CHAPTER 10

Deep Learning

> To get a deep-learning system to recognize a hot dog, you might have to feed it 40 million pictures of hot dogs. To get [a two year old] to recognize a hot dog, you show her a hot dog.[1]

The newest and shiniest member of the data science algorithm toolbox is deep learning. Today deep learning has captured the imagination of scientists, business leaders, and lay people. A local bank teller may not understand what deep learning is but talk to her about artificial intelligence (AI)—which is the ubiquitous avatar of deep learning and it would be surprising to learn how much she may have heard about it. The mentioned quote humorously captures the essence of what makes today's AI methods. Deep learning is a vast and rapidly emerging field of knowledge and requires a book on its own merit considering the wide-ranging architectures and implementation details it encompasses.

This chapter aims to provide an intuitive understanding of this complex topic. The hope is that this would establish a solid framework for a much more sophisticated understanding of the subject. Firstly, what constitutes the core of deep learning will be discussed and in order to do that a little computing history will be covered. The similarities between deep learning and familiar algorithms like regression will then be discussed and how deep learning is an extension of regression and artificial neural networks encountered in Chapter 4, Classification will be demonstrated. The essential differences between "traditional" algorithms like multiple linear regression, artificial neural networks and deep learning will, be pointed out by introducing the core concepts of deep learning that set it apart. One type of deep learning technique will then be explored in sufficient detail and implementation information will be provided to make this knowledge applicable to real life problems. Finally, a quick overview of some of the other newly emerging

[1] https://www.technologyreview.com/s/608911/is-ai-riding-a-one-trick-pony/.

Data Science. DOI: https://doi.org/10.1016/B978-0-12-814761-0.00010-1

BRINGING ARTIFICIAL INTELLIGENCE TO ENGINEERING

Computer-aided engineering (CAE) is a mainstay of analytical methods used by engineers (Fig. 10.1). CAE works by solving higher order partial differential equations to predict how engineered products respond to service loads that are put on them. This helps engineers design shapes and select materials for different components. For example, how does an F-15 wing behave at supersonic speeds? How much energy does the front bumper of a car absorb in a crash event?.

So how can AI help in such sophisticated activities? To understand this, one needs to dissect CAE its constituent steps. CAE is not a monolithic endeavor, but involves interaction between designers, engineers, and oftentimes computer systems people. The core activities include: creating geometric models of products, subdividing the geometries into discrete "finite elements" on which the laws of physics are applied, converting these physical laws into mathematical equations or formulations, solving these formulations, visualizing the solutions on the original geometric models (using intuitive heat maps, such as the one shown in Fig. 10.1.[2]). This entire process is not only time consuming, but also requires deep domain knowledge in graphics, mechanics, math, and high-performance computing.

All this would suggest that today's state-of-the-art AI could probably not help a whole lot. After all, from what is heard on the popular press AI is mostly chat-bots and identifying cats on the internet! However, that would be a superficial assessment. There are many smaller steps in each of the mentioned processes which are prime candidates for some of the techniques that will be learnt in this chapter.

As an illustration, we can expand on one of these tasks: subdivide the geometries in CAE. This is a crucial step and poor handling of this step will compromise all of the downstream activities. Unfortunately, it is also highly dependent on the skill of the designer who actually works on the discretization. Luckily CAE process engineers have developed many "expert" rules which even an entry level designer can use to make sure that the end product of their activities will meet rigorous standards. One expert rule is something like this: "if a part looks like a rubber gasket, use method A to discretize the geometry; if it looks like an aluminum fastener, then use method B." One can imagine in a modern-day car, for example, that there are hundreds of such similar looking parts. It is a time-consuming task—to go through an entire vehicle to correctly identify components and apply the right discretization method. It would be real value added if an AI could take over the determination of what "a part looks like" before letting an automated process apply the proper method to discretize. In a way this is essentially an application of the "cat identification" AI in a new—and more

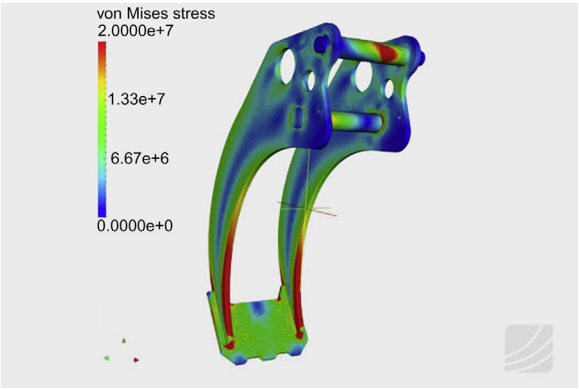

FIGURE 10.1
Computer-aided engineering.

