = 0;
14 end
15
16 % Sort the columns of w
17 ws   = w(:,real(diag(e)) < 0);
18
19 sinf = real(ws(rg/2+1:rg,:)/ws(1:rg/2,:));
```

The demo is for a double integrator. This is just

$$\ddot{z} = u \tag{6.17}$$

In state space form, this becomes

$$\begin{bmatrix} \dot{z} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} z \\ v \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u \tag{6.18}$$

The states are z, position, and v, velocity. u is the input. The following is the built-in demo code. The demo shows that the function really creates a controller.

```
1
2  a = [0 1;0 0];
3  b = [0;1];
4  q = eye(2);
5  r = 1;
6
7  k = QCR( a, b, q, r );
8
9  e = eig(a-b*k);
10
11 fprintf('\nGain = [%5.2f %5.2f]\n\n',k);
12 disp('Eigenvalues');
13 disp(e)
```

We chose the cost on the states and the control to all be 1.

```
>> QCR

Gain = [ 1.00  1.73]

Eigenvalues
  -0.8660 + 0.5000i
  -0.8660 - 0.5000i
```

We compute the eigenvalues to show that the result is well behaved. The result is critically damped, that is, damping ratio of 0.7071 in the second-order damped oscillator equation.

$$x^2 + 2\zeta\omega x + \omega^2 \tag{6.19}$$

6.3 Dynamical Model

6.3.1 Problem

Create a dynamical model.

6.3.2 Solution

Implement the plasma dynamics model in a MATLAB function.

6.3.3 How It Works

We first code the right-hand side function. We create the four matrices in the `DefaultDataStructure` function. This makes the right-hand side really simple.

RHSTokamak.m

```
function [xDot,z] = RHSTokamak( x, ~, d )

if( nargin < 3 )
  if( nargin == 1 )
    xDot = UpdateDataStructure(x);
  else
    xDot = DefaultDataStructure;
  end

  return;
end

u    = [d.vC;d.eLM];
vDot = (x(3) - d.vC)/d.tauT;
xDot = [d.aS*x(1:2) + d.bS*u;vDot];
z    = d.cS*x(1:2) + d.dS*u;

function d = DefaultDataStructure

d = struct( 'lAA', 42.5e-3, 'lAV', 0.432e-3, 'lVV', 0.012e-3,...
            'rAA', 35.0e-3, 'rVV',2.56e-3,'lAP',115.2e-6,'lVP',3.2e-6,...
            'aPP',0.449e-6,'tauT',310e-6,'iP',1.5e6,'aS',[],'bS',[],'cS',
                []','dS',[],...
            'eLM',0,'vC',0);

d = UpdateDataStructure( d );

function d = UpdateDataStructure( d )

kAV   = d.lAV^2/(d.lAA*d.lVV);
oMKAV = 1 - kAV;
```

```
kA    = 1/(d.lAA*oMKAV);
mVP   = d.aPP*d.lVV/d.lVP^2;
oMMVP = 1 - mVP;

if( mVP >= 1 )
  fprintf('mVP = %f should be less than 1 for an elongated plasma in a
      resistive vacuum vessel. aPP is probably too large\n',mVP);
end

if( kAV >= 1 )
  fprintf('kAV = %f should be less than 1 for an elongated plasma in a
      resistive vacuum vessel\n',kAV);
end

d.aS    =  (1/oMKAV)*[ -d.rAA/d.lAA d.rVV*kAV/d.lAV;...
                        d.rAA*kAV/d.lAV -(d.rVV/d.lVV)*(kAV - mVP)/oMMVP];
d.bS    =  [kA 0 0;kAV/(d.lAV*(1-kAV)) 1/(d.lVP*oMMVP) 0];
d.cS    = -[d.lAP d.lVP]/d.aPP/d.iP;
d.dS    =  [0 0 1]/d.aPP/d.iP;
eAS     = eig(d.aS);

disp('Eigenvalues')
fprintf('\n Mode 1 %12.2f\n Mode 2 %12.2f\n',eAS);
```

If we type `RHSTokamak` at the command line, we get the default data structure.

```
>> RHSTokamak

ans =

  struct with fields:

      lAA: 0.0425
      lAV: 4.3200e-04
      lVV: 1.2000e-05
      rAA: 0.0350
      rVV: 0.0026
      lAP: 1.1520e-04
      lVP: 3.2000e-06
      aPP: 4.0000e-07
     tauT: 3.1000e-04
       aS: [2x2 double]
       bS: [2x3 double]
       cS: [-288 -8]
```

```
       dS: [0 2500000 0]
      eLM: 0
       vC: 0
```

This is the four matrices plus all the constants. The two inputs, control voltage `d.vC`, and ELM disturbances, `d.eLM`, are zero. If you have your own values of `lAA`, and so on, you can do

```
d = RHSTokamak;
d.lAA = 0.046;
d.tauT = 0.00035;
d = RHSTokamak(d)
```

and it will create the matrices. There are two warnings to prevent you from entering invalid parameters.

To see that the system really is unstable type

```
>> RHSTokamak
Eigenvalues

 Mode 1        -2.67
 Mode 2       115.16
 Delay      -3225.81
```

These agree with the reference for JET. The third is the first-order lag. Note that

```
>> d.aPP

ans =

   4.4900e-07
```

This value was chosen so that the roots match the JET numbers.

6.4 Simulate the Plasma

6.4.1 Problem

We want to simulate the vertical position dynamics of the plasma with ELM disturbances.

6.4.2 Solution

Write a simulation script called `DisruptionSim`.

6.4.3 How It Works

The simulation script is an open-loop simulation of the plasma.

DisruptionSim.m

```
%% Constants
d           = RHSTokamak;
tau1ELM     = 6.0e-4;   % ELM time constant 1
tau2ELM     = 1.7e-4;   % ELM time constant 2
kELM        = 1.5e-6;   % ELM gain matches Figure 2.9 in Reference 2
tRepELM     = 48e-3;    % ELM repetition time (s)

%% The control sampling period and the simulation integration time step
dT          = 1e-4;

%% Number of sim steps
nSim        = 1200;

%% Plotting array
xPlot       = zeros(7,nSim);

%% Initial conditions
x           = [0;0;0]; % State is zero
t           = 0; % % Time
tRep        = 0.001; % Time for the 1st ELM
tELM        = inf; % Prevents an ELM at the start
zOld        = 0; % For the first difference rate equation

%% Run the simulation
for k = 1:nSim
  d.v   = 0;
  d.eLM = ELM( tau1ELM, tau2ELM, kELM, tELM );
  tELM  = tELM + dT;

  % Trigger another ELM
  if( t > tRep + rand*tRepELM )
         tELM   = 0;
         tRep   = t;
  end

  x           = RK4( @RHSTokamak, x, dT, t, d );
  [~,z]       = RHSTokamak( x, t, d );
  t           = t + dT;
  zDot        = (z - zOld)/dT;
  xPlot(:,k)  = [x;z;zDot;d.eLM];
end

```

```
%% Plot the results
tPlot = dT*(0:nSim-1)*1000;
yL    = {'I_A' 'I_V'  'v' 'z (m)' 'zDot (m/s)' 'ELM' 'ELMDot'};
k     = [1 2 4 5];
PlotSet( tPlot, xPlot(k,:), 'x label', 'Time (ms)', 'y label', yL(k), '
    figure title', 'Disruption Simulation' );
k     = [5, 6];
PlotSet( tPlot, xPlot(k,:), 'x label', 'Time (ms)', 'y label', yL(k), '
    figure title', 'ZDot and ELM' );
```

It prints out the eigenvalues for reference to make sure the dynamics work correctly.

```
>> DisruptionSim
Eigenvalues

 Mode 1        -2.67
 Mode 2       115.16
 Delay      -3225.81
```

The value for the magnitude of the ELMs was found by running the simulation and looking at the magnitude of $\dot{z}$ and matching them to the results in the reference.

```
tRepELM     = 48e-3;     % ELM repetition time (s)
```

The value of the time derivative of the plasma vertical position z is just the first difference.

```
   zDot         = (z - zOld)/dT;
```

The ELMs are triggered randomly inside of the simulation loop. `tRep` is the time of the last ELM. It adds a random amount of time to this number.

```
   if( t > tRep + rand*tRepELM )
          tELM   = 0;
          tRep   = t;
   end
```

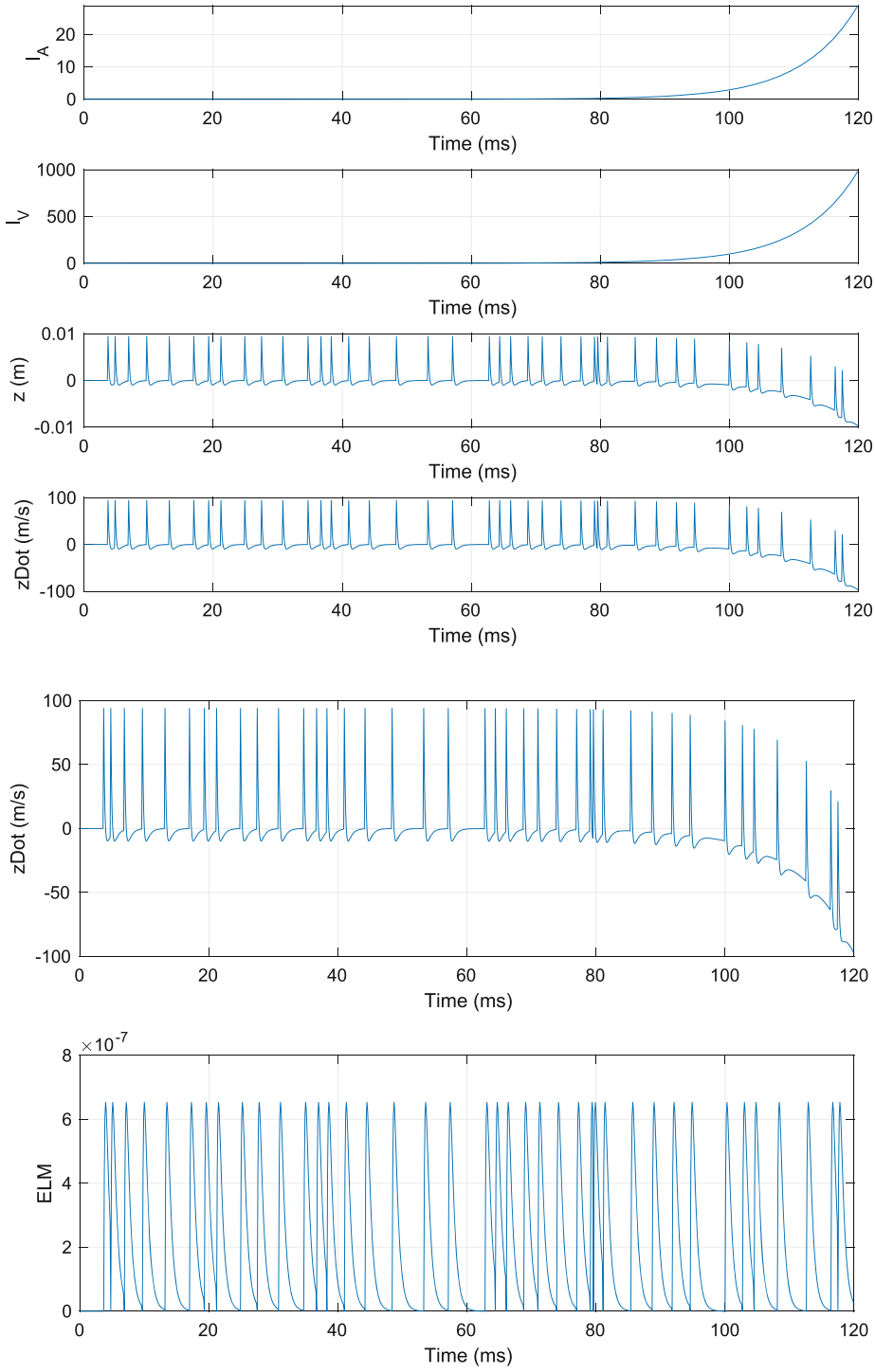
The results are shown in Figure 6.6. The currents grow with time due to the positive eigenvalue. The only disturbance is the ELMs, but they are enough to cause the vertical position to grow.

6.5 Control the Plasma

6.5.1 Problem

We want to control the plasma vertical position.

Figure 6.6: Plasma simulation. The position is unstable.



6.5.2 Solution

Write a simulation script called `ControlSim` to demonstrate closed loop control of the vertical position of the plasma.

6.5.3 How It Works

`ControlSim` is a closed loop simulation of the plasma. We added the controller with the gains computed by `QCR`.

ControlSim.m

```
%% Constants
d           = RHSTokamak;
tau1ELM     = 6.0e-4;   % ELM time constant 1
tau2ELM     = 1.7e-4;   % ELM time constant 2
kELM        = 1.5e-6;   % ELM gain matches Figure 2.9 in Reference 2
tRepELM     = 48e-3;    % ELM repetition time (s)
controlOn   = true;
vCMax       = 3e-4;

%% The control sampling period and the simulation integration time step
dT          = 1e-5;

%% Number of sim steps
nSim        = 20000;

%% Plotting array
xPlot       = zeros(8,nSim);

%% Initial conditions
x           = [0;0;0];
t           = 0;
tRep        = 0.001; % Time for the 1st ELM
tELM        = inf;   % This value will be change after the first ELM
zOld        = 0;     % For the rate equation
z           = 0;

%% Design the controller
kControl    = QCR( d.aS, d.bS(:,1), eye(2), 1 );

%% Run the simulation
for k = 1:nSim
  if( controlOn )
    d.vC = -kControl*x(1:2);
    if( abs(d.vC) > vCMax )
      d.vC = sign(d.vC)*vCMax;
    end
```

```
53    else
54      d.vC        = 0; %#ok<UNRCH>
55    end
56
57    d.eLM = ELM( tau1ELM, tau2ELM, kELM, tELM );
58    tELM  = tELM + dT;
59
60    % Trigger another ELM
61    if( t > tRep + rand*tRepELM )
62           tELM   = 0;
63           tRep   = t;
64    end
65
66    x           = RK4( @RHSTokamak, x, dT, t, d );
67    [~,z]       = RHSTokamak( x, t, d ); % Get the position
68    t           = t + dT;
69    zDot        = (z - zOld)/dT; % The rate of the vertical position
70    xPlot(:,k)  = [x;z;zDot;d.eLM;d.vC];
71  end
```

The controller is implemented in the loop. It applies the limiter.

```
48      if( abs(d.vC) > vCMax )
49        d.vC = sign(d.vC)*vCMax;
50      end
51    else
52      d.vC        = 0; %#ok<UNRCH>
53    end
54
55    d.eLM = ELM( tau1ELM, tau2ELM, kELM, tELM );
```

Results are shown in Figure 6.7.

6.6 Training and Testing

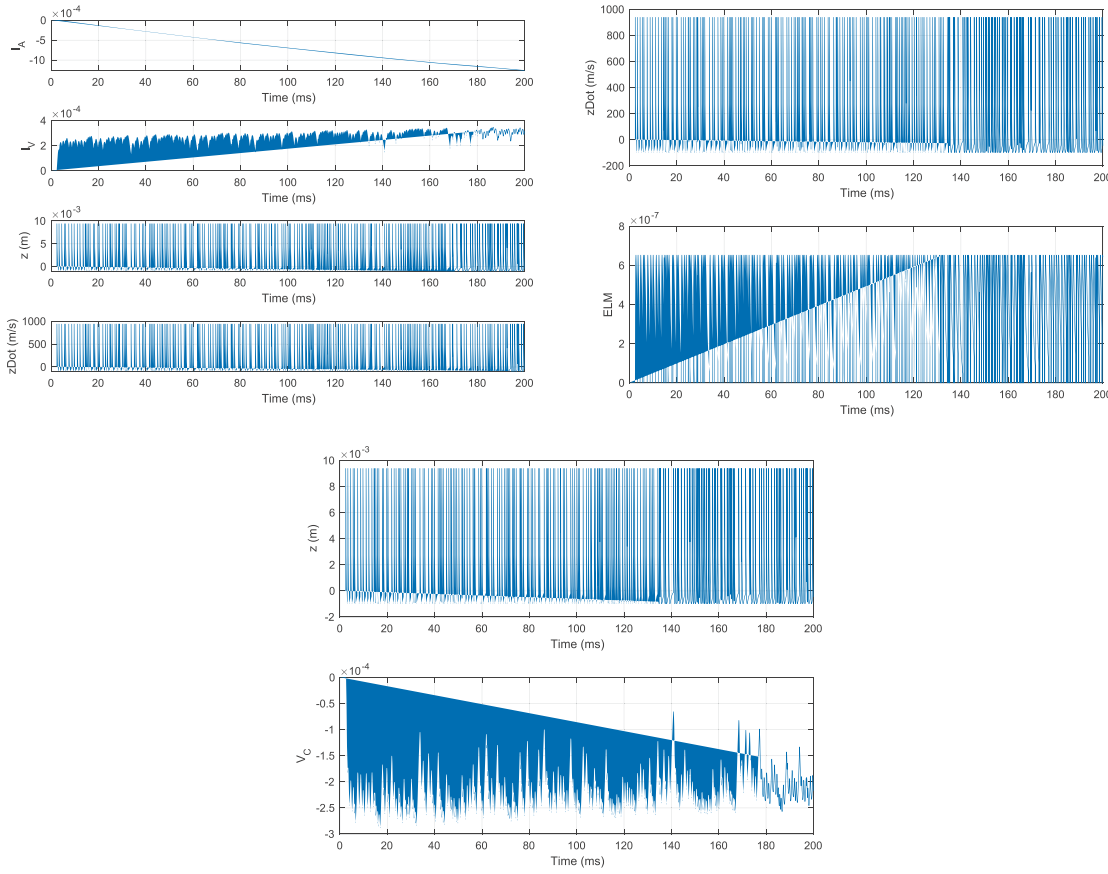
6.6.1 Problem

We want to detect measurements leading up to disruptions.

6.6.2 Solution

We use a BiLSTM (bidirectional long short-term memory) layer to detect disruptions by classifying a time sequence as leading up to a disruption or not. LSTMs are designed to avoid the dependency on old information. A standard RNN has a repeating structure. An LSTM also has a repeating structure, but each element has four layers. The LSTM layers decide what old information to pass on to the next layer. It may be all, or it may be none. There are many variants on LSTM, but they all include the fundamental ability to forget things. BiLSTM is generally better than an LSTM when we have the full time sequence.

Figure 6.7: Plasma simulation. The position is now bounded. Compare this to Figure 6.6.



6.6.3 How It Works

The following script, `TokamakNeuralNet.m`, generates the test and training data, trains the neural net, and tests it. The constants are initialized first.

TokamakNeuralNet.m

```
%% Constants
d            = RHSTokamak;
tau1ELM      = 6.0e-4;    % ELM time constant 1
tau2ELM      = 1.7e-4;    % ELM time constant 2
kELM         = 1.5e-6;    % ELM gain matches Figure 2.9 in Reference 2
tRepELM      = 48e-3;     % ELM repetition time (s)
controlOn    = true;      % Turns on the controller
disThresh    = 1.6e-6;    % This is the threshold for a disruption

% The control sampling period and the simulation integration time step
dT           = 1e-5;
```

```
17
18 % Number of sim steps
19 nSim         = 2000;
20
21 % Number of tests
22 n            = 100;
23 sigma1ELM    = 2e-6*abs(rand(1,n));
24
25 PlotSet(1:n,sigma1ELM,'x label','Test Case','y label','1 \sigma ELM Value
       ');
26
27 zData        = zeros(1,nSim); % Storage for vertical position
```

We design the controller as we did in `ControlSim`. The script runs 100 simulations. The linear quadratic controller demonstrated in `ControlSim` controls the position.

```
28 %% Initial conditions
29 x            = [0;0;0]; % The state of the plasma
30 tRep         = 0.001; % Time for the 1st ELM
31
32 %% Design the controller
33 kControl     = QCR( d.aS, d.bS(:,1), eye(2), 1 );
34
35 s            = cell(n,1);
36
37 %% Run n simulation
38 for j = 1:n
39         % Run the simulation
40   t      = 0;
41   tELM   = inf; % Prevents an ELM at the start
42         kELM   = sigma1ELM(j);
43   tRep   = 0.001; % Time for the 1st ELM
44
45   for k = 1:nSim
46     if( controlOn )
47       d.vC = -kControl*x(1:2);
48     else
49       d.vC       = 0; %#ok<UNRCH>
50     end
51
52     d.eLM        = ELM( tau1ELM, tau2ELM, kELM, tELM );
53     tELM         = tELM + dT;
54
55     % Trigger another ELM
56     if( t > tRep + rand*tRepELM )
57       tELM       = 0;
58       tRep       = t;
59     end
60
61     x            = RK4( @RHSTokamak, x, dT, t, d );
62     [~,z]        = RHSTokamak( x, t, d );
```

```
    t           = t + dT;
    zData(1,k)  = z;
  end
  s{j} = zData;
end

clear c
```

A disruption is any time response with z peaks over the threshold. Figure 6.8 shows the responses and the distribution of standard deviations. The blue line is a simulation that failed to keep vertical displacement of the plasma field under the prescribed threshold, and the red line is a simulation that succeeded in keeping the displacements below that threshold. The classification criteria are set in the following code.

```
%% Classify the results
j       = find(sigma1ELM > disThresh);
jN      = find(sigma1ELM < disThresh);
c(j,1)  = 1;
c(jN,1) = 0;

[t,tL] = TimeLabel((0:nSim-1)*dT);
PlotSet(t,[s{j(1)};s{jN(1)}],'x label',tL,'y label','z (m)','Plot Set'
    ,{1:2},'legend',{{'disruption','stable'}});
```

The training is done next.

```
%% Divide into training and testing data
nTrain  = floor(0.8*n); % Train on 80% of the cases
xTrain  = s(1:nTrain);
yTrain  = categorical(c(1:nTrain));
xTest   = s(nTrain+1:n);
yTest   = categorical(c(nTrain+1:n));

%% Train the neural net
numFeatures     = 1; % Just the plasma position
numClasses      = 2; % Disruption or non disruption
numHiddenUnits  = 200;

layers = [ ...
    sequenceInputLayer(numFeatures)
    bilstmLayer(numHiddenUnits,'OutputMode','last')
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
disp(layers)

options = trainingOptions('adam', ...
    'MaxEpochs',60, ...
    'GradientThreshold',2, ...
    'Verbose',0, ...
```

Figure 6.8: Time responses and distribution of 1-sigma values.

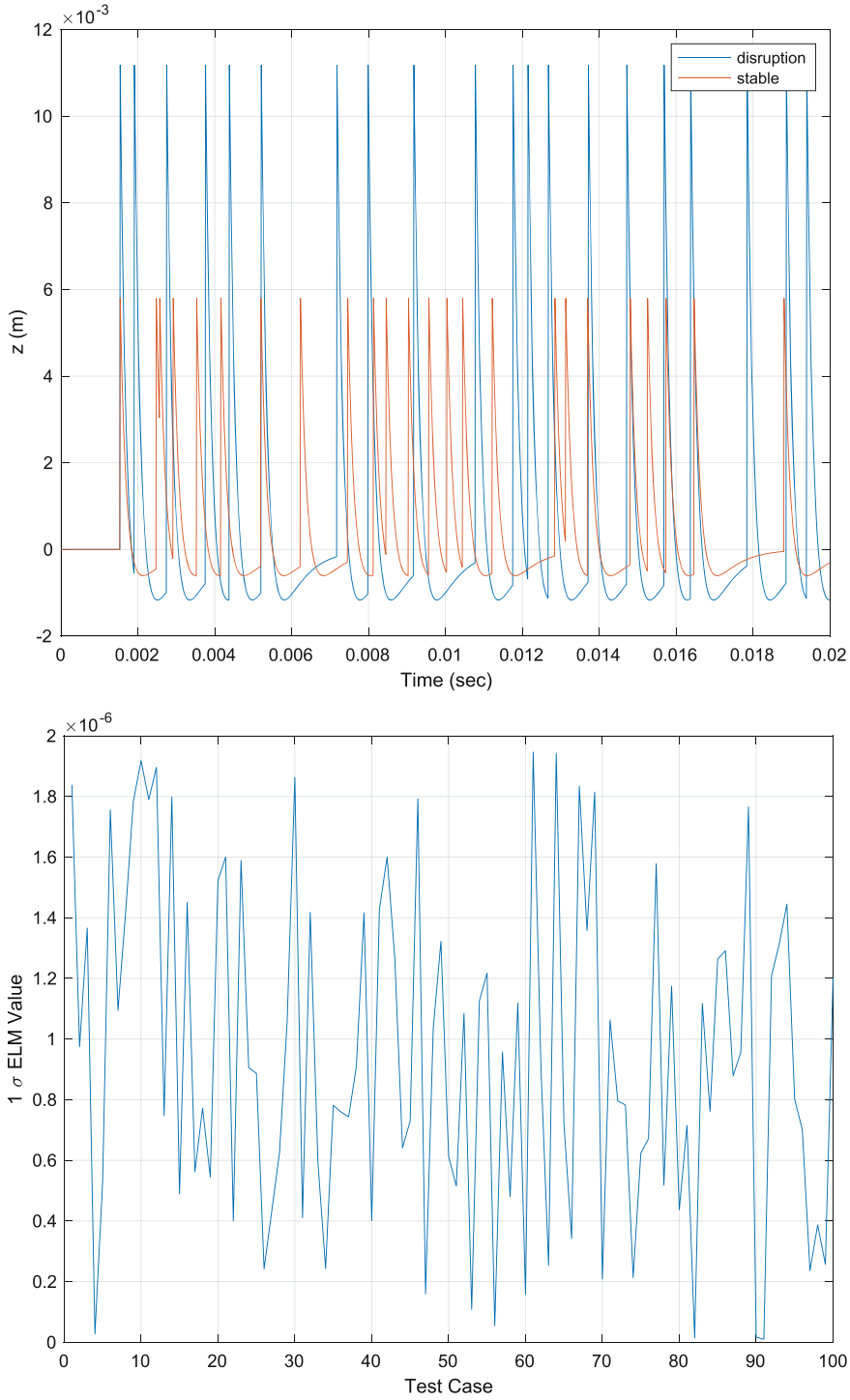
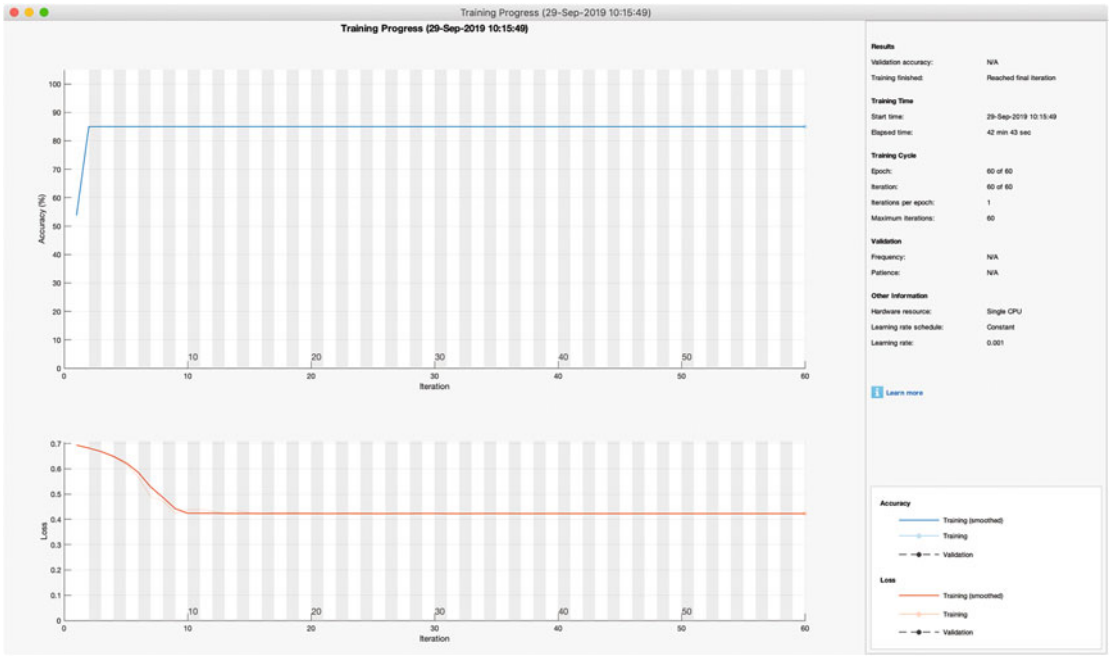


Figure 6.9: Training.




```
105      'Plots','training-progress');
106
107  net = trainNetwork(xTrain,yTrain,layers,options);
```

The training is shown in Figure 6.9.
The testing is done next.

```
108  %% Demonstrate the neural net
109
110  %%% Test the network
111  yPred    = classify(net,xTest);
112
113  % Calculate the classification accuracy of the predictions.
114  acc          = sum(yPred == yTest)./numel(yTest);
115  disp('Accuracy')
116  disp(acc);
```

The results are encouraging. ITER will require 95% of disruption predictions to be correct and to present an alarm 30 ms before a disruption [25]. Good results have been obtained using data from DIII-D [17].

```
>> TokamakNeuralNet
Eigenvalues

 Mode 1        -2.67
 Mode 2       115.16
  5x1 Layer array with layers:

     1   ''   Sequence Input          Sequence input with 1 dimensions
     2   ''   BiLSTM                  BiLSTM with 200 hidden units
     3   ''   Fully Connected         2 fully connected layer
     4   ''   Softmax                 softmax
     5   ''   Classification Output   crossentropyex
Accuracy
    0.7500
```

This chapter did not deal with recursive or online training. A disruption prediction would need to constantly incorporate new data into its neural network. In addition, the other criteria for disruption detection would also need to be incorporated.

CHAPTER 7

■ ■ ■

Classifying a Pirouette

7.1 Introduction

A pirouette is a familiar step in ballet. There are many types of pirouettes. We will focus on an en dehors (outside) pirouette from fourth position. The dancer pliés (does a deep knee bend) then straightens her legs producing both an upward force to get on the tip of her pointe shoe and a torque to turn about her axis of revolution.

In this chapter, we will classify pirouettes. Four dancers will each do ten double pirouettes, and we will use them to train the deep learning network. The network can then be used to classify pirouettes.

This chapter will involve real-time data acquisition and deep learning. We will spend a considerable amount of time in this chapter creating software to interface with the hardware. While it is not deep learning, it is important to know how to get data from sensors for use in deep learning work. We give code snippets in this chapter. Only a few can be cut and pasted into the MATLAB command window. You'll need to run the software in the downloadable library. Also remember, you will need the Instrument Control Toolbox for this project.

Our subject dancers showing a pirouette are shown in Figure 7.1. We have three female dancers and one male dancer. Two of the women are wearing pointe shoes. The measurements will be accelerations, angular rates, and orientation. There really isn't any limit to the movements the dancers could do. We asked them all to do double pirouettes starting from fourth position and returning to fourth position. Fourth position is with one foot behind the other and separated by a quarter meter or so. This is in contrast to fifth position where the feet are right against each other. Each is shown at the beginning, middle, and end of the turn. All have slightly different positions, though all are doing very good pirouettes. There is no one ''right'' pirouette. If you were to watch the turns, you would not be able to see that they are that different. The goal is to develop a neural network that can classify their pirouettes.

This kind of tool would be useful in any physical activity. An athlete could train a neural network to learn any important movement. For example, a baseball pitcher's pitch could be learned. The trained network could be used to compare the same movement at any other time to see if it has changed. A more sophisticated version, possibly including vision, might suggest

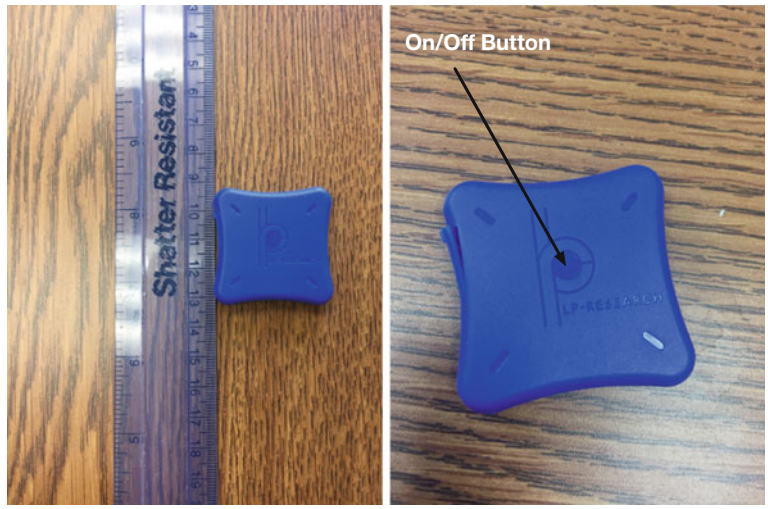
M. Paluszek and S. Thomas, *Practical MATLAB Deep Learning*,
https://doi.org/10.1007/978-1-4842-5124-9_7

Figure 7.1: Dancers doing pirouettes. The stages are from left to right.

Table 7.1: LPMS-B2: 9-Axis Inertial Measurement Unit (IMU).

Parameter	**Description**
Bluetooth	2.1 + EDR / Low Energy (LE) 4.1
Communication distance	< 20 m
Orientation range	Roll: $\pm 180°$; Pitch: $\pm 90°$; Yaw: $\pm 180°$
Resolution	$< 0.01°$
Accuracy	$< 0.5°$(static), $< 2°$ RMS (dynamic)
Accelerometer	3 axes, $\pm 2/\pm 4/\pm 8/\pm 16$ g, 16 bits
Gyroscope 3 axes	$\pm 125/\pm 245/\pm 500/\pm 1000/\pm 2000°$/s, 16 bits
Data output format	2 Raw data / Euler angle / Quaternion
Data transmission rate up to	2400Hz

Figure 7.2: LPMS-B2: 9-Axis Inertial Measurement Unit (IMU). The on/off button is highlighted on the right.



how to fix problems or identify what has changed. This would be particularly valuable for rehabilitation.

7.1.1 Inertial Measurement Unit

Our sensing means will be the LPMS-B2 IMU with its parameters shown in Table 7.1 that has Bluetooth. The range is sufficient to work in a ballet studio.

The IMU has many other outputs that we will not use. A close up of the IMU is shown in Figure 7.2.

We will first work out the details of the data acquisition. We will then build a deep learning algorithm to train the system and later to take data and classify the pirouette as being a pirouette done by a particular dancer. We'll build up the data acquisition by first writing the MATLAB

code to acquire the data. We will then create functions to display the data. We will then integrate it all into a GUI. Finally, we will create the deep learning classification system.

7.1.2 Physics

A pirouette is a complex multiflexible body problem. The pirouette is initiated by the dancer doing a pliè and then using his or her muscles to generate a torque about the spin axis and forces to get onto her pointe shoe and over her center of mass. Her muscles quickly stop the translational motion so that she can focus on balancing as she is turning. The equation of rotational motion is known as Euler's equation and is

$$T = I\dot{\omega} + \omega^{\times} I\omega \tag{7.1}$$

where ω is the angular rate and I is our inertia. T is our external torque. The external torque is due to a push off the floor and gravity. This is vector equation. The vectors are T and ω. I is a 3×3 matrix.

$$T = \begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix} \tag{7.2}$$

$$\omega = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} \tag{7.3}$$

Each component is a value about a particular axis. For example, T_x is the torque about the x-axis attached to the dancer. Figure 7.3 shows the system. We will only be concerned with rotation assuming all translational motion is damped. If the dancer's center of mass is not above the box of her pointe shoe, she will experience an overturning torque.

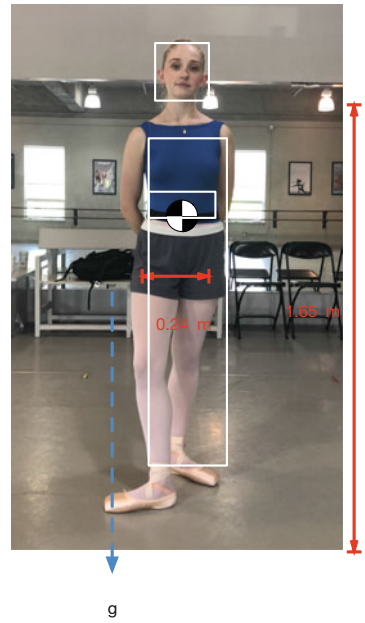
The dynamical model is three first-order couple differential equations. Angular rate ω is the state, that is, the quantity being differentiated. The equation says that the external torque (due to pushing off the floor or due to pointe shoe drag) is equal to the angular acceleration plus the Euler coupling term. This equation assumes that the body is rigid. For a dancer, it means she is rotating and no part is moving with respect to any other part. Now in a proper pirouette, this is never true if you are spotting! But let's suppose you are one of those dancers who don't spot. Let's forget about the angular rate coupling term, which only matters if the angular rate is large. Let's just look at the first two terms which are $T = I\dot{\omega}$. Expanded

$$\begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix} = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{xy} & I_{yy} & I_{yz} \\ I_{xz} & I_{yz} & I_{zz} \end{bmatrix} \begin{bmatrix} \dot{\omega}_x \\ \dot{\omega}_y \\ \dot{\omega}_z \end{bmatrix} \tag{7.4}$$

Let's look at the equation for T_z. We just multiply the first row of the inertia matrix times the angular rate vector.

$$T_z = I_{xz}\dot{\omega}_x + I_{yz}\dot{\omega}_y + I_{zz}\dot{\omega}_z \tag{7.5}$$

Figure 7.3: The center of mass of the dancer. External forces act at the center of mass. Rotations are about the center of mass.



This means a torque around the z-axis influences the angular rates about all 3 axes.

We can write out the required torque.

$$\begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix} = \begin{bmatrix} I_{xz} \\ I_{yz} \\ I_{zz} \end{bmatrix} \dot{\omega}_z \tag{7.6}$$

This is the perfect pirouette push off because it only creates rotation about the vertical axis, which is what we want in a pirouette. To this you need to add the forces needed to get on pointe with your center of mass over your pointe shoe tip.

While turning, the only significant external torque is due to friction between the pointe shoe tip and the floor. Friction resists both turning motion and translational motion. You don't want a slight side force, perhaps due to a less than great partner, to cause you to slide.

Our IMU measures angular rates and linear accelerations. Angular rates are the quantities in Euler equations. However, since the IMU is not at the dancer's center of mass, it will also measure angular accelerations along with the acceleration of the center of mass. We locate it at the dancer's waist so it is not too far from the spin axis but it still sees a component.

$$a_\omega = r_{\text{IMU}}^{\times} \dot{\omega} \tag{7.7}$$

where r_{IMU} is the vector from the dancer's center of mass to the IMU.

For a dancer doing a pirouette, Euler's equation is not sufficient. A dancer can transfer momentum internally to stop a pirouette and needs a little jump to get on demi-pointe or pointe. To model this, we add additional terms.

$$T = I\dot{\omega} + \omega^{\times}\left[I\omega + uI_i(\Omega_i + \omega_z) + uI_h(\Omega_h + \omega_z)\right] + u\left(T_i + T_h\right) \tag{7.8}$$

$$T_i = I_i\left(\dot{\Omega}_i + \dot{\omega}_z\right) \tag{7.9}$$

$$T_h = I_h\left(\dot{\Omega}_h + \dot{\omega}_z\right) \tag{7.10}$$

$$F = m\ddot{z} \tag{7.11}$$

where m is the mass, F is the vertical force, and z is the vertical direction. I_i is the internal inertia for control, and I_h is the head inertia. I includes both of these already. That is, I is the total body inertia that includes the internal ''wheel," body, and head. T_i is the internal torque. T_h is the head torque (for spotting). The internal torques, T_i, and T_h are between the body and the internal ''wheel" or head. For example, T_h causes the head to move one way and the body the other. If you are standing, the torque you produce from your feet against the floor prevents your body from rotating. T is the external torque due to friction and the initial push off by the feet. The unit vector is

$$u = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{7.12}$$

There are six equations in total. The first is a vector equation with three components, the second two are scalar equations. The vector equation is three equations, and each scalar equation is just one equation. We can use these to create a simulation of a dancer. The second component models all z-axis internal rotation, including spotting.

7.2 Data Acquisition

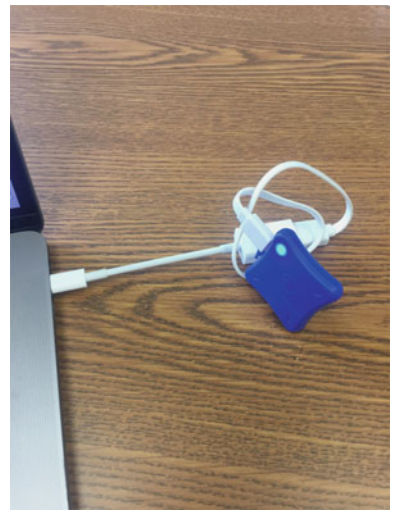
7.2.1 Problem

We want to get data from the Bluetooth IMU.

7.2.2 Solution

We will use the MATLAB `bluetooth` function. We'll create a function to read data from the IMU.

Figure 7.4: The IMU is connected to the bluetooth device from a MacBook Pro via USB C and a ubiquitous Mac dongle. This is only for charging purposes. Once it is charged, you can use without the dongle.



7.2.3 How It Works

We will write an interface to the bluetooth device. First make sure the IMU is charged. Connect it to your computer as shown in Figure 7.4. Push the button on the back. This turns it on and off. The status is indicated by the LED. The IMU comes with support software from the vendor, but you will not need any of their software as MATLAB does all the hard work for you.

Let's try commanding the IMU. Type `btInfo = instrhwinfo('Bluetooth')` and you should get the following:

```
>> btInfo = instrhwinfo('Bluetooth')

btInfo =

  HardwareInfo with properties:

        RemoteNames: {'LPMSB2-4B31D6'}
          RemoteIDs: {'btspp://00043E4B31D6'}
    BluecoveVersion: 'BlueCove-2.1.1-SNAPSHOT'
     JarFileVersion: 'Version 4.0'

Access to your hardware may be provided by a support package. Go to the
    Support Package Installer to learn more.
```

This shows that your IMU is discoverable. There is no support package available from the MathWorks. Now type `b = Bluetooth(btInfo.RemoteIDs1,1)` (this can be slow). The number is the channel. The `Bluetooth` function requires the Instrument Control Toolbox

for MATLAB.

