Generating a New Reality

From Autoencoders and Adversarial Networks to Deepfakes

Micheal Lanham

Apress®

Generating a New Reality

From Autoencoders and Adversarial Networks to Deepfakes

Micheal Lanham

Apress®

Generating a New Reality: From Autoencoders and Adversarial Networks to Deepfakes

Micheal Lanham
Calgary, AB, Canada

ISBN-13 (pbk): 978-1-4842-7091-2 ISBN-13 (electronic): 978-1-4842-7092-9
https://doi.org/10.1007/978-1-4842-7092-9

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Celestin Suresh John
Development Editor: Laura Berendson
Coordinating Editor: Aditee Mirashi

Cover designed by eStudioCalamar

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

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, Suite 4600, New York, NY 10004-1562, USA. 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 booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail 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 www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-7091-2. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

To my true loves: knowledge, my children, and Rhonda.

Table of Contents

About the Author

Micheal Lanham is a proven software and tech innovator with 20+ years of experience. During that time, he has developed a broad range of software applications in areas such as games, graphics, web, desktop, engineering, artificial intelligence, GIS, and machine learning applications for a variety of industries as an R&D developer. At the turn of the millennium, Micheal began working with neural networks and evolutionary algorithms in game development. He is an avid educator, and along with writing several books about game development, extended reality, and AI, he regularly teaches at meetups and other events. Micheal also likes to cook for his large family in his hometown of Calgary, Canada.

About the Technical Reviewer

Aneesh Chivukula is a technical expert and an analytics executive. He has strong academic research capacities in machine learning. He has developed innovative products with artificial intelligence. He brings thought leadership of technology trends in the enterprise solutions.

Aneesh has a doctorate of philosophy degree in data analytics and machine learning from the University of Technology Sydney, Australia. He holds a master of science degree by research in computer science and artificial intelligence from the International Institute of Information Technology Hyderabad, India.

Acknowledgments

This book, like many others, would not have been possible without the free exchange of knowledge provided in the AI/ML community, from countless researchers who tirelessly work on improving the field of artificial intelligence and generative modeling to the mass of AI enthusiasts who regularly produce open code repositories featuring new tools and a catalog of innovations.

I would like to thank and acknowledge all the contributions of those in the AI/ML field who work hard educating others. Many of the examples in this book have been collated from the numerous open-source repositories featuring deep learning and generative modeling. One such resource developed by Erik Linder-Norén, an ML engineer at Apple, inspired and contributed to several examples in this book's early chapters.

I would also like to thank you, the reader, for taking the opportunity to review this text and open your mind to new opportunities. It has always been a profound pleasure of mine watching that light bulb moment students experience when they first meld with a new concept. It's something I hope you will experience several times through the course of this book.

Lastly, as always, special thanks to my large family and friends. While I may not see all my nine children on a regular basis, they and their children always have a special place in my heart. I feel fortunate that my family supports my writing and continues to encourage new titles.

Introduction

We live in an era of fake news and uncertain reality. It's a world where reality has become blurred by digital wizardry artists and artificial intelligence practitioners. It's a digital reality now populated with fake news, images, and people. For many, the uncertainty and confusion are overwhelming. Yet, others, like yourself, search to embrace this new era of digital fakery to explore new opportunities.

This book takes an in-depth look at the technology that powers this new digital fake reality. The broad name for this technology or form of AI/ML is *generative modeling* (GM). It is a form of AI/ML modeling that looks to understand what something represents, as opposed to other forms of modeling that look to classify or predict something.

Generative modeling is not a new concept, but one that has emerged from the application of deep learning. The introduction of deep learning has launched the field into the mainstream. Unfortunately, not all mainstream use of GM is flattering or showcases the power of this diverse technology.

There is a real and speculated fear for most outside and inside the field of GM on what is possible. For many, the application of GM to produce fake anything is abhorrent and nonessential, but the broad applications GM introduces can benefit many industries across many tasks.

In this book, we begin with the assumption you have limited or little knowledge of deep learning and generative modeling. You have a basic knowledge of programming Python and applying data science, including the typical fundamental math knowledge in calculus, linear algebra, and statistics used in data science.

We will cover a wide range of GM techniques and applications in this book, starting with the fundamentals of building a deep learning network and then progressing to GM. Here is a brief overview of the chapters we will explore:

1. **The Basics of Deep Learning**: We begin by introducing the basic concepts of deep learning, autoencoders, and how to build simple models with PyTorch. The examples in this chapter demonstrate simple concepts we will apply throughout this book and should not be missed by newcomers.

2. **Unleashing Generative Adversarial Networks**: This chapter moves to the fundamentals of explaining the generative adversarial network (GAN) and how it can be used to generate new and novel content. Examples in this chapter explore the applications of GANs from generating fashion to faces.

3. **Exploring the Latent Space**: Fundamental to generative modeling is the concept of learning the latent or hidden representation of something. In this chapter, we explore how the latent space is defined and how we can better control it through hyperparameters, loss function, and network configuration.

