DEEP LEARNING ILLUSTRATED

JON KROHN
GRANT BEYLEVELD
AGLAÉ BASSENS

Deep Learning Illustrated

A Visual, Interactive Guide to Artificial Intelligence, First Editon

Jon Krohn
Beyleveld Grant
Bassens Aglaé

Addison-Wesley Professional

Part I. Introducing Deep Learning

1 Biological and Machine Vision

Throughout this chapter and much of this book, the visual system of biological organisms is used as an analogy to bring deep learning to, um... life. In addition to conveying a high-level understanding of what deep learning is, this analogy also provides insight into how deep learning approaches are so powerful and so broadly-applicable.

BIOLOGICAL VISION

Five hundred and fifty million years ago, in the prehistoric Cambrian Period, the number of species on the planet began to surge (Figure 1.1). From the fossil record, there is evidence [1] that this explosion was driven by the development of light detectors in the trilobite (Figure 1.2). A visual system, even a primitive one, bestows a delightful bounty of fresh capabilities. One can, as examples, spot food, foes, and friendly-looking mates at some distance. Other senses, like smell, enable animals to detect these as well, but not with the accuracy and light-speed pace of vision. Once the trilobite could see, the hypothesis goes, this set off an arms race that produced the Cambrian explosion: The trilobite's prey, as well as its predators, themselves had to evolve to survive.

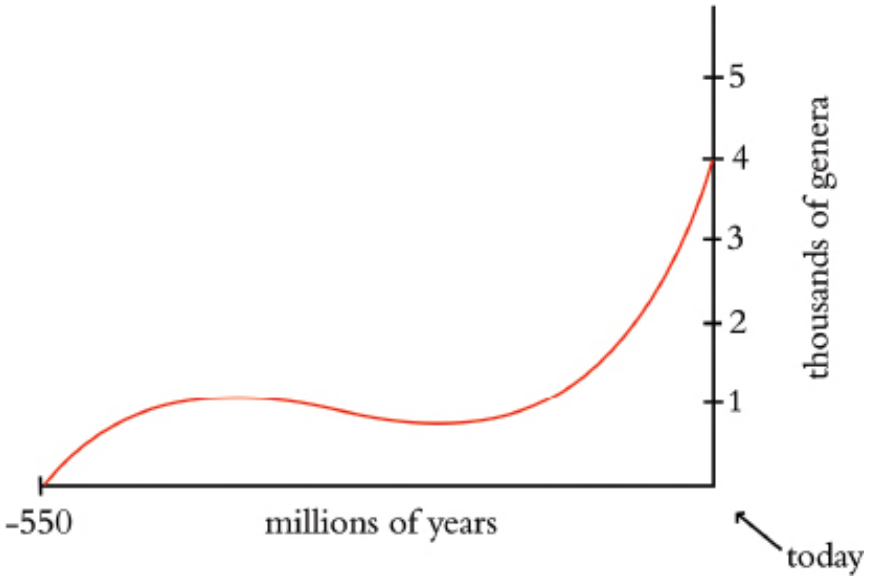

Figure 1-1 The "Cambrian explosion": the number of species on earth began to increase rapidly 550 million years ago, during the prehistoric Cambrian Period

Figure 1-2 A bespectacled trilobite

In the half-billion years since trilobites developed vision, the complexity of the sense has increased considerably. Indeed, in modern mammals, a large proportion of the *cerebral cortex*—the outer, grey matter of the brain [2] —is involved in visual perception. At Johns Hopkins University in the late 1950s, the physiologists David Hubel and Torsten Wiesel (Figure 1.3) began carrying out their pioneering research on how visual information is processed in the mammalian cerebral cortex, [3] work which contributed to them later being awarded a Nobel Prize. [4] As depicted in Figure 1.4, Hubel and Wiesel conducted their research by showing images to anaesthetized cats while simultaneously recording the activity of individual neurons from the *primary visual cortex*, the first part of the cerebral cortex to receive visual input from the eyes.

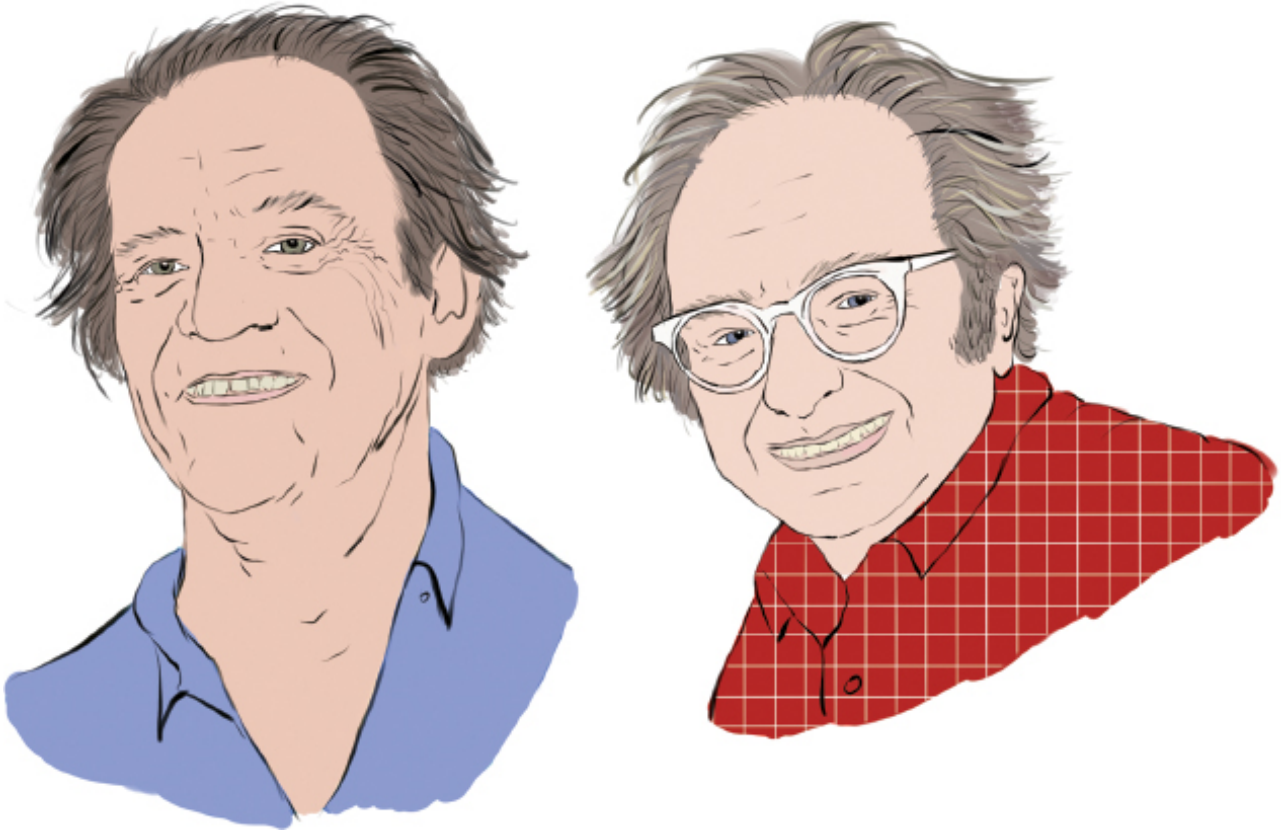

Figure 1-3 The Nobel Prize-winning neurophysiologists Torsten Wiesel and David Hubel

Projecting slides onto a screen, Hubel and Wiesel began by presenting simple shapes like the dot shown in Figure 1.4 to the cats. Their initial results were disheartening: Their efforts were met with no response from the neurons of the primary visual cortex. They grappled with the frustration of how these cells, which anatomically appear to be the gateway for visual information to the rest of the cerebral cortex, would not respond to visual stimuli. Distraught, Hubel and Wiesel tried in vain to stimulate the neurons by jumping and waving their arms in front of the cat. Nothing. And, then as with many of the great discoveries, from X-rays to penicillin to the microwave oven, Hubel and Wiesel made a serendipitous observation: As they removed one of their slides from the projector, its straight edge elicited the distinctive crackle of their recording equipment to alert them that a primary visual cortex neuron was firing. Overjoyed, they celebrated up and down the Johns Hopkins laboratory corridors.

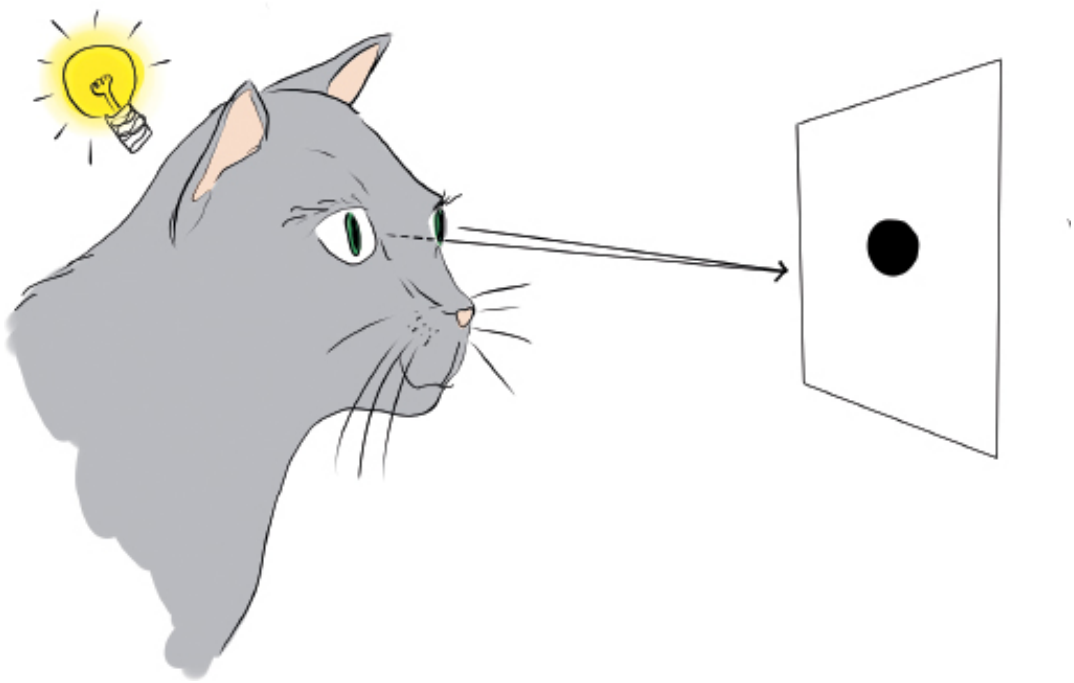

Figure 1-4 Hubel and Wiesel used a light projector to present slides to anaesthesized cats while they recorded the activity of neurons in the cats' primary visual cortex. In their experiments, electrical recording equipment was implanted within the cat's skull. Instead of illustrating this, we suspected it would be a fair bit more palatable to use a lightbulb to represent neuron activation. Depicted in this figure is a primary visual cortex neuron being serendipitously activated by the straight edge of a slide.

The serendipitously crackling neuron was not an anomaly. Through further experimentation, Hubel and Wiesel discovered that the neurons that receive visual input from the eye are in general most responsive to simple, straight edges. Fittingly then, they named these cells *simple* neurons.

As shown in Figure 1.5, Hubel and Wiesel determined that a given simple neuron responds optimally to an edge at a particular, specific orientation. A large group of simple neurons, with each specialized to detect a particular edge orientation, together are able to represent all 360 degrees of orientation. These edge-orientation detecting simple cells then pass along information to a large number of so-called *complex* neurons. A given complex neuron receives visual information that has already been processed by several simple cells so it is well-positioned to recombine multiple line orientations into a more complex shape like a corner or a curve.

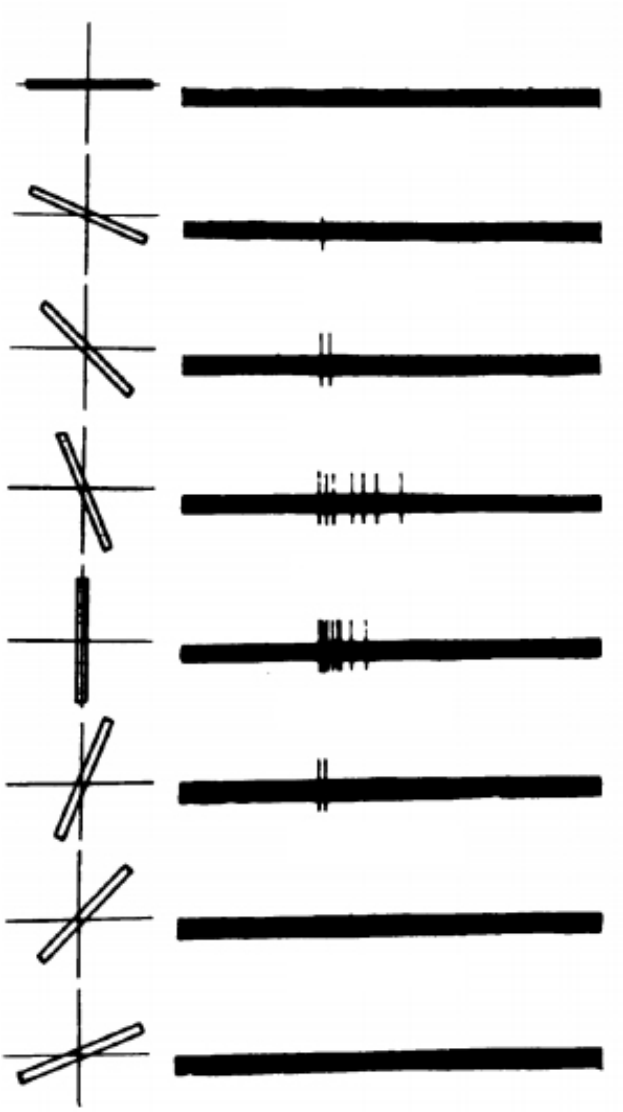

Figure 1-5 A "simple" cell in the primary visual cortex of a cat fires at different rates, depending on the orientation of a line shown to the cat. The orientation of the line is provided in the left-hand column of the figure, while the right-hand column shows the firing (electrical activity) in the cell over time (one second). A vertical line (in the fifth row) causes the most electrical activity for this particular simple cell. Lines slightly off vertical (in the intermediate rows) cause less activity for the cell, while lines approaching horizontal (in the top-most and bottom-most rows) cause little to no activity.

Figure 1.6 illustrates how, via many hierarchically-organized layers of neurons feeding information into increasingly higher-order neurons, gradually more complex visual stimuli can be represented by the brain. The eyes are focused on an image of a rat's head. Photons of light stimulate neurons located in the retina of each eye and this raw visual information is transmitted from the eyes to the primary visual cortex of the brain. The first layer of primary visual cortex neurons to receive this input—what Hubel and Wiesel termed *simple cells*—are specialized to detect edges (straight lines) at specific orientations. There would be many thousands of such neurons; for simplicity, we're only showing four. In our caricature, we're illustrating that neurons one, three, and four are activated by viewing the rat's head. These three simple neurons relay that information to a subsequent layer, where *complex cells* assimilate the information

about various edge orientations, enabling them to represent more complex visual stimuli, like the curvature of the rat's head. As information is passed through several subsequent further layers, the complexity and abstractness of the visual stimuli that can be represented incrementally increases. As depicted by the far-right layer of neurons, following many layers of such hierarchical processing, the brain is ultimately able to represent visual concepts as abstract as a rat, a cat, a bird or a dog.

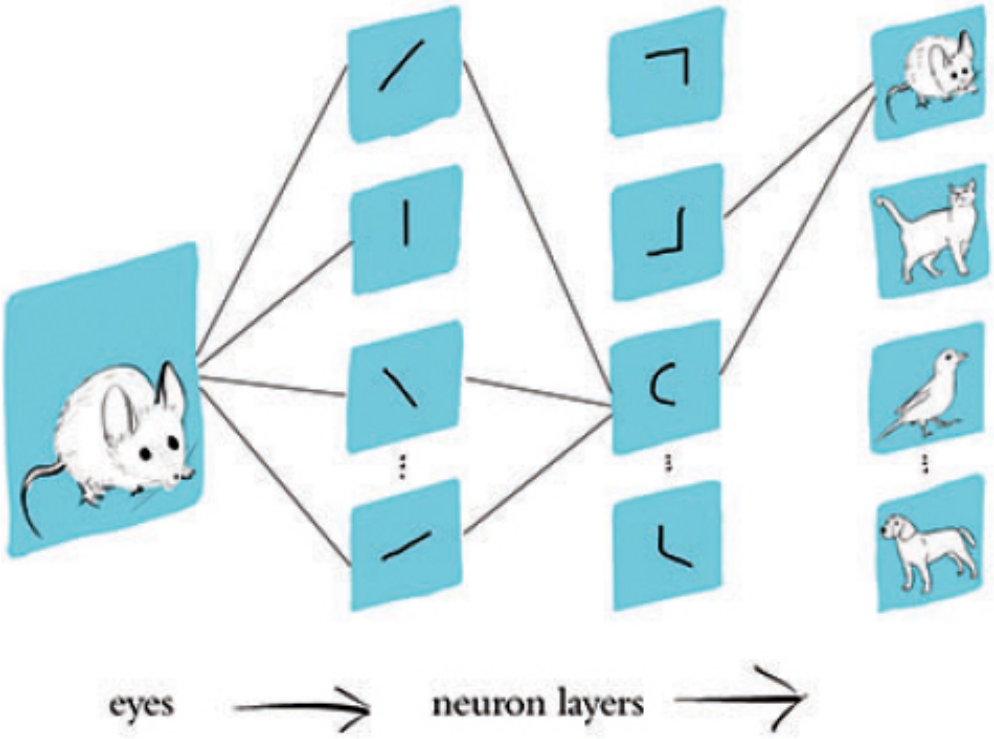

Figure 1-6 A caricature of how consecutive layers of biological neurons represent visual information in the brain of, e.g., a cat or a human. See main text for detail.

Today, through countless subsequent recordings from the cortical neurons of brain-surgery patients as well as non-invasive techniques like magnetic resonance imaging, [5] neuroscientists have pieced together a fairly high-resolution map of regions that are specialized to process particular visual stimuli, e.g., color, motion, faces (see Figure 1.7).

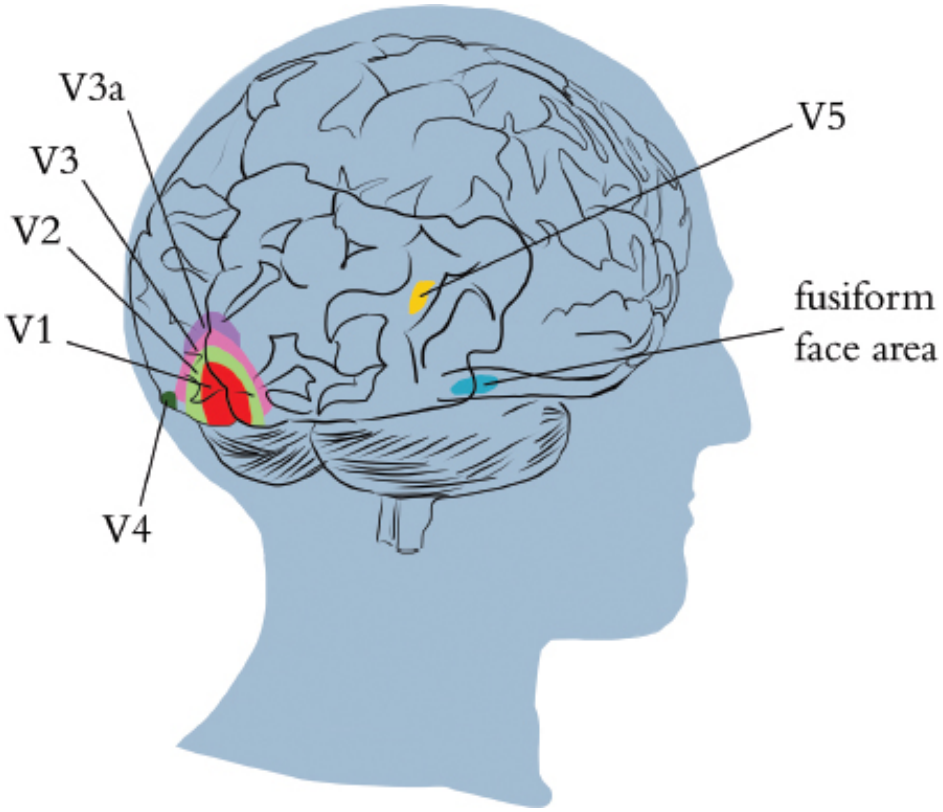

Figure 1-7 Regions of the visual cortex. The V1 region receives input from the eyes and contains the "simple" cells that detect edge orientations. Through the recombination of information via myriad subsequent layers of neurons (including within the V2, V3, and V3a regions), increasingly abstract visual stimuli are represented. In the human brain (shown here), there are regions containing neurons with concentrations of specializations in, as examples, the detection of color (V4), motion (V5), and people's faces (fusiform face area).

MACHINE VISION

We haven't been discussing the biological visual system solely because it's interesting (though hopefully you did find the previous section thoroughly interesting). We covered the biological visual system primarily because it served as the inspiration for the modern deep learning approaches to machine vision, as will become clear in this section.

Figure 1.8 provides a concise historical timeline of vision, in both biological organisms and machines. The top timeline, in blue, highlights the development of vision in trilobites as well as Hubel and Wiesel's 1959 publication on the hierarchical nature of the primary visual cortex, as covered in the previous section. The machine vision timeline is split into two parallel streams to call attention to two alternative approaches. The middle timeline, in pink, represents the deep learning track that is the focus of our book. The bottom timeline, in purple, meanwhile represents the traditional machine learning path to vision, which —through contrast —will clarify why deep learning is distinctively powerful and revolutionary.

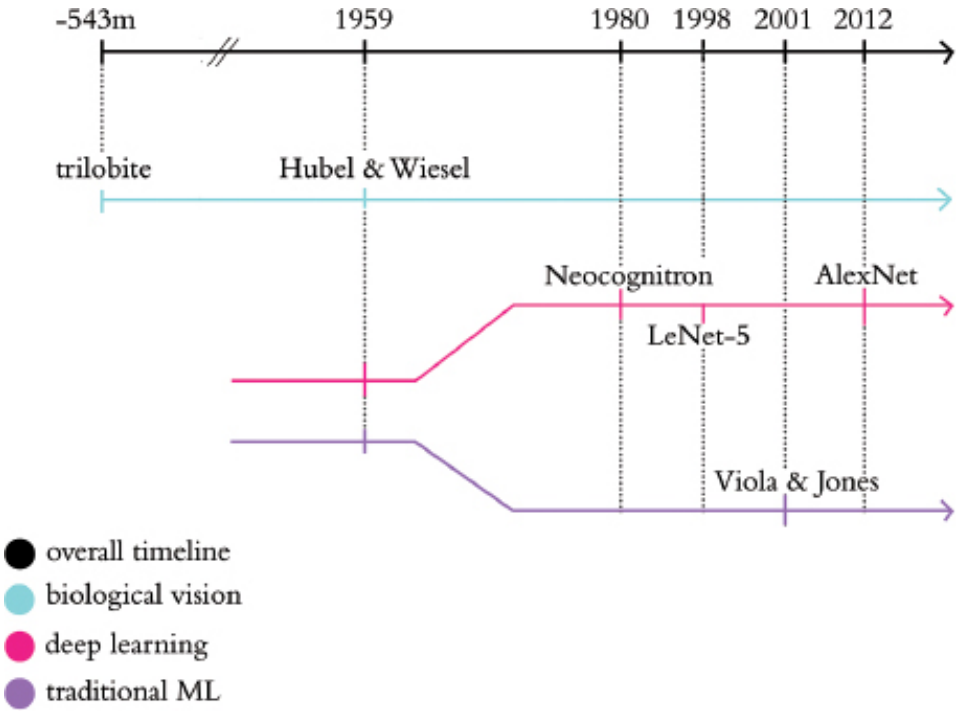

Figure 1-8 Abridged timeline of biological and machine vision, highlighting the key historical moments in the deep learning and traditional machine learning approaches to vision that are covered in this section.

The Neocognitron

Inspired by Hubel and Wiesel's discovery of the simple and complex cells that form the primary visual cortex hierarchy shown in Figure 1.6, in the late '70s the Japanese electrical engineer Kunihiko Fukushima proposed an analogous architecture for machine vision. [6] Figure 1.9 shows Fukushima's leading diagrams of this model architecture, which he named the *neocognitron*. Much of the detail of his diagrams is not important at this stage. There are, however, three particular items worth noting. First, Fukushima references Hubel and Wiesel explicitly; indeed, the paper refers to three of their landmark articles on the organization of the primary visual cortex. Second, Fukushima borrows the "simple" and "complex" cell language of Hubel and Wiesel, calling deeper (i.e., further right) layers *hypercomplex*. [7] The third point, and the most critical one, is that by arranging artificial neurons [8] in this hierarchical manner, they—like their biological inspiration—generally represent line orientations in the cells of the layers closest to the raw visual image (i.e., those on the far left, receiving input from the image U_0 in Figure 1.9) while successively deeper (i.e., further-right) layers represent successively complex, successively abstract objects. To make clear this potent property of the neocognitron and its deep learning descendants, we'll go through an interactive example at the end of this chapter that demonstrates it.

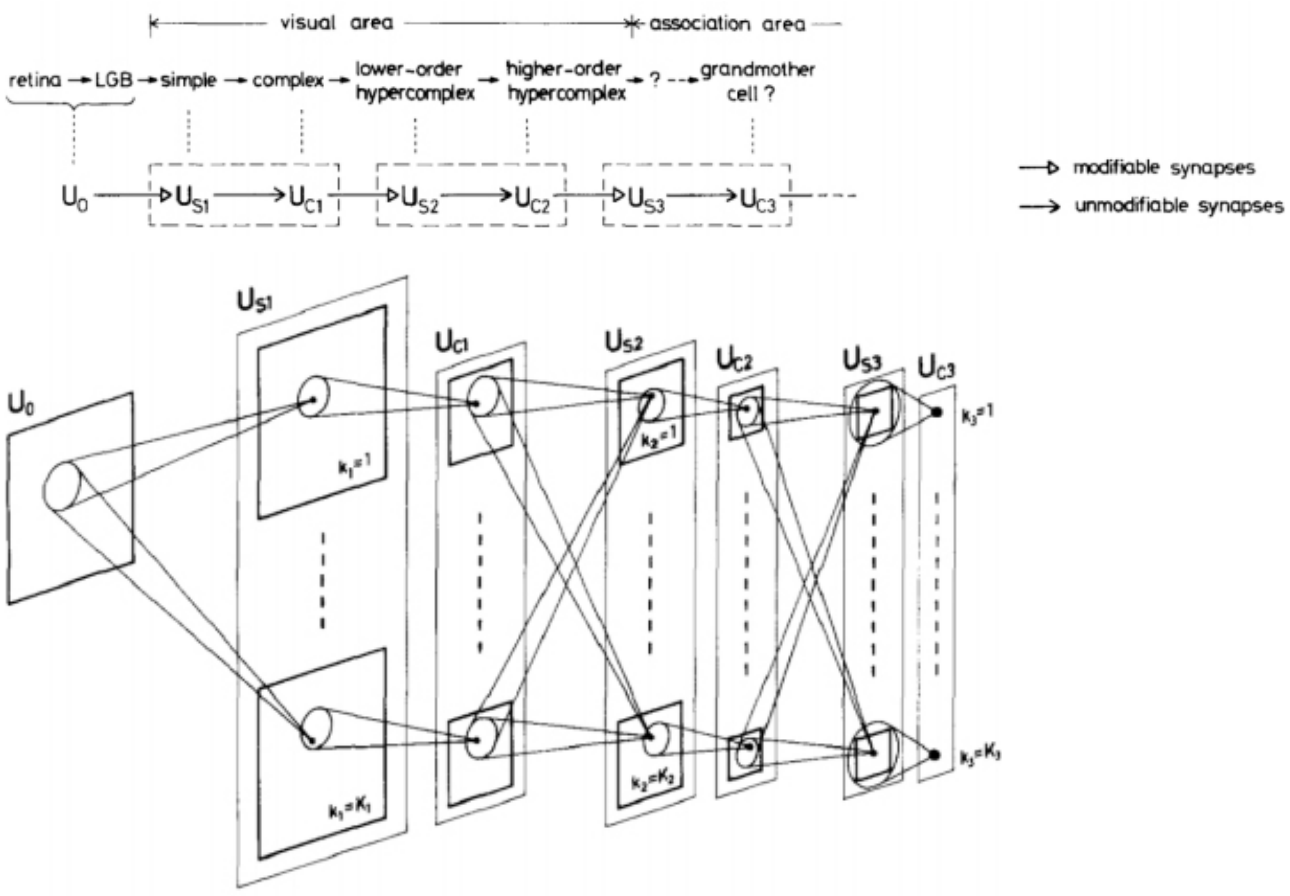

Figure 1-9 Diagrams of Kunihiko Fukushima's "neocognitron" from his 1980 paper. Borrowing Hubel and Wiesel's "simple" and "complex" cell language, Fukushima's artificial neural network model architecture emulates the hierarchy of the biological visual system. The image (U_0) is represented in the furthest-left layer of neurons (US_1) as edges (straight lines). As we move deeper (i.e., to the right), each successive layer facilitates increasingly complex and increasingly abstract visual representations. This is analogous to Figure 1.6, the caricature of hierarchical representations found in biological visual systems.

LeNet-5

While the neocognitron was capable of, for example, identifying handwritten characters, [9] the accuracy and efficiency of Yann LeCun (Figure 1.10) and Yoshua Bengio's (Figure 1.11) *LeNet-5* model[10] made it a significant development. LeNet-5's hierarchical architecture (Figure 1.12) built on Fukushima's lead and the biological inspiration uncovered by Hubel and Wiesel.[11] In addition, LeCun and his colleagues' benefited from superior data for training their model,[12] faster processing power and, critically, the backpropagation algorithm.

Figure 1-10 Paris-born Yann LeCun is one of the pre-eminent figures in artificial neural network and deep learning research. Professor LeCun is the Founding Director of the New York University Center for Data Science as well as the Director of AI Research at the social network Facebook.

Figure 1-11 Yoshua Bengio is another of the leading characters in artificial neural networks and deep learning. Born in France, he is a computer science professor at the University of Montreal and co-directs the renowned Machines and Brains program at the Canadian Institute for Advanced Research.

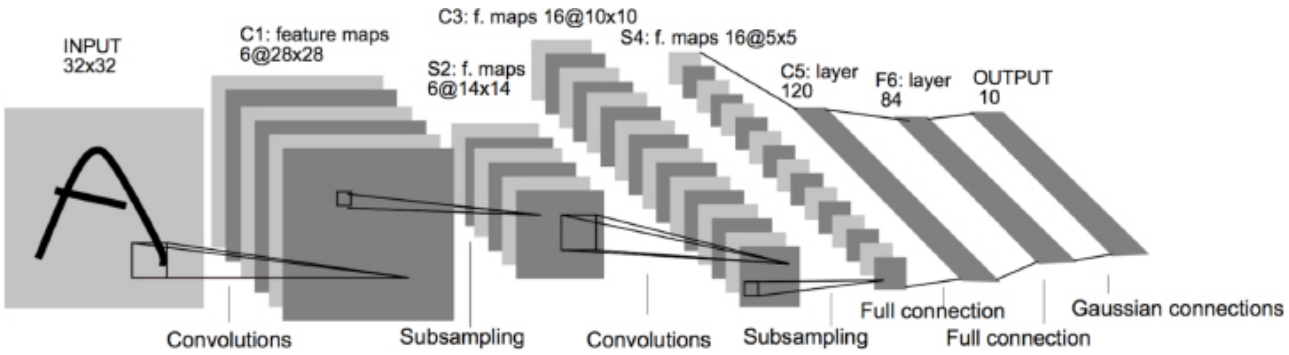

Figure 1-12 LeNet-5 retains the hierarchical architecture uncovered in the primary visual cortex by Hubel and Wiesel and leveraged by Fukushima in his neocognitron. As in those other systems, the left-most layer represents simple edges, while successive layers represent increasingly complex features.

Backpropagation, often abbreviated to *backprop*, facilitates efficient learning throughout the layers of artificial neurons within a deep learning model.[13] Together with their data and processing power, backprop rendered LeNet-5 sufficiently reliable to become an early commercial application of deep learning: It was used by the United States Postal Service to automate the reading of ZIP codes[14] written on mail envelopes. In Chapter 10, on machine vision, we will experience LeNet-5 first-hand by designing it ourselves and training it to (guess what!) recognize handwritten digits.

In LeNet-5, Yann LeCun and his colleagues had an algorithm that could correctly predict what handwritten digits had been drawn without them needing to include any expertise about handwritten digits in their code. As such, LeNet-5 provides an opportunity to introduce a fundamental difference between deep learning and the traditional machine learning ideology. As conveyed by Figure 1.13, the traditional machine learning (ML) approach is characterized by practitioners investing the bulk of their efforts into engineering features. This *feature engineering* is the application of clever, and often elaborate, algorithms to raw data in order to preprocess them into input variables that can be readily modeled by traditional statistical techniques. These techniques—e.g., regression, random forest, support vector machine—are seldom effective on unprocessed data, and so the engineering of input data has historically been a prime focus of machine learning professionals.

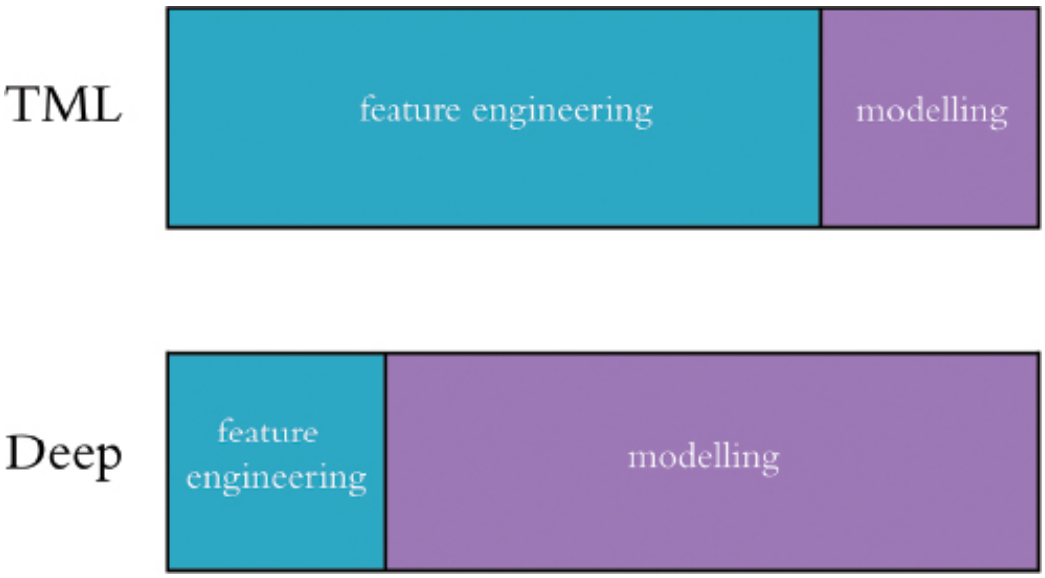

Figure 1-13 Feature engineering—the transformation of raw data into thoughtfully-transformed input variables—often predominates the application of traditional machine learning algorithms. In contrast, the application of deep learning often involves little to no feature engineering, with the majority of time spent instead on the design and tuning of model architectures.

In general, a minority of the traditional ML practitioner's time is spent optimizing ML models or selecting the most effective one from those available. The deep learning approach to modeling data turns these priorities upside-down. *The deep learning practitioner typically spends little to none of her time engineering features, instead spending it modeling data with various artificial neural network architectures that process the raw inputs into useful features automatically*. This distinction between deep learning and traditional machine learning is a core theme of this book. The next section provides a classic example of feature engineering to concretely explicate the distinction.

The Traditional Machine Learning Approach

Following LeNet-5, research into artificial neural networks, including deep learning, fell out of favor. The consensus became that the approach's automated feature generation was not pragmatic—that while it worked well for handwritten character recognition, the feature-free ideology was perceived to have limited breadth of applicability.[15] Traditional machine learning, including its feature engineering, appeared to hold more promise and funding shifted away from deep learning research.[16]

To make clear what feature engineering is, Figure 1.14 provides a celebrated example from Paul Viola and Michael Jones in the early noughties.[17] Viola and Jones employed rectangular filters such as the vertical or horizontal black and white bars shown in the figure. Features generated by passing these filters over an image can be fed into machine learning algorithms to reliably detect the presence of a face. Their work is

notable because the algorithm was efficient enough to be the first real-time face detector outside the realm of biology[18]. Devising clever face-detecting filters to process raw pixels into features for input into a machine learning model was accomplished via years of research and collaboration on the characteristics of faces. And, of course, it is limited to detecting faces in general, as opposed to being able to recognize a particular face as, say, Angela Merkel's or Oprah Winfrey's. To develop features for detecting Oprah in particular, or for detecting some non-face class of objects like houses, cars, or Yorkshire Terriers, would require the development of expertise in that category, which could again take years of academic-community collaboration to execute both efficiently and accurately. If only we could circumnavigate all that time and effort somehow…

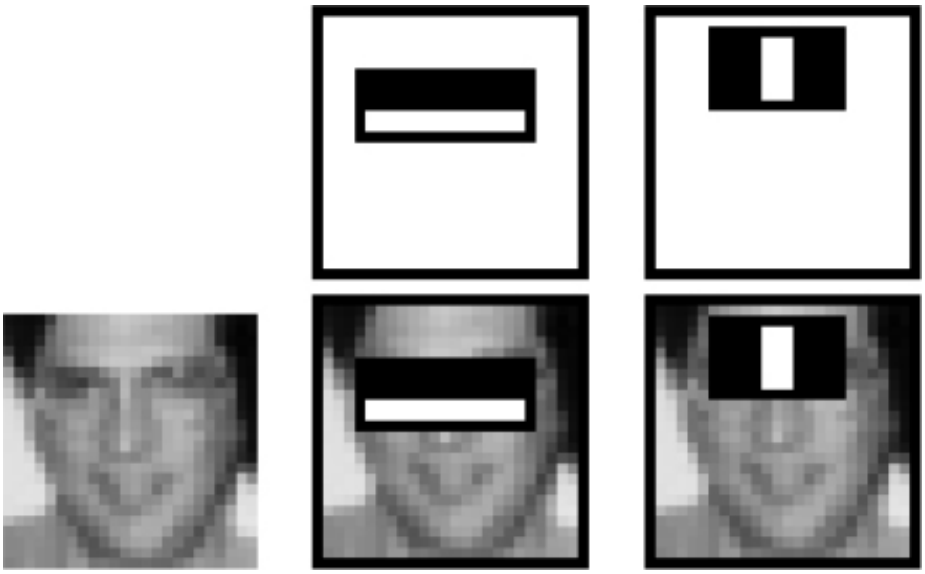

Figure 1-14 Engineered features leveraged by Viola and Jones (2001) to detect faces reliably. Their efficient algorithm found its way into FujiFilm cameras, facilitating real-time auto-focus.

ImageNet and the ILSVRC

As mentioned earlier, one of the advantages LeNet-5 had over the neocognitron was a larger, high-quality set of training data. The next breakthrough in neural networks was also facilitated by a high-quality public dataset—this time much larger: *ImageNet*, a labelled index of photographs devised by Fei-Fei Li (Figure 1.15), armed machine vision researchers with an immense catalog of training data.[19,20] For reference, the handwritten digit data used to train LeNet-5 contained tens of thousands of images. ImageNet, in contrast, contains tens of *millions*.

Figure 1-15 The hulking ImageNet data set was the brainchild of Chinese-American computer science professor Fei-Fei Li and her colleagues at Princeton at the time. In addition to her faculty position at Stanford, Li is the Chief Scientist of A.I./Machine Learning for Google's cloud platform.

The fourteen million images in the ImageNet data set are spread across 22,000 categories. These categories are as diverse as container ships, leopards, starfish and elderberries. Since 2010, Professor Li has run an open challenge called ILSVRC[21] on a subset of the ImageNet data that has become the premier ground for assessing the world's state-of-the-art machine vision algorithms. The ILSVRC subset consists of 1.4 million images across a thousand categories. In addition to providing a broad range of categories, many of the selected categories are breeds of dogs, thereby evaluating the algorithms' ability not only to distinguish broadly-varying images, but also to specialize in distinguishing subtly varying ones.[22]

AlexNet

As graphed in Figure 1.16, in the first two years of the ILSVRC all algorithms entered into the competition hailed from the feature-engineering-driven traditional machine learning ideology. In the third year, all entrants *except one* were traditional ML algorithms. If that one deep learning model in 2012 had not been developed or its creators not competed in ILSVRC, then the year-over-year image classification accuracy would have been negligible. Instead, Alex Krizhevsky and Ilya Sutskever—working out of the University of Toronto lab led by Geoffrey Hinton (Figure 1.17)—crushed the existing benchmarks with their submission, today referred to as AlexNet (Figure 1.18). This was a watershed moment. In an instant, deep learning architectures emerged from the fringes of machine learning to its fore. Academics and commercial practitioners scrambled to grasp the fundamentals of artificial neural networks as well as to create software libraries—many of them open-source—to experiment with deep learning models on their own data and use-cases, be they machine vision or otherwise. As Figure 1.16 illustrates, in the years since 2012 all of the top-performing models in the ILSVRC

have been deep learning-driven.

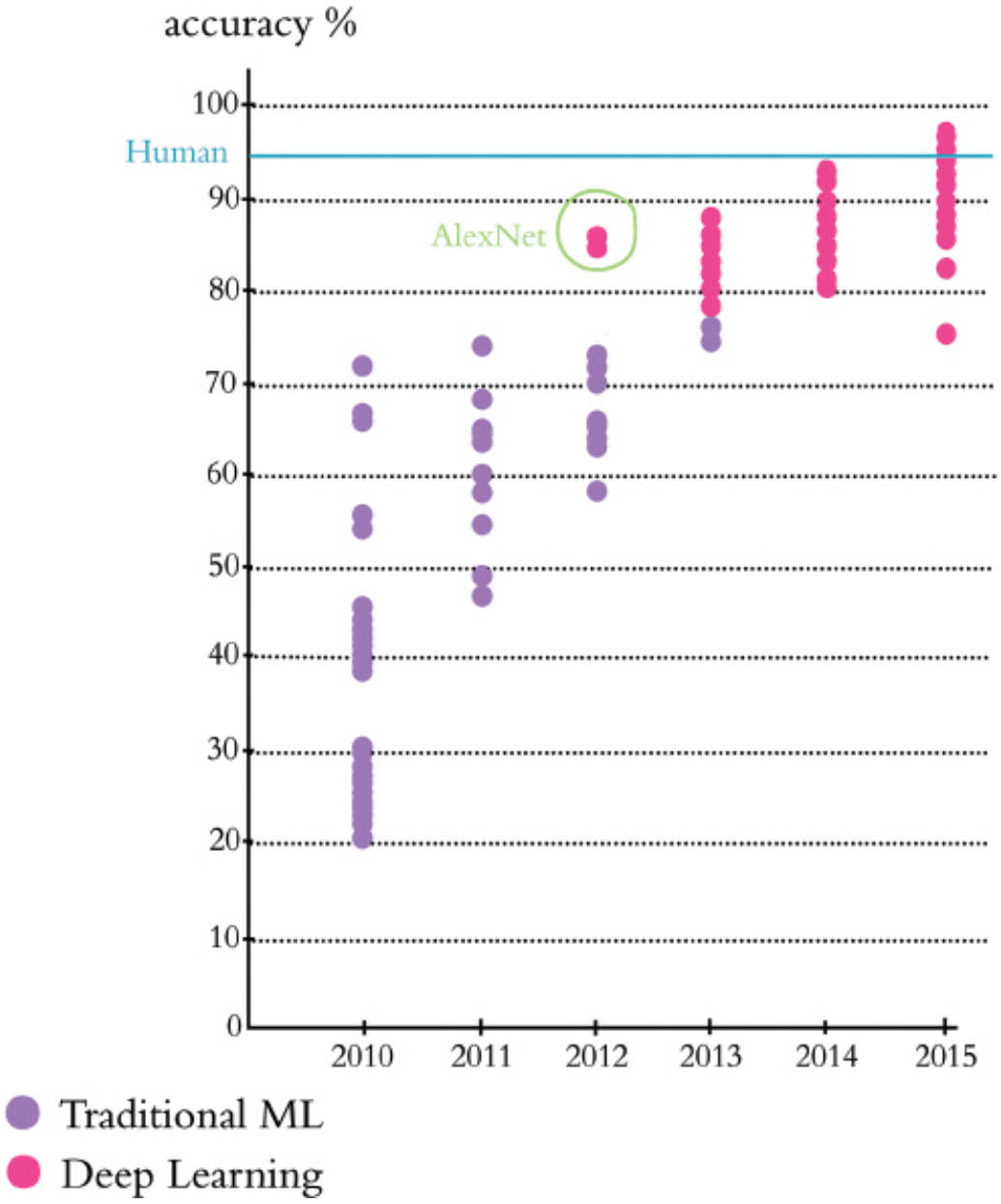

Figure 1-16 Performance of the top entrants to the ILSVRC by year. AlexNet was the victor by a head-and-shoulders margin in the 2012 iteration. All of the best algorithms since have been deep learning models. In 2015, machines surpassed human accuracy.

Figure 1-17 The eminent British-Canadian artificial neural network pioneer Geoffrey Hinton, habitually referred to as "the godfather of deep learning" in the popular press. Hinton is an Emeritus Professor at the University of Toronto and an Engineering Fellow at Google, responsible for managing the search giant's Brain Team, a research arm, in Toronto.

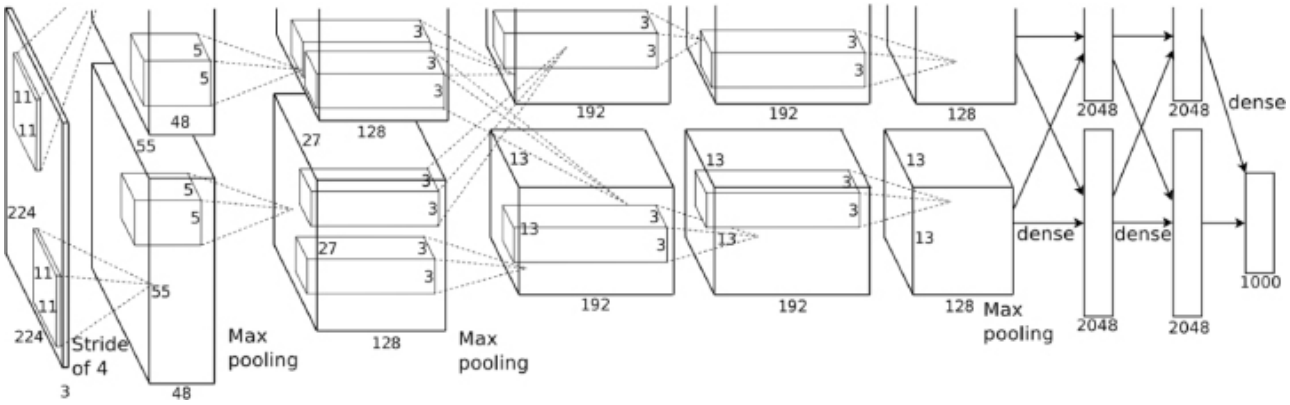

Figure 1-18 AlexNet's hierarchical architecture is reminiscent of LeNet-5 and the neocognitron with the first (left-hand) layer representing simple visual features like edges and deeper layers representing increasingly complex features and abstract concepts.

While the hierarchical architecture of AlexNet is reminiscent of LeNet-5, there are three principal factors that enabled it to be the state-of-the-art machine vision algorithm in 2012. First is the training data. Not only did Krizhevsky and his colleagues have access to the massive ImageNet index, they also artificially expanded the data available to them by applying transformations (e.g., horizontal reflection) to the training images. Second is processing power. Not only had computing power per unit of cost increased dramatically from 1998 to 2012, but Krizhevsky, Hinton and Sutskever also programmed two GPUs[23] to train their large data sets with previously unseen efficiency.

Third is architectural advances. AlexNet is deeper (has more layers) than LeNet-5, and it takes advantage of both a new type of artificial neuron[24] and a nifty trick[25] that helps generalize deep learning models beyond the data they're trained on. As with LeNet-5, we will build AlexNet ourselves in Chapter 10 and use it to classify images.

Our ILSVRC case study underlines how deep learning models like AlexNet are so widely useful and disruptive across industries and computational applications: They dramatically reduce the subject-matter expertise required for building highly accurate statistical models. This trend away from expertise-driven feature engineering and toward surprisingly powerful automatic-feature-generating deep learning models has been prevalently borne out across not only vision applications, but, as examples, the playing of complex games (the topic of Chapter 4) and natural language processing (Chapter 2) as well[26]. One no longer needs to be a specialist in the visual attributes of faces to create a face-recognition algorithm. One no longer requires a thorough understanding of a game's strategies to write a program that can master it. One no longer needs to be an authority on the structure and semantics of each of several languages to develop a language-translation tool. For a rapidly-growing list of use-cases, one's ability to apply deep learning techniques outweighs the value of domain-specific proficiency. While such proficiency may have necessitated a doctoral degree or perhaps years of postdoctoral research within a given domain, a functional level of deep learning capability can be developed with relative ease—as by working through this book!

TENSORFLOW PLAYGROUND

For a fun, interactive way to crystallize the hierarchical, feature-learning nature of deep learning, make your way to the *TensorFlow Playground* via the following URL: bit.ly/TFplayground. By using this custom link, your network should automatically look similar to the one shown in Figure 1.19. We'll be returning to define all of the terms on the screen in Part II; for the present exercise, they can be safely ignored. It suffices at this time to know that this is a deep learning model. The model architecture consists of six layers of artificial neurons: an input layer on the left (below the *FEATURES* heading), four "hidden" layers (which bear the responsibility of learning), and an output layer (the grid on the far right ranging from 6 to +6 on both axes). The network's goal is to learn how to distinguish orange dots (negative cases) from blue dots (positive cases) based solely on their location on the grid. As such, in the input layer, we are only feeding in two pieces of information about each dot: its horizontal position (X_1) and its vertical position (X_2). The dots that will be used as training data are shown by default on the grid. By clicking the *Show test data* toggle, you can also see the location of dots that will be used to assess the performance of the network as it learns. Critically, these

test data are not available to the network while it's learning, so they help us ensure that the network generalizes well to new, unseen data.

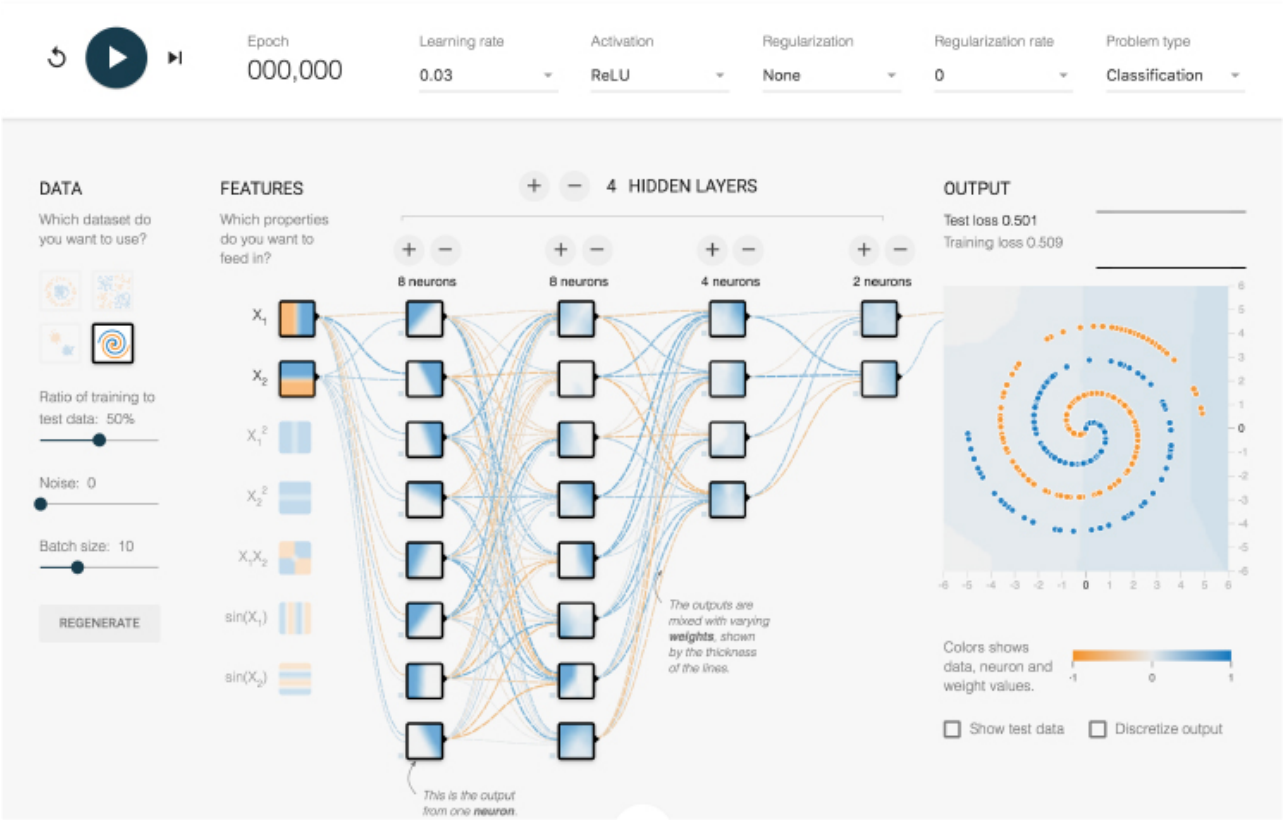

Figure 1-19 A deep neural network ready to learn how to distinguish a spiral of orange dots (negative cases) from blue dots (positive cases) based on their position on the X_1 and X_2 axes of the grid on the right.

Click the prominent *Play* arrow in the top-left corner. Enable the network to train until the "Training loss" and "Test loss" in the top-right corner have both approached zero, say less than 0.5. How long this takes will depend on the hardware you're using but will hopefully not be more than a few minutes.