```
>> b = Bluetooth(btInfo.RemoteIDs{1},1)

   Bluetooth Object : Bluetooth-btspp://00043E4B31D6:1

   Communication Settings
      RemoteName:          LPMSB2-4B31D6
      RemoteID:            btspp://00043e4b31d6
      Channel:             1
      Terminator:          'LF'

   Communication State
      Status:              closed
      RecordStatus:        off

   Read/Write State
      TransferStatus:      idle
      BytesAvailable:      0
      ValuesReceived:      0
      ValuesSent:          0
```

Note that the Communication State Status shows closed. We need to open the device by typing `fopen(b)`. If you don't have this device, just type

```
>> btInfo = instrhwinfo('Bluetooth')
btInfo =
  HardwareInfo with properties:

        RemoteNames: []
          RemoteIDs: []
    BluecoveVersion: 'BlueCove-2.1.1-SNAPSHOT'
     JarFileVersion: 'Version 4.0'
Access to your hardware may be provided by a support package. Go to the
    Support Package Installer to learn more.
```

This says it cannot recognize remote names or ids. You may need a support package for your device in this case.

Click connect and the device will open. Now type `a= fscanf(b)` and you will get a bunch of unprintable characters. We now have to write code to command the device. We will leave the device in streaming mode. The data unit format is shown in Table 7.2. Each packet is really 91 bytes long even though the table only shows 67 bytes. The 67 bytes are all the useful data.

We read the binary and put it into a data structure using `DataFromIMU`. `typecast` converts from bytes to float.

Table 7.2: Reply data.

Byte	Content (hex)	Meaning
0	3A	Packet Start
1	01	OpenMAT ID LSB (ID=1)
2	00	OpenMAT MSB
3	09	Command No. LSB (9d = GET_SENSOR_DATA)
4	00	Command No. MSB
5	00	Data Length LSB
6	00	Data Length MSB
7--10	xxxxxxxx	Timestamp
11--14	xxxxxxxx	Gyroscope data x-axis
15--18	xxxxxxxx	Gyroscope data y-axis
19--22	xxxxxxxx	Gyroscope data z-axis
23--26	xxxxxxxx	Accelerometer x-axis
27--30	xxxxxxxx	Accelerometer y-axis
31--34	xxxxxxxx	Accelerometer z-axis
35--38	xxxxxxxx	Magnetometer x-axis
39--42	xxxxxxxx	Magnetometer y-axis
43--46	xxxxxxxx	Magnetometer z-axis
47--50	xxxxxxxx	Orientation quaternion q0
51--54	xxxxxxxx	Orientation quaternion q1
55--58	xxxxxxxx	Orientation quaternion q2
59--62	xxxxxxxx	Orientation quaternion q3
63	xx	Check sum LSB
64	xx	Check sum MSB
65	0D	Message end byte 1
66	0A	Message end byte 2

DataFromIMU.m

```
25  function d = DataFromIMU( a )
26
27  d.packetStart   = dec2hex(a(1));
28  d.openMATIDLSB  = dec2hex(a(2));
29  d.openMATIDMSB  = dec2hex(a(3));
30  d.cmdNoLSB      = dec2hex(a(4));
31  d.cmdNoMSB      = dec2hex(a(5));
32  d.dataLenLSB    = dec2hex(a(6));
33  d.dataLenMSB    = dec2hex(a(7));
34  d.timeStamp     = BytesToFloat( a(8:11) );
35  d.gyro          = [ BytesToFloat( a(12:15) );...
36                      BytesToFloat( a(16:19) );...
37                      BytesToFloat( a(20:23) )];
38  d.accel         = [ BytesToFloat( a(24:27) );...
39                      BytesToFloat( a(28:31) );...
40                      BytesToFloat( a(32:35) )];
41  d.quat          = [ BytesToFloat( a(48:51) );...
42                      BytesToFloat( a(52:55) );...
```

```
                    BytesToFloat( a(56:59) );...
                    BytesToFloat( a(60:63) )];
d.msgEnd1       = dec2hex(a(66));
d.msgEnd2       = dec2hex(a(67));

%% DataFromIMU>BytesToFloat
function r = BytesToFloat( x )

r = typecast(uint8(x),'single');
```

We've wrapped all of this into the script `BluetoothTest.m`. We print out a few samples of the data to make sure our bytes are aligned correctly.

BluetoothTest.m

```
%% Script to read binary from the IMU

% Find available Bluetooth devices
btInfo = instrhwinfo('Bluetooth')

 % Display the information about the first device discovered
btInfo.RemoteNames(1)
btInfo.RemoteIDs(1)

% Construct a Bluetooth Channel object to the first Bluetooth device
b = Bluetooth(btInfo.RemoteIDs{1}, 1);

% Connect the Bluetooth Channel object to the specified remote device
fopen(b);

% Get a data structure
tic
t = 0;
for k = 1:100
  a   = fread(b,91);
  d   = DataFromIMU( a );
  fprintf('%12.2f [%8.1e %8.1e %8.1e] [%8.1e %8.1e %8.1e] [%8.1f %8.1f
      %8.1f %8.1f]\n',t,d.gyro,d.accel,d.quat);
  t = t + toc;
  tic
end
```

```
When we run the script we get the following output.

>> BluetoothTest

btInfo =

  HardwareInfo with properties:

        RemoteNames: {'LPMSB2-4B31D6'}
```

```
           RemoteIDs: {'btspp://00043E4B31D6'}
     BluecoveVersion: 'BlueCove-2.1.1-SNAPSHOT'
      JarFileVersion: 'Version 4.0'

Access to your hardware may be provided by a support package. Go to the
    Support Package Installer to learn more.

ans =

  1x1 cell array

    {'LPMSB2-4B31D6'}

ans =

  1x1 cell array

    {'btspp://00043E4B31D6'}

ans =

  1x11 single row vector

    1.0000    0.0014    0.0023   -0.0022    0.0019   -0.0105   -0.9896
           0.9200   -0.0037    0.0144    0.3915

ans =

  1x11 single row vector

    2.0000   -0.0008    0.0023   -0.0016    0.0029   -0.0115   -0.9897
           0.9200   -0.0037    0.0144    0.3915

ans =

  1x11 single row vector

    3.0000    0.0004    0.0023   -0.0025    0.0028   -0.0125   -0.9900
           0.9200   -0.0037    0.0144    0.3915
```

The first number in each row vector is the sample, the next three are the angular rates from the gyro, the next three the accelerations, and the last four the quaternion. The acceleration is mostly in the -z direction which means that +z is in the button direction. Bluetooth, like all wireless connections, can be problematic. If you get this error

```
Index exceeds the number of array elements (0).

Error in BluetoothTest (line 7)
btInfo.RemoteNames(1)
```

turn the IMU on and off. You might also have to restart MATLAB at times. This is because RemoteNames is empty, and this test is assuming it will not be. MATLAB then gets confused.

7.3 Orientation

7.3.1 Problem

We want to use quaternions to represent the orientation of our dancer in our deep learning system.

7.3.2 Solution

Implement basic quaternion operations. We need quaternion operations to process the quaternions from the IMU.

7.3.3 How It Works

Quaternions are the preferred mathematical representation of orientation. Propagating a quaternion requires fewer operations than propagating a transformation matrix and avoids singularities that occur with Euler angles. A quaternion has four elements, which corresponds to a unit vector a and angle of rotation ϕ about that vector. The first element is termed the ‘‘scalar component’’ s, and the next three elements are the ‘‘vector’’ components v. This notation is shown as follows [21]:

$$q = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix} = \begin{bmatrix} s \\ v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} \cos\frac{\phi}{2} \\ a_1 \sin\frac{\phi}{2} \\ a_2 \sin\frac{\phi}{2} \\ a_3 \sin\frac{\phi}{2} \end{bmatrix} \tag{7.13}$$

The ‘‘unit’’ quaternion which represents zero rotation from the initial coordinate frame has a unit scalar component and zero vector components. This is the same convention used on the Space Shuttle, although other conventions are possible.

$$q_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{7.14}$$

In order to transform a vector from one coordinate frame a to another b using a quaternion q_{ab}, the operation is

$$u_b = q_{ab}^T u_a q_{ab} \tag{7.15}$$

using quaternion multiplication with the vectors defined as quaternions with a scalar part equal to zero, or

$$x_a = \begin{bmatrix} 0 \\ x_a(1) \\ x_a(2) \\ x_a(3) \end{bmatrix} \tag{7.16}$$

For example, the quaternion

$$\begin{bmatrix} 0.7071 \\ 0.7071 \\ 0.0 \\ 0.0 \end{bmatrix} \tag{7.17}$$

represents a pure rotation about the x-axis. The first element is 0.7071 and equals the $\cos(90°/2)$. We cannot tell the direction of rotation from the first element. The second element is the 1 component of the unit vector, which in this case is

$$\begin{bmatrix} 1.0 \\ 0.0 \\ 0.0 \end{bmatrix} \tag{7.18}$$

times the argument $\sin(90°/2)$. Since the sign is positive, the rotation must be a positive 90° rotation.

We only need one routine that converts the quaternion, which comes from the IMU, into a transformation matrix for visualization. We do this because multiplying a 3× n array of vectors for the vertices of our 3D model by a matrix is much faster than transforming each vector with a quaternion.

QuaternionToMatrix.m

```
        2*(q(2)*q(4)+q(1)*q(3));...
        2*(q(2)*q(3)+q(1)*q(4)),...
        q(1)^2-q(2)^2+q(3)^2-q(4)^2,...
        2*(q(3)*q(4)-q(1)*q(2));...
        2*(q(2)*q(4)-q(1)*q(3)),...
        2*(q(3)*q(4)+q(1)*q(2)),...
        q(1)^2-q(2)^2-q(3)^2+q(4)^2];
```

Note that the diagonal terms have the same form. The off-diagonal terms also all have the same form.

7.4 Dancer Simulation

7.4.1 Problem

We want to simulate a dancer for readers who don't have access to the hardware.

7.4.2 Solution

We will write a right-hand side for the dancer based on the preceding equations and write a simulation with a control system.

7.4.3 How It Works

The right-hand side implements the dancer model. It includes an internal control ‘‘wheel” and a degree of freedom for the head movement. The default data structure is returned if you call it without arguments.

RHSDancer.m

```
%% RHSDANCER Implements dancer dynamics
% This is a model of dancer with one degree of translational freedom
% and 5 degrees of rotational freedom including the head and an
    internal
% rotational degree of freedom.
%% Form:
%   xDot = RHSDancer( x, ~, d )
%% Inputs
%   x       (11,1)      State vector [r;v;q;w;wHDot;wIDot]
%   t       (1,1)   Time (unused) (s)
%   d       (1,1)   Data structure for the simulation
%                   .torque   (3,1) External torque (Nm)
%                   .force    (1,1) External force (N)
%                   .inertia  (3,3) Body inertia (kg-m^2)
%                   .inertiaH (1,1) Head inertia (kg-m^2)
%                   .inertiaI (1,1) Inner inertia (kg-m^2)
%                   .mass     (1,1) Dancer mass (kg)
%
%% Outputs
%   xDot    (11,1)      d[r;v;q;w;wHDot;wIDot]/dt

function xDot = RHSDancer( ~, x, d )

% Default data structure
if( nargin < 1 )
   % Based on a 0.15 m radius, 1.4 m long cylinders
  inertia = diag([8.4479    8.4479    0.5625]);
  xDot    = struct('torque',[0;0;0],'force',0,'inertia',inertia,...
  'mass',50,'inertiaI',0.0033,'inertiaH',0.0292,'torqueH',0,'torqueI'
      ,0);
  return
end
```

The remainder mechanizes the equations given earlier. We add an additional equation for the integral of the z-axis rate. This makes the control system easier to write. We also include the gravitational acceleration in the force equation.

```
% Use local variables
v      = x(2);
q      = x(3:6);
w      = x(7:9);
wI     = x(10);
wH     = x(11);

% Unit vector
u      = [0;0;1];

% Gravity
g      = 9.806;

% Attitude kinematics (not mentioned in the text)
qDot    = QIToBDot( q, w );

% Rotational dynamics Equation 7.6
wDot    = d.inertia\(d.torque - Skew(w)*(d.inertia*w + d.inertiaI*(wI +
     w(3))...
       + d.inertiaH*(wH + w(3))) - u*(d.torqueI + d.torqueH));
wHDot = d.torqueH/d.inertiaH - wDot(3);
wIDot = d.torqueI/d.inertiaI - wDot(3);

% Translational dynamics
vDot  = d.force/d.mass - g;

% Assemble the state vector
xDot    = [v; vDot; qDot; wDot; wHDot; wIDot; w(3)];
```

The simulation setup gets default parameters from `RHSDancer`.

```
d        = RHSDancer;
n        = 800;
dT       = 0.01;
xP       = zeros(16,n);
x        = zeros(12,1);
x(3)     = 1;
g        = 9.806;
dancer   = 'Robot_1';
```

It then sets up the control system. We use a proportional derivative controller for z position and a rate damper to stop the pirouette. The position control is done by the foot muscles. The rate damping is our internal damper wheel.

```
% Control system for 2 pirouettes in 6 seconds
tPirouette  = 6;
```

```
zPointe     = 6*0.0254;
tPointe     = 0.1;
kP          = tPointe/dT;
omega       = 4*pi/tPirouette;
torquePulse = d.inertia(3,3)*omega/tPointe;
tFriction   = 0.1;
a           = 2*zPointe/tPointe^2 + g;
kForce      = 1000;
tau         = 0.5;
thetaStop   = 4*pi - pi/4;
kTorque     = 200;
state       = zeros(10,n);
```

The simulation loop calls the right-hand side and the control system. We call `RHSDancer.m` to get the linear acceleration.

```
%% Simulate
for k = 1:n
  d.torqueH = 0;
  d.torqueI = 0;

  % Get the data for use in the neural network
  xDot = RHSDancer(0,x,d);

  state(:,k) = [x(7:9);0;0;xDot(2);x(3:6)];

  % Control
  if( k < kP )
    d.force  = d.mass*a;
    d.torque = [0;0;torquePulse];
  else
    d.force  = kForce*(zPointe-x(1) -x(2)/tau)+ d.mass*g;
    d.torque = [0;0;-tFriction];
  end

  if( x(12) > thetaStop )
    d.torqueI = kTorque*x(9);
  end

  xP(:,k)   = [x;d.force;d.torque(3);d.torqueH;d.torqueI];
  x         = RungeKutta(@RHSDancer,0,x,dT,d);
end
```

The control system includes a torque and force pulse to get the pirouette going.

```
  % Control
  if( k < kP )
    d.force  = d.mass*a;
    d.torque = [0;0;torquePulse];
  else
    d.force  = kForce*(zPointe-x(1) -x(2)/tau)+ d.mass*g;
```

```
7       d.torque = [0;0;-tFriction];
8     end
```

The remainder of the script plots the results and outputs the data, which would have come from the IMU, into a file.

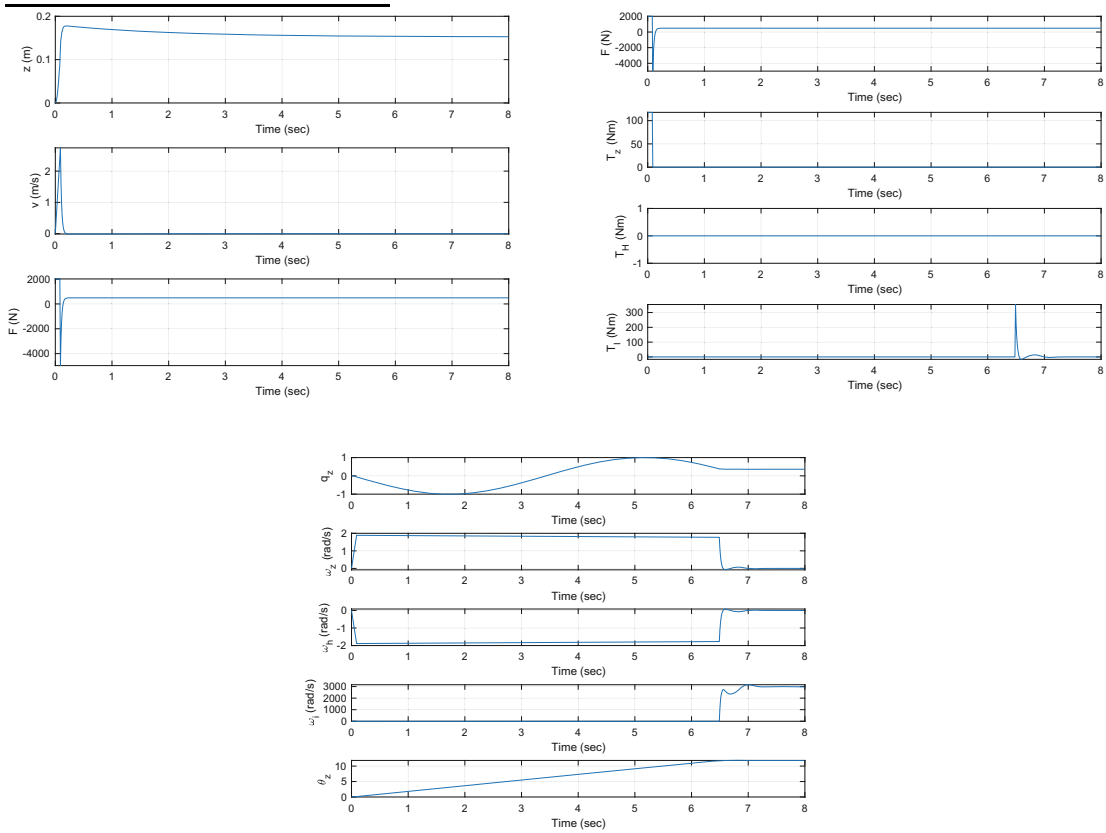
Simulation results for a double pirouette are shown in Figure 7.5. We stop the turn at 6.5 seconds, hence the pulse.

You can create different dancers by varying the mass properties and the control parameters.

```
zPointe      = 6*0.0254;
tPointe      = 0.1;
tFriction    = 0.1;
kForce       = 1000;
tau          = 0.5;
kTorque      = 200;
```

We didn't implement spotting control (looking at the audience as much as possible during the turn). It would rotate the head so that it faces forward whenever the head was within 90

Figure 7.5: Simulation of a double pirouette.



degrees or so of front. We'd need to add a head angle for that purpose to the right-hand side, much like we added the z-axis angle.

7.5 Real-Time Plotting

7.5.1 Problem

We want to display data from the IMU in real time. This will allow us to monitor the pirouettes.

7.5.2 Solution

Use `plot` with `drawnow` to implement multiple figures of plots.

7.5.3 How It Works

The main function is a switch statement with two cases. The function also has a built-in demo. The first case, `initialize`, initializes the plot figures. It stores everything in a data structure that is returned on each function call. This is one way for a function to have a memory. We return the data structure from each subfunction.

GUIPlots.m

```
27 switch( lower(action) )
28   case 'initialize'
29     g = Initialize( g );
30
31   case 'update'
32     g = Update( g, y, t );
33
34 end
```

The first case, `initialize`, initializes the figure window.

```
35 %% GUIPlots>Initialize
36 function g = Initialize( g )
37
38 lY = length(g.yLabel);
39
40 % Create tLim if it does not exist
41 if( ~isfield(g, 'tLim' ) )
42   g.tLim = [0 1];
43 end
44
45 g.tWidth = g.tLim(2) - g.tLim(1);
46
47 % Create yLim if it does not exist
48 if( ~isfield( g, 'yLim' ) )
49   g.yLim = [-ones(lY,1), ones(lY,1)];
50 end
51
52 % Create the plots
```

```
lP = length(g.yLabel);
y  = g.pos(2); % The starting y position
for k = 1:lP
  g.h(k) = subplot(lP,1,k);
  set(g.h(k),'position',[g.pos(1) y g.pos(3) g.pos(4)]);
  y = y - 1.4*g.pos(4);
  g.hPlot(k) = plot(0,0);
  g.hAxes(k) = gca;
  g.yWidth(k) = (g.yLim(k,2) - g.yLim(k,1))/2;
  set(g.hAxes(k),'nextplot','add','xlim',g.tLim);
  ylabel( char(g.yLabel{k}) )
  grid on
end
xlabel( g.tLabel );
```

The second case, update, updates the data displayed in the plot. It leaves the existing figures, subplots, and labels in place and just updates the plots of the line segments with new data. It can change the size of the axes as needed. The function adds a line segment for each new data point. This way no storage is needed external to the plot. It reads `xdata` and `ydata` and appends the new data to those arrays.

```
function g = Update( g, y, t )

% See if the time limits have been exceeded
if( t > g.tLim(2) )
  g.tLim(2)  = g.tWidth + g.tLim(2);
  updateAxes = true;
else
  updateAxes = false;
end

lP = length(g.yLabel);
for k = 1:lP
  subplot(g.h(k));
  yD = get(g.hPlot(k),'ydata');
  xD = get(g.hPlot(k),'xdata');
  if( updateAxes )
    set( gca, 'xLim', g.tLim );
    set( g.hPlot(k), 'xdata',[xD t],'ydata',[yD y(k)]);
  else
    set( g.hPlot(k), 'xdata',[xD t],'ydata',[yD y(k)] );
  end

end
```

The built-in demo plots six numbers. It updates the axes in time once. It sets up a figure window with six plots. You need to create the figure and save the figure handle before calling `GUIPlots`.

```
g.hFig  = NewFig('State');
```

The pause in the demo just slows down plotting so that you can see the updates. The height (the last number in `g.pos`) is the height of each plot. If you happen to set the locations of the plots out of the figure window, you will get a MATLAB error. `g.tLim` gives the initial time limits in second. The upper limit will expand as data is entered.

```
function Demo

g.yLabel = {'x' 'y' 'z' 'x_1' 'y_1' 'z_1'};
g.tLabel = 'Time (sec)';
g.tLim   = [0 100];
g.pos    = [0.100    0.88    0.8    0.10];
g.width  = 1;
g.color  = 'b';

g.hFig  = NewFig('State');
set(g.hFig, 'NumberTitle','off' );

g        = GUIPlots( 'initialize', [], [], g );

for k = 1:200
  y = 0.1*[cos((k/100))-0.05;sin(k/100)];
  g = GUIPlots( 'update', [y;y.^2;2*y], k, g );
  pause(0.1)
end

g        = GUIPlots( 'initialize', [], [], g );

for k = 1:200
  y = 0.1*[cos((k/100))-0.05;sin(k/100)];
  g = GUIPlots( 'update', [y;y.^2;2*y], k, g );
  pause(0.1)
end
```

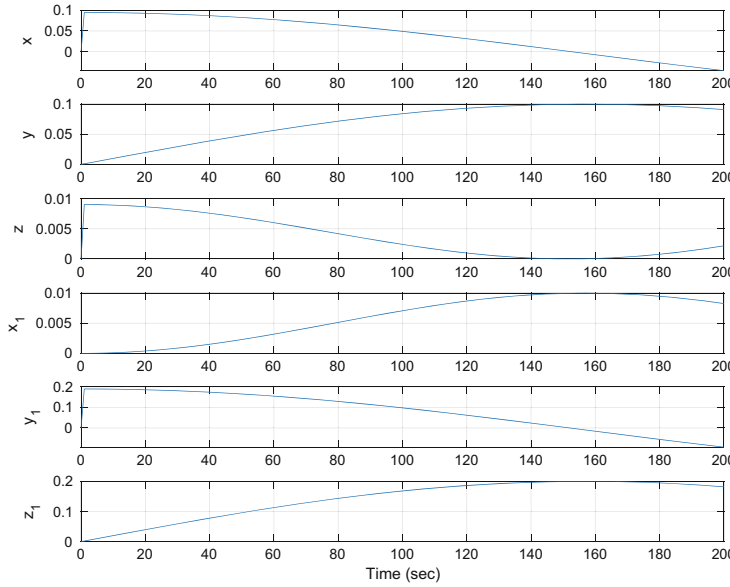
Figure 7.6 shows the real-time plots at the end of the demo.

7.6 Quaternion Display

7.6.1 Problem

We want to display the dancer's orientation in real time.

Figure 7.6: Real-time plots.



7.6.2 Solution

Use `patch` to draw an OBJ model in a three-dimensional plot. The figure is easier to understand than the four quaternion elements. Our solution can handle 3-axis rotation although typically we will only see single-axis rotation.

7.6.3 How It Works

We start with our `Ballerina.obj` file. It only has vertices and faces. A 3D drawing consists of a set of vertices. Each vertex is a point in space. The vertices are organized into faces. Each face is a triangle. Triangles are used for 3D drawings because they always form a plane. 3D processing hardware is designed to work with triangles, so this also gives the fastest results. The obj files for our software can only contain triangles. Each face can have only three vertices. Generally, obj files can have any size polynomials, that is, faces with more than three points. Most sources of obj files can provide tessellation services to convert polygons with more than three vertices into triangles. `LoadOBJ.m` will not draw models with anything other than triangular faces.

The main part of the function uses a case statement to handle the three actions. The first action just returns the defaults, which is the name of the default obj file. The second reads in the file and initializes the patches. The third updates the patches. `patch` is the MATLAB name for a set of triangles. The function can be passed a figure handle. A figure handle tells it into which figure the 3D model should be drawn. This allows it to be used as part of a GUI, as will be shown in the next section.

QuaternionVisualization.m

```
function m = QuaternionVisualization( action, x, f )

persistent p

% Demo
if( nargin < 1 )
  Demo
  return
end

switch( lower(action) )
    case 'defaults'
        m = Defaults;

    case 'initialize'
        if( nargin < 2 )
            d   = Defaults;
        else
            d   = x;
        end

        if( nargin < 3 )
          f = [];
        end

        p = Initialize( d, f );

    case 'update'
        if( nargout == 1 )
            m = Update( p, x );
        else
            Update( p, x );
        end
end
```

`Initialize` loads the obj file. It creates a figure and saves the object data structure. It sets shading to interpolated and lighting to Gouraud. Gouraud is a type of lighting model named after its inventor. It then creates the patches and sets up the axis system. We save handles to all the patches for updating later. We also place a light.

```
function p = Initialize( file, f )

if( isempty(f) )
  p.fig = NewFigure( 'Quaternion' );
else
  p.fig = f;
end

g     = LoadOBJ( file );
p.g   = g;
```

```
63
64  shading interp
65  lighting gouraud
66
67  c = [0.3 0.3 0.3];
68
69  for k = 1:length(g.component)
70    p.model(k)    = patch('vertices', g.component(k).v, 'faces', g.
          component(k).f, 'facecolor',c,'edgecolor',c,'ambient',1,'
          edgealpha',0 );
71  end
72
73  xlabel('x');
74  ylabel('y');
75  zlabel('z');
76
77  grid
78  rotate3d on
79  set(gca,'DataAspectRatio',[1 1 1],'DataAspectRatioMode','manual')
80
81  light('position',10*[1 1 1])
82
83  view([1 1 1])
```

In `Update` we convert the quaternion to a matrix, because it is faster to matrix multiply all the vertices with one matrix multiplication. The vertices are n by 3 so we transpose before the matrix multiplication. We use the patch handles to update the vertices. The two options at the end are to create movie frames or just update the drawing.

```
84  function m = Update( p, q )
85
86  s = QuaternionToMatrix( q );
87
88  for k = 1:length(p.model)
89    v = (s*p.g.component(k).v')';
90    set(p.model(k),'vertices',v);
91  end
92
93  if( nargout > 0 )
94          m = getframe;
95  else
96          drawnow;
97  end
```

This is the built-in demo. We vary the 1 and 4 elements of the quaternion to get rotation about the z-axis.

```
109 function Demo
110
111 QuaternionVisualization( 'initialize', 'Ballerina.obj' );
```

Figure 7.7: Dancer orientation. The obj file is by the artist loft_22 and is available from TurboSquid.

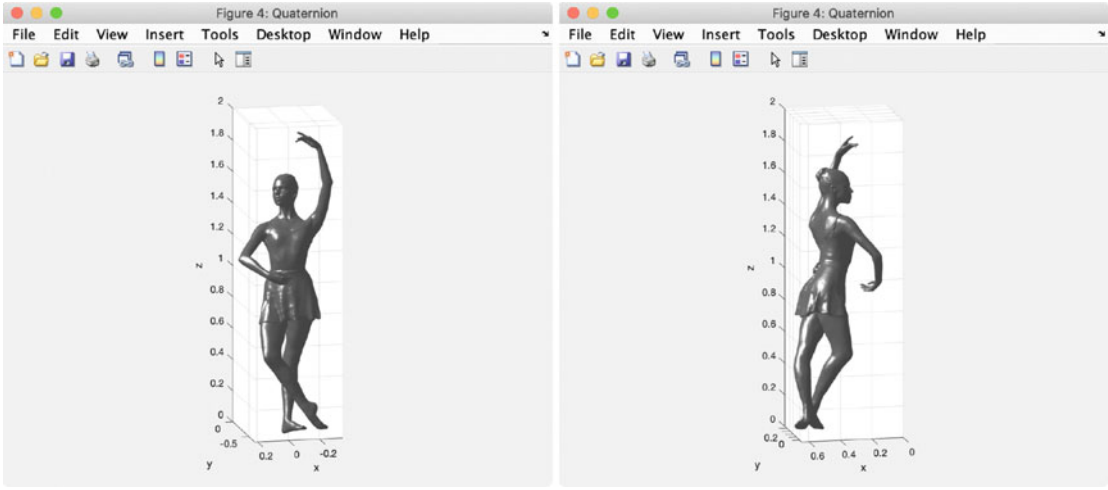


Figure 7.7 shows two orientations of the dancer during the demo. The demo produces an animation of the dancer rotating about the z-axis. The rotation is slow because of the number of vertices. The figure is not articulated so the entire figure is rotated as a rigid body. MATLAB doesn't make it easy to texture map so we don't bother. In any case, the purpose of this function is just to show orientation so it doesn't matter.

7.7 Data Acquisition GUI

7.7.1 Problem

Build a data acquisition GUI to display the real-time data and output it into training sets.

7.7.2 Solution

Integrate all the preceding recipes into a GUI.

7.7.3 How It Works

We aren't going to use MATLAB's Guide to build our GUI. We will hand code it, which will give you a better idea of how a GUI really works.

We will use nested functions for the GUI. The inner functions have access to all variables in the outer functions. This also makes using callbacks easy as shown in the following code snippet.

```
function DancerGUI( file )
function DrawGUI(h)
 uicontrol( h.fig,'callback',@SetValue);
   function SetValue(hObject, ~, ~ )
```

```
    % do something
   end
 end
 end
```

A callback is a function called by a `uicontrol` when the user interacts with the control. When you first open the GUI, it will look for the bluetooth device. This can take a while.

Everything in `DrawGUI` has access to variables in `DancerGUI`. The GUI is shown in Figure 7.8. The 3D orientation display is in the upper left corner. Real-time plots are on the right. Buttons are on the lower left, and the movie window is on the right.

The upper left picture shows the dancer's orientation. The plots on the right show angular rates and accelerations from the IMU. From top to bottom of the buttons

1. Turn the 3D on/off. The default model is big, so unless you add your own model with fewer vertices, it should be set to off.
2. The text box to its right is the name of the file. The GUI will add a number to the right of the name for each run.
3. Save saves the current data to a file.
4. Calibrate sets the default orientation and sets the gyro rates and accelerations to whatever it is reading when you hit the button. The dancer should be still when you hit calibrate. It will automatically compute the gravitational acceleration and subtract it during the test.
5. Quit closes the GUI.
6. Clear data clears out all the internal data storage.
7. Start/Stop starts and stops the GUI.

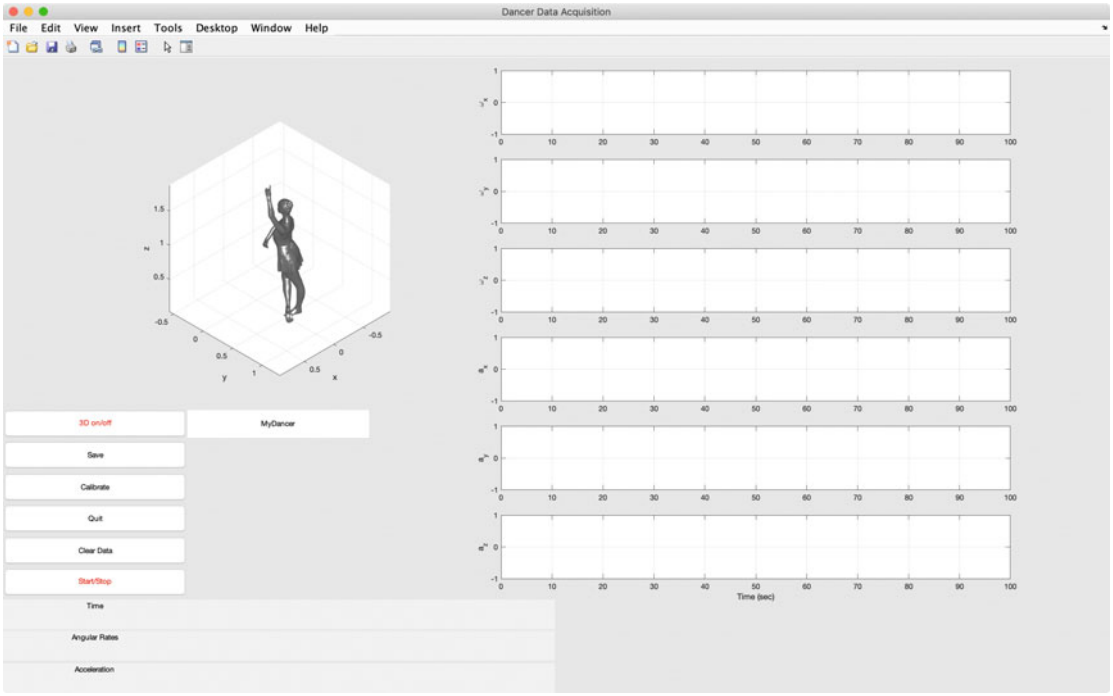
The remaining three lines display the time, the angular rate vector, and the acceleration vector as numbers. This is the same data that is plotted.

The first part creates the figure and draws the GUI. It initializes all the fields for `GUIPlots`. It reads in a default picture for the movie window as a placeholder.

DancerGUI.m

```
16  function DancerGUI( file )
17
18  % Demo
19  if( nargin < 1 )
20    DancerGUI('Ballerina.obj');
21    return
22  end
23
24  % Storage of data need by the deep learning system
25  kStore      = 1;
26  accelStore  = zeros(3,1000);
27  gyroStore   = zeros(3,1000);
```

Figure 7.8: Data acquisition GUI.