(*Continued*)

(Continued)

serious domain. But that is not the only task which AI can turbocharge. As will be covered in this chapter, neural networks are at the core of deep learning. But what does a neural network really do? ANN's establish a mapping between the behavior of a complicated system and the environment it functions in. For instance, they help create a mapping between the behavior of a customer (churn vs stay) and the customer's shopping and purchasing habits. Or they create a mapping between the nature of a transaction (fraud vs legitimate) and the characteristics of the transactional environment (location, amount, frequency, and so on).

Another challenging task in CAE is to determine the performance of a system given highly variable environmental conditions. Will the fuselage require maintenance after 10,000 hours of flying or 1000 hours? Will an airbag deploy as intended by design, in tropical climates? Will the reduction in gage thickness still meet the crash safety requirements? Today CAE helps to answer these questions by utilizing physics based computational models to make predictions—this is the mathematical formulation and solution phase identified earlier. Such computations can be costly (in CPU time) and cumbersome (model development time) so that minor changes in design cannot be quickly studied. Deep learning can help speed these up by creating mappings between the product's design elements and its final on-field performance provided sufficient data is generated (a one-time process) to train a model initially. This idea is not new—classical CAE can be used to develop what are called response surfaces to help with this. The objective of a response surface is to effectively map the design space and then attempt to find an optimum. But a crucial problem with response surfaces was that highly nonlinear or discontinuous behavior that physical systems often exhibit would make it impossible to find optima using conventional mathematical techniques and, thus, would reduce response surfaces to mere toys. In an engineering setting, the independent variable could represent a geometric design feature such as the gage thickness of a metal part whereas the dependent variable could represent a performance metric such as energy absorption.[3] If the data is linearly separable it can be handled by many of the traditional classification algorithm we encountered earlier.

However, complex physical systems rarely exhibit such behavior. Classifying (or mapping) such responses is not possible without ANNs and more specifically without "deep" neural networks. As will be seen in this chapter, a key strength of deep learning networks is in generating nonlinear mappings.

[2] *https://commons.wikimedia.org/wiki/File:Static_Structural_Analysis_of_a_Gripper_Arm.jpg.*
[3] *For an actual example, see Fig. 3 here: https://pdfs.semanticscholar.org/2f26/c851cab16ee20925c4e556eff5198d92ef3c.pdf.*

techniques will be provided, which are now considered as part of the deep learning repertoire of techniques.

10.1 THE AI WINTER

The first ANN were the Perceptrons developed in the 1950s which were a class of pattern recognition elements that weighed evidence and tested if it exceeded a certain threshold in order to make a decision, that is, to classify patterns. Fig. 10.2 shows the architecture of a single perceptron (later on called neuron) which has retained its basic structure through the years.

Each input, x_i has a weight w_i, associated with it and a dot product $\Sigma w_i x_i$ is computed at the perceptron and passed to the activation function g. If $g(\Sigma w_i x_i)$ evaluates above a threshold then the output is set to 1 (true) or

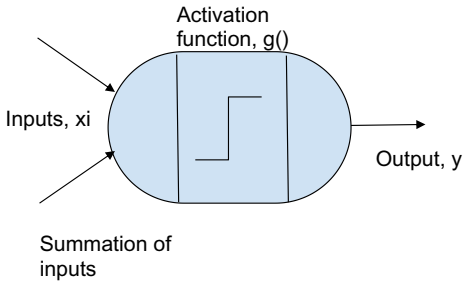

FIGURE 10.2
Conceptual architecture of a perceptron.

otherwise to 0 (false). The process of obtaining the weights, w_i, is called "learning" or "training" the perceptron. The *perceptron learning rule* was originally developed by Frank Rosenblatt (1957). Training data are presented to the network's inputs and the output is computed. The weights w_i are modified by an amount that is proportional to the product of the difference between the actual output, y, and the desired output, d, and the inputs, x_i.

The perceptron learning rule is basically:

1 Initialize the weights and threshold to small random numbers.
2 Feed the inputs x_i to the perceptron and calculate the output.
3 Update the weights according to: $w_i(t+1) = w_i(t) + \eta(d-y)x$
Where:
d is the desired output,
t is the time step, and
η is the learning rate, where $0.0 < \eta < 1.0$
4 Repeat steps 2 and 3 until:
a. the iteration error is less than a user-specified error threshold or
b. a predetermined number of iterations have been completed.

Note that learning only happens when the error is above a preset threshold, otherwise the weights are not updated. Also, every time the weights need an update, reading the input data (i.e., step 2) is required.