As captured in Figure 1.20, you should now see the network's artificial neurons representing the input data with increasing complexity and abstraction the deeper (further to the right) they are positioned—as in the neocognitron, LeNet-5, and AlexNet. Every time the network is run, the neuron-level details of how the network solves the spiral classification problem are unique, but the general approach remains the same (you can refresh the page and re-train the network to see this for yourself). The artificial neurons in the left-most "hidden" layer are specialized in distinguishing edges (straight lines), each at a different particular orientation. Neurons from the first hidden layer pass information to neurons in the second hidden layer, each of which recombine the edges into slightly more complex features like curves. The neurons in each successive layer recombine information from the neurons of the previous layer, gradually increasing the complexity and abstraction of the features they can represent. By the final (right-most) layer, the neurons are adept at representing the intricacies of

the spiral shape, enabling the network to accurately predict whether a dot is orange (a negative case) or blue (a positive case) based on its position (X_1 and X_2 coordinates) in the grid. Hover over a neuron to project it onto the far-right *OUTPUT* grid and examine its individual specialization in detail.

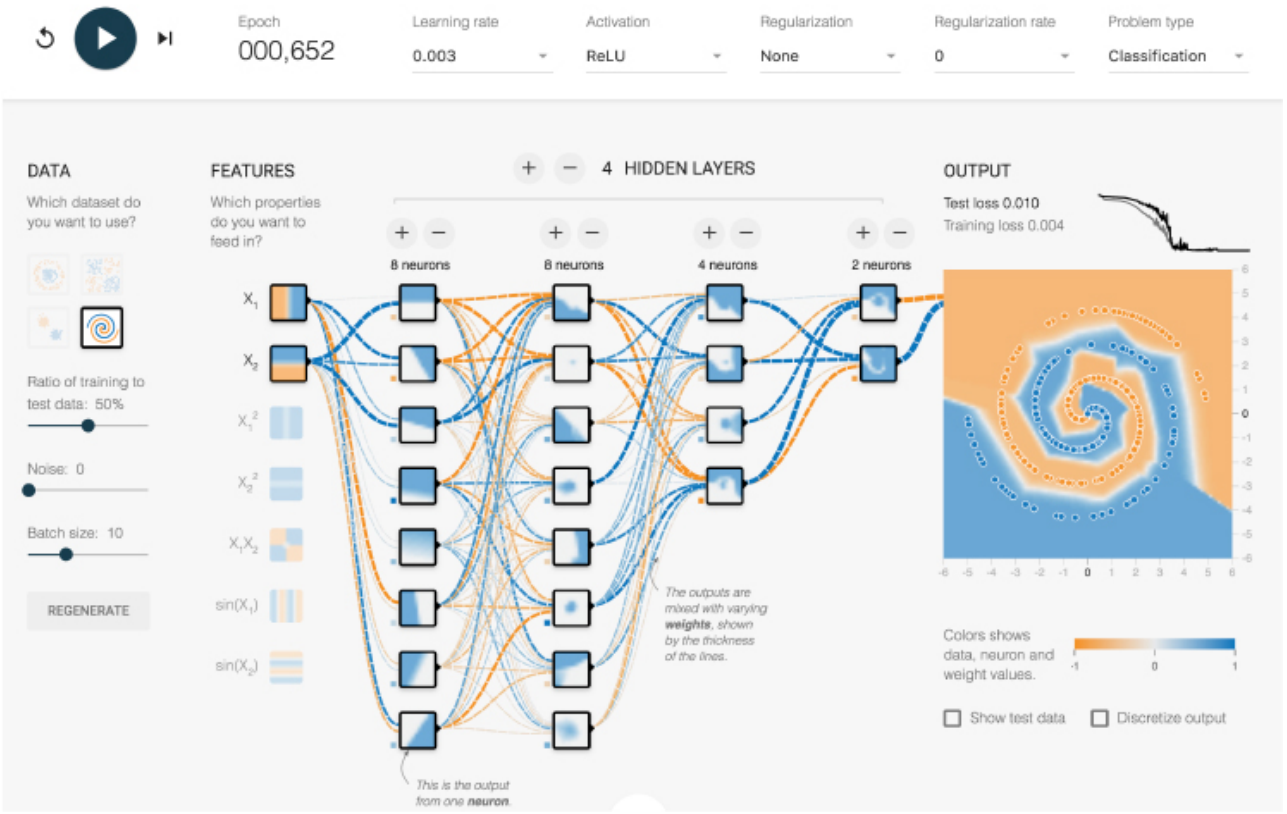

Figure 1-20 The network after training

QUICK, DRAW!

To interactively experience a deep learning network carrying out a machine vision task in real-time, navigate to quickdraw.withgoogle.com. Click *Let's Draw!* to begin playing the game. You will be prompted to draw an object and a deep learning algorithm will guess what you sketch. By the end of Chapter 10, we will have covered all of the theory and practical code examples needed to devise a machine vision algorithm akin to this one. To boot, the drawings you create will be added to the data set that we'll leverage in Chapter 12 when we create a deep learning model that can convincingly mimic human-drawn doodles. Hold onto your seat! We're embarking on a fantastic ride.

SUMMARY

Hopefully the parallel between biological vision and machine vision was clear to you. This is a theme that has popped up a few times in deep learning over the years: the ways in which deep learning models represent information are analogous with those same information processing systems in the natural world. Machine vision is a huge

subfield within deep learning, and one which has seen many great advancements in recent years. There is a staggering amount of image-based data available, and previously most of this data was difficult or impossible to access—we did not have a way for computers to *understand* what was happening in the images. We'll unpack the details of the various machine vision models in Chapter 10. *See* you there!

1. Parker, A. (2004). *In The Blink of an Eye: How Vision Sparked the Big Bang of Evolution*. New York: Basic Books.

2. Trilobite Reading SIDEBAR. A couple of tangential facts about the cerebral cortex: First, it is one of the more evolutionary-recent developments of the brain, contributing to the complexity of mammal behavior relative to the behavior of older classes of animals like reptiles and amphibians. Second, while the brain is informally referred to as *grey matter* because the cerebral cortex is the brain's external surface and this cortical tissue is grey in color, the bulk of the brain is in fact *white* matter. By and large, the white matter is responsible for carrying information over longer distances than the grey matter, so its neurons are coated in a white-colored, fatty covering that hurries the pace of signal conduction. A coarse analogy could be to consider neurons in the white matter to act as "highways". These high-speed motorways have scant on-ramps or exits, but can transport a signal from one part of the brain to another lickety-split. In contrast, the "local roads" of grey matter facilitate myriad opportunities for interconnection between neurons at the expense of speed. A gross generalization, therefore, is to consider the cerebral cortex—the grey matter—as the part of the brain where the most complex computations happen, affording the animals with the largest proportion of it—e.g., mammals, particularly the great apes like *Homo sapiens*—their complex behaviors.

3. Hubel, D. H., & Wiesel, T. N. (1959) Receptive fields of single neurones in the cat's striate cortex. *The Journal of Physiology, 148,* 574-91.

4. The 1981 Nobel Prize in Physiology or Medicine, shared with American neurobiologist Roger Sperry.

5. Especially *functional* MRI, which provides insight into which regions of cerebral cortex are notably active or inactive when the brain is engaged in a particular activity

6. Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a

mechanism of pattern recognition unaffected by shift in position. *Biological Cynbernetics, 36,* 193-202.

7 . Trilobite Attention SIDEBAR The theoretical concept of a single "grandmother cell" representing one's mental percept of their grandmother has not been convincingly demonstrated in the biological brain, where she is perhaps represented by the coordinated activation of an ensemble of neurons. As we shall see in the examples in this book, however, in machine-vision systems we *do* typically configure our model architectures so that the final layer contains individual artificial neurons representing individual concepts (e.g., your grandmother, Barack Obama, the number seven, a school bus) and this works very well.

8 . We will define precisely what these are in Chapter 7, "Artificial Neural Networks". For the moment, it's more than sufficient to think of each artificial neuron as a speedy, little algorithm.

9 . Fukushima, K., & Wake, N. (1991). Handwritten alphanumeric character recognition by the neocognitron. *IEEE Transactions on Neural Networks, 2,* 355-65.

10. LeCun, Y., et al. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE, 2,* 355-65.

11. LeNet-5 was the first *Convolutional Neural Network*, a deep learning variant that dominates modern machine vision and that we'll detail in Chapter 10.

12. Their classic data set, the handwritten MNIST digits, will be detailed later in Part II, "Essential Theory Illustrated".

13. We will detail the backpropagation algorithm later in Chapter 7.

14. The USPS term for postal code

15. At the time, there were stumbling blocks associated with optimizing deep learning models that have since been resolved, including poor weight initializations (covered in Chapter 9), covariate shift (also in Chapter 9) and the predominance of the relatively inefficient sigmoid activation function (Chapter 6).

16. Public funding for artificial neural network research ebbed globally, with the notable exception of continued support from the Canadian federal government enabling, e.g., the Universities of Montreal, Toronto, and Alberta to become

powerhouses in the field.

17. Viola, P., & Jones, M. (2001). Robust real-time face detection. *International Journal of Computer Vision, 57,* 137-54.

18. A few years later, the algorithm found its way into digital FujiFilm cameras, facilitating auto-focus on faces for the first time—a now everyday attribute of digital cameras and smartphones alike.

19. www.image-net.org

20. Deng, J., et al. (2009). ImageNet: A large-scale hierarchical image database. *Proceedings of the Conference on Computer Vision and Pattern Recognition*.

21. ImageNet Large Scale Visual Recognition Challenge

22. On your own time, try to distinguish photos of Yorkshire Terriers from Australian Silky Terriers. It's tough, but Westminster Dog Show judges, as well as contemporary machine vision models, can do it. Tangentially, these dog-heavy data are why deep learning models trained with ImageNet have a disposition toward "dreaming" about dogs (see, e.g., deepdreamgenerator.com).

23. Graphical Processing Units: These are designed primarily for rendering video games but are well-suited to performing the matrix multiplication that abounds in deep learning across hundreds of parallel computing threads.

24. The Rectified Linear Unit, which will be introduced in Chapter 7

25. Dropout, introduced in Chapter 9

26. An especially entertaining recounting of the disruption to the field of machine translation is provided by Gideon Lewis-Kraus in his article "The Great A.I. Awakening", published in *The New York Times Magazine* on December 14th, 2016.

2 Human and Machine Language

In the previous chapter, we introduced the high-level theory of deep learning via analogy to the biological visual system. All the while, we highlighted that one of the technique's core strengths lies in its ability to learn features automatically from data. In this chapter, we'll build atop our deep learning foundations by examining how it's incorporated into human language applications, with a particular emphasis on how it can automatically learn features that represent the meaning of words.

The Austro-British philosopher Ludwig Wittgenstein famously argued, in his posthumous and seminal work *Philosophical Investigations*, "The meaning of a word is its use in the language." [1] He further orated that, "One cannot guess how a word functions. One has to look at its use, and learn from that." Wittgenstein was suggesting that words on their own have no real meaning; rather, it is by their use within the larger context of language we're able to ascertain their meaning. As you'll see through this chapter, natural language processing with deep learning relies heavily on this premise —*word2vec* quite literally derives its semantic understanding of a word by analyzing it within its contexts across a large corpus.

Armed with this notion, let's begin by breaking down deep learning for natural language processing as a discipline, and then we'll go on to discuss modern deep learning techniques for representing words and language. By the end of the chapter, you should have a good grasp on what is possible with deep learning and NLP, the groundwork for writing such code in Chapter 11.

DEEP LEARNING FOR NATURAL LANGUAGE PROCESSING

The two core concepts in this chapter are *deep learning* and *natural language processing*. Initially, we'll cover the relevant aspects of these concepts separately, then we'll weave them together as the chapter progresses.

Deep Learning Networks Learn Representations Automatically

As established way back in this book's introduction, deep learning can be defined as the layering of simple algorithms called *artificial neurons* into networks several layers

deep. Via the Venn diagram in Figure 2.1, we show how deep learning resides within the machine learning family of *representation learning* approaches. The representation learning family, which contemporary deep learning dominates, includes any techniques that learn features from data automatically. Indeed, we can use the terms "feature" and "representation" interchangeably.

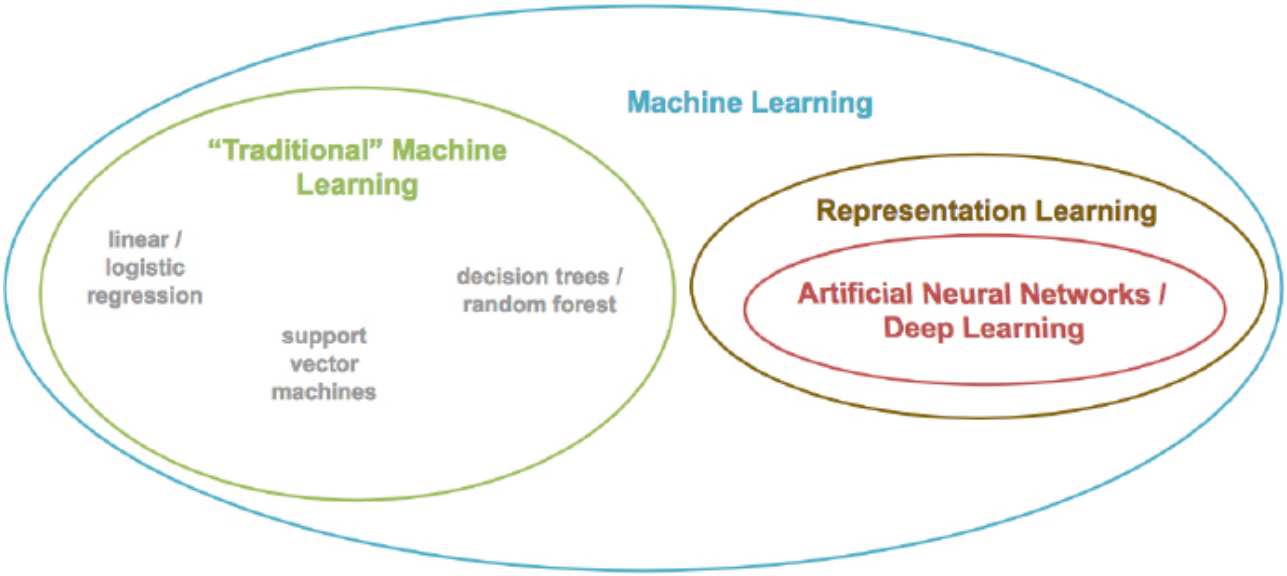

Figure 2.1 Venn diagram that distinguishes the "traditional" family from the "representation learning" family of machine learning techniques.

Figure 1.13 summarised the advantage of representation learning relative to traditional machine learning approaches. Traditional ML typically works well because of clever, human-designed code that transforms raw data—whether be it images, audio of speech, or text from documents—into input features for machine learning algorithms (e.g., regression, random forest, support vector machines) that are adept at weighting features but not particularly good at learning features from raw data directly. This manual creation of features is often a highly-specialized task. For working with language data, for example, it might require graduate-level training in linguistics. A primary benefit of deep learning is that it eases this requirement for subject-matter expertise. Instead of manually curating input features from raw data, the data can be fed directly into a deep learning model. Over the course of many examples provided to the deep learning model, the first layer of artificial neurons receiving the input data learn how to represent simple abstractions of these data, while each successive layer learns to represent increasingly complex non-linear abstractions on the layer that precedes it. As we'll discover in the current chapter, this isn't solely a matter of convenience; learning features automatically has additional advantages. Features engineered by humans tend to not be comprehensive, tend to be excessively specific, and can involve lengthy, ongoing loops of feature ideation, design and validation that could stretch for years. representation learning models, meanwhile, generate features quickly (typically over hours or days of model training), adapt straightforwardly to

changes in the data (e.g., new words, meanings, or ways of using language), and adapt automatically to shifts in the problem being solved.

Natural Language Processing

Natural language processing (NLP) is a field of research that sits at the intersection of computer science, linguistics, and "artificial intelligence" (Figure 2.2). NLP involves taking the naturally-spoken or naturally-written language of humans—like this sentence you're reading right now—and processing it with machines to automatically complete some task or to make a task easier for a human to do. Examples of language use that do not fall under the umbrella of *natural* language could include code written in a software language or short strings of characters within a spreadsheet.

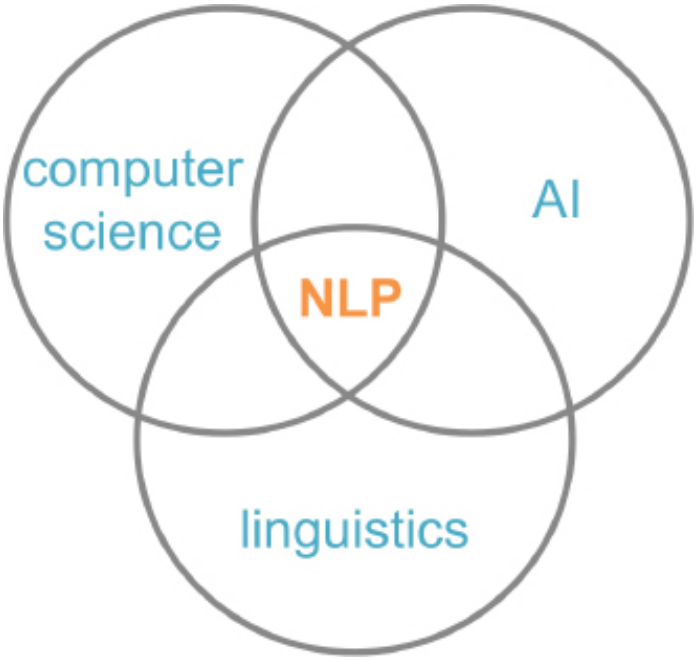

Figure 2.2 NLP sits at the intersection of the fields computer science, linguistics and artificial intelligence.

Examples of NLP in industry include:

œ *classifying documents*: using the language within a document (e.g., an email, a Tweet, or a review of a film) to classify it into a particular category (e.g., high urgency, positive sentiment, or predicted direction of the price of a company's stock)

œ *machine translation*: assisting language-translation firms with machine-generated suggestions from a source language (e.g., English) to a target language (e.g., German or Mandarin); increasingly, fully-automatic—though not always perfect—translations between languages

œ *search engines*: autocompleting users' searches and predicting what information or website they're seeking

- *speech recognition*: interpret voice commands to provide information or take action, as with virtual assistants like Amazon's Alexa, Apple's Siri or Microsoft's Cortana

- *chatbots*: modern chatbots fall short of convincingly carrying out a natural conversation for an extended period of time, but are nevertheless helpful for relatively linear conversations on narrow topics like the routine components of a given firm's customer-service phone calls

Some of the easiest NLP applications to build are spell-checkers, synonym-suggesters and keyword-search querying tools. These simple tasks can be fairly straightforwardly solved with deterministic, rules-based code using say, reference dictionaries or thesauruses. deep learning models are unnecessarily sophisticated for these applications and so they won't be discussed further in this book.

Intermediate-complexity NLP tasks include assigning a school-grade reading level to a document, predicting the most likely next words while making a query in a search engine, classifying documents (see above), and extracting information from documents or websites like prices or named entities. [2] These intermediate NLP applications are well-suited to solving with deep learning models. In Chapter 11, for example, we'll leverage a variety of deep learning architectures to predict the sentiment of film reviews.

The most sophisticated NLP implementations are required for machine translation (see above), automated question-answering and chatbots. These are tricky because they need to handle application-critical nuance (as an example, humor is particularly transient), a response to a question can depend on the intermediate responses to previous questions, and meaning can be conveyed over the course of a lengthy passage of text consisting of many sentences. Complex NLP tasks like these are beyond the scope of this book, however the content we cover will serve as superb foundations for their development.

A Brief History of Deep Learning for NLP

The timeline in Figure 2.3 calls out recent milestones in the application of deep learning to NLP. This timeline begins in 2011, when the University of Toronto computer scientist George Dahl and his colleagues at Microsoft Research revealed the first major breakthrough involving a deep learning algorithm applied to a large data set. [3] This breakthrough happened to involve natural language data. Dahl and his team trained a deep neural network to recognize a substantial vocabulary of words from audio recordings of human speech. A year later, and as detailed already in Chapter 1, the next landmark deep learning feat also came out of Toronto: AlexNet blowing the traditional

machine learning competition out of the water in the ImageNet Large-Scale Visual Recognition Competition (Figure 1.16). For a time, this staggering machine vision performance heralded a focus on applying deep learning to machine vision applications.

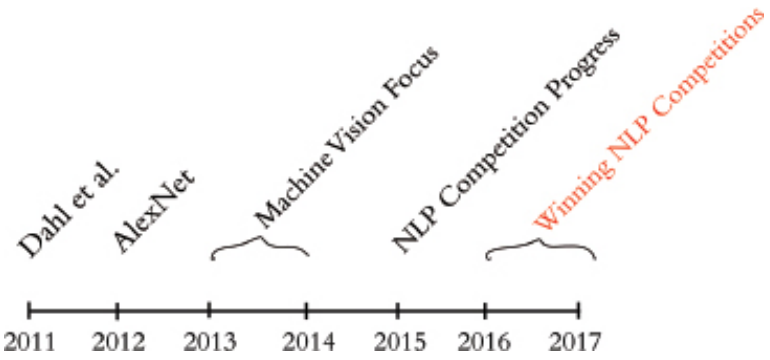

Figure 2.3 Milestones involving the application of deep learning to natural language processing. See text for details.

By 2015, the deep learning progress being made in machine vision began to spill over into NLP competitions such as those that assess the accuracy of machine translations from one language into another. These deep learning models approached the precision of traditional machine learning approaches, however they required less research and development time, while conveniently offering lower computational complexity. Indeed, this reduction in computational complexity provided Microsoft the opportunity to squeeze real-time machine translation software onto mobile phone processors—remarkable progress for a task that previously required an Internet connection and computationally-expensive calculations on a remote server. In 2016 and 2017, deep learning models entered into NLP competitions began to not only be more efficient than traditional machine learning models, they began outperforming them on accuracy as well. The remainder of this chapter will begin to illuminate how.

COMPUTATIONAL REPRESENTATIONS OF LANGUAGE

In order for deep learning models to process language, we have to supply that language to the model in a way that it can digest. For all computer systems, this means a quantitative representation of language, e.g., a two-dimensional matrix of numerical values. Two popular methods for converting text into numbers are one-hot encoding and word vectors. [4] We'll discuss both methods in turn in this section.

One-Hot Representations of Words

The traditional approach to encoding natural language numerically for processing it with a machine is *one-hot encoding* (Figure 2.4). In this approach, the words of natural language in a sentence (e.g., "the", "cat", "sat", "on", "the", and "mat") are represented by the columns of a matrix. Each row in the matrix, meanwhile, represents a unique

word. If there are a hundred unique words across the corpus [5] of documents you're feeding into your natural language algorithm, then your matrix of one-hot-encoded words will have one hundred rows. If there are a thousand unique words across your corpus, then there will be a thousand rows in your one-hot matrix, and so on.

The bat sat on the cat.

words						
the	1	0	0	0	1	0
bat	0	1	0	0	0	0
on	0	0	0	1	0	0
⋮						
n_{unique_words}						

Figure 2.4 One-hot encodings of words, such as this example, predominate the traditional machine learning approach to natural language processing.

Cells within one-hot matrices consist of binary values, i.e., they are a zero or a one. Each column contains at most a single one, but is otherwise made up of zeroes, meaning that one-hot matrices are *sparse*. [6] Values of one indicate the presence of a particular word (row) at a particular position (column) within the corpus. In Figure 2.4, our entire corpus has only six words in it, five of which are unique. Given this, a one-hot representation of the words in our corpus has six columns and five words. The first unique word—"the"—occurs in the first and fifth positions, as indicated by the cells filled with ones in the first row of the matrix. The second unique word in our wee corpus is "cat", which occurs only in the second position, so it is represented by a value of one in second row of the second column. One-hot word representations like this are fairly straightforward, and they are an acceptable format for feeding into a deep learning model (or, indeed, other machine learning models). As we shall see momentarily, however, the simplicity and sparsity of one-hot representations are limiting when incorporated into a natural language application.

Word Vectors

Vector representations of words are the information-dense alternative to one-hot encodings of words. While one-hot representations capture information about word

location only, *word vectors* capture information about word meaning as well as location. [7] This additional information renders word vectors favorable for a variety of reasons that we'll catalogue over the course of this chapter. The key advantage, however, is that—analogous to the visual features learned automatically by deep-learning machine-vision models in Chapter 1—word vectors enable deep-learning NLP models to automatically learn linguistic features.

When creating word vectors, the overarching concept is that we'd like to assign each word within a corpus to a particular, meaningful location within a multi-dimensional space called the *vector space*. Initially, each word is assigned to a random location within the vector space. By considering the words that tend to be used around a given word within the natural language of your corpus, however, the locations of the words within the vector space can gradually be shifted into locations that represent the meaning of the words. [8]

Figure 2.5 uses a toy-sized example to demonstrate in more detail the mechanics behind how word vectors are constructed. Commencing at the first word in our corpus and moving to the right one word at a time until we reach the final word in our corpus, we consider each word in our corpus to be the *target word*. At the particular moment captured in Figure 2.5, the target word that happens to be under consideration is `word`. The next target word would be `by`, followed by `the`, then `company`, and so on. For each target word in turn, we consider it relative to the words around it—its *context words*. In our toy example, we're using a context-word window size of three words. This means that while `word` is the target word, the three words to the left (`a`, `know` and `shall`) combined with the three words to the right (`by`, `company`, and `the`) together constitute a total of six context words. [9] When we move along to the subsequent target word (`by`), the windows of context words also shift one position to the right, dropping `shall` and `by` as context words while adding `word` and `it`.

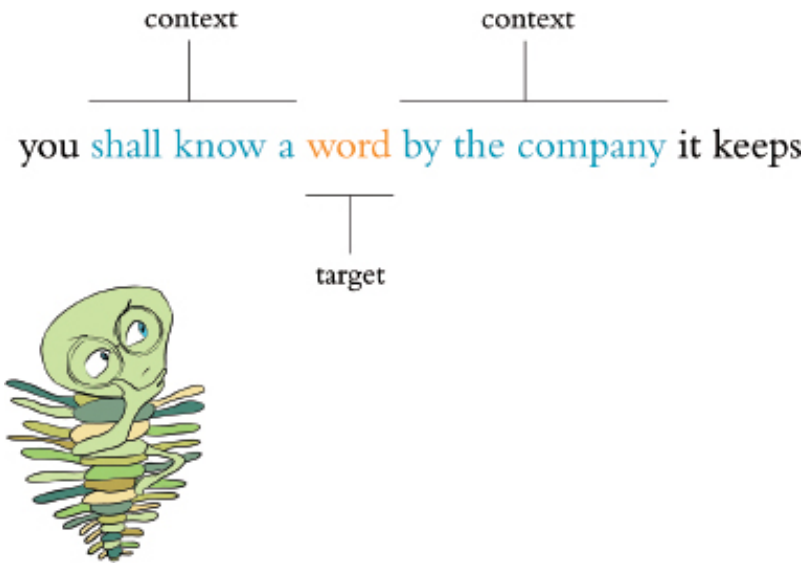

Figure 2.5 A toy example for demonstrating the high-level process behind techniques that convert natural language into word vectors like word2vec and GloVe. See text for details.

By a considerable margin, the two most popular techniques for converting natural language into word vectors are *word2vec*[10] and *GloVe*.[11] With either technique, our objective while considering any given target word is to accurately predict the target word given its context words[12]. Improving at these predictions, target word after target word over a large corpus, we gradually assign words that tend to appear in similar contexts to similar locations in vector space.

Figure 2.6 provides a cartoon of vector space. The space can have any number of dimensions so we can call it an *n*-dimensional vector space. In practice, depending on the richness of the corpus we have to work with and the complexity of our NLP application, we might create a word-vector space with dozens, hundreds or—in extreme cases—thousands of dimensions. As overviewed in the previous paragraph, any given word from our corpus (e.g., `king`) is assigned a location within the vector space. In, say a 100-dimensional space, the location of the word king is specified by a vector that we can call v_{king} that must consist of 100 numbers in order to specify the location of the word `king` across all of the available dimensions. Human brains aren't adept at spatial reasoning in more than three dimensions, so our cartoon in Figure 2.6 has only three dimensions. In this three-dimensional space, any given word from our corpus needs three numeric coordinates to define its location within the vector space: x, y and z. In this cartoon example then, the meaning of the word `king` is represented by a vector v_{king} that consists of three numbers. If v_{king} is located at the coordinates $x = 1.1$, $y = 2.4$, and $z = 3.0$ in the vector space, we can use the annotation `[-1.1, 2.4, 3.0]` to describe this location succinctly. This succinct annotation will come in handy later when we perform arithmetic operations on word vectors.

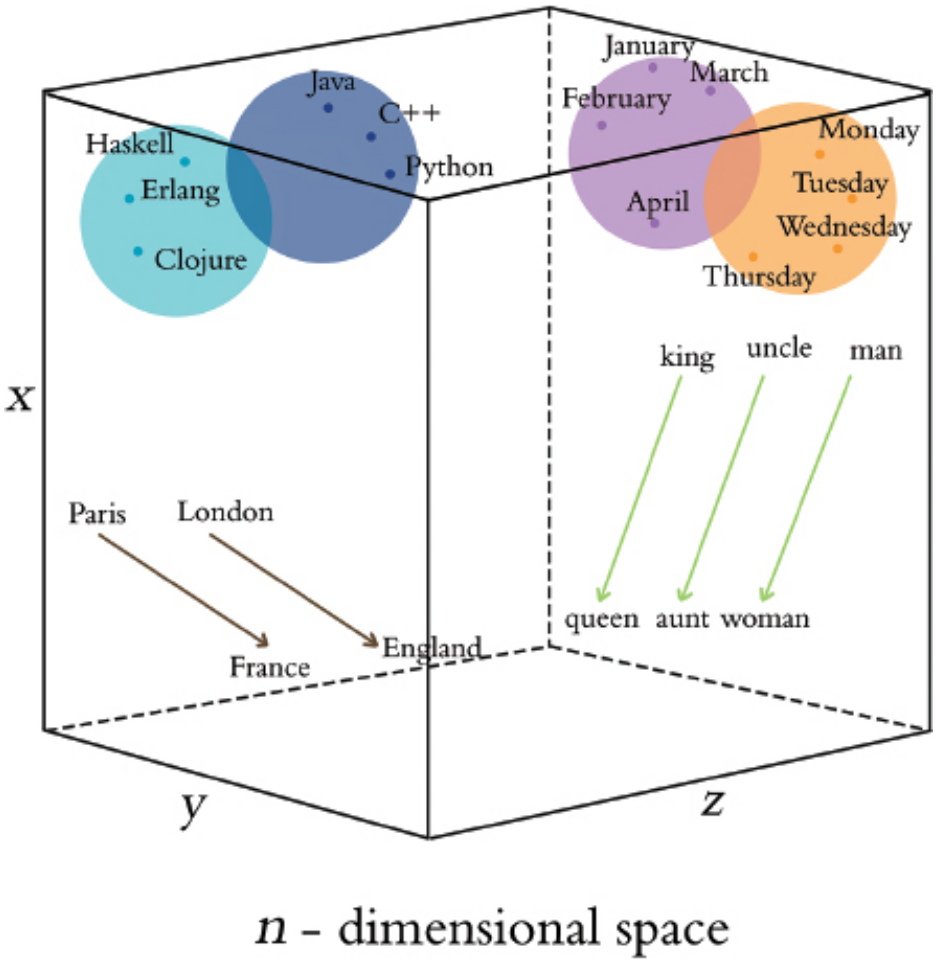

Figure 2.6 Diagram of word meaning as represented by a three-dimensional vector space. See text for details.

The closer two words within vector space,[13] the closer their meaning, as determined by the similarity of the context words appearing near them in natural language. Synonyms and common misspellings of a given word—because they share an identical meaning—would be expected to have near-identical context words and therefore near-identical locations in vector space. Words that are used in similar contexts, such as those that denote time for example, tend to occur near each other in vector space. In Figure 2.6, `Monday`, `Tuesday`, and `Wednesday` could be represented by the orange-colored dots located within the orange days-of-the-week cluster in the cube's top-right corner. Meanwhile, months of the year might occur in their own purple cluster, which is adjacent but distinct to the days of the week—they both relate to the date, but they're separate sub-clusters within a broader *dates* cluster. As a second example, we would expect to find programming languages clustering together in some location within the word vector space that is distant from the time-denoting words, say in the top-left corner. Again here, object-oriented programming languages like Java, C++, and Python would be expected to form one sub-cluster, while nearby we would expect to find functional programming languages like Haskell, Clojure and Erlang forming a separate sub-cluster. As we'll see in Chapter 11 when we build our own word vectors, less concretely-defined terms that nevertheless convey a specific meaning (e.g., the verbs `created`, `developed`, `built`) are also allocated positions within word-vector space

that enable them to be useful in NLP tasks.

Word Vector Arithmetic

Remarkably, because it turns out to be an efficient way for relevant word information to be stored in a vector space, particular movements across vector space come to represent relative particular meanings between words. This is a bewildering property.[14] Returning to our cube in Figure 2.6, the brown arrows represent the relationship between countries and their capital. That is, if we calculate the direction and distance between the coordinates of the words `Paris` and `France`, then trace this direction and distance from `London`, we should find ourselves in the neighborhood of the coordinate representing the word `England`. As a second example, we can calculate the direction and distance between the coordinates for `man` and `woman`. This movement through vector space represents gender and is symbolized by the green arrows in Figure 2.6. If we trace the green direction and distance from any given male-specific term (e.g., `king`, `uncle`), we should find our way to a coordinate near the term's female counterpart (`queen`, `aunt`).

A by-product of being able to trace vectors of meaning (e.g., gender, capital-country relationship) from one word in vector space to another is that we can perform *word vector arithmetic*. The canonical example of this is: If we begin at v_{king}, the vector representing `king` (continuing with our example from the previous section, this location is described by `[-1.1, 2.4, 3.0]`), subtract the vector representing `man` from it (let's say v_{man} = `[-1.1, 2.4, 3.0]`) and add the vector representing `woman` (let's say v_{woman} = `[-3.2, 2.5, 2.6]`), we should find a location near the vector representing `queen`. To make this arithmetic explicit by working through it dimension by dimension, we would estimate the location of v_{queen} by calculating:

$$\begin{aligned} x_{queen} &= x_{king} - x_{man} + x_{woman} = -0.9 + 1.1 - 3.2 = -3.0 \\ y_{queen} &= y_{king} - y_{man} + y_{woman} = 1.9 - 2.4 + 2.5 = 2.0 \\ z_{queen} &= z_{king} - z_{man} + z_{woman} = 2.2 - 3.0 + 2.6 = 1.8 \end{aligned} \tag{2.1}$$

All three dimensions together then, we expect v_{queen} to be near `[-3.0, 2.0, 1.8]`.

Figure 2.7 provides further, entertaining examples of arithmetic through a word vector space that was trained on a large natural language corpus crawled from the web. As we'll later observe in practice in Chapter 11, the preservation of these quantitative relationships of meaning between words across vector space is a robust starting point for deep learning models within NLP applications.

$$v_{king} - v_{man} + v_{woman} = v_{queen}$$
$$v_{bezos} - v_{amazon} + v_{tesla} = v_{musk}$$
$$v_{windows} - v_{microsoft} + v_{google} = v_{android}$$

Figure 2.7 Examples of word vector arithmetic.

word2viz

To develop your intuitive appreciation of word vectors, navigate to `lamyiowce.github.io/word2viz`. The default screen for the *word2viz* tool for exploring word vectors interactively is shown in Figure 2.8. Leaving the top-right dropdown box set to "Gender analogies", try adding in *pairs* of new words under the "Modify words" heading. If you add pairs of corresponding gender-specific words like `princess` and `prince`, `duchess` and `duke`, and `businesswoman` and `businessman`, you should find that they fall in instructive locations.

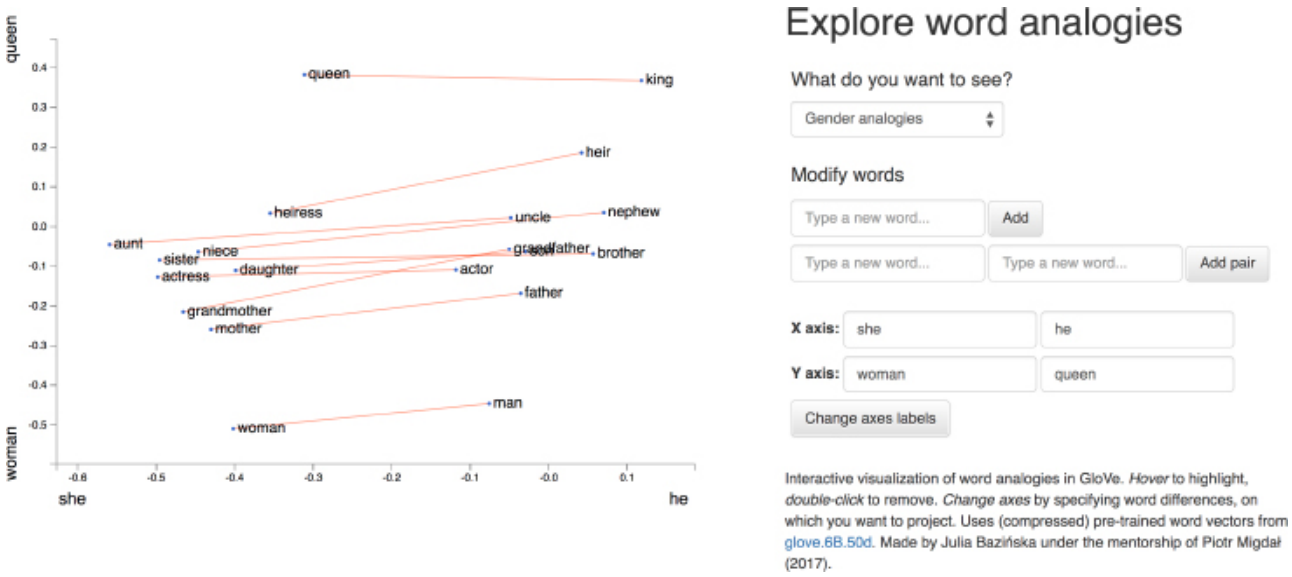

Figure 2.8 The default screen for word2viz, a tool for exploring word vectors interactively.

The developer of the word2viz tool, Julia Bazinska, compressed a fifty-dimensional word-vector space down to two dimensions in order to visualize the vectors on an *xy*-coordinate system.[15] For the default configuration, Bazinska scaled the *x*-axis from the words she to he as a reference point for gender, while the *y*-axis was set to vary from a common base toward a royal peak by orienting it to the words woman and queen. The displayed words, placed into vector space via training on a natural language data set consisting of six billion instances of 400,000 unique words[16], fall relative to the two axes based on their meaning. The more regal (`queen`-like) the words, the higher on the plot they should be shown, and the female (`she`-like) terms fall to the left of their male (`he`-like) counterparts.

When you've indulged yourself sufficiently with word2viz's "Gender analogies" view, you can experiment with other perspectives of the word vector space. Selecting "Adjectives-Analogies" from the "What do you want to see?" drop-down box, you could for example add the words `small` and `smallest`. Subsequently, you could change the *x*-axis labels to `nice` and `nicer`, and then again to `small` and `big`. Switching to the "Numbers say-write analogies" view via the drop-down box, you could play around with changing the *x*-axis to `3` and `7`.

You may build your own word2viz plot from scratch by moving to the "Empty" view. The (word vector) world is your oyster, but you could perhaps examine the country-capital relationships mentioned earlier when familiarizing ourselves with Figure 2.6. To do this, set the *x*-axis to range from `west` to `east` and the *y*-axis to `city` and `country`. Word pairs that fall neatly into this plot include `london—england`, `paris—france`, `berlin—germany` and `beijing—china`.

This paragraph is the Trilobite Attention SIDEBAR on Bias. While on the one hand word2viz is an enjoyable way to develop a general understanding of word vectors, on the other hand it can also be a serious tool for gaining insight into specific strengths or weaknesses of a given word-vector space. As an example, use the "What do you want to see?" drop-down box to load the "Verb tenses" view and then add the words `lead` and `led`. Doing this, it becomes apparent that the coordinates words were assigned to in this vector space mirror existing gender stereotypes that were present in the natural language data the vector space was trained on. Switching to the "Jobs" view, this gender bias becomes even more stark. It is probably safe to say that any large natural language data set is going to have some biases in it, whether intentional or not. The development of techniques for reducing biases in word vectors is an active area of research.[17] Mindful that these biases may be present in your data, however, the safest bet is to test your downstream NLP application in a range of situations that reflect a diverse user-base, checking that the results are appropriate. END SIDEBAR.

Localist versus Distributed Representations

With an intuitive understanding of word vectors under our figurative belts, we can contrast them with one-hot representations (Figure 2.4), which have been an established presence in the NLP world for longer. A summary distinction is that we can say word vectors store the meaning of words in a *distributed* representation across *n*-dimensional space. That is, with word vectors, word meaning is distributed gradually —"smeared"—as we move from location to location through vector space. One-hot representations, meanwhile, are *localist*—they store information on a given word discretely, within a single row of a typically-extremely-sparse matrix.

To more thoroughly characterize the distinction between the localist, one-hot approach and the distributed, vector-based approach to word representation, Table 2.1 compares them across a range of attributes. Firstly, one-hot representations lack nuance; they are simple binary flags. Vector-based representations, on the other hand, are extremely nuanced: Within them, information about words is smeared throughout a continuous, quantitative space. In this high-dimensional space, there are essentially infinite possibilities for capturing the relationships between words.

Table 2.1 Table contrasting attributes of localist, one-hot representations of words with distributed, vector-based representations

One-Hot	Vector-Based
not subtle	very nuanced
manual taxonomies	automatic
handle new words poorly	seamlessly incorporate new words
subjective	driven by natural language data
word similarity not represented	word similarity = proximity in space

Secondly, the use of one-hot representations in practice often requires labor-intensive, manually-curated taxonomies. These taxonomies include dictionaries and other specialised reference language databases.[18] Such external references are unnecessary for vector-based representations, which are fully-automatic with natural language data alone.

Third, one-hot representations don't handle new words well. A newly introduced word requires a new row in the matrix and then re-analysis relative to the existing rows of the corpus, followed by code changes—perhaps via reference to external information sources. With vector-based representations, new words can be incorporated by training the vector space on natural language that includes examples of the new words in their natural context. A new word gets its own new *n*-dimensional vector. Initially, there may be few training data points involving the new word so its vector might not be very accurately positioned within *n*-dimensional space, but the positioning of all existing words remain intact and the model will not fail to function. Over time, as the instances of the new word in natural language increases, the accuracy of its vector-space coordinates will improve.[19]

Fourth, and following on from the previous two points, the use of one-hot representations often involves subjective interpretations of the meaning of language. This is because they often require coded rules or reference databases that are designed

by (relatively small groups of) developers. The meaning of language in vector-based representations, meanwhile, is data-driven.[20]

Fifth, one-hot representations natively ignore word similarity: Similar words, like `couch` and `sofa` are represented no differently than `couch` and `cat`. In contrast, vector-based representations innately handle word similarity: As mentioned earlier with respect to Figure 2.6, the more similar two words, the closer they are in vector space.

ELEMENTS OF NATURAL HUMAN LANGUAGE

Thus far, we have considered only one element of natural human language: the *word*. Words, however, are made up of constituent language elements. In turn, words themselves are the constituents of more abstract, more complex language elements. We'll begin with the language elements that make up words and build up from there, following the schematic in Figure 2.9. With each element, we'll discuss how it is typically encoded from the traditional machine learning perspective as well as from the deep learning perspective. As we move through these elements, notice how the distributed deep learning representations are fluid and flexible vectors while the traditional ML representations are local and rigid (Table 2.2).

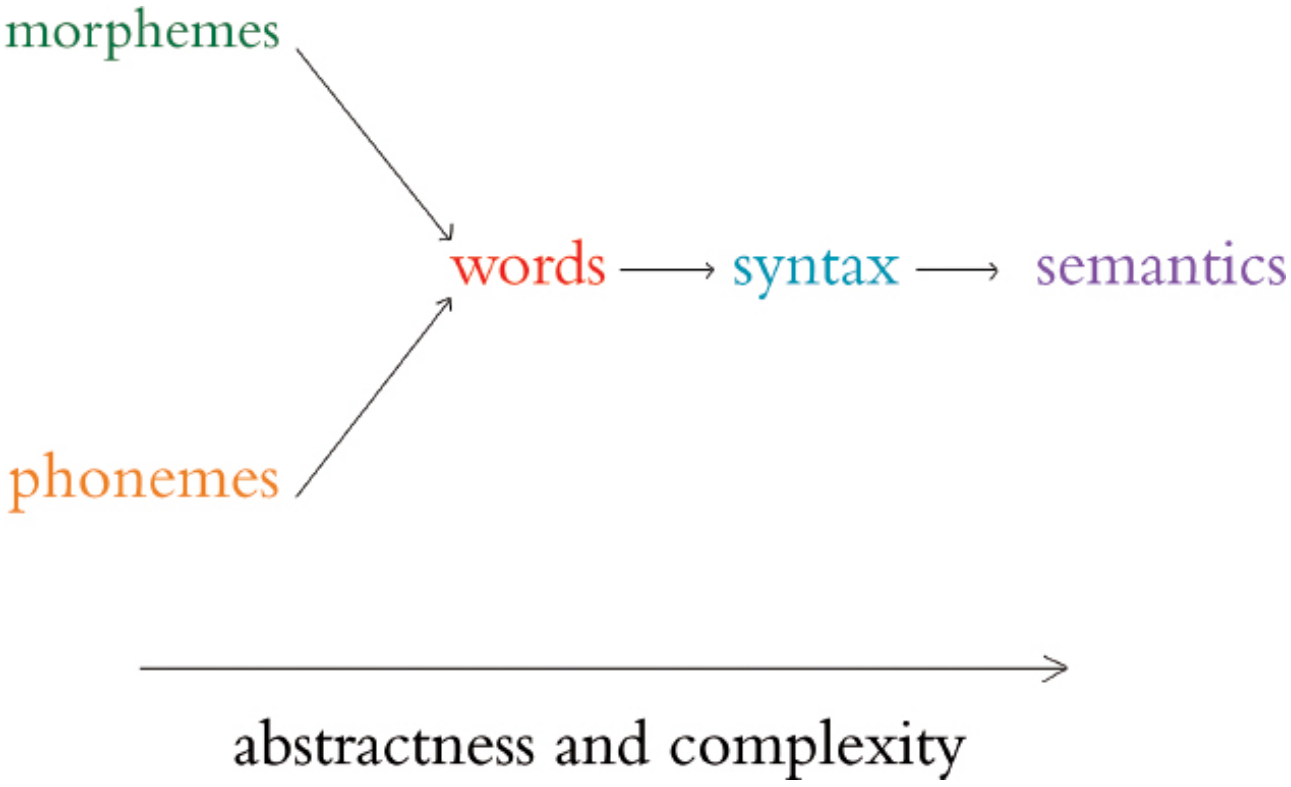

Figure 2.9 Relationships between the elements of natural human language. The left-most elements are building blocks for further-right elements. As we move to the right, the more abstract the elements become and therefore the more complex they are to model with an NLP application.

Table 2.2 Table of traditional machine learning and deep learning representations, by natural language element.

Representation	Traditional ML	Deep Learning	Audio-Only
phonology	all phonemes	vectors	true
morphology	all morphemes	vectors	false
words	one-hot encoding	vectors	false
syntax	phrase rules	vectors	false
sematics	lambda calculus	vectors	false

Phonology is concerned with the way that language sounds when it is *spoken*. Every language has a specific set of *phonemes* (sounds) that make up its words. The traditional ML approach is to encode segments of auditory input as specific phonemes from the language's range of available phonemes. With deep learning, we train a model to predict phonemes from features automatically learned from auditory input and then represent those phonemes in a vector space. In this book, we'll be working with natural language in text format only, but the techniques we cover can be applied directly to speech data if you're keen to do so in your own time.

Morphology is concerned with the forms of words. Like phonemes, every language has a specific set of *morphemes*, which are the smallest units of language that contain some meaning. For example, the three morphemes out, go, and ing combine to form the word outgoing. The traditional ML approach is to identify morphemes in text from a list of all the morphemes in a given language. With deep learning, we train a model to predict the occurrence of particular morphemes. Hierarchically-deeper layers of artificial neurons can then combine multiple vectors (e.g., the three representing out, go, and ing) into a single vector representing a word.

Phonemes (when considering audio) and morphemes (when considering text) combine to form *words*. Whenever we work with natural language data in this book, we'll work at the word level. We do this for four reasons. First, it's straightforward to define what a word is and everyone is familiar with what they are. Second, it's easy to break up natural language into words via a process called *tokenization*[21] that we'll work through in Chapter 11. Third, words are the most-studied level of natural language, particularly with respect to deep learning, so we can readily apply cutting-edge techniques to them. Fourth, and perhaps most critically, for the NLP models we'll be building, word vectors simply work well: they prove to be functional, efficient and accurate. In the previous section, we already detailed the shortcomings of localist, one-hot representations that predominate traditional ML relative to the word vectors used in deep learning models.

Words are combined to generate *syntax*. Syntax and morphology (already introduced above) together constitute the entirety of a language's grammar. Syntax is the arrangement of words into phrases and phrases into sentences in order to convey meaning in a way that is consistent across the users of a given language. In the traditional ML approach, phrases are bucketed into discrete, formal linguistic categories.[22] With deep learning (surprise, surprise!), we employ vectors. Every word and every phrase in a section of text can be represented by a vector in *n*-dimensional space, with layers of artificial neurons combining words into phrases.

Semantics is the most abstract of the elements of natural language in Figure 2.9 and Table 2.2; it is concerned with the *meaning* of sentences. This meaning is inferred from all the underlying language elements like words and phrases, as well as the overarching context that a piece of text appears in. Inferring meaning is complex because, for example, whether a passage is supposed to be taken literally or as a humorous and sarcastic remark can depend on subtle contextual differences and shifting cultural norms. Traditional ML, because it doesn't represent the fuzziness of language (e.g., the similarity of related words or phrases), is limited in capturing semantic meaning. With deep learning, vectors come to the rescue once again. Vectors can represent not only every word and every phrase in a passage of text, but every logical expression as well. As with the language elements already covered, layers of artificial neurons can recombine vectors of constituent elements—in this case to calculate semantic vectors via the non-linear combination of phrase vectors.

GOOGLE DUPLEX

One of the more attention-grabbing examples of deep-learning-based NLP in recent memory is that of Google Duplex, which was unveiled at the their *I/O* developers conference in May 2018. The search giant's CEO, Sundar Pichai, held spectators in rapture as he demonstrated Google Assistant making a phone call to a Chinese-food restaurant to book a reservation. The audible gasps from the audience were in response to the natural flow of Duplex's conversation. It had mastered the cadence of a human conversation, replete with the *uh*'s and *hhhm*'s that we sprinkle into conversations while we're thinking. Furthermore, the phone call was of average audio quality and the human on the line had a strong accent—Duplex never faltered, and managed to make the booking.

Bearing in mind that this is a demonstration—and not even a live one—what nevertheless impressed us was the breadth of deep-learning applications that had to come together to facilitate this technology. Consider the flow of information back-and-forth between the two agents on the call, Duplex and the restaurateur: Duplex needs a

sophisticated speech recognition algorithm that can process audio in realtime and handle an extensive range of accents and call qualities on the other end of the line, and also overcome the background noise.[23]

Once the human's speech has been faithfully transcribed, an NLP model needs to process the sentence and decide what it means. The intention is that the person on the line doesn't know they're speaking to a computer and so doesn't need to modulate their speech accordingly, but in turn, this means that humans respond with complex, multi-part sentences that can be tricky for a computer to tease apart:

"We don't have anything tomorrow, but we have the next day and Thursday, anytime before 8. Wait no… Thursday at 7 is out. But we do can after 8?"

This sentence is poorly structured—you'd never write an email like this—but in natural conversation, these sorts of on-the-fly corrections and replacements happen regularly, and Duplex needs to be able to follow along.

With the audio transcribed and meaning of the sentence processed, Duplex's NLP model conjures up a response. This response must either ask for more information if the human was unclear or if the answers were unsatisfactory, otherwise it should confirm the booking. The NLP model will generate a response in text form, so a test-to-speech engine is required to synthesize the sound.

This paragraph is a Trilobite Reading SIDEBAR Duplex uses a combination of *de novo* waveform synthesis using Tacotron[24] and WaveNet[25], as well as a more classical "concatenative" text-to-speech engine[26]. This is where the system crosses the uncanny valley[27]—the voice heard by restauranteur is not a human voice at all. WaveNet is able to generate completely synthetic waveforms, one sample at a time, using a deep neural network trained on real waveforms from human speakers. Beneath this, Tacotron maps sequences of *words* to corresponding sequences of *audio features*, which capture subtleties of human speech such as pitch, speed, intonation and even pronunciation. These features are then fed into WaveNet which synthesizes the actual waveform that the restauranteur hears. This whole system is able to produce natural sounding voice with the correct cadence, emotion and emphasis. During moments of more-or-less rote moments in the conversation, the simple concatenative TTS engine (comprised of recordings of it's *own* "voice") which is less computationally demanding to execute, is used. The entire model dynamically switches between the various models as needed. END SIDEBAR.

To misquote Jerry Maguire: you had all of this at "hello." The speech recognition

system, NLP models, and TTS engine all work in concert from the instant the call is answered. Things only stand to get more complex for the Duplex from then on. Governing all of this interaction is a deep neural network that is specialized in handling information that occur in a sequence[28]. This governor tracks the conversation and feeds the various inputs and outputs into the appropriate models. It should be clear from this overview that Google Duplex is a highly sophisticated system of deep learning models that work in harmony to produce a seamless interaction on the phone. For now, Duplex is limited to a few very specific domains: scheduling appointments. The system cannot carry out general conversations. So while this represents a significant step forward for artificial intelligence, there is still much work to be done.

SUMMARY

In this chapter, we learned about applications of deep learning to the processing of natural language. In so doing, we described further the capacity for deep learning models to automatically extract the most pertinent features from data, removing the need for labor-intensive one-hot representations of language. Instead, NLP applications involving deep learning make use of vector-space embeddings, which capture the meaning of words in a nuanced manner that improves both model performance and accuracy.