```
28  quatStore    = zeros(4,1000);
29  timeStore    = zeros(1,1000);
30  time         = 0;
31  on3D         = false;
32  quitNow      = false;
33
34  sZ = get(0,'ScreenSize') + [99 99 -200 -200];
35
36  h.fig = figure('name','Dancer Data Acquisition','position',sZ,'units','
        pixels',...
37    'NumberTitle','off','tag','DancerGUI','color',[0.9 0.9 0.9]);
38
39  % Plot display
40  gPlot.yLabel  = {'\omega_x' '\omega_y' '\omega_z' 'a_x' 'a_y' 'a_z'};
41  gPlot.tLabel  = 'Time (sec)';
42  gPlot.tLim    = [0 100];
43  gPlot.pos     = [0.45    0.88    0.46    0.1];
44  gPlot.color   = 'b';
45  gPlot.width   = 1;
46
47  % Calibration
48  q0            = [1;0;0;0];
49  a0            = [0;0;0];
50
```

```
dIMU.accel    = a0;
dIMU.quat     = q0;

% Initialize the GUI
DrawGUI;
```

The notation

```
'\omega_x'
```

is latex format. This will generate ω_x.

The next part tries to find Bluetooth. It first sees if Bluetooth is available at all. It then enumerates all Bluetooth devices. It looks through the list to find our IMU.

```
if( ~isempty(btInfo.RemoteIDs) )
   % Display the information about the first device discovered
  btInfo.RemoteNames(1)
  btInfo.RemoteIDs(1)
  for iB = length(btInfo.RemoteIDs)
    if( strcmp(btInfo.RemoteNames(iB),'LPMSB2-4B31D6') )
      break;
    end
  end
  b       = Bluetooth(btInfo.RemoteIDs{iB}, 1);
  fopen(b); % No output allowed for some reason
  noIMU   = false;
  a       = fread(b,91);
  dIMU    = DataFromIMU( a );
else
  warndlg('The IMU is not available.', 'Hardware Configuration')
  noIMU   = true;
end
```

The following is the run loop. If no IMU is present, it synthesizes data. If the IMU is found, the GUI reads data from the IMU in 91 byte chunks. The `uiwait` is to wait until the user hits the start button. When used for testing, the IMU should be on the dancer. The dancer should remain still when the start button is pushed. It will then calibrate the IMU. Calibration fixes the quaternion reference and removes the gravitational acceleration. You can also hit the calibration button at any time.

```
% Wait for user input
uiwait;
% The run loop
time  = 0;
tic
while(1)
  if( noIMU )
    omegaZ      = 2*pi;
    dT          = toc;
```

```
    time        = time + dT;
    tic
    a           = omegaZ*time;
    q           = [cos(a);0;0;sin(a)];
    accel       = [0;0;sin(a)];
    omega       = [0;0;omegaZ];
  else
    % Query the bluetooth device
    a       = fread(b,91);
    pause(0.1); % needed so not to overload the bluetooth device

    dT      = toc;
    time    = time + dT;
    tic

    % Get a data structure
    if( length(a) > 1 )
      dIMU    = DataFromIMU( a );
    end
    accel   = dIMU.accel - a0;
    omega   = dIMU.gyro;
    q       = QuaternionMultiplication(q0,dIMU.quat);

    timeStore(1,kStore)   = time;
    accelStore(:,kStore)  = accel;
    gyroStore(:,kStore)   = omega;
    quatStore(:,kStore)   = q;
    kStore = kStore + 1;
  end
```

```
      dIMU    = DataFromIMU( a );
    end
    accel   = dIMU.accel - a0;
    omega   = dIMU.gyro;
```

This code closes the GUI and displays the IMU data.

```
  if( quitNow )
    close( h.fig )
    return
  else
    if( on3D )
      QuaternionVisualization( 'update', q );
    end
    set(h.text(1),'string',sprintf('[%5.2f;%5.2f;%5.2f] m/s^2',accel));
    set(h.text(2),'string',sprintf('[%5.2f;%5.2f;%5.2f] rad/s',omega));
    set(h.text(3),'string',datestr(now));
    gPlot = GUIPlots( 'update', [omega;accel], time, gPlot );
  end
end
```

The drawing code uses `uicontrol` to create all the buttons. `GUIPlots` and `QuaternionVisualization` are also initialized. The `uicontrol` that require an action have callbacks.

```
  if( quitNow )
    close( h.fig )
    return
  else
    if( on3D )
      QuaternionVisualization( 'update', q );
    end
    set(h.text(1),'string',sprintf('[%5.2f;%5.2f;%5.2f] m/s^2',accel));
    set(h.text(2),'string',sprintf('[%5.2f;%5.2f;%5.2f] rad/s',omega));
    set(h.text(3),'string',datestr(now));
    gPlot = GUIPlots( 'update', [omega;accel], time, gPlot );
  end
end

%% DancerGUI>DrawButtons
function DrawGUI

  % Plots
  gPlot = GUIPlots( 'initialize', [], [], gPlot );

  % Quaternion display
  subplot('position',[0.05 0.5 0.4 0.4],'DataAspectRatio',[1 1 1],'
      PlotBoxAspectRatio',[1 1 1] );
  QuaternionVisualization( 'initialize', file, h.fig );

  % Buttons
  f   = {'Acceleration', 'Angular Rates' 'Time'};
  n   = length(f);
  p   = get(h.fig,'position');
  dY  = p(4)/20;
  yH  = p(4)/21;
  y   = 0.5;
  x   = 0.15;
  wX  = p(3)/6;

  % Create pushbuttons and defaults
  for k = 1:n
    h.pushbutton(k) = uicontrol( h.fig,'style','text','string',f{k},'
        position',[x    y wX yH]);
    h.text(k)       = uicontrol( h.fig,'style','text','string','',  '
        position',[x+wX y 2*wX yH]);
    y               = y + dY;
  end
```

`uicontrol` takes parameter pairs, except for the first argument that can be a figure handle. There are lot of parameter pairs. The easiest way to explore them is to type

```
h = uicontrol;
get(h)
```

All types of `uicontrol` that handle user interaction have ‘‘callbacks” that are functions that do something when the button is pushed or menu item is selected. We have five `uicontrol` with callbacks. The first uses `uiwait` and `uiresume` to start and stop data collection.

```
3    % Start/Stop button callback
4    function StartStop(hObject, ~, ~ )
5      if( hObject.Value )
6        uiresume;
7      else
8        SaveFile;
9        uiwait
10     end
11   end
```

The second uses `questdlg` to ask if you want to save the data that has been stored in the GUI. This produces the modal dialog shown in Figure 7.9.

```
12   % Quit button callback
13   function Quit(~, ~, ~ )
14     button = questdlg('Save Data?','Exit Dialog','Yes','No','No');
15     switch button
16       case 'Yes'
17         % Save data
18       case 'No'
19     end
20     quitNow = true;
21     uiresume
22   end
```

Figure 7.9: Modal dialog.

The third, `Clear`, clears the data storage arrays. It resets the quaternion to a unit quaternion.

```
% Clear button callback
function Clear(~, ~, ~ )
  kStore      = 1;
  accelStore  = zeros(3,1000);
  gyroStore   = zeros(3,1000);
  quatStore   = zeros(4,1000);
  timeStore   = zeros(1,1000);
  time        = 0;
end
```

The fourth, `calibrate`, runs the calibration procedure.

```
% Calibrate button callback
function Calibrate(~, ~, ~ )
  a       = fread(b,91);
  dIMU    = DataFromIMU( a );
  a0      = dIMU.accel;
  q0      = dIMU.quat;
  QuaternionVisualization( 'update', q0 )
end
```

The fifth, `SaveFile`, saves the recorded data into a mat file for use by the deep learning algorithm.

```
% Save button call back
function SaveFile(~,~,~)
  cd TestData
  fileName = get(h.matFile,'string');
  s = dir;
  n = length(s);
  fNames = cell(1,n-2);
  for kF = 3:n
    fNames{kF-2} = s(kF).name(1:end-4);
  end
  j = contains(fNames,fileName);
  m = 0;
  if( ~isempty(j) )
    for kF = 1:length(j)
      if( j(kF))
        f = fNames{kF};
        i = strfind(f,'_');
        m = str2double(f(i+1:end));
      end
    end
  end
```

We make it easier for the user to save files by reading the directory and adding a number to the end of the dancer filename that is one greater than the last filename number.

7.8 Making the IMU Belt

7.8.1 Problem

We need to attach the IMU to our dancer.

7.8.2 Solution

We use the arm strap that is available from the manufacturer. We buy an elastic belt and make one that fits around the dancer's waist.

7.8.3 How It Works

Yes, software engineers need to sew. Figure 7.10 shows the process. The two products used to make the data acquisition belt are

1. LPMS-B2 Holder (available from Life Performance Research)
2. Men's No Show Elastic Stretch Belt Invisible Casual Web Belt Quick Release Flat Plastic Buckle (available from Amazon)

Remove the holder from the LPMS-B2 Holder. Cut the belt at the buckle and slide the holder onto the belt. Sew the belt at the buckle.

The sensor on a dancer is shown in Figure 7.11. We had the dancer stand near the laptop during startup. We didn't have any range problems during the experiments. We didn't try it with across the floor movement as one would have during grande allegro.

Figure 7.10: Elastic belt manufacturing. We use the two items on the left to make the one on the right.

Figure 7.11: Dancer with the sensor belt. The blue light means it is collecting data.

7.9 Testing the System

7.9.1 Problem

We want to test the data acquisition system. This will find any problems with the data acquisition process.

7.9.2 Solution

Have a dancer do changements, which are small jumps changing the foot position on landing.

7.9.3 How It Works

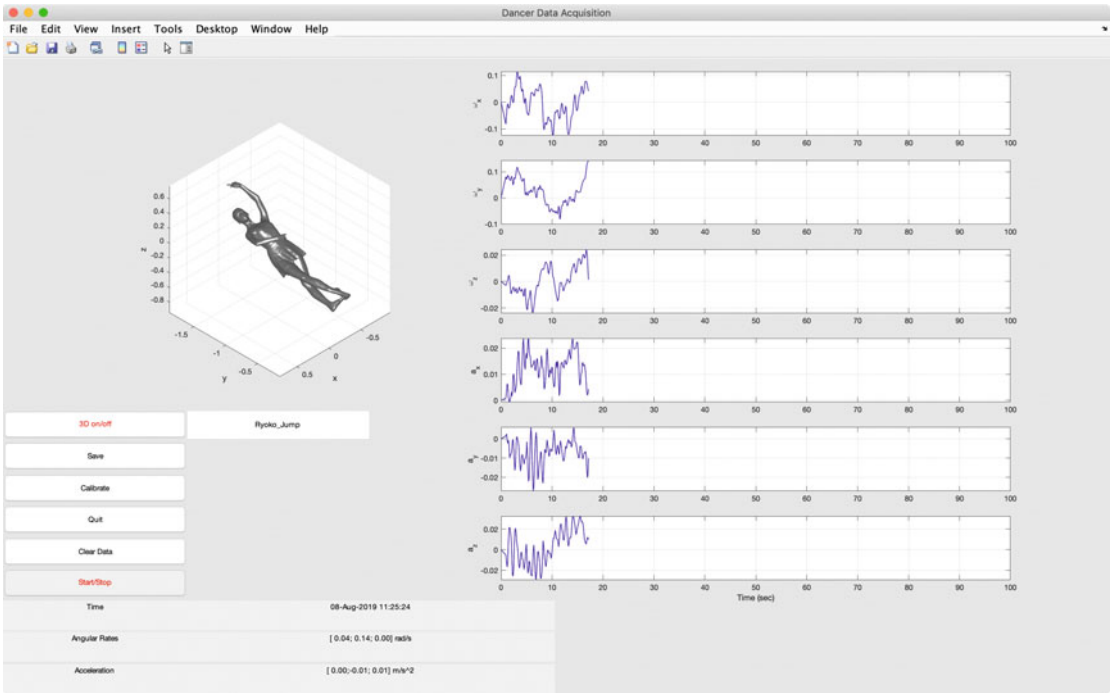
The dancer puts on the sensor belt, we push the calibrate button, then she does a series of changements. She stands about 2 m from your computer to make acquisition easier. The dancer will do small jumps, known as changements. A changement is a small jump where the feet change positions starting from fifth position. If the right foot is in fifth position front at the start, it is in the back at the finish. Photos are shown in Figure 7.12.

The time scale is a bit long. You can see that the calibration does not lead us to a natural orientation in the axis system in the GUI. It doesn't matter from a data collection point of view but is an improvement we should make in the future. The changement is shown in Figure 7.13. The dancer is still at the beginning and end.

The interface to the bluetooth device doesn't do any checking or stream control. Some bluetooth data collection errors occur from time to time. Typically, they happen after 40 seconds of data collection.

Figure 7.12: Dancer doing a changement. Notice the feet when she is preparing to jump. The second image shows her feet halfway through the jump.

Figure 7.13: Data collected during the changement.



```
index exceeds the number of array elements (0).
Error in instrhwinfo>bluetoothCombinedDevices (line 976)
        uniqueBTName = allBTName(uniqueRowOrder);
Error in instrhwinfo (line 206)
                    tempOut = bluetoothCombinedDevices(tempOut);
Error in DataAcquisition (line 13)
btInfo  = instrhwinfo('Bluetooth');
```

If this happens, turn the device on and off. Restart MATLAB if that doesn't work. Another bluetooth error is

```
ans =
  1x1 cell array
    {'LPMSB2-4B31D6'}
ans =
  1x1 cell array
    {'btspp://00043E4B31D6'}
Error using Bluetooth (line 104)
Cannot Create: Java exception occurred:
java.lang.NullPointerException
        at com.mathworks.toolbox.instrument.BluetoothDiscovery.
            searchDevice(BluetoothDiscovery.java:395)
        at com.mathworks.toolbox.instrument.BluetoothDiscovery.
            discoverServices(BluetoothDiscovery.java:425)
        at com.mathworks.toolbox.instrument.BluetoothDiscovery.
            hardwareInfo(BluetoothDiscovery.java:343)
        at com.mathworks.toolbox.instrument.Bluetooth.<init>(Bluetooth.
            java:205).
```

This is a MATLAB error and requires restarting MATLAB. It doesn't happen very often. We ran the entire data collection with four dancers doing ten pirouettes each without ever experiencing the problem.

7.10 Classifying the Pirouette

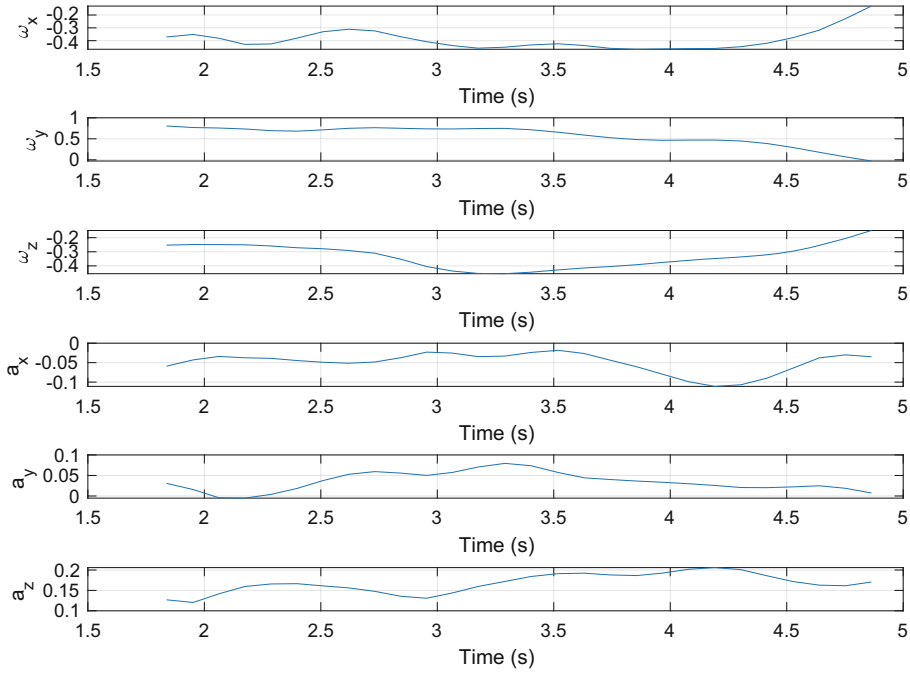
7.10.1 Problem

We want to classify the pirouettes of our four dancers.

7.10.2 Solution

Create an LSTM that classifies pirouettes according to dancer. The four labels are the dancer names.

Figure 7.14: A pirouette. Angular rate and linear acceleration are shown.



7.10.3 How It Works

The script takes one file and displays it. Figure 7.14 shows a double pirouette. A turn takes only a few seconds.

```
dancer = {'Ryoko' 'Shaye', 'Emily', 'Matanya'};

%% Show one dancer's data
cd TestData
s    = load('Ryoko_10.mat');
yL   = {'\omega_x' '\omega_y' '\omega_z' 'a_x' 'a_y' 'a_z'};
PlotSet(s.time,s.state(1:6,:),'x label','Time (s)','y label',yL,'figure
     title',dancer{1});
```

We load in the data and limit the range to 6 seconds. Sometimes the IMU would run longer due to human error. We also remove sets that are bad.

DancerNN.m

```
%% Load in and process the data
n = 40;
% Get the data and remove bad data sets
i = 0;
for k = 1:length(dancer)
  for j = 1:10
    s    = load(sprintf('%s_%d.mat',dancer{k},j));
    cS   = size(s.state,2);
```

```
    if( cS > 7 )
      i       = i + 1;
      d{i,1}  = s.state; %#ok<*SAGROW>
      t{i,1}  = s.time;
      c(i,1)  = k;
    end
  end
end

fprintf('%d remaining data sets out of %d total.\n',i,n)

for k = 1:4
  j = length(find(c==k));
  fprintf('%7s data sets %d\n',dancer{k},j)
end

n = i;

cd ..

% Limit the range to 6 seconds
tRange = 6;
for i = 1:n
  j = find(t{i} - t{i,1} > tRange );
  if( ~isempty(j) )
    d{i}(:,j(1)+1:end)= [];
  end
end
```

We then train the neural network. We use a bidirectional LSTM to classify the sequences. There are ten features, four quaternion measurements, three rate gyro, and three accelerometer. The four quaternion numbers are coupled through the relationship

$$1 = q_1^2 + q_2^2 + q_3^2 + q_4^2 \tag{7.19}$$

However, this should not impact the learning accuracy aside from slowing down the learning.

We then load in and process the data. Some data sets didn't have any data and need to be removed. We also limit the range to 6 seconds since sometimes the data collection did not stop after the pirouette ended.

```
%% Load in and process the data
n = 40;
% Get the data and remove bad data sets
i = 0;
for k = 1:length(dancer)
  for j = 1:10
    s   = load(sprintf('%s_%d.mat',dancer{k},j));
    cS  = size(s.state,2);
    if( cS > 7 )
      i       = i + 1;
```

```
        d{i,1}  = s.state; %#ok<*SAGROW>
        t{i,1}  = s.time;
        c(i,1)  = k;
      end
    end
end

fprintf('%d remaining data sets out of %d total.\n',i,n)

for k = 1:4
  j = length(find(c==k));
  fprintf('%7s data sets %d\n',dancer{k},j)
end

n = i;

cd ..

% Limit the range to 6 seconds
tRange = 6;
for i = 1:n
  j = find(t{i} - t{i,1} > tRange );
  if( ~isempty(j) )
    d{i}(:,j(1)+1:end)= [];
  end
end

%% Set up the network
numFeatures = 10; % 4 quaternion, 3 rate gyros, 3 accelerometers
numHiddenUnits = 400;
numClasses = 4; % Four dancers

layers = [ ...
    sequenceInputLayer(numFeatures)
    bilstmLayer(numHiddenUnits,'OutputMode','last')
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];
disp(layers)

options = trainingOptions('adam', ...
    'MaxEpochs',60, ...
    'GradientThreshold',1, ...
    'Verbose',0, ...
    'Plots','training-progress');
```

We then train the neural network. We use a bidirectional LSTM to classify the sequences. This is a good choice because we have access to the full sequence. For a classifier using `bilstmLayer`, we must set the `'outputMode'` to `'last'`. This is followed by a fully connected layer, a Softmax for producing normalized maximums, and finally the classification layer.

```
%% Train the network
nTrain  = 30;
kTrain  = randperm(n,nTrain);
xTrain  = d(kTrain);
yTrain  = categorical(c(kTrain));
net     = trainNetwork(xTrain,yTrain,layers,options);

%% Test the network
kTest   = setdiff(1:n,kTrain);
xTest   = d(kTest);
yTest   = categorical(c(kTest));
yPred   = classify(net,xTest);

% Calculate the classification accuracy of the predictions.
acc         = sum(yPred == yTest)./numel(yTest);
disp('Accuracy')
disp(acc);
```

```
>> DancerNN
36 remaining data sets out of 40 total.
   Ryoko data sets 6
  Shaye data sets 10
  Emily data sets 10
Matanya data sets 10
 5x1 Layer array with layers:

     1   ''   Sequence Input          Sequence input with 10 dimensions
     2   ''   BiLSTM                  BiLSTM with 400 hidden units
     3   ''   Fully Connected         4 fully connected layer
     4   ''   Softmax                 softmax
     5   ''   Classification Output   crossentropyex
```

The training GUI is shown in Figure 7.15. It converges fairly well.

We test neural network against the unused data.

```
kTrain  = randperm(n,nTrain);
xTrain  = d(kTrain);
yTrain  = categorical(c(kTrain));
net     = trainNetwork(xTrain,yTrain,layers,options);

%% Test the network
kTest   = setdiff(1:n,kTrain);
xTest   = d(kTest);
yTest   = categorical(c(kTest));
yPred   = classify(net,xTest);
```

```
Accuracy
    0.8333
```

Figure 7.15: Neural net training.

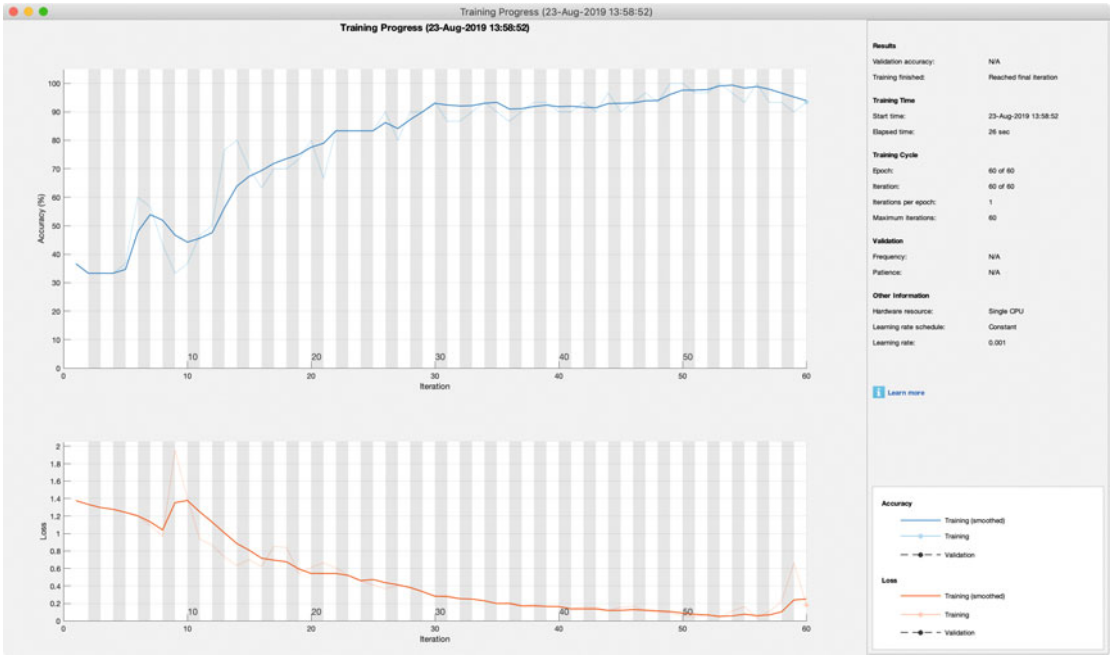


Table 7.3: Hardware.

Component	Supplier	Part Number	Price
IMU	LP-Research Inc.	LPMS-B2: 9-Axis Inertial Measurement Unit	$299.00
IMU Holder	LP-Research Inc.	LPMS-B2: Holder	$30.00
Belt	Amazon	Men's Elastic Stretch Belt Invisible Casual Trousers Webbing Belt Plastic Buckle Black Fits 24" to 42"	$10.99

The result, $> 80\%$, is pretty good considering the limited amount of data. Four Ryoko sets were lost due to errors in data collection. It is interesting that the deep learning network could distinguish the dancers' pirouettes. The data itself did not show any easy-to-spot differences. Calibration could have been done better to make the data more consistent between dancers. It would have been interesting to collect data on multiple days. Other experiments would be to classify pirouettes done in pointe shoes and without. We might also have had the dancers do different types of turns to see if the network could still identify the dancer.

7.11 Hardware Sources

Table 7.3 gives the hardware used in this chapter along with the prices (in US dollars) at the time of publication.

CHAPTER 8

Completing Sentences

8.1 Introduction

8.1.1 Sentence Completion

Completing sentences is a useful feature for text entry systems. Given a set of possible sentences, we want the system to predict a missing part of the sentence. We will use the Research Sentence Completion Challenge [31]. It is a database of 1040 sentences each of which has four imposter sentences and one correct sentence. Each imposter sentence differs from the correct sentence by one word in a fixed position. The deep learning system should identify the correct word in the sentence. Imposter words have similar occurrence statistics. The sentences were selected from Sherlock Holmes novels. The imposter words were generated using a language model trained using over five hundred nineteenth-century novels. Thirty alternative words for the correct word were produced. Human judges picked the four best imposter words from the 30 alternatives. The database can be downloaded from Google Drive [20].

The first question in the database and the five answers, including the four imposters, are given as follows.

```
I have it from the same source that you are both an orphan and a bachelor
 and are _____ alone in London.
a) crying  b)  instantaneously c) residing d) matched e) walking
```

(b) and (d) don't fit grammatically. (a) and (e) are incompatible with the beginning in which the speaker is recounting general information about the subject's state. (c) makes the most sense. If after ''are" we had ''often seen," then (a) and (e) would be possibilities and (c) would no longer make sense. You would need additional information to determine if (a) or (e) were correct.

M. Paluszek and S. Thomas, *Practical MATLAB Deep Learning*,
https://doi.org/10.1007/978-1-4842-5124-9_8

8.1.2 Grammar

Grammar is important in interpreting sentences. The structure of a language, that is, its grammar, is very important. Since all of our readers don't speak English as their primary language, we'll give some examples in other languages.

In Russian, the word order is not fixed. You can always figure out the words, and whether they are adjectives, verbs, nouns, and so forth from the declension and conjugation, but the word order is important as it determines the emphasis. For example, to say, ''I am an engineer" in Russian:

Я инженер

We could reverse the order:

инженер Я

which would mean the emphasis is on ''engineer" not ''I". While it is easy to know that the sentence is stating that ''I am an engineer," we don't necessarily know how the speaker feels about it. This may not be important in rote translation but certainly makes a difference in literature.

Japanese is known as a subject-object-verb language. In Japanese the verb is at the end of the sentence. Japanese also makes use of particles to denote word function such as subject or object. For the sentence completion problem, the particle would denote the function of the word. The rest of the sentence would determine what the word may mean. Here are some particles:

は''wa/ha" indicates the topic, which could be the object or subject.
を''wo/o" indicates the object.
が''ga" indicates the subject.

For example, in Japanese

私はエンジニアです

or ''watashi wa enjinia desu"

means, ''I am an engineer." は is the topic marker pointing to ''I". です is the verb. We'd need other sentences to predict the 私, ''I", or ''engineer".

Japanese also has the feature where everything, except the verb, can be omitted.

いただきます

or ''I ta da ki ma su." This means ''I will eat" whatever is given. You need to know the context or have other sentences to understand what is meant by the sentence.

In addition, in Japanese, many different Kanji, or symbols, can mean approximately the same thing, but the emphasis will be different. Other Kanji have different meanings depending on context. Japanese also does not have any spaces between words. You just have to know when a kana character, like は, is part of the preceding Kanji.

8.1.3 Sentence Completion by Pattern Recognition

Our approach is sentence completion by pattern recognition. Given a database of your sentences, the pattern recognition algorithm should be able to recognize the patterns you use and find errors. Also, in most languages, dialogs between people use far fewer words and simpler structures than the written languages. You will notice this if you watch a movie in a foreign language for which you have a passable knowledge. You can recognize a lot more than you would expect. Russian is an extreme in this regard; it is very hard to build vocabulary from reading because the language is so complex. Many Russian teachers teach the root system so that you can guess word meaning without constantly referring to a dictionary. Using word roots and sentence structure to guess words is a form of sentence completion. We'll leave that to our Russian readers.

8.1.4 Sentence Generation

As an aside, sentence completion leads to generative deep learning [12]. In generative deep learning, the neural network learns patterns and then can create new material. For example, a deep learning network might learn how a newspaper article is written and be able to generate new articles given basic facts the article is supposed to present. This is not a whole lot different than when writers are paid to write new books in a series such as *Tom Swift* or *Nancy Drew*. Presumably, the writer adds his or her personality to the story, but perhaps a reader, who just wants a page turner, wouldn't really care.

8.2 Generating a Database of Sentences

8.2.1 Problem

We want to create a set of sentences accessible from MATLAB.

8.2.2 Solution

Read in the sentences from the database. Write a function to read in tab-separated text.

8.2.3 How It Works

The database that we downloaded from Google Drive was an Excel `csv` file. We need to first open the file and save it as tab-delimited text. Once this is done, you are ready to read it into MATLAB. We do this for both `test_answer.csv` and `testing_data.csv`. We manually removed the first column `in test_answer.csv` in Excel because it was not needed. Only the `txt` files that we generated are needed in this book.

If you have the Statistics and Machine Learning Toolbox, you could use `tdfread`. We'll write the equivalent. There are three outputs shown in the header. They are the sentences, the range of characters where the word needed for completion fits, the five possible words and the answer.

We open the file using `f = fopen('testing_data.txt','r');`. This tells it that the file is a text file. We search for tabs and add the end of the line so that we can find the last

word. The second reads in the test answers and converts them from a character to a number. We removed all extraneous quotes from the text file with a text editor.

ReadDatabase.m

```
1
2  f = fopen('testing_data.txt','r');
3
4  % We know the size of the file simplifying the code.
5  u = zeros(1040,2);
6  a = zeros(1040,1);
7  s = strings(1040,1);
8  v = strings(1040,5);
9  t = sprintf('\t');
10 k = 1;
11
12 % Read in the sentences and words
13 while(~feof(f))
14   q     = fgetl(f); % This is one line of text
15   j     = [strfind(q,t) length(q)+1]; % This finds tabs that delimit words
16   s(k)  = convertCharsToStrings(q(j(1)+1:j(2)-1)); % Convert to strings
17   for i = 1:5
18     v(k,i) = convertCharsToStrings(q(j(i+1)+1:j(i+2)-1)); % Make strings
19   end
20   ul      = strfind(s(k),'_'); % Find the space where the answers go
21   u(k,:)  = [ul(1) ul(end)]; % Get the range of characters for the answer
22   k       = k + 1;
23 end
24
25 fclose(f);
26
27 % Read in the test answers
28 f = fopen('test_answer.txt','r');
29
30 k = 1;
31 while(~feof(f))
32   q             = fgetl(f);
33   a(k,1)        = double(q)-96;
34   k             = k + 1;
35 end
36
37 fclose(f);
```

If we run the function, we get the following outputs.

```
>> [s,u,v,a] = ReadDatabase;
>> s(1)
ans =
    "I have it from the same source that you are both an orphan and a
       bachelor and are _____ alone in London."
>> v(1,:)
ans =
  1x5 string array
```

```
    "crying"    "instantaneously"    "residing"    "matched"    "walking"
>> a(1)
ans =
     3
```

All outputs (except for the answer number) are strings. `convertCharsToStrings` does the conversion. Now that we have all of the data in MATLAB, we are ready to try and train the system to determine the best word for each sentence. As an intermediate step, we will convert the words to numbers.

8.3 Creating a Numeric Dictionary

8.3.1 Problem

We want to create a numeric dictionary to speed neural net training. This eliminates the need for string matching during the training process. Expressing a sentence as a numeric sequence as opposed to a sequence of character arrays (words) essentially gives us a more efficient way to represent the sentence. This will become useful later when we perform machine learning over a database of sentences to learn valid and invalid sequences.

8.3.2 Solution

Write a MATLAB function, `DistinctWords`, to search through text and find unique words.

8.3.3 How It Works

The function removes punctuation using `erase` in the following lines of code.

DistinctWords.m

```
% Remove punctuation
w = erase(w,';');
w = erase(w,',');
w = erase(w,'.');
```

It then uses `split` to break up the string and finds unique strings using `unique`.

```
% Find unique words
s = split(w)';
d = unique(s);
```

This is the built-in demo. It finds 38 unique words.

```
>> DistinctWords
w =
    "No one knew it then, but she was being held under a type of house
        arrest while the tax authorities scoured
    the records of her long and lucrative career as an actress, a
        luminary of the red carpet, a face of luxury
     brands and a successful businesswoman."
```

```
d =
  1x38 string array
  Columns 1 through 12
    "No"    "one"    "knew"    "it"    "then"    "but"    "she"    "was"
           "being"    "held"    "under"    "type"
  Columns 13 through 22
    "house"    "arrest"    "while"    "tax"    "authorities"    "scoured"
            "records"    "her"    "long"    "lucrative"
  Columns 23 through 33
    "career"    "as"    "an"    "actress"    "luminary"    "the"    "red"
            "carpet"    "face"    "of"    "luxury"
  Columns 34 through 38
    "brands"    "and"    "a"    "successful"    "businesswoman"
n =
  Columns 1 through 20
     1     2     3     4     5     6     7     8     9    10    11    36
            12    32    13    14    15    28    16    17
  Columns 21 through 40
    18    28    19    32    20    21    35    22    23    24    25    26
           36    27    32    28    29    30    36    31
  Columns 41 through 47
    32    33    34    35    36    37    38
```

`d` is a string array and maps onto array `n`.

8.4 Map Sentences to Numbers

8.4.1 Problem

We want to map sentences to unique numbers.

8.4.2 Solution

Write a MATLAB function to search through text and assign a unique number to each word.

8.4.3 How It Works

The function splits the string and searches using `d`. The last line removes any words (in this case, only punctuation) that are not in the dictionary.

MapToNumbers.m

```
function n = MapToNumbers( w, d )

% Demo
if( nargin < 1 )
  Demo;
  return
end
```

```
8
9  w = erase(w,';');
10 w = erase(w,',');
11 w = erase(w,'.');
12 s = split(w)';  % string array
13
14 n = zeros(1,length(s));
15 for k = 1:length(s)
16   ids = find(strcmp(s(k),d));
17   if ~isempty(ids)
18     n(k) = ids;
19   end
20
21 end
22
23 n(n==0) = [];
```

This is the built-in demo.