4. **GANs, GANs, and More GANs**: This book explores several variations of GANs, and in this chapter we look at five forms that attempt to learn the latent space differently. We build on knowledge from previous chapters to explore key differences in the way GANs learn and generate content.

5. **Image to Image Content Generation**: This chapter covers the advanced application of GANs to enhance the generation of content by learning through understanding translations. The examples in this chapter focus on showcasing paired and unpaired image translation using a variety of powerful GANs.

6. **Residual Network GANs**: Throughout this book we will constantly struggle with the generative ability to produce diverse and realistic features. The GANs in this chapter all use residual networks to help identify and learn more realistic feature generation.

7. **Attention Is All We Need**: This chapter explores the attention mechanism introduced into deep learning through the application of natural language processing. Attention provides a unique capability to identify and map relevant features with other features. The examples in this chapter demonstrate the power of using an attention mechanism with a GAN.

8. **Advanced Generators**: This chapter dives into the deep end and explores the current class of best-performing GANs. The examples in this chapter work from several open-source repositories that showcase how far the field of GM has come in a short time.
9. **Deepfakes and Face Swapping**: In this chapter, we switch gears and explore the application of GM for producing deepfakes. Where this whole chapter is dedicated to showcasing the ease of which you can produce a deepfake freely available open-source desktop software.
10. **Cracking Deepfakes**: From creating deepfakes and fake content for most of the book, we move on to understanding how generated content can be detected. This chapter looks at the techniques and research currently being done to expose fake content. In the future, these tools will be critical to controlling the digital reality we embrace and understanding what is real.

This book covers a wide range of complex subjects presented in a practical hands-on and technically friendly manner. To get the most out of this book, it is recommended that you engage and work with several of the 40+ examples. All the examples in this book have been tested and run to completion using Google Colab, the recommended platform for this book. While some examples in this book may take up to days to train, most can be run in under an hour.

Thank you for taking your precious time to read this book and ideally expand your opportunities and understanding in the field of AI/ML. The journey you have chosen is nontrivial and will be filled with frustration and anguish. It is one that will also be filled with awe and wonder the first time you generate your first fake face.

CHAPTER 1

The Basics of Deep Learning

Throughout history mankind has often struggled with making sense of what is real and what reality means. From hunter gatherers to Greek philosophers and then to the Renaissance, our interpretation of reality has matured over time. What we once perceived as mysticism is now understood and regulated by much of science. Not more than 10 years ago we were on track to understanding the reality of the universe, or so we thought. Now, with the inception of AI, we are seeing new forms of reality spring up around us daily. New realities being manifested by this new wave of AI are made possible by *neural networks* and *deep learning*.

Deep learning and neural networks have been on the fringe of computer science for more than 50 years, and they have their own mystique associated with them. For many, the abstract concepts and mathematics of deep learning make them inaccessible. Mainstream science shunned deep learning and neural networks for years, and in many industries they are still off-limits. Yet, among all those hurdles, deep learning has become the brave new leader in AI and machine learning for the 21st century.

In this book, we look at how deep learning and neural networks work at a fundamental level. We will learn the inner workings of networks and what makes them tick. Then we will quickly move on to understanding how neural networks can be configured to generate their own content and reality. From there, we will progress through many other versions of deep learning content generation including swapping faces, enhancing old videos, and creating new realities.

For this chapter, we will start at the basics of deep learning and how to build neural networks for several typical machine learning tasks. We will look at how deep learning can perform regression and classification of data as well as understand internally the process of learning. Then we will move on to understanding how networks can be

M. Lanham, *Generating a New Reality*, https://doi.org/10.1007/978-1-4842-7092-9_1

specialized to extract features in data with convolution. We will finish with building a full working image classifier using supervised deep learning.

As this is the first chapter, we will also cover several prerequisites and other helpful content to better guide your success through this book. Here is a summary of what we will cover in this chapter:

- Prerequisites
- Perceptrons
- Multilayer perceptrons
- PyTorch for deep learning
- Regression
- Classifying classes

This book will begin at the basics of data science, machine learning, and deep learning, but to be successful, be sure you meet most of the requirements in the next section.