Later, in Chapter 11, we'll ourselves construct an NLP application by making use of artificial neural networks that handle the input of natural language data all the way through to the output of an inference about those data. In such an "end-to-end" deep learning model, the initial layers create word vectors that flow seamlessly into deeper, specialized layers of artificial neurons, including layers that incorporate "memory". These model architectures will highlight both the strength and the ease-of-use of deep learning with word vectors.

1 . Wittgenstein, L. (1953). *Philosophical Investigations*. (Anscombe, G., Trans.). Oxford, UK: Basil Blackwell.

2 . Trilobite Attention SIDEBAR *Named entities* include places, well-known individuals, company names and products.

3 . Dahl, G., et al. (2011). Large vocabulary continuous speech recognition with context-dependent DBN-HMMs. *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*.

4 . If this were a book dedicated to NLP, then we would have been wise to also describe natural language methods based on word frequency, e.g., TF-IDF (term frequency-inverse document frequency) and PMI (pointwise mutual information).

5 . Trilobite Reading SIDEBAR A *corpus* is the collection of all of the documents you use as your input data for a given natural language application. In Chapter 11, we'll make use of a corpus that consists of eighteen classic books. Later in that chapter, we'll separately make use of a corpus of 25,000 film reviews. An example of a much larger corpus would be all of the English-language articles on Wikipedia. The largest corpuses are crawls of all the publicly-available data on the Internet, e.g., as at `commoncrawl.org`.

6 . Non-zero values are rare (i.e., they are *sparse*) within a sparse matrix. In contrast, *dense* matrices are rich in information: they typically contain few—perhaps even no—zero values.

7 . Strictly speaking, a one-hot representation is technically a "word vector" itself as each column in a one-hot word matrix consists of a vector representing a word at a given location. In the deep learning community, however, use of the term "word vector" is commonly reserved for the dense representations covered in this section—i.e., those derived by word2vec, GloVe, and related techniques.

8 . As mentioned at the beginning of this chapter, this understanding of the meaning of a word from the words around it was proposed by Ludwig Wittgenstein. Later, in 1957, the idea was captured succinctly by the British linguist J.R. Firth with his phrase "You shall know a word by the company it keeps".

9 . It is mathematically simpler and more efficient to not concern ourselves with the specific ordering of context words, particularly as word order tends to confer negligible extra information to the inference of word vectors. Ergo, we provide the context words in parentheses alphabetically, an effectively random order.

10. Mikolov, T., et al. (2013). Efficient estimation of word representations in vector space. *arXiv:1301.3781*.

11. Pennington, J., et al. (2014). GloVe: Global vectors for word representations. *Proceedings of the Conference on Empirical Methods in natural language processing*.

12. Or, alternatively, we could predict context words given a target word. More on that in Chapter 11.

13. Measured by Euclidean distance, which is the plain old straight-line distance between two points.

14. One of your esteemed authors, Jon, prefers terms like "mind-bending" and "trippy" to describe this property of word vectors, but he consulted a thesaurus to narrow in on a more professional-sounding adjective.

15. We'll detail how to perform vector space dimensionality reduction in Chapter 11.

16. Technically, 400,000 *tokens*—a distinction that we'll examine later.

17. E.g.: Bolukbasi, T., et al. (2016). Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings. *arXiv:1607.06520*. Caliskan, A., et al. (2017). Semantics derived automatically from language corpora contain human-like biases. *Science 356*: 183-6. Zhang, B., et al. (2018). Mitigating Unwanted Biases with Adversarial Learning. *arXiv:1801.07593*

18. E.g., WordNet (wordnet.princeton.edu), which describes synonyms as well as *hypernyms* ("is-a" relationships, so `furniture`, for example, is a hypernym of `chair`)

19. An associated problem not addressed here is when an in-production NLP algorithm encounters a word that was not included within its corpus of training data. This *out of vocabulary* problem impacts both one-hot representations and word vectors. There are approaches—e.g., Facebook's *fastText* library—that try to get around the issue by considering sub-word information, but these approaches are beyond the scope of this book.

20. Noting that they may nevertheless include biases found in natural language data. See the "Trilobite-Attention SIDEBAR" on Bias.

21. Essentially, tokenization is the use of characters like commas, periods and whitespace to assume where one word ends and the next begins.

22. These categories have names like "noun-phrase" and "verb-phrase".

23. This is known as the "cocktail-party problem"—or less jovially, "multi-talker speech separation". It's a problem that humans solve innately, isolating single voices from a cacophony quite well without explicit instruction on how to do so. Machines typically struggle with this, though a variety of groups have proposed solutions, e.g., see

Simpson, A., et al. (2015). Deep Karaoke: Extracting Vocals from Musical Mixtures Using a Convolutional Deep Neural Network. *arXiv:1504.04658*; Yu, D., et al. (2016). Permutation Invariant Training of Deep Models for Speaker-Independent Multi-talker Speech Separation. *arXiv:1607.00325*

24. ai.googleblog.com/2017/12/tacotron-2-generating-human-like-speech.html

25. deepmind.com/blog/wavenet-generative-model-raw-audio

26. Concatenative TTS engines use vast databases of pre-recorded words and snippets, which can be strung together to form sentences. This approach is common and fairly easy, but yields stilted, unnatural speech and cannot adapt the speed and intonation—you can't modulate a word to make it sound like a question is being asked, for example.

27. The uncanny valley is a dangerous space wherein humans find human-like simulations weird and creepy because they're too similar to real humans, but it's also clear they're *not* real humans. Product designers endeavor to avoid the uncanny valley. They've learned that users respond well to simulations that are either very robotic or not robotic at all.

28. Called a *recurrent neural network*. These feature in Chapter 11

3 Machine Art

In this chapter, we'll introduce some of the concepts that enable deep learning models to seemingly *create* art, an idea that may be paradoxical to some. The University of California, Berkeley philosopher Alva Noë, for one, opined "Art can help us frame a better picture of our human nature." [1] If this is true, how can machines create art? Or put differently, are the creations that emerge from these machines, in fact, art? Another interpretation—and one we like best—is that these creations are indeed art and that programmers are artists wielding deep-learning models as brushes.

By the end of this chapter, you'll have learned the essential ideas behind generative adversarial networks (GANs) and you will have seen how they can be used to generate completely novel works. We'll have hopefully drawn a convincing link between the word vector spaces of the previous chapter and the latent spaces associated with GANs. We'll also cover some intersections between art and deep learning that don't make use of GANs, wherein the deep learning models are more obviously tools that can be applied to some end. Enough with the philosophy, let's get our overalls covered in paint! But first… a drink.

A BOOZY ALL-NIGHTER

Sitting below Google's offices in Montreal sits a bar called *Les 3 Brasseurs*, a moniker that translates from French to *The 3 Brewers*. It was at this watering hole in 2014, while a Ph.D. student in Yoshua Bengio's renowned lab (Figure 1.11), that Ian Goodfellow conceived of an algorithm for fabricating realistic-looking images, [2] a technique that Yann LeCun (Figure 1.10) has hailed as the "most important" recent breakthrough in Deep Learning. [3]

Goodfellow's friends described to him a *generative model* they were working on, i.e., a computational model that aims to produce something novel, be it a quote in the style of Shakespeare, a musical melody, or a work of abstract art. In their particular case, the friends were attempting to design a model that could generate photorealistic images such as portraits of human faces. For this to work well via the Traditional Machine Learning approach (Figure 1.13), the engineers designing the model would need to not

only catalog and approximate the critical individual features of faces like eyes, noses and mouths, but also accurately estimate how these features should be arranged relative to each other. Thus far, their results had been underwhelming. The generated faces tended to be excessively blurry or they tended to be missing essential elements like the nose or the ears.

Perhaps with his creativity heightened by a pint of beer or two, [4] Goodfellow proposed a revolutionary idea: a deep learning model in which two artificial neural networks act against each other competitively as adversaries. As illustrated in Figure 3.1, one of these deep ANNs would be programmed to produce forgeries while the other would be programmed to act as a detective and distinguish the fakes from real images (which would be provided separately). These adversarial deep learning networks would play off one another: As the *generator* became better at producing fakes, the *discriminator* would need to become better at identifying them, and so the generator would need to produce even more compelling counterfeits, and so on. This virtuous cycle would eventually lead to convincing novel images in the style of the real training images, be they of faces or otherwise. Best of all, Goodfellow's approach would circumnavigate the need to program features into the generative model manually. As we've already expounded with respect to machine vision (Chapter 1) and natural language processing (Chapter 2), deep learning would sort the model's features out automatically.

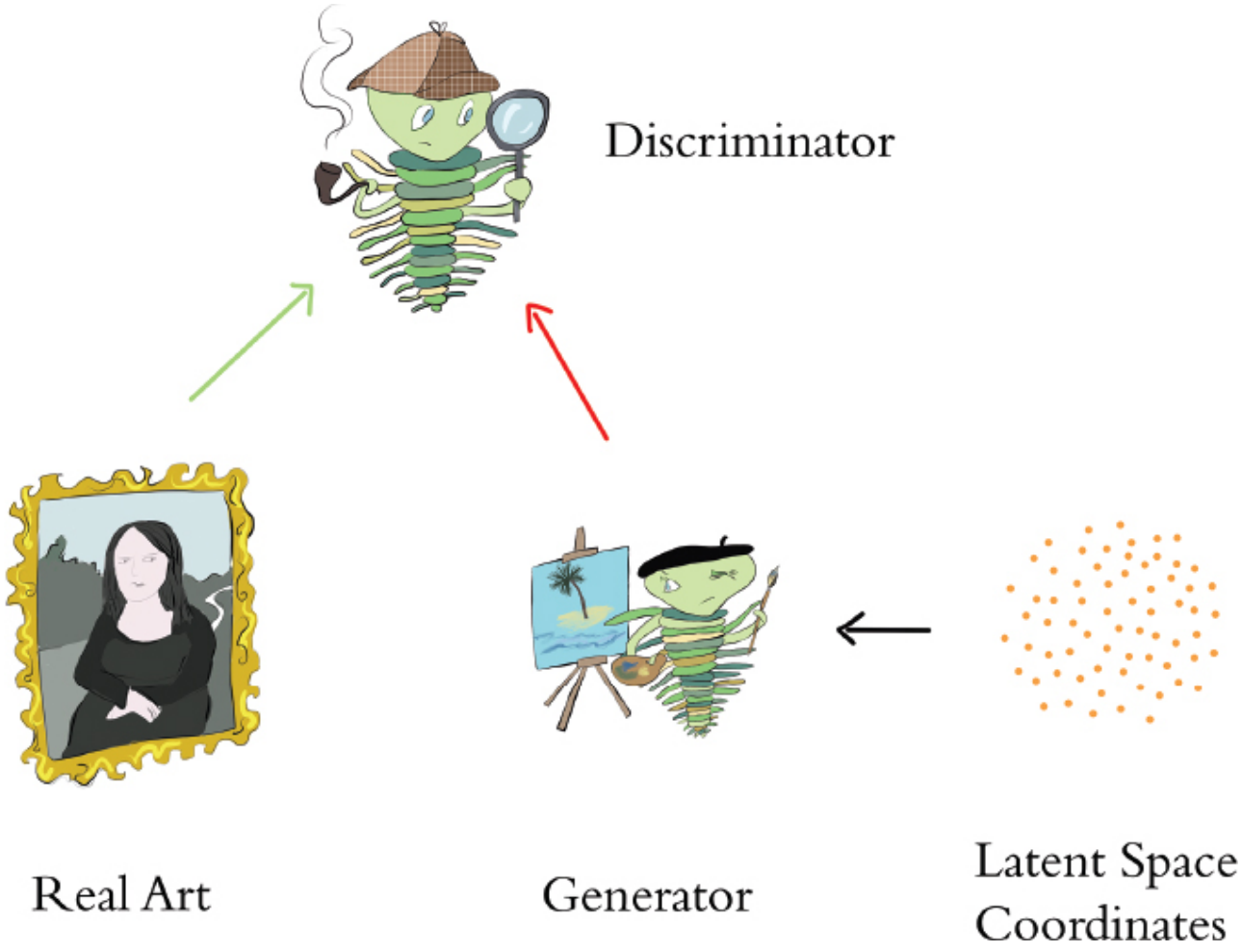

Figure 3.1 High-level schematic of a generative adversarial network (GAN). Real images, as well as forgeries produced by the generator are provided to the discriminator, which is tasked with identifying which are the genuine article. The purple cloud represents latent space (Figure 3.4) "guidance" that is provided to the forger. This guidance can either be random (as is generally the case during network training; see Chapter 12) or selective (during post-training exploration, as in Figure 3.3).

Goodfellow's friends were doubtful his imaginative approach would work. So, when he arrived home and found his girlfriend asleep, he worked late to architect his dual-ANN design. It worked the first time and the astounding deep learning family of *generative adversarial networks* was born!

That same year, Goodfellow and his colleagues revealed GANs to the world at the prestigious *Neural Information Processing Systems (NIPS)* conference. [5] Some of their results are shown in Figure 3.2. Their GAN produced these novel images by being trained on: (a) handwritten digits; [6] (b) photos of human faces; [7] and (c) & (d) photos from across ten classes (e.g., planes, cars, dogs). [8] The results in (c) are markedly less crisp than in (d) because the GAN that produced the latter featured neuron layers specialized for machine vision called *convolutional* layers [9] while the GAN that produced the former used a more general layer type only.[10]

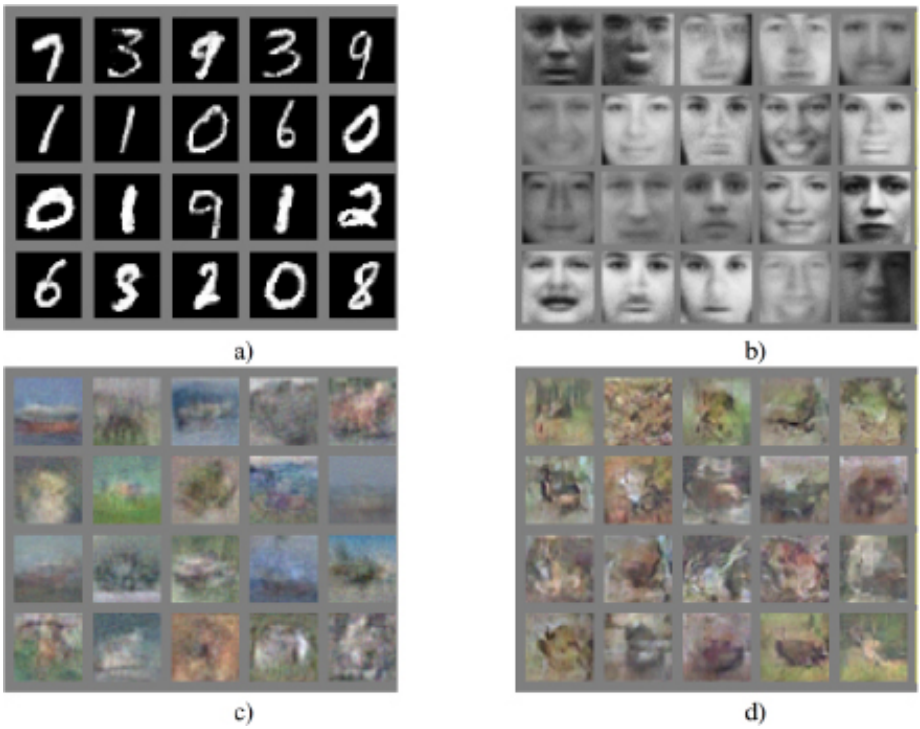

Figure 3.2 Results presented in Goodfellow and colleagues' (2014) GAN paper. See text for details.

ARITHMETIC ON FAKE HUMAN FACES

Following on from Goodfellow's lead, a research team led by the American machine-learning engineer Alec Radford determined architectural constraints for GANs that guide considerably more realistic image creation. Some examples of portraits of fake humans that were produced by their *deep convolutional* GANs[11] are provided in Figure 3.3. In their paper, Radford and his teammates cleverly demonstrated interpolation through, and arithmetic with, the *latent space* associated with GANs. Let's start off by explaining what latent space is before moving on to latent-space interpolation and arithmetic.

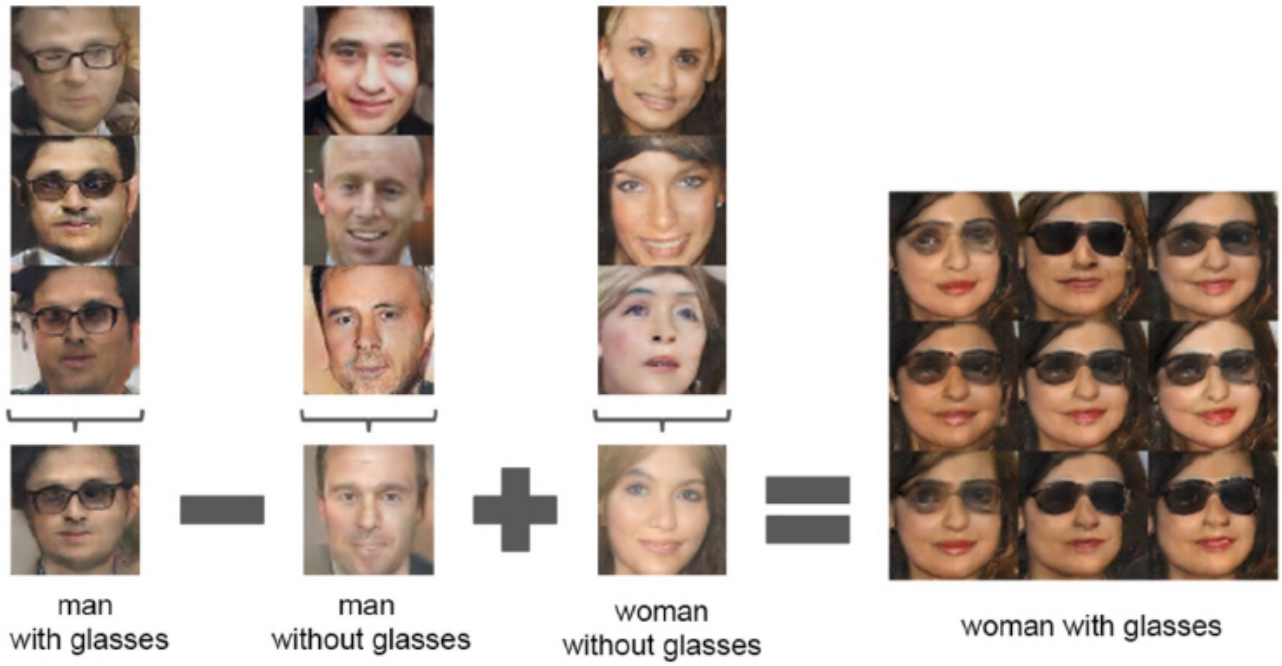

Figure 3.3 An example of latent-space arithmetic from Radford et al. (2016).

The latent-space cartoon in Figure 3.4 may be reminiscent of the word vector-space cartoon in Figure 2.6. As it happens, there are three major similarities between latent spaces and vector spaces. First, while the cartoon is only three-dimensional for simplicity and comprehensibility, latent spaces are `n`-dimensional spaces, usually in the order of hundreds of dimensions. The latent space of the GAN we'll later architect ourselves in Chapter 12, for example, will have `n` = 100 dimensions. Second, the closer two points are in the latent space, the more similar the images that those points represent are. And third, movement through the latent space in any particular direction can correspond to a gradual change in a concept being represented, such as age or gender for the case of photorealistic faces.

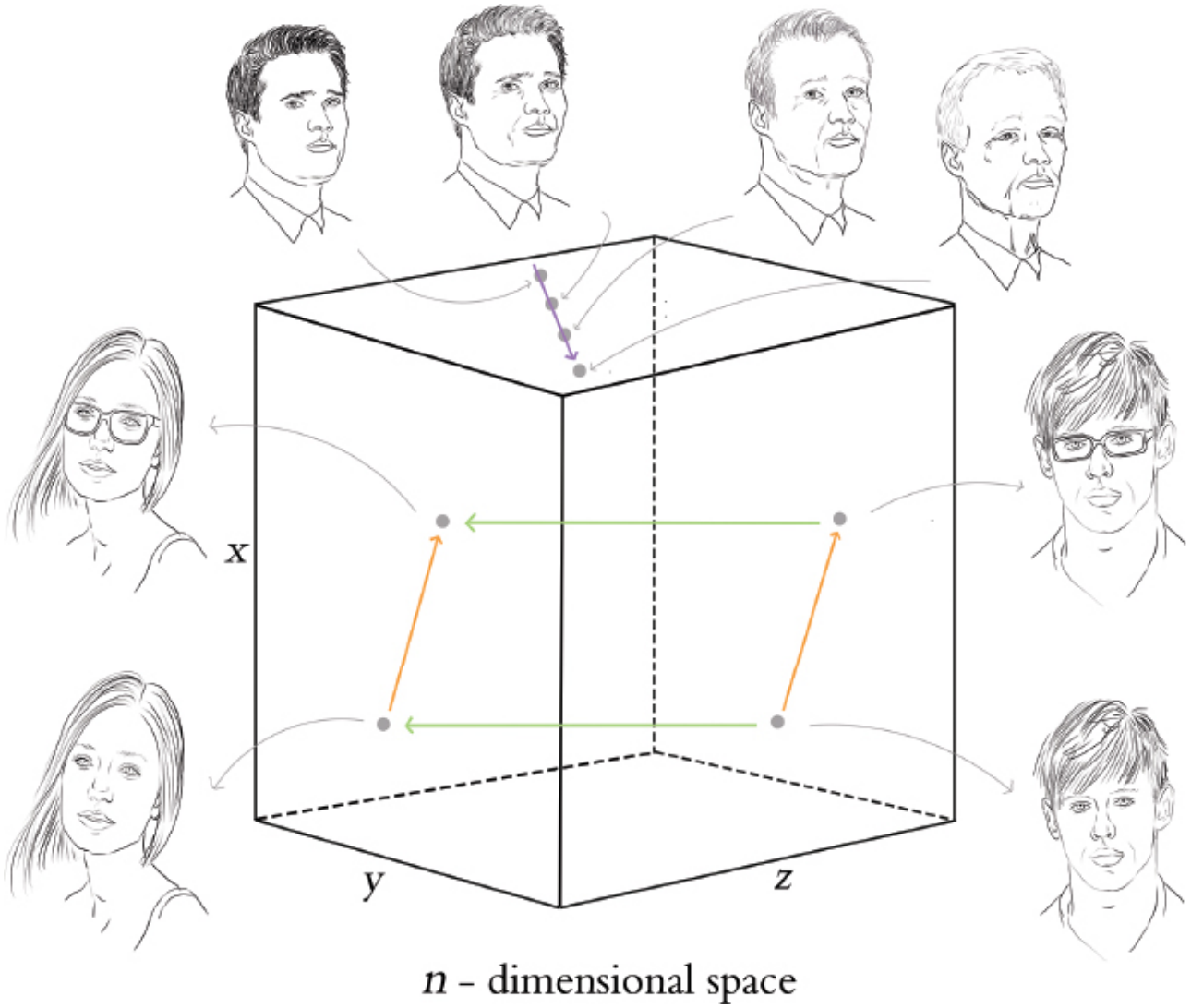

Figure 3.4 A cartoon of the latent space associated with generative adversarial networks (GANs). Moving along the purple arrow, the latent space corresponds to images of a similar-looking individual aging. The green arrow represents gender while the orange one represents the inclusion of glasses on the face.

By picking two points far away from each other along some *n*-dimensional axis representing age, interpolating between them, and sampling points from the interpolated line, we could find what appears to be the same (fabricated) man gradually appearing to be older and older.[12] In our latent-space cartoon (Figure 3.4), we represented such an "age" axis in purple. To observe interpolation through an authentic GAN latent space, we recommend scanning through Radford and colleagues' paper for, as an example, smooth rotations of the "photo angle" of synthetic bedrooms. At the time of writing the book manuscript you're presently reading, the state of the art in GANs can be viewed at `youtu.be/G06dEcZ-QTg`. This video, produced by researchers at the graphics-card manufacturer Nvidia[13], provides a breathtaking interpolation through high-quality portrait "photographs" of ersatz celebrities.

Moving a step further with what we've learned, we could now perform arithmetic with images sampled from a GAN's latent space. When sampling a point within the latent space, that point can be represented by the co-ordinates of its location—the resulting

vector is analogous to the word vectors described in Chapter 2. Just as with word vectors, we can perform arithmetic with these vectors and move through the latent space in a semantic way. Figure 3.3 showcases an instance of latent-space arithmetic from Radford and his co-workers. Starting with a point in their GAN's latent space that represents a man with glasses, subtracting a point that represents a man *without* glasses, and adding a point representing a *woman* without glasses, the resulting point exists in the latent space near to images that represent women *with* glasses. Our cartoon in Figure 3.4 illustrates how the relationships between meaning in latent space are stored (again, akin to the way they are in word-vector space), thereby making arithmetic on points in latent space possible.

STYLE TRANSFER: CONVERTING PHOTOS INTO MONET (AND VICE VERSA)

One of the more magical applications of GANs is *style transfer*. Zhu, Park and their coworkers from the University of California, Berkeley's A.I. Research lab introduced a new flavor of GAN[14] that enables stunning examples of this, as shown in Figure 3.5. Alexei Efros, one of the paper's co-authors, took photos while on holiday in France and they employed their CycleGAN to transfer these photos into the style of the Impressionist Claude Monet, the 19th-century Dutch artist Van Gogh and the Japanese "Ukiyo-e" genre, amongst others. If you navigate to `junyanz.github.io/CycleGAN`, you'll be delighted to discover instances of the inverse (Monet paintings converted into photorealistic images), as well as:

œ summer scenes converted into wintry ones, and vice versa

œ baskets of apples converted into baskets of oranges, and vice versa

œ flat, low-quality photos converted into what appear to be ones captured by high-end single-lens reflex cameras

œ a video of a horse running in a field converted into a zebra

œ a video of a drive taken during the day converted into a nighttime one

Figure 3.5 Photos converted into the styles of well-known painters by CycleGANs.

MAKE YOUR OWN SKETCHES PHOTOREALISTIC

Another GAN application out of Alexei Efros' A.I. Research lab at Berkeley, and one that you can amuse yourself with yourself straightaway, is *pix2pix*.[15] If you make your way to `affinelayer.com/pixsrv` (convert to bit.ly), you can interactively translate images from one type to another. Using the "edges2cats" tool, for example, we sketched the three-eyed cat in the left-hand panel of Figure 3.6 to generate the photorealistic(-ish) mutant kitty in the right-hand panel. As it takes your fancy, you are also welcome to convert your own creative visions of felines, shoes, handbags and building façades into photorealistic analogs within your browser. The authors of the pix2pix paper called their approach a *conditional* GAN (cGAN for short) because the generative adversarial network produces an output that is conditional on the particular input provided to it.

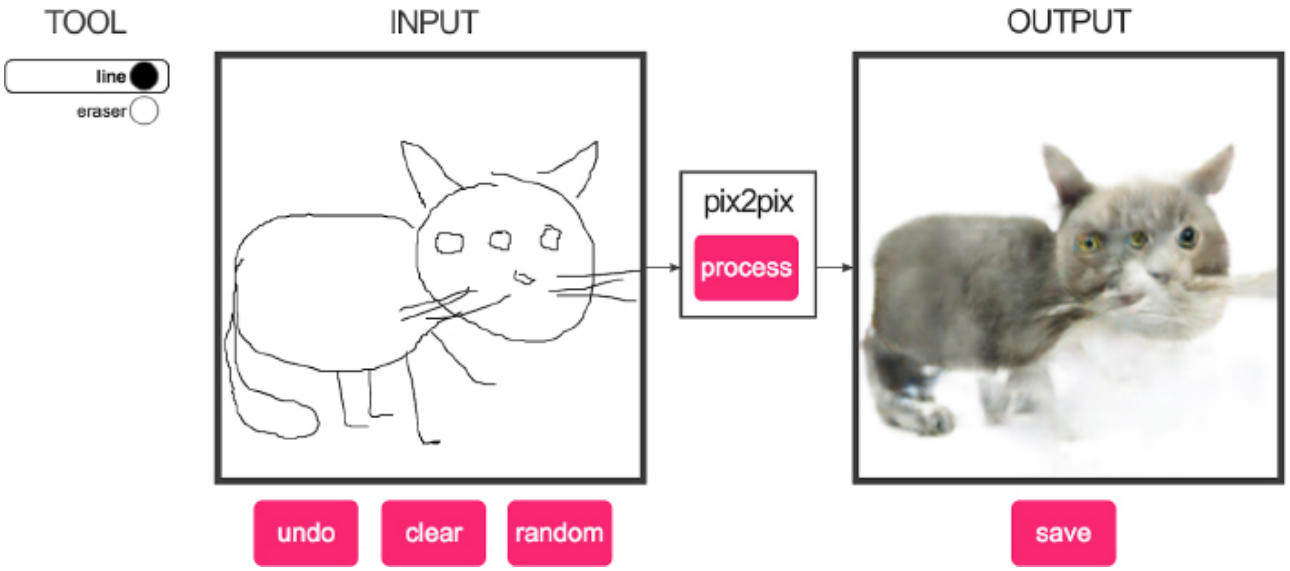

Figure 3.6 A mutant three-eyed cat (right-hand panel) synthesized via the pix2pix web application. The sketch in the left-hand panel that the GAN output was conditioned on was clearly not doodled by this book's illustrator, Aglaé, but one of its other authors (who shall remain nameless).

CREATING PHOTOREALISTIC IMAGES FROM TEXT

To round out this chapter, we'd like you to take a *gan*der at the truly photorealistic high-resolution images in Figure 3.7. These images were generated by StackGAN[16], an approach that *stacks* two GANs on top of each other. The first GAN in the architecture is configured to produce a rough, low-resolution image with the general shape and colors of the relevant objects in place. This is then supplied to the second GAN as its input, where the forged "photo" is refined by fixing up imperfections and adding considerable detail. The StackGAN is a cGAN like the pix2pix network in the previous section, however, the image output is conditioned on *text* input instead of an image.

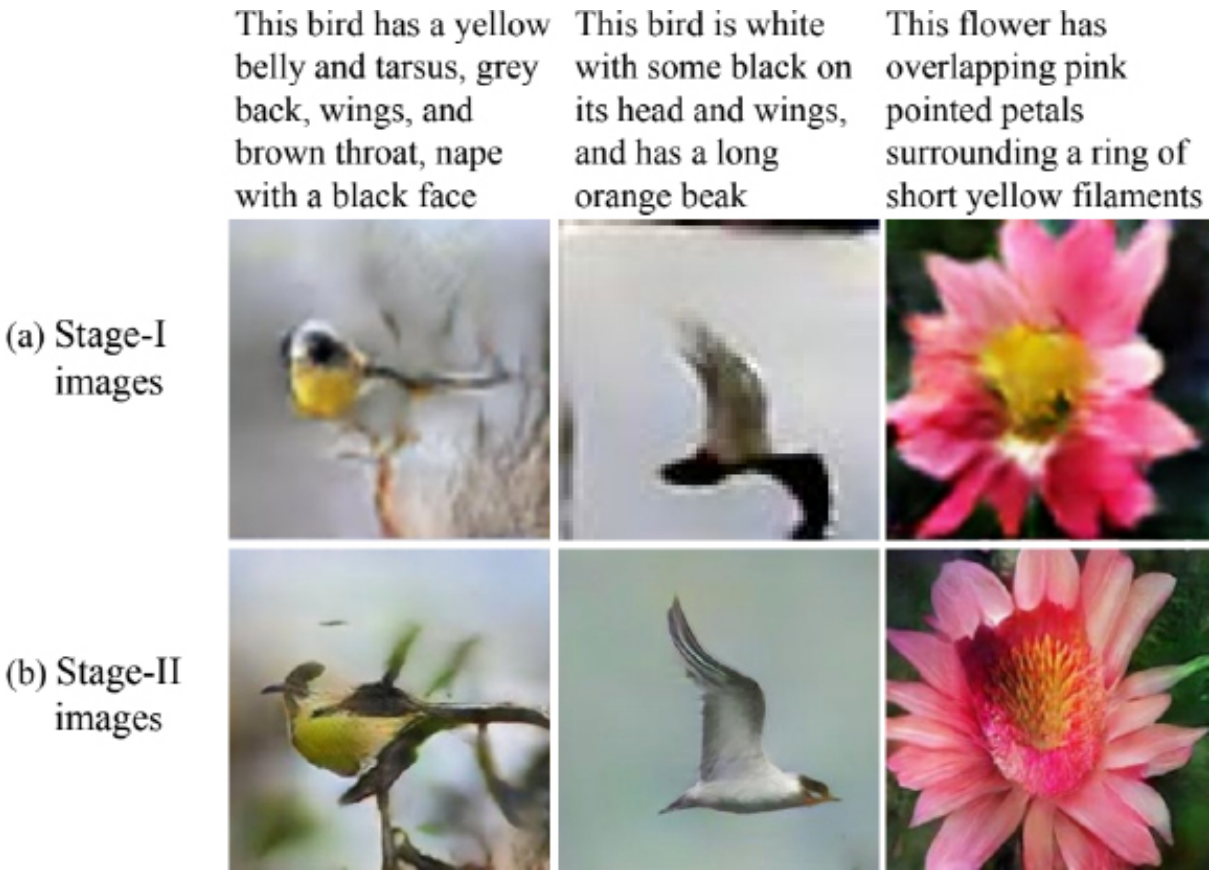

Figure 3.7 Photorealistic high-resolution images output by StackGAN, which involves two GANs stacked upon each other. See text for further detail.

IMAGE PROCESSING USING DEEP LEARNING

Since the advent of digital camera technology, image processing (both on-device and post-processing) has become a staple in most (if not all) photographers' workflows. This ranges from simple on-device processing, such as increasing saturation and sharpness immediately after capture, to complex editing of RAW image files in software applications like Adobe Photoshop and Lightroom.

Machine learning has been used extensively in on-device processing, where the camera manufacturer would like the image that the consumer sees to be vibrant and pleasing to the eye with minimal end-user effort. Some examples of this are:

œ early face recognition algorithms in point-and-shoot cameras which optimize the exposure and focus for faces or even selectively fire the shutter when they recognize that the subject is smiling (as in Figure 1.14);

œ scene-detection algorithms that adjust the exposure settings to capture the whiteness of snow or activate the flash for nighttime photos.

In the post-processing arena, a variety of automatic tools exist although generally photographers who are taking the time to post-process images are investing considerable time and domain-specific knowledge into color and exposure correction,

de-noising, sharpening, tone-mapping and touching up (to name just a few of the corrections that may be applied).

These corrections have been difficult to execute programmatically because, for example, de-noising might need to be applied selectively to different images and even different parts of the same image. This is exactly the kind of intelligent application that deep learning is poised to excel at.

In a 2018 paper from Chen Chen and his collaborators at Intel Labs[17], deep learning was applied to the enhancement of images that were taken in near total darkness, with astonishing results (Figure 3.8). In a phrase, their deep learning model involves convolutional layers organized into the innovative *U-Net*[18] architecture (that we'll break down for you in Chapter 10). The authors generated a custom dataset for training this model: the See-in-the-Dark (SID) dataset consists of 5094 raw images taken in near total darkness using a short-exposure (that is, a short enough exposure time to enable practical hand-held capture without motion blur but which renders images too dark to be useful) with a corresponding long-exposure image (using a tripod for stability) of the same scene. The exposure times on the long-exposure images were 100–300 times that of the short-exposure training images, with actual exposure times in the range of 10–30 seconds. The raw short-exposure images were fed into the convolutional layers, using the long-exposure images as the ground truth.

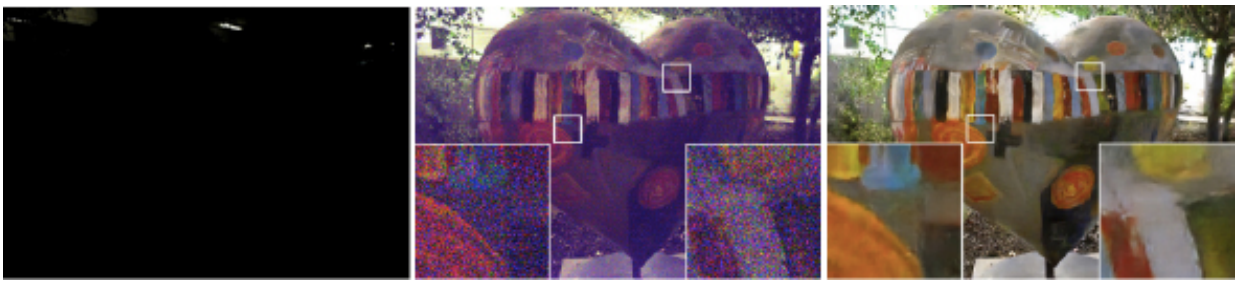

Figure 3.8 A sample image (left) processed using a traditional pipeline (center) and the deep learning pipeline by Chen at al. (2018)

The trained model demonstrates a remarkable ability to brighten images from near total darkness with successful noise suppression and correct color transformation (Figure 3.8)—the deep-learning image-processing pipeline far outperforms the results of the traditional pipeline shown in the center panel. There are, however, limitations as yet:

œ the model is not fast enough to perform this correction in real time (and certainly not on-device), however runtime optimization will certainly help here;

œ a dedicated network must be trained for different camera models and sensors,

whereas a more generalized and camera model-agnostic approach would be favorable;

œ while the results far exceed the capabilities of traditional pipelines, there are still some artifacts present in the enhanced photos which could stand to be improved;

œ and, finally, the dataset is limited to selected static scenes and needs to be expanded to other subjects (most notably, humans).

Limitations aside, this work nevertheless provides a beguiling peek into how deep learning can adaptively correct images in photograph post-processing pipelines with a level of sophistication not before seen from machines.

SUMMARY

I'm not sure if this chapter will have convinced you that GANs are artists, or even that you can be an artist yourself if you skip ahead to Chapter 12, but hopefully it has introduced the idea that deep learning models—GANs in particular, in this case—encode some fairly sophisticated representations in their latent spaces. That said, a model is only as good as the data, and that certainly applies here. These networks produce some sensational results, but like all deep-learning models, they're constrained by what they're trained on. A model that turns your family photos into Monet paintings can't do very much else. As it turns out, the model has just learned the *features* typical to Monet paintings and can mathematically transform pixels so that they conform and we're thusly entertained. This, I think, is a long way from general creativity. This has been a brief introduction to an incomplete list of creative applications for deep learning. In Chapter 12 we'll be making our own GAN using sketches from Google's *Quick, Draw!* dataset (introduced in Chapter 1). Take a gander at Figure 3.9 for a preview.

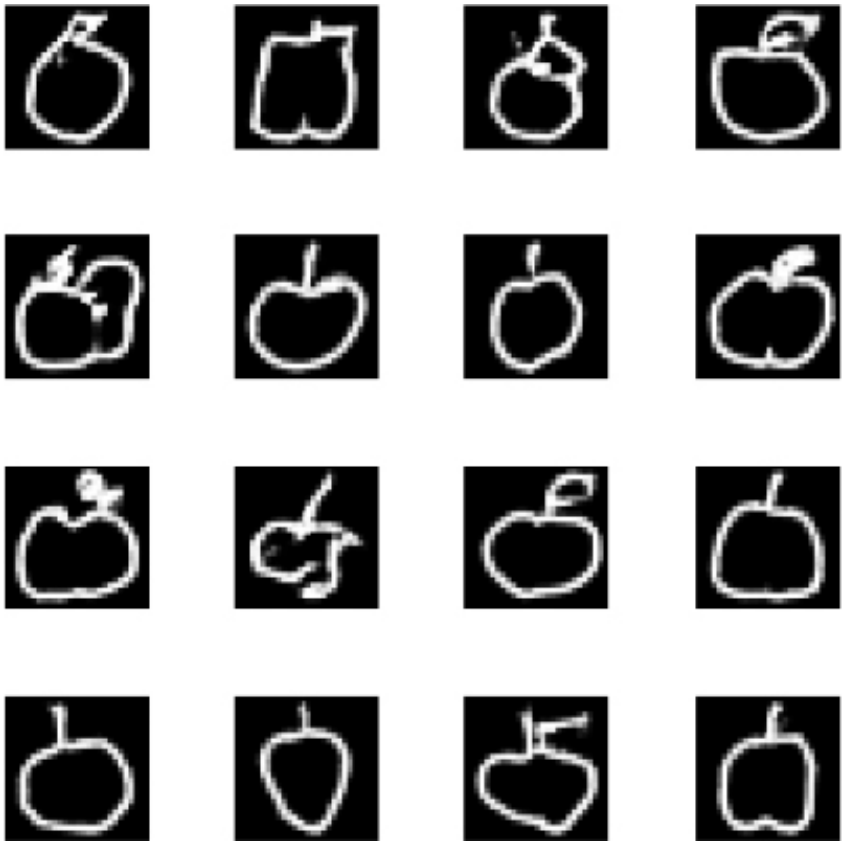

Figure 3.9 Novel "hand-drawings" of apples produced by the GAN architecture we'll develop together in Chapter 12. Using this approach, you'll trivially be able to produce machine-drawn "sketches" from across any of the hundreds of categories involved in the Quick, Draw! game.

1 . Noë, A. (2015, October 5). What Art Unveils. *The New York Times*. bit.ly/unveils

2 . Giles, M. (2018, February 21). The GANfather: The man who's given machines the gift of imagination. *MIT Technology Review*.

3 . LeCun, Y. (2016, July 28). *Quora*. `bit.ly/DLbreakthru`.

4 . Jarosz, A., et al. (2012). Uncorking the muse: Alcohol intoxication facilitates creative problem solving. *Consciousness and Cognition, 21,* 487-93.

5 . Goodfellow, I., et al. (2014). Generative Adversarial Networks. *arXiv:1406.2661*.

6 . From LeCun's classic MNIST data set that we'll use ourselves in Part II.

7 . From the Hinton research group's *Toronto Face Database*.

8 . The *CIFAR-10* dataset, which is named after the Canadian Institute for Advanced

Research that supported its creation.

9 . We'll detail these in Chapter 10.

10. *Dense* layers, which will be introduced in the next chapter and detailed in Chapter 7.

11. Radford, A., et al. (2016). Unsupervised Representation Learning with Deep Convolutional Generative Adversarial networks. *arXiv:1511.06434v2*.

12. A technical aside: As was the case with vector spaces, this "age" axis (or any other direction within latent space that represents some meaningful attribute) may be orthogonal to all of the *n* dimensions that constitute the axes of the *n*-dimensional space.

13. Karras, T., et al. (2018). Progressive Growing of GANs for Improved Quality, Stability, and Variation. *Proceedings of the International Conference on Learning Representations*.

14. Called "CycleGANs" because they retain image consistency over multiple cycles of network training. Zhu, J.-Y., et al. (2017). Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks. *arXiv:1703.10593*.

15. Isola, P., et al. (2017). Image-to-Image Translation with Conditional Adversarial Networks. *arXiv:1611.07004*.

16. Zhang, H., et al. (2017). StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks. *arXiv:1612.03242v2*.

17. Chen, C. et al. (2018) Learning to See in the Dark. *arXiv:1805.01934*

18. Ronneberger et al. (2015) U-Net: Convolutional Networks for Biomedical Image Segmentation. *arXiv: 1505.04597*

4 Game-Playing Machines

Alongside the generative adversarial networks introduced in Chapter 3, deep *reinforcement* learning has produced some of the most surprising artificial-neural-network advances, including the lion's share of the headline-grabbing "artificial intelligence" breakthroughs of recent years. In this chapter, we'll introduce what reinforcement learning is as well as how its fusion with deep learning has enabled machines to meet or surpass human-level performance on a diverse range of complex challenges, including Atari video games, the board-game Go, and subtle physical-manipulation tasks.

DEEP LEARNING, AI, AND OTHER BEASTS

Earlier in this book, we introduced deep learning with respect to vision (Chapter 1), language (Chapter 2) and the generation of novel "art" (Chapter 3). In doing this, we've loosely alluded to deep learning's relationship to the concept of artificial intelligence. At this stage, as we begin to cover deep reinforcement learning, it is worthwhile to define these terms more thoroughly as well as the terms' relationships to one another. As usual, we will be assisted by visual cues—in this case, the Venn diagram in Figure 4.1.

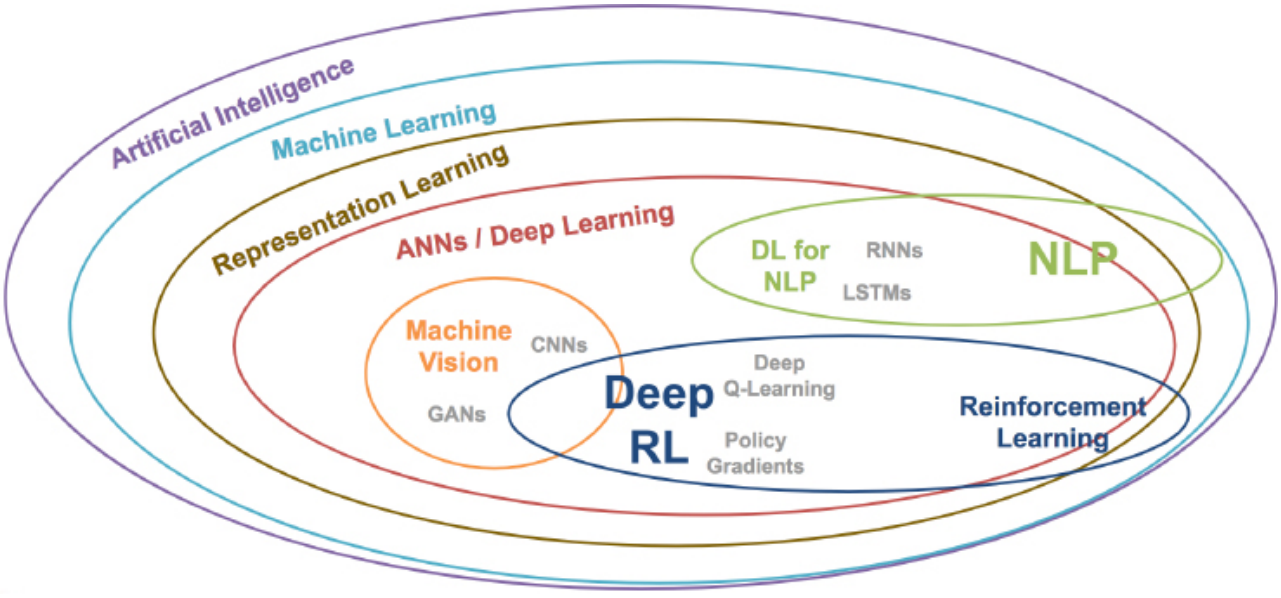

Figure 4.1 Venn diagram that conveys the relative positioning of the major concepts covered over the course of this book.

Artificial Intelligence

Artificial intelligence is the buzziest, vaguest and broadest of the terms we'll be covering in this section. Taking a stab at a technical definition regardless, a decent one is that AI involves a machine processing information from its surrounding environment and then factoring that information into decisions toward achieving some desired outcome. Perhaps given this, some consider the goal of AI to be the achievement of "general intelligence"—intelligence as it is generally referred to with respect to broad reasoning and problem-solving capabilities. [1] In practice and particularly in the popular press, "AI" is used to describe any cutting-edge machine capability. Presently, these capabilities include voice recognition, describing what's happening in a video, question-answering, driving a car, industrial robots that mimic human exemplars in the factory, and dominating humans at "intuition-heavy" board games like Go. Once an AI capability becomes commonplace (e.g., recognizing handwritten digits, which was cutting-edge in the '90s; see Chapter 1), the "AI" moniker is typically dropped by the popular press for that capability such that the goal posts on the definition of AI are always moving.

Machine Learning

Machine learning is a subset of AI alongside other facets of AI like robotics. It is a field of computer science concerned with setting up software in a manner so that the software can recognize patterns in data without the programmer needing to explicitly dictate how the software should carry out all aspects of this recognition. That said, the programmer would typically have some insight into or hypothesis about how the problem might be solved, and would thereby provide a rough model framework and relevant data such that the learning software is well-prepared and well-equipped to solve the problem. As depicted in Figure 1.13 and discussed time and again within the earlier chapters of this book, machine learning traditionally involves cleverly—albeit manually, and therefore laboriously—processing raw inputs to extract features that jive well with data-modeling algorithms.

Representation Learning

Peeling back another layer of the Figure 4.1 onion, we find *representation learning*. This term was introduced at the start of Chapter 2 so we won't explicate on it in too much detail again here. To recap briefly, representation learning is a branch of machine learning in which models are constructed in a way that—provided they are fed enough data—they learn features (or *representations*) automatically. These learned features may wind up being both more nuanced and more comprehensive than their manually-curated cousins. The trade-off is that the learned features might not be as well

understood nor as straightforward to explain, although academic and industrial researchers alike are increasingly tackling these hitches. [2]

Artificial Neural Networks

Artificial neural networks (ANNs) dominate the field of representation learning today. As was touched on in earlier chapters and will be laid bare in Chapter 6, artificial neurons are simple algorithms inspired by biological brain cells, especially in the sense that individual neurons—whether biological or artificial—receive input from many other neurons, perform some computation, and then produce a single output. An artificial neural network, then, is a collection of artificial neurons arranged so that they send and receive information between each other. Data (e.g., images of handwritten digits) are fed into an ANN, which processes these data in some way with the goal of producing some desired result (e.g., an accurate guess as to what digits are represented by the handwriting).

Deep Learning

Of all the terms in Figure 4.1, *deep learning* is the easiest to define because it's so precise. We mentioned a couple of times already in this book that a network composed of at least a few layers of artificial neurons can be called a deep learning network. As exemplified by the classic architectures in Figures 1.9, 1.12 and 1.18; diagramed simply in Figure 4.2; and will be fleshed out fully in Chapter 7, deep learning networks have a total of five or more layers with the following structure:

1. A single *input* layer that is reserved for the data being fed into the network.

2. Three or more *hidden* layers that learn representations from the input data. A general-purpose and frequently used type of hidden layer is the *dense* type, in which all of the neurons in a given layer can receive information from each of the neurons in the previous layer (it is apt, then, that a common synonym for "dense layer" is *fully-connected layer*). In addition to this versatile hidden-layer type, there is a cornucopia of specialized types for particular use cases; we'll touch on the most popular ones as we make our way through this section.

3. A single *output* layer that is reserved for the values (e.g., predictions) that the network yields.

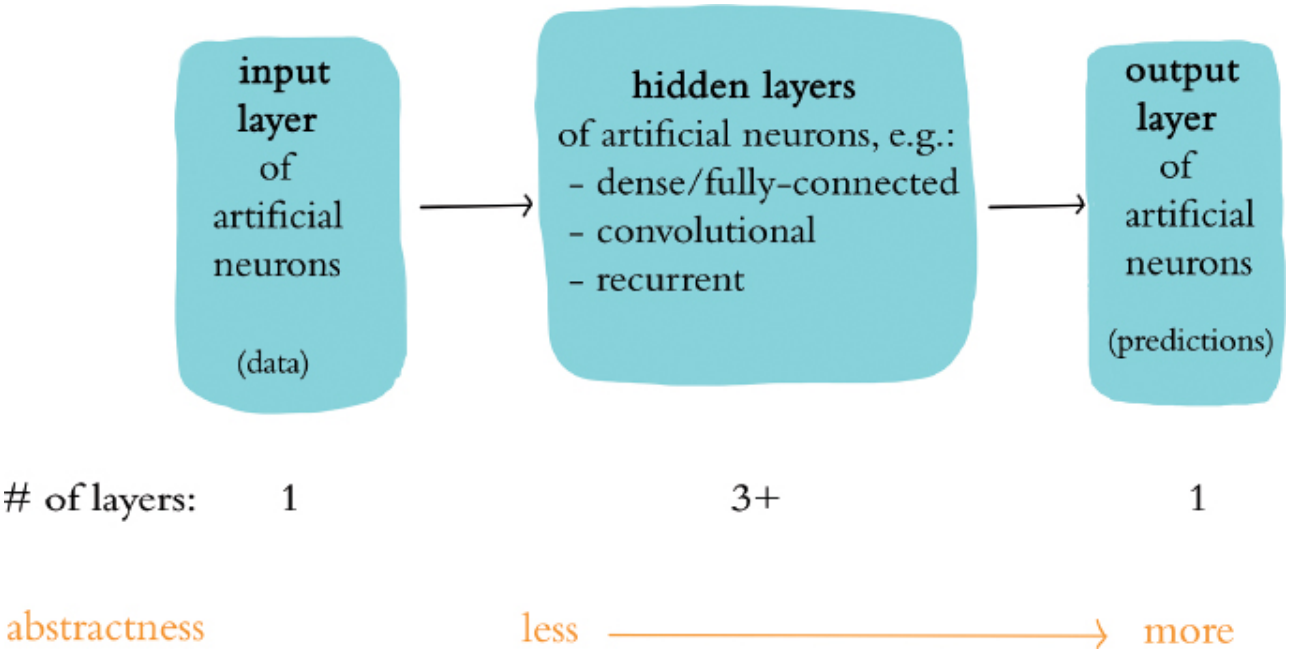

Figure 4.2 Generalization of deep-learning model architectures.

With each successive layer in the network being able to represent increasingly abstract, non-linear recombinations of the previous layers, deep learning models with fewer than a dozen layers of artificial neurons are often sufficient for learning the representations that are of value for a given problem being solved with a given data set. That said, deep learning networks with hundreds or even upwards of a thousand layers have in occasional circumstances been demonstrated to be of utility. [3]

As rapidly-improving accuracy benchmarks and countless competition wins since AlexNet's 2012 victory in the ILSVRC (Figure 1.16) have demonstrated, the deep learning approach to modeling excels at a broad range of machine-learning tasks. Indeed, with deep learning driving so much of the contemporary progress in AI capabilities, the words "deep learning" and "artificial intelligence" are used essentially interchangeably by the popular press.

Let's move inside the deep learning ring of Figure 4.1 to explore classes of tasks that deep learning algorithms are leveraged for: machine vision, natural language processing and reinforcement learning.

Machine Vision

Via analogy to the biological vision system, Chapter 1 introduced *machine vision*. There we focused on object recognition tasks such as distinguishing handwritten digits or breeds of dogs. Other prominent examples of applications that involve machine vision algorithms include self-driving cars, face-tagging suggestions, and phone-unlocking via face recognition on smartphones. More broadly, machine vision is relevant to any AI that is going to need to recognize objects by their appearance at a distance or navigate a real-world environment.