```
  >> MapToNumbers
w =
    "No one knew it then, but she was being held under a type of house
        arrest while the tax authorities scoured the records of her long
        and lucrative career as an actress, a luminary of the red carpet,
         a face of luxury brands and a successful businesswoman."
n =
  Columns 1 through 19
     1     2     3     4     0     6     7     8     9    10    11    36
            12    32    13    14    15    28    16
  Columns 20 through 38
    17    18    28    19    32    20    21    35    22    23    24    25
             0    36    27    32    28    29     0
  Columns 39 through 46
    36    31    32    33    34    35    36    37
```

8.5 Converting the Sentences

8.5.1 Problem

We want to convert the sentences to numeric sequences.

8.5.2 Solution

Write a MATLAB function to take each sentence, add the words, and create a sequence. Each sentence is classified as correct or incorrect.

8.5.3 How It Works

The script reads in the database. It creates a numeric dictionary for all of the sentences and then converts them to numbers. This first part creates 5200 sentences. Each is classified as correct or incorrect. Note how we initialize a string array.

PrepareSequences.m

```
%% See also
% ReadDatabase, extractBefore, extractAfter, MapToNumbers

[s,u,v,a] = ReadDatabase;

% Whatever you want in the training
nSentences = 100; %length(s);

i = 1;
c = zeros(size(v,2)*nSentences,1);
z = strings(size(v,2)*nSentences,1);
for k = 1:nSentences
  q1    = extractBefore(s(k),u(k,1));
  q2    = extractAfter(s(k),u(k,2));
  for j = 1:size(v,2)
    z(i)  = q1 + v(k,j) + q2;
    if( j == a(k,1) )
      c(i) = 1;
    else
      c(i) = 0;
    end
    i = i + 1;
  end
end
```

The next section concatenates all of the sentences into a gigantic string and creates a dictionary.

```
%% Create a numeric dictionary
r = z(1);
for k = 2:length(z)
  r = r + " " + z(k); % append all the sentences to one string
end

d = DistinctWords( r ); % find the distinct words
```

The final part creates the numeric sentences and saves them. The loop that prints the lines shows a handy way of printing an array using `fprintf`.

```
  nZ{k} = MapToNumbers( z(k), d );
end

% Print 2 sentences
for k = 1:10
```

```
    fprintf('Category: %d',c(k));
    fprintf('%5d',nZ{k})
    fprintf('\n')
    if( mod(k,5) == 0 )
      fprintf('\n')
    end
  end

  %% Save the numbers and category in a mat-file

  save('Sentences','nZ', 'c');
```

As expected, only one word is different in each set of five sentences.

```
>> PrepareSequences
Category: 0   1 428 538 541 553 103   6 535 149  10   7 170   8 546 544
      9 546  10   2  12 404
Category: 0   1 428 538 541 553 103   6 535 149  10   7 170   8 546 544
      9 546  10   3  12 404
Category: 1   1 428 538 541 553 103   6 535 149  10   7 170   8 546 544
      9 546  10   4  12 404
Category: 0   1 428 538 541 553 103   6 535 149  10   7 170   8 546 544
      9 546  10   5  12 404
Category: 0   1 428 538 541 553 103   6 535 149  10   7 170   8 546 544
      9 546  10  11  12 404

Category: 1 323 481 378  19 465 544  18 546  19 465 544  20 549  21  22
    14 404  24  25 546
Category: 0 323 481 378  19 465 544  18 546  19 465 544  20 549  21  22
    15 404  24  25 546
Category: 0 323 481 378  19 465 544  18 546  19 465 544  20 549  21  22
    16 404  24  25 546
Category: 0 323 481 378  19 465 544  18 546  19 465 544  20 549  21  22
    17 404  24  25 546
Category: 0 323 481 378  19 465 544  18 546  19 465 544  20 549  21  22
    23 404  24  25 546
```

8.6 Training and Testing

8.6.1 Problem

We want to build a deep learning system to complete sentences. The idea is that the full database of correct and incorrect sentences provides enough information for the neural net to deduce the grammar and meaning.

8.6.2 Solution

Write a MATLAB script to implement an LSTM to classify the sentences as correct or incorrect. The LSTM will be trained with complete sentences. No information about the words, such as whether a word is noun, verb, or adjective, nor of any grammatical structure will be used.

8.6.3 How It Works

We will produce the simplest possible design. It will read in sentences, classified as correct or incorrect, and attempt to determine if new sentences are correct or incorrect just from the learned patterns. This is a very simple and crude approach. We aren't taking advantage of our knowledge of grammar, word types (verb, noun, etc.), or context to help with the predictions. Language modeling is a huge field, and we are not using any results from that body of work. Of course, applications of all of the rules of grammar don't necessarily ensure success; otherwise there would be more 800s on the SAT verbal tests.

We use the same code to make sure the sequences are valid.

SentenceCompletionNN.m

```
%% Load the data
s = load('Sentences');
n = length(s.c);        % number of sentences

% Make sure the sequences are valid. One in every 5 is complete.
for k = 1:10
  fprintf('Category: %d',s.c(k));
  fprintf('%5d',s.nZ{k})
  fprintf('\n')
  if( mod(k,5) == 0 )
    fprintf('\n')
  end
end

%% Set up the network
```

Each set has one correct sentence. The remainder have the wrong answers.

```
>> SentenceCompletionNN
Category: 0   1 428 538 541 553 103   6 535 149  10   7 170   8 546 544
      9 546  10   2  12 404
Category: 0   1 428 538 541 553 103   6 535 149  10   7 170   8 546 544
      9 546  10   3  12 404
Category: 1   1 428 538 541 553 103   6 535 149  10   7 170   8 546 544
      9 546  10   4  12 404
Category: 0   1 428 538 541 553 103   6 535 149  10   7 170   8 546 544
      9 546  10   5  12 404
Category: 0   1 428 538 541 553 103   6 535 149  10   7 170   8 546 544
      9 546  10  11  12 404

Category: 1 323 481 378  19 465 544  18 546  19 465 544  20 549  21  22
    14 404  24  25 546
```

```
Category: 0 323 481 378  19 465 544  18 546  19 465 544  20 549  21  22
    15 404  24  25 546
Category: 0 323 481 378  19 465 544  18 546  19 465 544  20 549  21  22
    16 404  24  25 546
Category: 0 323 481 378  19 465 544  18 546  19 465 544  20 549  21  22
    17 404  24  25 546
Category: 0 323 481 378  19 465 544  18 546  19 465 544  20 549  21  22
    23 404  24  25 546
```

Because we have access to full sequences at prediction time, we use a bidirectional LSTM layer in the network. A bidirectional LSTM layer learns from the full sequence at each time step. The training code follows. We convert the classes, 0 and 1, to a categorical variable.

```
numFeatures = 1;
numHiddenUnits = 400;
numClasses = 2;

layers = [ ...
    sequenceInputLayer(numFeatures)
    bilstmLayer(numHiddenUnits,'OutputMode','last')
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer];

disp(layers)

options = trainingOptions('adam', ...
    'MaxEpochs',60, ...
    'MiniBatchSize',20,...
    'GradientThreshold',1, ...
    'SequenceLength','longest', ...
    'Shuffle','never', ...
    'Verbose',1, ...
    'InitialLearnRate',0.01,...
    'Plots','training-progress');
%    'SequenceLength','longest', ...

%% Train the network - Uniform set
nSentences  = n/5; % number of complete sentences in the database
nTrain      = floor(0.75*nSentences);       % use 75% for training
xTrain      = s.nZ(1:5*nTrain);             % sentence indices, in order
yTrain      = categorical(s.c(1:5*nTrain)); % complete or not?
net         = trainNetwork(xTrain,yTrain,layers,options);
```

The output is

```
5x1 Layer array with layers:

   1   ''   Sequence Input          Sequence input with 1 dimensions
   2   ''   BiLSTM                 BiLSTM with 400 hidden units
   3   ''   Fully Connected         2 fully connected layer
```

```
      4   ''   Softmax                 softmax
      5   ''   Classification Output   crossentropyex
 Uniform set
     0.8000
 Random set
     0.6176
```

The first layer says the input is a one-dimensional sequence. The second is the bidirectional LSTM. The next layer is a fully connected layer of neurons. This is followed by a Softmax layer and then by the classification layer. The standard Softmax is

$$\sigma_k = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{8.1}$$

which is essentially a normalized output.

The testing code is

```
%% Train the network - Uniform set
nSentences  = n/5; % number of complete sentences in the database
nTrain      = floor(0.75*nSentences);       % use 75% for training
xTrain      = s.nZ(1:5*nTrain);             % sentence indices, in
    order
yTrain      = categorical(s.c(1:5*nTrain)); % complete or not?
net         = trainNetwork(xTrain,yTrain,layers,options);

% Test this network - 80% accuracy
xTest       = s.nZ(5*nTrain+1:end);
yTest       = categorical(s.c(5*nTrain+1:end));
yPred       = classify(net,xTest);

% Calculate the classification accuracy of the predictions.
acc         = sum(yPred == yTest)./numel(yTest);
disp('Uniform set')
disp(acc);

%% Train the network using randomly selected sentences
kTrain  = randperm(n,5*nTrain); % nTrain (30!) integers in range 1:n
xTrain  = s.nZ(kTrain);
yTrain  = categorical(s.c(kTrain));
net     = trainNetwork(xTrain,yTrain,layers,options);

% Test the network
kTest = setdiff(1:n,kTrain);
xTest = s.nZ(kTest);
yTest = categorical(s.c(kTest));
yPred = classify(net,xTest);

% Calculate the classification accuracy of the predictions.
acc = sum(yPred == yTest)./numel(yTest);
```

```
43
44  disp('Random set')
45  disp(acc);
```

Figure 8.1 shows learning with a uniform set. All five sentences are input; only one has the correct answer. The learning system quickly figures out that it can get 80% accuracy by classifying each sentence as wrong.

Figure 8.2 shows learning with a random set. The sentences are drawn in random order from the entire database. Training now reaches 93% accuracy. If you run it multiple times, you will see that the results vary.

Figure 8.3 shows a second run. The random sets produce networks with lower probabilities of success, but at least the network is trying to find correct sentences. This particular approach is very simple. Nonetheless, it shows the potential for working with text and ultimately understanding written language.

Figure 8.1: Learning with a uniform input of sentences.

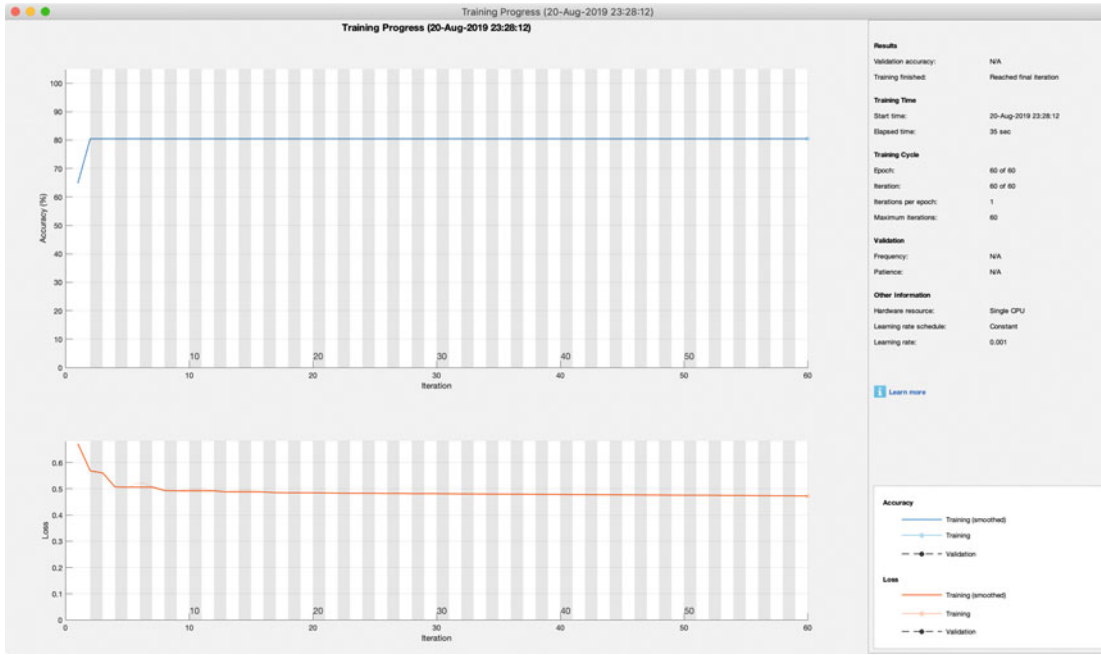


Figure 8.2: Learning with a random set of sentences.

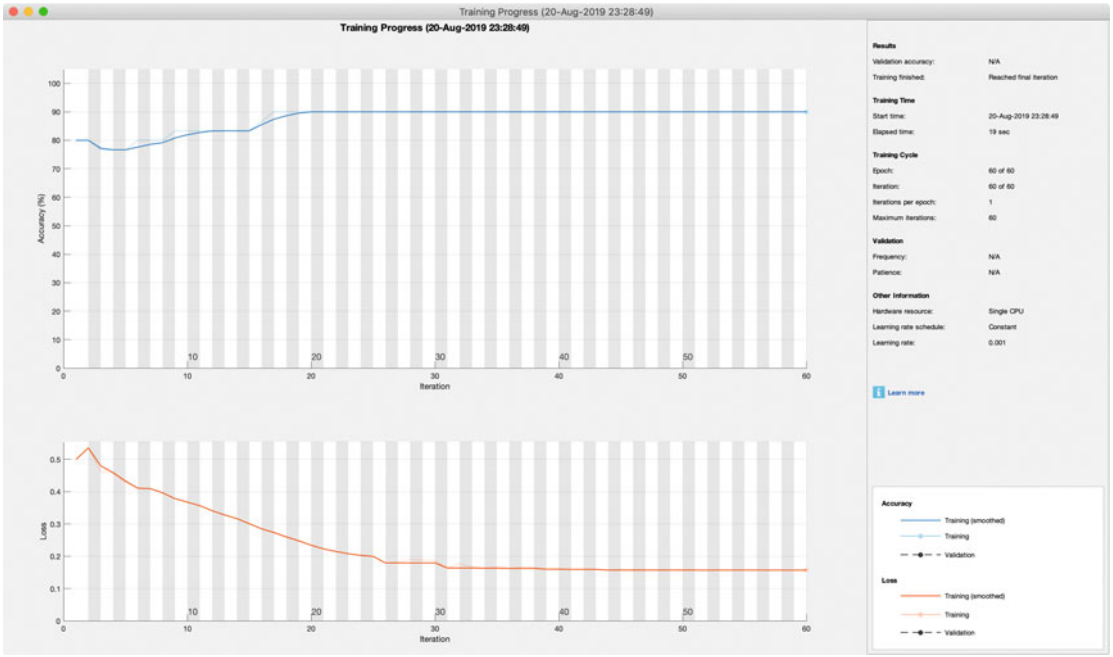
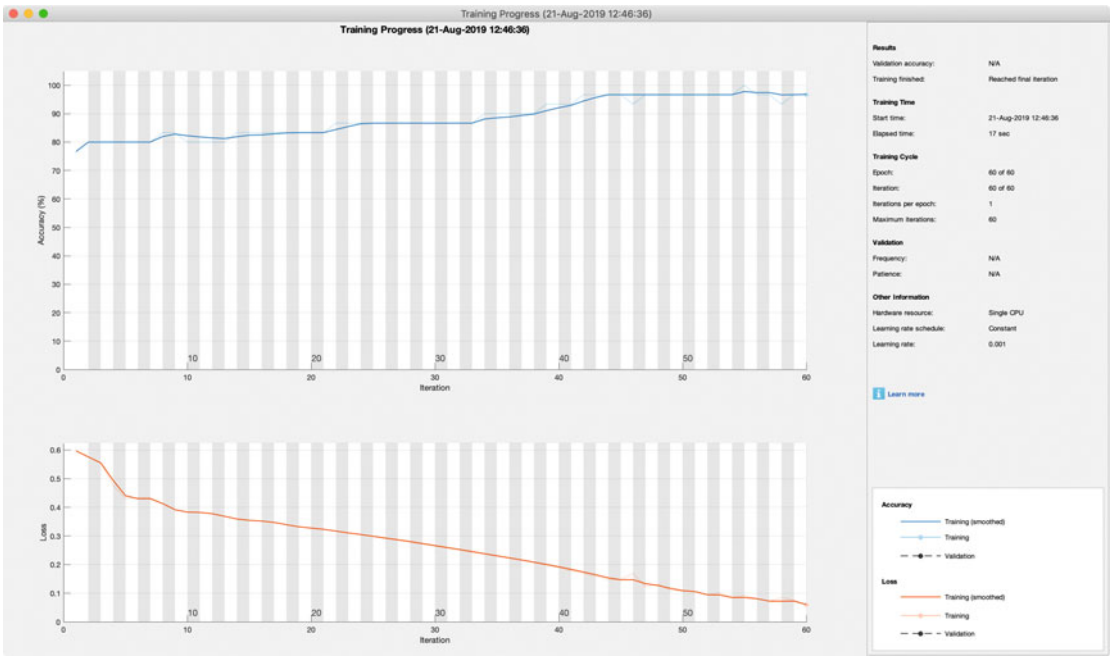


Figure 8.3: Learning with a different random set of sentences. This reaches 95% accuracy with the training set.



CHAPTER 9

Terrain-Based Navigation

9.1 Introduction

Prior to the widespread availability of GPS, Loran, and other electronic navigation aids, pilots used visual cues from terrain to navigate. Now everyone uses GPS. We want to return to the good old days of terrain-based navigation. We will design a system that will be able to match terrain with a database. It will then use that information to determine where it is flying.

9.2 Modeling Our Aircraft

9.2.1 Problem

We want a three-dimensional aircraft model that can change direction.

9.2.2 Solution

Write the equations of motion for three-dimensional flight.

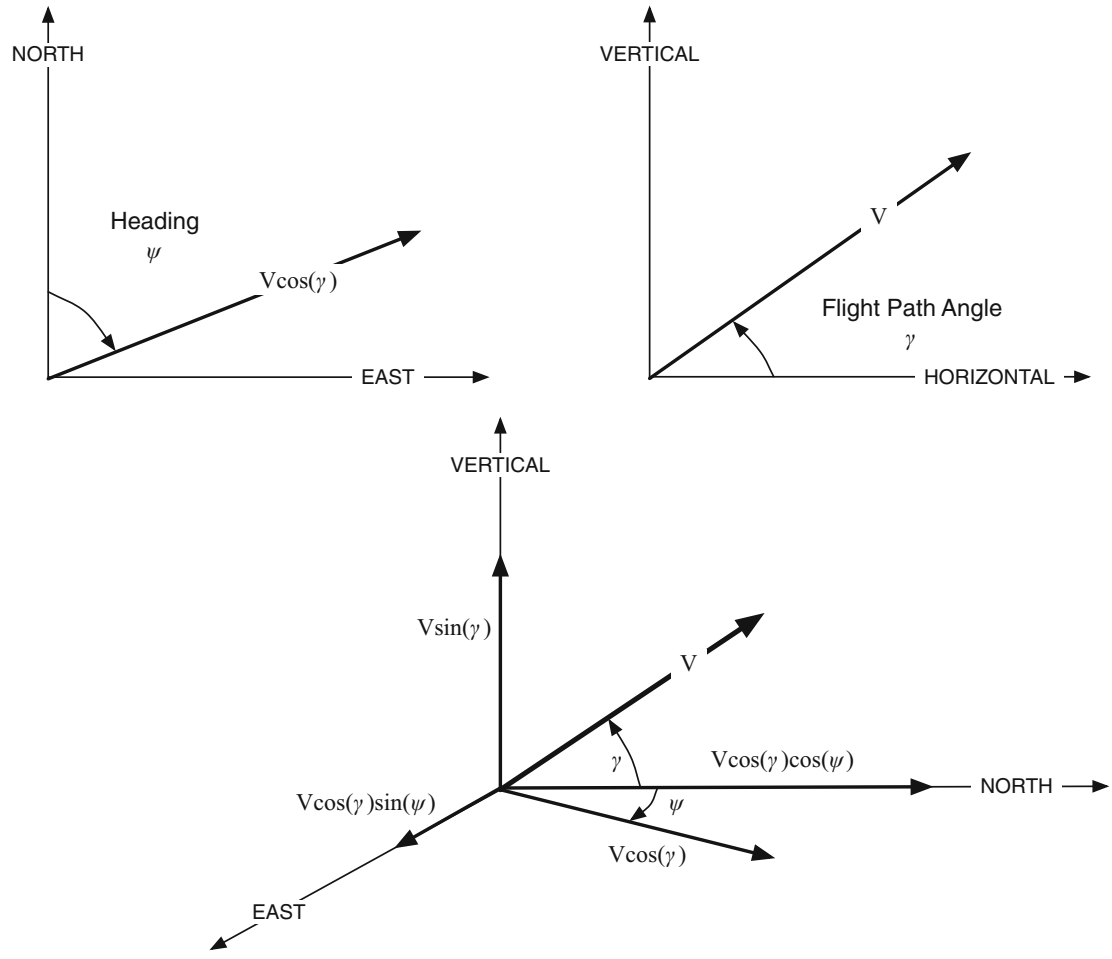
9.2.3 How It Works

The motion of a point mass through three-dimensional space has 3 degrees of freedom. Our aircraft model is therefore given 3 degrees of spatial freedom. The velocity vector is expressed as a wind-relative magnitude (V) with directional components for heading (ψ) and flight path angle (γ). The position is a direct integral of the velocity, and is expressed in $y =$ North, $x =$ East, $h =$ Vertical coordinates. In addition, the engine thrust is modeled as a first-order system where the time constant can be changed to approximate the engine response times of different aircrafts.

Figure 9.1 shows a diagram of the velocity vector in the North-East-Up coordinate system. The time derivatives are taken in this frame. This is not a purely inertial coordinate system, because it is rotating with the Earth. However, the rate of rotation of the Earth is sufficiently small compared to the aircraft turning rates so that it can be safely neglected.

M. Paluszek and S. Thomas, *Practical MATLAB Deep Learning*,
https://doi.org/10.1007/978-1-4842-5124-9_9

Figure 9.1: Velocity in North-East-Up coordinates.



The point mass aircraft equations of motion are

$$\dot{v} = (T\cos\alpha - D - mg\sin\gamma)/m - f_v \tag{9.1}$$

$$\dot{\gamma} = \frac{1}{mv}\left((L + T\sin\alpha)\cos\phi - mg\cos\gamma + f_\gamma\right) \tag{9.2}$$

$$\dot{\psi} = \frac{1}{mv\cos\gamma}\left((L + T\sin\alpha)\sin\phi - f_\psi\right) \tag{9.3}$$

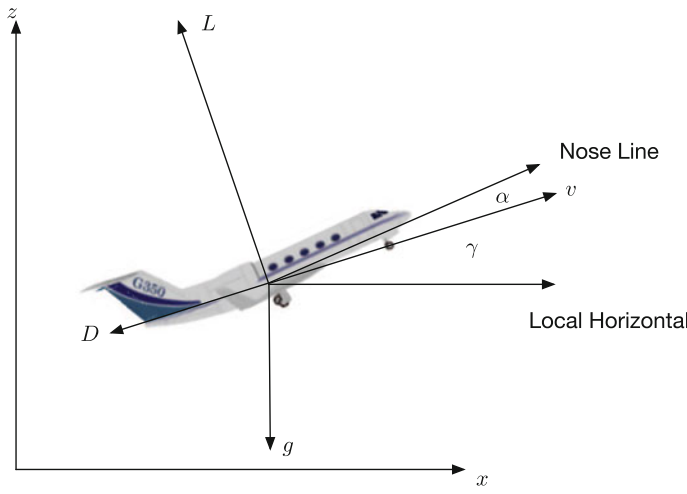
$$\dot{x_e} = v\cos\gamma\sin\psi + W_x \tag{9.4}$$

$$\dot{y_n} = v\cos\gamma\cos\psi + W_y \tag{9.5}$$

$$\dot{h} = v\sin\gamma + W_h \tag{9.6}$$

$$\dot{m} = -\frac{T}{u_e} \tag{9.7}$$

Figure 9.2: Aircraft model showing lift, drag, and gravity.



where v is the true airspeed, T is the thrust, L is the lift, g is the acceleration of gravity, γ is the air-relative flight path angle, ψ is the air-relative heading (measured clockwise from North), ϕ is the bank angle, x and y are the East and North positions, respectively, and h is the altitude. The mass is the total of dry mass and fuel mass. The terms $\{f_v, f_\gamma, f_\psi\}$ represent additional forces due to modeling uncertainty, and the terms $\{W_x, W_y, W_h\}$ are wind speed components. If the vertical wind speed is zero, then $\gamma = 0$ produces level flight. α, ϕ, and T are the controls. Figure 9.2 shows the longitudinal symbols for the aircraft. γ is the angle between the velocity vector and local horizontal. α is the angle of attack which is between the nose of the aircraft and the velocity vector. The wings may be oriented, or have airfoils, that give lift at zero angle of attack. Drag is opposite velocity and lift is perpendicular to drag. Lift must balance gravity and any downward component of drag; otherwise the aircraft will descend.

We are using a very simple aerodynamic model. The lift coefficient is defined as

$$c_L = c_{L_\alpha}\alpha \tag{9.8}$$

The lift coefficient is really a nonlinear function of angle of attack. It has a maximum angle of attack above which the wing stalls and all lift is lost. For a flat plate, $c_{L_\alpha} = 2\pi$. The drag coefficient is

$$c_D = c_{D_0} + \frac{c_L^2}{\pi A_R \epsilon} \tag{9.9}$$

where A_R is the aspect ratio and ϵ is the Oswald efficiency factor which is typically from 0.8 to 0.95. The efficiency factor is how efficiently lift is couple to drag. If it is less than one, it means that lift produces more lift-induced drag than the ideal. The aspect ratio is the ratio of the wing span (from the point nearest the fuselage to the tip) and the chord (the length from the front to the back of the wing).

The dynamic pressure, the pressure due to the motion of the aircraft, is

$$q = \frac{1}{2}\rho v^2 \tag{9.10}$$

where v is the speed and ρ is the atmospheric density. This is the pressure on your hand if you stick it out of the window of a moving car. The lift and drag forces are

$$L = qc_L s \tag{9.11}$$

$$D = qc_D s \tag{9.12}$$

where s is the wetted area. The wetted area is the surface of the aircraft that produces lift and drag. We make it the same for lift and drag, but in a real aircraft, some parts of the aircraft cause drag (like the nose) but don't produce any lift. In essence, we assume the aircraft is all wing.

We create a right-hand side function for the model. This will be called by the numerical integration function. The following has the dynamical model.

RHSPointMassAircraft.m

```
1
2  if( nargin < 1 )
3    xDot = DefaultDataStructure;
4    return
5  end
6
7  v          = x(1);
8  gamma      = x(2);
9  psi        = x(3);
10 h          = x(6);
11 cA         = cos(d.alpha);
12 sA         = sin(d.alpha);
13 cG         = cos(gamma);
14 sG         = sin(gamma);
15 cPsi       = cos(psi);
16 sPsi       = sin(psi);
17 cPhi       = cos(d.phi);
18 sPhi       = sin(d.phi);
19
20 mG         = d.m*d.g;
21 qS         = 0.5*d.s*Density( 0.001*h )*v^2;
22 cL         = d.cLAlpha*d.alpha;
23 cD         = d.cD0 + cL^2/(pi*d.aR*d.eps);
24 lift       = qS*cL;
25 drag       = qS*cD;
26 vDot       = (d.thrust*cA - drag - mG*sG)/d.m + d.f(1);
27 fN         = lift + d.thrust*sA;
28 gammaDot   = (fN*cPhi - mG*cG + d.f(2))/(d.m*v);
29 psiDot     = (fN*sPhi - d.f(3))/(d.m*v*cG);
30 xDot       = [vDot;gammaDot;psiDot;v*cG*sPsi;v*cG*cPsi;v*sG];
```

The default data structure is defined in the subfunction, `DefaultDataStructure`. The data structure includes both constant parameters and control inputs.

```
32
33 d = struct('cD0',0.01,'aR',2.67,'eps',0.95,'cLAlpha',2*pi,'s',64.52,...
34             'g',9.806,'alpha',0,'phi',0,'thrust',0,'m',19368.00,...
35             'f',zeros(3,1),'W',zeros(3,1));
```

We use a modified exponential atmosphere for the density:

```
37 function rho = Density( h )
38
39 rho = 1.225*exp(-0.0817*h^1.15);
```

We want to maintain a force balance so that the speed of the aircraft is constant and the aircraft does not change its flight path angle. For example, in level flight, the aircraft would not ascend or descend. We need to control the aircraft in level flight so that the velocity stays constant and $\gamma = 0$ for any ϕ. The relevant equations are

$$0 = T\cos\alpha - D \tag{9.13}$$

$$0 = (L + T\sin\alpha)\cos\phi - mg \tag{9.14}$$

We need to find T and α given ϕ.

A simple way is to use `fminsearch`. It will call `RHSPointMassAircraft` and numerically find controls that, for a given ψ, h and v have zero time derivatives. The following code finds equilibrium angle of attack and thrust. `RHS` is called by `fminsearch`. It returns a scalar cost that is a quadratic of the acceleration (time derivative of velocity) and derivative of the flight path angle. Our initial guess is a value of thrust that balances the drag. Even with an angle of attack guess of 0, it converges with the default set of parameters `opt = optimset('fminsearch')`.

EquilibriumControls.m

```
1  function d = EquilibriumControls( x, d )
2
3  if( nargin < 1 )
4    Demo
5    return
6  end
7
8  [~,~,drag]  = RHSPointMassAircraft( 0, x, d );
9  u0          = [drag;0];
10 opt         = optimset('fminsearch');
11 u           = fminsearch( @RHS, u0, opt, x, d );
12 d.thrust    = u(1);
13 d.alpha     = u(2);
14
15 %% EquilibriumControls>RHS
16 function c = RHS( u, x, d )
17
18 d.thrust  = u(1);
19 d.alpha   = u(2);
```

```
xDot      = RHSPointMassAircraft( 0, x, d );
c         = xDot(1)^2 + xDot(2)^2;
```

The demo is for a Gulfstream 350 flying at 250 m/s and 10 km altitude.

```
function Demo

d     = RHSPointMassAircraft;
d.phi = 0.4;
x     = [250;0;0.02;0;0;10000];
d     = EquilibriumControls( x, d );
r     = x(1)^2/(d.g*tan(d.phi));

fprintf('Thrust          %8.2f N\n',d.thrust);
fprintf('Altitude        %8.2f km\n',x(6)/1000);
fprintf('Angle of attack %8.2f deg\n',d.alpha*180/pi);
fprintf('Bank angle      %8.2f deg\n',d.phi*180/pi);
fprintf('Turn radius     %8.2f km\n',r/1000);
```

The results of the demo, shown in the following, are quite reasonable.

```
>> EquilibriumControls
Thrust           7614.63 N
Altitude           10.00 km
Angle of attack     2.41 deg
Bank angle         22.92 deg
Turn radius        15.08 km
```

With these values, the plane will turn without changing altitude or airspeed. We simulate the Gulfstream in `AircraftSim`. The first part runs our equilibrium computation demo.

AircraftSim.m

```
%% Script to simulate a Gulfstream 350 in a banked turn

n   = 500;
dT  = 1;
rTD = 180/pi;

%% Start by finding the equilibrium controls
d     = RHSPointMassAircraft;
d.phi = 0.4;
x     = [250;0;0.02;0;0;10000];
d     = EquilibriumControls( x, d );
r     = x(1)^2/(d.g*tan(d.phi));

fprintf('Thrust          %8.2f N\n',d.thrust);
fprintf('Altitude        %8.2f km\n',x(6)/1000);
fprintf('Angle of attack %8.2f deg\n',d.alpha*180/pi);
fprintf('Bank angle      %8.2f deg\n',d.phi*180/pi);
fprintf('Turn radius     %8.2f km\n',r/1000);
```

The next part does the simulation. It breaks the loop if the aircraft altitude is less than 0, that is, it crashes. We call `RHSPointMassAircraft` once to get the lift and drag value for plotting. It is then called by `RungeKutta` to do the numerical integration. `@` denotes a pointer to the function.

```
%% Simulation
xPlot = zeros(length(x)+5,n);

for k = 1:n

  % Get lift and drag for plotting
  [~,L,D]     = RHSPointMassAircraft( 0, x, d );

  % Plot storage
  xPlot(:,k)  = [x;L;D;d.alpha*rTD;d.thrust;d.phi*rTD];

  % Integrate
  x           = RungeKutta( @RHSPointMassAircraft, 0, x, dT, d );

  % A crash
  if( x(6) <= 0 )
    break;
  end
end
```

The remainder produces three plots. The first plot is the states that are numerically integrated. The next gives the controls, lift, and drag. The final plot shows the planar trajectory. We do unit conversions since degrees and kilometers are a bit clearer.

```
%% Plot the results
xPlot         = xPlot(:,1:k);
xPlot(2,:)    = xPlot(2,:)*rTD;
xPlot(4:6,:)  = xPlot(4:6,:)/1000;
yL            = {'v (m/s)' '\gamma (deg)' '\psi (deg)' 'x_e (km)'  'y_n
     (km)'...
                  'h (km)' 'L (N)' 'D (N)' '\alpha (deg)' 'T (N)' '\phi
                      (deg)'};
[t,tL]        = TimeLabel(dT*(0:(k-1)));

PlotSet( t, xPlot(1:6,:), 'x label', tL, 'y label', yL(1:6),...
  'figure title', 'Aircraft State' );
PlotSet( t, xPlot(7:11,:), 'x label', tL, 'y label', yL(7:11),...
  'figure title', 'Aircraft Lift, Drag and Controls' );
```

As you can see in Figure 9.4, the radius of the turn is 15 km as expected. The drag and lift remain constant. In practice we would have a velocity and flight path angle control system to handle disturbances or parameter variations. For the purpose of our deep learning example, we just use the ideal dynamics. Figure 9.3 shows the simulation outputs.

Figure 9.3: Simulation outputs. States (the integrated quantities) are on the top. Lift, drag, and the controls ϕ, α, and T are on the bottom.

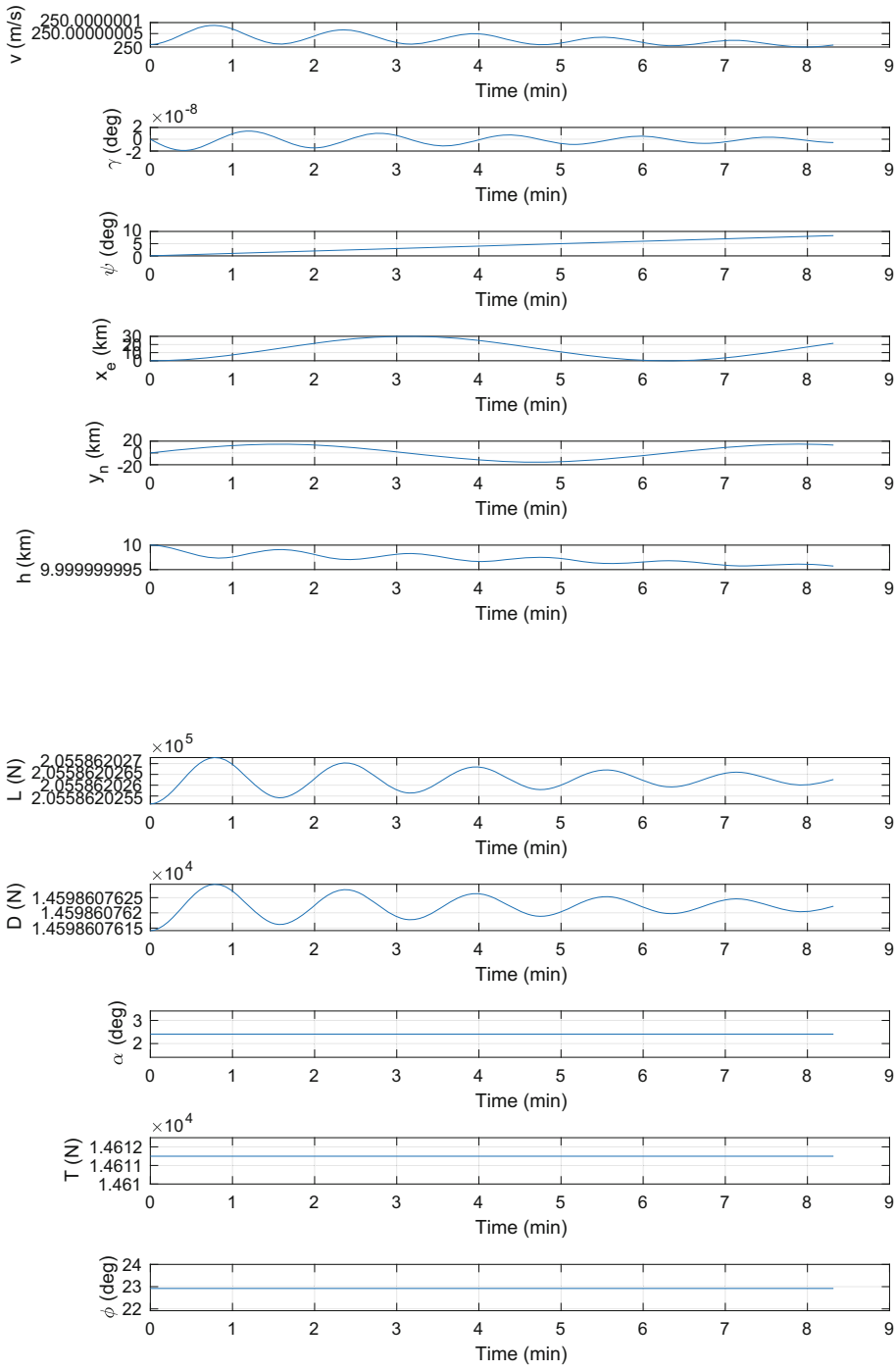


Figure 9.4: Aircraft trajectory.

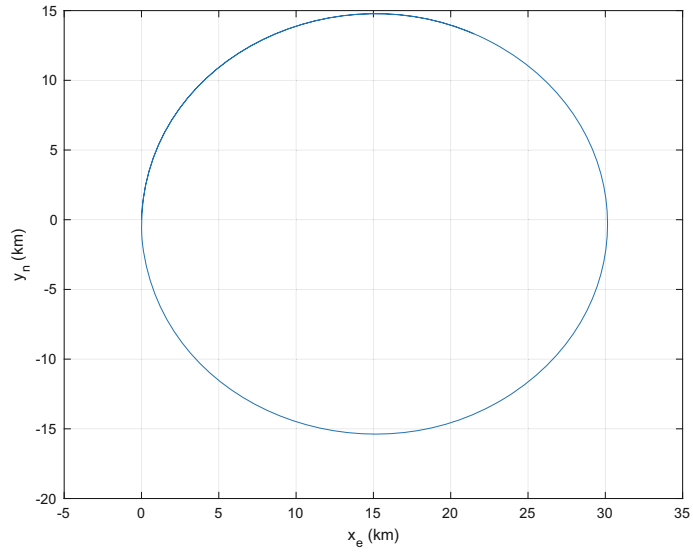


Figure 9.4 will provide a nice trajectory for our deep learning examples. You can change the aircraft simulation to produce other trajectories.

9.3 Generating a Terrain Model

9.3.1 Problem

We want to create an artificial terrain model from a set of terrain ‘‘tiles.’’ A tile is a segment of terrain from a bigger picture, much like bathroom tiles make up a bathroom wall. Unless, of course, you have the modern fiberglass shower.

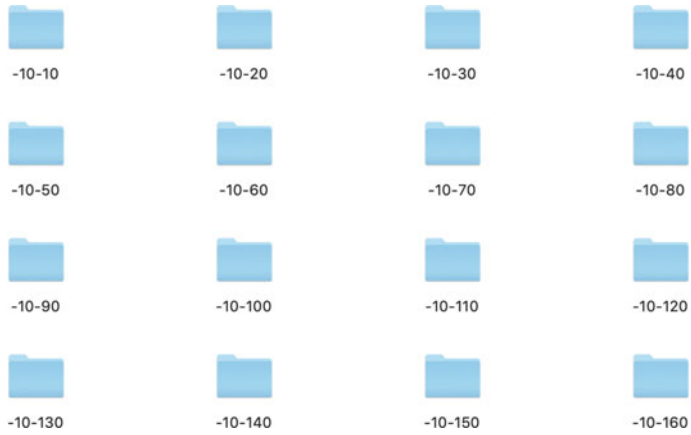
9.3.2 Solution

Find images of terrain and tile them together. There are many sources of terrain tiles. GoogleEarth is one.

9.3.3 How It Works

We start by compiling a database of terrain tiles. We have them in the folder terrain in our MATLAB package. A segment of the terrain folder is shown in Figure 9.5. This is just one way to get terrain tiles. There are online sources for downloading tiles. Also many flight simulator games have extensive terrain libraries. The name of the folder is `latitude longitude`. For example, `-10-10` is -10 degrees latitude and -10 degrees longitude. Our database only extends to $\pm$ 60 degrees latitude. The first block creates a list of the folders in terrain. An important thing with this code is that your script needs to be in the correct directory. We don’t do any fancy directory searching.

Figure 9.5: A segment of the terrain folder.



CreateTerrain.m

```
function CreateTerrain( lat, lon, scale )

% Demo
if( nargin < 1 )
  Demo;
  return
end

d           = dir('terrain');
latA        = zeros(1,468);
lonA        = zeros(1,468);
folderName  = cell(1,468);
for k = 1:468
  q               = d(k).name;
  folderName{k}   = q;
  if( q(2) == '0' )
    latA(k) = str2double(q(1:2));
    lonA(k) = str2double(q(3:end));
  else
    latA(k) = str2double(q(1:3));
    lonA(k) = str2double(q(4:end));
  end
end
```

The next code block finds the indices for the desired tiles.