Prerequisites

While many of the concepts regarding machine learning and deep learning should be taught at the high school level, in this book we will go way beyond the basic introduction of deep learning. Generating content with deep learning networks is an advanced endeavor that can be learned, but to be successful, it will be helpful if you meet most of the following prerequisites:

- **Interest in mathematics**: You don't need a degree in math, but you should have an interest in learning math concepts. Thankfully, most of the hard math is handled by the coding libraries we will use, but you still need to understand some key differences in math concepts. Deep learning and generative modeling use the following areas of mathematics:
    - **Linear algebra**, working with matrices and systems of equations

  - **Statistics and probability**, understanding how descriptive statistics work and basic probability theory
  - **Calculus**, understanding the basics of differentiation and how it can be used to understand the rate of change

- **Programming knowledge**: Ideally you have used and programmed with Python or another programming language. If you have no programming knowledge at all, you will want to pick up a course or textbook on Python. As part of your knowledge of programming, you may also want to take a closer look at the following libraries:
  - **NumPy**[1]: NumPy (pronounced "numb pie") is a library for manipulating arrays or tensors of numbers. It and the concepts it applies are fundamental to machine learning and deep learning. We will cover various uses of NumPy in this book, but it is suggested you study it further on your own as needed.
  - **PyTorch**[2]: This will be the basis for the deep learning projects in this book. It will be assumed you have little to no knowledge of PyTorch, but you may still want to learn more on your own what this impressive library has to offer.
  - **MatPlotLib**[3]: This module will be the foundation for much of the output we display in this book. There will be plenty of examples showing how it is used, but additional homework may be helpful.
- **Data science and/or machine learning**: It will be helpful if you have previously taken a data science course, one that covers the statistical methods used in machine learning and what aspects to be aware of when working with data.

[1]NumPy is an open source project at http://numpy.org.

[2]PyTorch is an open source project at http://pytorch.org.

[3]Matplotlib is an open source package heavily used with Python, available at https://matplotlib.org/.

- **Computer**: All the examples in this book are developed on the cloud, and while it is possible to use them with a mobile computing device, for best results it is suggested you use a computer.
    - Instructions have been provided in Appendix A for setting up and using the code examples on your local computer. This may be a consideration if you have a machine with an advanced GPU or need to run an example for more than 12 hours.
- **Time**: Generative modeling can be time-consuming. Some of the examples in this book may take hours and possibly days to run if you are up for it. In most cases, you will benefit more from running the example to completion, so please be patient.
- **Open to learn**: We will do our best to cover most of the material you need to use the exercises in this book. However, to fully understand some of these concepts, you may need to extend your learning outside this text. If your career is data science and machine learning or you want it to be, you likely already realize your path to learning will be continuous.

While it is highly recommended that you have some background in the prerequisites mentioned, you may still get by if you are willing to extend your knowledge as you read this book. There are many sources of text, blogs, and videos that you may find useful to help you fill in gaps in your knowledge. The primary prerequisites I ask you bring are an open mind and a willingness to learn.

In the next section, we jump into the foundation of neural networks, the perceptron.

The Perceptron

There is some debate, but most people recognize that the inspiration for neural networks was the brain, or, more specifically, the brain cell or neuron. Figure 1-1 shows the biological neuron over the top of a mathematical model called the *perceptron*. Frank Rosenblatt developed the basic perceptron model as far back as 1957. The model was later improved on to what is shown in the figure by Marvin Minsky in his book called *Perceptrons*. Unfortunately, the book was overly critical of the application of the

perceptron for anything other than simple Boolean logic problems like XOR. Much of this criticism was unfounded as we later discovered, but the fallout of this critique is often blamed for the first AI winter.

An *AI winter* is when all research and development using AI is stopped or placed in storage. These winters are often brought on by some major roadblock that stops progress in the field. The first winter was brought on by Minsky's critique of the perceptron and his belief that it could solve the XOR problem only. There have been two AI winters thus far. The dates of these winters are up for debate and may vary by exact discipline.