Convolutional neural networks (ConvNets or CNNs for short) are a prominent type of deep learning architecture in contemporary machine vision applications. A CNN is any deep learning model architecture that features hidden layers of the *convolutional* type. We mentioned convolutional layers with respect to Ian Goodfellow's *generative adversarial network* results in Figure 3.2; we will detail and deploy them in Chapter 10.

Natural Language Processing

In Chapter 2, we covered language and *natural language processing*. Deep learning doesn't dominate natural language applications as comprehensively as it does machine vision applications, so our Venn diagram in Figure 4.1 shows "NLP" in both the deep learning region as well as the broader machine-learning territory. As depicted by the timeline in Figure 2.3, however, deep learning approaches to NLP are beginning to overtake traditional machine learning approaches in the field with respect to both efficiency and accuracy. Indeed, in particular NLP areas like voice recognition (e.g., Amazon's Alexa or Google's Assistant), machine translation (including real-time voice translation over the phone), and aspects of Internet-search engines (like predicting the characters or words that will be typed next by a user), deep learning already predominates. More generally, deep learning for NLP is relevant to any AI that interacts via natural language—be it spoken or typed—including to answer a complex series of questions automatically.

A type of hidden layer that is incorporated into many deep learning architectures in the NLP sphere is the *long short-term memory* (LSTM) cell, a member of the *recurrent neural network* (RNN) family. RNNs are applicable to any data that occur in a sequence such as financial time series data, inventory levels, traffic and weather. We will expound on RNNs, including LSTMs, in Chapter 11 when we incorporate them into predictive models involving natural language data. These language examples will provide a firm foundation even if you're primarily seeking to apply deep learning techniques to the other classes of sequential data.

THREE CATEGORIES OF MACHINE LEARNING PROBLEMS

The one remaining section of the Venn diagram in Figure 4.1 involves reinforcement learning, which is the focus the rest of this chapter. To introduce reinforcement learning, we'll contrast it with the two other principal categories of machine-learning problems: supervised and unsupervised learning.

Supervised Learning

In *supervised learning* problems, we have both an x variable and a y variable, where:

- x represents the data we're providing as input into our model, and
- y represents an outcome we're building a model to predict. This y variable can also be called a *label*.

The goal with supervised learning is to have our model learn some function that uses x to approximate y. Supervised learning typically involves either:

1. *Classification*, where our y-values consist of labels that assign each instance of x into a particular category. In other words, y is a so-called *categorical variable*. Examples include identifying handwritten digits (we will code up models that do this in Chapter 10) or predicting whether someone who has reviewed a film loved it or loathed it (as we'll do in Chapter 11).

2. *Regression*, where our y is a *continuous variable*. Examples include predicting the number of sales of a product, or predicting the future price of an asset like a home or a share in an exchange-listed company.

Unsupervised Learning

Unsupervised learning problems are distinguishable from supervised learning problems by the absence of a label y. Ergo, in unsupervised learning problems, we have some data x that we can put into a model, but we have no outcome y to predict. Rather, our goal with unsupervised learning is to have our model discover some hidden, underlying structure within our data. An oft-used example is that of grouping news articles by their theme. Instead of providing a pre-defined list of categories that the news articles belong to (politics, sports, finance, etc.), we configure the model to group those with similar topics for us automatically. Other examples of unsupervised learning include creating a word-vector space (see Chapter 2) from natural language data (we'll do this in Chapter 11), or producing novel images with a generative adversarial network (as in Chapter 12).

Reinforcement Learning

Returning to Figure 4.1, we're now well-positioned to cover *reinforcement learning* problems, which are markedly different from the supervised and unsupervised varieties. As illustrated light-heartedly in Figure 4.3, reinforcement learning problems are ones that we can frame as having an *agent* take actions within some *environment*. The agent could, for example, be a human or an algorithm playing an Atari video game, or it could be a human or an algorithm driving a car. Perhaps the primary way in which

reinforcement learning problems diverge from supervised or unsupervised ones is that the actions that the agent takes influences the information that the environment provides to the agent—that is, the agent receives direct feedback on the actions it takes. In supervised or unsupervised problems, in contrast, the model never impacts the underlying data, it simply consumes it.

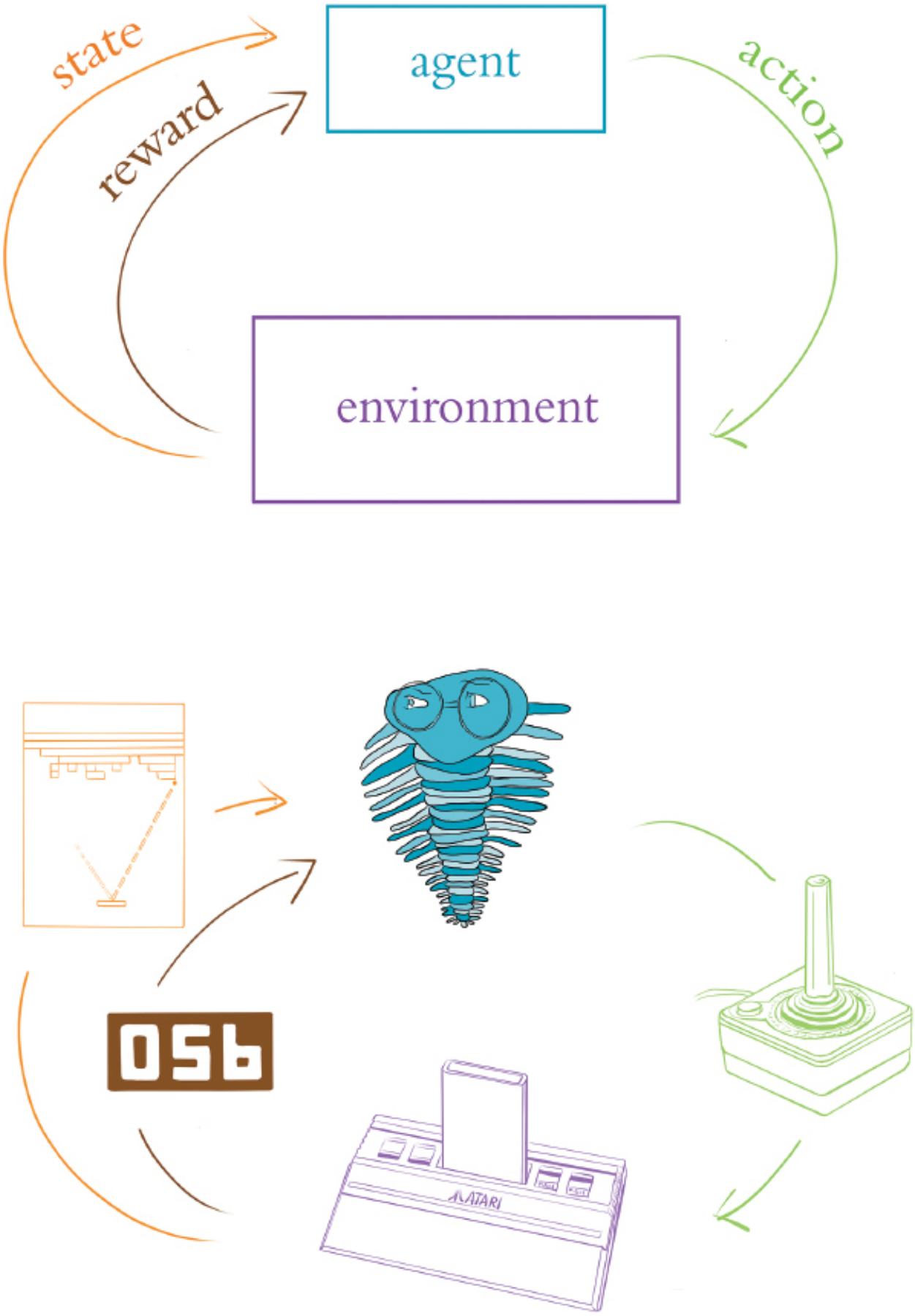

Figure 4.3 The reinforcement learning loop. The top diagram is a generalized version. The bottom diagram is specific to the example elaborated on in the text of an agent playing a video game on an Atari console. To our knowledge, trilobites can't actually play video games; we're using the trilobite as a symbolic representation of the reinforcement learning agent, which could either be a human or a machine.

Let's dive a bit further into the relationship between a reinforcement learning agent and its environment by exploring some examples. In Figure 4.3, the agent is represented by an anthropomorphized trilobite but this agent could be either human or it could be a machine. Where the agent is playing an Atari video game:

œ The possible *actions* that can be taken are the buttons that can be pressed on the video game controller. [4]

œ The *environment* (the Atari console) returns information back to the agent. This information comes in two delicious flavors: *state* (the pixels on the screen that represent the current condition of the environment) and *reward* (the point score in the game, which is what the agent is endeavoring to maximize).

œ If the agent is playing Pac-Man, then selecting the action of pressing the "up" button results in the environment returning an updated state where the pixels representing the video-game character on the screen have moved upward. Prior to playing any of the game, a typical reinforcement learning algorithm would not even have knowledge of this simple relationship between the "up" button and the Pac-Man character moving upward; everything is learned from the ground up via trial and error.

œ If the agent selects an action that causes Pac-Man to cross paths with a pair of delectable cherries, then the environment will return a *positive reward*: an increase in points. On the other hand, if the agent selects an action that causes Pac-Man to cross paths with a spooky ghost, then the environment will return a *negative reward*: a decrease in points.

In a second example where the agent is driving a car:

œ The available *actions* are much broader and richer than for Pac-Man. The agent can adjust the steering column, the accelerator and the brakes to varying degrees ranging from subtle to dramatic.

œ The *environment* in this case is the real world, consisting of roads, traffic, pedestrians, trees, sky and so on. The *state* then is the condition of the vehicle's surroundings, as perceived by a human agent's eyes and ears, or by an autonomous vehicle's cameras and lidar.

œ The *reward*, in the case of an algorithm, could be programmed to be *positive* for, say, every meter of distance travelled toward a destination; it could be slightly *negative* for minor traffic infractions and severely negative in the event of a collision.

DEEP REINFORCEMENT LEARNING

At long last, we reach the *deep reinforcement learning* section near the center of the Venn diagram in Figure 4.3. A reinforcement learning algorithm earns its "deep" prefix when an artificial neural network is involved in it, e.g., to learn what actions to take when presented with a given state from the environment in order to have a high probability of obtaining a positive reward. [5] As we'll see in the examples coming up in the next section, the marriage of deep learning and reinforcement learning approaches has proved a prosperous one. This is because:

œ Deep neural networks excel at processing the complex sensory input provided by real environments or advanced, simulated environments in order to distill relevant signals out from a cacophony of incoming data. This is analogous to the functionality of the biological neurons of your brain's visual and auditory cortices, which receive input from the eyes and ears, respectively.

œ Reinforcement learning algorithms, meanwhile, shine at selecting an appropriate action from a vast scope of possibilities.

Taken together, deep learning and reinforcement learning are a powerful problem-solving combination. Increasingly complex problems tend to require increasingly large data sets for deep reinforcement learning agents to wade through vast noise as well as vast randomness in order to discover an effective policy for what actions it should take in a given circumstance. Since many reinforcement learning problems take place in a simulated environment, obtaining a sufficient amount of data is usually not a problem: The agent can simply be trained on further rounds of simulations.

While the theoretical foundations for deep reinforcement learning have been around for a couple of decades, [6] as with AlexNet for vanilla deep learning (Figure 1.18) deep reinforcement learning has in the past few years benefited from a confluence of tailwinds:

1. Exponentially larger data sets and much richer simulated environments;

2. Parallel computing across many graphics processing units (GPUs) to model efficiently with large data sets as well as the breadth of associated possible states and possible actions;

3. A research ecosystem that bridges academia and industry, producing a quickly-developing body of new ideas on deep neural networks in general as well as on deep reinforcement learning algorithms in particular, e.g., to identify optimal actions across

a wide variety of noisy states.

VIDEO GAMES

Many readers of this book recall learning a new video game as a child. Perhaps while at an arcade or staring at the family's heavy cathode-ray-tube television set, it quickly became apparent that missing the ball in *Pong* or *Breakout* was an unproductive move. We processed the visual information on the screen and, yearning for a score in excess of our friends', devised strategies to manipulate the controller effectively and achieve this aim. In recent years, researchers at a firm called DeepMind have been producing software that likewise learns how to play classic Atari games.

DeepMind was a British technology startup founded by Demis Hassabis (Figure 4.4), Shane Legg and Mustafa Suleyman in London in 2010. Their stated mission was to "solve intelligence", which is to say they were interested in extending the field of AI by developing increasingly general-purpose learning algorithms. One of their early contributions was the introduction of Deep Q-Learning Networks (DQNs; noted within Figure 4.1). Via this approach, a single model architecture was able to learn to play multiple Atari 2600 games well—from scratch, simply through trial and error.

Figure 4.4 Demis Hassabis co-founded DeepMind in 2010 after completing his Ph.D. in cognitive neuroscience at University College London.

In 2013, Volodymyr Mnih [7] and his DeepMind colleagues published [8] on their DQN agent, a deep reinforcement learning approach that we will come to understand intimately when we construct a variant of it ourselves line by line in Chapter 13. Their agent received raw pixel values from its *environment*, a video-game emulator, [9] as its

state information—akin to the way human players of Atari games view a TV screen. In order to efficiently process this information, Mnih et al.'s DQN included a convolutional neural network (CNN), a common tactic for any deep reinforcement learning model that is fed visual data (this is why we elected to overlap "Deep RL" somewhat with "Machine Vision" in Figure 4.1). The handling of the flood of visual input from Atari games (in this case, a little over two million pixels per second) underscores how well-suited deep learning in general is to filtering out pertinent features from noise. Further, playing Atari games within an emulator is a problem that is well-suited to deep reinforcement learning in particular: While they provide a rich set of possible actions that are engineered to be challenging to master, there is thankfully no finite limit on the amount of training data available since the agent can engage in endless rounds of play.

During training, the DeepMind DQN was not provided any hints or strategies—it was provided only with state (screen pixels), reward (its point score, which it is programmed to maximize) and the range of possible actions (game-controller buttons) available in a given Atari game. The model was not altered for specific games, yet it was able to outperform existing machine-learning approaches in six of the seven games Mnih and his co-workers tested it on, even surpassing the performance of expert human players on three. Presumably influenced by this impressive progress, Google acquired DeepMind in 2014 for the equivalent of half a billion U.S. dollars.

In a follow-up paper published in the distinguished journal *Nature*, Mnih and his teammates at now-Google DeepMind assessed their DQN algorithm across 49 Atari games.[10] The results are shown in Figure 4.5: It outperformed other machine-learning approaches on all but three of the games (94% of them), and astonishingly, it scored above human level on the majority of them (59%).[11]

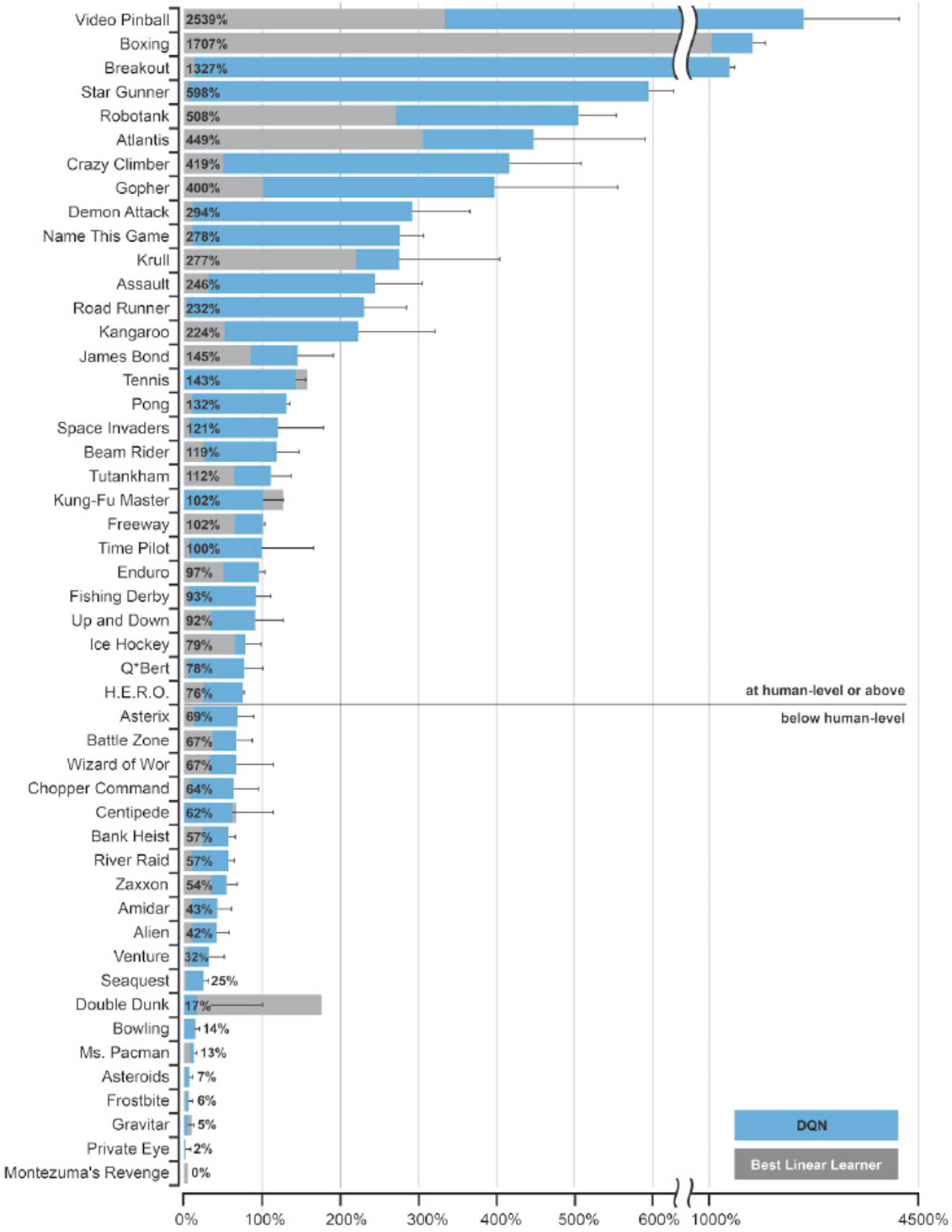

Figure 4.5 The normalized performance scores of Mnih and colleagues' (2015) DQN relative to a professional game tester: Zero percent represents random play, while 100% represents the pro's best performance. The horizontal line represents the authors' defined threshold of "human-level" play: the 75th percentile of professionals' scores.

BOARD GAMES

It might sound sensible that board games would serve as a logical prelude to video games given their analog nature and their chronological head-start, however the use of

software emulators provided a simple and easy way to interact with video games digitally. Instead, the availability of these emulation tools provided the means, and so the principal advances in modern deep reinforcement learning initially took place in the realm of video games. Additionally, relative to Atari games, the complexity of some classical board games is much greater. There are myriad strategies and long-plays associated with chess expertise that are not readily apparent in *Pac-Man* or *Space Invaders*, for example. In this section, we provide an overview of how deep reinforcement learning strategies mastered the board games Go, chess and shogi despite the data-availability and computational-complexity headwinds.

AlphaGo

Invented several millennia ago in China, *Go* (illustrated in Figure 4.6) is a very popular two-player strategy board game in Asia. The game has a simple set of rules based around the idea of capturing one's opponents' pieces (called *stones*) by encircling them with one's own.[12] This uncomplicated premise belies intricacy in practice, however. The larger board and the larger set of possible moves per turn make the game much more complex than, say, chess, for which we've had algorithms that can defeat the best human players for two decades.[13] There are a touch over 2×10^{170} possible legal board positions in Go, which is far more than the number of atoms in the universe[14] and about a *googol* (10^{100}) more complex than chess.

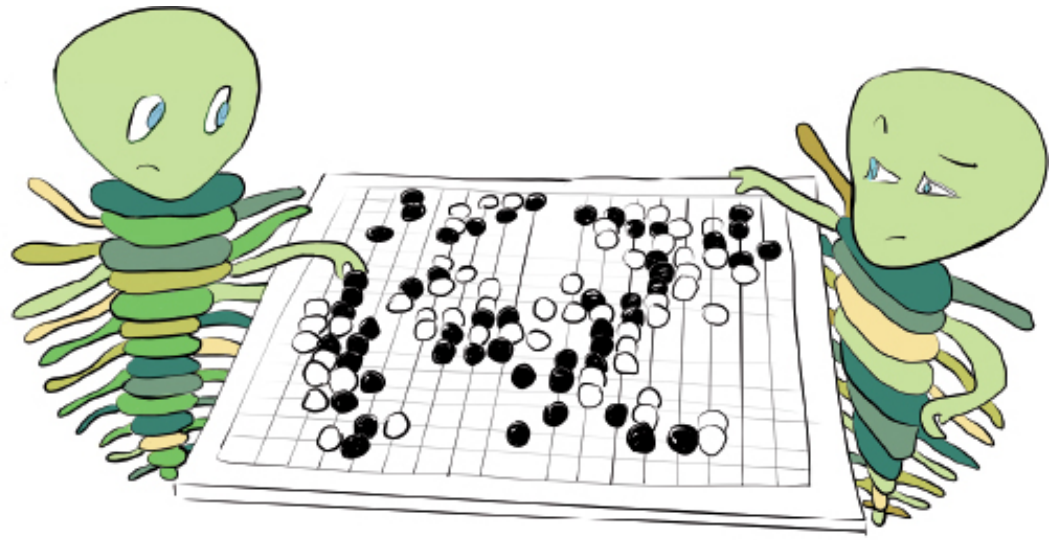

Figure 4.6 The Go board game. One player uses the white stones while the other uses the black stones. The objective is to encircle the stones of your opponent, thereby capturing them.

An algorithm called *Monte Carlo tree search* (MCTS) can be employed to play uncomplicated games competently. In its purest form, MCTS involves selecting random moves[15] until the end of gameplay. By repeating this many times, moves that tended to lead to victorious game outcomes can be weighted as favorable options. Because of the extreme complexity and sheer number of possibilities within sophisticated games like

Go, pure MCTS approach is impractical: There are simply too many options to search through and evaluate. Instead of pure MCTS, an alternative approach involve MCTS applied to a much more finite subset of actions that were curated by, for example, an established policy of optimal play. This curated approach has proved sufficient for defeating amateur human Go players but is uncompetitive against professionals. To bridge the gap from amateur- to professional-level capability, David Silver (Figure 4.7) and his colleagues at Google DeepMind devised a program called *AlphaGo* that combines MCTS with both supervised learning and deep reinforcement learning.[16].

Figure 4.7 David Silver is a Cambridge- and Alberta-educated researcher at Google DeepMind. He has been instrumental in combining the deep learning and reinforcement learning paradigms.

This paragraph will be in a trilobite-reading sidebar. Silver et al. (2016) used supervised learning on a historical database of expert human Go moves to establish something called a *policy network*, which provides a shortlist of possible moves for a given situation. Subsequently, this policy network was refined via *self-play* deep reinforcement learning, wherein both opponents are Go-playing agents of a comparable skill level. Through this self-play, the agent iteratively improves upon itself, and whenever it improves, it is pitted against its now-improved self, producing a positive-feedback loop of continuous advancement. Finally, the cherry atop the AlphaGo algorithm: a so-called *value network* that predicts the winner of the self-play games, thereby evaluating positions on the board and learning to identify strong moves. The combination of these policy and value networks reduces the breadth of search space for

the MCTS. END SIDEBAR.

AlphaGo was able to win the vast majority of games it played against other computer-based Go players. Perhaps most strikingly, AlphaGo was also able to defeat Fan Hui, the then-reigning European Go champion, five games to zero. This marked the first time a computer defeated a professional human player in a full play of the game. As exemplified by the Elo ratings[17] in Figure 4.8, AlphaGo performed at or above the level of the best players in the world. Following this success, AlphaGo was famously matched against Lee Sedol in March 2016 in Seoul, South Korea. Sedol has 18 world titles and is considered one of the great players. The five-game match was broadcast and viewed live by 200 million people. AlphaGo won the match 4-1 launching DeepMind, Go, and the artificially-intelligent future into public imagination.

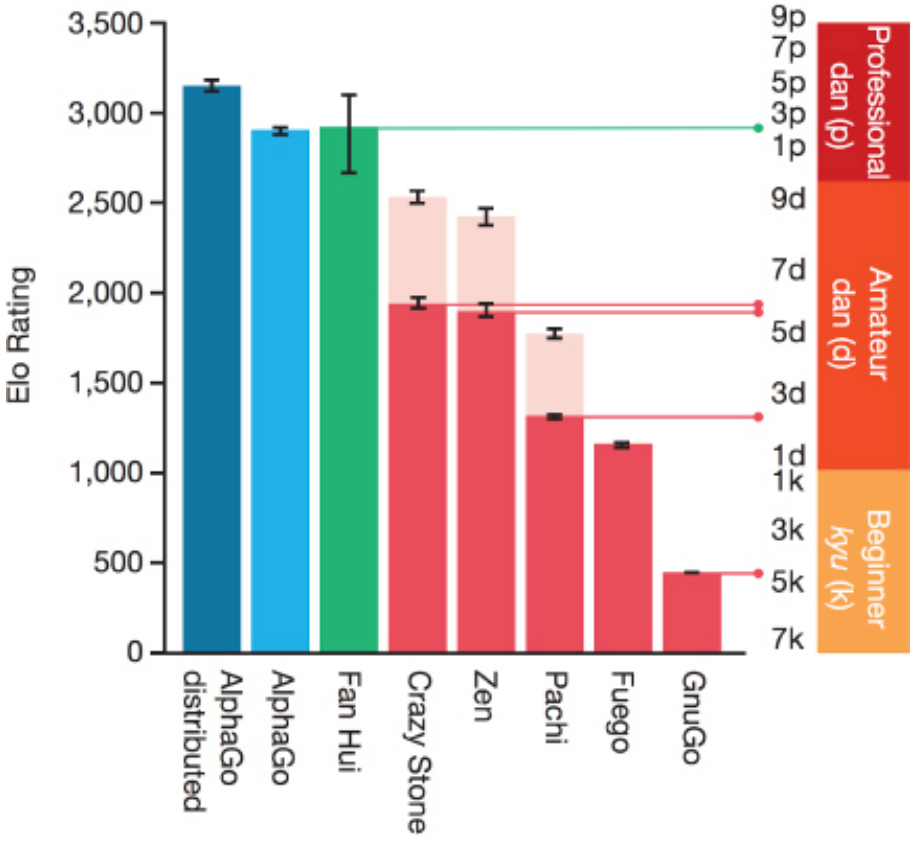

Figure 4.8 The Elo score of AlphaGo (blue) relative to Fan Hui (green) and several Go programs (red). The approximate human rank is shown on the right.

AlphaGo Zero

Having just learned about the incredible feats achieved by AlphaGo, it should come as no surprise that the folks at DeepMind took their work further and created a second-generation Go player: AlphaGo Zero. If you recall from the previous section, AlphaGo was initially trained in a supervised manner; that is, expert human moves were used to train the network initially, whereafter the network learned by reinforcement learning through self-play. While this is still an impressive achievement, it doesn't exactly "solve intelligence" as DeepMind's founders would have liked. A better approximation of

general intelligence would be a network that can learn to play Go in a completely *de novo* setting —where the network is not supplied with any human input or domain knowledge, but improves by deep reinforcement learning alone. Enter: AlphaGo Zero.

As we've alluded to before, the game of Go requires sophisticated lookahead capabilities through vast search spaces. That is, there are so many *possible* moves and such a tiny fraction of them are *good* moves in the short- and long-play of the game that performing a search for the optimal move, keeping the likely future state of the game in mind, becomes impossibly complex and computationally impractical. It is for this reason that it was thought at the time that Go would be a final frontier for machine intelligence —indeed, it was thought that the achievements of AlphaGo in 2016 were still a decade or more away. Working off the momentum from the AlphaGo-Sedol match in Seoul, researchers at DeepMind created *AlphaGo Zero* which learns to play Go far beyond the level of the original AlphaGo—while being revolutionary in several ways.[18] First and foremost, it is trained without any data from human gameplay. That means it learns purely by trial-and-error. Second, it uses only the stones on the board as input features. Contrastingly, AlphaGo received 15 additional features during training, which included information such as how many turns since a move was played or how many opponent stones would be captured. Third, a single (deep) neural network was used to evaluate the board and decide on a next move, rather than separate policy and value networks. Finally, the tree search is simpler and relies on the neural network to evaluate positions and possible moves.

AlphaGo Zero played almost five million games of self-play over three days, taking an estimated 0.4s per move to "think". Within 36 hours, it had begun to outperform the model that beat Lee Sedol in Seoul (retrospectively termed *AlphaGo Lee*), which—in stark contrast—took several months to train. At the 72-hour mark, the model was pitted against AlphaGo Lee in match conditions, where it handily won every single one of a hundred games. Even more remarkable is that AlphaGo Zero achieved this on a single machine with four tensor processing units (TPUs)[19] whereas AlphaGo Lee was distributed over multiple machines and used 48 TPUs (Similarly, AlphaGo Fan, which beat Fan Hui, was distributed over 176 GPUs). In Figure 4.9, the Elo score[20] for AlphaGo Zero is shown over time compared to the scores for AlphaGo Master[21] and AlphaGo Lee. On the right we can see the absolute Elo scores for a variety of iterations of AlphaGo and some other Go programs. AlphaGo Zero is far-and-away the superior Go player amongst this prestigious group.

Another interesting point that emerged from this research was that the nature of the game-play by AlphaGo Zero is qualitatively different to that of human players and AlphaGo Lee. AlphaGo Zero began with random play, but quickly learned professional

joseki —corner sequences that are considered heuristics of good play. Eventually, after further training, the mature model tended to prefer novel *joseki* that were previously unknown to humankind. AlphaGo Zero did spontaneously learn a whole range of classical Go moves, implying a pragmatic alignment with these techniques. However, the model did this in a novel manner: It did not learn the concept of *shicho* (ladder sequences), for example, until much later in its training, whereas this is one of the first concepts taught to new human players. The authors additionally trained another iteration of the model in a supervised manner. This supervised model performed better initially, however it began to succumb to the self-learned model within the first 24 hours of training and ultimately achieves a lower Elo score (Figure 4.9). Together, these results suggest that the self-learned model might have a distinct style of play to that of human players; a more dominating style and one that the supervised model fails to develop.

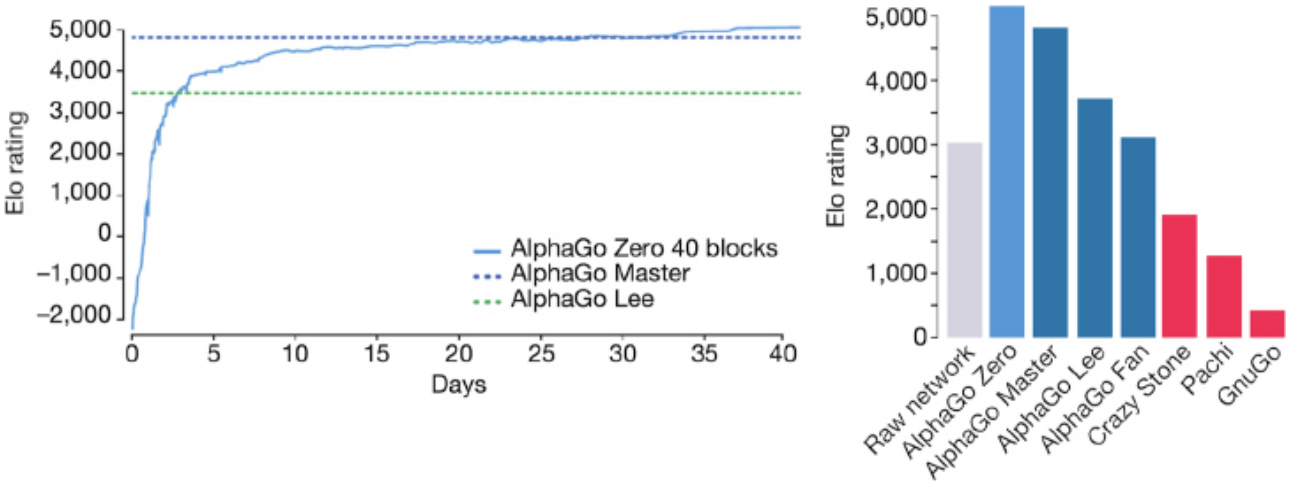

Figure 4.9 Comparing Elo scores between AlphaGo Zero and other AlphaGo variations or other Go programs. In the left-hand figure, the comparison is over days of AlphaGo Zero training.

AlphaZero

Following these successes, the team at DeepMind went on to look into even more general game-playing neural networks. While AlphaGo Zero is adept at playing Go, could a comparable network learn to play multiple games well? To put this to the test, they added two new games to the repertoire: chess and shogi[22]. Most readers are likely familiar with the game of chess, and shogi—referred to by some as Japanese chess—is similar. Both games are two-player strategy games, both take place on a grid-format board, both culminate in a checkmate of the opponent's king, and both consist of a range of pieces with different moving abilities. However, shogi is significantly more complex than chess given the larger board size (9x9, versus 8x8 in chess) and the fact that opponent pieces can be replaced anywhere on the board after their capture.

Historically, artificial intelligence has had a rich interaction with the game of chess.

Over several decades, computer programs that play chess have been developed extensively. Perhaps the most famous to date is *Deep Blue*, a creation of IBM, that went on to beat the world champion Garry Kasparov in 1997[23]. It was heavily reliant on brute-force computing power[24] to execute complex searches through possible moves, and combined this with hand-crafted features and domain-specific adaptations. The evaluation function of Deep Blue was fine-tuned by analyzing thousands of master games (it was a supervised-learning system!) and this function was even tweaked between games.[25] It's clear that Deep Blue, and other chess programs like it, were hard-coded to play chess and nothing else.

While Deep Blue was an achievement two decades ago, however their system was not generalizable—it conceivably would not perform well at any task other than chess. After AlphaGo Zero demonstrated that the game of Go could be learned by a neural network from first principles alone, given nothing but the board and the rules of the game, Silver and his colleagues set out to devise a generalist neural network, a *single network architecture* that could dominate not only at Go, but other board games as well.

Compared to Go, chess and shogi present pronounced obstacles. The rules of the games are position-dependent (in that pieces can move differently based on where they are on the board), asymmetrical (pieces can only move in one direction)[26], long-range actions are possible (such as the queen moving across the entire game), captured pieces can be replaced in shogi, and the game can result in a draw.

AlphaZero feeds the board positions into a neural network and outputs a vector of move probabilities for each possible action, as well as a scalar[27] outcome value for that move. The network learns the parameters for these move probabilities and outcomes entirely from self-play deep reinforcement learning, AlphaGo Zero did. An MCTS is then performed on the reduced space guided by these probabilities, returning a vector of probabilities over the possible moves. Where AlphaGo Zero optimized the probability of winning (Go is a binary win/loss game), Alpha Zero instead optimizes for the expected outcome. During self-play, AlphaGo Zero would retain the best *player* to date, and evaluate updated versions of itself against that player, continually replacing the player with the next best version; Alpha Zero instead maintains a single network and at any given time is playing against the latest version of itself. Alpha Zero was trained to play each of chess, shogi and Go for a mere 24 hours. There were no game-specific modifications, with the exception of a noise parameter that promotes move exploration —this was scaled to the number of legal moves in each game.

After one hundred games, Alpha Zero had not lost a single game against the 2016 Top Chess Engine Championship (TCEC) world champion *Stockfish*. In shogi, the Computer

Shogi Association (CSA) world champion *Elmo* managed to beat Alpha Zero only eight times in 100 games. Perhaps its most worthy opponent, AlphaGo Zero was able to defeat Alpha Zero in forty of their hundred games. Figure 4.10 shows the Elo scores for Alpha Zero relative these three adversaries. Not only was Alpha Zero superior; it was efficient. Alpha Zero's Elo score exceeded its greatest foes' after just two, four and eight hours for shogi, chess and Go, respectively. This is a sensationally rapid rate of learning, considering that in the case of Elmo and Stockfish, these computer programs represent the culmination of decades of research and fine-tuning in a focused, domain-specific manner. The generalizable Alpha Zero algorithm is able to play all three games with aplomb: Simply switching out learned weights from otherwise identical neural network architectures imbues each with the same skills that have taken *years* to develop by other means. These results demonstrate that deep reinforcement learning is a strikingly powerful approach for developing general expert gameplay in an undirected fashion.

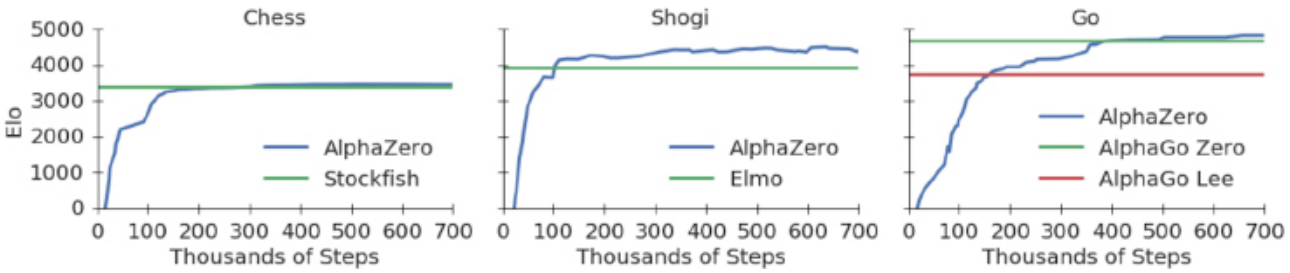

Figure 4.10 Comparing Elo scores between Alpha Zero and each of its opponents in chess, shogi and Go. Alpha Zero rapidly outperformed all three opponents.

MANIPULATION OF OBJECTS

As this chapter's title might have suggested, we've centered our coverage of deep reinforcement learning on its game-playing applications. While games offer a hot testbed for exploring the generalization of machine intelligence, in this section we'll spend a few moments expounding on a practical, real-world applications of deep reinforcement learning as well. We mentioned some such applications earlier in this chapter: autonomous vehicles are an excellent example (and it might be plain to see, this application isn't *that* different from game-playing, albeit with higher stakes).

As an example, we'll provide an overview of research by Sergey Levine, Chelsea Finn (Figure 4.11) and lab-mates at the University of California, Berkeley[28]. These researchers trained a robot to perform a number of motor skills that require complex visual understanding and depth perception, such as screwing the cap back onto a bottle, removing a nail with a toy hammer, placing a hanger on a rack or inserting a cube in a shape-fitting game (Figure 4.12).

Figure 4.11 Chelsea Finn is a doctoral candidate at the University of California, Berkeley in its AI Research Lab.

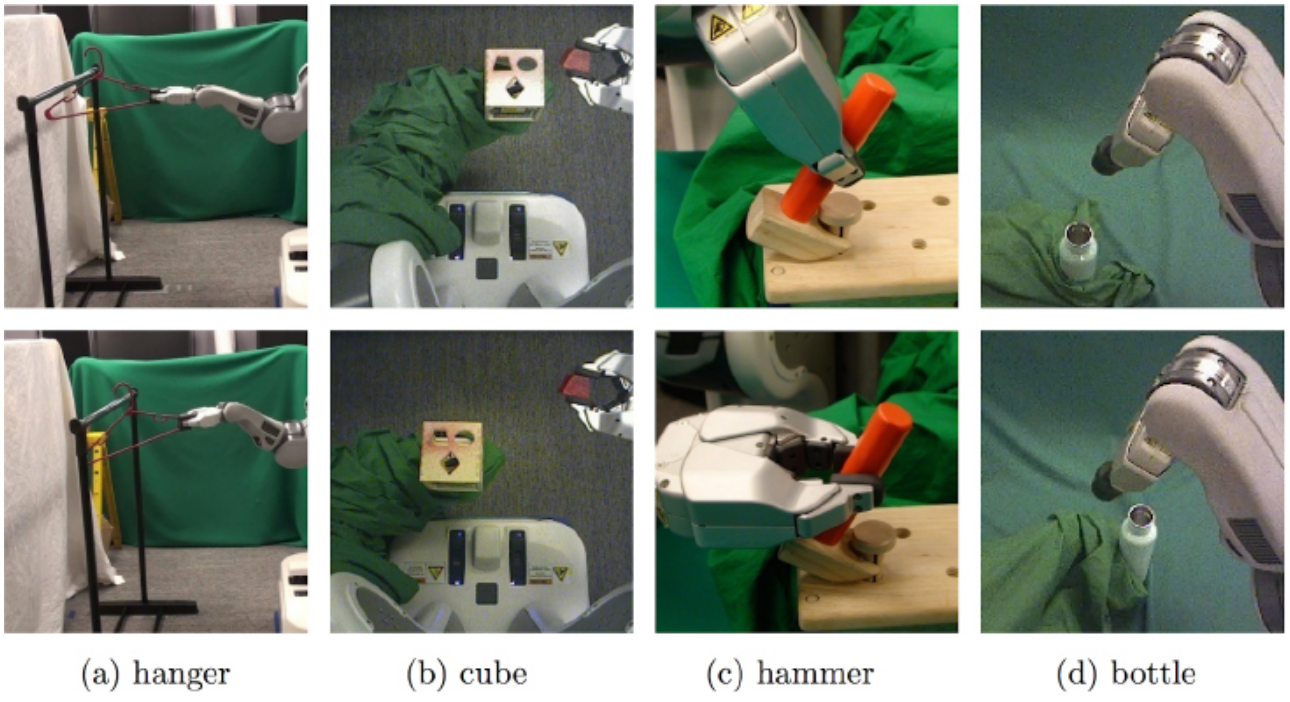

Figure 4.12 Sample images from Levine, Finn et al. (2016) exhibiting various object-manipulation actions the robot was trained to perform.

Levine, Finn and colleagues' algorithm maps raw visual input directly to the movement of the motors in the robot's arm. Their policy network was a seven-layer-deep convolutional neural network (CNN) consisting of less than a hundred thousand artificial neurons—a minuscule amount in deep-learning terms, as we'll see when we train orders-of-magnitude larger networks later in this book. While it would be tricky to

elaborate further on this approach before we've delved much into artificial-neural-network theory (Part II, which is just around the corner), there are three takeaway points we'd like to highlight on this elegant practical application of deep reinforcement learning. First, it is an "end-to-end" deep learning model in the sense that the model takes in raw images (pixels) as inputs and then outputs directly to the robot's motors. Second, the model generalizes neatly to a broad range of unique object-manipulation tasks. Third, it is an example of the *policy gradient* family of deep reinforcement learning approaches, rounding out the terms featured in the Venn diagram in Figure 4.1. Policy gradient methods are distinct from the DQN approach that is the focus of Chapter 13 but we'll touch on it then too.

POPULAR DEEP REINFORCEMENT LEARNING ENVIRONMENTS

Over the last few sections, we've talked a fair bit about software emulation of environments in which to train reinforcement learning models. This area of development is crucial to the ongoing progression of reinforcement learning—without environments in which our agents can play and explore (and gather data!) there would be no training of models. Here we'll introduce a the three most popular environments, discussing their high-level attributes.

OpenAI Gym

The OpenAI Gym[29] is developed by the non-profit AI research company OpenAI[30]. The mission of OpenAI is to advance artificial general intelligence (more on that in the next section!) in a safe and equitable manner. To that end, the researchers at OpenAI have produced and open-sourced a number of tools for AI research, including the OpenAI Gym. This toolkit is designed to provide an interface for training reinforcement learning models, be they deep or otherwise. As captured in Figure 4.13, the Gym includes a wide variety of environments, including a number of Atari 2600 games,[31] multiple robotics simulators, a few simple text-based algorithmic games and several robotics simulations using the MuJoCo physics engine[32]. In Chapter 13, we'll install OpenAI Gym in a single line of code and then employ an environment it provides to train the DQN agent that we build. The gym is written in Python and is compatible with any deep-learning computation library, e.g., TensorFlow (Chapter 14) and PyTorch (Chapter 15).

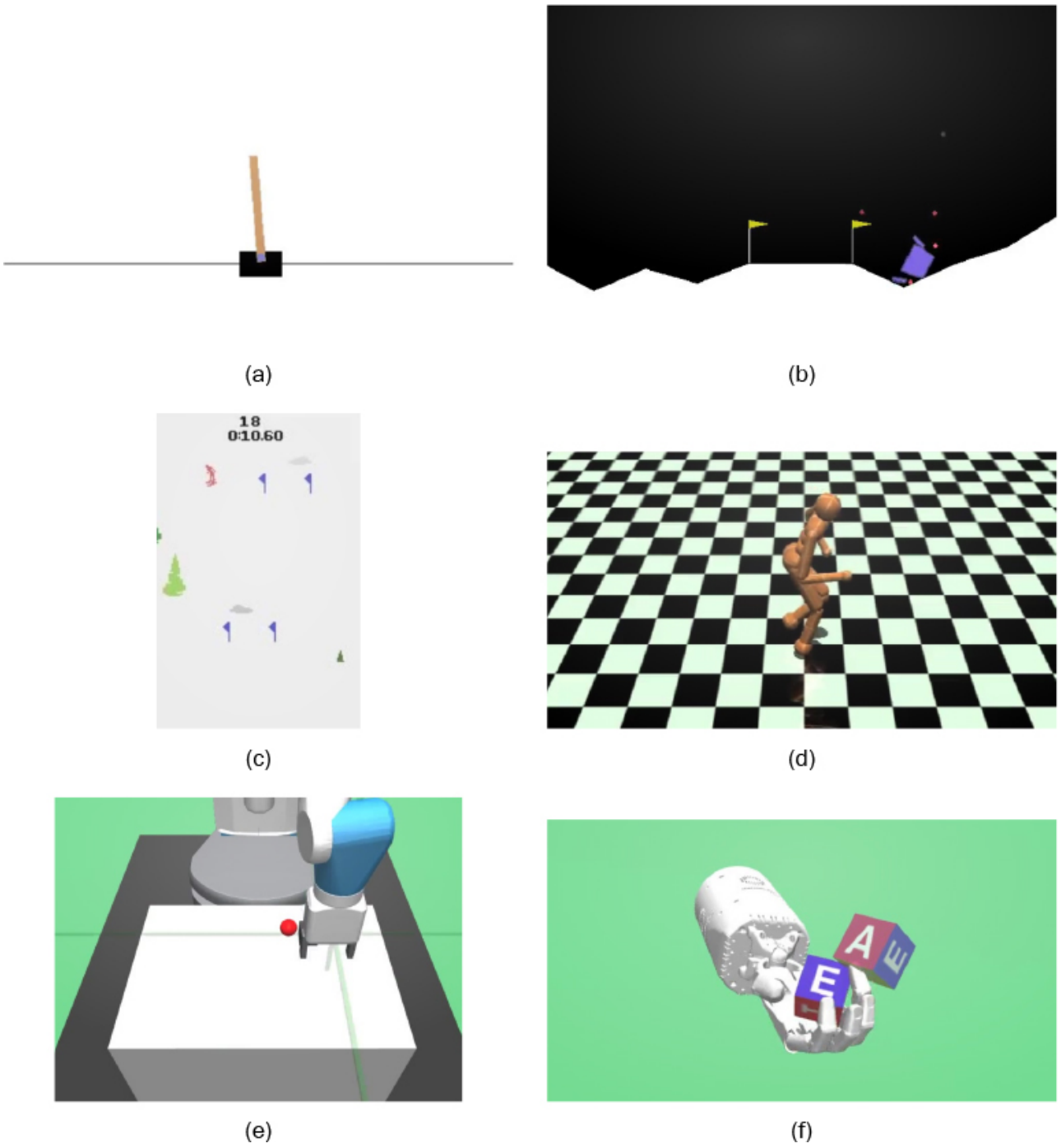

Figure 4.13 A sampling of OpenAI Gym environments. (a) CartPole, a classic control-theory problem. (b) LunarLander, a continuous-control task run inside a two-dimensional simulation. (c) Skiing, an Atari 2600 game. (d) Humanoid, a three-dimensional MuJuCo physics engine simulation of a bipedal person. (e) FetchPickAndPlace, one of several available simulations of real-world robot arms, in this case involving one called Fetch with the goal of grasping a block and placing it in a target location. (f) HandManipulateBlock, another practical simulation of a robotic arm, the Shadow Dexterous Hand.

DeepMind Lab

DeepMind Lab[33] is another RL environment, this time from the developers at Google DeepMind (although they point out that DeepMind Lab is *not* an official Google product). As can be seen in Figure 4.14, the environment is built on top of *id software*'s Quake III Arena[34] and provides a sci-fi inspired three-dimensional world in which agents can explore. The agent experiences the environment from the first-person

perspective, which is distinct from the Atari emulators available via the OpenAI Gym.

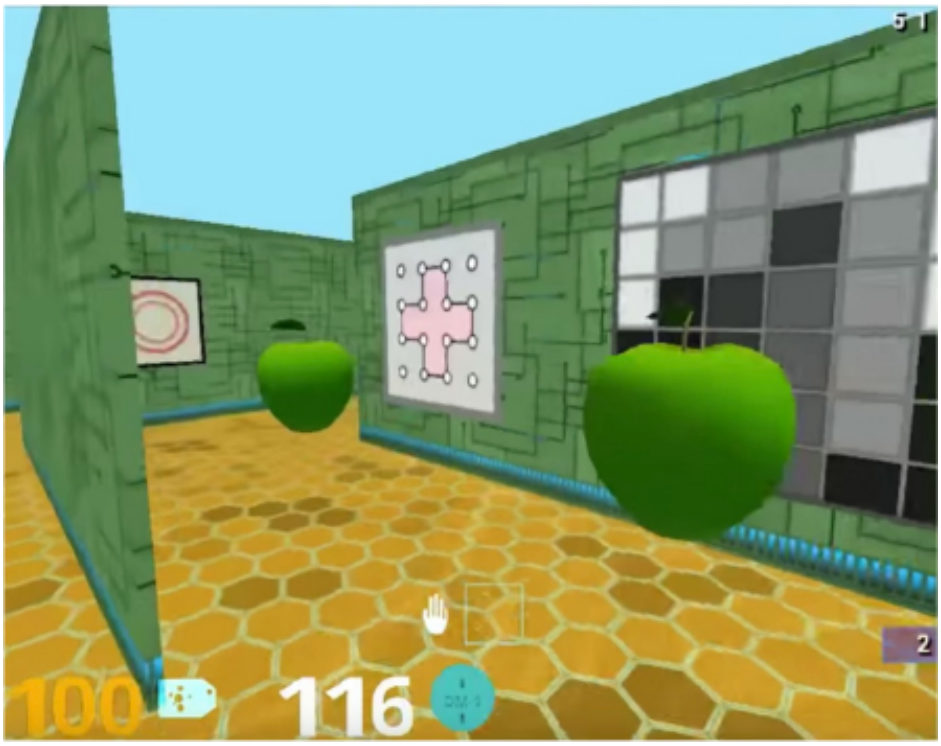

Figure 4.14 A DeepMind Lab environment, in which positive-reward points are awarded for capturing scrumptious green apples.

There are a variety of levels available which can be roughly divided into four categories:

1. Fruit gathering levels, where the agent simply tries to find and collect rewards (apples and melons) while avoiding penalties (lemons).

2. Navigation levels with a static map, where the agent is tasked with finding a goal and remembering the layout of the map. The agent can either be randomly placed within a map at the start of each episode while the goal remains stationary, which tests initial exploration followed by a reliance on memory to repeatedly find the goal; or the agent can start in the same place while the goal is moved for every episode, testing the agents ability to explore.

3. Navigation levels with random maps, where the agent is required to explore a novel map in each episode and find the goal, and then repeatedly return to the goal as many times as possible within a time limit.

4. Laser-tag levels, where the agent is rewarded for hunting and attacking bots in an array of different scenes. The color and texture of the bots are randomized for each episode to prevent the agent from recognizing simple colors too easily.

Installation of DeepMind Lab is not as straightforward as the OpenAI Gym,[35] but it provides a rich, dynamic first-person environements in which to train agents, and the

levels provide complex scenarios involving navigation, memory, strategy, planning and fine-motor skills. These challenging environments push the limits of what is tractable with contemporary deep reinforcement learning.

Unity ML-Agents

Unity is a highly sophisticated engine for two- and three-dimensional video games and digital simulations. Given everything we've learned about reinforcement learning over the course of this chapter, it should come as no surprise that the makers of a popular game engine are also in the business of providing environments to incorporate reinforcement learning into video games. The Unity ML-Agents plugin[36] enables reinforcement learning models to be trained within Unity-based video games or simulations and, perhaps more fitting with the purpose of Unity itself, allows reinforcement learning models to guide the actions of agents within the game. From a distance, it appears inevitable that the development of sophisticated reinforcement learning models to control other characters within video games is going to be a momentous step forward in the gaming experience.

As with DeepMind Lab, installation of Unity ML-Agents is not a one-liner.[37]

THREE CATEGORIES OF AI

Of all deep learning topics, deep reinforcement learning is perhaps the one most closely tied to the popular perception of artificial intelligence as a system for replicating the cognitive, decision-making capacity of humans. In light of that, to wrap up this chapter, in this section we introduce three categories of AI.

Artificial Narrow Intelligence

Artificial narrow intelligence (ANI) is machine expertise at a very specific task. Many diverse examples of ANI exist today, and we've mentioned plenty already, such as the visual recognition of objects, real-time machine translation between natural languages, automated financial-trading systems, AlphaZero and self-driving cars.

Artificial General Intelligence

Artificial general intelligence (AGI) would involve a single algorithm that could perform well at all of the tasks described in the previous paragraph: It would be able to recognize your face, translate this book into another language, optimize your investment portfolio, beat you at Go, and take you safely to your holiday destination. Indeed, this algorithm would be approximately indistinguishable from all of the intellectual capabilities that humans have. There are so many hurdles to overcome in

order for AGI to be realized that it is probably impossible to guess when it will be achieved, if it will be achieved at all. That said, AI experts are happy to wave a finger in the air and speculate on timing. In a study conducted by the philosopher Vincent Müller and the influential futurist Nick Bostrom,[38] the median estimate across hundreds of professional AI researchers was that AGI will be attained in the year 2040.