```
% Center lower left corner is start
latF    = floor(lat);
lonF    = floor(lon);
latI    = zeros(1,9);
lonI    = zeros(1,9);
```

```
lon0  = lonF - 10;
latJK = latF - 10;
lonJK = lon0;
i     = 1;
for j = 1:3
  for k = 1:3
    lonI(i) = lonJK;
    latI(i) = latJK;
    lonJK   = lonJK + 10;
    i       = i + 1;
  end
  lonJK = lon0;
  latJK = latJK + 10;
end

fldr = zeros(1,9);
for k = 1:9
  j       = find(latI(k)==latA);
  i       = lonI(k)==lonA(j);
  fldr(k) = j(i);
end
```

The following code creates the filenames based on our latitudes and longitudes. We just create correctly formatted strings. This shows one way to create strings. Notice we use `%d` to create integers. It automatically makes them the right length. We need to check for positive and negative so that the + and - signs are correct.

```
% Generate the file names
imageSet  = cell(1,9);
for k = 1:9
  j = fldr(k);
  if( latA(j) >= 0  )
    if( lonA(j) >= 0 )
      imageSet{k} = sprintf('grid10x10+%d+%d',latA(j)*100,lonA(j)*100);
    else
      imageSet{k} = sprintf('grid10x10+%d-%d',latA(j)*100,lonA(j)*100);
    end
  else
    if( lonA(j) >= 0 )
      imageSet{k} = sprintf('grid10x10-%d+%d',latA(j)*100,lonA(j)*100);
    else
      imageSet{k} = sprintf('grid10x10-%d-%d',latA(j)*100,lonA(j)*100);
    end
  end
end
```

The next block reads in the image, flips it upside down, and scales the image. The images happen to be north down and south up. We first change directory to be in `terrain` then `cd` to go into each folder. `cd ..` changes directories back into `terrain`.

```
% Assuming we are one directory above
cd terrain

im    = cell(1,9);
for k = 1:9
        j = fldr(k);
  cd(folderName{j})
        im{k} = ScaleImage(flipud(imread([imageSet{k},'.jpg'])),scale);
  cd ..
end
```

The next block of code calls `image` to draw each image in the correct spot on the 3 by 3 tiled map.

```
del   = size(im{1},1);
lX    = 3*del;

% Draw the images
x     = 0;
y     = 0;
for k = 1:9
  image('xdata',[x;x+del],'ydata',[y;y+del],'cdata', im{k} );
  hold on
  x = x + del;
  if ( x == lX )
    x = 0;
    y = y + del;
  end
end
axis off
axis image

cd ..
```

The subfunction `ScaleImage` scales the image by doing a mean of the pixels that are scaled down to 1 pixel. At the very end, we `cd ..` putting us into the original directory.

```
97  %% CreateTerrain>ScaleImage
98  function s2 = ScaleImage( s1, q )
99
100 n = 2^q;
101
102 [mR,~,mD] = size(s1);
103
104 m = mR/n;
105
106 s2 = zeros(m,m,mD,'uint8');
107
108 for i = 1:mD
109   for j = 1:m
110     r = (j-1)*n+1:j*n;
111     for k = 1:m
112       c          = (k-1)*n+1:k*n;
113       s2(j,k,i) = mean(mean(s1(r,c,i)));
114     end
115   end
116 end
```

The demo picks a latitude and longitude in the Middle East. The results are the 3 by 3 tiled image is shown in Figure 9.6. We won't use this image for the neural net because it would be too low resolution for anything but a satellite.

```
117 %% CreateTerrain>Demo
118 function Demo
119
120 NewFigure('EarthSegment');
121 CreateTerrain( 30,60,1 )
```

Figure 9.6: Terrain tiled image of the Middle East.

9.4 Close Up Terrain

9.4.1 Problem

We want higher-resolution terrain.

9.4.2 Solution

Specialize the terrain code to produce a small segment of higher-resolution terrain suitable for experiments with a commercial drone.

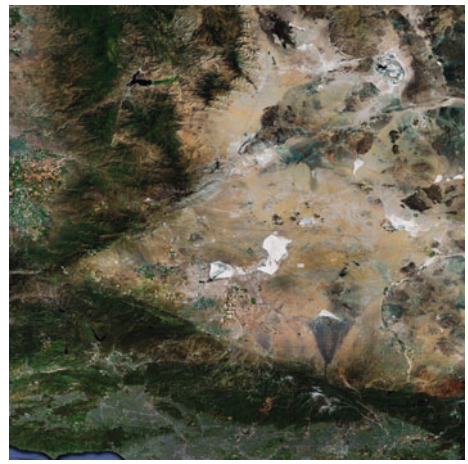
9.4.3 How It Works

The preceding terrain code would work well for an orbiting satellite, but not so well for a drone. Per FAA regulations, the maximum altitude for small unmanned aircraft is 400 feet, or about 122 meters. A satellite in Low Earth Orbit (LEO) typically has an altitude of 300--500 km. Thus, drones are typically about 2500--4000 times closer to the surface than a satellite! We take the code and specialize it to read in just four images. It is much simpler than `CreateTerrain` and is less flexible. If you want to change it, you will need to change the code in the file.

CreateTerrainClose.m

```
function CreateTerrainClose

% Generate the file names
imageSet  = {'grid1x1+3400-11800','grid1x1+3400-11900',...
             'grid1x1+3500-11800','grid1x1+3500-11900'};
p = [2 1 4 3];

% Assuming we are one directory above
cd terrainclose

im = cell(1,4);
for k = 1:4
        im{k} = flipud(imread([imageSet{k},'.jpg']));
end

del = size(im{1},1);

% Draw the images
x     = 0;
y     = 0;
i     = 0;
for k = 1:2
  for j = 1:2
    i = i + 1;
    image('xdata',[x;x+del],'ydata',[y;y+del],'cdata', im{p(i)} );
    hold on
    x = x + del;
  end
  x = 0;
  y = y + del;
```

Figure 9.7: Close up terrain.



```
31  end
32  axis off
33  axis image
34
35  cd ..
```

We don't have any options for scaling. This runs the function:

```
>> NewFigure('EarthSegmentClose');
>> CreateTerrainClose
```

Figure 9.7 shows the terrain. It is 2 degrees by 2 degrees.

9.5 Building the Camera Model

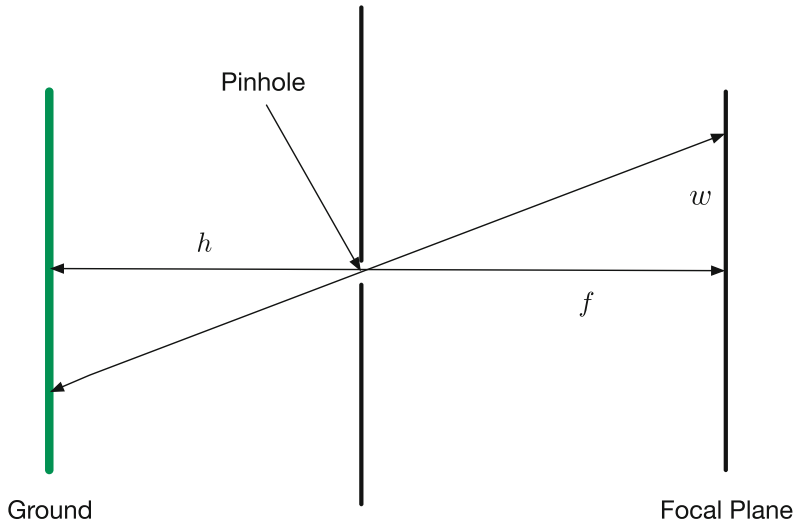
9.5.1 Problem

We want to build a camera model for our deep learning system. We want a model that emulates the function of a drone-mounted camera. Ultimately, we will use this camera model as part of a terrain-based navigation system, and we'll apply deep learning techniques to do the terrain navigation.

9.5.2 Solution

We will model a pinhole camera and create a high altitude aircraft. A pinhole camera is the lowest order approximation to a real optical system. We'll then build the simulation and demonstrate the camera.

Figure 9.8: Pinhole camera.



9.5.3 How It Works

We've already created an aircraft simulation in Recipe 9.2. The addition will be the terrain model and the camera model. A pinhole camera is shown in Figure 9.8. A pinhole camera has infinite depth of field, and the images are rectilinear.

A point $P(x, y, z)$ is mapped to the imaging plane by the relationships

$$u = \frac{fx}{h} \tag{9.15}$$

$$v = \frac{fy}{h} \tag{9.16}$$

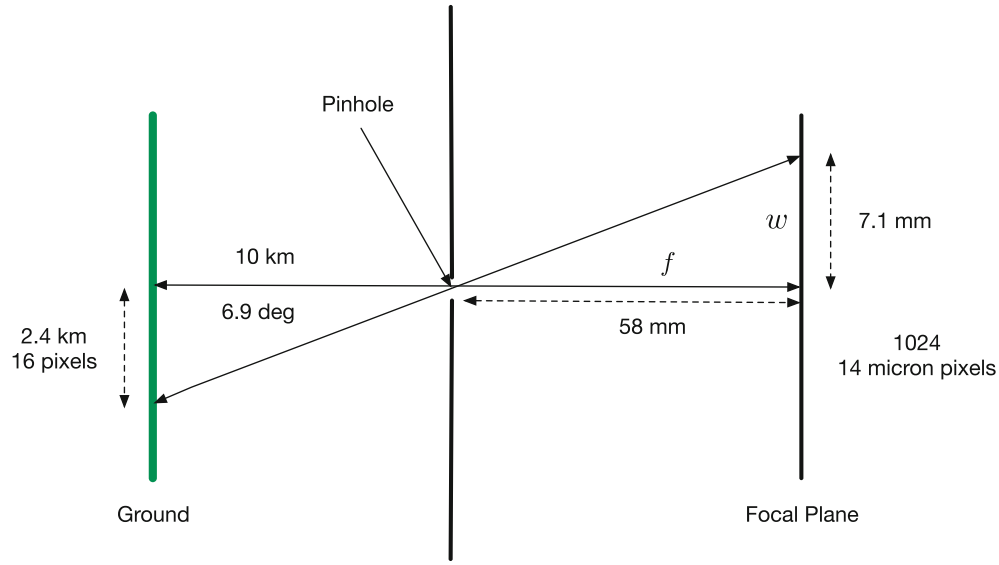
where u and v are coordinates in the focal plane, f is the focal length, and h is the distance from the pinhole to the point along the axis normal to the focal plane. This assumes that the z-axis of the coordinate frame x, y, z is aligned with the boresight of the camera. The angle that is seen by the imaging chip is

$$\theta = \tan^{-1}\left(\frac{w}{2f}\right) \tag{9.17}$$

where f is the focal length. The shorter the focal length, the larger the image. The pinhole camera does not have any depth of field, but that is unimportant for this far-field imaging problem. The field of view of a pinhole camera is limited only by the sensing element. Real cameras have lenses, and the images are not perfect across the imaging array. This presents practical problems that need to be solved in real machine vision systems.

We want our camera to see 16 pixels by 16 pixels from the terrain image in Figure 9.7. We will assume a flight altitude of 10 km. Figure 9.9 gives the dimensions.

Figure 9.9: Pinhole camera with dimensions.



We are not actually simulating any camera. Instead, our camera model is producing 16 by 16 pixel maps given an input of a position. The output is a data structure with the x and y coordinates and an image. If no inputs are given, it will create a tiled map of the image. We scaled the image in the GraphicConverter app so that it is exactly

```
   672    672      3
```

and saved it in the file `TerrainClose.jpg`. The numbers are x pixels, y pixels, and three layers for red, green, and blue. The third index is for the red, blue, and green matrices. This is a three-dimensional matrix, typical for color images.

The code is shown in the following. We convert everything to pixels, get the image using `[~,~,i] = getimage(h)`, and get the segment.

The first part of the code is to provide defaults for the user.

TerrainCamera.m

```
1  function d = TerrainCamera( r, h, nBits, w, nP )
2
3  % Demo
4  if( nargin < 1 )
5    Demo;
6    return
7  end
8
9  if( nargin < 3 )
10   nBits = [];
11 end
```

```
12
13  if( nargin < 4 )
14    w = [];
15  end
16
17  if( nargin < 5 )
18    nP = 64;
19  end
20
21  if( isempty(w) )
22    w = 4000;
23  end
24
25  if( isempty(nBits) )
26    nBits = 16;
27  end
```

The next part computes the pixels.

TerrainCamera.m

```
1  dW = w/nP;
2
3  k  = floor(r(1)/dW) + nP/4 + 1;
4  j  = floor((w/2-r(2))/dW) - nP/4 + 1;
5
6  kR = k:(k-1 + nBits);
7  kJ = j:(j-1 + nBits);
```

The remainder displays the image.

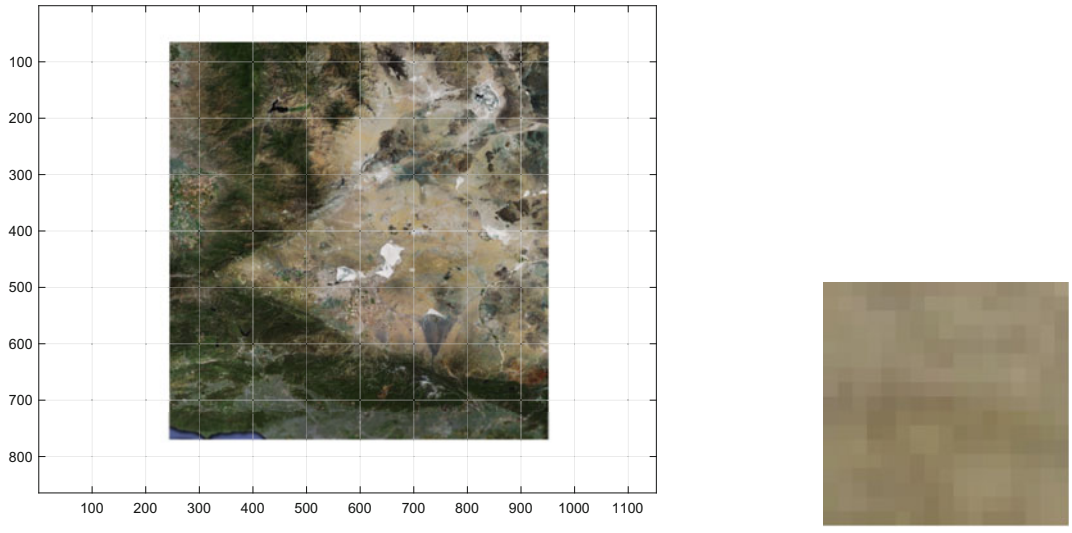
TerrainCamera.m

```
1  [~,~,i] = getimage(h);
2
3  d.p     = i(kR,kJ,:);
4  d.r     = r(1:2);
5
6  if( nargout < 1 )
```

The demo draws the source image and then the camera image. Both are shown in Figure 9.10.

```
7    axis off
8    axis image
9    clear p
10 end
11
12 %% CreateTerrain>Demo
13 function Demo
14
15 h = NewFigure('Earth Segment');
```

Figure 9.10: Terrain camera source image and camera view. The camera view is 16 × 16 pixels.




```
16 i = imread('TerrainClose64.jpg');
17 image(i);
18 grid
19
20 NewFigure('Terrain Camera');
21 x = linspace(0,10,20);
```

The terrain image from the camera is blurry because it has so few pixels.

9.6 Plot Trajectory over an Image

9.6.1 Problem

We want to plot our trajectory over an image.

9.6.2 Solution

Create a function to draw the image and plot the trajectory on top.

9.6.3 How It Works

We write a function that reads in an image and plots the trajectory on top. We scale the image using `image`. The x-dimension is set and the y-dimension is scaled to match.

PlotXYTrajectory.m

```
1 %% PLOTXYTRAJECTORY Draw an xy trajectory over an image
2 % Can plot multiple sets of data. Type PlotXYTrajectory for a demo.
3 %% Input
```

```
% x        (:,:) X coordinates (m)
% y        (:,:) Y coordinates (m)
% i        (n,m) Image
% w        (1,1) x dimension of the image
% xScale   (1,1) Scale of x dimension
% name     (1,:) Figure name

function PlotXYTrajectory( x, y, i, xScale, name )

if( nargin < 1 )
  Demo
  return
end

s    = size(i);
xI   = [-xScale xScale];
yI   = [-xScale xScale]*s(2)/s(1);

NewFigure(name);
image(xI,yI,flipud(i));
hold on
n = size(x,1);
for k = 1:n
  plot(x(k,:),y(k,:),'linewidth',2)
end
set(gca,'xlim',xI,'ylim',yI);
grid on
axis image
xlabel('x (m)')
ylabel('y (m)')
```

The demo draws a circle over our terrain image. This is shown in Figure 9.11.

```
%% PlotXYTrajectory>Demo
function Demo

i = imread('TerrainClose.jpg');
a = linspace(0,2*pi);
x = [30*cos(a);35*cos(a)];
y = [30*sin(a);35*sin(a)];
PlotXYTrajectory( x, y, i, 111, 'Trajectory' )
```

While the deep learning system will analyze all of the pixels in the image, it is interesting to see how the mean values of the pixels for each color vary for each image across the image. This is shown in Figure 9.12. The x-axis is the image number, going by rows of constant y. As can be seen, there is considerable variation even in nearby images. This indicates that there is sufficient information in each image for our deep learning system to be able to find locations. It also shows that it might be possible just to use mean values to identify location. Remember that each image varies from the previous by only 16 pixels.

Figure 9.11: Trajectory plot.

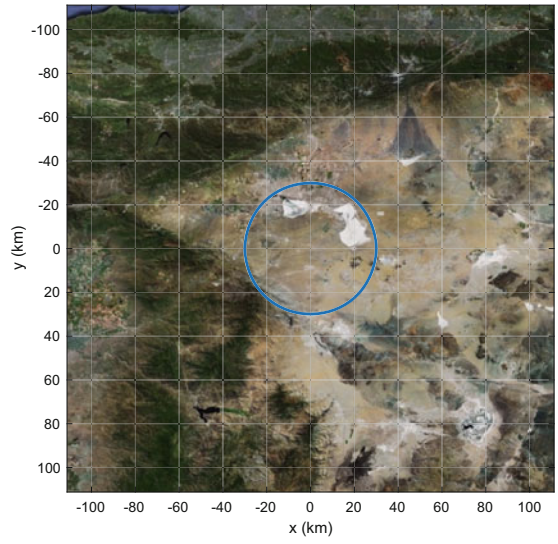
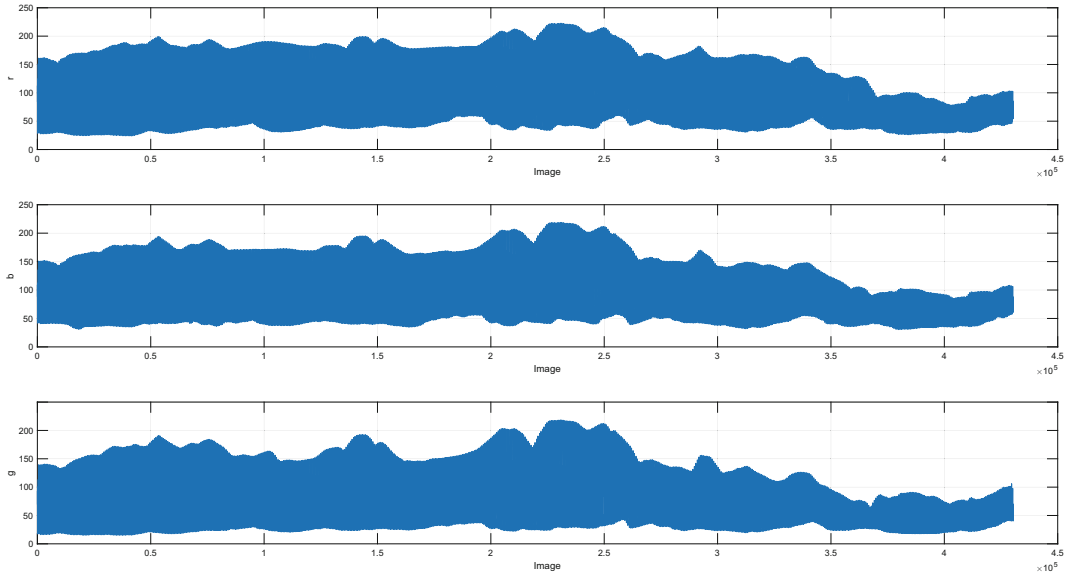


Figure 9.12: Mean red, green, and blue values for the images.



9.7 Creating the Test Images

9.7.1 Problem

We want to create test images for our terrain model.

9.7.2 Solution

We build a script to read in the 64 by 64 bit image and create training images.

9.7.3 How It Works

We first create a 64-bit version of our terrain, using any image processing app. We've already done that, and it is saved as `TerrainClose64.jpg`. The following script reads in the image and generates training images by displacing the index one pixel at a time. We save the images in the folder `TerrainImages`. We also create labels. Each image is a different label. For each terrain snippet, we create `nN` copies with noise. Thus there will be `nN` images with the label `l`. We add noise with the code

```
uint8(floor(sig*rand(nBits,nBits,3)))
```

since the noise must be `uint8` like the image. You'll get an error if you don't convert to `uint8`. You can also select different strides, that is, moving the images more than 1 pixel. The first code sets up the image processing. We choose 16-bit images because (after the next step of training) there is enough information in each image to classify each one. We tried 8 bits but it didn't converge.

CreateTerrainImages.m

```
im    = flipud(imread('TerrainClose64.jpg')); % Read in the image
wIm   = 4000; % m
nBits = 16;
dN    = 1; % The delta bits is 2
nBM1  = nBits-1;
[n,m] = size(im); % Size of the image
nI    = (n-nBits)/dN + 1; % The number of images down one side
nN    = 10;     % How many copies of each image we want
sig   = 3;      % Set to > 0 to add noise to the images
dW    = wIm/64; % Delta position for each image (m)
x0    = -wIm/2+(nBits/2)*dW;   % Starting location in the upper left
    corner
y0    =  wIm/2-(nBits/2)*dW;   % Starting location in the upper left
    corner
```

This line is very important. It makes sure the names correspond to distinct images. We will make copies of each image for training purposes.

CreateTerrainImages.m

```
% Make an image serial number so they remain in order in the
    imageDatastore
kAdd = 10^ceil(log10(nI*nI*nN));
```

We do some directory manipulations here.

CreateTerrainImages.m

```
% Set up the directory
if ~exist('TerrainImages','dir')
  warning('Are you in the right folder? No TerrainImages')
  [success,msg] = mkdir('./','TerrainImages')
end
cd TerrainImages
delete *.jpg % Starting from scratch so delete existing images
```

The image splitting is done in this code. We add noise, if desired.

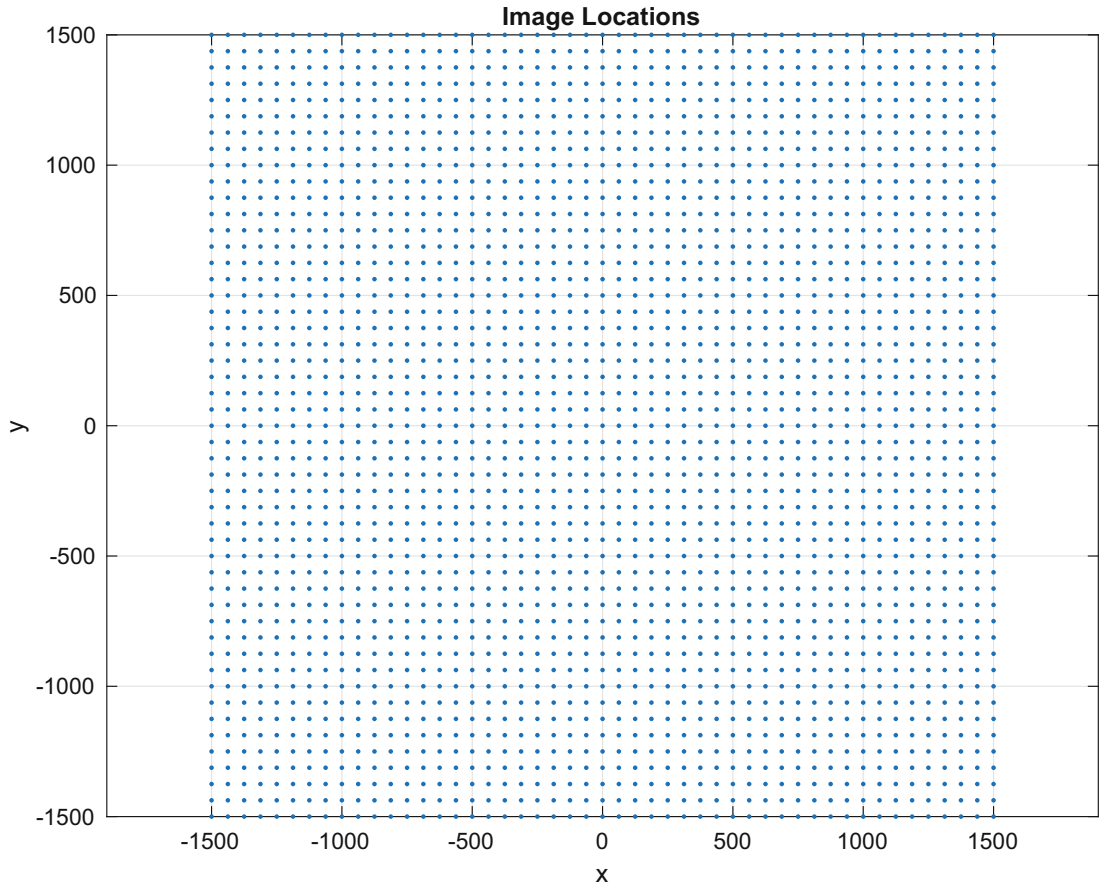
CreateTerrainImages.m

```
i   = 1;
l   = 1;
t   = zeros(1,nI*nI*nN); % The label for each image
x   = x0; % Initial location
y   = y0; % Initial location
r   = zeros(2,nI*nI); % The x and y coordinates of each image
id  = zeros(1,nI*nI);
iR  = 1;
rgbs = [];
for k = 1:nI
  disp(k)
  kR = dN*(k-1)+1:dN*(k-1) + nBits;
  for j = 1:nI
    kJ = dN*(j-1)+1:dN*(j-1) + nBits;
    thisImg = im(kR,kJ,:);
    rgbs(end+1,:) = [mean(mean(thisImg(:,:,1))) mean(mean(thisImg
        (:,:,2))) mean(mean(thisImg(:,:,3)))];
    for p = 1:nN
      s       = im(kR,kJ,:) + uint8(floor(sig*rand(nBits,nBits,3)));
      q       = s>256;
      s(q)    = 256;
      q       = s <0;
      s(q)    = 0;
      imwrite(s,sprintf('TerrainImage%d.jpg',i+kAdd));
      t(i)    = l;
      i       = i + 1;
    end
```

```
      r(:,iR)   = [x;y];
      id(iR)    = iR;
      iR = iR + 1;
      l = l + 1;
      y = y - dW;
    end
    x = x + dW;
    y = y0;
end
```

Figure 9.13 shows that the images really cover the area. We also verified that the sum of R, G, and B was different for each image. This indicates that there is enough information for the machine learning algorithm.

Figure 9.13: This figure shows that the images cover the landscape.



9.8 Training and Testing

9.8.1 Problem

We want to create and test a convolutional neural network. The neural net will be trained to associate images with an x and y location.

9.8.2 Solution

We create and test a convolutional neural network in `TerrainNeuralNet.m`. This will be trained on the images created earlier and will be able to return the x and y coordinates. Convolutional neural networks are widely used for image identification.

9.8.3 How It Works

This example is much like the one in Chapter 3. The difference is that each image is a separate category. This is like face identification where each category is a different person.

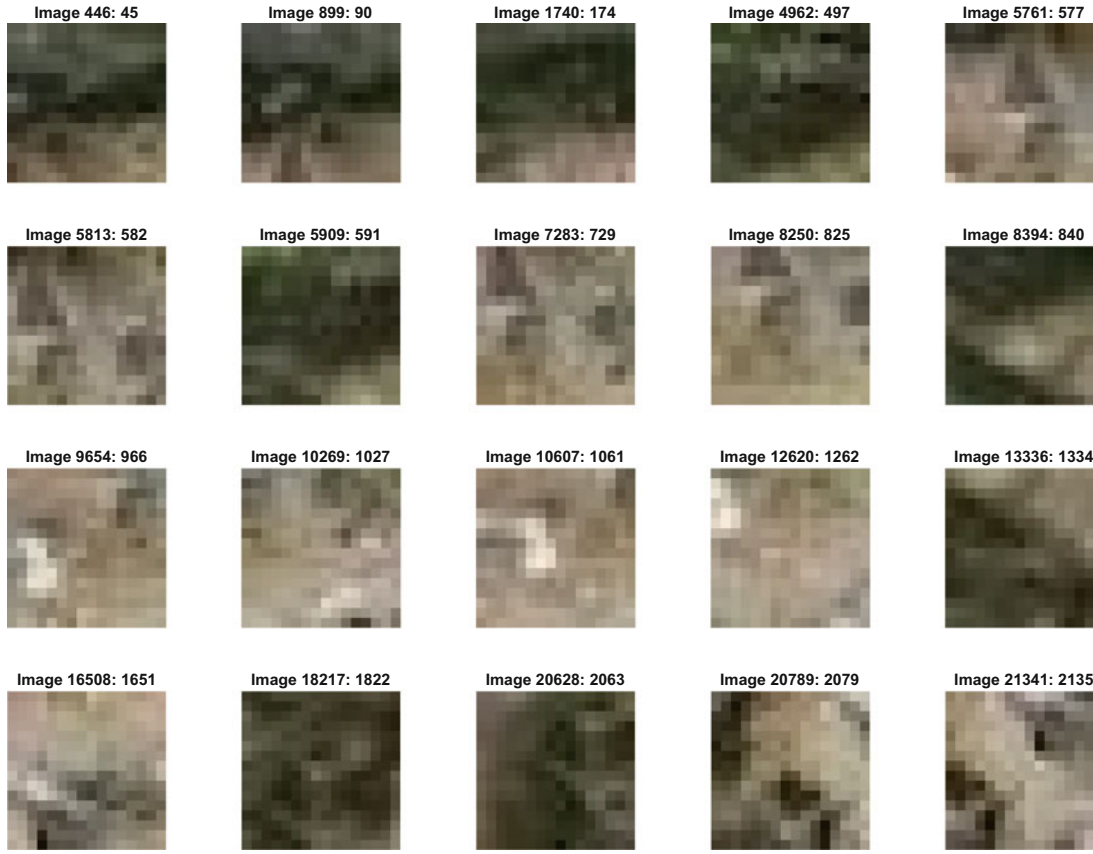
TerrainNeuralNet.m

```
%% Script implementing the terrain neural net
% You must have created the images in TerrainImages with
    CreateTerrainImages
% before running this script.

%% Get the images
cd TerrainImages
label = load('Label');
cd ..

t          = categorical(label.t);
nClasses   = max(label.t);
imds       = imageDatastore('TerrainImages','labels',t);
labelCount = countEachLabel(imds);

% Display a few snapshots
NewFigure('Terrain Snapshots');
n = 4;
m = 5;
ks = sort(randi(length(label.t),1,n*m)); % random selection
for i = 1:n*m
        subplot(n,m,i);
        imshow(imds.Files{ks(i)});
  title(sprintf('Image %d: %d',ks(i),label.t(ks(i))))
end

% We need the size of the images for the input layer
img = readimage(imds,1);

% Split into training and testing sets
fracTraining = 0.8;
[imdsTrain,imdsTest] = splitEachLabel(imds,fracTraining,'randomized');
```

```
%% Training
% This gives the structure of the convolutional neural net
layers = [
    imageInputLayer(size(img))

    convolution2dLayer(3,8,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,32,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,'Stride',2)

    fullyConnectedLayer(nClasses)
    softmaxLayer
    classificationLayer
        ];
disp(layers)

options = trainingOptions('sgdm', ...
    'InitialLearnRate',0.01, ...
    'MaxEpochs',6, ...
    'MiniBatchSize',100,...
    'ValidationData',imdsTest, ...
    'ValidationFrequency',10, ...
    'ValidationPatience',inf,...
    'Shuffle','every-epoch', ...
    'Verbose',false, ...
    'Plots','training-progress');
disp(options)
fprintf('Fraction for training %8.2f%%\n',fracTraining*100);

terrainNet = trainNetwork(imdsTrain,layers,options);

 %% Test the neural net
predLabels  = classify(terrainNet,imdsTest);
testLabels  = imdsTest.Labels;

accuracy = sum(predLabels == testLabels)/numel(testLabels);
fprintf('Accuracy is %8.2f%%\n',accuracy*100)

save('TerrainNet','terrainNet')
```

We have an image layer to read in each image. We next convolve them with filters. The weights of the filters are determined during the learning. We normalize the outputs and pass through the reLu activation function. Pooling compresses the data. Padding sets the output size equal to the

Figure 9.14: These selected terrain images show what the neural net is classifying.



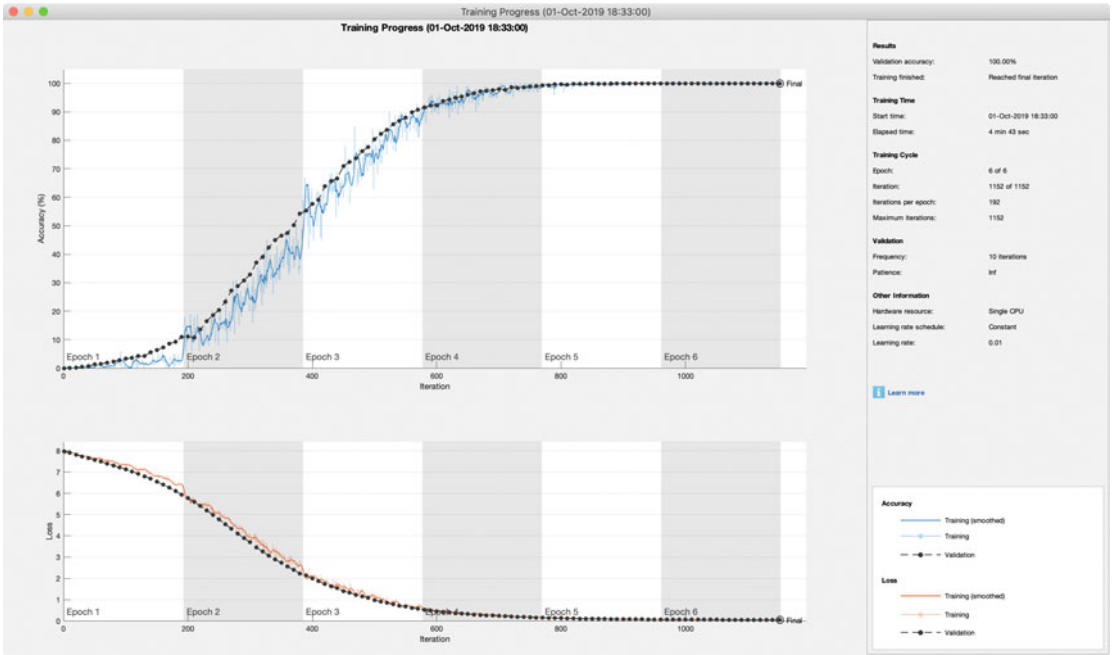
input size. As seen by the layers printout, no padding is needed since the images are all the same size. The first layer has eight 3 by 3 pixel filters. The second layer has 32 3 by 3 pixel filters. The final set of layers is used to classify the images. As noted in the previous section, each image has a unique ‘‘class” which is associated with its location. We use a constant learning rate. The batch size is smaller than the default.

Figure 9.14 shows some of the images. Figure 9.15 shows the training window. It is able to categorize the images after 7 epochs. The difference between two adjacent images is only 16 pixels. It isn’t a lot of data, but the neural net can categorize each image with 100% accuracy.

In each epoch in Figure 9.15, it is processing all of the training data.