AI Winter: 1970's

Perceptrons were able to solve a range of decision problems, in particular they were able to represent logic gates such as "AND", "OR," and "NOT." The *perceptron learning rule* tended to converge to an optimal set of weights for several classes of input patterns. However, this was not always guaranteed. Another limitation arose when the data were not linearly separable—for example, the classic "XOR." A XOR gate resolves to "true" if the two

$$y = x_1 \oplus x_2$$

Row No.	xor	x	y
1	true	1	0
2	false	0	0
3	false	1	1
4	false	0	0
5	true	1	0
6	true	0	1
7	true	1	0
8	false	1	1
9	false	1	1
10	true	1	0
11	false	0	0
12	true	1	0

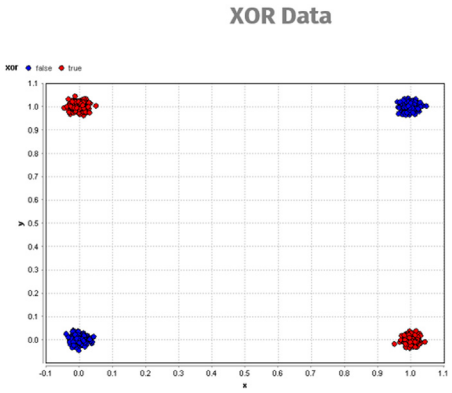

FIGURE 10.3
RapidMiner XOR example.

inputs are different and resolves to "false" if both inputs are the same (Fig. 10.3). Minsky and Papert published these and other core limitations of perceptrons in a 1969 book called *Perceptrons*, which arguably reduced further interest in these types of ANN and the so-called AI Winter had set in.

Mid-Winter Thaw of the 1980s

ANN, however, had a brief resurgence in the 1980s with the development of the multi-layer perceptron (MLP) which was heralded as the solution for nonlinearly separable functions: for example, changing the activation function in an MLP from a linear step function to a nonlinear type (such as sigmoid) could overcome the decision boundary problem seen in the XOR case.

Fig. 10.4 shows that with a linear activation function a two-layer MLP still fails to achieve more than 50% accuracy on the XOR problem using TensorFlow [4] playground. However, a simple switch of the activation function to the nonlinear "sigmoid" helps achieve more than 80% accuracy with the same architecture (Fig. 10.5).

Another important innovation in the 1980s that was able to overcome some of the limitations of the perceptron training rule was the use of "backpropagation" to calculate or update the weights (rather than reverting back to the

[4] playground.tensorflow.org.

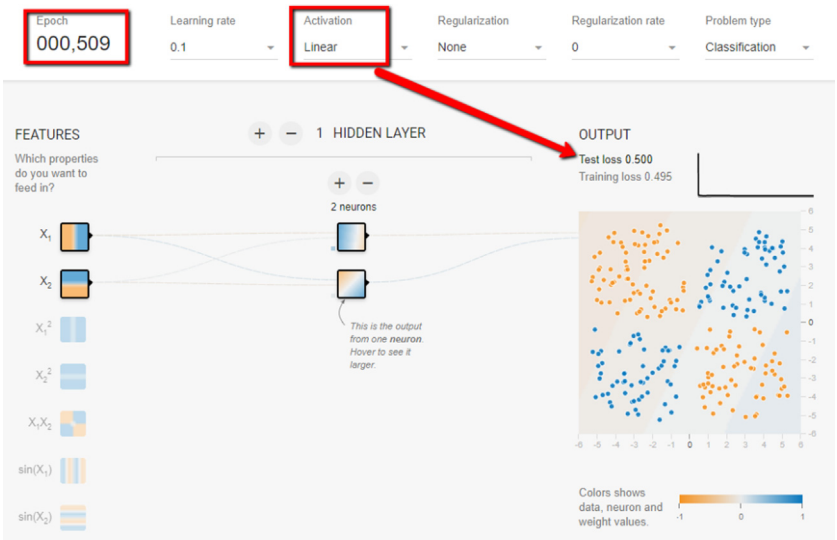

FIGURE 10.4
Exploring network architectures using TensorFlow playground.