Artificial Super Intelligence

Artificial Super Intelligence (ASI) is difficult to describe because it's properly mind-boggling. ASI would be an algorithm that is markedly more advanced than the intellectual capabilities of a human.[39] *If* AGI is possible, then ASI may be as well. Of course, there are even more hurdles on the road to ASI than to AGI, presumably most of which we can't foresee clearly today. Citing the Müller and Bostrom (2014) survey again, however, AI experts' median estimate for the arrival of ASI is 2060, a rather hypothetical date that falls within the lifespan of many earthlings alive today. In Chapter 16, at which point you'll be well-versed in deep learning both in theory and in practice, we'll discuss both how deep learning models could contribute to AGI as well as the present limitations associated with deep learning that would need to be bridged in order to attain AGI or, gasp, ASI.

SUMMARY

The chapter began with an overview relating deep learning to the broader field of artificial intelligence. We then detailed deep reinforcement learning, an approach that blends deep learning with the feedback-providing reinforcement learning paradigm. As discussed via real-world examples ranging from the boardgame Go to the grasping of physical objects, such deep reinforcement learning enables machines to process vast amounts of data and take sensible actions on complex tasks, associating it with popular conceptions of AI.

1 . Defining "intelligence" is not straightforward and the great debate on it is beyond the scope of this book. A century-old definition of the term that we find amusing and that still today has some proponents among contemporary experts is that "intelligence is whatever IQ-tests measure." See, e.g.: van der Mass, H., et al. (2014). Intelligence Is What the Intelligence Test Measures. Seriously. *Journal of Intelligence, 2,* 12-15.

2 . E.g.: Kindermans, P.-J., et al. (2018). Learning how to explain neural networks: PatternNet and PatternAttribution. *International Conference on Learning Representations*.

3 . E.g.: He, K., et al. (2016). Identity Mappings in Deep Residual Networks. *arXiv:1603.05027*.

4 . We're not aware of video game-playing algorithms that literally press the buttons on the game console's controllers. They would typically interact with a video game directly via a software-based emulation. We'll go through the most popular open-source packages for doing this at the end of the chapter.

5 . Earlier in this chapter (see Figure 4.2), we indicated that the "deep learning" moniker applies to an artificial neural network that has at least three hidden layers. While in general this is the case, when used by the reinforcement learning community, the term "deep reinforcement learning" may be used even if the artificial neural network involved in the model is shallow, i.e., composed of as few as one or two hidden layers.

6 . Tesauro, G. (1995). Temporal Difference Learning and TD-Gammon. *Communications of the Association for Computing Machinery, 38,* 58-68.

7 . Mnih obtained his doctorate at the University of Toronto under the supervision of Geoff Hinton (Figure 1.17).

8 . Mnih, V. et al. (2013). Playing Atari with Deep Reinforcement Learning. *arXiv: 1312.5602*.

9 . Bellemare, M. et al. (2012). The Arcade Learning Environment: An Evaluation Platform for General Agents. *arXiv: 1207.4708*.

10. Mnih, V., et al. (2015). Human-level control through deep reinforcement learning. *Nature, 518,* 529-33.

11. You can be entertained by watching the Google DeepMind DQN learn to master *Space Invaders* and *Pong* here: youtu.be/iqXKQf2BOSE

12. Indeed, Go in Chinese translates literally to "encirclement board game".

13. IBM's *Deep Blue* defeated Garry Kasparov, arguably the world's greatest-ever chess player, in 1997. More on that storied match coming up shortly in this section.

14. There are an estimated 10^{80} atoms in the observable universe.

15. Hence "Monte Carlo": the casino-dense district of Monaco evokes imagery of random outcomes.

16. Silver, D., et al. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature, 529,* 484-9.

17. Elo ratings enable the skill level of human and artificial game-players alike to be compared. Derived from calculations of head-to-head wins and losses, an individual with a higher Elo score is more likely to win a game against an opponent with a lower score. The larger the score gap between the two players, the greater the probability that the player with the higher score will win.

18. Silver, D., et al. (2016). Mastering the game of Go without human knowledge. *Nature* 550, 354-359.

19. Google built custom processor units for training neural networks, known as tensor processing units (TPUs). They took the existing architecture of a GPU and specifically optimized it for performing tensor calculations (these will become familiar in Chapter 14) during neural network training. At the time of writing, TPUs were available to the public via the Google Cloud Platform.

20. The Elo rating system was created by Arpad Elo as a means to compare the relative skill levels in zero-sum games such as chess. A 200-point margin corresponds to a 75% likelihood of winning a game.

21. AlphaGo Master is a hybrid between AlphaGo Lee and AlphaGo Zero, however it uses the extra input features enjoyed by AlphaGo Lee and initializes training in a supervised manner. AlphaGo Master famously played online anonymously in January 2017 under the pseudonyms *Master* and *Magister*. It won all 60 of the games it played against some of the world's strongest Go players.

22. Silver, D., et al. (2017). Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *arXiv:1712.01815*.

23. It's worth pointing out that Deep Blue lost its first match against Kasparov in 1996, and after significant upgrades went on to narrowly beat Kasparov in 1997. This was not the total domination of man by machine that AI proponents might have hoped for.

24. Deep Blue was the planet's 259th most powerful supercomputer at the time of the

match against Kasparov.

25. This was a point of contention between IBM and Kasparov after his loss in 1997. IBM refused to release the logs initially and dismantled Deep Blue. Their computer system never received an official chess rating because it played so few games against rated chess masters.

26. This makes expanding the training data via synthetic augmentation—an approach used copiously for AlphaGo—more challenging

27. A single value

28. Levine, S., Finn, C., et al. (2016). End-to-End Training of Deep Visuomotor Policies. *Journal of Machine Learning Research*, 17, 1-40.

29. github.com/openai/gym

30. openai.com

31. OpenAI Gym uses the Arcade Learning Environment to emulate Atari 2600 games. This same framework is used in the Mnih et al. (2013) paper described in the *Video Games* section. You can find the framework yourself at https://github.com/mgbellemare/Arcade-Learning-Environment

32. MuJoCo is an abbreviation of Multi-Joint dynamics with Contact. It is a physics engine that was developed by Emo Todorov for Roboti LLC.

33. Beattie, C. et al. (2016). DeepMind Lab. *arXiv:1612.03801*

34. id software. Quake3, 1999. https://github.com/id-Software/Quake-III-Arena.

35. First the Github repository (github.com/deepmind/lab) is cloned, and then the software must be built using Bazel (https://docs.bazel.build/versions/master/install.html). The DeepMind Lab repository provides detailed instructions (https://github.com/deepmind/lab/blob/master/docs/users/build.md)

36. github.com/Unity-Technologies/ml-agents

37. It requires the user to first install Unity (for download and installation instructions, see https://store.unity.com/download) and then clone the Github repository. Full instructions are available at the Unity ML-Agents Github repository

(https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Installation.md)

38. Müller, V. and Bostrom, N. (2014). Future Progress in Artificial Intelligence: A Survey of Expert Opinion. In V. Müller (Ed.), *Fundamental Issues of Artificial Intelligence*. Berlin: Springer.

39. In 2015, Tim Urban provided a lengthy two-part series of posts that entertainingly covers ASI and the related literature. It is available here (to bitly): waitbutwhy.com/2015/01/artificial-intelligence-revolution-1.html

Part II. Essential Theory Illustrated

5 The (Code) Cart Ahead of the (Theory) Horse

In Part I, we provided a high-level overview of deep learning by demonstrating its use across a spectrum of cutting-edge applications. Along the way, we sprinkled in foundational deep learning concepts from its hierarchical, representation-learning nature through to its relationship to the field of artificial intelligence. Repeatedly, as we touched on a concept, we noted that in the second part of the book we would dive into the low-level theory and mathematics behind it. While we promise this *is* true, we are going to take this final opportunity to put the fun, hands-on coding cart ahead of the proverbial—in this case, theory-laden—horse.

In this chapter we will do a line-by-line walk through of a notebook of code featuring a deep learning model. While you will need to bear with us because we have not yet detailed much of the theory underpinning the code, this serpentine approach will make the apprehension of theory in the subsequent chapters easier: Instead of being an abstract idea, each element of theory we introduce in this part of the book will be rooted by a tangible line of applied code.

PREREQUISITES

Working through the examples in this book will be easiest if you are familiar with the basics of the Unix command line. These are provided by Zed Shaw in Appendix A of his deceptively enjoyable *Learn Python the Hard Way*. [1]

Speaking of Python, since it is comfortably the most popular software language in the data science community (at time of writing, anyway), it's the language we selected for our example code throughout the book. Python's prevalence extends both across the composition of standalone scripts through to the deployment of machine-learning models into production systems. If you're new to Python or you're feeling a tad rusty, Shaw's book serves as an appropriate general reference while Daniel Chen's *Pandas for Everyone* [2] is appropriate for applying the language to modeling data in particular.

INSTALLATION

Regardless of whether you're planning on executing our code notebooks via Unix,

Linux, Mac OS or Windows, we have made step-by-step installation instructions available in the GitHub repository that accompanies this book:

github.com/illustrated-series/deep-learning-illustrated. If you'd prefer to view the completed notebooks instead of running them on your own machine, you are more than welcome to do that from the GitHub repo as well.

We elected to provide our code within the comfort of interactive Jupyter notebooks. [3] Jupyter is a common option today for writing and sharing scripts, particularly during exploratory phases in which a data scientist is experimenting with preprocessing, visualizing and modeling her data. Our installation instructions suggest running Jupyter from within a Docker container. [4] This containerization ensures that you'll have all of the software dependencies you need to running our notebooks while simultaneously preventing these dependencies from clashing with software you already have installed on your system.

A SHALLOW NETWORK IN KERAS

To kick off the code portion of our book, we will:

œ Detail a revered data set of handwritten digits,

œ Load these data into a Jupyter notebook,

œ Use Python to prepare the data for modeling, and

œ Write a few lines of code in the high-level API Keras to construct an artificial neural network (in TensorFlow, behind the scenes) that predicts what digit a given handwritten sample represents.

The MNIST Handwritten Digits

Back in Chapter 1 when we introduced the LeNet-5 machine-vision architecture (Figure 1.12), we mentioned that one of the advantages Yann LeCun (Figure 1.10) and his colleagues had over previous deep-learning practitioners was a superior data set for training their model. This data set of handwritten digits, called *MNIST* (see the samples in Figure 5.1), came up again in the context of being imitated by Ian Goodfellow's generative adversarial network (Figure 3.2a). The MNIST data set is ubiquitous across deep-learning tutorials, and for good reason. By modern standards, the data set is small enough that it can be modeled rapidly, even on a laptop computer processor. In addition to their portable size, the MNIST digits are also handy because they occupy a sweet spot with respect to how challenging they are to classify: The handwriting

samples are sufficiently diverse and contain complex enough details that they are not *easy* for a machine-learning algorithm to identify with high accuracy, and yet by no means do they pose an insurmountable problem. However, as we shall observe ourselves as this part of the book develops, a well-designed deep-learning model can near-faultlessly classify the handwriting as the appropriate digit.

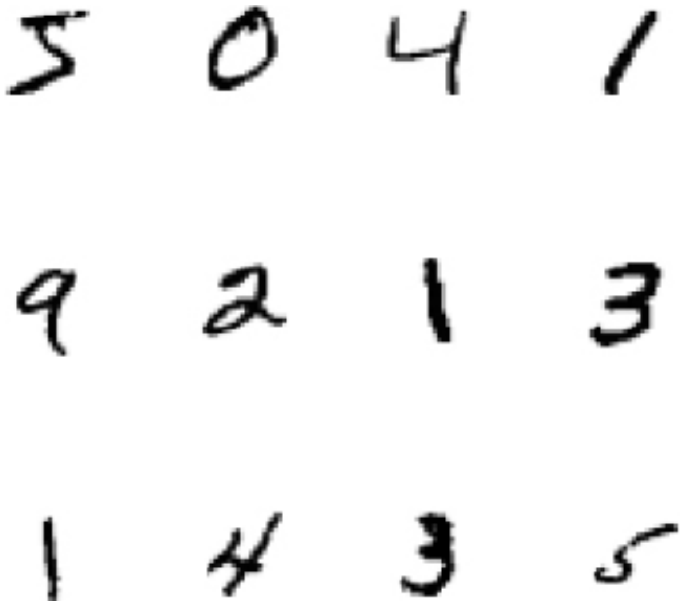

Figure 5.1 A sample of a dozen images from the MNIST dataset. Each image contains a single digit handwritten by either a high-school student or a U.S. census worker.

The MNIST data set was curated by LeCun, Corinna Cortes (Figure 5.2) and the Microsoft-AI-researcher-turned-musician Chris Burges in the 1990s. [5] It consists of sixty thousand handwritten digits for training an algorithm and ten thousand more for validating the algorithm's performance on previously unseen data. The data are a subset (nay, a *modification*) of a larger body of handwriting samples collected from high school students and census workers by the United States' National Institute of Standards and Technology (*NIST*).

Figure 5.2 The Danish computer scientist Corinna Cortes is Head of Research at Google's New York office. Among her countless contributions to both pure and applied machine learning, Cortes (with Chris Burges and Yann LeCun) curated the ubiquitous MNIST dataset.

As exemplified by Figure 5.3, every MNIST digit is a 28-by-28 pixel image. [6] We quickly became bored of drawing individual pixels so only depicted them individually in the top-left corner of the figure, but of course the entirety of the handwritten digit (in this example, the number *two*) is represented by pixels. Each pixel is 8-bit, meaning that the pixel darkness can vary from zero (white) to 255 (black), with the intervening range of integers representing gradually darker shades of gray.

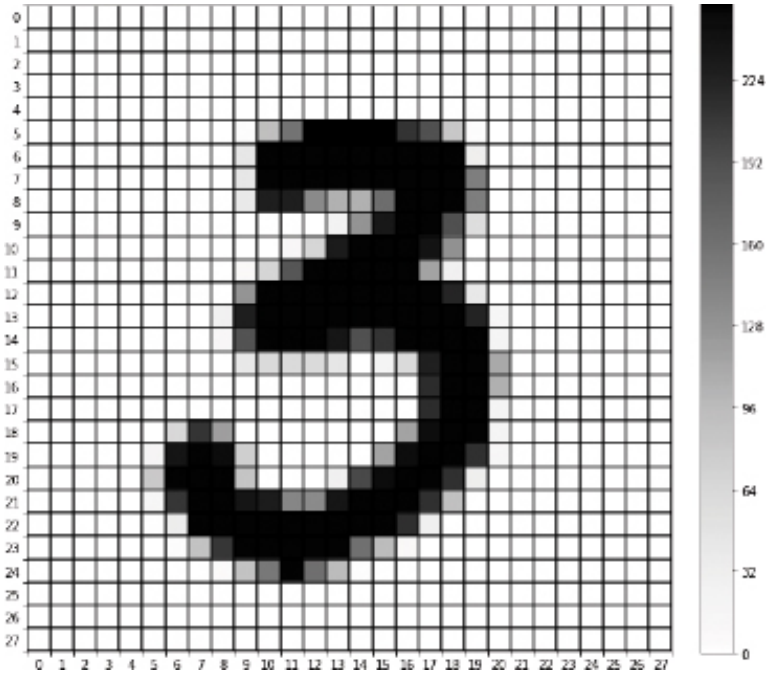

Figure 5.3 Each handwritten MNIST digit is stored as a 28-by-28-pixel grayscale image. See the Jupyter notebook titled MNIST digit pixel by pixel that accompanies this book for the code we used to create this figure.

A Schematic Diagram of the Network

In our *Shallow Net in Keras* Jupyter notebook [7] , we create an artificial neural network to predict what digit a given handwritten MNIST image represents. As shown in the rough schematic diagram in Figure 5.4, this artificial neural network features one hidden layer of artificial neurons for a total of three layers. Recalling Figure 4.2, with so few layers this ANN would not generally be considered a *deep* learning architecture; hence it is *shallow*.

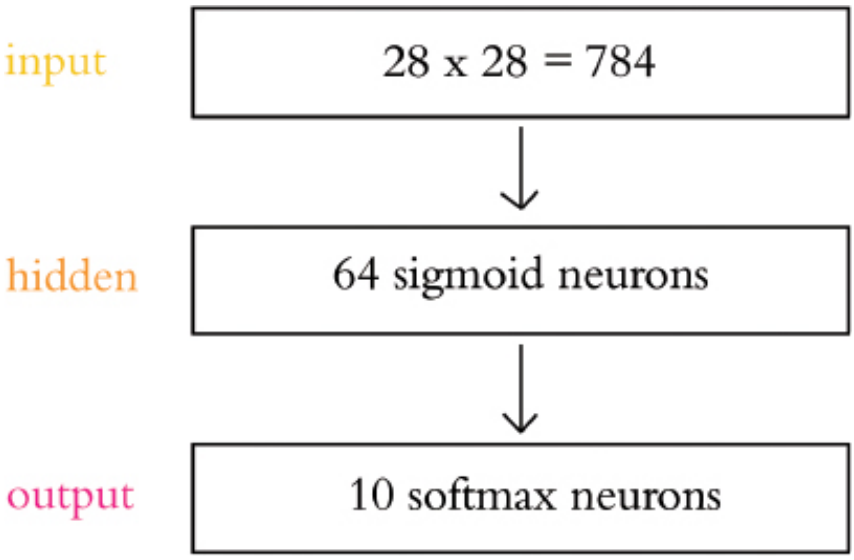

Figure 5.4 A rough schematic of the shallow artificial-neural-network architecture we're whipping up in this chapter. We'll cover the particular sigmoid and softmax flavors of artificial neurons in Chapter 6.

The first layer of the network is reserved for inputting our MNIST digits. As they are

28-by-28 pixel images, each one has a total of 784 values. After we load in the images, we'll flatten them from their native, two-dimensional 28-by-28 shape to a one-dimensional array of 784 elements.

This paragraph is a Trilobite-attention SIDEBAR. You could argue that collapsing the images from two dimensions to one will cause us to lose a lot of the meaningful structure of the handwritten digits. Well, if you argued that, you'd be right! Working with one-dimensional data, however, means we can use less sophisticated deep learning models, which is appropriate at this early stage in our journey. Later, in Chapter 10, we'll be in a position to appreciate more complex models that can handle multi-dimensional inputs. END SIDEBAR.

The pixel-data inputs will be passed through a single, hidden layer of 64 artificial neurons. [8] The number (64) and type (*sigmoid*) of these neurons aren't critical details at present; we'll begin to explain these model attributes in the next chapter. The key piece of information at this time is that, like we demonstrated in Chapter 1 (see Figures 1.19 and 1.20), the neurons in the hidden layer are responsible for learning representations of the input data so that the network can predict what digit a given image represents.

Finally, the information output by the hidden layer will be passed to ten neurons in the output layer. Again, we'll detail how *softmax* neurons work in the next chapter but, in essence, we have ten neurons because we have ten types of digit to classify. These ten neurons each output a probability: one for each of the ten possible digits that a given MNIST image could represent. As an example, a fairly-well-trained network which is fed the image in Figure 5.3 might output that there is a 0.92 probability that the image is of a *two*, a 0.06 probability that it's a *three*, a 0.02 probability that it's an *eight*, and a zero probability for the other seven digits.

Loading the Data

At the top of the notebook we import our software dependencies, which is unexciting but necessary:

Example 5.1 Software dependencies for shallow net in Keras

```
import keras

from keras.datasets import mnist

from keras.models import Sequential
```

```
from keras.layers import Dense

from keras.optimizers import SGD

from matplotlib import pyplot as plt
```

We import Keras because that's the library we're using to fashion our neural network. We also import the MNIST dataset because these, of course, are the data we're working with in this example. The lines ending in `Sequential`, `Dense` and `SGD` will make sense later; no need to worry about them at this stage. Finally, the `matplotlib` line will enable us to plot MNIST digits out to our screen.

With these dependencies imported, we can conveniently load the MNIST data in a single line of code:

```
(X_train, y_train), (X_valid, y_valid) = mnist.load_data()
```

Let's examine these data. As mentioned in Chapter 4, the mathematical notation *x* is used to represent the data we're feeding into a model as input while *y* is used for the labelled output that we're training the model to predict. With this in mind, `X_train` stores the MNIST digits we'll be training our model on. [9] Executing `X_train.shape` yields the output `(60000, 28, 28)`. This shows us that, as expected, we have 60,000 images in our training data set, each of which is a 28-by-28 matrix of values. Running y_train.shape, we unsurprisingly discover we have 60,000 labels indicating what digit each of the 60,000 training images contains. y_train[0:12] outputs an array of twelve integers representing the first dozen labels, so we can see that the first handwritten digit in the training set (X_train[0]) is the number *five*, the second is a *zero*, the third is a *four*, and so on:

```
array([5, 0, 4, 1, 9, 2, 1, 3, 1, 4, 3, 5], dtype=uint8)
```

These happen to be the same dozen MNIST digits that were shown above in Figure 5.1, a figure we created by running the following chunk of code:

```
plt.figure(figsize=(5,5))

for k in range(12):

    plt.subplot(3, 4, k+1)

    plt.imshow(X_train[k], cmap='Greys')

    plt.axis('off')

plt.tight_layout()

plt.show()
```

Akin to the training data, by examining the shape of the validation data (`X_valid.shape, y_valid.shape`), we note that there are the expected ten thousand 28-by-28 pixel validation images, each with a corresponding label: (`10000, 28, 28`), (`10000`,). Investigating the values that make up an individual image like `X_valid[0]`, we observe that the matrix of integers representing the handwriting is primarily zeros (whitespace). Tilting our head, you might even be able to make out that the digit in this example is a *seven* with the highest integers (e.g., `254, 255`) representing the black core of the handwritten figure and the outline of the figure (composed of intermediate integers) fading toward white. To corroborate that this is indeed the number seven, we both printed out the image with `plt.imshow(X_valid[0], cmap='Greys')` (output shown in Figure 5.5) and printed out its label with `y_valid[0]` (output was 7).

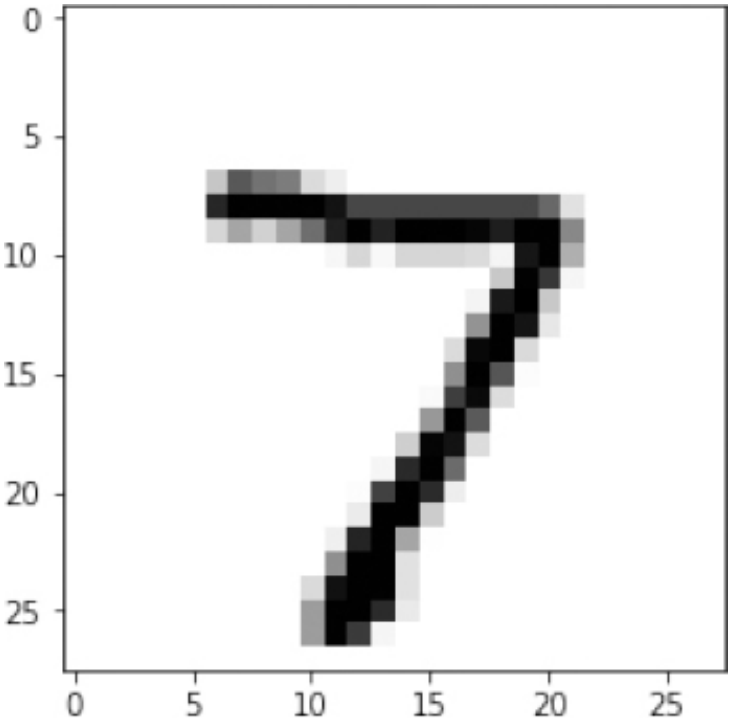

Figure 5.5 The first MNIST digit in the validation data set (X_valid[0]) is a seven.

Reformatting the Data

The MNIST data now loaded, we come across the heading *Preprocess data* in the notebook. We won't, however, be preprocessing the images by applying functions to, say, extract features that provide hints to our artificial neural network. We will simply be rearranging the *shape* of the data so that they match up with the shapes of the input and output layers of the network.

Thus, we'll flatten our 28-by-28 pixel images into 784-element-long arrays. We employed the `reshape()` method to do this:

```
X_train = X_train.reshape(60000, 784).astype('float32')

X_valid = X_valid.reshape(10000, 784).astype('float32')
```

Simultaneously, we used astype(`'float32'`) to convert the pixel darknesses from integers into single-precision float values.[10] This conversion was preparation for the subsequent step, in which we divided all of the values by 255 so that they range from zero to one:[11]

```
X_train /= 255
```

```
X_valid /= 255
```

Revisiting our example handwritten *seven* from Figure 5.5 by running `X_valid[0]`, we can verify that it is now represented by a one-dimensional array made up of float values as low as zero and as high as one.

That's all for reformatting our model inputs *X*. For the labels *y*, we need to convert them from integers into *one-hot encodings*; we'll demonstrate what these are via application:

```
n_classes = 10

y_train = keras.utils.to_categorical(y_train, n_classes)

y_valid = keras.utils.to_categorical(y_valid, n_classes)
```

There are ten possible handwritten digits, so we set n_classes equal to 10. In the other two lines of code we use a convenient utility function to_categorical, which is provided within the Keras library, to transform both the training and validation labels from integers into the one-hot format. Execute y_valid to see how the label *seven* is represented now:

```
array([0., 0., 0., 0., 0., 0., 0., 1., 0., 0.], dtype=float32
```

Instead of using an integer to represent *seven*, we have an array of length ten consisting entirely of zeroes, with the exception of a 1 in the eighth position. In such a one-hot encoding, the label *zero* would be represented by a lone 1 in the first position, *one* by a lone 1 in the second position, and so on. We arrange the labels with such one-hot encodings so that they line up with the ten probabilities being output by the final layer of our artificial neural network. They represent the ideal output that we are striving to attain with our network: If the input image is a handwritten seven then a perfectly-trained network would output a probability of 1.00 that it is a seven and a probability of 0.00 for each of the other nine classes of digits.

Designing a Neural Network Architecture

From your authors' perspective, this is the most pleasurable bit of any script featuring

deep learning code: architecting the artificial neural net itself. There are infinite possibilities here and, as we progress through the book, you will begin to develop an intuition that guides the selection of the architectures you experiment with for a given problem. Referring back to Figure 5.4, for the time being, we're keeping the architecture as elementary as possible in Example 5.2:

Example 5.2 Keras code to architect a shallow neural network

```
model = Sequential()

model.add(Dense(64, activation='sigmoid', input_shape=(784,))

model.add(Dense(10, activation='softmax'))
```

In the first line of code, we instantiate the simplest type of neural-network model object, the `sequential` type[12] and—in a dash of extreme creativity—name the model `model`. In the second line, we use the `add()` method of our `model` object to specify the attributes of our network's hidden layer (`64` sigmoid-type artificial neurons in the general-purpose, fully-connected arrangement defined by the `Dense()` method)[13] as well as the shape of our input layer (one-dimensional array of length `784`). In the third and final line we use the `add()` method again to specify the output layer and its parameters: ten artificial neurons of the `softmax` variety, corresponding to the ten probabilities (one for each of the ten possible digits) that the network will output when fed a given handwritten image.

Training a Deep Learning Model

Later, we'll return to the `model.summary()` and `model.compile()` steps of the *Shallow Net in Keras* notebook, as well as its three lines of arithmetic. For now, skipping ahead to the model-fitting step:

Example 5.3 Keras code to train our shallow neural network

```
model.fit(X_train, y_train,

                batch_size=128, epochs=200,

                verbose=1,
```

```
                validation_data=(X_valid, y_valid))
```

The critical aspects are that:

1. The `fit()` method of our `model` object enables us to train our artificial neural network with the training images `X_train` as inputs and their associated labels `y_train` as the desired outputs.

2. As the network trains, the `fit()` method also provides us with the option to evaluate the performance of our network by passing our validation data `X_valid` and `y_valid` into the `validation_data` parameter.

3. With machine learning, and especially with deep learning, it is commonplace to train our model on the same data multiple times. One pass through all of our training data (60,000 images in the current case) is called one *epoch* of training. By setting the `epochs` parameter to `200`, we cycle through all 60,000 training images two hundred separate times.

4. By setting `verbose` to `1`, `model.fit()` will provide us with plenty of feedback as we train. At the moment, we'll focus on the `val_acc` statistic output following each epoch of training. *Validation accuracy* is the proportion of the 10,000 handwritten images in `X_valid` where the network's highest probability in the output layer corresponds to the correct digit as per the labels in `y_valid`.

Following the first epoch of training, we observe `val_acc: 0.1010`.[14,15] That is, 10.1% of the images from the held-out validation dataset were correctly classified by our shallow architecture. Given that there are ten classes of handwritten digits, we'd expect a random process to guess ten percent of the digits correctly by chance, so this is not an impressive result. As the network continues to train, however, the results improve. After ten epochs of training, it is correctly classifying 36.5% of the validation images—far better than would be expected by chance! And this is only the beginning: After 200 epochs, the network's improvements appears to be plateauing as it approaches 86% validation accuracy.

Since we constructed such an uninvolved, shallow neural-network architecture, this is not too shabby!

SUMMARY

Putting the cart before the horse, in this chapter we coded up a shallow, elementary artificial neural network. With decent accuracy, it is able to classify the MNIST images. Over the coming chapters, as we dive into theory, unearth artificial neural network best-practices, and layer up to authentic deep learning architectures, we should surely be able to do much better, no? Let's see...

1. Shaw, Z. (2013). *Learn Python the Hard Way, 3rd Ed.* New York, NY: Addison-Wesley. This relevant appendix, Shaw's *Command Line Crash Course*, is available online at learnpythonthehardway.org/book/appendixa.html

2. Chen. D. (2017). *Pandas for Everyone: Python Data Analysis.* New York, NY: Addison-Wesley.

3. jupyter.org; we recommend familiarizing yourself with the hot keys to breeze through Jupyter notebooks with pizzazz.

4. docker.com

5. yann.lecun.com/exdb/mnist/

6. Python uses zero-indexing so the first row and column are denoted with a *zero*. The 28th row and 28th column of pixels are therefore both denoted with 27.

7. Within this book's GitHub repository, navigate into the *notebooks* directory.

8. "Hidden" layers are so called because they are not exposed; data impact them only indirectly, via the input layer or the output layer of neurons.

9. The convention is to use an upper-case letter like X when the variable being represented is a two-dimensional matrix or a data structure with even higher dimensionality. In contrast, a lower-case letter like x is used to represent a single value (a scalar) or a one-dimensional array.

10. The data are initially stored as `uint8`, which is an unsigned integer from 0 to 255. This is more memory efficient, but of course it doesn't encapsulate much precision since there are only 256 possible values. Without specifying, Python would default to a

64-bit float which is overkill. Thus, by specifying a 32-bit float we can be sure of what we're feeding to the network.

11. Machine learning models tend to learn more efficiently when fed standardized inputs. Binary inputs would typically be a `0` or a `1`, while distributions are often normalized to have a mean of zero and a standard deviation of one. As we've done here, pixel intensities are generally scaled to range from zero to one.

12. So named because each layer in the network passes information to only the next layer in the *sequence* of layers

13. Once more, these now-esoteric terms will become comprehensible over the coming chapters.

14. Artificial neural networks are *stochastic* (due to the way they're initialized as well as the way they learn) so your results will vary slightly from ours. Indeed, if you re-run the whole notebook (e.g., by clicking on the *Kernel* option in the Jupyter menu bar and selecting *Restart & Run All*), you should obtain new, slightly different results yourself.

15. By the end of Chapter 8, we'll have enough theory under our belts to study the output `model.fit()` in all its glory. For our immediate "cart before the horse" purposes, coverage of the *validation accuracy* metric alone suffices.

6 Artificial Neurons Detecting Hot Dogs

Having received tantalizing exposure to applications of deep learning in the first part of this book and having coded up a functioning neural network in the preceding chapter, the moment has come to delve into the nitty-gritty theory underlying these capabilities. We will begin by dissecting artificial neurons, the units that—when wired together—constitute an artificial neural network.

BIOLOGICAL NEUROANATOMY 101

As presented in the opening paragraphs of this book, ersatz neurons are inspired by biological ones. Given that, let's take a gander at Figure 6.1 for a précis of the first lecture in any neuroanatomy course: A given biological neuron receives input into its *cell body* from many (generally thousands) of *dendrites*, with each dendrite receiving signals of information from another neuron in the nervous system—a biological neural network. When the signal conveyed along a dendrite reaches the cell body, it causes a small change in the voltage of the cell body. [1] Some dendrites cause a small positive change in voltage, while the others cause a small negative change. If the cumulative effect of these changes causes the voltage to increase from its resting state of -70 millivolts to the critical threshold of -55 millivolts, the neuron will fire something called an *action potential* away from its cell body, down its axon, which transmits a signal to other neurons in the network.

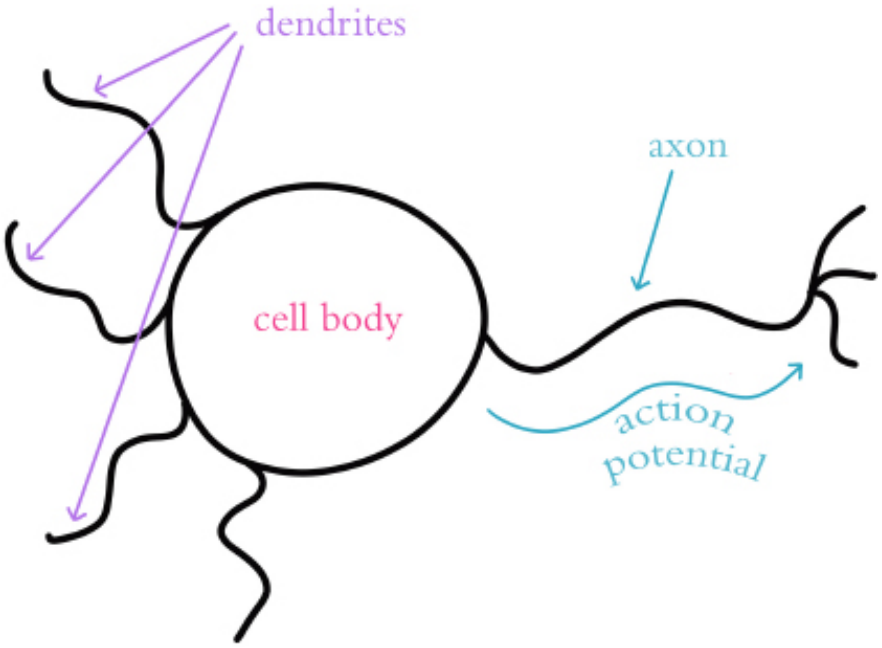

Figure 6.1 Cartoon of the anatomy of a biological neuron.

To summarize, biological neurons typically:

1. receive information from many other neurons,

2. aggregate this information via changes in cell voltage at the cell body, and

3. transmit a signal if the cell voltage crosses a threshold level, which can be received by many other neurons in the network.

TRILOBITE SIDEBAR: An observant reader might have noticed the use of matching colors in the text and figures here. This is intentional, and it's a tool we'll use more often in the coming chapters as we begin to discuss a few important equations and the variables they contain—so keep your eye out for it!

THE PERCEPTRON

In the late 1950s, the American neurobiologist Frank Rosenblatt (Figure 6.2) published on his *perceptron*, an algorithm influenced by his understanding of biological neurons, making it the earliest formulation of an artificial neuron. [2] Analogous to their living inspiration, the perceptron (Figure 6.3) can:

1. receive input from multiple other neurons,

2. aggregate those inputs via a simple arithmetic operation called the *weighted sum*, and

3. generate an output if this weighted sum crosses a threshold level, which can then be

sent on to many other neurons within a network.

Figure 6.2 The American neurobiology and behavior researcher Frank Rosenblatt. He conducted much of his work out of the Cornell Aeronautical Laboratory, including physically constructing his Mark I Perceptron there. This machine, an early relic of artificial intelligence, can today be viewed at the Smithsonian Institution in Washington, D.C.

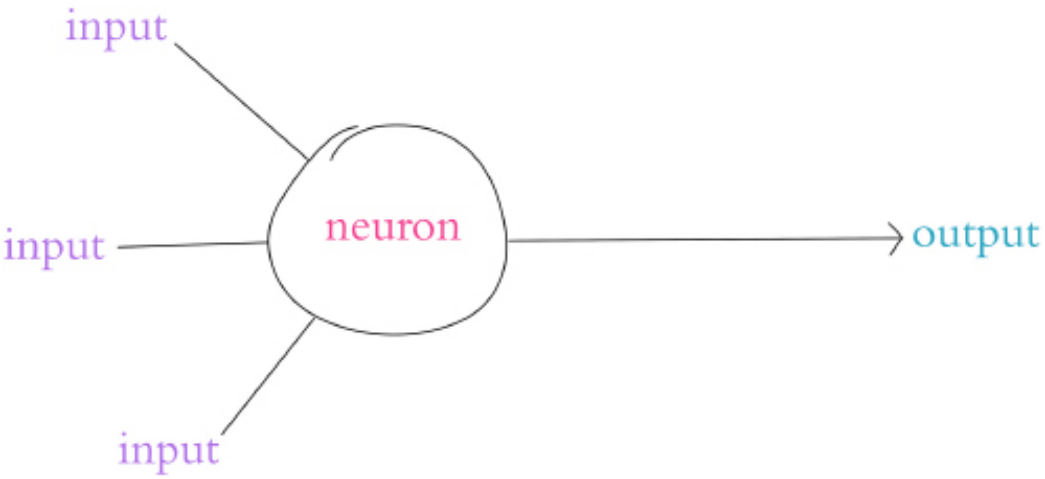

Figure 6.3 Schematic diagram of a perceptron, an early artificial neuron. Note the structural similarity to the biological neuron in Figure 6.1.

The Hot Dog / Not Hot Dog Detector

Let's work through a light-hearted example to understand how the perceptron algorithm works. We're going to look at a perceptron that is specialized in distinguishing whether a given object is a hot dog or, well... not a hot dog.

A critical attribute of perceptrons is that they can only be fed binary information as inputs, and their output is restricted to being binary as well. Thus, our hot dog-detecting perceptron must be fed its particular three inputs (indicating whether the object involves ketchup, mustard, or a bun, respectively) as either a 0 or a 1. In Figure 6.4:

- The first input (a purple 1) indicates the object being presented to the perceptron involves ketchup.
- The second input (also a purple 1) indicates the object has mustard.
- The third input (a purple 0) indicates the object does *not* include a bun.

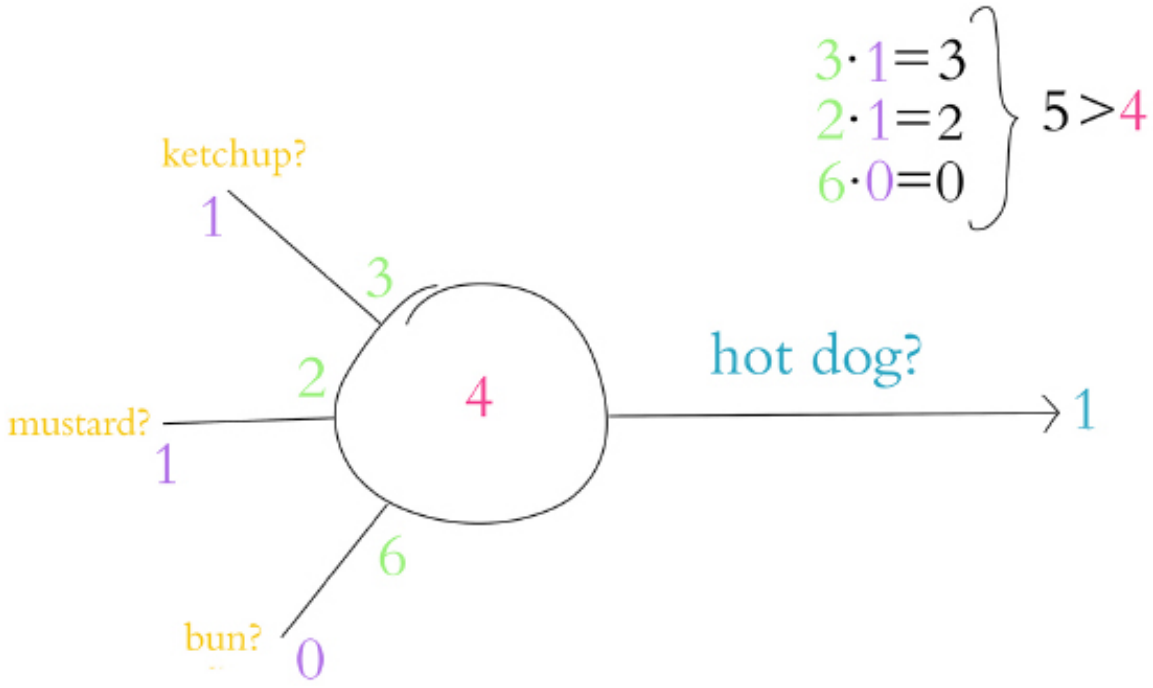

Figure 6.4 Example 1 of a hot dog-detecting perceptron: In this instance, it predicts there is indeed a hot dog.

To make a prediction as to whether the object is a hot dog or not, the perceptron independently *weights* each of these three inputs. [3] The weights that we arbitrarily selected in this (entirely contrived) hot dog example indicate that the presence of a bun, with its weight of six, is the most influential predictor of whether the object is a hot dog or not. The intermediate predictor is ketchup with its weight of three, and the least influential predictor is mustard with a weight of two.

Let's determine the weighted sum of the inputs: One input at a time (i.e., *element-wise*), we multiply the input by its weight, and then sum the individual results. So, first let's calculate the weighted inputs:

1. for the *ketchup* input: $3 \times 1 = 3$

2. for *mustard*: 2 × 2 = 2

3. and for *bun*: 6 × 0 = 0

With those three products, we can compute that the weighted sum of the inputs is five: 3 + 2 + 0 = 5. To generalize from this example, the calculation of the weighted sum of inputs is captured by Equation 6.1:

$$\sum_{i=1}^{n} w_i \cdot x_i \qquad (6.1)$$

Where:

- w_i is the weight of a given input i (in our example, $w_1 = 3$, $w_2 = 2$ and $w_3 = 6$)
- x_i is the value of a given input i (in our example, $x_1 = 1$, $x_2 = 1$ and $x_3 = 0$)
- $w_i \cdot x_i$ represents the product of w_i and x_i—i.e., the weighted value of a given input i
- $\sum_{i=1}^{n}$ indicates that we sum all of the individual weighted inputs $w_i \cdot x_i$, where n is the total number of inputs (in our example, we had three inputs but artificial neurons can have any number of inputs).

The final step of the perceptron algorithm is to evaluate whether the weighted sum of the inputs is greater than the neuron's *threshold*. As with the weights above, we have again arbitrarily chosen a threshold value for our perceptron example: *four* (shown in red in the center of the neuron in Figure 6.4). The perceptron algorithm is shown below :

$$\sum_{i=1}^{n} w_i x_i \begin{cases} \geqslant \textit{threshold}, \text{output } 1 \\ < \textit{threshold}, \text{output } 0 \end{cases} \qquad (6.2)$$

Where:

- If the weighted sum of a perceptron's inputs is greater than its threshold, then it outputs a `1`, indicating that the perceptron predicts the object is a hot dog.
- Otherwise, if the weighted sum is less than or equal to the threshold, the perceptron outputs a `0`, indicating that it predicts there is *not* a hot dog.

Knowing this, we can wrap up our example from Figure 6.4: The weighted sum of five is greater than the neuron's threshold of four, and so our hot dog-detecting perceptron

outputs a 1.

Riffing on our first hot dog example, in Figure 6.5 the object evaluated by the perceptron now includes mustard only—there is no ketchup and it is still without a bun. In this case the weighted sum of inputs comes out to 2. Because 2 is less than the perceptron's threshold, the neuron outputs 0, indicating that it predicts this object is *not* a hot dog.

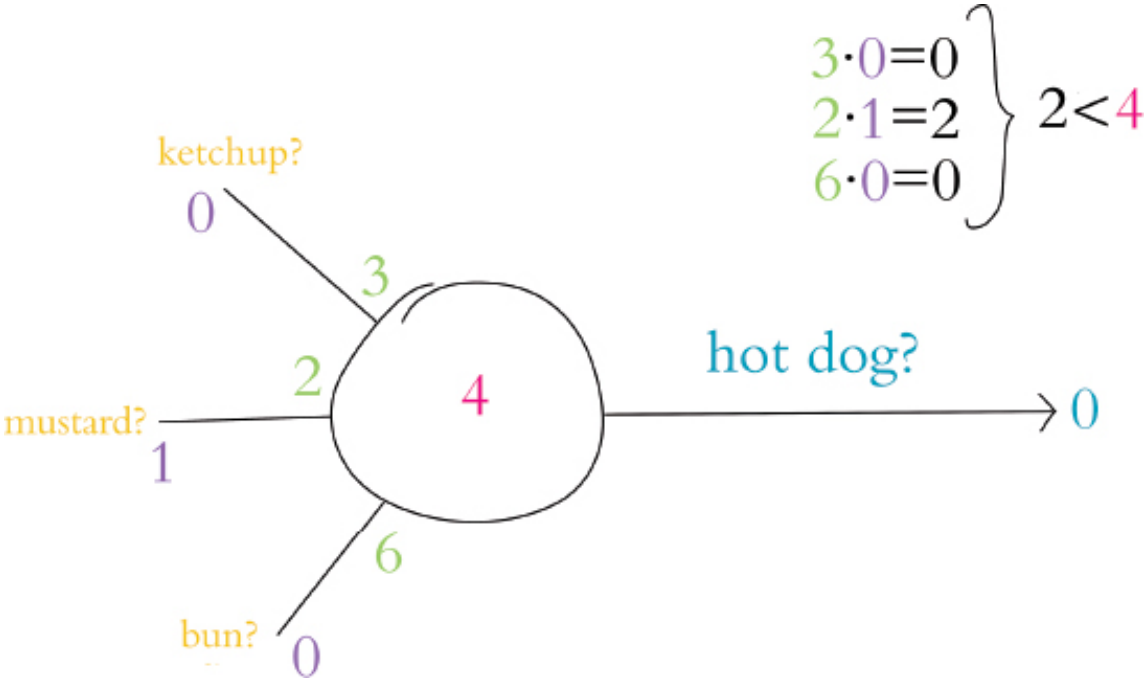

Figure 6.5 Example 2 of a hot dog-detecting perceptron: In this instance, it predicts there is not a hot dog.

In our third and final perceptron example, shown in Figure 6.6, the artificial neuron evaluates an object that involves neither mustard nor ketchup, but *is* on a bun. The presence of a bun alone corresponds to the calculation of a weighted sum of 6. Since 6 is greater than the perceptron's threshold, the algorithm predicts the object is a hot dog and outputs a 1.

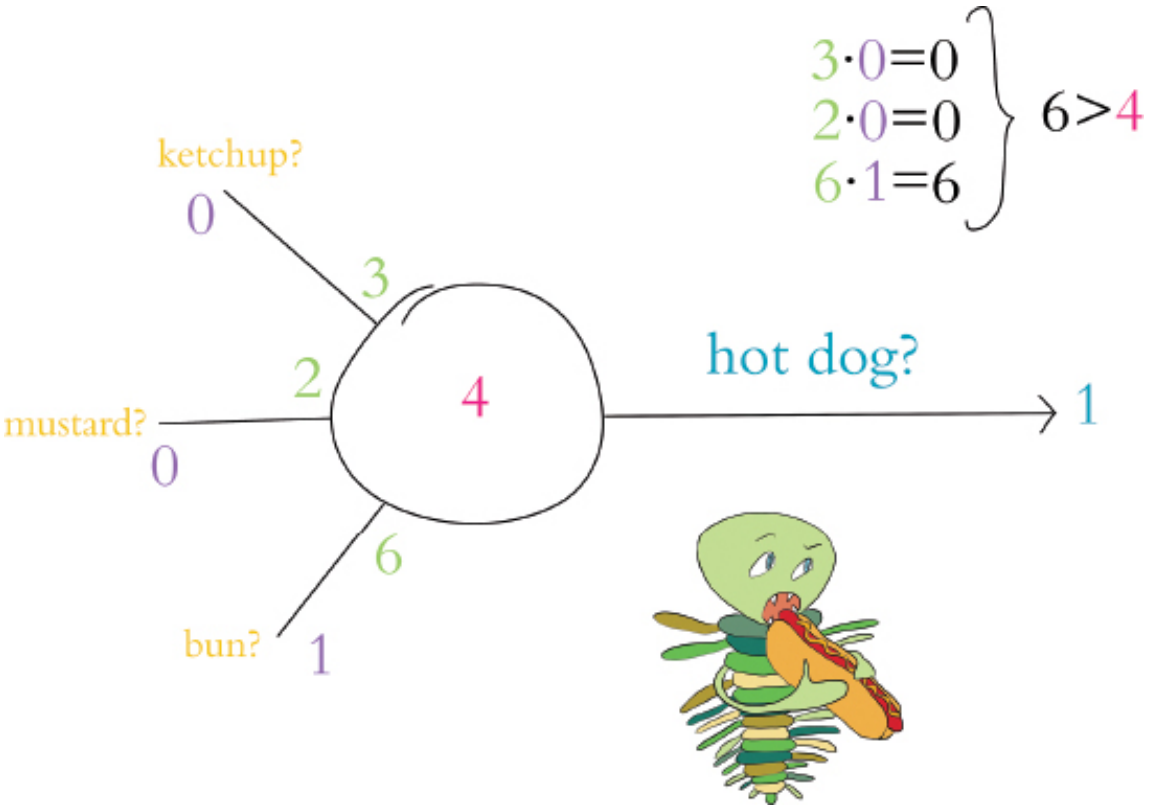

Figure 6.6 Example 3 of a hot dog-detecting perceptron: In this instance, it again predicts the object presented to it is a hot dog.

The Most Important Equation in this Book

To achieve the formulation of a simplified and universal perceptron equation, we must introduce a term called the *bias*, which we annotate as b and which is equivalent to the negative of an artificial neuron's threshold value (Equation 6.3):

$$b \equiv -threshold \tag{6.3}$$

Together, a neuron's bias and its weights constitute all of its *parameters*—the changeable variables that prescribe what the neuron will output in response to its inputs.

With the concept of a neuron's bias now available to us, we arrive at the most widely-used perceptron equation (Equation 6.4):

$$\text{output} \begin{cases} 1 \text{ if } w \cdot x + b > 0 \\ 0 \text{ otherwise} \end{cases} \tag{6.4}$$

Notice that we made the following five updates to our initial perceptron equation (from Equation 6.2):

1. substituted the bias b in place of the neuron's *threshold*

2. flipped b onto the same side of the equation as all of the other variables

3. used the array w to represent all of the w_i weights from w_1 through to w_n

4. likewise, used the array x to represent all of the x_i values from x_1 through to x_n

5. used the *dot product* notation $w\ x$ to abbreviate the representation of the weighted sum of neuron inputs (the longer form of this was already shown in Equation 6.1: $\sum_{i=1}^{n} w_i x_i$)

Right at the heart of the perceptron equation in Equation 6.4 is $w \cdot x + b$, which we have cut out for emphasis and placed alone in Figure 6.7. *If there is one item you note down to remember from this chapter, it should be this three-variable formula, which is an equation that represents artificial neurons in general.* We will refer back to this equation many times over the course of this book, especially over the remainder of Part II and in Chapter 14 when we move beyond the Keras API to create our own artificial neurons from scratch in TensorFlow proper.

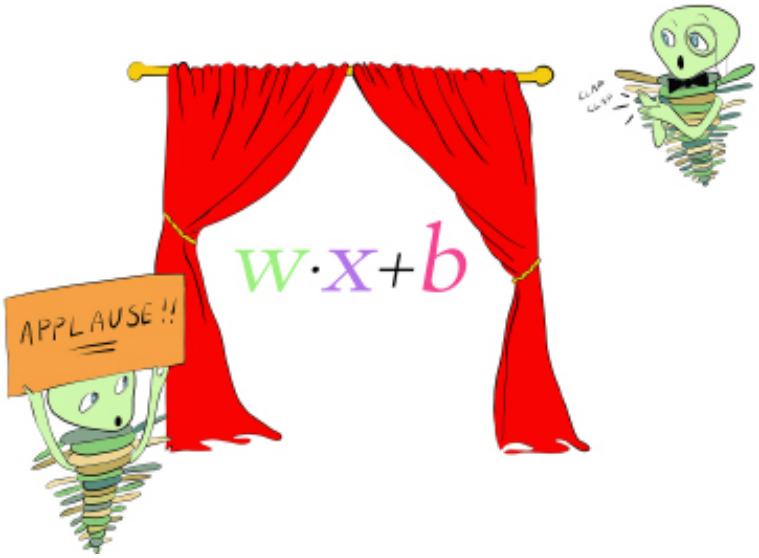

Figure 6.7 The general equation for artificial neurons that we will return to time and again. It is the most important equation in this book.

This paragraph is a Trilobite-attention SIDEBAR. To keep the arithmetic as undemanding as possible in our hot dog-detecting perceptron examples, all of the parameter values we made up—the perceptron's weights as well as its bias—were positive integers. These parameters could, however, be negative values and, in practice, they would rarely be integers—instead, parameters are configured as float values, which are less clunky.

This paragraph is also in the same Trilobite-attention SIDEBAR. Finally, while all of the parameters in these examples were fabricated by us, they would usually be learned through the training of artificial neurons on data. In Chapter 8, we'll cover how this training is accomplished in practice. END SIDEBAR.

MODERN NEURONS AND ACTIVATION FUNCTIONS

Modern artificial neurons, such as those in the hidden layer of the shallow architecture we built in the previous chapter (look back to Figure 5.4 or to our *Shallow Net in Keras* notebook), are not perceptrons. While the perceptron provides a relatively uncomplicated introduction to artificial neurons, it is not used in practice today. The most obvious restriction of the perceptron is that operates solely with binary values: it receives only binary inputs and provides only a binary output. In many cases, we'd like to make predictions from inputs that are continuous variables not binary integers, and so this restriction alone would make perceptrons unsuitable.

A less obvious (yet even more critical) corollary of the perceptron's binary-only restriction is that it makes learning rather challenging. Consider Figure 6.8, in which we use a new term, z, as short-hand for the value of the lauded $w \cdot x + b$ equation from Figure 6.7. When z is any value less than or equal to zero, the perceptron outputs its smallest possible output, `0`. If z becomes positive to even the tiniest extent, the perceptron outputs its largest possible output, `1`. This sudden and extreme transition is not optimal during training: When we train a network, we make slight adjustments to w and b based on whether it appears the adjustment will improve the network's output.[4] With the perceptron, the majority of slight adjustments to w and b would make no difference whatsoever to its output; z would generally be moving around at negative values much lower than zero or at positive values much higher than zero. That behavior on its own would be unhelpful, but the situation is even worse: Every once and a while, a slight adjustment to w or b will cause z to cross from negative to positive (or vice versa), leading to a whopping, drastic swing in output from zero all the way to one (or vice versa). Essentially, the perceptron has no finesse—it's either yelling or it's silent.