```
>> TerrainNeuralNet
  12x1 Layer array with layers:

     1   ''   Image Input             16x16x3 images with 'zerocenter'
         normalization
     2   ''   Convolution             8 3x3 convolutions with stride [1
         1] and padding 'same'
```

Figure 9.15: Training window.




```
     3   ''   Batch Normalization     Batch normalization
     4   ''   ReLU                    ReLU
     5   ''   Max Pooling             2x2 max pooling with stride [2  2]
         and padding [0  0  0  0]
     6   ''   Convolution             32 3x3 convolutions with stride [1
          1] and padding 'same'
     7   ''   Batch Normalization     Batch normalization
     8   ''   ReLU                    ReLU
     9   ''   Max Pooling             2x2 max pooling with stride [2  2]
         and padding [0  0  0  0]
    10   ''   Fully Connected         2401 fully connected layer
    11   ''   Softmax                 softmax
    12   ''   Classification Output   crossentropyex
TrainingOptionsSGDM with properties:

                    Momentum: 0.9000
            InitialLearnRate: 0.0100
   LearnRateScheduleSettings: x[11 struct]
           L2Regularization: 1.0000e-04
     GradientThresholdMethod: 'l2norm'
           GradientThreshold: Inf
                   MaxEpochs: 6
               MiniBatchSize: 100
                     Verbose: 0
             VerboseFrequency: 50
```

```
                ValidationData: x[11 matlab.io.datastore.ImageDatastore]
           ValidationFrequency: 10
            ValidationPatience: Inf
                       Shuffle: 'every-epoch'
                CheckpointPath: ''
          ExecutionEnvironment: 'auto'
                    WorkerLoad: []
                     OutputFcn: []
                         Plots: 'training-progress'
                SequenceLength: 'longest'
          SequencePaddingValue: 0
          DispatchInBackground: 0
Fraction for training    80.00%
Accuracy is   100.00%
```

We get 100% accuracy. You can explore changing the number of layers and trying different activation functions.

9.9 Simulation

9.9.1 Problem

We want to test our deep learning algorithm using our terrain model.

9.9.2 Solution

We build a simulation using the trained neural net.

9.9.3 How It Works

We reproduce the simulation from the previous section and remove some unneeded output so that we can focus on the neural net. We read in the trained neural net.

AircraftNNSim.m

```
%% Load the neural net
```

The neural net classifies the image obtained by the camera. We convert the category into an integer using `int32`. The `subplot` displays the image the neural net identifies as matching the camera image and the camera image. The simulation loop stops if your altitude, `x(6)`, is less than 1.

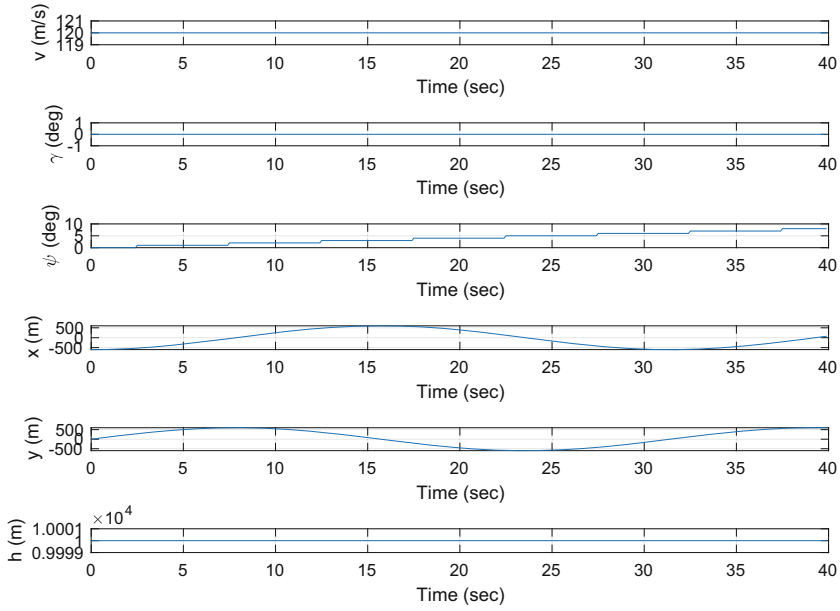
```
%% Start by finding the equilibrium controls
d     = RHSPointMassAircraft;
v     = 120;
d.phi = atan(v^2/(r*d.g));
x     = [v;0;0;-r;0;10000];
d     = EquilibriumControls( x, d );

%% Simulation
xPlot = zeros(length(x)+3,n);
```

```
43
44 % Put the image in a figure so that we can read it
45 h = NewFigure('Earth Segment');
46 i = imread('TerrainClose64.jpg');
47 image(i);
48 axis image
49
50 NewFigure('Camera');
51
52 for k = 1:n
53
54   % Get the image for the neural net
55   im          = TerrainCamera( x(4:5), h, nBits );
56
57   % Run the neural net
58   l           = classify(nN.terrainNet,im.p);
59
60   % Plot storage
61   i           = int32(l);
62   xPlot(:,k)  = [x;rI.r(:,i);i];
63
64   % Integrate
65   x           = RungeKutta( @RHSPointMassAircraft, 0, x, dT, d );
66
67   % A crash
68   if( x(6) <= 0 )
69     break;
70   end
71 end
72
73 %% Plot the results
74 xPlot         = xPlot(:,1:k);
75 xPlot(2,:)    = xPlot(2,:)*rTD;
76 xPlot(4:6,:)  = xPlot(4:6,:);
77 yL            = {'v (m/s)' '\gamma (deg)' '\psi (deg)' 'x (m)'  'y (m)'
    ...
78                  'h (m)' 'x_c (m)', 'y_c (m)' };
```

Figure 9.16 shows the trajectory and the camera view. We simulate one full circle.

The identified terrain segment and the path, based on the neural network location, are shown in Figure 9.17. The neural net classifies the terrain it is seeing. The location of each image is read out and used to plot the trajectory.

Figure 9.16: The camera view and trajectory. This is one full circle.



The 2D trajectory is shown in Figure 9.18 for a circular path. We make sure we are in the regions where each image is a one pixel change from the previous image. In the corners the camera would stay in one image until that image were exited. On the edges there is one image border. If we were in that region, the resolution would be low. The trajectory from the images is reasonably close to the actual trajectory. Better results would require higher resolution. In practice, the measured positions would be inputs to a Kalman filter [30] that modeled the aircraft dynamics, given earlier in this chapter. This would smooth the trajectory and improve accuracy.

This chapter shows how a neural network can be used to identify terrain for the purposes of aircraft navigation. We simplify things by flying at a constant altitude, use a pinhole camera model with a fixed image orientation and ignore clouds and other complications. We use a convolutional neural network to train the neural net with good results. As noted, higher resolution images and a Kalman filter would produce a smoother trajectory.

Figure 9.17: The identified terrain segments and the aircraft path.

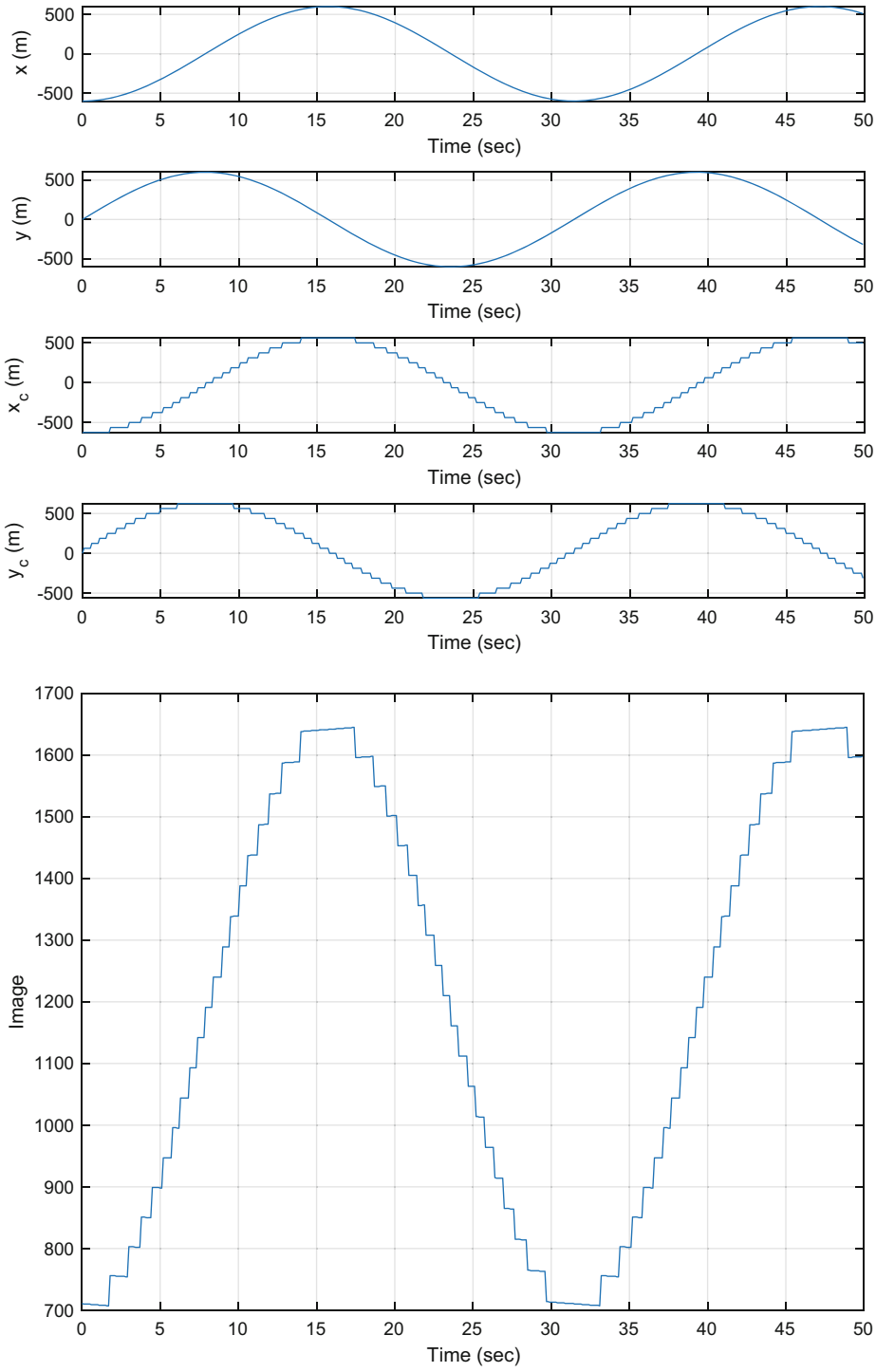
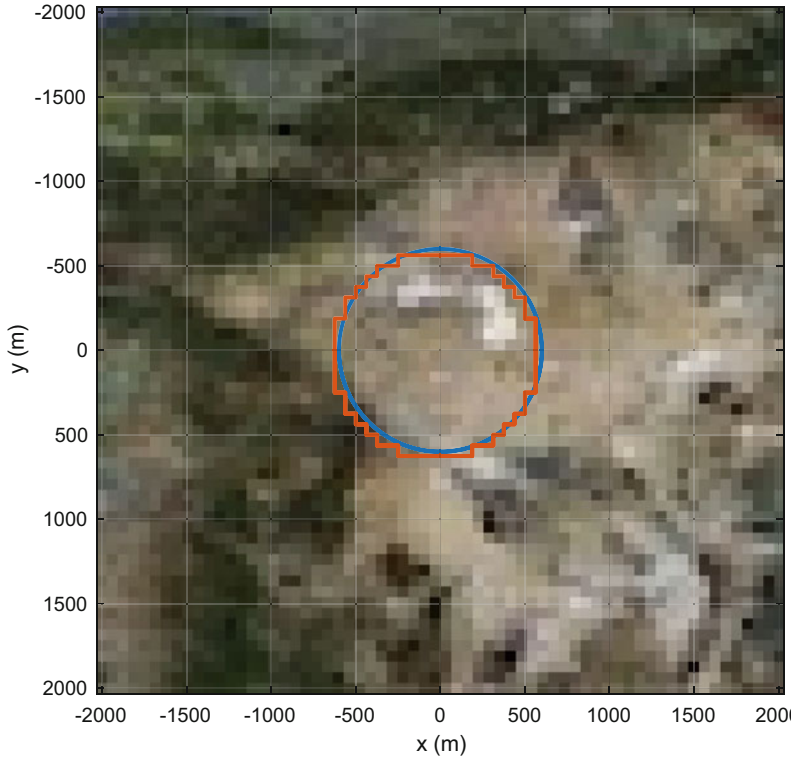


Figure 9.18: The identified terrain segments and the aircraft path.



CHAPTER 10

Stock Prediction

10.1 Introduction

The goal of a stock prediction algorithm is to recommend a portfolio of stocks that will maximize an investor's return. The investor has a finite amount of money and wants to create a portfolio to maximize her or his return on investment. The neural network in this chapter will predict the behavior of a stock given its history. This could then be used to select a portfolio of stocks with some idea of the future performance. The stock market model is based on Geometric Brownian Motion. Given that we could do statistical analysis that would allow us to pick stocks. We'll show that a neural net, which does not have any knowledge of model, can do as well in modeling the stocks.

10.2 Generating a Stock Market

10.2.1 Problem

We want to create an artificial stock market that replicates real stocks.

10.2.2 Solution

Implement Geometric Brownian Motion. This was invented by Paul Samuelson, Nobel Laureate [25].

10.2.3 How It Works

Paul Samuelson [10] created a stock model based on Geometric Brownian Motion. This approach produces realistic numbers and will not go negative. This is effectively a random walk in log-space. The stochastic differential equation is

$$dS(t) = rSdt + \sigma SdW(t) \tag{10.1}$$

S is the stock price. $W(t)$ is a Brownian, random walk, process. t is the time and dt is the time differential. r is the drift and σ is the volatility. Both range from 0 to 1. It could also be written

M. Paluszek and S. Thomas, *Practical MATLAB Deep Learning*,
https://doi.org/10.1007/978-1-4842-5124-9_10

in differential equation form as

$$\frac{dS}{dt} = \left(r + \sigma \frac{dW(t)}{dt} \right) S \tag{10.2}$$

The solution is

$$S(t) = S(0) e^{\left[(r - \frac{1}{2}\sigma^2)t + \sigma W(t) \right]} \tag{10.3}$$

The following shows the code used to generate the stock trends. We use `cumsum` to sum the random numbers for the random walk. We use a Gaussian or normal distribution produced by `randn` to create the random numbers. The function can create multiple stocks.

StockPrice.m

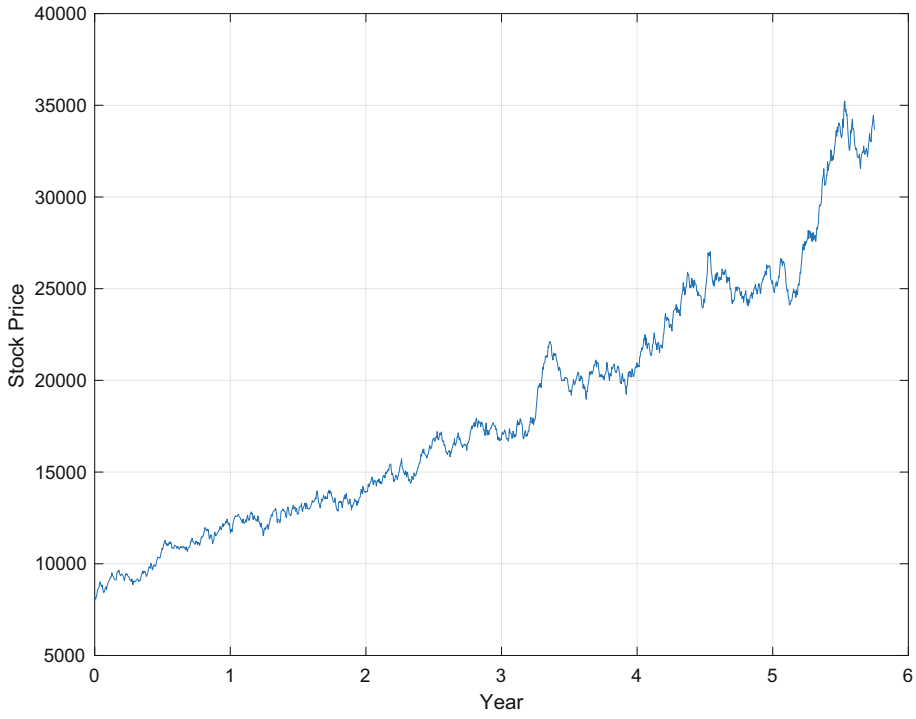
```
function [s, t] = StockPrice( s0, r, sigma, tEnd, nInt )

if( nargin < 1 )
  Demo
  return
end

delta   = tEnd/nInt;
sDelta  = sqrt(delta);
t       = linspace(0,tEnd,nInt+1);
m       = length(s0);
w       = [zeros(m,1) cumsum(sDelta.*randn(m,nInt))];
s       = zeros(1,nInt+1);
f       = r - 0.5*sigma.^2;
for k = 1:m
  s(k,:) = s0(k)*exp(f(k)*t + sigma(k)*w(k,:));
end
```

The demo is based on the Wilshire 5000 statistics. It is an index of all US stocks. If you run it, you will get different values since the input is random.

```
%% StockPrice>Demo
function Demo

tEnd  = 5.75;
n     = 1448;
s0    = 8242.38;
r     = 0.1682262;
sigma = 0.1722922;
StockPrice( s0, r, sigma, tEnd, n );
```

The results are shown in Figure 10.1. They look like a real stock. Changing the drift or volatility will change the overall shape. For example, if you set the volatility, $\sigma = 0$, you get the very nice stock shown in Figure 10.2. Increasing r makes the stock grow faster. This gives us the general rule that we want high r and low σ. See Figure 10.3.

Figure 10.1: A random stock based on statistics from the Wilshire 5000.



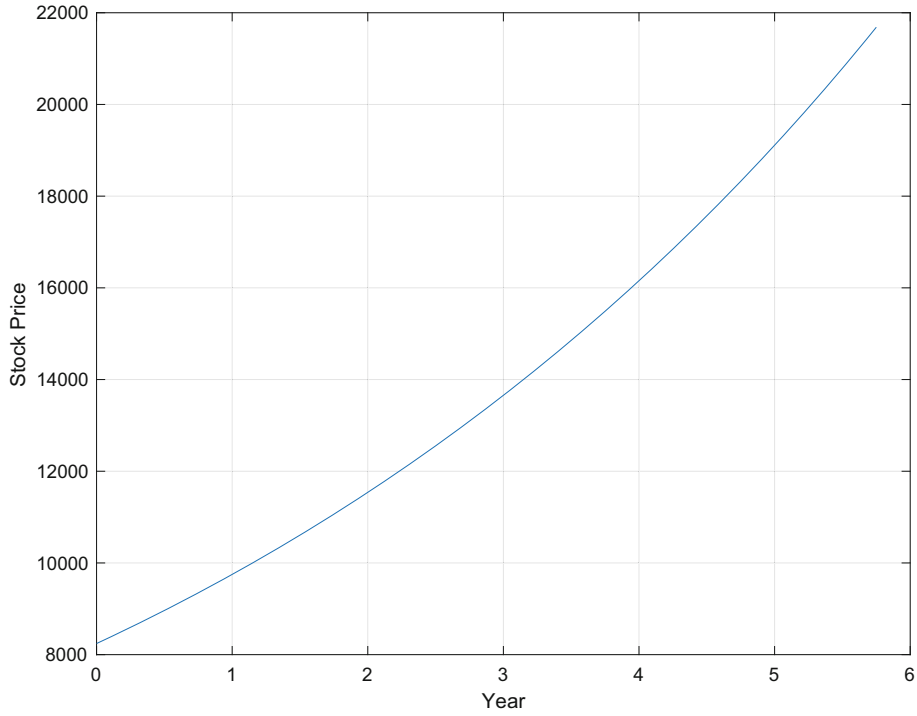
Our model is based on two coefficients. We could make a stock picking algorithm by just fitting stock price curves and computing r and σ. However, we want to see how well Deep Learning does. Remember, this is a simple model of stock prices. Both σ and r could also be functions of time, or random variables by themselves. Of course, there are other stock models too! The idea here is that deep learning creates its own internal model without a need to be told about the model underlying the observed data.

The function `PlotStock.m` plots the stock price. Notice that we format the y tick labels ourselves to get rid of the exponential format that MATLAB would normally employ. `gca` returns the current axes handle.

PlotStock.m

```
function PlotStock(t,s,symb)

if( nargin < 1 )
  Demo;
  return;
end

```

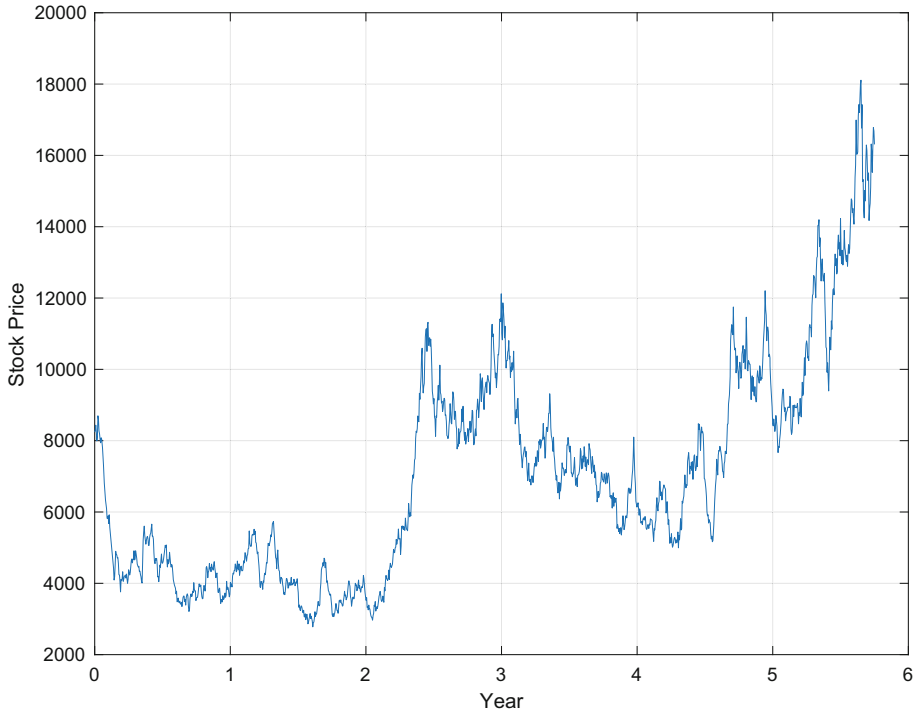
Figure 10.2: A stock with zero volatility. This is a good stock to own, though the index fund isn't too bad either.




```
8  m = size(s,1);
9
10 PlotSet(t,s,'x label','Year','y label','Stock Price','figure title',...
11     'Stocks','Plot Set',{1:m},'legend',{symb});
12
13 % Format the ticks
14 yT  = get(gca,'YTick');
15 yTL = cell(1,length(yT));
16 for k = 1:length(yT)
17         yTL{k} = sprintf('%5.0f',yT(k));
18 end
19 set(gca,'YTickLabel', yTL)
```

The built-in demo of PlotStock is the same as in `StockPrice`.

Figure 10.3: A stock with high volatility and low drift such that $r - \frac{1}{2}\sigma^2 < 1$. In this case, $r = 0.1$ and $\sigma = 0.6$.




```
20  function Demo
21
22  tEnd  = 5.75;       % years
23  nInt  = 1448;       % intervals
24  s0    = 8242.38;    % initial price
25  r     = 0.1682262; % drift
26  sigma = 0.1722922;
27  [s,t] = StockPrice( s0, r, sigma, tEnd, nInt );
28  PlotStock(t,s,{})
```

10.3 Create a Stock Market

10.3.1 Problem

We want to create a stock market.

10.3.2 Solution

Use the stock price function to create 100 stocks with randomly chosen parameters.

10.3.3 How It Works

We write a function that randomly picks stock starting prices, volatility, and drift. It also creates random three letter stock names. We use a half normal distribution for the stock prices. This code generates the random market. We limit the drift to between 0 and 0.5. This creates more stocks (for small markets) that go down.

StockMarket.m

```
function d = StockMarket(  nStocks, s0Mean, s0Sigma, tEnd, nInt )

if( nargin < 1 )
  Demo
  return
end

d.s0    = abs(s0Mean + s0Sigma*randn(1,nStocks));
d.r     = 0.5*rand(1,nStocks);
d.sigma = rand(1,nStocks);
s       = 'A':'Z';
for k = 1:nStocks
  j            = randi(26,1,3);
  d.symb(k,:)  = s(j);
end
```

The following code plots all of the stocks on one plot. We create a legend and make the y labels integers (using `PlotStock`).

```
% Output
if( nargout < 1 )
  s = StockPrice( d.s0, d.r, d.sigma, tEnd, nInt );
  t     = linspace(0,tEnd,nInt+1);
  PlotStock(t,s,d.symb);
  clear d
end
```

The demo is shown as follows

```
%% StockPrice>Demo
function Demo

nStocks = 15;    % number of stocks
s0Mean  = 8000;  % Mean stock price
s0Sigma = 3000;  % Standard  dev of price
tEnd    = 5.75;  % years duration for market
nInt    = 1448;  % number of intervals
StockMarket( nStocks, s0Mean, s0Sigma, tEnd, nInt );
```

Figure 10.4: Two runs of random five stock markets.

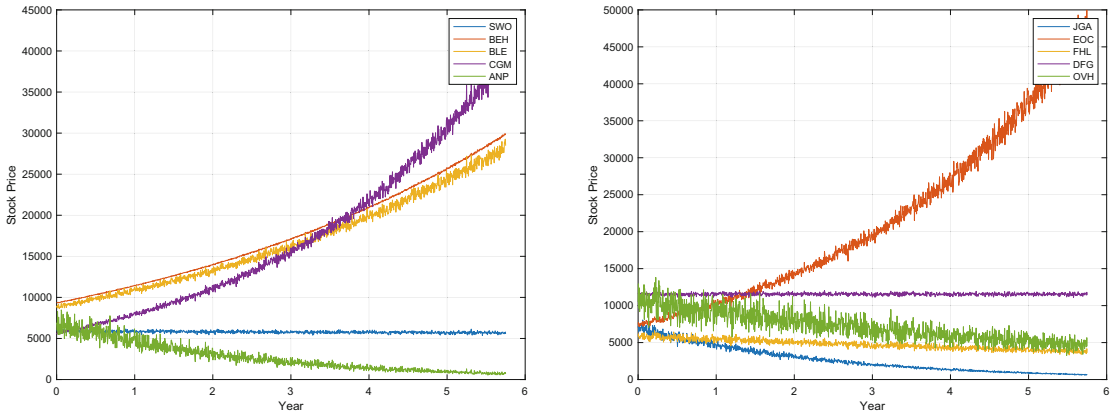
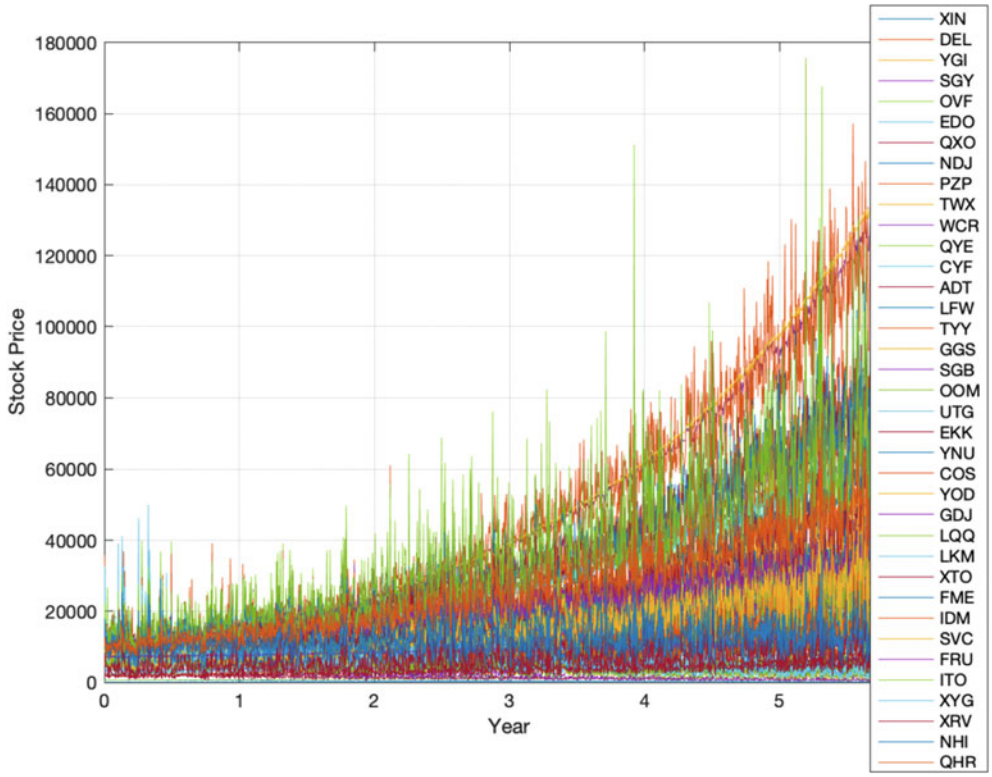


Figure 10.5: Hundred stock market.



Two runs are shown in Figure 10.4.
A stock market with a hundred stocks is shown in Figure 10.5.

10.4 Training and Testing

10.4.1 Problem

We want to build a deep learning system to predict the performance of a stock. This can be applied to the stock market created earlier to predict the performance of a portfolio.

10.4.2 Solution

The history of a stock is a time series. We will use a long short-term memory (LSTM) network to predict the future performance of the stock based on past data. Past performance is not necessarily indicative of future results. All investments carry some amount of risk. You are encouraged to consult with a certified financial planner prior to making any investment decisions. This utilizes the deep learning toolbox's `lstmLayer` layer. We will use part of the time series to test the results.

10.4.3 How It Works

An LSTM layer learns long-term dependencies between time steps in a time series. It automatically deweights past data. LSTMs have replaced recursive neural nets (RNNs) in many applications.

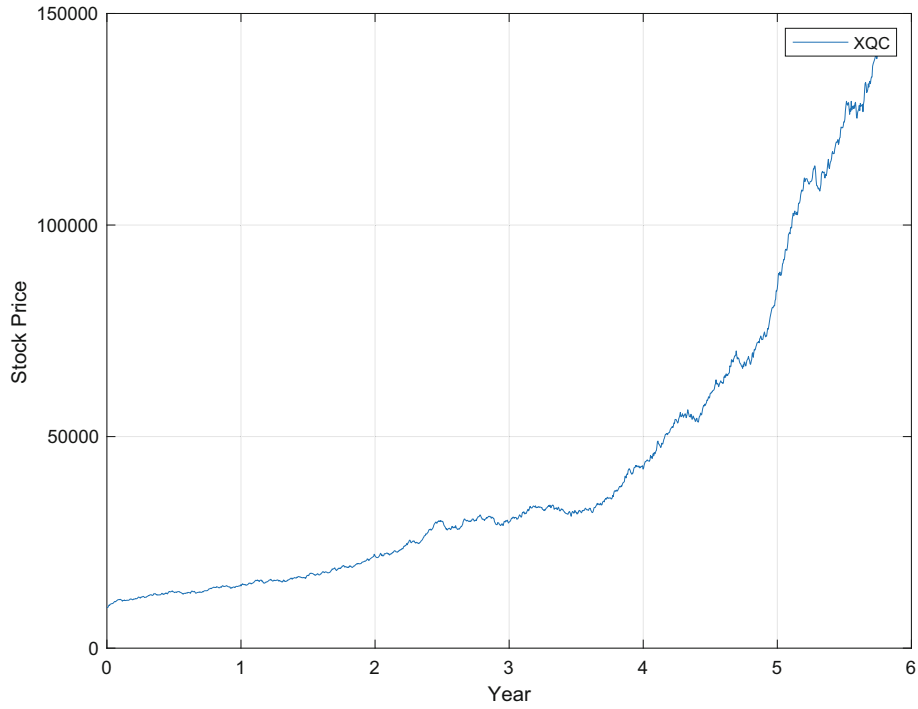
`StockMarketNeuralNet` implements the neural network. The first part creates a market with a single stock. We set the random number generator to its default value, `rng('default')`, so that every time you run the script, you will get the same result. If you remove this line, you will get different results each time. The neural network training data is the time sequence and the time sequence shifted by one time step.

StockMarketNeuralNet.m

```
%% Script using LSTM to predict stock prices
%% See also:
% lstmLayer, sequenceInputLayer, fullyConnectedLayer, regressionLayer,
% trainingOptions, trainNetwork, predictAndUpdateState

% Rest the random number generator so we always get the same case
rng('default')

layerSet = 'two lstm'; % 'lstm' 'bilstm' and 'two lstm' are available

%% Generate the stock market example
n     = 1448;
tEnd  = 5.75;
d     = StockMarket( 1, 8000, 3000, tEnd, n );
s     = StockPrice( d.s0, d.r, d.sigma, tEnd, n );
t     = linspace(0,tEnd,n+1);

PlotStock(t,s,d.symb);
```

The stock price is shown in Figure 10.6. We divide the outputs into training and testing data. We used the testing data for validation.

Figure 10.6: A stock price.



StockMarketNeuralNet.m

```
%% Divide into training and testing data
n             = length(s);
nTrain        = floor(0.8*n);
sTrain        = s(1:nTrain);
sTest         = s(nTrain+1:n);
sVal          = sTest;

% Normalize the training data
mu            = mean(sTrain);
sigma         = std(sTrain);

sTrainNorm    = (sTrain-mu)/sigma; % normalize the data to zero mean

% Normalize the test data
sTestNorm     = (sTest - mu) / sigma;
sTest         = sTestNorm(1:end-1);
```

The next part trains the network. We use the ‘‘Adam” method [17]. Adam is a first-order gradient-based optimization of stochastic objective functions. It is computationally efficient and works well with problems with noisy or sparse gradients. See the reference for more details. We have a four-layer network including an LSTM layer.