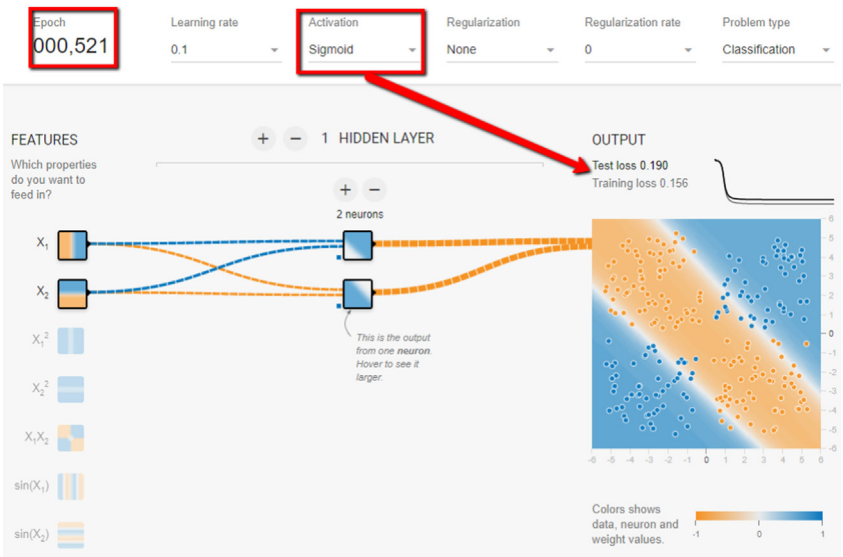

FIGURE 10.5
Modifying network architecture to solve the XOR problem using Tensorflow playground.

inputs every time there was an error—step 2 of the perceptron learning rule). The perceptron learning rule updated the weights by an amount that was proportional to the error times of the inputs (the quantity $\eta(d-y)x$ in step 2). These weights w_i are the heart of the network and training an multi-layer perceptron (MLP) or ANN is really all about finding these weights. Finding a robust and repeatable process for updating the weights becomes critical. To do this an extra layer of neurons were added in between the input and the output nodes. Now the error quantity $(d-y)$ becomes a summation and to avoid sign bias, the error may be squared, that is, $\Sigma(d_j-y_j)^2$. The challenge now was determining which direction to change the weights, w_i so that this error quantity is minimized. The algorithm now involved these steps:

1. Computing the output vector given the inputs and a random selection of weights in a "forward" computational flow.
2. Computing the error quantity.
3. Updating the weights to reduce this error at the output layer.
4. Repeat two and three for the hidden layer going backward.

This backpropagation method was introduced by Rumelhart, Hinton, and Williams (1986).[5] Their network was trainable to detect mirror symmetry; to predict one word in a triplet when two of the words were given and other such basic applications. More sophisticated ANNs were built using backpropagation, that could be trained to read handwriting (LeCun, 1989).[6] However, successful business applications of ANN were still limited and it failed to capture the public imagination the way it has currently.

Part of the reason was the state of computing hardware at the time when these algorithms were introduced. But one can argue that a bigger hurdle preventing a wider adoption back in the 1980s and 1990s was a lack of data. Many of the machine learning algorithms were developed and successfully demonstrated during this time: Hidden Markov Models and Convolutional Neural Nets were described in 1984 and 1989 respectively. However, a successful deployment of these algorithms on a practical business scale did not occur until nearly a decade later. Data (or lack thereof) was the primary reason for this. Data became more readily available and accessible only after the introduction of the internet in 1993. Wissner-Gross (2016) cites several interesting examples of breakthroughs in AI algorithms, effectively concluding that the average time period for an AI innovation to become practical was 18 years (after the introduction of the algorithm) but only 3 years after the first large scale datasets (that could be used to train that algorithm) became available.[7]

[5] https://www.iro.umontreal.ca/~vincentp/ift3395/lectures/backprop_old.pdf.
[6] http://yann.lecun.com/exdb/publis/pdf/lecun-89.pdf.
[7] http://www.spacemachine.net/views/2016/3/datasets-over-algorithms.

This brings us to the defining moment in the short but rapidly evolving history of deep learning. In 2006, Hinton (the same Hinton who was part of the team that introduced backpropagation) and Salakhutdinov, demonstrated that by adding more layers of computation to make neural networks "deep," larger datasets could be better utilized to solve problems such as handwriting recognition (Hinton and Salakhutdinov, 2006). These 3-hidden layer deep learning networks had significantly lower errors than the traditional single hidden layer ANNs that were discussed in Chapter 5, Regression Methods. With this innovation, the field of AI emerged from its long winter. This emergence is best summarized in the authors' own words[8]:

> It has been obvious since the 1980s that backpropagation through deep autoencoders would be very effective. . .provided that computers were fast enough, data sets were big enough, and the initial weights were close enough to a good solution. All three conditions are now satisfied.

Using massive datasets, deep network architectures with new and powerful graphics processing units (GPUs) originally developed for video games, real-world AI applications such as facial recognition, speech processing and generation, machines defeating humans at their own board games have become possible. ANNs had moved decisively from the research lab to mainstream media hype.

The Spring and Summer of Artificial Intelligence: 2006—Today

In spite of all these exciting developments, today's AI is still far from being what is considered artificial general intelligence (AGI). The quote at the beginning of the chapter summarizes the main aspect of today's AI: the need for massive amounts of data to train a machine to recognize concepts which seem simple even to a 2-year-old human brain. There are two main questions one would have at this point: How far away from AGI is AI today? What are the hurdles that stand in the way?