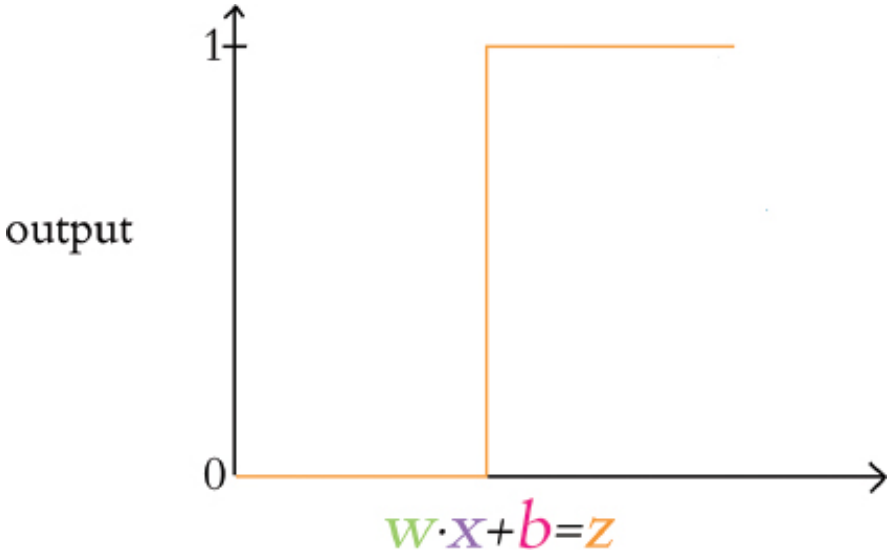

Figure 6.8 The perceptron's transition from outputting zero to outputting one happens suddenly, making it challenging to gently tune w and b to match a desired output.

The Sigmoid Neuron

Figure 6.9 provides an alternative to the erratic behavior of the perceptron: a gentle curve from `0` to `1`. This particular curve shape is called the *sigmoid* function and is defined by $\sigma(z) = \frac{1}{1+e^{-z}}$, where:

- z is equivalent to $w \cdot x + b$
- e is the mathematical constant beginning in 2.718... that is perhaps best known for its starring role in the natural exponential function
- σ is the Greek letter *sigma*, the root word for "sigmoid"

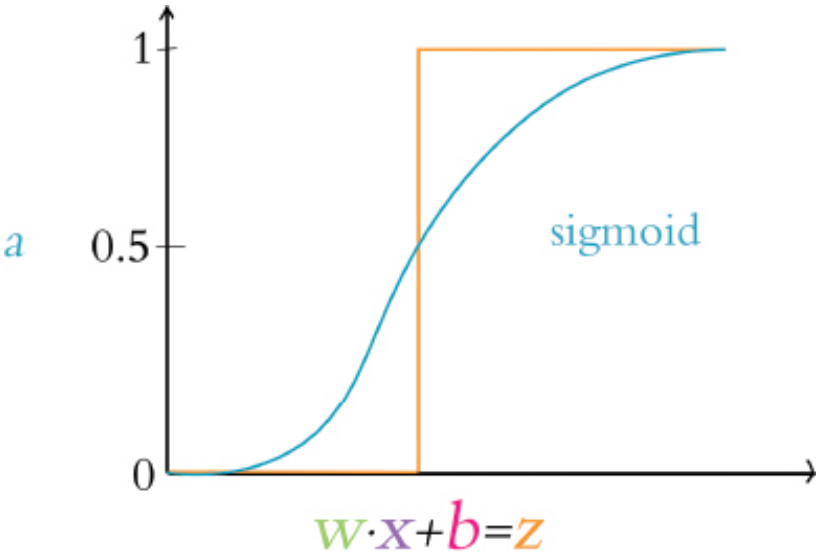

Figure 6.9 The sigmoid activation function. z is the input into the

The sigmoid function is our first example of an artificial neuron *activation function*. It may be ringing a bell for you already because it was the neuron type that we selected for the hidden layer of our *Shallow Net in Keras* from Chapter 5. As we'll see as this section progresses, the sigmoid function is the canonical activation function; so much so that the Greek letter σ (sigma) is conventionally used to denote *any* activation function. The output from any given neuron's activation function is referred to simply as its *activation* and throughout this book, we will use the variable term a—as shown along the vertical axis in Figure 6.9—to denote it.

In our view, there is no need to memorize the sigmoid function (or indeed any of the activation functions). Instead, we believe it's easier to understand a given function by playing around with its behavior interactively. With that in mind, feel free to join us in the *Sigmoid Function* Jupyter notebook from the book's GitHub repository as we work through the following lines of code.

Our only dependency in the notebook is the constant e, which we load with `from math`

`import e`. Next is the fun bit, where we define the sigmoid function itself:

```
def sigmoid(z):

    return 1/(1+e**-z)
```

As depicted in Figure 6.9 and demonstrated by executing `sigmoid(.00001)`, near-zero inputs into the sigmoid function will lead it to return values near `0.5`. Increasingly large positive inputs will result in values that approach 1. As an extreme example, an input of `10000` results in an output of `1.0`. Moving more gradually with our inputs—this time in the negative direction—we obtain outputs that gently approach zero: As examples, `sigmoid(-1)` returns `0.2689` while `sigmoid(-10)` returns `4.5398e-05`.[5]

Any artificial neuron that features the sigmoid function as its activation function is called a *sigmoid neuron* and the advantage of these over the perceptron should now be tangible: Small, gradual changes in a given sigmoid neuron's parameters w or b cause small, gradual changes in z, thereby producing similarly gradual changes in the neuron's activation, a. Large negative or large positive values of z illustrate an exception: At extreme z values, sigmoid neurons—like perceptrons—will output `0`'s (when z is negative) or `1`'s (when z is positive). Just like the perceptron, this means that subtle updates to the weights and biases during training will have little to no effect on the output and thus learning will stall. This situation is called neuron *saturation* and can occur with any activation function. Thankfully, there are tricks to avoid saturation, as we'll see in Chapter 9.

The Tanh Neuron

A popular cousin of the sigmoid neuron is the *tanh* (pronounced "tanch" in the deep-learning community) neuron. The *tanh* activation function is pictured in Figure 6.10 and is defined by $\sigma(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$. The shape of the *tanh* curve is similar to the sigmoid curve, with the chief distinction being that the sigmoid function exists in the range [0 : 1], while the *tanh* neuron's output has the range [−1 : 1]. This difference is more than cosmetic. With negative z inputs corresponding to negative a activations, z = 0 corresponding to a = 0, and positive z corresponding to positive a activations, the output from *tanh* neurons tends to be centered near zero. As we'll cover further in Chapters 7 through 9, these zero-centered a outputs usually serve as the inputs x to other artificial neurons in a network, and such zero-centered inputs make (the

dreaded!) neuron saturation markedly less likely, thereby enabling the entire network to learn more efficiently.

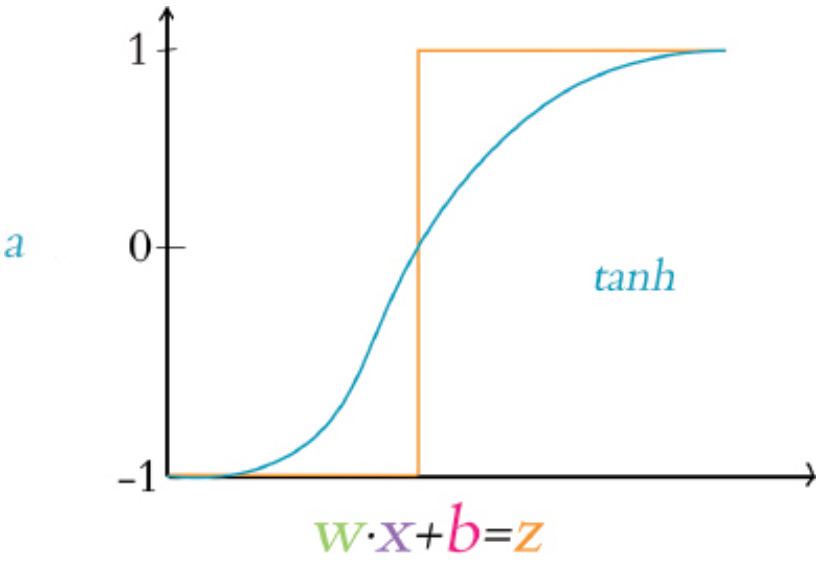

Figure 6.10 The tanh activation function.

ReLU: Rectified Linear Units

The final neuron we'll detail in this book is the *Rectified Linear Unit*, or *ReLU* neuron, whose behavior we've graphed in Figure 6.11. The ReLU activation function, whose shape diverges glaringly from the sigmoid and *tanh* sorts, was inspired by properties of biological neurons [6] and popularized within artificial neural networks by Vinod Nair and Geoff Hinton (Figure 1.17). [7] The shape of the ReLU function is defined by a = *max*(0, z). This function is uncomplicated:

œ If z is a positive value, the ReLU activation function returns z (unadulterated) as a = z.

œ If z = 0 or z is negative, the function returns its floor value of zero, i.e., the activation a = 0.

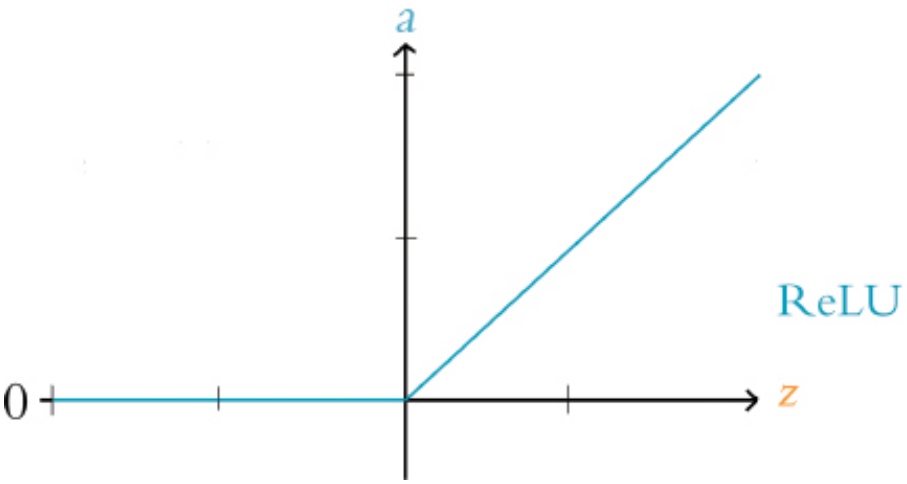

Figure 6.11 The ReLU activation function.

The ReLU function is one of the simplest functions to imagine that is *non-linear*. That is, like the sigmoid and *tanh* functions, its output a does not vary uniformly linearly across all values of z. The ReLU is in essence *two* distinct linear functions combined (one at negative z values returning zero, and the other at positive z values returning z, as is visible in Figure 6.11) to form a straightforward, non-linear function overall. This non-linear nature is a critical property of all activation functions used within deep learning architectures. As demonstrated via a series of captivating interactive applets in Chapter 4 of Michael Nielsen's *Neural Networks and Deep Learning* e-book, these non-linearities permit deep learning models to approximate any continuous function. [8] This universal ability to approximate some output y given some input x is one of the hallmarks of deep learning—the characteristic that makes the approach so effective across such a breadth of applications.

The relatively simple shape of the ReLU function's particular brand of non-linearity works to its advantage. As we'll see in Chapter 8, learning appropriate values for w and b within deep learning networks involves partial derivative calculus, and these calculus operations are much more computationally efficient on the linear portions of the ReLU function relative to on the curves of, say, the sigmoid and *tanh* functions. [9] As a testament to its utility, the incorporation of ReLU neurons into AlexNet (Figure 1.18) was one of the factors behind it trampling existing machine-vision benchmarks in 2012 and shepherding in the era of deep learning. Today, ReLU units are the most widely-used neuron within the hidden layers of deep artificial neural networks and they appear in the majority of the Jupyter notebooks associated with this book.

CHOOSING A NEURON

Within a given hidden layer of an artificial neural network, you are able to choose any activation function you fancy. With the constraint that you should select a non-linear function if you'd like to be able to approximate any continuous function with your deep learning model, you're nevertheless left with quite a bit of room for choice. To assist your decision-making process, let's rank the neuron types we already discussed in this chapter, ordering them from those we recommend least through to those we recommend most:

1. The *perceptron*, with its binary inputs and the aggressive step of its binary output, is not a practical consideration for deep learning models.

2. The *sigmoid* neuron is an acceptable option but it tends to lead to neural networks that train less rapidly than those composed of, say, *tanh* or ReLU neurons. Thus, we recommend limiting your use of sigmoid neurons to situations where it would be

helpful to have a neuron provide output within the range of [0, 1].[10]

3. The *tanh* neuron is a solid choice. As we covered above, their zero-centered output helps deep learning networks learn rapidly.

4. Our preferred neuron is the *ReLU* because of how efficiently learning algorithms can perform computations with them. In our experience they tend to lead to well-calibrated artificial neural networks in the shortest period of training time.

In addition to the neurons covered in this chapter, there is a veritable zoo of activation functions available and the list is ever-growing. At time of writing, some of the "advanced" activation functions provided by Keras[11] are the *Leaky ReLU*, the *Parametric ReLU*, and the *Exponential Linear Unit*—all three of which are derivations from the ReLU neuron. We encourage you to check these activations out in the Keras documentation and read about them on your own time. Furthermore, you are welcome to swap out the neurons we use in any of the Jupyter notebooks in this book to compare the results. We'd be pleasantly surprised if you discover they provide efficiency or accuracy gains in your neural networks that are far beyond the performance of ours.

SUMMARY

In this chapter, we detailed the mathematics behind the neural units that make up artificial neural networks, including deep learning models. We also summarized the pros and cons of the most established neuron types, providing you with guidance on which ones you might select for your own deep learning models. In the upcoming chapter, we'll cover how artificial neurons are networked together in order to learn features from raw data and approximate complex functions.

KEY CONCEPTS

This section should probably appear as a sidebar-style box. As the list lengthens in subsequent chapters, it would probably appear tidier if the list were laid out across multiple columns.

As we move through the chapters of the book, we will gradually add terms to this list of *Key Concepts*. If you keep these foundational concepts fresh in your mind, you should have little difficulty understanding subsequent chapters and, by book's end, possessing a firm grip on deep learning theory and application. The critical concepts thus far are:

- parameter
- weight w

œ bias b

œ activation a

œ artificial neuron

œ sigmoid

œ *tanh*

œ ReLU

1. More precisely, it causes a change in the voltage *difference* between the cell's interior and its surroundings.

2. Rosenblatt, F. (1958). The Perceptron: A Probabilistic Model for Information Storage and the Organization in the Brain. *Psychological Review, 65,* 386-408.

3. If you are well-accustomed to regression modeling, this should be a familiar paradigm.

4. *Improvement* here means providing output more closely in line with the true output y given some input x. We'll discuss this further soon, in Chapter 8.

5. The *e* in `4.5398e-05` should not be confused with the base of the natural logarithm. Here, it refers to an *exponent*, so the output is the equivalent of 4.5398×10^{-5}.

6. The action potentials of biological neurons have only a "positive" firing mode; they have no "negative" firing mode. See Hahnloser, R., & Seung, H. (2001). Permitted and Forbidden Sets in Symmetric Threshold-Linear Networks. *Proceedings of Neural Information Processing Systems*.

7. Nair, V. & Hinton, G. (2010). Rectified Linear Units Improve Restricted Boltzmann Machines. *Proceedings of the International Conference on Machine Learning*.

8. neuralnetworksanddeeplearning.com/chap4.html

9 . In addition, there is mounting research that suggests ReLU activations encourage *parameter sparsity*—i.e., less elaborate neural-network-level functions that tend to generalize to validation data better. More on model generalization coming up in Chapter 9.

10. In Chapter 11, we will encounter a couple of these situations—most notably, with a sigmoid neuron as the sole neuron in the output layer of a binary-classifier network.

11. See keras.io/layers/advanced-activations

7 Artificial Neural Networks

In the preceding chapter, we examined the intricacies of artificial neurons. The theme of the current chapter is the natural extension of that: We'll cover how individual neural units are linked together to form artificial neural networks, including deep learning networks.

THE INPUT LAYER

In our *Shallow Net in Keras* Jupyter notebook (a schematic of which is available back in Figure 5.4), we crafted an artificial neural network with:

1. an *input* layer consisting of 784 neurons, one for each of the 784 pixels in an MNIST image

2. a *hidden* layer composed of 64 sigmoid neurons

3. an *output* layer consisting of 10 *softmax* neurons, one for each of the ten classes of digits

Of these three, the input layer is the most straightforward to detail. We'll start with it, and then move onto discussion of the hidden and output layers.

Neurons in the input layer don't perform any calculations; they are simply placeholders for input data. This place-holding is essential because, as we'll see first-hand in Chapter 14 when we begin writing code in low-level TensorFlow, the use of artificial neural networks involves performing computations on matrices that have pre-defined dimensions. At least one of these pre-defined dimensions in the network architecture corresponds directly to the shape of the input data.

DENSE LAYERS

There are many kinds of hidden layers, but as mentioned in Chapter 4, the most general type is the *dense layer*, which can also be called a *fully-connected layer*. Dense layers are found in many deep learning architectures, including the majority of the models

we'll go over in this book. Their definition is uncomplicated: Each of the neurons in a given dense layer receive information from every one of the neurons in the previous layer of the network. In other words, a dense layer is *fully-connected* to the layer before it!

While they might not be as specialized nor as efficient as the other flavors of hidden layers we'll get to in Part III, dense layers are broadly useful because they can non-linearly recombine the information provided by the previous layer of the network. [1] Reviewing the TensorFlow Playground demo from the end of Chapter 1, we're now better-positioned to appreciate the deep learning model we built. Breaking it down layer by layer, the network in Figures 1.19 and 1.20 has:

1. An *input layer with two neurons*: one for storing the vertical position of a given dot within the grid on the far right, and the other for storing the dot's horizontal position.

2. A *hidden layer composed of eight ReLU neurons*. Visually, we can see that this is a *dense layer* because each of the eight neurons in it is connected (i.e., is receiving information) from both of the input-layer neurons, for a total of 16 (= 8 × 2) incoming connections.

3. Another *hidden layer composed of eight ReLU neurons*. We can again discern that this is a *dense layer* because its eight neurons each receive input from each of the eight neurons in the previous layer, for a total of 64 (= 8 × 8) inbound connections. Note how the neurons in this layer are non-linearly recombining the straight-edge features provided by the neurons in the first hidden layer to produce more elaborate features like curves and circles. [2]

4. A third *dense hidden layer*, this one *consisting of four ReLU neurons* for a total of 32 (= 4 × 8) connecting inputs. This layer non-linearly recombines the features from the previous hidden layer to learn more complex features that begin to look directly relevant to the binary (orange versus blue) classification problem shown in the grid on the right.

5. A fourth and final *dense hidden layer*. With its *two ReLU neurons*, it receives a total of eight (= 2 × 4) inputs from the previous layer. The neurons in this layer devise such elaborate features via non-linear recombination that they visually approximate the overall boundary dividing blue from orange on the grid.

6. An *output layer made up of a single sigmoid neuron*. Sigmoid is the typical choice of neuron for a binary classification problem like this one. As shown in Figure 6.9, the sigmoid function outputs activations that range from zero up to one, allowing us to

obtain the network's estimated probability that a given input x is a positive case (a blue dot in the current example) or inversely, the probability that it is a negative case. Like the hidden layers, the *output layer is dense* too: Its neuron receives information from both neurons of the final hidden layer for a total of two (= 1 × 2) connections.

In summary, *every* layer within the networks provided by the TensorFlow Playground is a dense layer. We can call such a network a *dense network* and we'll be experimenting with these versatile creatures for the remainder of Part II.

A HOT DOG-DETECTING DENSE NETWORK

Let's further strengthen our comprehension of dense networks by returning to two old flames of ours from Chapter 6: a frivolous hot dog-detecting binary classifier and the mathematical notation we used to define artificial neurons. As shown in Figure 7.1, our hot dog classifier is no longer a single neuron; in this chapter, it is a dense network of artificial neurons. More specifically, with this network architecture:

œ We have reduced the number of input neurons down to two for simplicity:

œ The first input neuron, x_1, represents the volume of ketchup (in, say, milliliters, which abbreviates to *mL*) on the object being considered by the network. (We are no longer working with perceptrons, so we are no longer restricted to binary inputs only.)

œ The second input neuron, x_2, represents mL of mustard.

œ We have two dense hidden layers:

œ The first hidden layer has three ReLU neurons.

œ The second hidden layer has two ReLU neurons.

œ The output neuron is denoted by $\hat{y}$ in the network. This is a binary classification problem, so—as outlined in the previous section—this neuron should be sigmoid. As in our perceptron examples in Chapter 6, $y = 1$ corresponds to the presence of a hot dog and $y = 0$ corresponds to the presence of some other object.

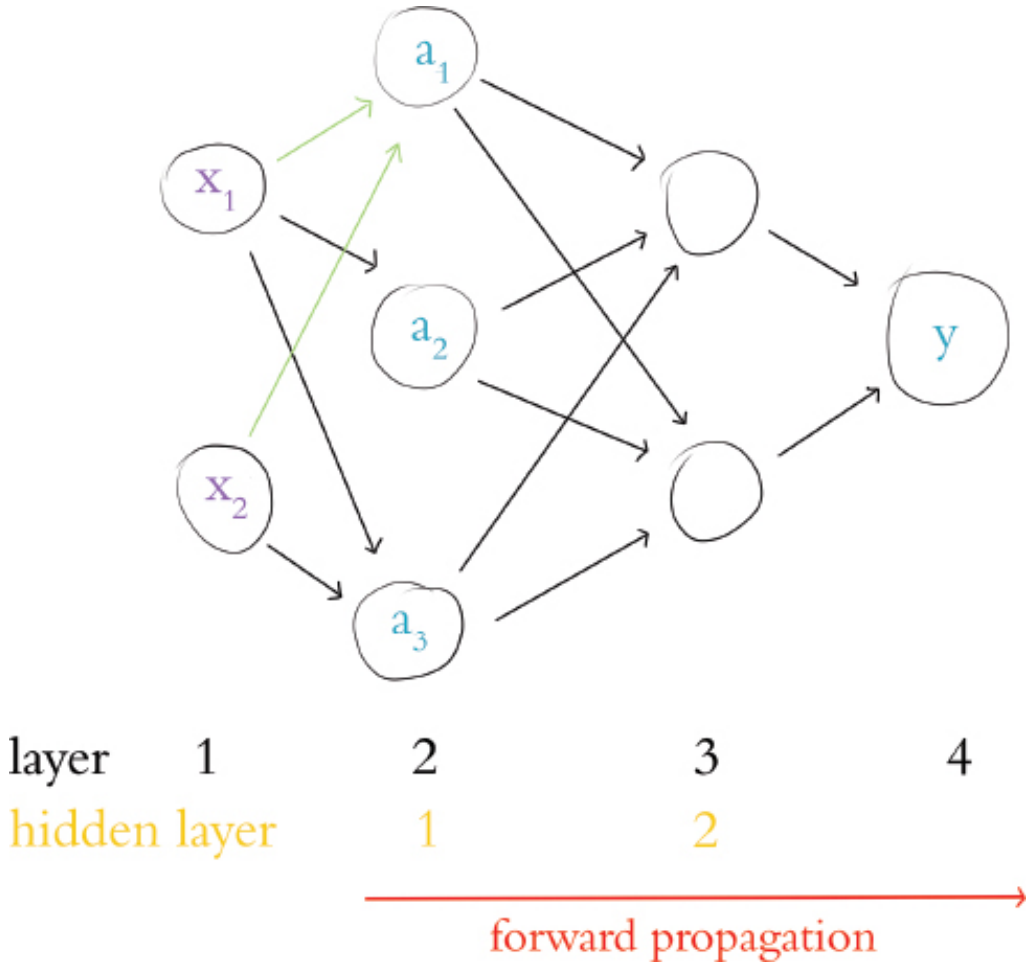

Figure 7.1 A dense network of artificial neurons, highlighting the inputs to the neuron labelled a_1.

Forward Propagation through the First Hidden Layer

Having described the architecture of our hot dog-detecting network, let's turn our attention to its functionality by focusing on the neuron labelled a_1. [3] This particular neuron, like its siblings a_2 and a_3, receives input regarding a given object's *ketchup-y-ness* and *mustard-y-ness* from x_1 and x_2, respectively. Despite receiving the same data as a_2 and a_3, a_1 treats these data uniquely by having its own unique parameters. Remembering Figure 6.7, "the most important equation in this book" —$w\,x + b$—we may grasp this behavior more concretely. Breaking this equation down for the neuron labelled a_1, we consider that it has two inputs from the previous layer, x_1 and x_2. This neuron also has two weights: w_1 (which applies to the importance of the ketchup measurement x_1) and w_2 (which applies to the importance of the mustard measurement x_2). With these five pieces of information we can calculate z, the weighted input to that neuron:

$$z = w \cdot x + b$$
$$z = (w_1 x_1 + w_2 x_2) + b \qquad (7.1)$$

In turn, with the z value for the neuron labelled a_1, we can calculate the activation a it outputs. Since the neuron labelled a_1 is a ReLU neuron, we use the equation introduced

in Figure 6.11:

$$a = max(0, z) \tag{7.2}$$

To make this computation of the output of neuron a_1 tangible, let's concoct some numbers and work through the arithmetic together:

- x_1 is 4.0 mL of ketchup for a given object presented to the network
- x_2 is 3.0 mL of mustard for that same object
- $w_1 = -0.5$
- $w_2 = 1.5$
- $b = -0.9$

To calculate z let's start with Equation 7.1 and then fill in our contrived values:

$$\begin{aligned} z &= w \cdot x + b \\ &= w_1 x_1 + w_1 x_2 + b \\ &= -0.5 \times 4.0 + 1.5 \times 3.0 - 0.9 \\ &= -2 + 4.5 - 0.9 \\ &= 1.6 \end{aligned} \tag{7.3}$$

Finally, to compute a—the activation output of the neuron labelled a_1—we can leverage Equation 7.2:

$$\begin{aligned} a &= max(0, z) \\ &= max(0, 1.6) \\ &= 1.6 \end{aligned} \tag{7.4}$$

As suggested by the right-facing arrow along the bottom of Figure 7.1, executing the calculations through an artificial neural network from the input layer (the x values) through to the output layer ($\hat{y}$) is called *forward propagation*. Immediately above, we detailed the process for forward propagating through a single neuron in the first hidden layer of our hot dog-detecting network. To forward propagate through the remaining neurons of the first hidden layer—that is, to calculate the a values for the neurons labelled a_2 and a_3—we would follow the same process as we did for the neuron labelled a_1. The inputs x_1 and x_2 are identical for all three neurons, but despite being fed the same measurements of ketchup and mustard, each neuron in the first hidden layer will output a different activation a because the parameters w_1, w_2 and b vary for each of the

neurons in the layer.

Forward Propagation through Subsequent Layers

The process of forward propagating through the remaining layers of the network is essentially the same as propagating through the first hidden layer, but for clarity's sake, let's work through it together. In Figure 7.2, we'll assume that we've already calculated the activation value a for each of the neurons in the first hidden layer. Returning our focus to the neuron labelled a_1, the activation it outputs ($a_1 = 1.6$) becomes one of the three inputs into the neuron labelled a_4 (and, as highlighted in the figure, this same activation of $a = 1.6$ is also fed as one of the three inputs into the neuron labelled a_5).

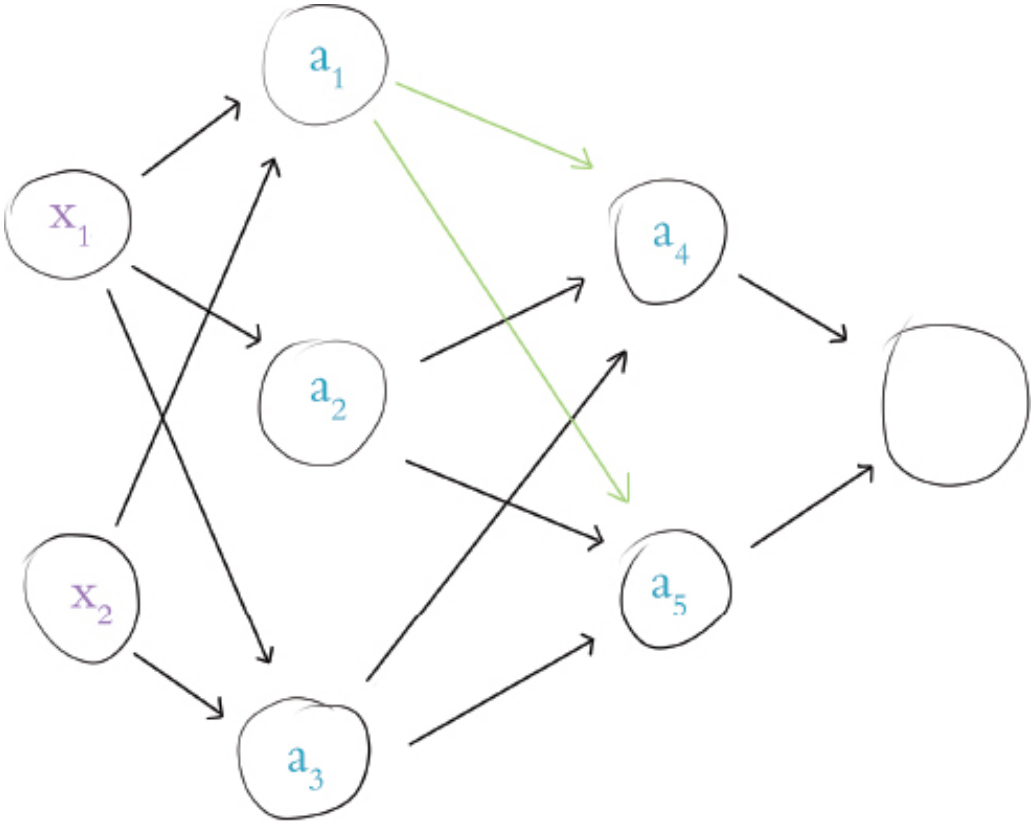

Figure 7.2 Our hot dog-detecting network from Figure 7.1, now highlighting the activation output of neuron a_1, which is provided as an input to both neuron a_4 and neuron a_5.

To provide an example of forward propagation through the second hidden layer, let's compute a for the neuron labelled a_4. Again, we employ the all-important equation $w \cdot x + b$. For brevity's sake, we've combined it with the ReLU activation function:

$$\begin{aligned} a &= max(0, z) \\ &= max(0, (w \cdot x + b)) \\ &= max(0, (w_1 x_1 + w_2 x_2 + w_3 x_3 + b)) \end{aligned} \tag{7.5}$$

This is sufficiently similar to Equations 7.3 and 7.4 that it would be superfluous to walk through the arithmetic again with feigned values. The only twist, as we propagate

through the second hidden layer, is that the layer's inputs (i.e., x in the equation $w\,x + b$) come not from outside the network—instead they are provided by the first hidden layer. Thus, in Equation 7.5:

- x_1 is the value $a = 1.6$, which we obtained earlier from the neuron labelled a_1
- x_2 is the activation output a (whatever it happens to equal) from the neuron labelled a_2, and
- x_3 is likewise a unique activation a from the neuron labelled a_3

In this manner, the neuron labelled a_4 is able to non-linearly recombine the information provided by the three neurons of the first hidden layer. The neuron labelled a_5 also non-linearly recombines this information, but it would do it in its own distinctive way: The unique parameters w_1, w_2, w_3 and b for this neuron would lead it to output a unique a activation of its own.

Having illustrated forward propagation through all of the hidden layers of our hot dog-detecting network, let's round the process off by propagating through the output layer. Figure 7.3 highlights that our single output neuron receives its inputs from the neurons labelled a_4 and a_5. Let's begin by calculating z for this output neuron. The formula is identical to Equation 7.1, which we used to calculate z for the neuron labelled a_1, except that the (contrived, as usual) values we plug into the variables are different:

$$\begin{aligned} z &= w \cdot x + b \\ &= w_1 x_1 + w_2 x_2 + b \\ &= 1.0 \times 2.5 + 0.5 \times 2.0 - 5.5 \\ &= 3.5 - 5.5 \\ &= -2.0 \end{aligned} \tag{7.6}$$

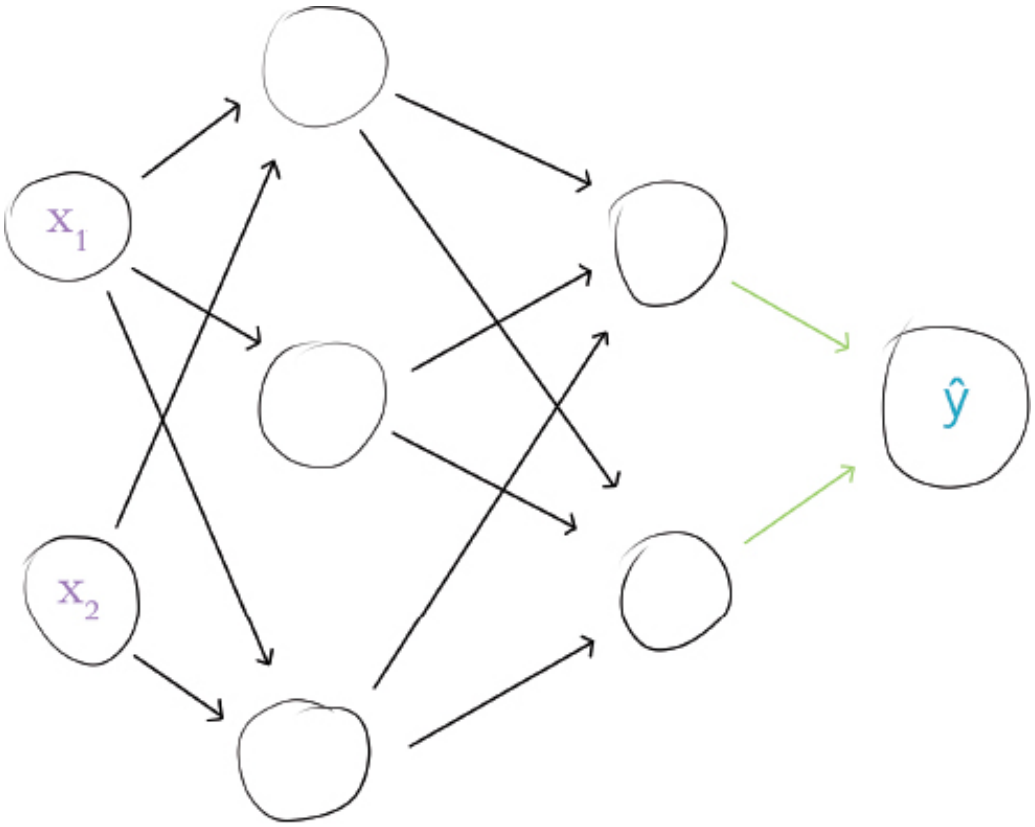

Figure 7.3 Our hot dog-detecting network, with the activations providing input to the output neuron $\hat{y}$ highlighted.

The output neuron is sigmoid so to compute its activation a we pass its z value through the sigmoid function from Figure 6.9:

$$\begin{aligned} a &= \sigma(z) \\ &= \frac{1}{1+e^{-z}} \\ &= \frac{1}{1+e^{-(-2.0)}} \\ &\approx 0.1192 \end{aligned} \tag{7.7}$$

We are lazy, so we didn't work out the final line of this equation manually. Instead, we used the *Sigmoid Function* Jupyter notebook that we created in Chapter 6. By executing the line `sigmoid(-2.0)` within it, our machine did the heavy lifting for us and kindly informed us that a comes out to `0.1192` and change.

The activation a computed by the sigmoid neuron in the output layer is a very special case because it is the final output of our entire hot dog-detecting neural network. Since it's so special, we assign it a distinctive designation: $\hat{y}$, which is pronounced "why hat". This value $\hat{y}$ is the network's guess as to whether the object presented to it was a hot dog or not a hot dog, and we can express this in probabilistic language. Given the inputs x_1 and x_2 that we fed into the network—that is, 4.0 mL of ketchup and 3.0 mL of mustard—the network estimates that there is an 11.92% chance that an object with

those particular condiment measurements is a hot dog. [4] If the object presented to the network was indeed a hot dog ($y = 1$) then this $\hat{y}$ of 0.1192 was pretty far off the mark. On the other hand, if the object was truly not a hot dog ($y = 0$) then the $\hat{y}$ is quite good. We'll formalize the evaluation of $\hat{y}$ in Chapter 8, but the general notion is is that the closer $\hat{y}$ is to the true value y, the better.

THE SOFTMAX LAYER OF A FAST FOOD-CLASSIFYING NETWORK

As demonstrated thus far in the chapter, the sigmoid neuron suits us well if we're building a network to distinguish two classes, e.g., a blue dot versus an orange dot, or a hot dog versus something other than hot dog. In many other circumstances, however, we have more than two classes to distinguish between. For example, MNIST consists of the ten numerical digits, so our *Shallow Net in Keras* from Chapter 6 had to accommodate ten output probabilities—one representing each digit.

When concerned with a multi-class problem, the solution is to use a *softmax* layer as the output layer of our network. Softmax is in fact the activation function that we specified for the output layer in our *Shallow Net in Keras* Jupyter notebook, but we initially suggested you not worry yourself with that detail too much. Now, a couple of chapters later, the time to unravel softmax has arrived.

In Figure 7.4, we've provided a new architecture that builds upon our binary hot dog classifier. The schematic is the same—right down to its volume-of-ketchup-and-mustard inputs—except that instead of having a single output neuron, we now have three. This multi-class output layer is still dense, so each of the three neurons receives information from both of the neurons in the final hidden layer. Continuing on with our proclivity for fast food, let's say that now:

- y_1 represents hot dogs,
- y_2 is for burgers, and
- y_3 is for pizza.

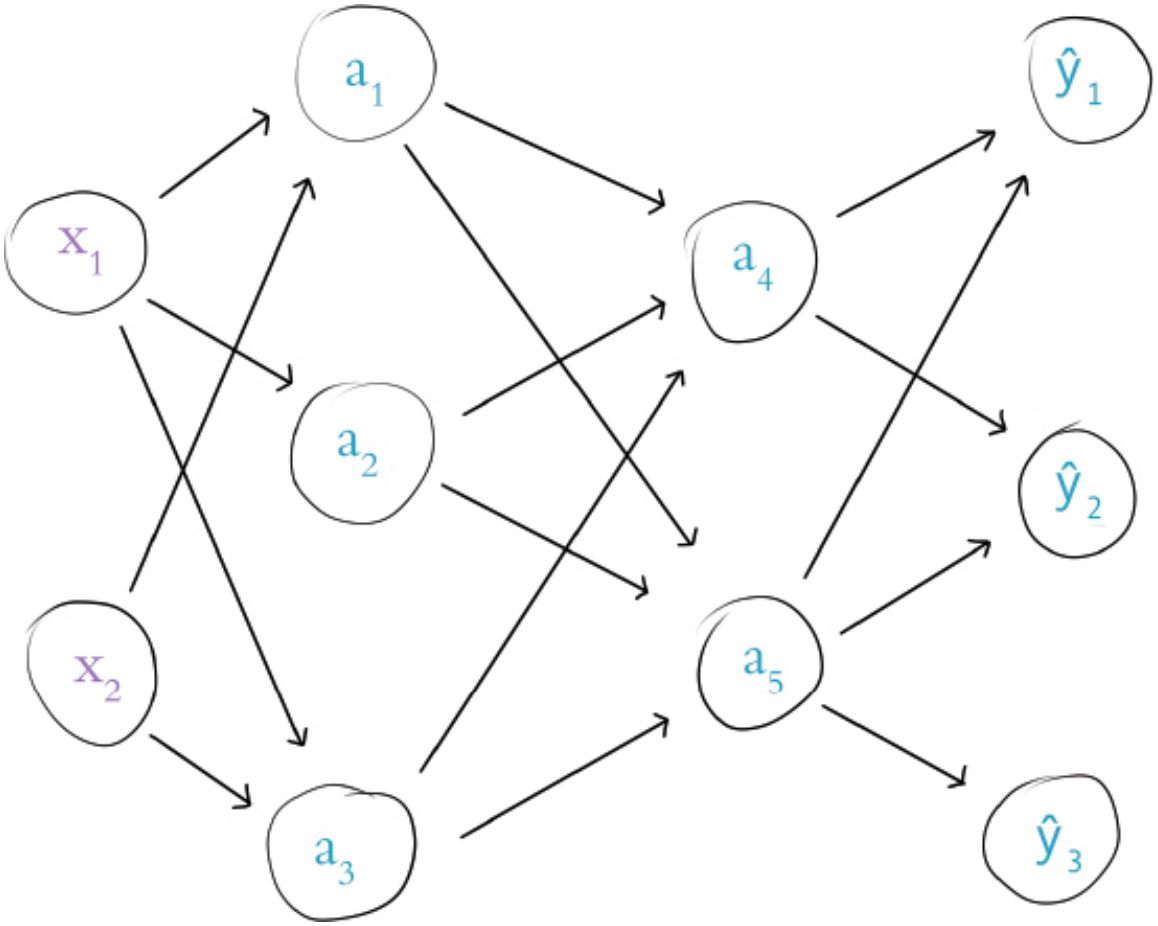

Figure 7.4 Our hot dog-detecting network, now with three softmax neurons in the output layer.

Note that with this configuration, there can be no alternatives to hot dog, burger or pizza. The assumption is that all objects presented to the network belong to one of these three classes of fast food, and *one* of the classes only.

Because the sigmoid function applies only to binary problems, the output neurons in Figure 7.4 take advantage of the softmax activation function. Let's use code from our *Softmax Demo* Jupyter notebook to elucidate how this activation function operates. The only dependency is the `exp` function, which calculates the natural exponential of whatever value it's given. More specifically, if we pass x into it with the command `exp(x)`, we will get back e^x. The effect of this exponentiation will become clear as we move through the forthcoming example. We import the `exp` function into the notebook with `from math import exp`.

To concoct another example, let's say that we presented a slice of pizza to the network in Figure 7.4. Presumably this pizza slice has negligible amounts of ketchup and mustard on it, and so x_1 and x_2 are near-zero values. Provided these inputs, we use forward propagation to pass information through the network toward the output layer. Based on the information that the three neurons receive from the final hidden layer, they individually use our old friend $w \cdot x + b$ to calculate three unique z values:

- z for the neuron labelled y_1, which represents hot dogs, comes out to -1.0
- for the neuron labelled y_2, which represents burgers, z is 1.0
- and for the pizza neuron y_3, z comes out to 5.0

These values indicate that the network estimates that the object presented to it is most likely to be pizza and least likely to be a hot dog. Expressed as z, however, it isn't straightforward to intuit *how much* more likely the network predicts the object to be pizza relative to the other two classes. That's where the softmax function comes in.

After importing our dependency, we create a list named z to store our three z values:

```
z = [-1.0, 1.0, 5.0]
```

Applying the softmax function to this list involves a three-step process. The first step is to calculate the exponential of each of the z values. More explicitly:

- `exp(z[0])` comes out to `0.3679` for hot dog [5]
- `exp(z[1])` gives us `2.718` for burger
- and `exp(z[2])` gives us the much much larger (exponentially so!) `148.4` for pizza

The second step of the softmax function is to sum up our exponentials:

```
total = exp(z[0]) + exp(z[1]) + exp(z[2])
```

With this `total` variable we can execute the third and final step, which provides proportions for each of our three classes relative to sum of all of the classes:

- `exp(z[0])/total` outputs `0.002428`, indicating that the network estimates there's a ~0.2% chance that the object presented to it is a hot dog
- `exp(z[1])/total` gives us `0.01794`, indicating an estimated ~1.8% chance that it's a burger, and
- `exp(z[2])/total` returns `0.9796` for an estimated ~98.0% chance that the object is pizza

Given the above arithmetic, the etymology of the "softmax" name should now be discernible: The function returns z with the highest value (the *max*), but it does so *soft*-ly. That is, instead of indicating that there's a 100% chance the object is pizza and a zero percent chance it's either of the other two fast food classes (that would be… *hard*), the network hedges its bets, to an extent, and provides a likelihood that the object is each of the three classes. This leaves us to make the decision over how much confidence we would require before we make a decision. [6]

Trilobite-reading sidebar: The use of the softmax function with a single neuron is a special case of softmax that is mathematically equivalent to using a sigmoid neuron.

REVISITING OUR SHALLOW NETWORK

With the knowledge of dense networks that we developed over the course of this chapter, we can return to our *Shallow Net in Keras* notebook and understand the model summary within it. Example 5.2 shows the three lines of Keras code we used to architect a shallow neural network for classifying MNIST digits. As detailed in Chapter 5, over those three lines of code we instantiated a model object and added layers of artificial neurons to it. By calling the summary() on the model, we see the model-summarizing table provided in Figure 7.5. The table has three columns:

- `Layer (type)`: the name and type of each of our layers
- `Output Shape`: the dimensionality of the layer
- `Param #`: the number of parameters (weights w and biases b) associated with the layer

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 64)                50240
_________________________________________________________________
dense_2 (Dense)              (None, 10)                650
=================================================================
Total params: 50,890
Trainable params: 50,890
Non-trainable params: 0
_________________________________________________________________
```

Figure 7.5 A summary of the model object from our "Shallow Net in Keras" Jupyter notebook.

The input layer performs no calculations and never has any of its own parameters so no information on it is displayed directly. The first row in the table, therefore, corresponds

to the first hidden layer of the network. The table indicates that this layer:

- is called `dense_1`; this is a default name as we did not designate one explicitly
- is a `Dense` layer, as we specified in Example 5.2
- is composed of `64` neurons, as we further specified in Example 5.2
- has `50240` parameters associated with it, broken down into:
  - `50176` weights, corresponding to each of the 64 neurons in this dense layer receiving input from each of the 784 neurons in the input layer (`64*784`)
  - plus 64 biases, one for each of the neurons in the layer
  - giving us a total of 50240 parameters:

$n_{parameters} = n_w + n_b = 50176 + 64 = 50240$

The second row of the table in Figure 7.5 corresponds to the model's output layer. The table tells us that this layer:

- is called `dense_2`
- is a `Dense` layer, as we specified it to be
- consists of `10` neurons—yet again, as we specified
- has `650` parameters associated with it:
  - `640` weights, corresponding to each of the ten neurons receiving input from each of the 64 neurons in the hidden layer (`64*10`)
  - plus 10 biases, one for each of the output neurons

From the parameter counts for each layer, we can calculate for ourselves the `Total params` line displayed in Figure 7.5:

$$\begin{aligned} n_{total} &= n_1 + n_2 \\ &= 50240 + 650 \\ &= 50890 \end{aligned} \tag{7.8}$$

All `50890` of these parameters are "`Trainable params`" because—during the

subsequent `model.fit()` call in the *Shallow Net in Keras* notebook—they are permitted to be tuned during model training. This is the norm, but as we'll see in Part III, there are situations where it is fruitful to freeze some of the parameters in a model rendering them "`Non-trainable params`".

SUMMARY

In this chapter, we detailed how artificial neurons are networked together to approximate an output y given some inputs x. In the remaining chapters of Part II, we'll detail how a network learns to improve its approximations of y by using data to tune the parameters of its constituent artificial neurons. Simultaneously, we'll broaden our understanding of best practices for designing and training artificial neural networks so that we can add hidden layers and form a high-calibre deep learning model.

KEY CONCEPTS

Here are the essential foundational concepts thus far. New terms from the current chapter are highlighted in purple:

œ parameters:

œ weight w

œ bias b

œ activation a

œ artificial neurons:

œ sigmoid

œ *tanh*

œ ReLU

œ input layer

œ hidden layer

œ output layer

œ layer types:

œ dense (fully-connected)

œ softmax

œ forward propagation

1 . This statement assumes the dense layer is made up of neurons with a non-linear activation function like the sigmoid, *tanh* and ReLU neurons introduced in the previous chapter, which should be a safe assumption.

2 . By optionally returning to playground.tensorflow.org, you can observe these features by hovering over these neurons with your mouse.

3 . We're using a shorthand notation for conveniently identifying neurons in this chapter. See Appendix 17 (This should presumably be labelled Appendix A) for a more precise and formal neural network notation.

4 . Don't say we didn't warn you from the start that this was a silly example! If we're lucky, its outlandishness will make it memorable.

5 . Recall that Python uses zero indexing so z[0] corresponds to the z of neuron y_1.

6 . Typically, we'd simply choose the class with the highest likelihood. This is easily achieved with the `argmax()` function in Python, which returns the index position (i.e., the class label) of the largest value.

8 Training Deep Networks

In the preceding chapters, we described artificial neurons comprehensively and we walked through the process of forward propagating information through a network of neurons to output a prediction, such as whether a given fast-food item is a hot dog, a juicy burger or a greasy slice of pizza. In those culinary examples from Chapters 6 and 7, we fabricated numbers for the neuron parameters—i.e., weights and biases. In real-world applications, however, these parameters are not typically concocted arbitrarily: They are learned by training the network on data.

In this chapter, we will become acquainted with two methods—called *gradient descent* and *backpropagation*—that work in tandem to learn artificial neural network parameters. As usual in this book, our presentation of these methods is not only theoretical: We will provide pragmatic best practices for implementing the techniques. The chapter will culminate in the application of these practices to the construction of an intermediate-depth neural network.

COST FUNCTIONS

In Chapter 7, we discovered that, upon forward propagating some input values all the way through an artificial neural network, the network provides its estimated output, which is denoted $\hat{y}$. If a network was perfectly calibrated, it would output $\hat{y}$ values that are exactly equal to the true label y. In our binary classifier for detecting hot dogs, for example (Figure 7.3), $y = 1$ indicated that the object presented to the network is a hot dog while $y = 0$ indicated that it's something else. In an instance where we have in fact presented a hot dog to the network, therefore, it would ideally output $\hat{y} = 1$.

In practice, the gold standard of $\hat{y} = y$ is not always attained and so may be an excessively stringent definition of the "correct" $\hat{y}$. Instead, we might be quite pleased to see a $\hat{y}$ of, say, 0.9997 as that would indicate that the network has an extremely high confidence that the object is a hot dog. A $\hat{y}$ of 0.9 might be considered acceptable, $\hat{y} = 0.6$ to be rather disappointing and $\hat{y} = 0.1192$ (as computed in Equation 7.7) to be downright awful.

To quantify the spectrum of output-evaluation sentiments from "quite pleased" all the way to "downright awful", machine learning algorithms often involve *cost functions* (also known as *loss functions*). The two such functions that we'll cover in this book are called *quadratic cost* and *cross-entropy cost*. Let's cover them in turn.

Quadratic Cost

Quadratic cost is one of the simplest cost functions to calculate. It is alternatively called *mean squared error*, which handily describes all that there is to its calculation:

$$C = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \qquad (8.1)$$

For any given instance i, we calculate the difference (the *error*) between the true label y_i and the network's estimated $\hat{y}_i$. We then *square* this difference, for two reasons:

1. Squaring ensures that whether y is greater than $\hat{y}$ or vice versa, the difference between the two is stated as a positive value.

2. Squaring penalizes large differences between y and $\hat{y}$ much more severely than small differences.

Having obtained a *squared error* for each instance i with $(y_i - \hat{y}_i)^2$, we can at last calculate the *mean* cost C across all n of our instances by:

1. Summing up all of our instances with $\sum_{i=1}^{n}$

2. Dividing by however many instances we have with $\frac{1}{n}$

By taking a peek inside the *Quadratic Cost* Jupyter notebook from the book's GitHub repo, you can play around with Equation 8.1 yourself. At the top of the notebook, we define a function to calculate squared error for an instance i:

```
def squared_error(y, yhat):

    return (y - yhat)**2
```

By plugging a true `y` of `1` and the ideal `yhat` of `1` into the function with `squared_error(1, 1)`, we observe that—as desired—this perfect estimate is associated with a cost of `0`. Likewise, minor deviations from the ideal such as a `yhat` of

`0.9997`, correspond to an extremely small cost: `9.0e-08`. [1] As the difference between `y` and `yhat` increases, we witness the expected exponential increase in cost: Holding `y` steady at `1` but lowering `yhat` from `0.9` to `0.6`, and then to `0.1192`, the cost climbs increasingly rapidly from `0.01` to `0.16` and then `0.78`. As a final bit of amusement in the notebook, we note that had `y` truly been `0`, our `yhat` of `0.1192` would be associated with a small cost: `0.0142`.

Saturated Neurons

While quadratic cost serves as a straightforward introduction to loss functions, it has a vital flaw. Consider Figure 8.1, in which we summarize the *tanh* activation function from back in Figure 6.10. The issue presented in the figure, called *neuron saturation*, is common across all activation functions but we'll use *tanh* as our lone exemplar. A neuron is considered saturated when the combination of its inputs and parameters (interacting as per "the most important equation" $z = w \cdot x + b$) produce extreme values of z—the areas encircled with red in the plot. In these areas, changes in z (via adjustments to the neuron's underlying parameters w and b) cause only teensy-weensy changes in the neuron's activation a. [2]

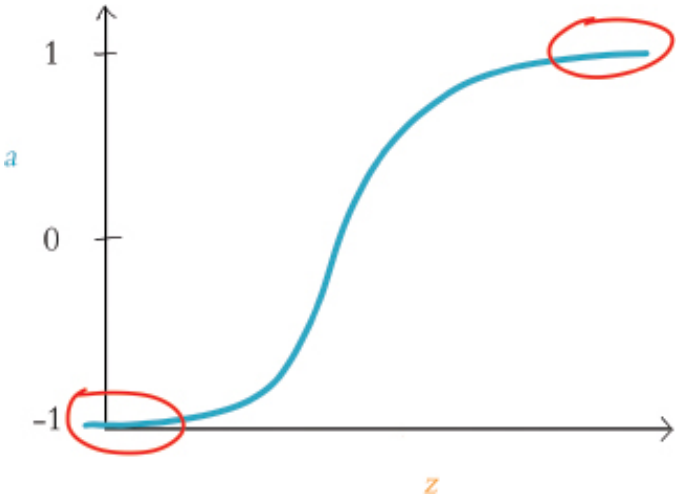

Figure 8.1 Plot reproducing the tanh activation function shown in Figure 6.10, drawing attention to the high and low values of z at which a neuron is saturated.

Using methods that we'll cover later in this chapter—namely, gradient descent and backpropagation—a neural network is able to learn to approximate y through the tuning of its neurons' parameters w and b. In a saturated neuron, where changes to w and b lead to only minuscule changes in a, this learning slows to a crawl: If adjustments to w and b make no discernible impact on a given neuron's activation a then these adjustments cannot have any discernible impact downstream (via forward propagation) on the network's $\hat{y}$, its approximation of y.

Cross-Entropy Cost

One of the ways [3] to minimize the impact of saturated neurons on learning speed is to use *cross-entropy cost* in lieu of quadratic cost. This alternative loss function is configured to enable efficient learning anywhere within the activation function curve of Figure 8.1. Because of this, it is a far more popular choice of cost function and it is the selection that predominates the remainder of this book.

You need not preoccupy yourself heavily with the equation for cross-entropy cost, but for the sake of completeness, here it is:

$$C = -\frac{1}{n}\sum_{i=1}^{n}[y_i \ln \hat{y}_i + (1 - y_i)\ln(1 - \hat{y}_i)] \qquad \textbf{(8.2)}$$

The most pertinent aspects of the equation are that:

1. Like quadratic cost, divergence of from $\hat{y}$ from y corresponds to increased cost.

2. Also, similar to the use of the square in quadratic cost, the use of the natural logarithm *ln* in cross-entropy cost causes larger differences between $\hat{y}$ and y to be associated with exponentially larger cost.

3. Cross-entropy cost is structured so that the larger the difference between $\hat{y}$ and y, *the faster the neuron is able to learn.* [4]

To make it easier to remember that the greater the cost, the more quickly a neural network incorporating cross-entropy cost learns, here's a flippant analogy that would absolutely never happen to any of your esteemed authors: Let's say you're at a cocktail party leading the conversation to a group of cool, stylish people that you've met that evening. The strong martini you're holding has already gone to your head and so you go out on a limb by throwing a vulgar joke into your otherwise charming repartee. Unexpectedly, your audience reacts with immediate, visible disgust. With this response clearly indicating that your quip was well off the mark, you learn pretty darn quickly. It's exceedingly unlikely you'll be repeating the joke anytime soon.

Anyway, that's plenty enough on disasters of social etiquette. The final item to note on cross-entropy cost is that, by including $\hat{y}$, the formula provided in Equation 8.2 applies to only the output layer. Recall from the previous chapter (specifically the discussion of Figure 7.3) that $\hat{y}$ is a special case of a: It's actually just another plain old a value—except that it's being calculated by neurons in the output layer of a neural network. With this in mind, Equation 8.3 could be expressed with a_i substituted in for $\hat{y}_i$ so that the equation generalizes neatly beyond the output layer to neurons in any layer of a network:

$$C = -\frac{1}{n}\sum_{i=1}^{n}[y_i \ln a_i + (1 - y_i)\ln(1 - a_i)] \quad (8.3)$$

To cement all of this theoretical chatter about cross-entropy cost, let's interactively explore our aptly-named *Cross Entropy Cost* Jupyter notebook. There is only one dependency in the notebook: the `log` function from the *NumPy* package, which enables us to compute the natural logarithm *ln* shown twice in Equation 8.3. We load this dependency with `from numpy import log`.

Next, we define a function for calculating cross-entropy cost for an instance *i*:

```
def cross_entropy(y, a):

    return -1*(y*log(a) + (1-y)*log(1-a))
```

Plugging the same values into our `cross_entropy()` function as we did into the `squared_error()` function earlier in this chapter, we observe comparable behavior. As shown in Table 8.1, by holding y steady at 1 and gradually decreasing a from the near-ideal estimate of `0.9997` downward, we get exponential increases in cross-entropy cost. The table further illustrates that—again, consistent with the behavior of its quadratic cousin—cross-entropy cost would be low with an a of `0.1192` if y happened to in fact be `0`. These results reiterate for us that the chief distinction between the quadratic and cross-entropy functions is not the particular cost value that they calculate *per se*, but rather it is the rate at which they learn within a neural net—especially if saturated neurons are involved.

Table 8.1 Table of cross-entropy costs associated with selected example inputs

y	a	C
1	0.9997	0.0003
1	0.9	0.1
1	0.6	0.5
1	0.1192	2.1
0	0.1192	0.1269
1	1-0.1192	0.1269

OPTIMIZATION: LEARNING TO MINIMIZE COST

Cost functions provide us with a quantification of how incorrect our model's estimate of the ideal y is. This is most helpful because it arms us with a metric we can track if we'd like to reduce our network's incorrectness. And, we'd pretty well always like to reduce its incorrectness!

As alluded to a couple of times already in this chapter, the primary approach for minimizing cost in deep learning paradigms is to pair a method called gradient descent with another one called backpropagation. Together, these methods are *optimizers* that enable the network to *learn*. They accomplish this by adjusting the model's parameters so that its estimated $\hat{y}$ gradually converges toward the target of y, and thus the cost decreases. We'll cover gradient descent now and move on to backpropagation immediately afterward.

Gradient Descent

Gradient is a handy, efficient tool for adjusting a model's parameters with the aim of minimizing cost, particularly if we have a lot of data. It is widely used in other families of machine learning techniques as well, not only in deep learning.

In Figure 8.2, we've used a nimble trilobite in a cartoon to illustrate how gradient descent works. Along the horizontal axis in each frame is some parameter that we've denoted as p. In an artificial neural network, this parameter would be either a neuron's weight w or bias b. In the top frame, the trilobite finds itself on a hill. Its goal is to *descend* the gradient, thereby finding the location with the minimum cost, C. But, there's a twist: The trilobite is blind! It cannot see whether deeper valleys lie far away somewhere, it can only use its cane to investigate the slope of the terrain in its immediate vicinity.

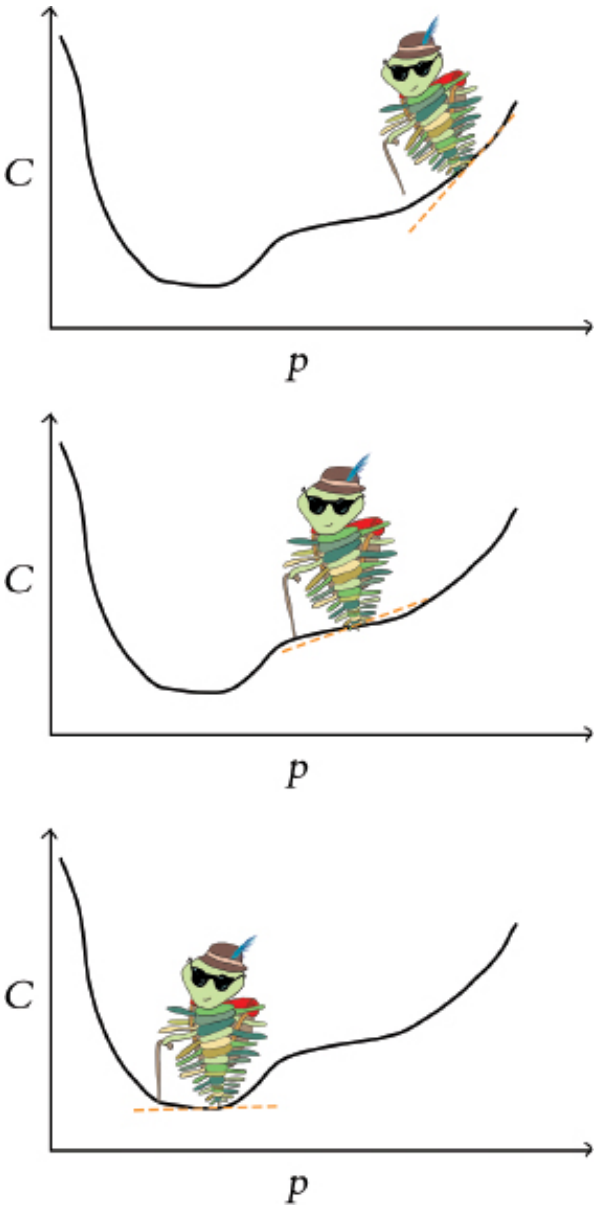

Figure 8.2 A trilobite using gradient descent to find the value of a parameter p associated with minimal cost, C.

The orange line in Figure 8.2 indicates the blind trilobite's calculation of the slope at the point that it finds itself. According to that slope line, if the trilobite takes a step to the left (i.e., to a slightly lower value of p), it would be moving to a location with smaller cost. On the hand, if the trilobite takes a step to the right (a slightly *higher* value of p), it would be moving to a location with *higher* cost. Given the trilobite's desire to descend the gradient, it chooses to take a step to the left.

By the second frame, the trilobite has taken several steps to the left. Here again, we see it evaluating the slope with the orange line and discovering that, yet again, a step to the left will bring it to a location with lower cost and so it takes another step left. In the third frame, the trilobite has succeeded in making its way to the location—i.e., the value of the parameter p—corresponding to the minimum cost. From this position, if it were to take a step to the left *or* to the right, cost would go up, so it gleefully remains in place.

In practice, a deep learning model would not have just one parameter in it. It is not

uncommon for deep learning networks to have millions of parameters and some industrial applications have billions of them. Even our *Shallow Net in Keras*—one of the smallest models we'll build in this book—has 50,890 parameters (Figure 7.5).

While it's impossible for the human mind to imagine a billion-dimensional space, the two-parameter cartoon shown in Figure 8.3 provides a sense of how gradient descent scales up to minimize cost across multiple parameters simultaneously. Across however many trainable parameters there are in a model, gradient descent iteratively evaluates slopes [5] to identify the adjustments to those parameters that correspond to the steepest reduction in cost. With two parameters, as in the trilobite cartoon in Figure 8.3 for example, this procedure can be likened to a blind hike through the mountains where:

œ latitude represents one parameter, say p_1

œ longitude represents the other parameter, p_2

œ altitude represents cost—the lower the altitude, the better!

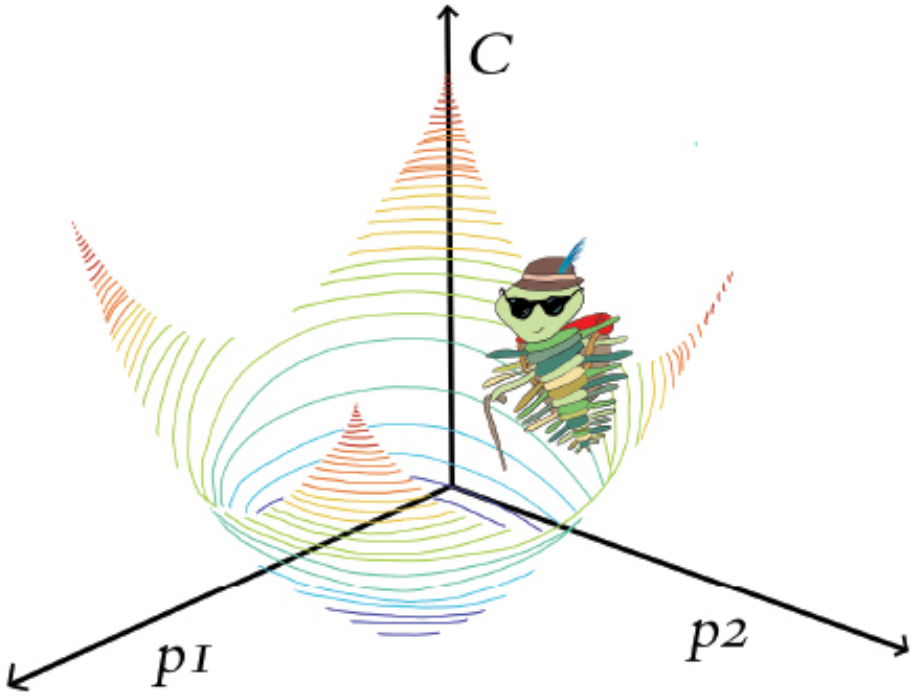

Figure 8.3 A trilobite exploring along two model parameters, p_1 and p_2, in order to minimize cost via gradient descent. In a mountain-adventure analogy, p_1 and p_2 could be thought of as latitude and longitude, while altitude represents cost.

The trilobite randomly finds itself at a location in the mountains. From that point, it identifies the direction of the step it can take that will reduce its altitude the most. It then takes that single step. Repeating this process many times, the trilobite may eventually find itself at the latitude and longitude coordinates that correspond to the lowest-possible altitude (minimum cost), at which point the trilobite's surreal alpine

adventure is complete.

Learning Rate

For conceptual simplicity, in Figure 8.4, let's return to a blind trilobite navigating a single-parameter world. Now, let's imagine that we have a ray-gun that can shrink or enlarge trilobites. In the middle panel, we've used our ray-gun to make our trilobite very small. The trilobite's steps will then be correspondingly small and so it will take our intrepid little hiker a very long time to find its way to the legendary valley of minimum cost. On the other hand, consider the bottom panel, in which we've used our ray-gun to make the trilobite very large. The situation here is even worse! The trilobite's steps will now be so large that it will step right over the valley of minimum cost and so it never has any hope of finding it.

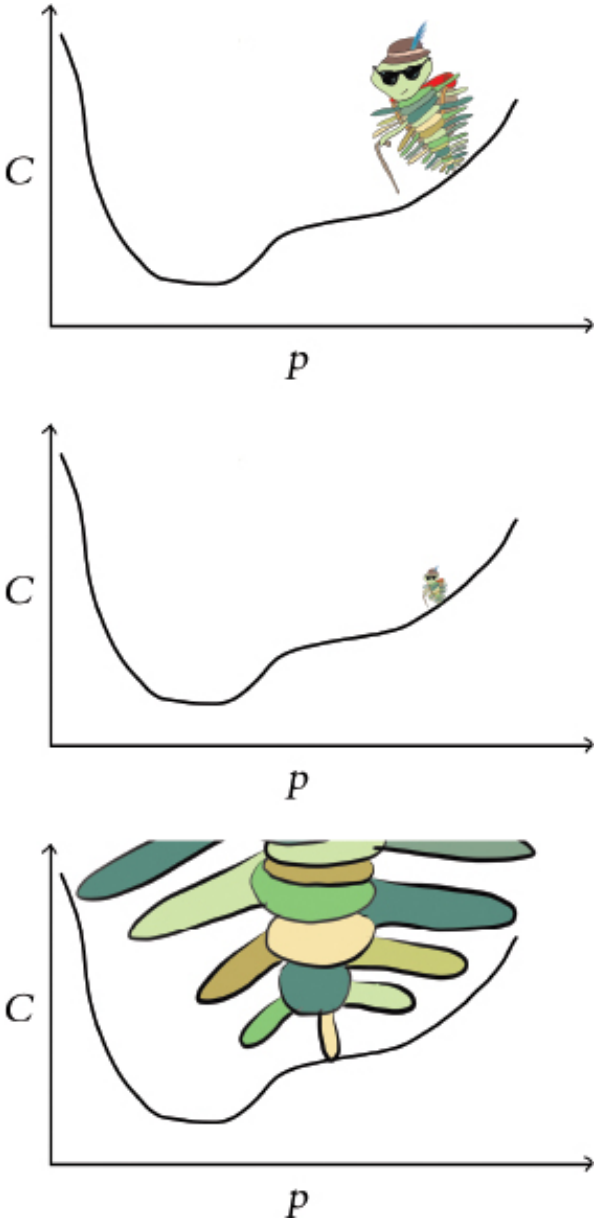

Figure 8.4 The learning rate (η) of gradient descent expressed as the size of a trilobite. The middle panel has a small learning rate and the bottom panel, a large one.

In gradient descent terminology, step size is referred to as *learning rate* and denoted with the Greek letter η (eta, pronounced "ee-ta"). Learning rate is the first of several model *hyperparameters* that we will cover in this book. In machine learning, including deep learning, hyperparameters are aspects of the model that we configure before we begin training the model. So, hyperparameters like η are pre-set while, in contrast, parameters—namely, w and b—are learned during training.

Getting your hyperparameters right for a given deep learning model often requires some trial and error. For the learning rate η, it's something like the fairy tale of *Goldilocks and the Three Bears*: too small and too large are both inadequate, but there's a sweet spot in the middle. More specifically, as we cartooned in Figure 8.4, if η is too small, then it will take many, many iterations of gradient descent (read: a *long time*) to reach the minimal cost. On the other hand, picking an η that is too large means we might never reach minimal cost at all: The gradient descent algorithm will act erratically as it zooms right over the parameters associated with minimal cost.

Coming up in Chapter 9, we have a clever trick waiting for you that will circumnavigate the need for you to pick a given neural network's η hyperparameter entirely. In the interim, however, here are our rules of thumb on the topic:

œ Begin with a learning rate of about 0.01 or 0.001.

œ If your model is able to learn (i.e., if cost decreases consistently epoch over epoch) but training happens very slowly (i.e., each epoch, the cost decreases only a small amount), then increase your learning rate by an order of magnitude (e.g., from 0.01 to 0.1). If the cost begins to jump up and down erratically epoch over epoch, then you've gone too far, so reign your learning rate back down.

œ At the other extreme, if your model is unable to learn, then it could be because your learning rate is too high. Try decreasing it by orders of magnitude (e.g., from 0.001 to 0.0001) until cost decreases consistently epoch over epoch.

Batch Size and Stochastic Gradient Descent

When we introduced gradient descent, we suggested that it is efficient for machine learning problems that involve a large data set. In the strictest sense, we outright lied to you. The truth is that if we have a very large data set, *ordinary* gradient descent would not work at all because it wouldn't be possible to fit all of the data into the memory (RAM) of our machine.

Memory isn't the only potential snag—compute power could cause us headaches too. A

relatively large data set might squeeze into the memory of our machine, but if we tried to train a neural network containing millions of parameters with all those data, vanilla gradient descent would be highly *in*efficient because of the computational complexity of the associated high-volume, high-dimensional calculations.

Thankfully, there's a solution to these memory and compute limitations: the *stochastic* variant of gradient descent. With this variation, we split up our training data into *mini-batches*—small subsets of our full training data set—to render gradient descent both manageable and productive.

Although we didn't focus on it at the time, when we trained the model in our *Shallow Net in Keras* notebook back in Chapter 5 we were already using stochastic gradient descent by setting our `optimizer` to `SGD` in the `model.compile()` step. Further, in the subsequent line of code when we called the `model.fit()` method, we set `batch_size` to `128` to specify the size of our mini-batches—the number of training data points that we use for a given iteration of SGD. Like the learning rate η presented earlier in this chapter, *batch size* is also a hyperparameter.

Let's work through some numbers to make the concepts of batches and stochastic gradient descent more tangible. In the MNIST data set, there are 60,000 training images. With a batch size of 128 images, we then have 468.75 = 469 batches [6] of gradient descent per epoch:

$$\begin{aligned} \text{number of batches} &= \left\lceil \frac{\text{size of training data set}}{\text{batch size}} \right\rceil \\ &= \left\lceil \frac{60,000 \text{ images}}{128 \text{images}} \right\rceil \\ &= \lceil 468.75 \rceil \\ &= 469 \end{aligned} \tag{8.4}$$

Before carrying out any training, we initialize our network with random values for each neuron's parameters w and b. [7] To begin the first epoch of training:

1. We shuffle and divide the training images into mini-batches of 128 images each. These 128 MNIST images provide 784 pixels each, which all together constitute the inputs x that are passed into our neural network. The *shuffling* step puts the *stochastic* into stochastic gradient descent.

2. By forward propagation, information about the 128 images is processed by the network, layer through layer, until the output layer ultimately produces $\hat{y}$ values.

3. A cost function (e.g., cross-entropy cost) evaluates the network's $\hat{y}$ values against the true y values, providing a cost C for this particular mini-batch of 128 images.

4. To minimize cost and thereby improve the network's estimates of y given x, the *gradient decent* part of stochastic gradient descent is performed: Every single w and b parameter in the network is adjusted proportional to how much each contributed to the error (i.e., the cost) in this batch (note that the adjustments are scaled by the learning rate hyperparameter η). [8]

The above four steps constitute a *round of training*, as summarized by Figure 8.5.

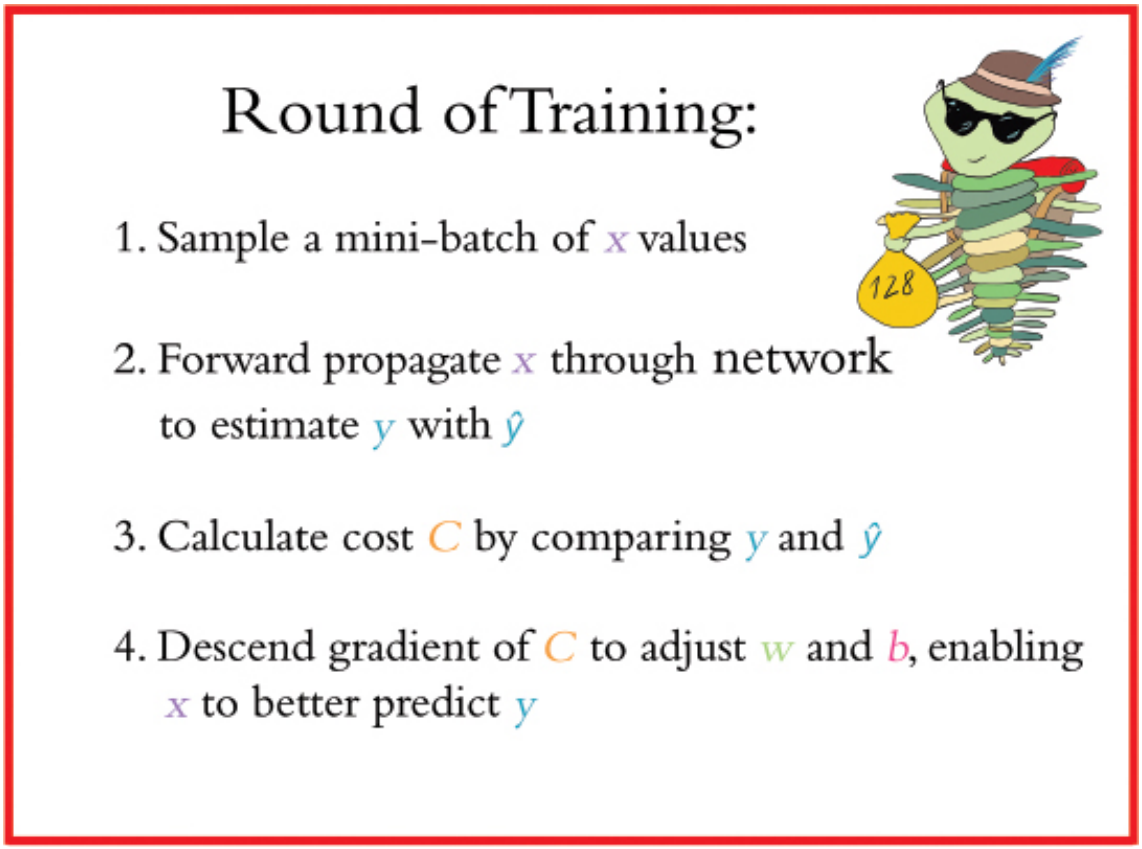

Figure 8.5 An individual round of training with stochastic gradient descent. Although mini-batch size is a hyperparameter that can vary, in this particular case, the mini-batch consists of 128 MNIST digits, as exemplified by our hike-loving trilobite carrying a small bag of data.

Figure 8.6 captures how rounds of training are repeated until we run out of training images to sample. The sampling in step one is done *without replacement*, meaning that at the end of an epoch each image has been seen by the algorithm only once, yet between different epochs the mini-batches are sampled randomly. After a total of 468 rounds, the last batch contains only 96 samples.

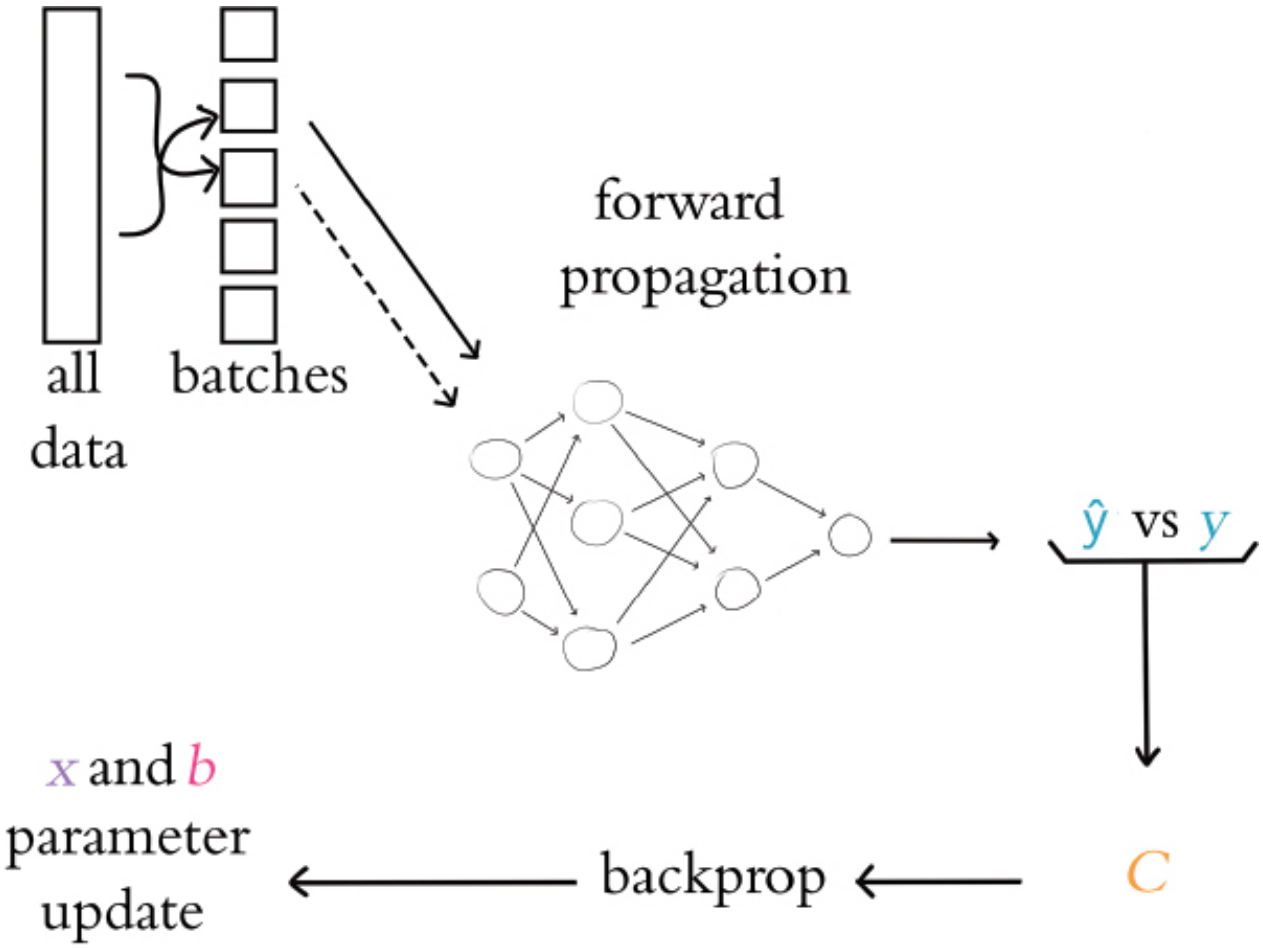

Figure 8.6 An outline of the overall process for training a neural network with stochastic gradient descent. The entire dataset is shuffled and split into batches. Each batch is forward propagated through the network, the output is compared to the ground truth and the cost is calculated, backpropagation calculates the gradients, and the parameters are updated. The next batch (indicated by a dotted line) is forward propagated, and so on until all of the batches have moved through the network. Once all the batches have been used, a single epoch is complete and the process starts again with a re-shuffling of the data.

This marks the end of the first epoch of training. Assuming we've set our model up to train for further epochs, we begin the next epoch by replenishing our pool with all 60,000 training images. As we did through the previous epoch, we then proceed through a further 468 rounds of stochastic gradient descent. [9] Training continues in this way until the total desired number of epochs is reached.

The total *number of epochs* that we set our network to train for is yet another hyperparameter, by the way. This hyperparameter, though, is one of the easiest to get right:

œ If the cost on your validation data is going down epoch over epoch, and your final epoch attained the lowest cost yet, then you can try training for additional epochs.

œ Once the cost on your validation data begins to creep upward, that's an indicator that your model has begun to *overfit* to your training data because you've trained for too

many epochs. (We'll elaborate much more on overfitting in Chapter 9.)

œ There are methods you can use to automatically monitor training and validation cost and stop training early if things start to go awry. In this way, you could set the number of epochs to be arbitrarily large and know that training will continue until the validation cost stops improving—and certainly before the model begins overfitting!

Escaping the Local Minimum

In all of the examples of gradient descent thus far in the chapter, our hiking trilobite has encountered no hurdles on its journey toward minimum cost. There are no guarantees that this would be the case, however. Indeed, such smooth sailing would be unusual.

Figure 8.7 shows the mountaineering trilobite exploring the cost of some new model that is designed for solving some new problem. With this new problem, the relationship between the parameter p and cost C is more complex. To have our neural network estimate y as accurately as possible, gradient descent needs to identify the parameter values associated with the lowest-attainable cost. However, as our trilobite makes its way from its random starting point in the top panel, gradient descent leads it to getting trapped in a *local minimum*. As shown in the middle panel, while in the local minimum a step to the left or a step to the right both lead to an increase in cost and so the blind trilobite stays put, completely oblivious to the existence of a deeper valley—the *global minimum*—lying yonder.

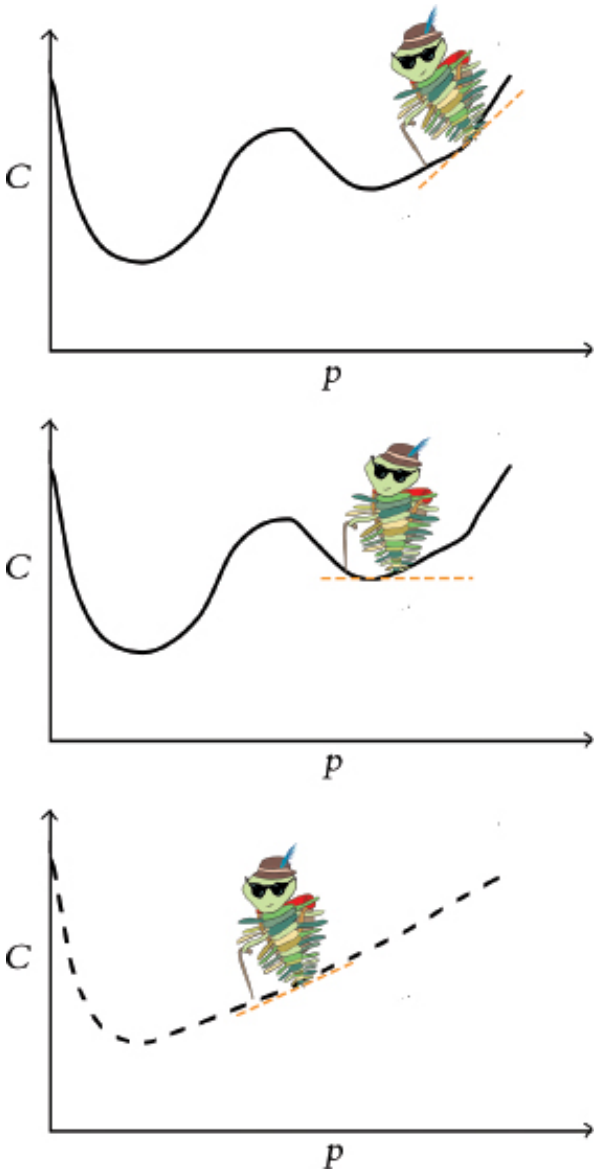

Figure 8.7 A trilobite applying vanilla gradient descent from a random starting point (top panel) is ensnared by a local minimum of cost (middle panel). By turning to stochastic gradient descent in the bottom panel, the daring trilobite is able to bypass the local minimum and make its way toward the global minimum.

All is not lost, friends, for stochastic gradient descent comes to the rescue here again. The sampling of mini-batches can have the effect of smoothing out the cost curve, as exemplified by the dotted curve shown in the bottom panel of Figure 8.7. This smoothing happens because when estimating the gradient from a smaller mini-batch (versus from the entire data set), the estimate is inherently noisier—while the actual gradient in the local minimum truly is zero, estimates of the gradient from small subsets of the data don't tell the whole picture and might give an *inaccurate* reading causing our trilobite to take a step thinking there is a gradient when there really isn't one. This is a good thing! The incorrect gradient may result in a step that is large enough for the trilobite to escape the valley and continue down the mountain. Thus, by estimating the gradient many times on these mini-batches, the noise of all of these gradient estimates is eventually smoothed and we are able to avoid local minima. Conversely, if the gradient were estimated from the entire data set[10] there would be no

noise. The trilobite would receive a noise-free reading of a zero gradient in that local minimum and would never know there was a whole world of cost-gains to be had just over the rise, and so this approach will get stuck in the first local minimum it fell into.[11] So, although each mini-batch on its own lacks complete information about the cost curve, in the long run—over a large number of mini-batches—this tends to work to our advantage.

Like the learning rate hyperparameter η, there is also a Goldilocks-style sweet spot for batch size. If the batch size is too large (perhaps even enveloping the entire data set), the estimate of the gradient of the cost function is far more accurate. In this way, the trilobite has a more complete image of the mountain in that moment and is able to take a step (proportional to η) in the direction of the steepest possible descent. However, the model is at risk of becoming trapped in local minima as we described above. Besides that, the model might not fit in memory on your machine and the compute time per iteration of gradient descent could be very long. On the other hand, if the batch size is too small, each gradient estimate will be noisier (since a very small subset of the data is being used to estimate the gradient of the entire data set) and the corresponding path down the mountain will be more circuitous—training will take longer because of these erratic gradient descent steps. Furthermore, you're not taking advantage of the memory and compute resources on your machine.[12] With that in mind, here are our rules of thumb for finding the batch-size sweet spot:

œ Start with a batch size of 32 or 64.

œ If the mini-batch is too large to fit into memory on your machine or if epochs of training proceed very slowly, try decreasing your batch size by powers of two (e.g., from 32 to 16).

œ If your model trains well (i.e., cost is going down consistently) but each epoch is taking very long and you are aware that you have RAM to spare,[13] try increasing your batch size by powers of two, (e.g., from 64 to 128).

BACKPROPAGATION

While stochastic gradient descent operates well on its own to adjust parameters and minimize cost in many types of machine learning models, for deep learning models in particular there is an extra hurdle: We need to be able to efficiently adjust parameters *through multiple layers* of artificial neurons. To do this, stochastic gradient descent is partnered up with a method called *backpropagation*.

Backpropagation—or backprop for short—is an elegant application of the "chain rule"

from calculus.[14] As shown along the bottom of Figure 8.6 and as suggested by its very name, backpropagation courses through a neural network in the opposite direction of forward propagation. While forward propagation carries information about the input x through successive layers of neurons to approximate y with $\hat{y}$, backpropagation carries information about the cost C *backwards* through the layers in reverse and, with the overarching aim of reducing cost, adjusts neuron parameters throughout the network.

While the nitty-gritty of backpropagation has been relegated to an appendix, it's worth understanding (in broad strokes) what the backpropagation algorithm does: As we've seen thus far, any given model is randomly initialized with network parameters (w and b values). Thus, at the very beginning of training when the first x value is fed in, the network essentially outputs a completely random guess at $\hat{y}$. Of course, this won't be a very good guess and the associated cost of the random guess will be high. At this point, we need to update the weights in order to minimize the cost—the very essence of learning in neural networks. Backpropagation calculates the *gradient* of the cost function with respect to each weight in the network. Recall from our mountaineering analogies earlier that the cost function represents a hiking trail and our trilobite is trying to reach basecamp. At each step along the way, the trilobite finds the gradient (or the slope) of the cost function and moves *down* that gradient. That movement that the trilobite just made is a weight update: By adjusting the weight in proportion to the cost function's gradient *with respect to that weight*, we essentially adjust that weight in a way that reduces the cost! We know that last sentence might be hard to digest at first, so hang with us. If you recall the most important equation from Figure 6.7 in Chapter 6 ($w \cdot x + b$), and you follow that neural networks are stacked and everything feeds together, it shouldn't be hard to imagine that any given weight in the network contributes to the final $\hat{y}$ output, and thus the cost. Using backpropagation, we move layer-by-layer backwards through the network, starting at the cost in the output layer, and we find the gradients of every single parameter. We then use the product of the gradient of that parameter—i.e., the relative *amount* which that parameter contributed to the total cost—and the learning rate η to update the parameter.

This is not the lightest section of this book, by a wide margin. Also, you wouldn't be the first deep learning practitioner who isn't able to sketch out the specifics of backpropagation on a whiteboard. So if there's only one thing you take away from this whole section, let it be this: Backpropagation uses the cost to calculate the relative contribution by every single parameter to the total cost, and then updates each parameter accordingly. In this way, the network slowly begins to reduce cost and, well… learn!

NETWORK DEPTH: TUNING HIDDEN-LAYER COUNT

As with learning rate and batch size, the number of hidden layers you add into your neural network is also a hyperparameter. And as with the previous two, there is yet again a Goldilocks sweet spot for your network's count of layers. Throughout this book, we've reiterated that with each additional hidden layer within a deep learning network, the more abstract the representations that the network can represent. That is the primary advantage of adding layers.

The *dis*advantage of adding layers is that backpropagation becomes less effective: As demonstrated by the plot of learning speed across the layers of a four-hidden-layer network in Figure 8.8, backprop is able to have its greatest impact on the parameters of the hidden layer of neurons closest to the output $\hat{y}$. The further a layer is away from $\hat{y}$ (where cost is calculated), the more diluted the effect of that layer's parameters becomes on the overall cost. This is because with more hidden layers there are simply more parameters, and the relative contribution of each parameter is diminished. Thus, the fourth layer, which is closest to the output $\hat{y}$, learns most rapidly because those weights will have larger gradients. In contrast, the second layer, which is three layers away from the cost calculation, learns about an order of magnitude more slowly than the final layer.

Here are our rules of thumb for selecting the number of hidden layers in your network:

œ The more abstract the ground-truth value y you'd like to estimate with your network is, the more helpful additional hidden layers may be. With that in mind, we recommend starting off with about two to four hidden layers.

œ If reducing the number of layers does not increase the cost you can achieve on your validation data set, then do it. Following the problem-solving principle called *Occam's razor*, the simplest network architecture that can provide the desired result is the best—it will train more quickly and require fewer compute resources.

œ On the other hand, if increasing the number of layers decreases the validation cost then you should layer away!

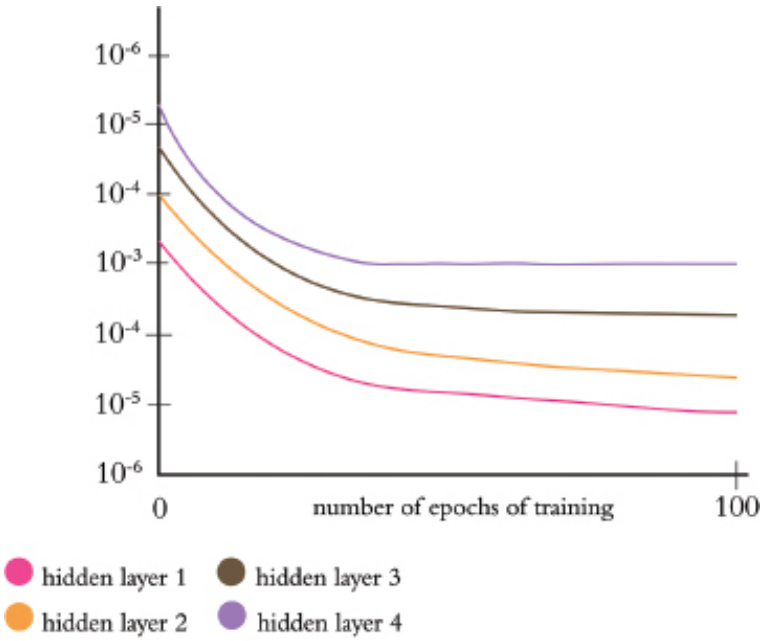

Figure 8.8 The speed of learning over epochs of training for a deep learning network with four hidden layers. The fourth layer, which is closest to the output $\hat{y}$ learns about an order of magnitude more quickly than the second hidden layer.

AN INTERMEDIATE NET IN KERAS

To wrap up this chapter, let's incorporate the new material we've uncovered into a neural network to see if we can outperform our previous *Shallow Net in Keras* model at classifying handwritten digits.

The first few stages of our *Intermediate Net in Keras* Jupyter notebook are identical to its *Shallow Net* predecessor. We load the same Keras dependencies, load the MNIST data set in the same way, and preprocess the data the same way. The situation begins to get interesting at the cell where we design our neural network architecture:

Example 8.1 Keras code to architect an intermediate-depth neural network

```
model = Sequential()

model.add(Dense(64, activation='relu', input_shape=(784,)))

model.add(Dense(64, activation='relu'))

model.add(Dense(10, activation='softmax'))
```

The first line of this code chunk, `model = Sequential()`, is still the same as before

(refer back to Example 5.2)—this is our instantiation of a neural network model object. It's in the second line that we begin to diverge. In it, we specify that we'll substitute the sigmoid activation function in the first hidden layer with our most-highly-recommended neuron from Chapter 6, the `relu`. Other than this neuron swap, the first hidden layer remains the same: It still consists of `64` neurons and the dimensionality of the `784`-neuron input layer is unchanged.

The other significant change in Example 8.1 relative to the shallow architecture of Example 5.2 is that we specify a second hidden layer of artificial neurons. By calling the `model.add()` method, we near-effortlessly add a second `Dense` layer of `64 relu` neurons, providing us with the notebook's namesake: an intermediate-depth neural network. With a call to `model.summary()`, we can see from Figure 8.9 that this additional layer corresponds to an additional 4160 trainable parameters relative to our shallow architecture (refer back to Figure 7.5). We can break these parameters down into:

- 4096 weights, corresponding to each of the 64 neurons in the second hidden layer densely receiving input from each of the 64 neurons in the first hidden layer (64 × 64 = 4096)
- plus 64 biases, one for each of the neurons in the second hidden layer
- giving us a total of 4160 parameters: $n_{parameters} = n_w + n_b = 4096 + 64 = 4160$

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 64)                50240
_________________________________________________________________
dense_2 (Dense)              (None, 64)                4160
_________________________________________________________________
dense_3 (Dense)              (None, 10)                650
=================================================================
Total params: 55,050
Trainable params: 55,050
Non-trainable params: 0
_________________________________________________________________
```

Figure 8.9 A summary of the model object from our "Intermediate Net in Keras" Jupyter notebook.

In addition to changes to the model architecture, we've also made changes to the parameters we specify when compiling our model:

Example 8.2 Keras code to compile our intermediate-depth neural network

```
model.compile(loss='categorical_crossentropy',
                          optimizer=SGD(lr=0.1),
                          metrics=['accuracy'])
```

With these lines from Example 8.2, we:

œ set our loss function to cross-entropy cost with `loss='categorical_crossentropy'` (in *Shallow Net in Keras*, we used quadratic cost by setting `loss='mean_squared_error'`)

œ set our cost-minimizing method to stochastic gradient descent with `optimizer=SGD`

œ specified our SGD learning rate hyperparametr η to `lr=0.1`[15]

œ indicated that, in addition to the Keras default of providing feedback on `loss`, by setting `metrics=['accuracy']`, we'd also like to receive feedback on model *accuracy*[16]

Finally, we train our intermediate net by running:

Example 8.3 Keras code to train our intermediate-depth neural network

```
model.fit(X_train, y_train,
                  batch_size=128, epochs=20,
                  verbose=1,
                  validation_data=(X_valid, y_valid))
```

Relative to the way we trained our shallow net (see code at Example 5.3), the only change we've made is reducing our `epochs` hyperparameter from `200` down by an order of magnitude to `20`. As we'll see next, our much-more-efficient intermediate architecture required far fewer epochs to train.

Figure 8.10 provides the results of the first three epochs of training the network. Recalling that our shallow architecture plateaued as it approached 86% accuracy on the validation dataset after 200 epochs, our intermediate-depth network is clearly superior: The `val_acc` field shows that we attained 92.34% accuracy *after a single epoch of training*. This accuracy climbs to over 95% by the third epoch and appears to plateau around 97.6% by the twentieth. My, how far we've come already!

```
Epoch 1/20
60000/60000 [==============================] - 1s 15us/step - loss: 0.4744 - acc: 0.8637 - va
l_loss: 0.2686 - val_acc: 0.9234
Epoch 2/20
60000/60000 [==============================] - 1s 12us/step - loss: 0.2414 - acc: 0.9289 - va
l_loss: 0.2004 - val_acc: 0.9404
Epoch 3/20
60000/60000 [==============================] - 1s 12us/step - loss: 0.1871 - acc: 0.9452 - va
l_loss: 0.1578 - val_acc: 0.9521
Epoch 4/20
60000/60000 [==============================] - 1s 12us/step - loss: 0.1538 - acc: 0.9551 - va
```

Figure 8.10 The performance of our intermediate-depth neural network over its first three epochs of training.

Breaking down the verbose `model.fit()` output shown in Figure 8.10 in a further detail:

œ The progress bar shown below fills in over the course of the 468 "rounds of training" (Figure 8.5):

```
60000/60000 [==============================]
```

œ `1s 15us/step` indicates that all 468 rounds in the first epoch required one second to train, at an average rate of 15 microseconds per round.

œ `loss` shows the average cost on our training data for the epoch. For the first epoch this is `0.4744` and, epoch over epoch, this cost is reliably minimized via stochastic gradient descent and backpropagation, eventually diminishing to `0.0332` by the 20th epoch.

œ `acc` is the classification accuracy on training data for the epoch. The model correctly classified 86.37% for the first epoch, increasing to over 99% by the twentieth. Because a model can easily overfit to the training data, one shouldn't be overly impressed by high accuracy on the training data.

œ Thankfully, our cost on the validation data (`val_loss`) set does generally decrease as well, eventually plateauing as it approaches 0.08 over the final five epochs of training.

- Corresponding to the decreasing cost of the validation data is a corresponding increase in accuracy (`val_acc`). As mentioned in the previous paragraph, validation accuracy plateaued at about 97.6%, which is a vast improvement over the 86% of our shallow net.

SUMMARY

We covered a lot of ground in this chapter. Starting from an appreciation of how a neural network with fixed parameters processes information, we developed an understanding of the cooperating methods—cost functions, stochastic gradient descent, and backpropagation—that enable network parameters to be learned so that we can approximate any y that has a continuous relationship to some input x. Along the way, we introduced several network hyperparameters, including learning rate, mini-batch size, and number of epochs of training—as well as our rules of thumb for configuring each of these. The chapter concluded by applying our new-found knowledge to a develop an intermediate-depth neural network that greatly outperformed our previous, shallow network on the same handwritten-digit-classification task. Up next, we have techniques for improving the stability of artificial neural networks as they deepen, enabling us to architect and train a bonafide deep learning model for the first time.

KEY CONCEPTS

Here are the essential foundational concepts thus far. New terms from the current chapter are highlighted in purple:

- parameters:
  - weight w
  - bias b
- activation a
- artificial neurons:
  - sigmoid
  - *tanh*
  - ReLU
- input layer

- hidden layer
- output layer
- layer types:
  - dense (fully-connected)
  - softmax
- cost (loss) functions:
  - quadratic (mean squared error)
  - cross-entropy
- forward propagation
- backpropagation
- optimizers:
  - stochastic gradient descent
- optimizer hyperparameters:
  - learning rate η
  - batch size

1. `9.0e-08` is equivalent to 9.0×10^{-8}

2. Recall from Chapter 6 that $a = \sigma(z)$, where σ is some activation function—in this example, the *tanh* function.

3. More methods for attenuating saturated neurons and their negative effects on a network will be covered in Chapter 9.

4. This footnote is a Trilobite-Reading SIDEBAR. To understand how the cross-entropy cost function in Equation 8.2 enables a neuron with larger cost to learn more rapidly, we require a touch of partial-derivative calculus. (Since we endeavor to

minimize the use of advanced mathematics in this book, we've relegated this calculusfocused explanation to this sidebar.) Central to the two computational methods that enable neural networks to learn—gradient descent and backpropagation—is the comparison of the rate of change of cost C relative to neuron parameters like weight w. Using partial-derivative notation, we can represent these relative rates of change as $\frac{\partial C}{\partial w}$. The cross-entropy cost function is deliberately structured so that, when we calculate its derivative, $\frac{\partial C}{\partial w}$ is related to $(\hat{y} - y)$. Thus, the larger the difference between the ideal output y and the neuron's estimated output $\hat{y}$, the greater the rate of change of cost C with respect to weight w.

5 . Using partial-derivative calculus

6 . Since 60,000 is not perfectly divisible by 128, that 469th batch would only contain $0.75 \times 128 = 96$ images.

7 . We'll detail parameter initialization with random values in Chapter 9.

8 . This error-proportional adjustment is calculated during backpropagation. We haven't covered backpropagation explicitly yet, but it's coming up in the next section so hang on tight!

9 . Because we're sampling randomly, the order in which we select training images for our 468 mini-batches is completely different for every epoch.

10. This is often not even a possibility due to memory constraints.

11. It's worth noting that the learning rate η plays a role here. If the size of the local minimum was *smaller* than the step size, the trilobite would likely breeze right past the local minimum just as we step over cracks in the sidewalk.

12. A batch-size of one is also known as *online learning*. It's worth noting that this is *not* the fastest method in terms of compute - as it happens, the matrix multiplications in mini-batches are highly optimized and so training is several orders of magnitude faster when using mini-batches as compared to online learning.

13. On a Unix-based operating system, including Mac OS, RAM usage could be assessed by running the `top` or `htop` command within a Terminal window.

14. To elucidate the mathematics underlying backpropagation, a fair bit of partial-

derivative calculus is necessary. While we encourage the development of an in-depth understanding of the beautiful phenomenon of backprop, we also appreciate that calculus might not be the most appetizing topic for everyone. As such, we've placed our content on backprop mathematics in Appendix 18. (This should presumably be labelled Appendix B)

15. On your own time, you can play around with increasing this learning rate by several orders of magnitude as well as decreasing it by several orders magnitude, and observing how it impacts training.

16. Trilobite-Attention Sidebar: While loss provides the most important metric for tracking a model's performance epoch over epoch, its particular values are specific to the characteristics of a given model and are not generally interpretable or comparable between models. Because of this, other than knowing that we would like our loss to be as close to zero as possible, it can be esoteric to interpret how close to zero loss should be for any particular model. Accuracy, on the other hand, is highly interpretable and highly generalizable: We know exactly what it means (e.g., "the shallow neural network correctly classified 86% of the handwritten digits in the validation dataset") and we can compare this classification accuracy to any other model ("86% is worse than the performance of our deep neural network").

9 Improving Deep Networks

In Chapter 6, we detailed individual artificial neurons. In Chapter 7, we arranged these neural units together as the nodes of a network, enabling the forward propagation of some input x through the network to produce some output $\hat{y}$. Most recently, in Chapter 8, we described how to quantify the inaccuracies of a network (compare $\hat{y}$ to the true y with a cost function) as well as how to minimize these inaccuracies (adjust the network parameters w and b via optimization with stochastic gradient descent and backpropagation). In this chapter, we'll cover common barriers to the creation of high-performing neural networks and techniques that overcome them. We'll apply these ideas directly in code while architecting our first deep neural network. [1] Combining this additional network depth with our new-found best-practices, we'll see if we can outperform the handwritten-digit classification accuracy of our simpler, shallower architectures from previous chapters.

WEIGHT INITIALIZATION

Back in Chapter 8, we introduced the concept of neuron saturation (see Figure 8.1), where very low or very high values of z diminish the capacity for a given neuron to learn. At the time, we offered cross-entropy cost as a solution. While cross-entropy does effectively attenuate the effects of neuron saturation, pairing it with thoughtful *weight initialization* will reduce the likelihood of saturation occurring in the first place. As mentioned in a footnote in Chapter 1, modern weight initialization provided a significant leap forward in deep learning capability: It is one of the landmark theoretical advances between LeNet-5 (Figure 1.12) and AlexNet (Figure 1.18) that dramatically broadened the range of problems artificial neural networks could reliably solve. In this section, we'll play around with several weight initializations to develop an intuition around how they're so impactful.

While describing neural-network training in Chapter 8, we mentioned that the parameters w and b are initialized with random values such that a network's starting approximation of y will be far off the mark, thereby leading to a high initial cost C. We haven't needed to dwell on this much because, in the background, Keras by default constructs TensorFlow models that are initialized with sensible values for w and b. In

Chapters 14 and 15, we'll get our hands dirty with raw TensorFlow and PyTorch code, at which point we'll explicitly need to make decisions about parameter initialization ourselves. Even now however, it's worthwhile discussing this initialization: not only to be aware of another method for avoiding neuron saturation, but also to fill in a gap in our understanding of how network training works. While Keras does a sensible job of choosing default values—and that's a key benefit of using Keras in the first place—it's certainly possible, and sometimes even necessary, to change these defaults to suit your problem.

To make this section interactive, we encourage you to check out our accompanying Jupyter notebook, *First TensorFlow Neurons*. This marks our inaugural foray into TensorFlow code, but we'll save the details for Chapter 14, skimming over them for the moment.

As shown in the upcoming chunk of code, our library dependencies are NumPy (for numerical operations), matplotlib (for generating plots) and, as promised, TensorFlow:

```
import numpy as np

import tensorflow as tf

import matplotlib.pyplot as plt
```

In this notebook, we're going to simulate 784 pixel values as inputs to a single dense layer of artificial neurons. The inspiration behind our simulation of these 784 inputs comes of course from our beloved MNIST digits (Figure 5.3). For the number of neurons in the dense layer, we picked a number large enough so that, when we make some plots later on, they look pretty:

```
n_input = 784

n_dense = 256
```

When Keras creates TensorFlow networks for us, it handily generates all of the components of the network, including arrays of data for storing all of the relevant network values. As we'll detail in Chapter 14, these arrays are called *tensors*. Without Keras doing the heavy lifting for us, we'll have to initialize all the relevant tensors ourselves. We begin by creating a tensor for holding our 784 input values:

```
x = tf.placeholder(tf.float32, [None, n_input])
```

Now, for the primary point of this section: the initialization of the network parameters w and b. Before we begin passing training data into our network, we'd like to start with reasonably scaled parameters. This is because:

1. Large w and b values tend to correspond to larger z values, and therefore saturated neurons.

2. Large parameter values would imply that the network has a strong opinion about how x is related to y—before any training on data has occurred, any such strong opinions are wholly unmerited.