```
%% Train the neural net

% We are training the LSTM using the previous step
xTrain    = sTrainNorm(1:end-1);
yTrain    = sTrainNorm(2:end);

% Validation data
muVal     = mean(sVal); % Must normalize over just this data
sigmaVal        = std(sVal);
sValNorm  = (sVal-muVal)/sigmaVal;

xVal      = sValNorm(1:end-1);
yVal      = sValNorm(2:end);

numFeatures     = 1;
numResponses    = 1;
numHiddenUnits  = 200;

switch layerSet
  case 'lstm'
    layers = [sequenceInputLayer(numFeatures)
              lstmLayer(numHiddenUnits)
              fullyConnectedLayer(numResponses)
              regressionLayer];
  case 'bilstm'
    layers = [sequenceInputLayer(numFeatures)
              bilstmLayer(numHiddenUnits)
              fullyConnectedLayer(numResponses)
              regressionLayer];
  case 'two lstm'
    layers = [sequenceInputLayer(numFeatures)
              lstmLayer(numHiddenUnits)
              reluLayer
              lstmLayer(numHiddenUnits)
              fullyConnectedLayer(numResponses)
              regressionLayer];
  otherwise
    error('Only 3 sets of layers are available');
end

analyzeNetwork(layers);

options = trainingOptions('adam', ...
    'MaxEpochs',300, ...
    'GradientThreshold',1, ...
    'InitialLearnRate',0.005, ...
    'LearnRateSchedule','piecewise', ...
    'LearnRateDropPeriod',125, ...
    'LearnRateDropFactor',0.2, ...
    'Shuffle','every-epoch', ...
    'ValidationData',{xVal,yVal}, ...
```

```
88      'ValidationFrequency',5, ...
89      'Verbose',0, ...
90      'Plots','training-progress');
91
92  net = trainNetwork(xTrain,yTrain,layers,options);
```

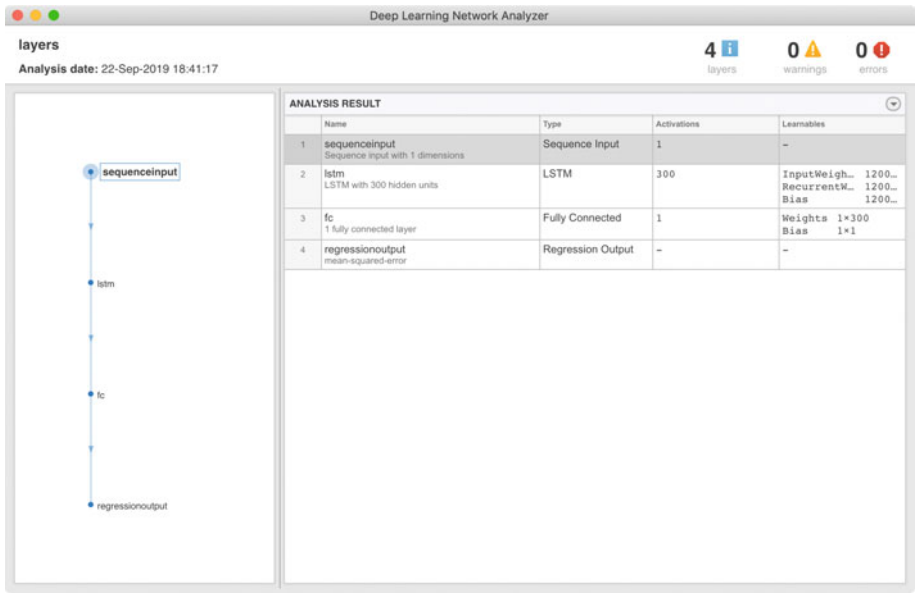
The neural net consists of four layers:

```
layers = [sequenceInputLayer(numFeatures)
          lstmLayer(numHiddenUnits)
          fullyConnectedLayer(numResponses)
          regressionLayer];
```

This is the minimum set of layers. The layer structure is shown by `analyzeNetwork` in Figure 10.7. `analyzeNetwork` isn't too interesting for such a simple structure. It is more interesting when you have dozens or hundreds of layers. We also provide the option to try a BiLSTM layer and two LSTM layers.

1. `sequenceInputLayer(inputSize)` defines a sequence input layer. `inputSize` is the size of the input sequence at each time step. In our problem, the sequence is just the last value in the time sequence so `inputSize` is 1. You could have longer sequences.
2. `lstmLayer(numHiddenUnits)` creates a long short-term memory layer. numHiddenUnits is the number of hidden units in the layer. The number of hidden units is the number of neurons in the layer.
3. `fullyConnectedLayer` creates a fully connected layer with specified output size.

Figure 10.7: Layer structure.



4. `regressionLayer` creates a regression output layer for a neural network. Regression is data fitting.

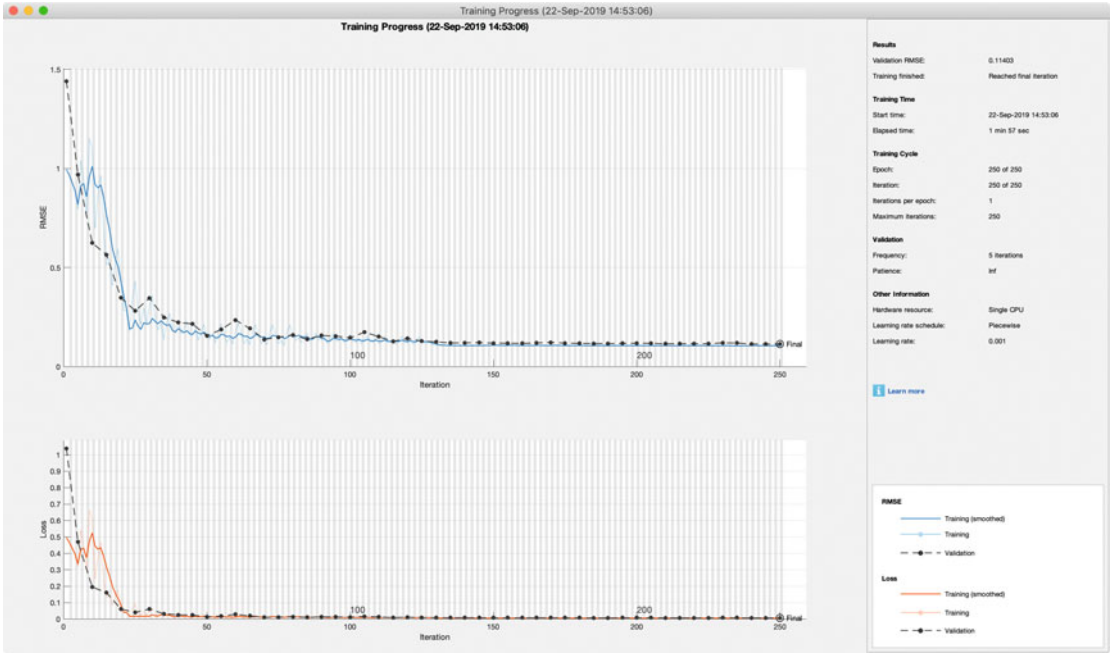
The learn rate starts with 0.005. It is decreased by a factor of 0.2 every 125 epochs in a piecewise manner using these options:

```
'InitialLearnRate',0.005, ...
'LearnRateSchedule','piecewise', ...
'LearnRateDropPeriod',125, ...
'LearnRateDropFactor',0.2, ...
```

We let ''patience" be `inf`. This means the learning will continue to the last epoch even if no progress is made. The training window is shown in Figure 10.8. The top plot shows the root-mean-square error (RMSE) calculated from the data and the bottom plot the loss. We are also using the test data for validation. Note that the validation data needs to be normalized with its own mean and standard deviation.

The final part tests the network using `predictAndUpdateState`. We need to unnormalize the output for plotting.

Figure 10.8: The training window with 250 iterations.



```
%% Demonstrate the neural net
yPred     = predict(net,sTest);
yPred(1)  = yTrain(end-1);
yPred(2)  = yTrain(end);
yPred     = sigma*yPred + mu;

%% Plot the prediction
NewFigure('Stock prediction')
plot(t(1:nTrain-1),sTrain(1:end-1));
hold on
plot(t,s,'--g');
grid on
hold on
k = nTrain+1:n;
plot(t(k),[s(nTrain) yPred],'-')
xlabel("Year")
ylabel("Stock Price")
title("Forecast")
legend(["Observed" "True" "Forecast"])

% Format the ticks
yT  = get(gca,'YTick');
yTL = cell(1,length(yT));
```

Compare Figure 10.9 with Figure 10.6. The red is the prediction. The prediction reproduces the trend of the stock. It gives you an idea of how it might perform. The neural network cannot predict the exact stock history but does recreate the overall performance that is expected.

Results for the BiLSTM layer and two LSTM layers are shown in Figure 10.10. All produce acceptable models.

This chapter demonstrates that an LSTM can produce an internal model that replicates the behavior of a system from just observations of the process. In this case, we had a model, but in many systems, a model does not exist or has considerable uncertainty in its form. For this reason, neural nets can be a powerful tool when working with dynamical systems. We haven't tried this with real stocks. Do not use this for predicting real stock performance.

Figure 10.9: The prediction with one LSTM layer.

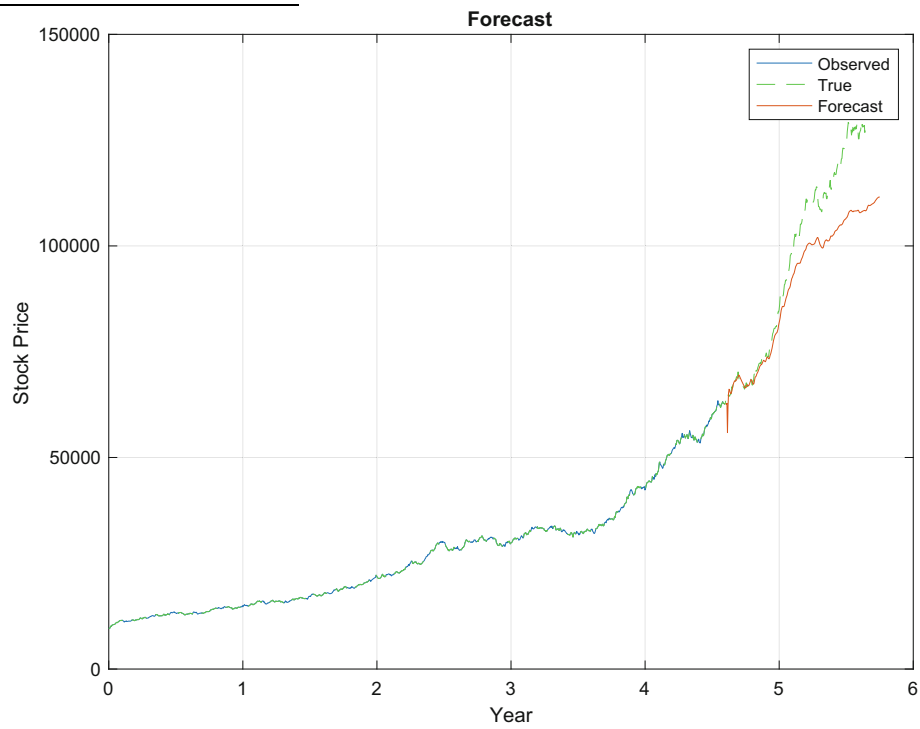
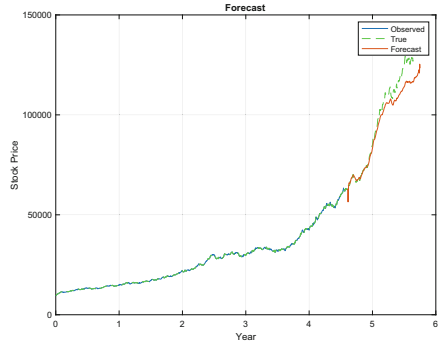
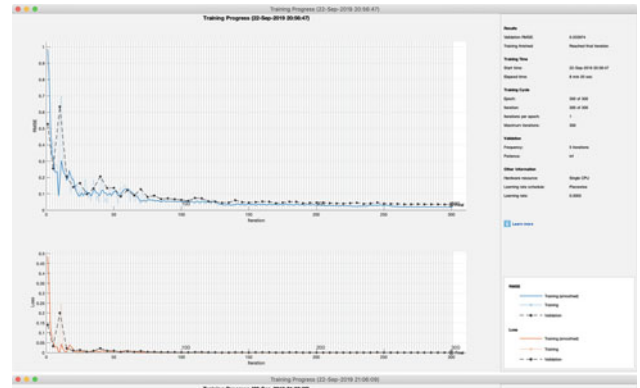
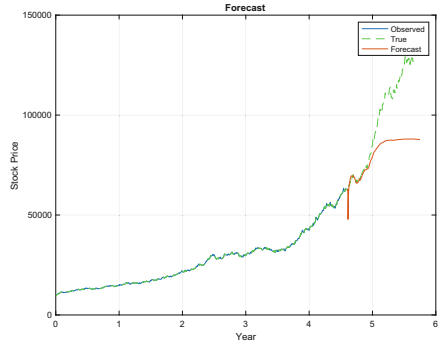
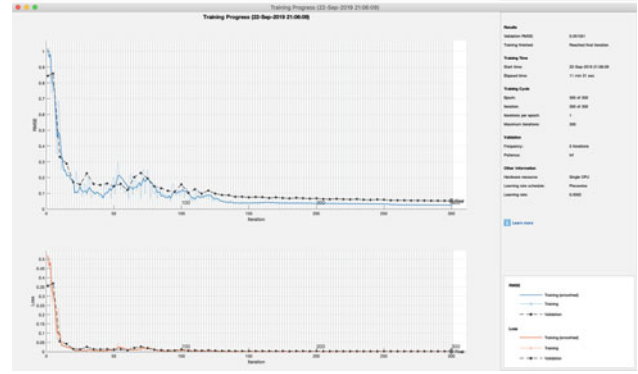


Figure 10.10: The top sets are the BiLSTM set and the bottom are the two LSTM layer sets.









CHAPTER 11

Image Classification

11.1 Introduction

Image classification can be done with pretrained networks. MATLAB makes it easy to access and use these networks. This chapter shows you two examples.

11.2 Using a Pretrained Network

11.2.1 Problem

We want to use a pretrained network for image identification. First we will use AlexNet, then GoogLeNet.

11.2.2 Solution

Install AlexNet and GoogLeNet from the Add-On Explorer. Load some images and test. These are classification networks so we will use `classify` to run them.

11.2.3 How It Works

First we need to download the support packages with the Add-On Explorer. If you attempt to run `alexnet` or `googlenet` without having them installed, you will get a link directly to the package in the Add-On Explorer. You will need your MathWorks password.

AlexNet is a pretrained convolutional neural network (CNN) that has been trained on approximately 1.2 million images from the ImageNet data set (http://image-net.org/index). The model has 23 layers and can classify images into 1000 object categories. It can be used for all sorts of object identification. However, if an object was not in the training, it won't be able to identify the object.

AlexNetTest.m

```
%% Load the network
% Access the trained model. This is a SeriesNetwork.
net = alexnet;
```


M. Paluszek and S. Thomas, *Practical MATLAB Deep Learning*,
https://doi.org/10.1007/978-1-4842-5124-9_11

```
net

% See details of the architecture
net.Layers
```

The network layers' printout is shown as follows.

```
>> AlexNetTest

ans =

  25x1 Layer array with layers:

     1   'data'     Image Input                   227x227x3 images with 'zerocenter' normalization
     2   'conv1'    Convolution                   96 11x11x3 convolutions with stride [4  4] and padding [0
                    0  0  0]
     3   'relu1'    ReLU                          ReLU
     4   'norm1'    Cross Channel Normalization   cross channel normalization with 5 channels per element
     5   'pool1'    Max Pooling                   3x3 max pooling with stride [2  2] and padding [0  0  0  0]
     6   'conv2'    Grouped Convolution           2 groups of 128 5x5x48 convolutions with stride [1  1] and
                    padding [2  2  2  2]
     7   'relu2'    ReLU                          ReLU
     8   'norm2'    Cross Channel Normalization   cross channel normalization with 5 channels per element
     9   'pool2'    Max Pooling                   3x3 max pooling with stride [2  2] and padding [0  0  0  0]
    10   'conv3'    Convolution                   384 3x3x256 convolutions with stride [1  1] and padding [1
                    1  1  1]
    11   'relu3'    ReLU                          ReLU
    12   'conv4'    Grouped Convolution           2 groups of 192 3x3x192 convolutions with stride [1  1] and
                    padding [1  1  1  1]
    13   'relu4'    ReLU                          ReLU
    14   'conv5'    Grouped Convolution           2 groups of 128 3x3x192 convolutions with stride [1  1] and
                    padding [1  1  1  1]
    15   'relu5'    ReLU                          ReLU
    16   'pool5'    Max Pooling                   3x3 max pooling with stride [2  2] and padding [0  0  0  0]
    17   'fc6'      Fully Connected               4096 fully connected layer
    18   'relu6'    ReLU                          ReLU
    19   'drop6'    Dropout                       50% dropout
    20   'fc7'      Fully Connected               4096 fully connected layer
    21   'relu7'    ReLU                          ReLU
    22   'drop7'    Dropout                       50% dropout
    23   'fc8'      Fully Connected               1000 fully connected layer
    24   'prob'     Softmax                       softmax
    25   'output'   Classification Output         crossentropyex with 'tench' and 999 other classes
```

There are many layers in this convolutional network. ReLU and Softmax are the activation functions. In the first layer, ''zerocenter" normalization is used. This means the images are normalized to have a mean of zero and a standard deviation of 1. Two layers are new, cross-channel normalization and grouped Convolution. Filter groups, also known as grouped convolution, were introduced with AlexNet in 2012. You can think of the output of each filter as a channel and filter groups as groups of the channels. Filter groups allowed more efficient parallelization across GPUs. They also improved performance. Cross-channel normalization normalizes across channels, instead of one channel at a time. We've discussed convolution in Chapter 3. The weights in each filter are determined during training. Dropout is a layer that ignores nodes, randomly, when training the weights. This prevents interdependencies between nodes.

For our first example, we load an image that comes with MATLAB, of a set of peppers. We use the top left corner as input to the net. Note that each pretrained network has a fixed input image size that we can determine from the first layer.

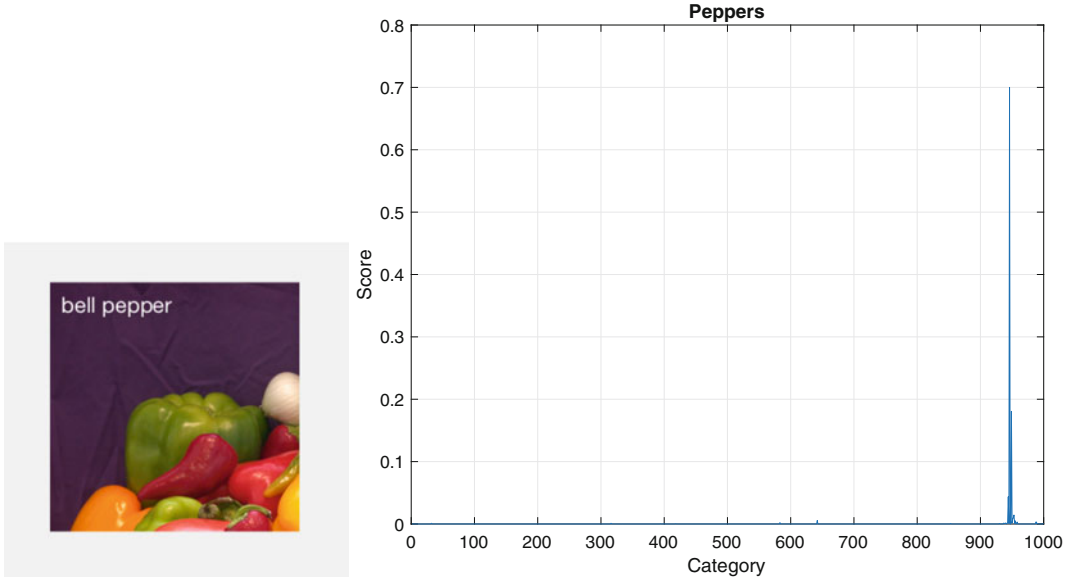
AlexNetTest.m

```
%% Load a test image and classify it
% Read the image to classify
I = imread('peppers.png');  % ships with MATLAB

% Adjust size of the image to the net's input layer
sz = net.Layers(1).InputSize;
I = I(1:sz(1),1:sz(2),1:sz(3));

% Classify the image using AlexNet
[label, scorePeppers] = classify(net, I);

%  Show the image and the classification results
NewFigure('Pepper'); ax = gca;
imshow(I);
title(ax,label);

PlotSet(1:length(scorePeppers),scorePeppers,'x label','Category',...
        'y label','Score','plot title','Peppers');
```

The images and results for the AlexNet example are shown in Figure 11.1. The pepper scores are tightly clustered.

For fun, and to learn more about this network, we print out the categories that had next highest scores, sorted from high to low. The categories are stored in the last layer of the net in its `Classes`.

Figure 11.1: Test image labeled with the classification and the scores. The image is classified as a ''bell pepper.''



AlexNetTest.m

```
% What other categories are similar?
disp('Categories with highest scores for Peppers:')
kPos = find(scorePeppers>0.01);
[vals,kSort] = sort(scorePeppers(kPos),'descend');
for k = 1:length(kSort)
  fprintf('%13s:\t%g\n',net.Layers(end).Classes(kPos(kSort(k))),vals(k)
      );
end
```

The results show that the net was considering all fruits and vegetables! The Granny Smith had the next highest scores, followed by cucumber, while the fig and lemon had much smaller scores. This makes sense since Granny Smiths and cucumbers are also usually green.

```
Categories with highest scores for Peppers:

   bell pepper:  0.700013
  Granny Smith:  0.180637
      cucumber:  0.0435253
           fig:  0.0144056
         lemon:  0.0100655
```

We also have two of our own test images. One is of a cat and one of a metal box, shown in Figure 11.2.

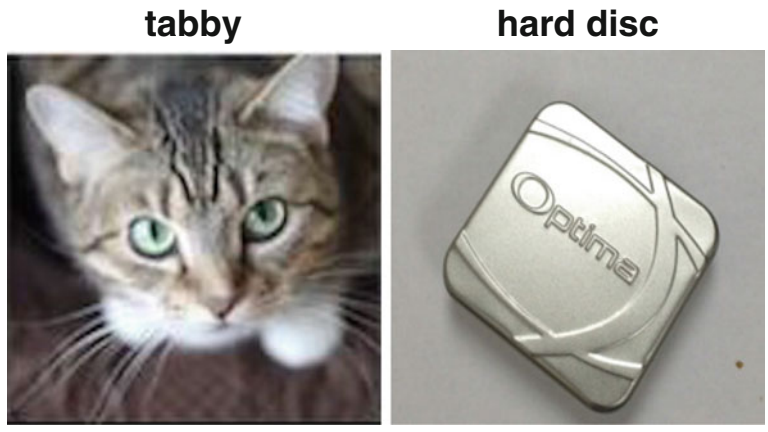
The scores for the cat classification are shown as follows.

```
Categories with highest scores for Cat:
                tabby:   0.805644
         Egyptian cat:   0.15372
            tiger cat:   0.0338047
```

The selected label is *tabby*. It is clear that the net can recognize that the photo is of a cat, as the other highest scored categories are also kinds of cats. Although what a tiger cat might be, as distinguished from a tabby, we can't say...

Figure 11.2: Raw test images Cat.png and Box.jpg.

Figure 11.3: Test images and the classification by AlexNet. They are classified as ‘‘tabby” and ‘‘hard disc.”



The metal box proves the biggest challenge to the net. The category scores above 0.05 are shown as follows and the images with their label are shown in Figure 11.3.

```
Categories with highest scores for Box:
    hard disc:  0.291533
        loupe:  0.0731844
        modem:  0.0702888
         pick:  0.0610284
         iPod:  0.0595867
    CD player:  0.0508571
```

In this case, the hard disc is by far the highest score, but the score is much lower than that of the tabby cat—roughly 0.3 vs. 0.8. The summary of scores is

```
AlexNet results summary:

Pepper        0.7000
Cat           0.8056
Box           0.2915
```

Now let’s compare these results to GoogLeNet. GoogLeNet is a pretrained model that has also been trained on a subset of the ImageNet database which is used in the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC). The model is trained on more than a million images, has 144 layers (a lot more than the AlexNet), and can classify images into 1000 object categories. First we load the pretrained network as before.

GoogleNetTest.m

```
1 %% Load the pretrained network
2 net = googlenet;
3 net  % display the 144 layer network
```

The net display is shown as follows. It is a different type than AlexNet, a `DAGNetwork`:

```
net =
DAGNetwork with properties:

       Layers: [144x1 nnet.cnn.layer.Layer]
  Connections: [170x2 table]
```

Next we test it on the image of peppers.

GoogleNetTest.m

```
%% The pepper
% Read the image to classify
I = imread('peppers.png');
sz = net.Layers(1).InputSize;
I = I(1:sz(1),1:sz(2),1:sz(3));
[label, scorePeppers] = classify(net, I);
NewFigure('Pepper');
imshow(I);
title(label);
% What other categories are similar?
disp('Categories with highest scores for Peppers:')
kPos = find(scorePeppers>0.01);
[vals,kSort] = sort(scorePeppers(kPos),'descend');
for k = 1:length(kSort)
  fprintf('%13s:\t%g\n',net.Layers(end).Classes(kPos(kSort(k))),vals(k)
      );
end
```

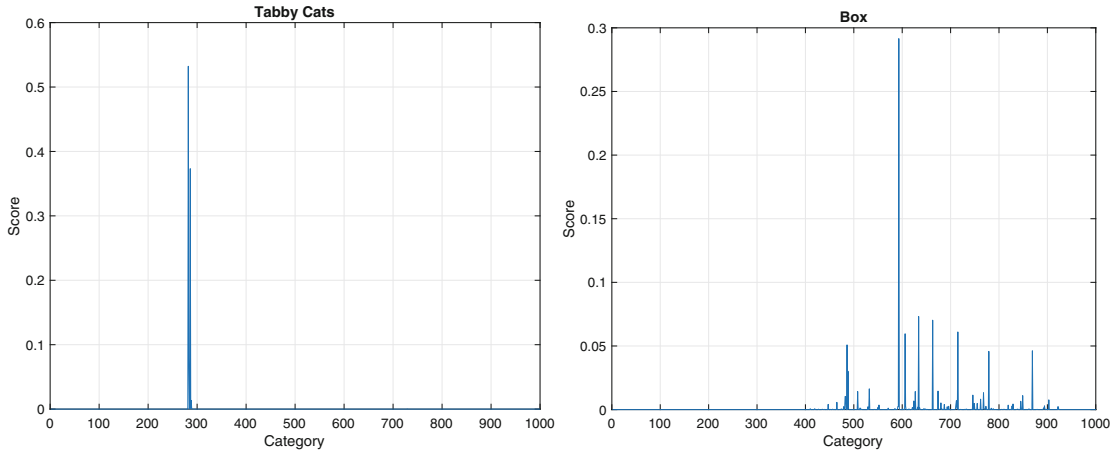
As before, the image is correctly identified as having a bell pepper, and the score is similar to AlexNet. However, the remaining categories are a little different. In this case, the cucumber, and for some reason, a maraca, scored higher than a Granny Smith. Maracas are also round and oblong. The highest categories are shown as follows.

```
Categories with highest scores for Peppers:
  bell pepper:  0.708213
     cucumber:  0.0955994
       maraca:  0.0503938
 Granny Smith:  0.0278589
```

We also test this net on the images of the cat and box. The image size for this network is 224x224. The categories for the cat are the same, with the addition of a lynx, and note that the tabby score is significantly lower than for AlexNet.

```
Categories with highest scores for Cat:
               tabby:   0.532261
        Egyptian cat:   0.373229
           tiger cat:   0.0790764
                lynx:   0.0135277
```

Figure 11.4: GoogLeNet scores for cat, left, and box, right.



The box scores prove the most interesting, and while hard disc is among the highest scores, in this case the net returns *iPod*. A cellular telephone is added to the mix this time. The net clearly knows that it is a rectangular metal object, but beyond that there is no clear evidence for one category over another.

```
Categories with highest scores for Box:
                iPod:   0.443666
           hard disc:   0.212672
  cellular telephone:   0.0787301
               modem:   0.0766429
                pick:   0.0545631
              switch:   0.0169888
               scale:   0.0165957
      remote control:   0.0154203
```

The GoogLeNet score arrays for cat and box are shown in Figure 11.4. The box scores are visibly spread all over the place. This reinforces that the choice of ‘‘ipod” is less certain than the pepper or cat. This shows that even highly trained networks are not necessarily reliable if the input strays too far from the test set.

```
The summary of the GoogleNet results are:

GoogleNet results summary:

Pepper        0.7082
Cat           0.5323
Box           0.4437
```

We can also grab random images from the Internet. The site https://picsum.photos calls itself the ‘‘Lorem Ipsum’’ for photos, and provides a random photo with every call to the URL. Four examples are shown in Figure 11.5.

Figure 11.5: Volcano, lakeside, seashore and geyser.

Figure 11.6: Author headshots with GoogLeNet labels.

Consider, for example

```
>> I = imread('https://picsum.photos/224/224');
>> figure, imshow(I);
>> title(classify(net,I))
```

We got some interesting results using this web site. It produces good results for some landscape photos, but other times sees objects that are not there, see Figure 11.5.

These nets are **not** trained on people; however it can be interesting to test them on images of people. We tested GoogLeNet on our author headshots in Figure 11.6. In both cases, it identified our clothing fairly accurately!

While these nets perform very well on images that do in fact exist in their database, from lions to landscapes, it is important to remember that they are limited in application. Results can be unexpected and even silly.

CHAPTER 12

Orbit Determination

12.1 Introduction

Determining orbits from measurements has been done for hundreds of years. The general approach is to take a series of measurements of the object from the ground. This is a set of angles at different times. Given the location on the Earth, and this set of data, one can reconstruct the orbit. Ideal orbits, which make the assumption that the Earth's gravity is a point at the center of the Earth, are conic sections. Those that stay near the Earth are ellipses. These can be defined as a set of orbital elements. In this chapter, we will design a neural network to find the values for two of the elements. Our model will be simpler than that which astronomers must use. We will assume that all of our orbits are in the Earth's equatorial plane and that the observer is at the center of the Earth.

The purpose of this chapter is to show that a neural net can do orbit determination. For comparison with traditional methods, see the classic textbook from 1965 by Escobal [11].

12.2 Generating the Orbits

12.2.1 Problem

We want to create a set of orbits for testing and training a neural net.

12.2.2 Solution

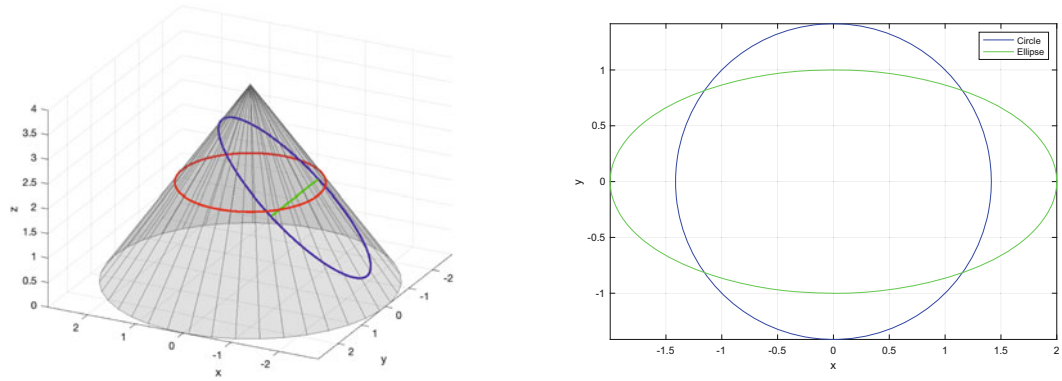
Implement a random orbit generator using Keplerian propagation of elements.

12.2.3 How It Works

An orbit involves at least two bodies, for example, a planet and a spacecraft. In the ideal two-body case, the two bodies rotate about the common center of mass, known as the barycenter. For all practical spacecraft cases, the spacecraft mass itself is negligible, and this means that the satellite follows a conic section path about the primary body's center of mass. A conic section is a curve that fits on a cone, as shown in Figure 12.1. Two conics, a circle and ellipse, are drawn. Hyperbolas and parabolas are also conic sections, but we will only look at elliptical orbits in this chapter.

M. Paluszek and S. Thomas, *Practical MATLAB Deep Learning*,
https://doi.org/10.1007/978-1-4842-5124-9_12

Figure 12.1: Ellipse and circle on a cone and viewed along their normal.



The code that draws this picture is in the following script. It calls two functions, `Cone` and `ConicSectionEllipse`. `r0` and `h` are only needed to draw the cone. The algorithm only cares about `theta`, the cone half angle.

ConicSection.m

```
theta = pi/4;
h     = 4;
r0    = h*sin(theta);

ang   = linspace(0,2*pi);
a     = 2;
b     = 1;
cA    = cos(ang);
sA    = sin(ang);
n     = length(cA);
c     = 0.5*h*sin(theta)*[cA;sA;ones(1,n)];
e     = [a*cA;b*sA;zeros(1,n)];

% Show a planar representation
NewFigure('Orbits');
plot(c(1,:),c(2,:),'b')
hold on
plot(e(1,:),e(2,:),'g')
grid
xlabel('x')
ylabel('y')
axis image
legend('Circle','Ellipse');

[z,phi,x]   = ConicSectionEllipse(a,b,theta);
ang         = pi/2 + phi;
e           = [cos(ang) 0 sin(ang);0 1 0; -sin(ang) 0 cos(ang)]*e;
e(1,:)      = e(1,:) + x;
e(3,:)      = e(3,:) + h - z;

Cone(r0,h,40);
```

```
hold on
plot3(c(1,:),c(2,:),2*ones(1,n),'r','linewidth',2);
plot3(e(1,:),e(2,:),e(3,:),'b','linewidth',2);
line([x x],[-b b],[h-z h-z],'color','g','linewidth',2);
view([0 1 0])
```

The view is set to look along the y-axis which is the axis of rotation for the ellipse. The function `Cone` draws the cone. `line` draws the axis of rotation that is along the short axis.

The solution that is used to draw the conic sections is derived in the last section of this chapter. The orbit may be elliptical, with an eccentricity less than 1, parabolic with an eccentricity equal to 1, or hyperbolic with an eccentricity greater than 1. Figure 12.2 shows the geometry of an elliptical orbit. This is a planar orbit in which the orbital motion is two-dimensional. The semi-major axis a is

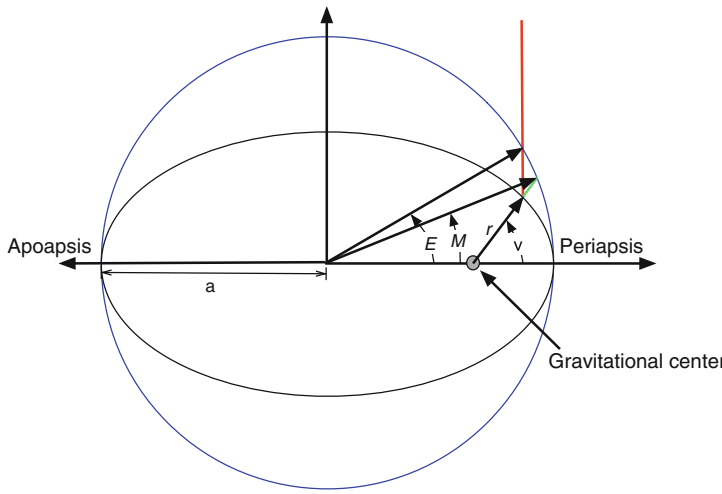
$$a = \frac{r_a + r_p}{2} \tag{12.1}$$

where r_a is the apoapsis (apogee for the Earth) radius, or point furthest from the central planet, and r_p is the periapsis radius (perigee for the Earth), or closest point to the planet. The eccentricity, e, of the orbit is

$$e = \frac{r_a - r_p}{r_a + r_p} \tag{12.2}$$

When $r_a = r_p$, the orbit is circular and $e = 0$. This formula is not meaningful for parabolic or hyperbolic orbits. Figure 12.2 shows three angular measurements, M mean anomaly, E eccentric anomaly, and ν true anomaly. All are measured from periapsis. The mean anomaly is

Figure 12.2: Elliptical orbit.



related to the mean orbit rate n through a simple function of time.

$$M = M_0 + n(t - t_0) \tag{12.3}$$

The eccentric anomaly is the angle to the current position as projected onto the ellipse's circumscribing circle, drawn in blue. It is related to the mean anomaly by Kepler's equation.

$$M = E - e \sin E \tag{12.4}$$

This equation needs to be solved numerically in general, but for small values of e, $e < 0.1$, this approximation can be used.

$$E \approx M + e \sin M + \frac{1}{2} e^2 \sin 2M \tag{12.5}$$

This is because apoapsis is not well defined for very small e. Higher-order formulas can also be found. The true anomaly is related to the eccentric anomaly through the equation

$$\tan \frac{\nu}{2} = \sqrt{\frac{1+e}{1-e}} \tan \frac{E}{2} \tag{12.6}$$

Finally, the orbit radius is

$$r = \frac{a(1-e)(1+e)}{1 + e \cos \nu} \tag{12.7}$$

If $e > 1$ in this equation, r will go to ∞, as is expected for parabolic or hyperbolic orbits.

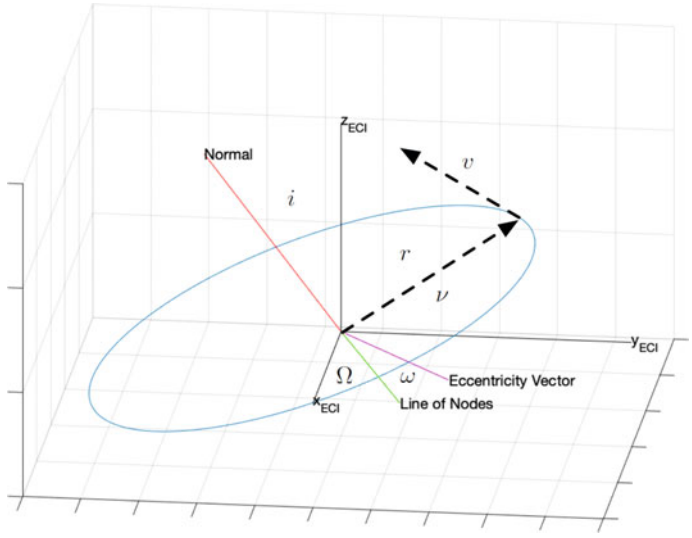
Seven parameters are necessary to define an orbit of a spacecraft about a spherically symmetric body. One is the gravitational parameter, generally denoted by the symbol μ. The gravitational parameter is

$$\mu = G(m_1 + m_2) \tag{12.8}$$

where m_1 is the mass central body and m_2 is the mass of the orbiting body. G is the gravitational constant with units of $m^3/kg/s^2$. For the Earth, $G = 6.6774 \times 10^{-11}$. μ for the Earth is 3.98600436×10^8 m^3/s^2. There are many ways of representing the other six elements. The two most popular sets are position and velocity (r and v) vectors, and Keplerian orbital elements. Each representation uses six independent variables to describe the orbit, plus μ. Both are shown in Figure 12.3.

The Keplerian elements are defined as follows. Two elements define the elliptical orbit. The size of the orbit is determined by the semi-major axis a, which is the average of the perigee radius and apogee radius. The size and shape of the orbit are defined by the eccentricity, e. Two elements define the orbital plane. Ω is the longitude which is the right ascension of the ascending node, or the angle from the $+X$ axis of the reference frame to the line where the orbit plane intersects the xy-plane. i is the inclination and is the angle between the xy-plane and the orbit plane. ω is the argument of perigee and is the angle in the orbit plane between the ascending node line and perigee (where the orbit is closest to the center of the central body). ν is the true anomaly and is the angle between perigee and the spacecraft. The mean anomaly M may be used in the element set instead of ν. M or ν tells us where the spacecraft is in its orbit.

Figure 12.3: Orbital elements. The underlying plot was drawn using `DrawEllipticOrbit`.



To summarize, the Keplerian elements are

$$x = \begin{bmatrix} a \\ i \\ \Omega \\ \omega \\ e \\ M \end{bmatrix} \tag{12.9}$$

The orbit period, with units of seconds, is

$$P = 2\pi\sqrt{\frac{a^3}{\mu}} \tag{12.10}$$

The orbit parameter, with units of distance (conventionally km), is

$$p = a(1-e)(1+e) \tag{12.11}$$

The in-plane position and velocity are

$$r_p = \frac{p}{1+e\cos\nu}\begin{bmatrix} \cos\nu \\ \sin\nu \\ 0 \end{bmatrix} \tag{12.12}$$

$$v_p = \sqrt{\frac{\mu}{p}}\begin{bmatrix} -\sin\nu \\ e+\cos\nu \\ 0 \end{bmatrix} \tag{12.13}$$

The transformation matrix from planar to three-dimensional coordinates is

$$c = \begin{bmatrix} \cos\Omega\cos\omega - \sin\Omega\sin\omega\cos i & -\cos\Omega\sin\omega - \sin\Omega\cos\omega\cos i & \sin\Omega\sin i \\ \sin\Omega\cos\omega + \cos\Omega\sin\omega\cos i & -\sin\Omega\sin\omega + \cos\Omega\cos\omega\cos i & -\cos\Omega\sin i \\ \sin\omega\sin i & \cos\omega\sin i & \cos i \end{bmatrix} \tag{12.14}$$

That is

$$r = cr_p \tag{12.15}$$

$$v = cv_p \tag{12.16}$$

For the purposes of creating the neural net, we will look at orbits with the inclination, $i = 0$, and the ascending node, $\Omega = 0$. The transformation matrix reduces to a rotation about z.

$$c = \begin{bmatrix} \cos\omega & -\sin\omega & 0 \\ \sin\omega & \cos\omega & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{12.17}$$

We now want to propagate the orbit forward in time. There are two alternative approaches for doing so. One approach is to use Keplerian propagation, where we keep five of the elements constant, and simply march the mean anomaly forward in time at a constant rate of $n = \sqrt{\mu/a^3}$. At each point in time, we can convert the set of six orbital elements into a new position and velocity. This approach is limited though, in that it assumes the orbit follows a Keplerian orbit (the only external foci is the gravitation of a central body with uniform mass distribution). The second approach, which gives us more flexibility, for external forces like thrust and drag is to numerically integrate the dynamic equations of motion. The state equations for orbit propagation are

$$\dot{v} = -\mu\frac{r}{|r|^3} + a \tag{12.18}$$

$$\dot{r} = v \tag{12.19}$$

The terms on the right-hand side of the velocity derivative equation are the point mass gravity acceleration with additional acceleration a. This is implemented in `RHSOrbit`.