Defense Advanced Research Projects Agency (DARPA) has developed a nice classification[9] of the evolution of AI into three "waves" based on the main dimensions which reflect the capabilities of the systems: ability to learn, ability to abstract, and ability to reason.[10]

---

[8] https://www.cs.toronto.edu/~hinton/science.pdf.

[9] https://www.darpa.mil/attachments/AIFull.pdf.

[10] DARPA also lists a 4th dimension: ability to perceive. However this dimension is very close to "ability to learn". According to Webster, the difference between 'learn' and 'perceive' is tied to the senses. Learning is defined as understanding by study or experience, perception is defined as understanding or awareness through senses. For our purposes, we ignored these subtle differences."

The first wave of AI evolution includes "handcrafted knowledge" systems. These are the expert systems and chess playing programs of the 1980s and 1990s. Humans encode into the machines an ability to make decisions based on input data from a specific domain. In other words, these systems have a limited ability to reason but no ability to learn let alone abstract.

Second wave systems include today's machine learning and deep learning systems and are generally terms systems capable of "statistical learning." The uniqueness of these systems is the ability to separate data into different sets or patterns based on learning by relying on large volumes of data. While a rule engine can be added to these statistical learners, these systems still lack the ability to abstract knowledge. To clarify this: consider that while a facial recognition system is successful at identifying faces, it cannot explicitly explain why a particular face was categorized as such. On the other hand, a human can explain that a particular face was classified as a man because of the facial hair and body dimensions, for example.

In the yet-to-be developed third wave systems, AI cannot only apply encoded rules and learn from data, but can also explain why a particular data point was classified in a particular way. This is termed a "contextually adaptive" system. DARPA also calls these systems "Explainable AI" or XAI[11] that *"produce more explainable models, while maintaining a high level of learning performance (prediction accuracy)."* A first step toward developing XAI is the integration of the now conventional deep machine learning with reinforcement learning (RL), which is introduced later in this chapter.

To conclude this section, before the technical brass-tacks of deep learning are explained, it is helpful to acknowledge/recognize these facts:

- Majority of AI today is machine learning and deep learning.
- Majority of that learning is supervised.
- Majority of that supervised learning is classification.

10.2 HOW IT WORKS

In this section, the connection between conventional machine learning and deep learning will be further discussed. Firstly, linear regression and how it can be represented using an ANN will be examined more closely and then logistic regression will be discussed to reinforce the similarities between conventional machine learning techniques and deep learning. This will serve as an entry point to introduce some of the fundamental concepts in ANN and by extension deep learning. This in-depth understanding is essential to

[11] https://www.darpa.mil/program/explainable-artificial-intelligence.

confidently navigate some of the more sophisticated and specialized deep learning techniques that will come later.

To begin with, ANNs (and deep learning) should be regarded as a mathematical process. An ANN basically creates a mapping of data between outputs and inputs that is established using calculus-based optimization techniques. In a simplified mathematical format, an ANN can be essentially represented as:

$$\begin{aligned} Y &= f(X) \\ Y &= f(g(b)) \quad \text{where } X = g(b) \end{aligned} \tag{10.1}$$

where b, X and Y are vectors or more generally, tensors. In this context, vectors are one dimensional array of numbers, matricies are two dimensional arrays and tensors are more general n-dimensional arrays. The process of training ANN is mostly about finding values for the coefficients, b, to complete the mapping. The coefficients are calculated by performing a constrained optimization of an error function (error is the difference between a predicted output y' and the known output, y). A technique to iteratively perform this optimization is called backpropagation which will be discussed in this section. A basic difference between deep learning and "shallow" machine learning is in the count of the coefficients, b. Deep learning deals with weight or coefficient counts in the hundreds of thousands to millions whereas conventional machine learning may deal with a few hundred at best. This enhancement of the numerosity in the computation gives deep learning their significant power to detect patterns within data.

10.2.1 Regression Models As Neural Networks

Eq. (10.1) is a vectorized form of Eq. (5.1) which was the general statement of a multiple linear regression problem. Section 5.1.1 discussed how a linear regression problem would be solved using methods of calculus, in particular, gradient descent. The gradient descent technique is the cornerstone of all deep learning algorithms and it is advisable to revisit Section 5.1.1 at this time to become comfortable with it. The linear regression model of Section 5.1.1 can be rewritten as an ANN: Fig. 10.6 is the simple linear regression model shown as a network. Note that when $x_0 = 1$, this captures Eq. (5.2). This network is only two layers deep: it has an input layer and an output layer. Multiple regression models simply require additional nodes (one node for each variable/feature) in the input layer and no additional/intermediate layers.