Parameter values of zero, on the other hand, imply the weakest opinion on how x is related to y. To bring back the fairytale yet again, we're aiming for a Goldilocks-style, middle-of-the-road approach that starts training off from a balanced and *learnable* beginning. With that in mind, let's use the TensorFlow `zeros()` method to initialize the 256 neurons in our dense layer with $b = 0$:

```
b = tf.Variable(tf.zeros([n_dense]))
```

Following the line of thinking from the previous paragraph to its natural conclusion, we might be tempted to think that we should also initialize our network weights w with zeros as well. In fact, this would be a training disaster: If all weights and biases were identical, many neurons in the network would treat a given input x identically, giving stochastic gradient descent a minimum of heterogeneity for identifying individual parameter adjustments that might reduce the cost C. It would be more productive to initialize weights with a range of different values so that neurons treat a given x in unique ways, thereby providing SGD with a wide variety of starting points for approximating y. By chance, some of the initial neuron outputs may partly contribute to a sensible mapping from x to y. While this contribution will be weak at first, SGD can experiment with it to determine if it might contribute to a reduction in the cost C between the predicted $\hat{y}$ and the target y.

As worked through earlier (e.g., in discussion of Figures 7.5 and 8.9), the vast majority of the parameters in a typical network are weights; relatively few are biases. As such, it's acceptable (indeed, it's the most common practice) to initialize biases with zeros

and the weights with a range of values *near* zero. One straightforward way to generate random values near zero is to use TensorFlow's `random_normal()` method to sample values from a normal [2] distribution like so:

Example 9.1 Weight initialization with values sampled from a normal distribution

```
W = tf.Variable(tf.random_normal([n_input, n_dense]))
```

To observe the impact of the weight initialization we've chosen, we write some code to represent our dense layer of neurons:

Example 9.2 Code for calculating the output of a layer of neurons

```
z = tf.add(tf.matmul(x, W), b)

a = tf.sigmoid(z)
```

If you decompose the first line, you can see that it is our "most important equation" (Figure 6.7), $z = w \cdot x + b$:

- `tf.matmul(x, W)` uses the TensorFlow matrix multiplication operation [3] to calculate the dot product $w \cdot x$
- `tf.add()` adds b to that product, returning us z

Simply beautiful, isn't it? In the second line of Example 9.2, we go on to apply whatever activation function we fancy—in this case the `sigmoid()` function—to `z`, giving us the neuron activation `a`. Since these activation functions are such a core part of deep learning, TensorFlow has implemented many of them within the library (recall how we defined the `sigmoid()` function in the *Sigmoid Function* Jupyter notebook in Chapter 6), and they're optimized to help speed up the compute time!

We won't explore the details until Chapter 14, but in the following lines of code we use the NumPy `random()` method [4] to feed 784 random numbers as inputs into our dense layer of 256 neurons, returning 256 a activations to a variable named `layer_output`:

```
initializer_op = tf.global_variables_initializer()
```

```
with tf.Session() as session:
    session.run(initializer_op)
    layer_output = session.run(a,
        {x: np.random.random([1, n_input])})
```

With one more line of code, we use a histogram to visualize the a values stored in `layer_output`: [5]

```
_ = plt.hist(np.transpose(layer_output))
```

Your result will look slightly different from ours because of the random() method we used to generate our input values, but your outputs should look approximately like those shown in Figure 9.1.

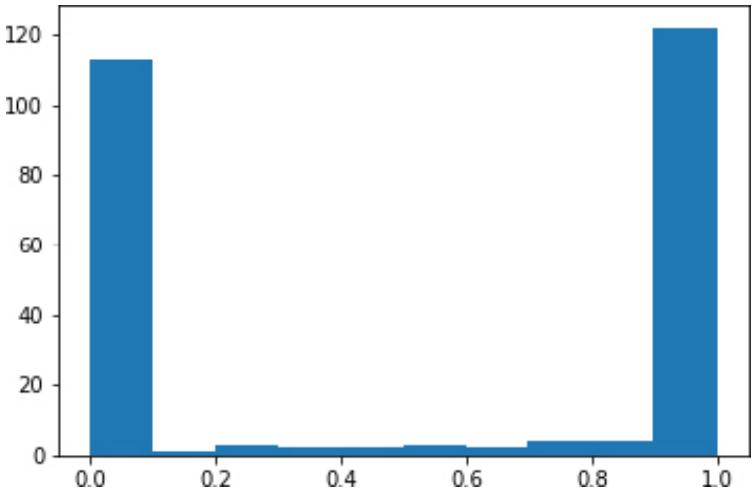

Figure 9.1 Histogram of the a activations output by a layer of sigmoid neurons, with weights initialized using a normal distribution.

As expected given Figure 6.9, the a activations output from our sigmoid layer of neurons is constrained to a range from zero to one. What is undesirable about these activations, however, is that they are chiefly pressed up against the extremes of the range: Most of them are either immediately adjacent to 0 or immediately adjacent to 1. This indicates that with the normal distribution that we sampled from to initialize the layer's weights w, we ended up encouraging our artificial neurons to produce large z values. This is unwelcome for two reasons:

1. It means the vast majority of the neurons in the layer are saturated.

2. It implies that the neurons have strong opinions about how x would influence y *prior* to any training on data.

Thankfully, this ickiness can be resolved by initializing our network weights with values sampled from alternative distributions.

Xavier Glorot Distributions

In deep-learning circles, popular distributions for sampling weight-initialization values were devised by Xavier Glorot and Yoshua Bengio [6] (portrait provided in Figure 1.11). These *Glorot distributions*, as they are typically called, are tailored such that sampling from them will lead to neurons initially outputting small z values. Let's examine them in action. By replacing the normal-distribution-sampling code (Example 9.1) of our *First TensorFlow Neurons* notebook with the following line, we sample from a Glorot distribution instead:

Example 9.3 Weights initialization with values sampled from a Glorot distribution

```
W = tf.get_variable('W', [n_input, n_dense],
                    initializer=tf.contrib.layers.xavier_initializer())
```

By restarting and re-running the notebook, [7] you should now observe a distribution of `layer_output` similar to the histogram shown in Figure 9.2.

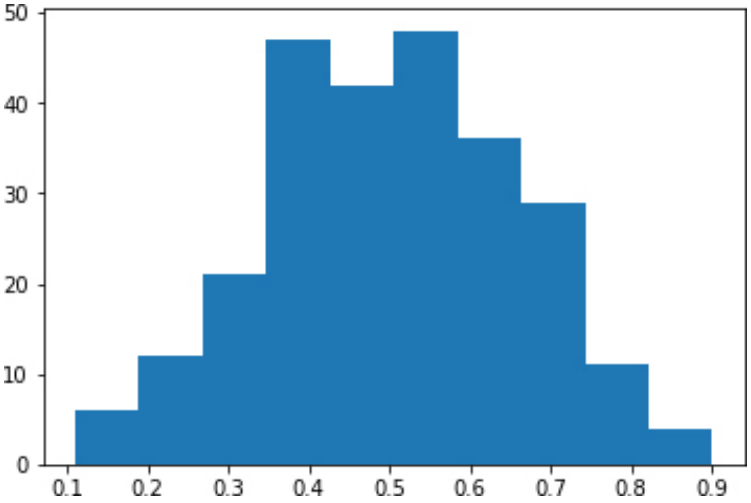

Figure 9.2 Histogram of the a activations output by a layer of sigmoid neurons, with weights initialized using a Glorot distribution.

In stark contrast to Figure 9.1, the a activations obtained from our layer of sigmoid neurons is now normally distributed with a mean of ~0.5 and few (if any) values at the extremes of the sigmoid range (i.e., less than 0.1 or greater than 0.9). This is a good starting point for a neural network because:

1. Few, if any, of the neurons are saturated. [8]

2. It implies that the neurons generally have weak opinions about how x would influence y, which—prior to any training on data—is sensible.

This paragraph is a Trilobite-attention SIDEBAR: As demonstrated in this section, one of the potentially confusing aspects of weight initialization is that, if we would like the a values returned by a layer of artificial neurons to be normally distributed (and we do!), we should *not* sample our initial weights from a standard normal distribution. END SIDEBAR.

There are two Glorot distributions you can select between: Glorot *uniform* and Glorot *normal*. By using the TensorFlow `xavier_initializer()` method in Example 9.3, we were using the default option, which is Glorot uniform. On the other hand, by setting the method's `uniform` parameter to false—as in, `xavier_initializer(uniform=False)`—we are opting to sample values from the Glorot normal distribution. The impact of selecting one of these Glorot distributions over the other when initializing your model weights is generally imperceptible. You're welcome to re-run the notebook while sampling values from the Glorot normal distribution; your histogram of activations should come out more or less indistinguishable from Figure 9.2.

By swapping out the sigmoid activation function in Example 9.2 with *tanh* `(a = tf.tanh(z))` or ReLU `(a = tf.nn.relu(z))` in the *First TensorFlow Neurons* notebook, you can observe the consequences of initializing weights with values sampled from a standard normal distribution (Figure 9.1) relative to a Glorot distribution (Figure 9.2). Regardless of activation function, weight initialization with the standard normal leads to relatively extreme a activations from the layer of dense neurons, as shown in Figure 9.3.

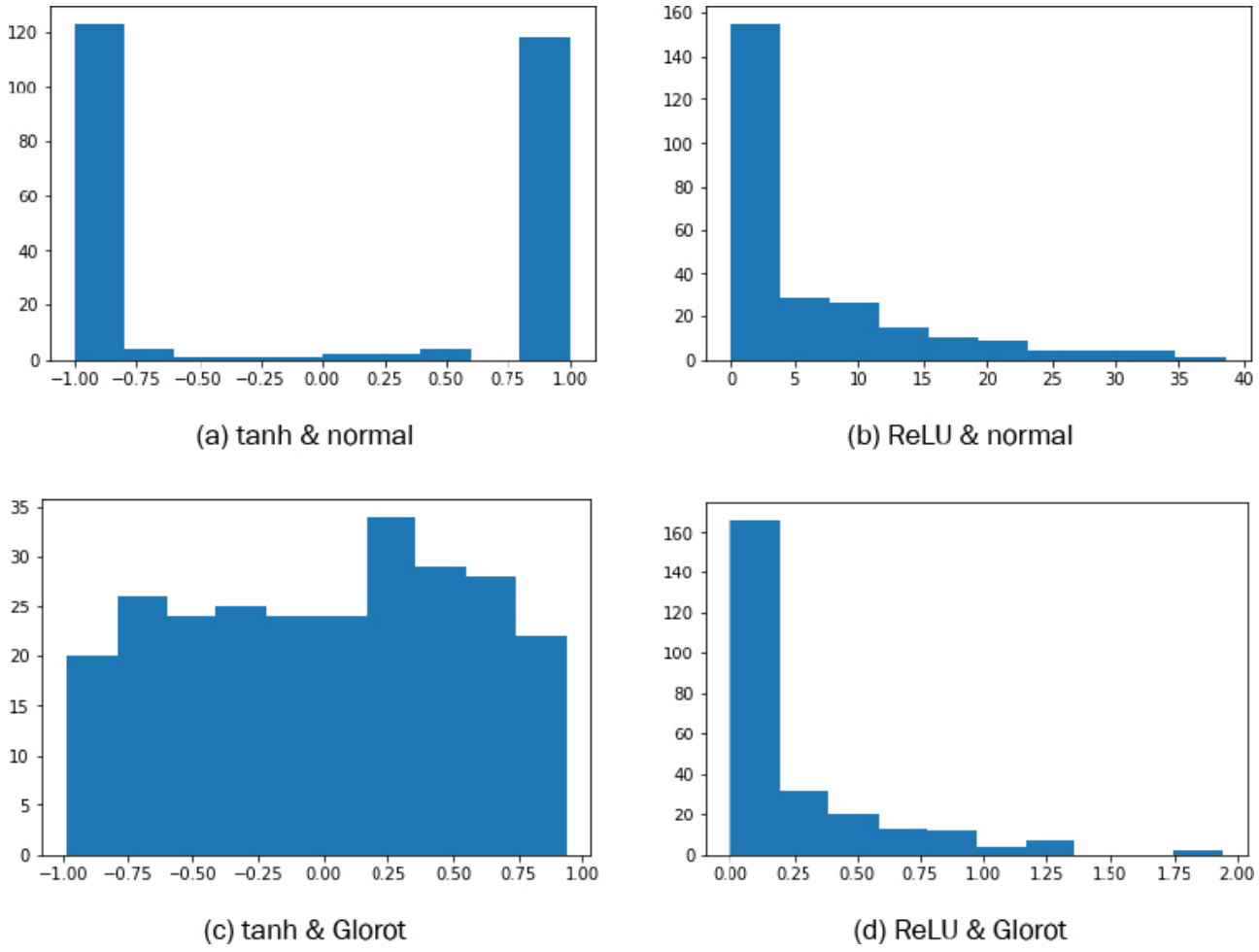

Figure 9.3 The activations output by a dense layer of 256 neurons, while varying activation function (tanh or ReLU) and weight initialization (standard normal or Glorot uniform)

We hope the exposure to TensorFlow in this section instilled a sense of eager anticipation for what's to come in Chapter 14. With respect to parameter initialization in Keras, you can delve into the API's documentation on a layer-by-layer basis but, just as we've suggested here, its default configuration is typically to initialize biases with zero and to initialize weights with a Glorot distribution.

UNSTABLE GRADIENTS

Another issue associated with artificial neural networks, and one that becomes especially perturbing as we add more hidden layers, is *unstable gradients*. Unstable gradients can either be vanishing or explosive in nature. We'll cover both varieties in turn here.

Vanishing Gradients

Recall that using the cost C between the network's predicted $\hat{y}$ and the true y, as diagrammed in Figure 8.6, backpropagation works its way from the output layer toward the input layer, adjusting network parameters with the aim of minimizing cost. As exemplified by the mountaineering trilobite in Figure 8.2, the parameters are each

adjusted in proportion to their *gradient* with respect to cost: If, for example, the gradient of a parameter (*with respect to the cost*) was large and positive, this implies that the parameter contributes a *large* amount to the cost and so *decreasing* it proportionally would correspond to a decrease in cost. [9]

In the hidden layer that is closest to the output layer, the relationship between its parameters and cost is the most direct. The further away a hidden layer is from the output layer, the more muddled the relationship between its parameters and cost becomes. The impact of this is that, as we move from the final hidden layer toward the first hidden layer, the gradient of a given parameter relative to cost tends to flatten—it gradually *vanishes*. As a result of this, and as plotted in Figure 8.8, the further a layer is from the output layer, the more slowly it tends to learn. Because of the *vanishing gradient* problem, if we were to naïvely add more and more hidden layers to our neural network, eventually the hidden layers furthest from the output would not be able to learn to any extent, crippling the capacity for the network as a whole to learn to approximate y given x.

Exploding Gradients

While they occur much less frequently than vanishing gradients, certain network architectures can induce *exploding gradients*. In this case, the gradient between a given parameter relative to cost becomes increasingly *steep* as we move from the final hidden layer toward the first hidden layer. As with vanishing gradients, exploding gradients can inhibit an entire neural network's capacity to learn by saturating the neurons with extreme values (recall that this was a problem from our discussion about weights initialization).

Batch Normalization

During the course of normal training, the distribution of hidden parameters in a layer may gradually move around—this is known as *internal covariate shift*. In fact, this is sort of the point—we want the parameters to change in order to learn things about the underlying data. But as the distribution of the weights in a layer changes, so the inputs to the next layer might be *shifted* away from an ideal distribution. Enter *batch normalization* (or *batch norm* for short).[10] Batch norm takes the a activations output from the previous layer and subtracts the batch mean and divides by the batch standard deviation. This acts to re-center the distribution of the a values with a mean of 0 and a standard deviation of 1 (Figure 9.4). Thus, if there are any extreme values in the previous layer, they won't cause exploding or vanishing gradients in the next layer. Batch norm has a few advantages:

œ It allows layers to learn more independently from each other, since large values in one layer won't adversely influence the next layer.

œ It allows for selection of a higher learning rate because there are no extreme values in the normalized activations, thus enabling faster learning.

œ The layer outputs are normalized to the *batch* mean and standard deviation, which adds a noise element (especially with smaller batch sizes) which, in turn, contributes to regularization. (Regularization will be covered in the next section, but suffice to say here that regularization helps a network generalize, which is a good thing.)

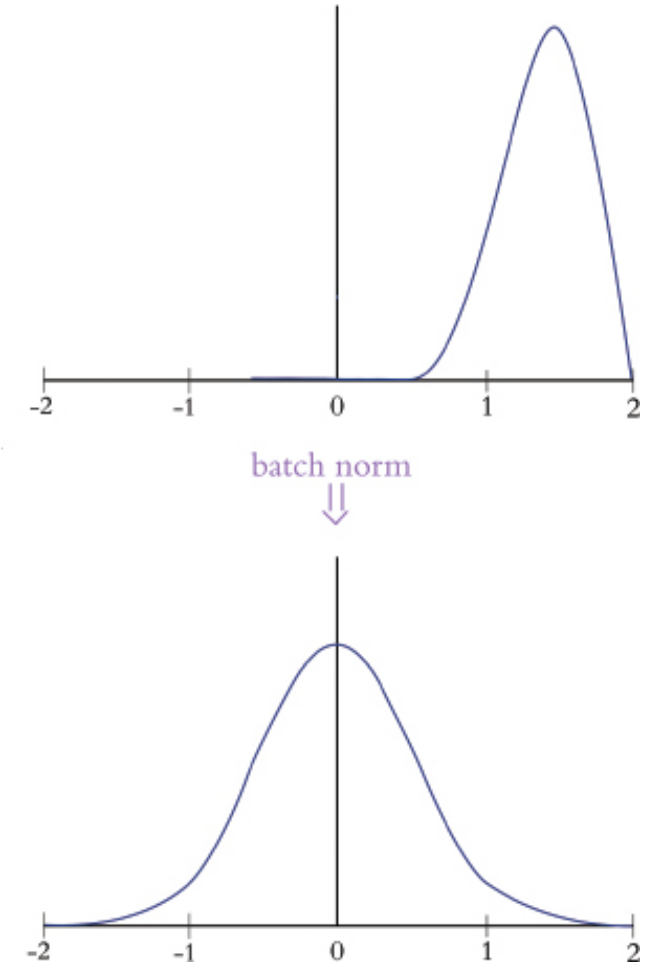

Figure 9.4 Batch normalization transforms the distribution of the activations output by a given layer of neurons toward a standard normal distribution.

Another point to consider with batch norm is that it adds two extra learnable parameters to the normalized layers: γ (gamma) and β (beta). In the final step of batch norm, the outputs are linearly transformed by multiplying by γ and adding β, where γ is analogous to the standard deviation, and β to the mean. If your math is on point, you'll notice this is the exact inverse of the operation that normalized the output values in the first place! However, the output values were originally normalized by the *batch* mean and *batch* standard deviation, whereas these two parameters are learned by SGD. We initialize the batch norm layer with $\gamma = 1$ and $\beta = 0$, thus at the start of training this

linear transformation makes no changes—batch norm is allowed to normalize the outputs as intended. As the network learns though, it may determine that *de-normalizing* any given layer's activations is optimal for reducing cost. In this way, if batch norm is not helpful the network will *learn* to stop using it on a layer-by-layer basis. In fact, the network can decide *to what degree* it would like to de-normalize the outputs, depending on what works best to minimize the cost. Pretty neat!

MODEL GENERALIZATION (AVOIDING OVERFITTING)

In Chapter 8, we mentioned that after training a model for a certain number of epochs the cost calculated on the validation dataset—which may have been decreasing nicely over earlier epochs—could begin to increase paradoxically, despite the fact that the cost calculated on the *training* dataset is still decreasing! This situation—where training cost continues to go down while validation cost goes up—is formally known as *overfitting*.

Overfitting is nicely illustrated in Figure 9.5. Notice we have the same data points scattered along x and y axes in each panel. We can imagine that there is some distribution that describes these points, and here we have a sampling from that distribution. Our goal is to generate a model that explains the relationship between x and y, but perhaps most importantly that also *approximates* the original distribution—in this way, the model will be able to generalize to new data points drawn from the distribution and not just model the sampling of points we already have. In the first panel (top left), we use a single-parameter model, which is limited to fitting a straight line to the data.[11] This straight line *underfits* the data: The cost (represented by the vertical gaps between the line and the data points) is high and the model would not generalize well to new data points. Put simply, the line *misses* most of the points because this kind of model is not complex enough. In the next panel (top right), we use a model with two parameters, which fits a parabola-shaped curve to the data.[12] With this parabolic model, the cost is much lower relative to the linear model and it appears the model would also generalize well to new data—great!

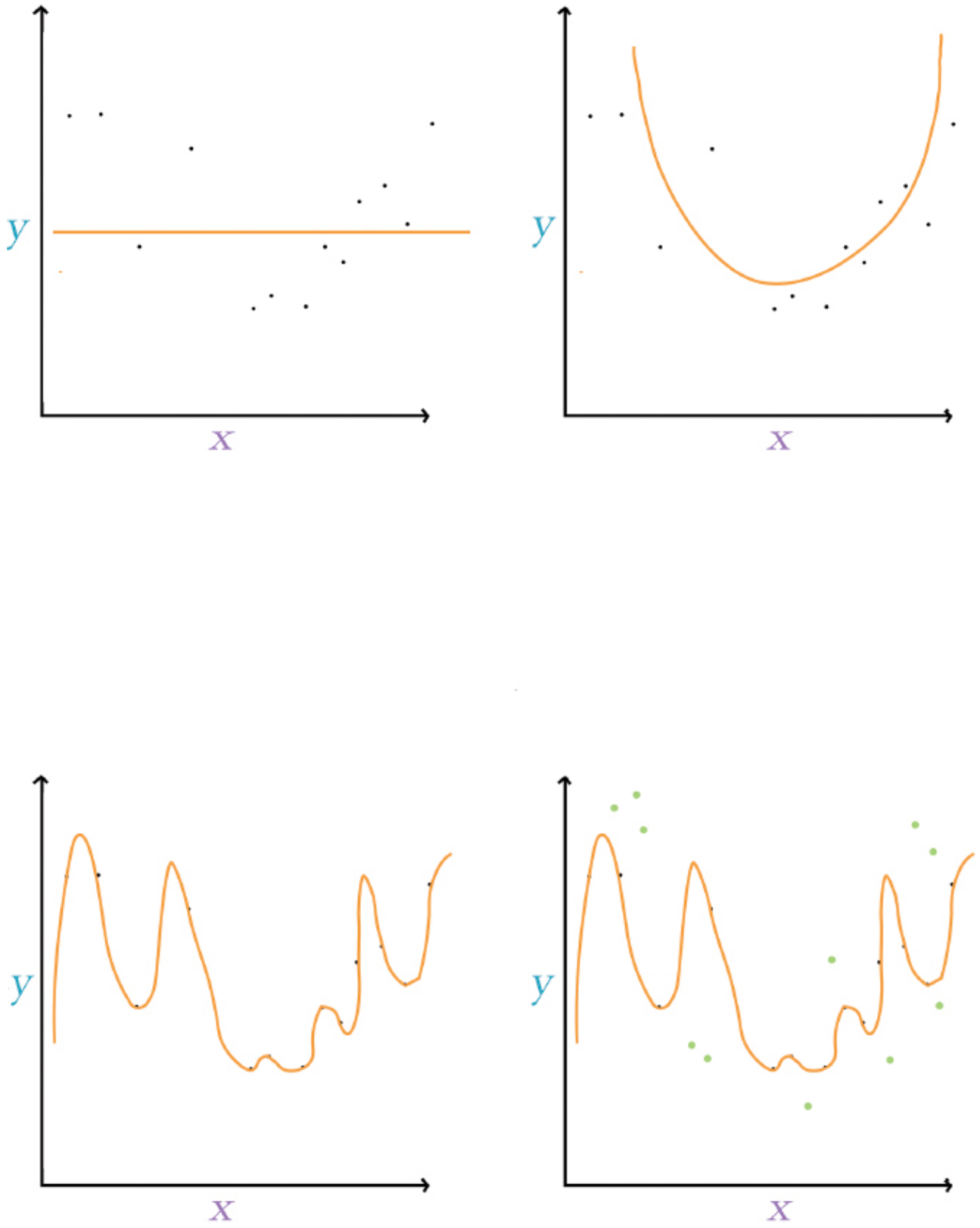

Figure 9.5 Fitting y given x using models with varying numbers of parameters. Top right: A single-parameter model underfits the data. Top left: A two-parameter model fits a parabola that suits the relationship between x and y well. Bottom left: A many-parameter model overfits the data, generalizing poorly to new data points (shown in green in the bottom-right panel).

In the third panel (bottom left) of Figure 9.5, we use a model with too many parameters —more parameters than we have data points. With this approach we reduce the cost associated with our training data points to nil: There is no perceptible gap between the curve and the data. In the last panel (bottom right), however, we show new data points

from the original distribution in green, which were unseen by the model during training and so can be used to validate the model. Despite eliminating *training* cost entirely, the model fits these validation data poorly and so it gets a correspondingly sizeable validation cost. The many-parameter model, dear friends, is overfit: It is a perfect model for the training data, but it doesn't actually capture the true relationship between x and y—rather, it has learned the exact features of the training data too closely and subsequently it performs badly on unseen data.

Consider how in three lines of code in Example 5.2, we created a shallow neural network architecture with over 50,000 parameters (Figure 7.5). Given this, it should not be surprising that deep learning architectures regularly have millions of parameters.[13] This hints at why deep learning models typically require large amounts of data: Working with data sets that may only have thousands of training samples but millions of parameters[14] could be a recipe for severe overfitting. Since we yearn to capitalize on deep, sophisticated network architectures even if we don't have oodles of data at hand, thankfully we can rely on techniques specifically designed to reduce overfitting. We'll cover three of the best-known such techniques now.

L1 and L2 Regularization

In branches of machine learning other than deep learning, the use of *L1 regularization* or *L2 regularization* to reduce overfitting is prevalent. These techniques—which are alternately known as *ridge regression* and *LASSO*[15] *regression*, respectively—both *penalize* models for including parameters by adding the parameters to the model's cost function. The larger a given parameter's size, the more that parameter adds to the cost function. Because of this, parameters will not be retained by the model unless they appreciably contribute to the reduction of the difference between the model's estimated $\hat{y}$ and the true y. In other words, extraneous parameters are pared away.

This paragraph is a Trilobite-reading SIDEBAR: The distinction between L1 and L2 regularization is that L1's additions to cost correspond to the *square* of parameter sizes while L2's additions correspond to the absolute value. The net effect of this is that L1 regularization tends to lead to the inclusion of a larger number of smaller-sized parameters in the model, while L2 regularization tends to lead to the inclusion of a smaller number of larger-sized parameters. END SIDEBAR.

Dropout

L1 and L2 regularization work fine to reduce overfitting in deep learning models, but deep learning practitioners tend to favor the use of a neural network-specific regularization technique instead. This technique, called *dropout*, was developed by

Geoff Hinton (Figure 1.17) and his colleagues at the University of Toronto[16] and was made famous its incorporation in their benchmark-smashing AlexNet architecture (Figure 1.18).

Hinton and his coworkers' intuitive yet powerful concept for preventing overfitting is captured by Figure 9.6. In a nutshell, dropout simply *pretends* that a randomly-selected proportion of the neurons in each layer *don't exist* during each round of training. To illustrate this, three rounds of training[17] are shown in the figure. For each round, we remove a specified proportion of hidden layers by random selection:

œ For the first hidden layer in the network, we've configured it to drop out one third (33.3%) of the neurons.

œ For the second hidden layer, we've configured 50% of the neurons to be dropped out.

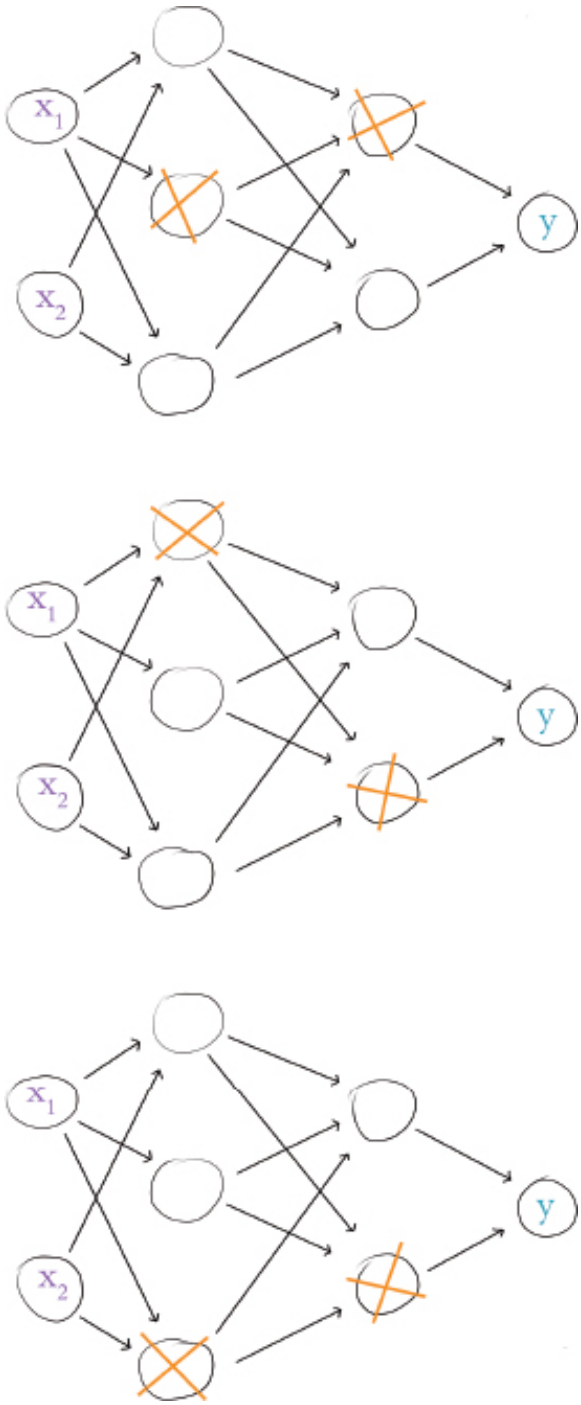

Figure 9.6 Dropout, a technique for reducing model overfitting, involves the removal of randomly-selected neurons from a network's hidden layers in each round of training. Three rounds of training with dropout are shown here.

Let's cover each of the three training rounds in turn:

1. In the top panel, the second neuron of the first hidden layer and the first neuron of the second hidden layer are randomly dropped out.

2. In the middle panel, it is the first neuron of the first hidden layer and the second one of the second hidden layer that are selected for dropout. There is no "memory" of which neurons have been dropped out on previous training rounds, and so it is by chance alone that the neurons dropped out in the second round are distinct from those dropped out in the first.

3. In the bottom panel, the third neuron of the first hidden layer is dropped out for the first time. For the second consecutive round of training, the second neuron of the second hidden layer is also randomly selected.

Instead of reining in parameter sizes toward zero (as with batch normalization), dropout doesn't (in theory) constrain how large a given parameter value can become. Dropout is nevertheless an effective regularization technique because it prevents any single neuron from become excessively influential within the network: Dropout makes it challenging for some very specific aspect of the training dataset to create an overly specific forward-propagation pathway through the network because, on any given round of training, neurons along that pathway could be removed. In this way, the model doesn't become over-reliant on certain features of the data to generate a good prediction.

When validating a neural network model that was trained using dropout, or indeed when making real-world inferences with such a network, we must take an extra step first. During validation or inference, we would like to leverage the power of the full network, i.e., its total complement of neurons. The snag is that, during training, we only ever used a subset of the neurons to forward propagate x through the network and estimate $\hat{y}$. If we were to naïvely carry out this forward propagation with suddenly *all* of the neurons, our $\hat{y}$ would emerge befuddled: There are now too many parameters and the totals after all the mathematical operations would be larger than expected. To compensate for the additional neurons, we must correspondingly adjust our neuron parameters downward. If we had, say, dropped out half of the neurons in a hidden layer during training, then we multiply the layer's parameters by 0.5 prior to validation or inference. For a hidden layer in which we dropped out 33.3% of the neurons during training, we multiply the layer's parameters by 0.667 prior to validation. Thankfully, Keras handles this parameter-adjustment process for us automatically. When working in low-level TensorFlow, however, you need to be mindful and remember to carry out these adjustments yourself.

This paragraph is a Trilobite-Reading Sidebar. If you're familiar with creating ensembles of statistical models (e.g., a single random forest out of multiple random decision trees), then it may already be evident to you that dropout produces such an ensemble. During each round of training, a random subnetwork is created and its parameter values are tuned. Later, at the conclusion of training, all of these subnetworks are reflected in the parameter values throughout the final network—in this way, the final network is an aggregated *ensemble* of its constituent subnetworks. END SIDEBAR.

Like learning rate and mini-batch size discussed in Chapter 8, network architecture options pertaining to dropout are hyperparameters. Here are our rules of thumb for choosing which layers to apply dropout to and how much of it to apply:

œ If your network is overfitting to your training data (i.e., your validation cost increases while your training cost goes down), then dropout is warranted somewhere in the network.

œ Even if your network isn't obviously overfitting to your training data, adding some dropout to the network may improve validation accuracy—especially in later epochs of training.

œ Applying dropout to *all* of the hidden layers in your network may be overkill. If your network has a fair bit of depth, it *may* be sufficient to apply dropout solely to later layers in the network (the shallowest layers may be harmlessly identifying features). To test this out, you could begin by applying dropout only to the deepest layer and observing if this is sufficient for curtailing overfitting; if not, add dropout to the next deepest layer, test it, and so on.

œ If your network is struggling to reduce validation cost or to recapitulate low validation costs attained when less dropout was applied, then you've added too much dropout—pare it back! As with other hyperparameters, there is a Goldilocks-zone for dropout too.

œ With respect to *how much* dropout to apply to a given layer, each network behaves uniquely and so some experimentation is required. In our experience, dropping out 20% up to 50% of the hidden layer neurons in machine-vision applications tends to provide the highest validation accuracies. In natural language applications, where individual words and phrases can convey particular significance, we have found that dropping out a smaller proportion—between 20% and 30% the neurons in a given hidden layer—tends to be optimal.

Data Augmentation

In addition to regularizing our model's parameters to reduce overfitting, another approach is to increase the size of our training dataset. If it is possible to collect additional high-quality training data for the particular modeling problem you're working on, then you should do so! The more data provided to a model during training, the better the model will be able to generalize to unseen validation data.

In many cases, collecting fresh data is a pipe dream. It may nevertheless be possible to

generate new training data from existing data by augmenting it, thereby artificially expanding your training dataset. With the MNIST digits, for example, many different types of transforms would yield training samples that constitute suitable handwritten digits, e.g.:

- skewing the image
- blurring the image
- shifting the image a few pixels
- applying random noise to the image
- rotating the image slightly

Indeed, as shown on the website of Yann LeCun (Figure 1.10), many of the record-setting MNIST validation dataset classifiers took advantage of such artificial training dataset expansion.[18]

FANCY OPTIMIZERS

So far in this book we've only used one optimization algorithm: stochastic gradient descent. While SGD performs well, researchers have devised shrewd ways to improve them.

Momentum

The first SGD improvement is to consider *momentum*. Here's an analogy of the principle: Let's imagine it's winter and out intrepid trilobite is skiing down a snowy gradient-mountain. If a local minimum is encountered (as in the middle panel of Figure 8.7), the momentum of the trilobite's movement down the slippery hill will keep it sailing by and the minimum will be easily bypassed. In this way, the gradients on *previous* steps have influenced the current step.

We calculate momentum in SGD by taking a moving average of the gradients for each parameter and using that to update the weights in each step. When using momentum, we have additional hyperparameter β (beta), which ranges from zero to one, and which controls how many previous gradients are incorporated in the moving average. Small β values permit older gradients to contribute to the moving average, which can be unhelpful—the trilobite wouldn't want the steepest part of the hill to guide its speed as it approaches the lodge for the après-ski portion of the day. Typically we'd use larger β values, with $\beta = 0.9$ serving as a reasonable default.

Nesterov Momentum

Another version of momentum is called *Nesterov momentum*. In this approach, the moving average of the gradients is *first* used to update the weights and find the gradients at whatever that position may be—this is equivalent to a quick peek at where momentum might take us. We then use the gradients from this sneak-peek position to execute a gradient step *from our original position*. In other words, our trilobite is suddenly aware of its speed down the hill, so it's taking that into account, guessing where its own momentum might be taking it, and then adjusting its course before it even gets there.

AdaGrad

While both momentum approaches improve SGD, a shortcoming is that they both use a single learning rate η for all parameters. Imagine, if you will, that we could have an individual learning rate for each parameter, thus enabling those parameters which have already reached their optimum to slow or halt learning, whilst those that are far from their optima can keep going. Well, you're in luck! That's exactly what can be achieved with the other optimizers we'll discuss in this section: AdaGrad, AdaDelta, RMSProp and Adam.

The name *AdaGrad* comes from "Adaptive Gradient".[19] In this variation, every parameter has a unique learning rate that scales depending on the importance of that feature. This is especially useful for sparse data where some features occur only rarely: When those features do occur, we'd like to make larger updates their parameters. We achieve this individualization by maintaining a matrix of the sum of squares of the past gradients for each parameter, and dividing the learning rate by its square root. AdaGrad is the first introduction to the parameter ε (epsilon), which is a doozy: Epsilon is a smoothing factor to avoid divide-by-zero errors and can safely be left at its default value of $\varepsilon = 1 \times 10^{-8}$.[20]

An added benefit of AdaGrad is that it minimizes the need to tinker with the learning rate hyperparameter η. You can generally just set-and-forget-it at its default of $\eta = 0.01$. A considerable downside of AdaGrad is that, as the matrix of past gradients increases in size, the learning rate is increasingly divided by a larger and larger value, which eventually renders the learning rate impractically small and so learning essentially stops.

AdaDelta and RMSProp

AdaDelta resolves the gradient-matrix-size shortcoming of AdaGrad by maintaining a *moving average* of previous gradients in just the same way that momentum does.[21]

AdaDelta also eliminates the η term so a learning rate doesn't need to be set.[22]

RMSProp (Root Mean Square Propagation) was developed by Geoff Hinton (Figure 1.17) at about the same time as AdaDelta.[23] It works similarly except it retains the learning rate η parameter. Both RMSProp and AdaDelta involve an extra hyperparameter ρ (rho), or decay rate, which is analogous to the β value from momentum and which guides the size of the window for the moving average. Recommended values for the hyperparameters are $\rho = 0.95$ for both, and $\eta = 0.001$ for RMSProp or $\eta = 1$ for AdaDelta.

Adam

The final optimizer we'll discuss in this section is also the one we'll employ most often in the book. *Adam*—short for Adaptive Moment Estimation—builds on the optimizers that came before it.[24] It's essentially the RMSProp algorithm with two exceptions:

1. An extra moving average is calculated, this time of past gradients for each parameter (called the average first moment of the gradient,[25] or just the mean) and this is used to inform the update instead of the actual gradients at that point.

2. A clever bias trick was used to help prevent these moving averages from skewing towards zero at the start of training.

Adam has two β hyperparameters, one for each of the moving averages that are calculated. Recommended defaults are $\beta_1 = 0.9$ and $\beta_2 = 0.999$. The learning rate default with Adam is $\eta = 0.001$.

Since RMSProp, AdaDelta and Adam are so similar they may be used interchangeably in similar applications, although the bias correction may help Adam later in training. Even though these new-fangled optimizers are in vogue, there is still a strong case for simple SGD with momentum (or Nesterov momentum), which in some cases performs better. As with other aspects of deep learning models, you can experiment with optimizers and observe what works best for your particular problem.

A DEEP NEURAL NETWORK IN KERAS

We can now sound the trumpet as we're reached a momentous milestone! With the additional theory we've covered in this chapter, we have enough knowledge under our belts to competently design and train a deep learning model. If you'd like to follow along interactively as we do so, pop into the accompanying *Deep Net in Keras* Jupyter notebook. Relative to our shallow and intermediate-depth model notebooks (refer to Example 5.1), we have a pair of additional dependencies—namely, dropout and batch

normalization:

Example 9.4 Additional dependencies for deep net in Keras

```
from keras.layers import Dropout

from keras.layers.normalization import BatchNormalization
```

We load and preprocess the MNIST data the same was as previously. It's the neural network architecture cell where we begin to diverge:

Example 9.5 Deep net in Keras model architecture

```
model = Sequential()



model.add(Dense(64, activation='relu', input_shape=(784,)))

model.add(BatchNormalization())



model.add(Dense(64, activation='relu'))

model.add(BatchNormalization())



model.add(Dense(64, activation='relu'))

model.add(BatchNormalization())

model.add(Dropout(0.2))



model.add(Dense(10, activation='softmax'))
```

As before, we instantiate a `Sequential` model object. After we add our first hidden layer to it, however, we also add a `BatchNormalization()` layer. In doing this we are not adding an actual layer replete with neurons, but rather we're adding the batch norm transformation for the activations a from the layer before (the first hidden layer). As with the first hidden layer, we also add a `BatchNormalization()` layer atop the second hidden layer of neurons. Our output layer is identical to the one used in the shallow and intermediate-depth nets, but to create an honest-to-goodness *deep* neural network, we are further adding a third hidden layer or neurons. As with the first and second hidden layers, the third hidden layer consists of `64` batch-normalized `relu` neurons. We are, however, supplementing this final hidden layer with `Dropout`, set to remove a fifth (`0.2`) of the layer's neurons during each round of training.

The only other change relative to our intermediate-depth network is that we use the Adam optimizer (`optimizer='adam'`) in place of ordinary SGD optimization:

```
model.compile(loss='categorical_crossentropy',

              optimizer='adam',

              metrics=['accuracy'])
```

Note that we need not supply any hyperparameters to the Adam optimizer because Keras handily includes all the sensible defaults we detailed in the previous section automatically. For all of the other optimizers we covered, Keras (and TensorFlow for that matter) has implementations that can easily be dropped in in place of ordinary SGD or Adam. Simply refer to the documentation for those libraries to see exactly how it's done.

When we call the `fit()` method on our model,[26] we discover that our digestion of all the additional theory in this chapter paid off: With our intermediate-depth network, our validation accuracy plateaued around 97.6%, but our deep net attained 97.87% validation accuracy after the 15th epoch of training, shaving 11% of our already-small error rate away. To squeeze even more juice out of the error-rate lemon that that, we're going to need machine vision-specific neuron layers such as those introduced in the upcoming Chapter 10.

TENSORBOARD

When evaluating the performance of your model epoch over epoch, it can be tedious and time-consuming to read individual results numerically, as in Figure 9.7, particularly if the model has been training for many epochs. Instead, TensorBoard (Figure 9.8) is a convenient, graphical tool for:

œ visually tracking model performance in real time,

œ reviewing historical model performances, and

œ comparing model performances.

```
Epoch 15/20
60000/60000 [==============================] - 1s 23us/step - loss: 0.0288 - acc: 0.9906 - val_los
s: 0.0865 - val_acc: 0.9787
```

Figure 9.7 Our deep neural network architecture peaked at a 97.87% validation accuracy at the 15th epoch, besting the accuracy of our shallow and intermediate-depth architectures. Due to the randomness of network initialization and training, you may obtain a slightly lower or a slightly higher accuracy with the identical architecture.

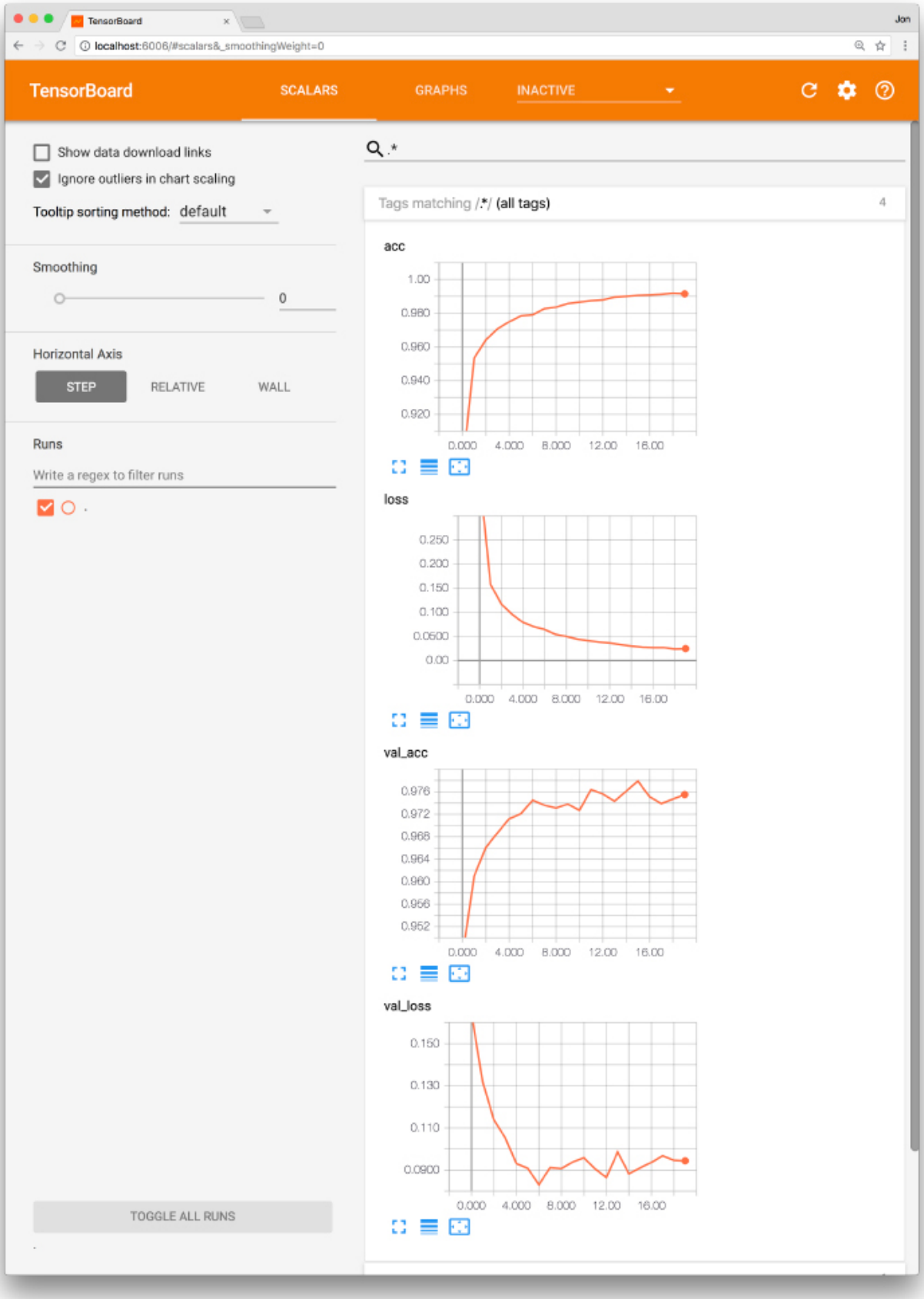

Figure 9.8 The TensorBoard dashboard enables you to, epoch over epoch, visually track your model's cost (`loss`) and accuracy (`acc`) across both your training data and your validation `val` data.

TensorBoard comes automatically with the TensorFlow library and instructions for getting it up and running are available via the TensorFlow site.[27] It's generally straightforward to set up though. Here, for example, is a procedure that adapts our

Deep Net in Keras notebook for TensorBoard use on a Unix-based operating system, including Mac OS:

1. As shown in Example 9.6, change your Python code as follows:[28]

a. Import the `TensorBoard` dependency from `keras.callbacks`

b. Instantiate a TensorBoard object (we'll call it `tensorboard`) and specify a new, unique directory name (e.g., `deep-net`) that you'd like to create and have TensorBoard log data written into for this particular run of model-fitting: `tensorboard = TensorBoard(log_dir='logs/deep-net')`

c. Pass the TensorBoard object as a `callback` parameter to the `fit()` method:

```
callbacks = [tensorboard]
```

2. In your terminal, run:[29]

```
tensorboard --logdir='logs/deep-net' --port 6006
```

3. Navigate to `localhost:6006` in your favorite web browser.

Example 9.6 Code to use TensorBoard while fitting a model in Keras

```
from keras.callbacks import TensorBoard

tensorboard = TensorBoard('logs/deep-net')

model.fit(X_train, y_train,
                batch_size=128, epochs=20,
                verbose=1,
                validation_data=(X_valid, y_valid),
                callbacks=[tensorboard])
```

By following the above steps or an analogous procedure for the circumstances of your particular operating system, you should see something like Figure 9.8 in your browser window. From there, you can visually track any given model's cost and accuracy across both your training and validation data sets in real time as these metrics change epoch by epoch. This kind of performance tracking is one of the primary uses of TensorBoard, though the dashboard interface also provides heaps of other functionality, like visual breakdowns of your neural-network graph and the distribution of your model weights. You can learn about these additional features by reading the TensorBoard docs and exploring the interface on your own.

SUMMARY

Over the course of the chapter, we discussed common pitfalls in modeling with neural networks and covered strategies for eliminating these pitfalls—or at least minimizing their impact on model performance. We wrapped up the chapter by applying all of the theory learned thus far in the book to construct our first bonafide deep learning network, which provided us with our best-yet accuracy on MNIST handwritten-digit classification. While such deep, dense neural nets are applicable to generally approximating any given output y when provided some input x, they may not be the most efficient option for specialized modeling. Coming up next in Part III, we'll introduce neural network layers and deep learning approaches that excel at particular tasks, including machine vision, natural language processing, the generation of art, and playing games.

KEY CONCEPTS

Here are the essential foundational concepts thus far. New terms from the current chapter are highlighted in purple:

œ parameters:

œ weight w

œ bias b

œ activation a

œ artificial neurons:

œ sigmoid

œ *tanh*

- ReLU
- input layer
- hidden layer
- output layer
- layer types:
- dense (fully-connected)
- softmax
- cost (loss) functions:
- quadratic (mean squared error)
- cross-entropy
- forward propagation
- backpropagation
- unstable (especially vanishing) gradients
- Glorot weight initialization
- batch normalization
- dropout
- optimizers:
- stochastic gradient descent
- Adam
- optimizer hyperparameters:
- learning rate η
- batch size

1 . Recall from Chapter 4 that a neural network earns the *deep* moniker if it consists of at least three hidden layers.

2 . Also known as a Gaussian distribution.

3 . This is part of the magic of a library like TensorFlow. While there are other operations in Python already that perform matrix multiplication, such a `numpy.matmul()`, TensorFlow has implemented highly optimized versions of these operations. Deep learning performs such a staggering number of calculations like this one, that without some heavy optimization of the underlying code, the calculations might take a long time. Additionally, TensorFlow also implements methods to perform these calculations on a GPU rather than a CPU if needed. Wherever possible, it's best to make use of TensorFlow's operations over their NumPy counterparts.

4 . In an apparent affront to the previous footnote, we're not using `tf.random_uniform()`, but in this instance it doesn't matter—this method is only called once at the start of training to initialize out trivial (and random!) example, and so it won't slow things down by a measurable amount.

5 . In case you're wondering, the leading underscore (`_ =`) keeps the Jupyter notebook tidier by outputting the plot only, instead of the plot *as well as* an object that stores the plot.

6 . Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. *Proceedings of Machine Learning Research, 9*, 249-56.

7 . Select `Kernel` from the Jupyter notebook menu bar and choose `Restart & Run All`. This ensures we start completely fresh and don't re-use old parameters from the previous run.

8 . It can be helpful to remember that some neurons *should* be saturated, i.e. their values should be very large or very small. We just endeavor to create a situation where the network learns where this is appropriate and does so intentionally, as opposed to starting off that way.

9 . The change is directly proportional to the negative magnitude of the gradient,

scaled by the learning rate η.

10. Ioffe, S., & Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv: 1502.03167.*

11. This is essentially a linear relationship, the simplest form of regression.

12. Recall the quadratic function from high school algebra.

13. Indeed, as early as Chapter 10, we'll be encountering models with tens of millions of parameters.

14. Which can be annotated as $n \gg p$, indicating the *n*umber of samples is much greater than the *p*arameter count.

15. Least Absolutely Shrinkage and Selection Operator

16. Hinton, G., et al. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *arXiv:1207.0580.*

17. If the phrase *round of training* is not immediately familiar, refer back to Figure 8.5 for a refresher.

18. yann.lecun.com/exdb/mnist

19. Duchi, J., et al. (2011) Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research, 12*, 2121-59.

20. AdaGrad, AdaDelta, RMSProp and Adam all use ε for the same purpose and it can be left at its default across all of these methods.

21. Zeiler, M.D. (2012). ADADELTA: An Adaptive Learning Rate Method. *arXiv:1212.5701*

22. This is achieved through a crafty mathematical trick that we don't think is worth expounding on here. You may notice, however, that Keras and Tensorflow still have a learning rate parameter in their implementations of AdaDelta. It is recommended to leave them at $\eta = 1$, i.e., no scaling and therefore no functional learning rate as we have come to know it in this book.

23. This optimizer remains unpublished. It was first proposed in Lecture 6e of Hinton's Coursera Course "Neural Networks for Machine Learning". The slides can be found at:

www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

24. Kingma, D.P, & Ba, J. (2014). Adam: A Method for Stochastic Optimization. *arXiv:1412.6980*

25. The other moving average is of the squares of the gradient, which is the second moment of the gradient or the variance.

26. This `model.fit()` step is exactly the same as for our *Intermediate Net in Keras* notebook, i.e., Example 8.3.

27. tensorflow.org/guide/summaries_and_tensorboard

28. This is also laid out in our *Deep Net in Keras with TensorBoard* notebook.

29. Note: we specified the same logging directory location that the TensorBoard object was set to use above. Since we specified a relative path not an absolute path for our logging directory, we need to run the `tensorboard` command from the same directory as our *Deep Net in Keras with TensorBoard* notebook.