```
function xDot = RHSOrbit(~,x,d)

r     = x(1:2);
v     = x(3:4);
xDot  = [v;-d.mu*r/(r'*r)^1.5 + d.a];
```

We will create a script that simulates multiple orbits. The simulation will use `RHSOrbit`. The first part of the orbit generation script sets up the random orbital elements.

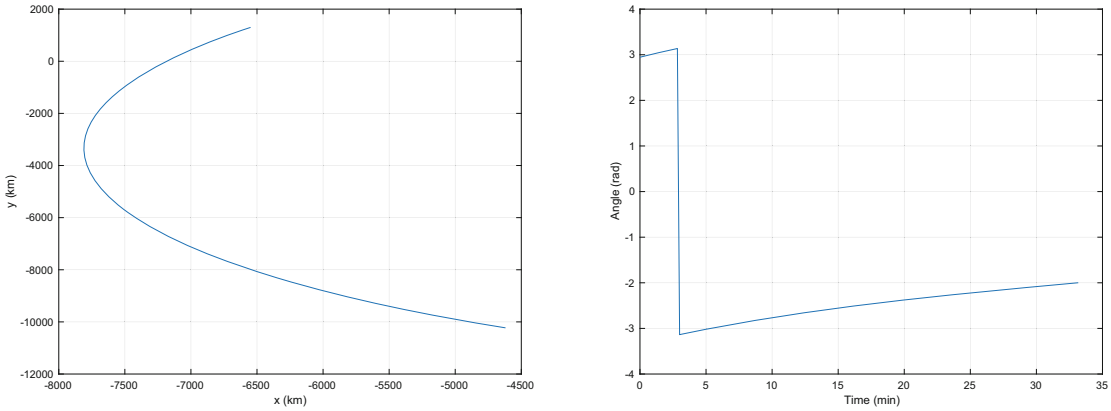
Orbits.m

```
%% Generate Orbits for angles-only element estimation
% Saves a mat-file called OrbitData.
%% See also
% El2RV, RungeKutta, RHSOrbit, TimeLabel, PlotSet

nEl   = 500;          % Number of sets of data
d     = struct;       % Initialize
d.mu  = 3.98600436e5; % Gravitational parameter, km^3/s^2
d.a   = [0;0];        % Perturbing acceleration

% Random elements
e     = 0.6*rand(1,nEl);          % Eccentricity
a     = 8000 + 1000*randn(1,nEl); % Semi-major axis
M     = 0.25*pi*rand(1,nEl);      % Mean anomaly
```

The next section runs the simulations and saves the angles. Each simulation has 2000 steps, and each step is 2 seconds. We are only using one in ten points for the orbit determination. We save the orbital elements for testing the neural network. We are not applying any external acceleration. We could have used Kepler propagation, but by simulating the orbit, we have the option for studying how well the neural network performs with disturbances.

```
% Set up the simulation
nSim  = 2000; % Number of simulation steps
dT    = 2; % Time step

% Only use some of the sim steps
jUse  = 1:10:nSim;

% Data for Deep Learning
data  = cell(nEl,1);

%% Simulate each of the orbits
x     = zeros(4,nSim);
t     = (0:(nSim-1))*dT;
el(nEl) = struct('a',7000,'e',0); % initialize struct array

for k = 1:nEl
  [r,v] = El2RV([a(k) 0 0 0 e(k) M(k)]);
  x     = [r(1:2);v(1:2)];
  xP    = zeros(4,nSim);
  for j = 1:nSim
    xP(:,j) = x;
    x       = RungeKutta( @RHSOrbit, 0, x, dT, d );
  end
  data{k}     = atan2(xP(2,jUse),xP(1,jUse));
  el(k).a     = a(k);
  el(k).e     = e(k);
end
```

Figure 12.4: The last test orbit. The measured angle is on the right. These are only showing the data used in orbit determination.



The final part plots the orbits and saves the data to a file.

```
%% Save for the Deep Learning algorithm
save('OrbitData','data','el');
```

The last orbit is shown in Figure 12.4. The jump in angle is due to angles being defined from $-\pi$ to $+\pi$. We could have used `unwrap` to get rid of this jump. We are only measuring for part of an orbit. We can set up the simulation to measure any part of an orbit, or even multiple orbits.

12.3 Training and Testing

12.3.1 Problem

We want to build a deep learning system to compute the eccentricity and semi-major axis for an orbit from angle measurements.

12.3.2 Solution

The orbit history is a time series of angles. We will take angles at uniform time intervals. We will use `fitnet` to fit the data.

12.3.3 How It Works

We load in the data from the mat file and separate it into training and testing sets.

OrbitNeuralNet.m

```
%% Train and test the Orbit Neural Net
%% See also:
% Orbits, fitnet, configure, train, sim, cascadeforwardnet,
    feedforwardnet

s       = load('OrbitData');
n       = length(s.data);
nTrain  = floor(0.9*n);

%% Set up the training and test sets
kTrain  = randperm(n,nTrain);
sTrain  = s.data(kTrain);
nSamp   = size(sTrain{1},2);
xTrain  = zeros(nSamp,nTrain);
aMean   = mean([s.el(:).a]);

for k = 1:nTrain
  xTrain(:,k) = sTrain{k}(1,:);
end

elTrain     = s.el(kTrain);
yTrain      = [elTrain.a;elTrain.e];
yTrain(1,:) = yTrain(1,:)/aMean;
% Normalize the data so it is the same magnetic as the eccentricity
kTest       = setdiff(1:n,kTrain);
sTest       = s.data(kTest);
nTest       = n-nTrain;
xTest       = zeros(nSamp,nTest);
for k = 1:nTest
  xTest(:,k) = sTest{k}(1,:);
end

elTest      = s.el(kTest);
yTest       = [elTest.a;elTest.e];
yTest(1,:)   = yTest(1,:)/aMean;
```

The neural network will use sequences of angles and their related times as the input. The output will be the two orbital elements: semi-major axis and eccentricity. In general, if we know the position and velocity at a point in the orbit, we can always compute the orbital elements. This is done in the function `El2RV`. Although we don't directly measure velocity, it can be estimated by differencing position measurements. With angle-only measurements, we don't have a measure of range. The question is, can the neural network infer the range from the time variation of the angles?

We train the network using `fitnet`. Note that we normalized the semi-major axis so that the magnitude is the same order as the eccentricity. This improves the fitting.

```
%% Train the network
net        = fitnet(10);

net        = configure(net, xTrain, yTrain);
net.name   = 'Orbit';
net        = train(net,xTrain,yTrain);
```

We use the test data to test the network.

```
%% Test the network
yPred       = sim(net,xTest);
yPred(1,:)  = yPred(1,:)*aMean;
yTest(1,:)  = yTest(1,:)*aMean;
yM          = mean(yPred-yTest,2);
yTM         = mean(yTest,2);
fprintf('\nFit Net\n');
fprintf('Mean semi-major axis error %12.4f (km) %12.2f %%\n',yM(1),100*
    abs(yM(1))/yTM(1));
fprintf('Mean eccentricity    error %12.4f      %12.2f %%\n',yM(2),100*
    abs(yM(2))/yTM(2));
%% Plot the results
yL  = {'a' 'e'};
yLeg = {'Predicted','True'};
PlotSet(1:nTest,[yPred;yTest],'x label','Test','y label',yL,...
'figure title','Predictions using Fitnet','plot set',{[1 3] [2 4]},...
'legend',{yLeg yLeg});
```

The results are best for `fitnet`. However, the results will vary with each run.

```
>> OrbitNeuralNet
>> OrbitNeuralNet

Fit Net
Mean semi-major axis error      31.9872 (km)          0.41 %
Mean eccentricity    error       0.0067               2.48 %

Cascade Forward Net
Mean semi-major axis error     -89.8603 (km)          1.15 %
Mean eccentricity    error      -0.0100               3.74 %

Feed Forward Net
Mean semi-major axis error      40.2986 (km)          0.52 %
Mean eccentricity    error       0.0001               0.03 %
```

Figures 12.5, 12.6, and 12.7 show the test results. Both semi-major axis and eccentricity results are reasonably good. You can experiment with different spans of data and different sampling intervals. The code is in the script `OrbitNeuralNet.m`.

Figure 12.5: Test results using `fitnet`.

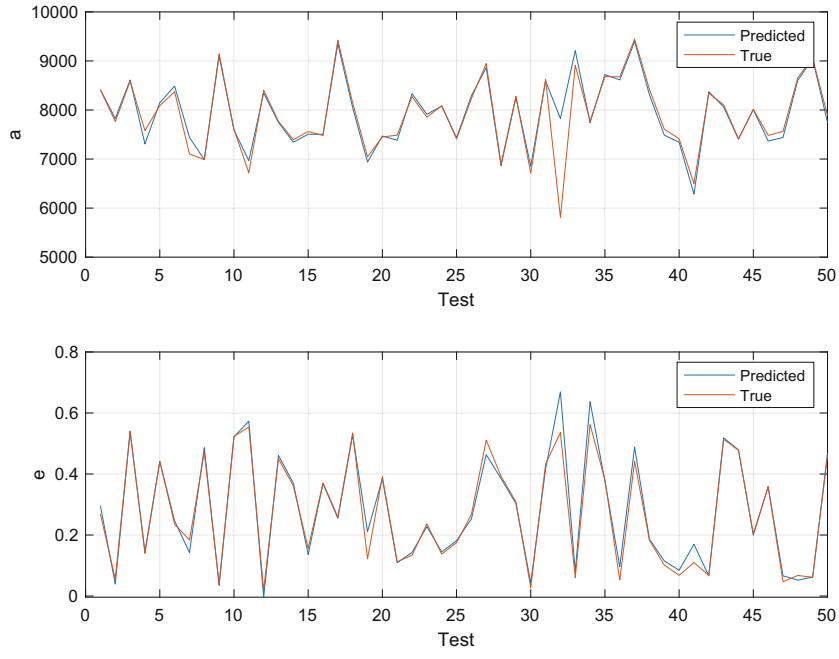


Figure 12.6: Test results using `cascadeforwardnet`.

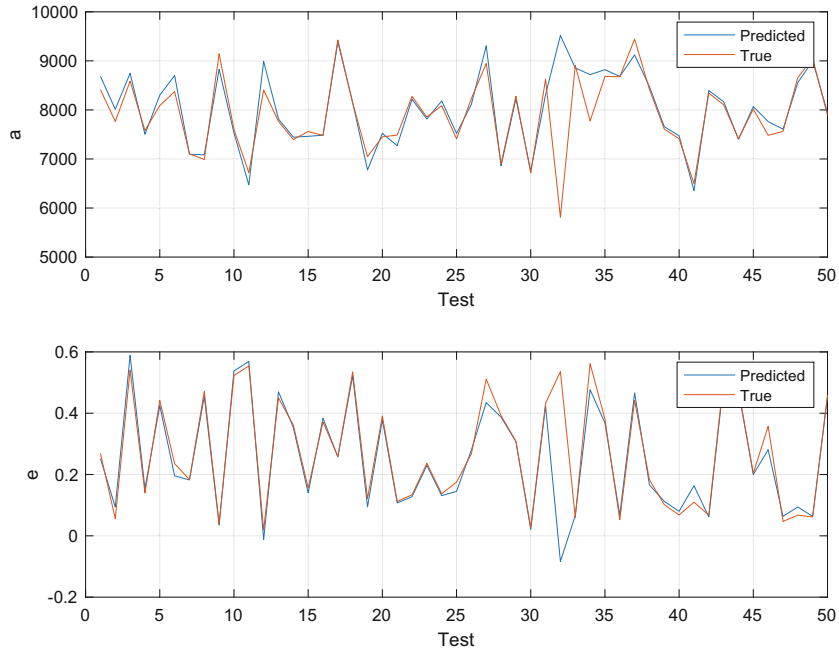
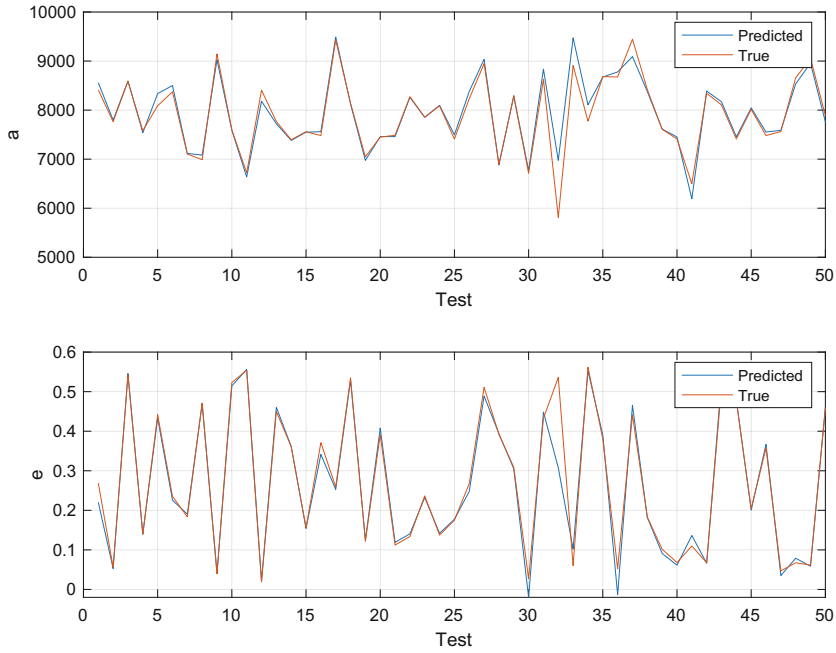


Figure 12.7: Test results using `feedforwardnet`.



We train the network using `cascadeforwardnet`. The code doesn't change except for the function name.

```
%% Train the cascade forward network
net        = cascadeforwardnet(10);

net        = configure(net, xTrain, yTrain);
net.name   = 'Orbit';
net        = train(net,xTrain,yTrain);
```

We finally train it using `feedforwardnet`.

```
%% Train the feed forward network
net        = feedforwardnet(10);

net        = configure(net, xTrain, yTrain);
net.name   = 'Orbit';
net        = train(net,xTrain,yTrain);
```

12.4 Implementing an LSTM

12.4.1 Problem

We want to build a long short-term memory neural net (LSTM) to estimate the orbital elements. LSTMs have been demonstrated in previous chapters. They are an alternative to the functions shown earlier.

12.4.2 Solution

The orbit history is a time series of angles. We will use a bidirectional LSTM to fit the data. We will take angles at uniform time intervals.

12.4.3 How It Works

We load in the data from the mat file and separate it into training and testing sets. The data format is different from the feedforward networks. `xTrain` is a cell array, but `yTrain` is a matrix with a row for each cell array in `xTrain`.

OrbitLSTM.m

```
%% Script to train and test the Orbit LSTM
% It will estimate the orbit semi-major axis and eccentricity from a time
% sequence of angle measurements.
%% See also
% Orbits, sequenceInputLayer, bilstmLayer, dropoutLayer,
    fullyConnectedLayer,
% regressionLayer, trainingOptions, trainNetwork, predict

s           = load('OrbitData');
n           = length(s.data);
nTrain      = floor(0.9*n);

%% Set up the training and test sets
kTrain      = randperm(n,nTrain);
aMean       = mean([s.el(:).a]);
xTrain      = s.data(kTrain);
nTest       = n-nTrain;

elTrain     = s.el(kTrain);
yTrain      = [elTrain.a;elTrain.e]';
yTrain(:,1) = yTrain(:,1)/aMean;
kTest       = setdiff(1:n,kTrain);
xTest       = s.data(kTest);

elTest      = s.el(kTest);
yTest       = [elTest.a;elTest.e]';
yTest(:,1)      = yTest(:,1)/aMean;
```

We train the network using `trainNetwork`.

```
%% Train the network with validation
numFeatures       = 1;
numHiddenUnits1   = 100;
numHiddenUnits2   = 100;
numClasses        = 2;

layers = [ ...
    sequenceInputLayer(numFeatures)
    bilstmLayer(numHiddenUnits1,'OutputMode','sequence')
    dropoutLayer(0.2)
    bilstmLayer(numHiddenUnits2,'OutputMode','last')
    fullyConnectedLayer(numClasses)
    regressionLayer]

maxEpochs = 20;

options = trainingOptions('adam', ...
    'ExecutionEnvironment','cpu', ...
    'GradientThreshold',1, ...
    'MaxEpochs',maxEpochs, ...
    'Shuffle','every-epoch', ...
    'ValidationData',{xTest,yTest}, ...
    'ValidationFrequency',5, ...
    'Verbose',0, ...
    'Plots','training-progress');

net = trainNetwork(xTrain,yTrain,layers,options);
```

`options` is given validation data. Note the cell array that is required for the validation data.

```
      'ValidationData',{xTest,yTest}, ...
```

We shuffle the data. This generally improves the results since the learning algorithm sees the data in a different order on each epoch. We use the test data to test the network. `predict` produces results based on the test data. This is the same data used for validation during learning.

The results are given as follows.

```
>> OrbitLSTM
layers =
  6x1 Layer array with layers:

     1   ''   Sequence Input       Sequence input with 1 dimensions
     2   ''   BiLSTM               BiLSTM with 100 hidden units
     3   ''   Dropout              20% dropout
     4   ''   BiLSTM               BiLSTM with 100 hidden units
     5   ''   Fully Connected      2 fully connected layer
     6   ''   Regression Output    mean-squared-error

biLSTM
Mean semi-major axis error      -63.4780 (km)
Mean eccentricity    error        0.0024
```

We use two BiLSTM layers with a 20% dropout between layers. Dropout removes neurons and helps prevent overfitting. Overfitting is when the results correspond too closely toward a particular set of data. This makes it hard for the trained network to identify patterns in new data. The first BiLSTM layer produces a sequence as its output. The second BiLSTM layer's `'OutputMode'` is set to `'last'`. The `numClasses` is 2 because we are estimating two parameters. The fully connected layer connects the two BiLSTM outputs to the two parameters we want to identify in the regression layer. The training window is shown in Figure 12.8. We could have continued the training for more epochs as the root-mean-square error (RMSE) is still improving.

This particular set of layers is to show you how to build a neural network. It is by no means the ''best" architecture for this problem. We did try a single LSTM layer and a single BiLSTM layer worked better.

Figure 12.9 shows the test results. The results are not quite as good as the feedforward nets given earlier. We've only used two layers. From Chapter 11, you see that ''professional" networks can have dozens if not hundreds of layers. The difference is due to the smaller number of neurons in the LSTM. You can experiment with this network to improve the results.

In this chapter, we have compared two approaches, in MATLAB, to solving the orbit determination problem. Using the MATLAB functions worked a bit better than the LSTM we implemented. We made the argument of perigee constant to make the problem easier. The next step would be to try and find the full set of orbital elements and then try to design a system that works from a fixed point on the Earth. In the latter case, we would need to account for the rotation of the Earth. Another improvement would be to take the measurements at different time steps. For an elliptical orbit, taking many measurements at perigee is more productive than at apogee because the spacecraft is moving faster. One could write a preprocessor to select inputs to our neural network based on the angular change with respect to time. Orbit determination systems, using algorithmic approaches, can also compute errors in the observer's location. You could also try other measurements, such as range and range rate. These measurements are used for deep space and geosynchronous spacecraft.

Figure 12.8: Training window.

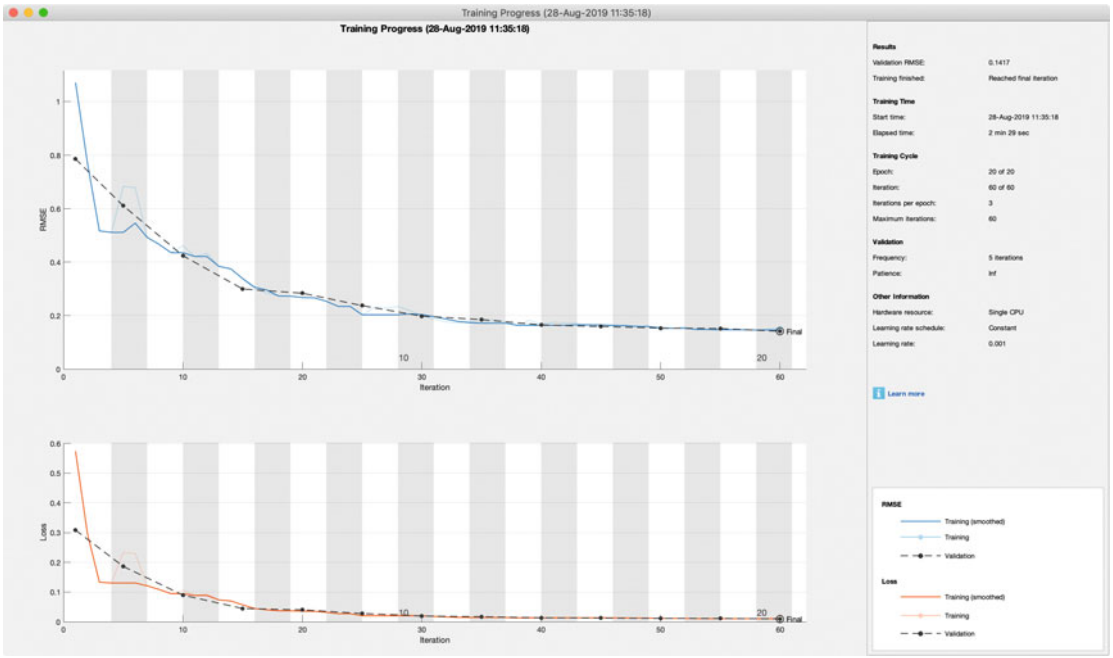
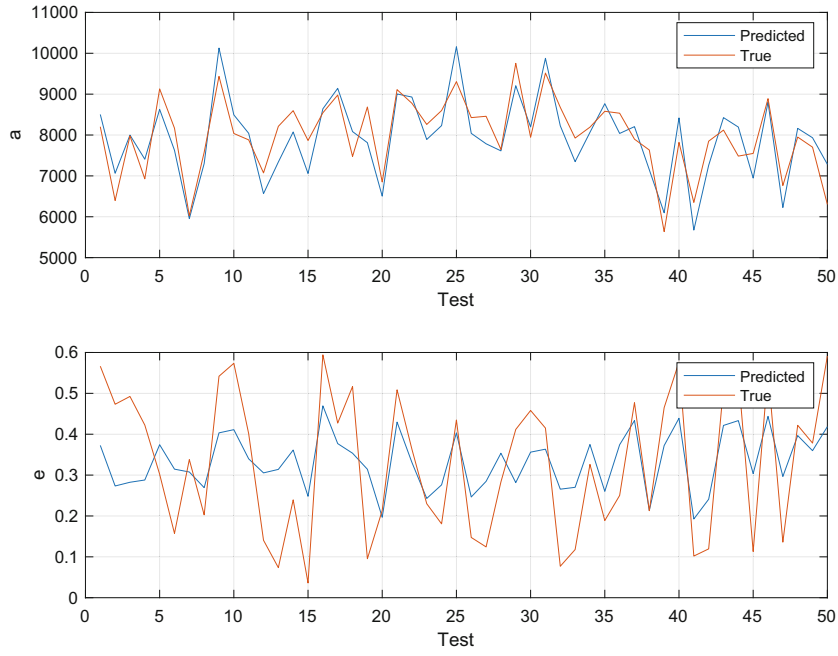


Figure 12.9: Test results using the bidirectional LSTM.



12.5 Conic Sections

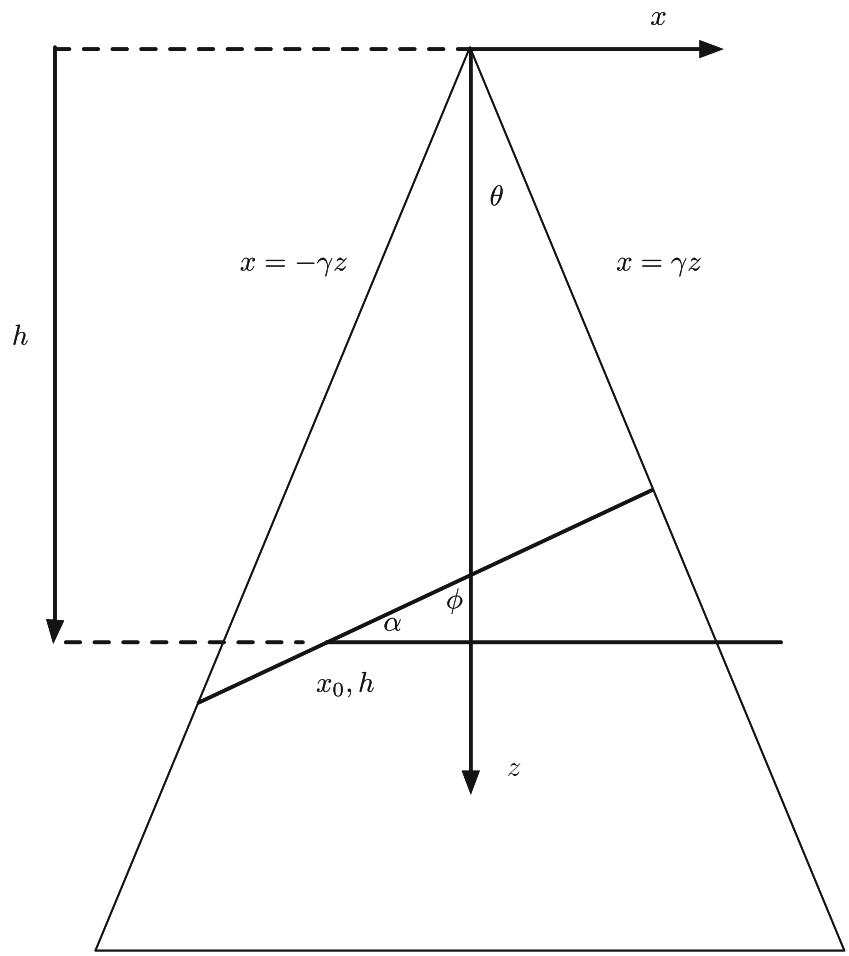
For a given ellipse and cone, we need to solve for the location of the center of the plane that cuts the cone and its angle. The problem can be solved by working in the zy-plane. This is shown in Figure 12.10. The equation for an ellipse is

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1 \tag{12.20}$$

The equation for a right circular cone is

$$x^2 + y^2 = \gamma^2 z^2 \tag{12.21}$$

Figure 12.10: Conic section.



where $\gamma = \cos\theta$ and θ is the cone half angle. This is just saying that the radius of the cone at position z is γz. The equation for a plane is

$$z = h \tag{12.22}$$

That is, z is constant for all x and y. If we rotate this about the y-axis by α and translate it by x_0, we get the equation for the cone. The ellipse comes from the intersection of the plane with the cone. b is along the y-axis. The angle between the plane that cuts the cone and the xy-plane is α and is a function of a, b, and the cone half angle θ. The following equations are for $\theta = \pi/4$.

$$\tan^2\alpha = 1 - \frac{b^2}{a^2} \tag{12.23}$$

Noting that

$$\phi = \frac{\pi}{2} - \alpha \tag{12.24}$$

The relationship between the plane and the vertical for a cone with a half angle of $\frac{\pi}{4}$ is then

$$\phi = \frac{\pi}{2} - \text{atan}\sqrt{1 - \frac{b^2}{a^2}} \tag{12.25}$$

The cone can be viewed in the plane. On the right side, the equation is

$$x = \gamma x \tag{12.26}$$

where $\gamma = \cos\theta$, where θ is the cone half angle. On the left side

$$x = -\gamma x \tag{12.27}$$

We then write the equations for the line along the major axis of the ellipse on each side of the triangle. On the right

$$x = x_0 + a\cos\alpha \tag{12.28}$$
$$z = h - a\sin\alpha \tag{12.29}$$

where $\alpha = \pi/2 - \phi$. On the left

$$x = x_0 - a\cos\alpha \tag{12.30}$$
$$z = h + a\sin\alpha \tag{12.31}$$

Substituting into the equations for the cone, we get

$$\begin{bmatrix} 1 & -\gamma \\ 1 & \gamma \end{bmatrix} \begin{bmatrix} x_0 \\ h \end{bmatrix} = a \begin{bmatrix} -\gamma\sin\alpha - \cos\alpha \\ \cos\alpha - \gamma\sin\alpha \end{bmatrix} \tag{12.32}$$

The code that solves the equations follows. We could have solved the inverse analytically since it is 2 by 2.

ConicSectionEllipse.m

```
function [h,phi,x] = ConicSectionEllipse(a,b,theta)

if( nargin < 1 )
  [h, phi, y] = ConicSectionEllipse(2,1,pi/4);
  fprintf('h   = %12.4f\n',h);
  fprintf('phi = %12.4f (rad)\n',phi);
  fprintf('x   = %12.4f\n',y);
  clear h
  return
end

phi   = pi/2 - atan(sqrt(1-b^2/a^2));

alpha = pi/2 - phi;
c     = cos(alpha);
s     = sin(alpha);
gamma = cos(theta);
f     = a*[-gamma*s - c;c - gamma*s];
q     = [1 -gamma;1 gamma]\f;
x     = q(1);
h     = q(2);
```

Bibliography

[1] Imagenet classification with deep convolutional neural networks. Technical report.

[2] M.M.M. Al-Husari, B. Hendel, I.M. Jaimoukha, E.M. Kasenally, D.J.N. Limebeer, and A. Portone. Vertical stabilisation of Tokamak Plasmas. In *Proceedings of the 30th Conference on Decision and Control*, December 1992.

[3] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling. *arXiv*, April 2018.

[4] Ilker Birbil and Shu-Chering Fang. An electromagnetism-like mechanism for global optimization. *Journal of Global Optimization*, 25:263--282, 03 2003.

[5] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[6] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *SIAM Review*, 60:223--311, 2016.

[7] A. Bryson and Y. Ho. *Applied Optimal Control*. Hemisphere Publishing Company, 1975.

[8] Barbara Cannas, Gabriele Murgia, A Fanni, Piergiorgio Sonato, Augusto Montisci, and M.K. Zedda. Dynamic Neural Networks for Prediction of Disruptions in Tokamaks. *CEUR Workshop Proceedings*, 284, 01 2007.

[9] Wroblewski D. and et al. Tokamak disruption alarm based on neural network model of high-beta limit. *Nuclear Fusion*, 37(725), 11 1997.

[10] Steven R. Dunbar. Stochastic Processes and Advanced Mathematical Finance. Technical report, University of Nebraska-Lincoln.

[11] Pablo Ramon Escobal. *Methods of Orbit Determination*. Krieger Publishing Company, 1965.

[12] David Foster. *Generative Deep Learning*. O'Reilly Media, Inc., June 2019.

[13] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley.

[14] S. Haykin. *Neural Networks*. Prentice-Hall, 1999.

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

[15] Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. Extreme learning machine: Theory and applications. *Neurocomputing*, 70(1):489 -- 501, 2006. Neural Networks.

[16] P. Jackson. *Introduction to Expert Systems, Third Edition*. Addison-Wesley, 1999.

[17] Diederik P. Kingma and Jimmy Lei Ba. ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION. 2015.

[18] Y. Liang and JET EFDA Contributors. Overview of Edge Localized Modes Control in Tokamak Palsama. Technical Report Preprint of Paper for Fusion Science and Technology, JET-EFDA.

[19] A. J. Lockett and R. Miikkulainen. Temporal convolutional machines for sequence learning. Technical Report Technical Report AI-09-04.

[20] Microsoft. sentence-completion. https://drive.google.com/drive/folders/0B5eGOMdyHn2mWDYtQzlQeGNKa2s, 2019.

[21] M. Paluszek, Y. Razin, G. Pajer, J. Mueller, and S. Thomas. *Spacecraft Attitude and Orbit Control: Third Edition*. Princeton Satellite Systems, 2019.

[22] G.A. Ratta, J. Vega, A. Murari, the EUROfusion MST Team, and JET Contributors. AUG-JET cross-tokamak disruption predictor. In *2nd IAEA TM*, 2017.

[23] L.M. Rasdi Rere, Mohamad Ivan Fanany, and Aniati Murni Arymurthy. Simulated annealing algorithm for deep learning. *Procedia Computer Science*, 72:137--144, 2015.

[24] S. Russell and P. Norvig. *Artificial Intelligence A Modern Approach Third Edition*. Prentice-Hall, 2010.

[25] Paul A. Samuelson. Mathematics of speculative price. *SIAM Review*, 15(1):1--42, 1973.

[26] R.O. Sayer, Y.K.M. Peng, J.C. Wesley, S.C. Jardin, CA General Atomics, San Diego, and NJ Princeton Univ. ITER disruption modeling using TSC (Tokamak Simulation Code). 11 1989.

[27] Luigi. Scibile. *Non-linear control of the plasma vertical position in a tokamak*. PhD thesis, University of Oxford, 1997.

[28] Richard Socher. *Recursive Deep Learning for Natural Language Processing and Computer Vision*. PhD thesis, August 2014.

[29] Stephanie Thomas and Michael Paluszek. *MATLAB Machine Learning*. Apress, 2017.

[30] Stephanie Thomas and Michael Paluszek. *MATLAB Machine Learning Recipes: A Problem-Solution Approach*. Apress, 2019.

[31] Geoffrey Zweig and Chris J.C. Burges. The microsoft research sentence completion challenge. Technical Report MSR-TR-2011-129, December 2011.

Index

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