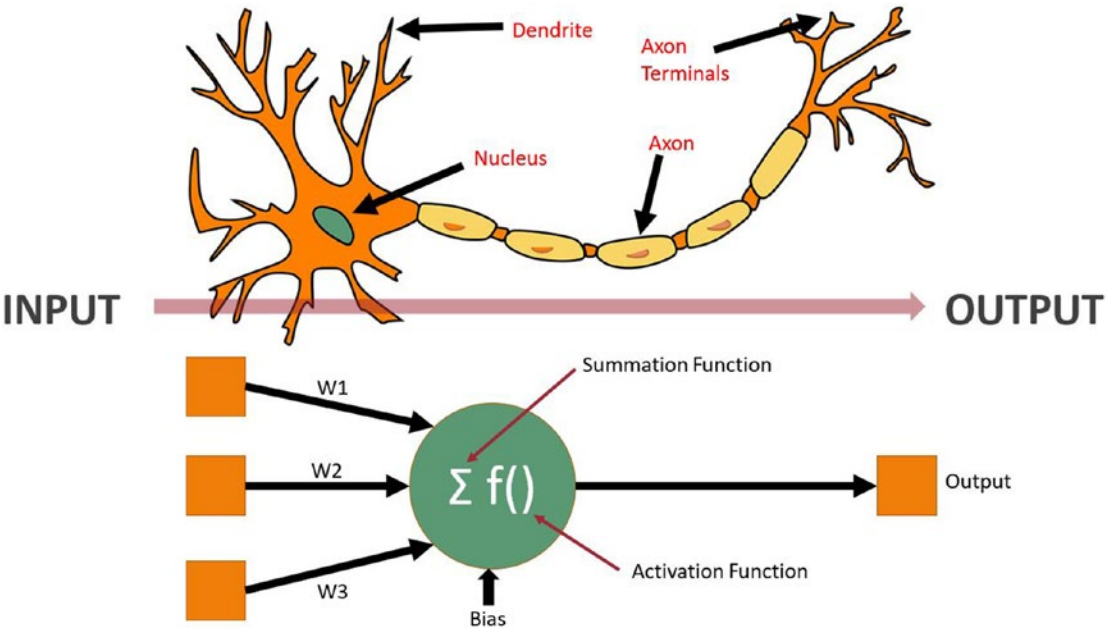

Figure 1-1. *A comparison of a biological neuron and the perceptron*

It is perhaps this association with the brain that causes some of the criticism with the perceptron and deep learning. This association also drives the mystique and uncertainty of neural networks. However, the perceptron itself is just a model of connectivity, and we may often refer to this type of learning as connectionism. If anything, the model of the perceptron only relates to a neuron in the way it connects and really nothing more. Actual neural brain function is far more complex and works nothing like a perceptron.

If we return to Figure 1-1 and the perceptron model, you can see how the system can take several inputs denoted by the boxes. These inputs are multiplied by a value we call a *weight* to weigh or adjust the strength of the input to the next stage. Before that, though,

we have another input called a *bias*, with a value of 1.0, that we multiply by another weight. The bias allows the perceptron to offset the results. After the inputs and bias are all weighed/scaled, they are then collectively summed in the summation function.

The results of the summation function are then passed to an activation function. The purpose of the activation function may be to further scale, squish, or cut off the value to be output. Let's take a look at how a simple perceptron can be modeled in code in Exercise 1-1.

EXERCISE 1-1. CODING A PERCEPTRON

1. Open the `GEN_1_XOR_perceptron.ipynb` notebook from the project's GitHub site. If you are unsure on how to access the source, check Appendix B.
2. In the first code block of the notebook, we can see some imports for NumPy and Matplotlib. Matplotlib is used to display plots.

```
import numpy as np
import matplotlib.pyplot as plt
```

3. Scroll to the XOR problem code block, as shown here. This is where the data is set up; the data consists of the `X` and `Y` values that we want to train the perceptron on. The `X` values represent the inputs, and the `Y` values denote the desired output. We will often refer to `Y` as the label or the expected output. We use the `numpy np` module to create the lists of inputs to a tensor using `np.array`. At the bottom of this block, we output the shape of these tensors.

```
X = np.array([[0,0],[0,1],[1,0],[1,1]])
Y = np.array([0,1,1,0])

print(X.shape)
print(Y.shape)
```

4. The values we are using for this initial test problem are from the XOR truth table shown here:

Inputs		Outputs
X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0

5. Scroll down and execute the following code block. This block uses the `matplotlib plt` module to output a 3D representation of the same truth table. We use array index slicing to display the first column of `X`, then `Y`, and finally the last column of `X` as the third dimension.

```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(X[:,0], Y, X[:,1], c='r', marker='o')
```

6. Our first step in coding a perceptron is determining the number of inputs and creating the weights for those inputs. We will do that with the following code. In this code, you can see we get the number of inputs by taking the first value of the `X.shape[1]`, which is 2. Then we randomly initialize the weights using `np.random.rand` and adding one input for the bias. Recall, the bias is a way the perceptron can offset a function.

```
no_of_inputs = X.shape[1]
weights = np.random.rand(no_of_inputs + 1)
print(weights.shape)
```

7. With the weights initialized to random values, we have a working perceptron. We can test this by running the next code block. In this block, we loop through the `inputs` called `X` and apply multiplication and addition using the dot product with the `np.dot` function. The output from this calculation yields the summation of the perceptron. The output of this code block will not mean anything yet since we still need to train the weights.

```
for i in range(len(X)):
  inputs = X[i]
  print(inputs)
  summation = np.dot(inputs, weights[1:]) + weights[0]
  print(summation)
```

8. In the next code block is the training code to train the weights in the perceptron. We always train a perceptron or neural network iteratively over the data called in a cycle called an *epoch*. During each epoch or iteration, we will feed each sample of our data into the perceptron or network either singly or in batches. As each sample is fed, we compare the output of the summation function to the `label` or expected value, `Y`. The difference between the prediction and label is called the *loss*. Based on this loss, we can then adjust the weights based on a formula we will review in detail later. The entire training code is shown here:

```
learning_rate = .1
epochs = 100
history = []
for _ in range(epochs):
  for inputs, label in zip(X, Y):
    prediction = summation = np.dot(inputs, weights[1:]) + weights[0]
    loss = label - prediction
    history.append(loss*loss)
    print(f"loss = {loss*loss}")
    weights[1:] += learning_rate * loss * inputs
    weights[0] += learning_rate * loss
```