Similarly, a logistic regression model can also be represented by a simple two-layer network model with one key difference. As was discussed in Section 5.2.2 on Logistic Regression, the output of logistic regression is the probability of an event, p rather than a real number value as in linear

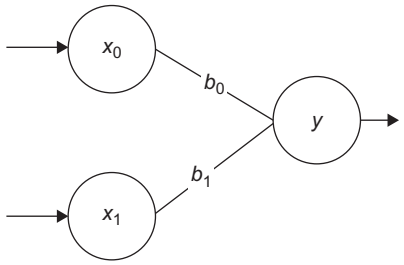

FIGURE 10.6
Regression depicted as a network.

regression. So, what needed to be done was transform the output variable in such a way that its domain now ranges from 0 to 1 instead of $-\infty-$ to $+\infty$. It was observed that by replacing the right-hand side expression of 5.6 with the log of the odds-ratio or the *logit*, this transformation was achieved.[12]

$$\log\left(\frac{p}{1-p}\right) = b_0 + b_1 x_1 \tag{10.2}$$

$$\Rightarrow p = \left(\frac{1}{1+e^{-z}}\right) \quad \text{where } z = b_0 + b_1 x_1 \text{ after rearranging the terms} \tag{10.3a}$$

more generally, the output is summarized as:

$$p(y) = \sigma(z) \tag{10.3b}$$

So, the output node in the above network could be rewritten as shown in Fig. 10.7.

Eqs. (10.3a) and (10.3b) represent the *sigmoid* function. The sigmoid's domain is [0,1] for $z \in [-\infty, \infty]$ so that any arbitrary values for b and x will always result in $p(y_n) = [0,1]$. Note that $p(y_n)$ is the prediction from the logistic regression model for sample n, which needs to be compared with the actual class value, p_n for that sample, in order to evaluate the model. How would one quantitatively compare these two across all data samples? Recall that in linear regression squared error $\Sigma(y_n - y_n')^2$ was used. The binary nature of p_n requires that the error be maximum when the predicted $p(y_n)$ and actual p_n are opposite and vice versa.

10.2.2 Gradient Descent

In Section 5.2.2 on Logistic Regression, an error function or a cost function was introduced, which can now be generalized by taking a log on the terms so that a summation is obtained instead of a product when one needs to

[12] Note that we are shortening probability of y, that is $p(y)$ to p for ease of notation in this line.

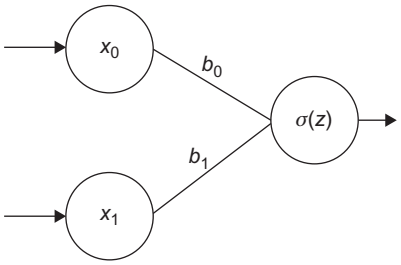

FIGURE 10.7
Adding an activation function to the regression "network."

compute across all samples. Note that y_n is the calculated value of probability based on the model and can range between [0−1] and p_n is the target which is either 0 or 1. N is the number of samples.

$$J = -\sum_{n=1}^{N} \left[p_n log(y_n) + (1 - p_n) log(1 - y_n)\right] \tag{10.4}$$

J is called the *cross-entropy cost function*. In the spirit of considering this as a cost function, a negative sign is added in front and with the aim of minimizing the value. Thus, b's need to be found which minimizes this function.[13] This has been done before using calculus in the case of linear regression (Section 5.1.1). It is easy to use the chain rule of differentiation to compute the derivative. But as will be seen this will turn out to be a constant and, thus, b cannot be solved for by setting it equal to 0. Instead, once an initial slope is obtained, gradient descent will be used to iteratively find the location where it is minimum.

Note that the cross-entropy cost function is easily expressed in terms of weights b, by substituting:

$$y = \sigma(z) = \frac{1}{(1 + e^{-z})} \quad \text{where } z = b_0 x_0 + b_1 x_1$$

The weights, b, can now be found by minimizing J, expressed in terms of b by using the chain rule of differentiation and setting this derivative to 0:

$$\frac{dJ}{db} = 0$$
$$\Rightarrow \frac{dJ}{dy} \times \frac{dy}{dz} \times \frac{dz}{db} = 0 \tag{10.5}$$

[13] Why can the same cost function not be used that was used for linear regression? That is, $\frac{1}{2} \times (p_n - y_n)^2$? It turns out that this function is not a "convex" function—in other words does not necessarily have a single global optimum.