9. After the last code cell is run, run the last code cell, shown here, that generates a plot of the loss, as shown in Figure 1-2.

```
plt.plot(history)
```

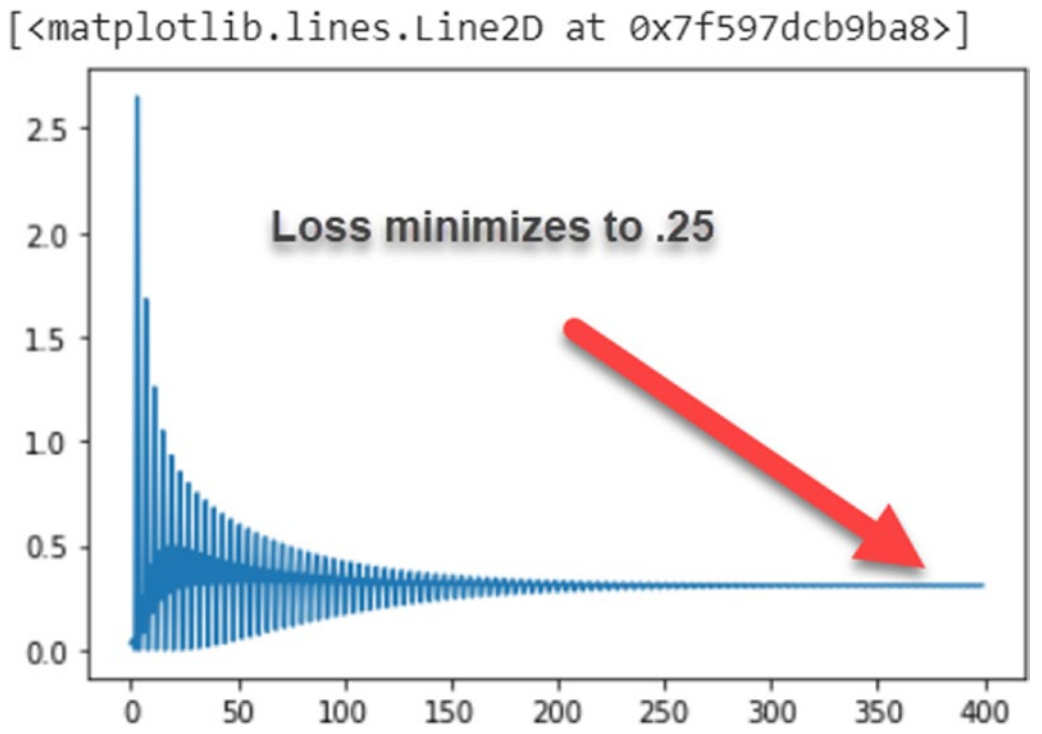

Figure 1-2. *Output of loss on XOR training of perceptron*

The results from this exercise are not so impressive. We were only able to obtain a minimized loss of .25. Feel free to continue running the example with more epochs or training cycles; however, the results won't get much better. This is the point Dr. Minsky was making in his book *Perceptrons*. A single perceptron or single layer of perceptrons is unable to solve the simple XOR problem. However, a single perceptron is able to solve some much harder problems.

Before we explore using the perceptron on a harder problem, let's revisit the learning lines of code from the previous example and understand how they work. For review, the learning lines of code are summarized here:

```
prediction = summation = np.dot(inputs, weights[1:]) + weights[0]
loss = label - prediction
...
weights[1:] += learning_rate * loss * inputs
weights[0] += learning_rate * loss
```

We already covered the summation/prediction function that uses `np.dot` to calculate. The loss is calculated by taking the difference from `label – prediction`. Then the weights are updated using the update function shown here:

$$W_i = W_i + \alpha * loss * input$$

where:

W_i = the weight that matches the input slot

α (alpha) = learning rate

loss = the difference from `label – prediction`

input = the input value for the input slot in the perceptron

This simple equation is what we use to update the weights during each pass of an input into the perceptron. The learning rate is used to scale the amount of update and is typically a value of .01, or 1 percent, or less. We want the learning rate to scale each update to a small amount; otherwise, each pass could cause the perceptron to over- and under-learn. The learning rate is the first in a class of variables we call *hyperparameters*.

Hyperparameters are a class of variables that we often need to tune manually. They are differentiated as hyperparameters since we refer to the internal weights as *parameters*.

The problem with a single perceptron or single layer of perceptrons is that they can solve a linear function only. The XOR problem is not a linear function. To solve XOR, we will need to introduce more than one layer of perceptron called a *multilayer perceptron*. Before we do that, though, let's revisit the perceptron and see what it is able to solve.

For the next exercise, we are going to look at a harder problem that can be solved with a linear method like the perceptron. The problem we will look at is solving a two-dimensional linear regression problem. Just 15 years ago, this class of problem would have been difficult to solve with typical regression methods. We will cover more about regression in a later section; for now let's jump into Exercise 1-2.

EXERCISE 1-2. LINEAR REGRESSION WITH A PERCEPTRON

1. Open the `GEN_1_perceptron_class.ipynb` notebook from the project's GitHub site. If you are unsure on how to access the source, check Appendix B.
2. This time we will run the linear regression problem code block to set the data, as shown here:

```
X = np.array([[1,2,3],[3,4,5],[5,6,7],[7,8,9],[9,8,7]])
Y = np.array([1,2,3,4,5])

print(X.shape)
print(Y.shape)
```

3. The next code block renders the input points on a graph:

```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(X[:,0], X[:,1], X[:,2], c='r', marker='o')
```

4. In this case, we display just the input points in 3D on the plot shown in Figure 1-3. Our goal in this problem is to train the perceptron so that it can learn how to map those points to our output labels, Y.

```
<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x7f597e2abb00>
```

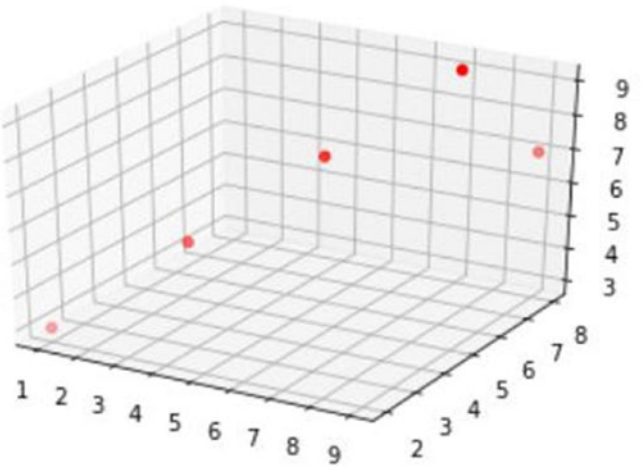

Figure 1-3. *Input points plotted on 3D graph*

5. We next move to the code section where we set up the parameters and hyperparameters. In this exercise, we have adjusted the hyperparameters, `epochs` and `learning_rate`. We decreased `learning_rate` to .01. Doing this effectively makes each update training pass or epoch less effective. However, in this case, the perceptron can learn to map those values much quicker than the XOR problem, so we will also reduce the number of epochs.

```
no_of_inputs = X.shape[1]
epochs = 50
learning_rate = .01
weights = np.random.rand(no_of_inputs + 1)
print(weights.shape)
```

6. For this exercise, we will introduce an activation function. An activation function scales the output for better input or prediction. In this example, we use a rectified linear function (ReLU). This function effectively negates output that is 0 or less and otherwise just passes the output linearly.

```
def relu_activation(sum):
  if sum > 0: return sum
  else: return 0
```

7. Next, we will embed the entire functionality of our perceptron into a Python class for better encapsulation and reuse. The following code is the combination of all our previous perceptron and setup code:

```
class Perceptron(object):
  def __init__(self, no_of_inputs, activation):
    self.learning_rate = learning_rate
    self.weights = np.zeros(no_of_inputs + 1)
    self.activation = activation

  def predict(self, inputs):
    summation = np.dot(inputs, self.weights[1:]) + self.weights[0]
    return self.activation(summation)

  def train(self, training_inputs, training_labels, epochs=100,
  learning_rate=0.01):
    history = []
    for _ in range(epochs):
```

```
        for inputs, label in zip(training_inputs, training_labels):
          prediction = self.predict(inputs)
          loss = (label - prediction)
          loss2 = loss*loss
          history.append(loss2)
          print(f"loss = {loss2}")
          self.weights[1:] += self.learning_rate * loss * inputs
          self.weights[0] += self.learning_rate * loss
      return history
```

8. We can instantiate and train this class with the following:

```
perceptron = Perceptron(no_of_inputs, relu_activation)
history = perceptron.train(X,Y, epochs=epochs)
```

9. Figure 1-4 shows the history output from the training function call and is a result of running the last group of cells. We can clearly see the loss is reduced to almost 0. This means our perceptron is able to predict and map the results given our inputs.