Calculate each derivative listed in 10.5 with these three steps:

Step I. $\frac{dJ}{dy} = \left(\frac{p_n}{y_n}\right) - \left(\frac{1-p_n}{1-y_n}\right)$ using Eq. (10.3a) and (10.3b)

Step II. $y = 1/1 + e^{-z}$

$$\Rightarrow \frac{dy}{dz} = \frac{-e^{-z}}{(1+e^{-z})}$$

with proper substitution and rearrangement of terms, the right side of the derivative can be reduced to:

$$\Rightarrow \frac{dy}{dz} = \frac{y_n}{(1-y_n)}$$

Finally, Step III. $z = b_0x_0 + b_1x_1 = b_0 + b_1x_1$, noting that x_0 is usually set to 1 for the bias term, b_0

Using subscript notation:

$$\Rightarrow \frac{dz}{db} = x_i \quad \text{where } i = 1, 2, \ldots, n.$$

Putting them all together now, the derivative can be written as:

$$\frac{dJ}{db} = -\sum_{n=1}^{N}\left[\left(\frac{p_n}{y_n}\right) - \left(\frac{1-p_n}{1-y_n}\right)\right] \times \left[y_n(1-y_n)\right] \times [x_1]$$

which simplifies to:

$$\frac{dJ}{db} = -\sum_{n=1}^{N}(p_n - y_n)x_1$$

Expanding it to a general matrix form, where B, P, X, and Y are vectors:

$$\frac{dJ}{dB} = (P_n - Y_n)^T \cdot X \tag{10.6}$$

Note that B is a $(d \times 1)$ vector, where d is the number of independent variables and the other three vectors are $(n \times 1)$ where n is the number of samples. J and its derivative are scalars. The dot product between the two vectors in 10.6, will account for the summation.

As mentioned before, rather than setting the above equation to 0, an iterative approach is adopted to solve for the vector B using gradient descent. Start with an initial value for weight vector, B_j, where j is the iteration step and a step size (called the learning rate) and use this slope Eq. (10.6), to iteratively reach the point where J is minimized.

$$B_{j+1} = B_j - \textit{Learning Rate} \times [P_n - Y_n]^T \cdot X \tag{10.7}$$

In practice one would stop after a set number of iterations or when the incremental difference between B_j and B_{j+1} is very small. In general, one can calculate the weights for any error function using a formula similar to 10.6. The key is to compute the gradient of the cost function, dJ/dB, and plug it into this form:

$$B_{j+1} = B_j + Learning\ \ Rate \times \frac{dJ}{dB} \cdot X \tag{10.8}$$

Notice that the iterative update component of gradient computation for logistic regression in (10.6) is remarkably similar to the one derived for linear regression, $X^T(\hat{Y} - Y_i)$—see Section 5.1.1. They both have the same form:

$$(Predicted\ Vector - Target\ Vector) \cdot (Matrix\ of\ Input\ Data)$$

if the slight inconsistency in the nomenclature of Y is disregarded. The key difference is the way the Y's and the P are computed. In logistic regression these are evaluated using the *sigmoid transformation* ($\gamma = \sigma(b_0 + b_1 x)$), whereas in linear regression a *unit transformation* ($\gamma = b_0 + b_1 x$) is used that is, no scaling or transformation is applied to the computed output.

This is essentially the concept of an *activation function* in ANN and deep learning. Think of activation functions like a rule based weighted averaging scheme. If the weighted average ($b_0 + b_1 x$) crosses a preset threshold, the output evaluates to 1, if not it evaluates to 0—which is what happens if the activation function is the *sigmoid*. Clearly there are many candidate functions which can be used (sigmoid is simply one of them). Shown below are examples of the most commonly used activation functions used in deep learning: (1) *sigmoid*, (2) *tanh*, and (3) *rectified linear unit (RELU)* (Fig. 10.8).

Observe the similarity in shape between *sigmoid* and *tanh*—the only difference between the two is the scaling: *tanh* scales the output between $[-1, 1]$. The *RELU* is also an interesting function—the output linearly increases if the

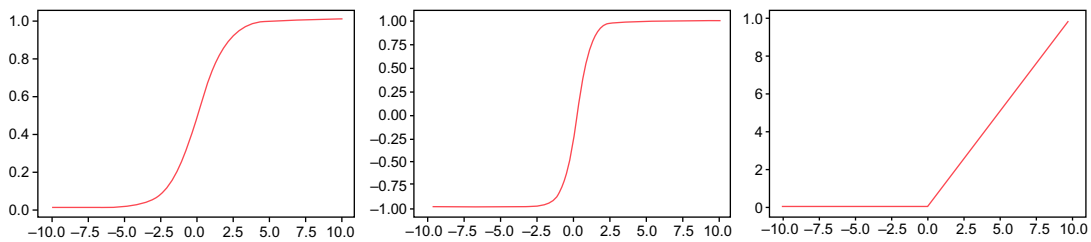

FIGURE 10.8

Commonly used activation functions: sigmoid (left), tanh (center), and RELU (right). *RELU*, Rectified linear unit.

weighted average exceeds a set threshold, otherwise it is set to 0. Note that all three are *nonlinear* functions which is an important characteristic that enables ANN and deep learning networks to classify linearly *nonseparable* data discussed in Section 10.2.