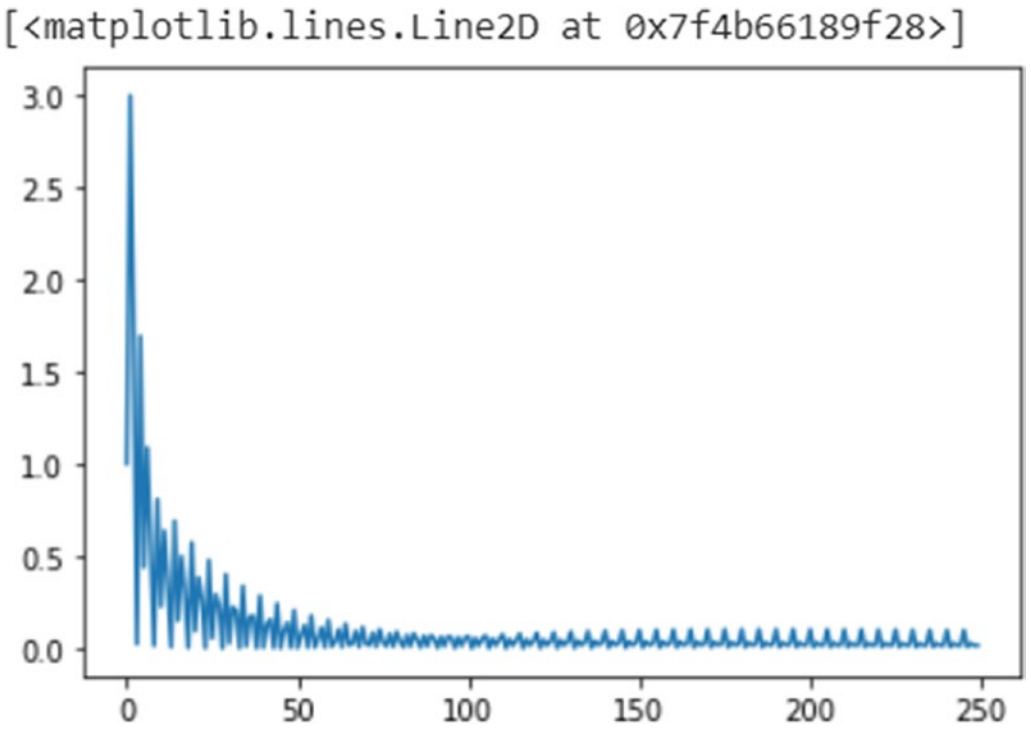

Figure 1-4. *Output loss of perceptron on linear regression problem*

You can see a noticeable wobble in the loss of the network in Figure 1-4. This wobble is caused in part by the learning rate, which is likely too high, and the way we are feeding the data into the network. We will look at how to resolve issues like this as we proceed through the book.

The results from this exercise were far more successful at mapping the inputs to expected outputs, even with a typically harder mathematical problem. Results like those we just witnessed are what kept the perceptron alive during the first cold AI winter. It wasn't until after this winter that we found the ability to stack perceptrons into layers could do far more and eventually solve the XOR problem. We will jump into the multilayer perceptron in the next section.

The Multilayer Perceptron

Fundamentally, the notion of stacking perceptrons into layers is not a difficult concept. Figure 1-5 demonstrates a three-layer multilayer perceptron (MLP). The top layer is called the *input layer*, the last layer the *output layer*, and the in-between layers the *middle* or *hidden layers*.

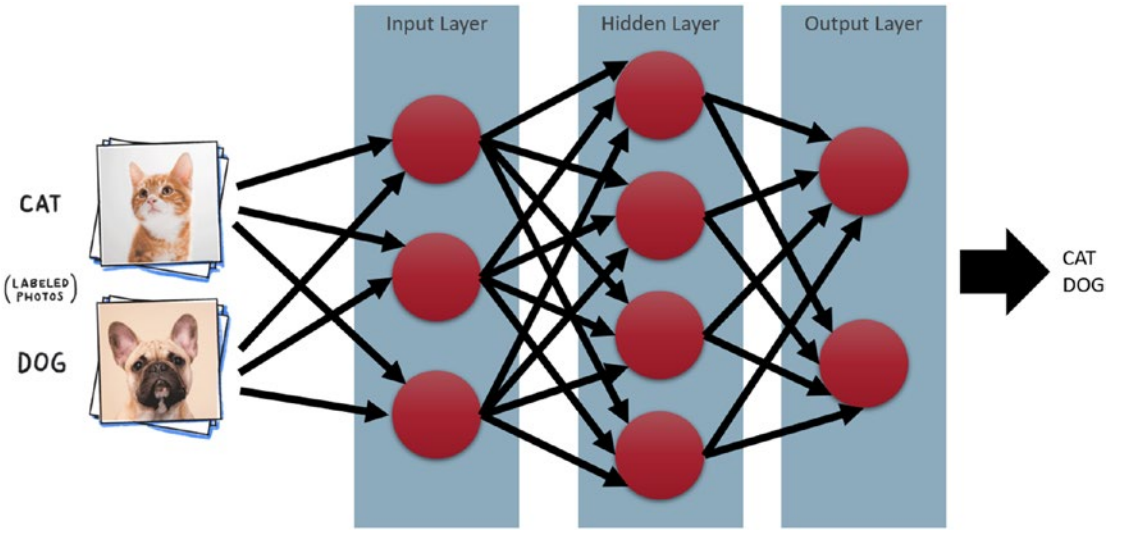

Figure 1-5. *Example of MLP network*

Figure 1-5 shows how we may feed images of cats and dogs to a network and have it classify the output. We will talk about how we can classify outputs later. Each node or circle in the figure represents a single perceptron, and each perceptron is fully connected to the successive layers in the network. The term we use for these types of networks is a *fully connected sequential network*.

The prediction of forward pass through the network runs the same as our perceptron, with the only difference that the output from the first layer becomes the input to the next, and so on. Calculating the output by passing an input into the network is called the *forward pass* or *prediction*. Computationally, through the use of the dot product function, the forward pass in DL is very efficient and one of the great strengths of neural networks.

If you recall from the previous section, the `np.dot` function we used did the summation of weights with the inputs. This function is optimized on a GPU to perform very quickly. So even if we had 1 million inputs (and yes, that is possible), the calculation could be done in one operation on a GPU.

The reason the `np.dot` function is optimized on a GPU is due to the advancement of computer 3D graphics. The dot product operation is quite common in graphics processing. In a sense, the development of games and graphics engines has been a big help for AI and deep learning.

While the forward pass or prediction step can run quickly, it is not exceedingly difficult to compute. Unfortunately, the opposite of training the updates or what we call the *backward pass* is not so easy. The problem we face when we stack perceptrons is that the simple update equation we applied before won't work across network layers.

The problem we encounter when updating the multiple layer networks is determining how much of the loss needs to be applied to not only which layer, but which perceptron in that layer. We can no longer just subtract the loss from the prediction and apply that value to a single weight. Instead, we need to calculate the impact of each weight applied to the resulting output or prediction.

To calculate how we can apply the loss to each weight, in each perceptron, and in each layer, we use calculus. Calculus allows us to determine the amount of loss to apply using differentiation. We use calculus to differentiate the forward or predict function along with the activation function. By differentiating these functions with respect to the weights, we can determine the amount of impact each weight contributes to the result.

Backpropagation

We call the update process or backward pass through network backpropagation since we are backpropagating the errors or loss to each weight. Figure 1-6 shows the backpropagation of error or loss back through the network. The figure also shows the equations to calculate this amount of error.

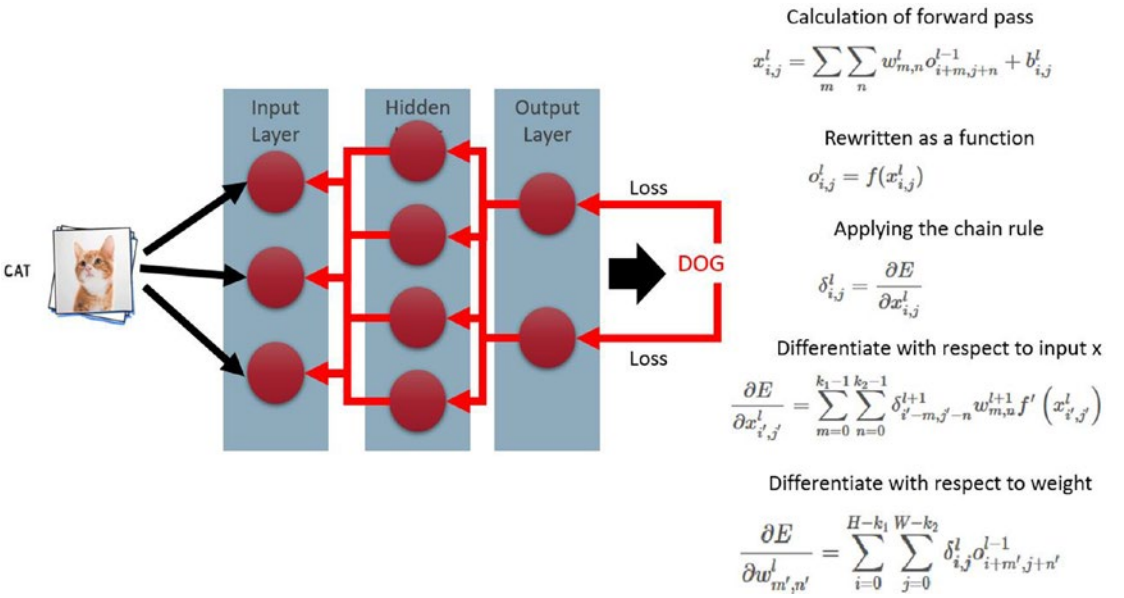

Figure 1-6. *Backpropagation explained*

The first equation in Figure 1-6 shows the calculation of the forward pass through the network. Moving to the next equation, we write this as a parameterized function. Following that, we apply the chain rule from calculus to differentiate the forward equation with respect to the input. By understanding how much each input affects the change, we can differentiate again this time with respect to the weights. The last equation shows how we calculate the change for each weight in the network.

Now, we don't have to worry about managing the mathematics or sorting out the equations to make this work. All deep learning libraries provide an automatic differentiation mechanism that does this for us. The critical bit to understand is how the last equation is used to push the loss back to each weight in the network. Output from this equation is a gradient describing the direction and amount of change. To perform the update, we reverse this gradient and scale it with the learning rate hyperparameter.