The table below summarizes the formulation of some common quantities that arise in the application of gradient descent for linear and logistic regression. For the cost functions that arise in a general ANN, calculation of the gradient in a closed form, like in Section 5.1.1, is not feasible, and it has to be computed using numerical approximation. In practice, this is what packages such as TensorFlow (TF) help implement. Such implementations rely heavily on TF for the purposes of gradient calculation and associated tensor or matrix method bookkeeping that is required.

Method	Cost Function	Derivative of Cost Function	Closed Form Solution	Gradient Form Solution
Linear egression	$J = 1/N \sum_{i=1}^{N} (y_i - b^T x_i)^2$	$dJ/db = 2/N \sum_{i=1}^{N} (y_i - b^T x_i)(-x_i)$	$B = (X^T X)^{-1} Y^T X$	$b_{i+1} = b_i - \eta\, dJ/dB$, where η is the learning rate and $dJ/dB = X^T \bullet (\hat{Y} - Y_i)$
Logistic regression	$J = -\sum (p_n \log y_n + (1 - p_n) \log(1 - y_n))$	$\frac{dJ}{db} = \frac{dJ}{dy}\frac{dy}{dz}\frac{dz}{db}$ where $y = \sigma(z)$ and $z = \Sigma b_i x_i$	None	$b_{i+1} = b_i - \eta dJ/dB$; where $dJ/dB = (P_n - Y_n)^T \cdot X$

10.2.3 Need for Backpropagation

Weighted averaging is one intuition that would allow a full ANN to be conceptualized starting from the simple two-layer models of logistic and linear regression. As discussed, in Section 10.2.2, gradient descent is about incrementally updating the weights based on the output generated from a previous iteration. The first iteration is kicked off by randomly choosing the weights. Can the process be made a bit more efficient and "well-rounded" by choosing a series of starting weights in parallel? That is, can one start by building, say 3 logistic (or linear) regression units or models in parallel so that instead of (b_1, b_2) as the starting weights, there are 3 sets: (b_{11}, b_{21}), (b_{12}, b_{22}), and (b_{13}, b_{23})? Here the first subscript refers to the input node or feature and the second subscript refers to the node in the intermediate or so-called "hidden" layer. This is illustrated in the Fig. 10.9. It turns out that by doing this, a small computational price might be paid at each iteration, but the output may be arrived at quicker by reducing the number of iterations. Finally, the output from the hidden layer is once again weight-averaged by 3 more weights (c_1, c_2, and c_3) before the final output

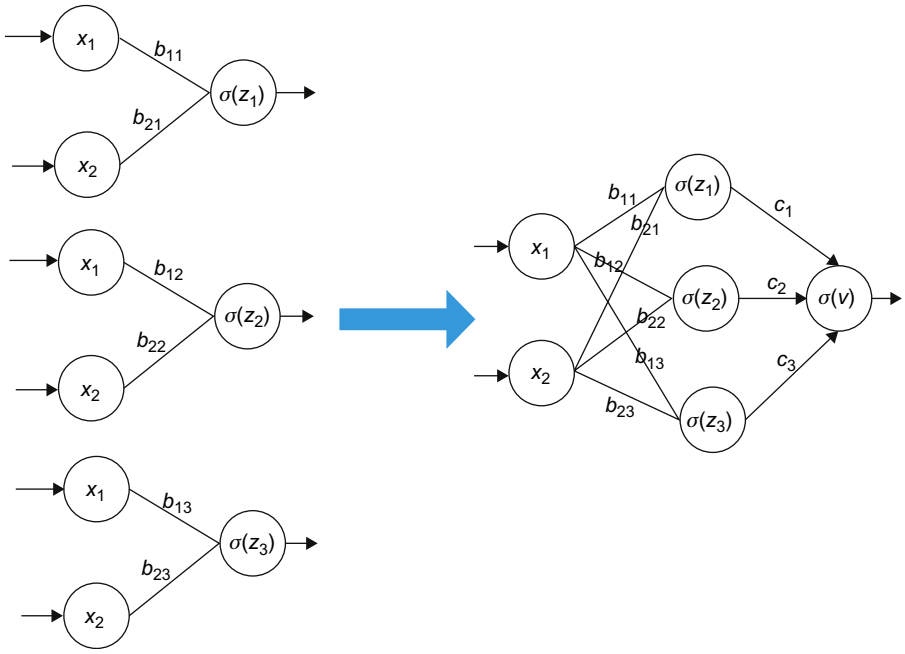

FIGURE 10.9
Combining multiple logistic regression models into a neural network.

is computed. Also observe that the right-hand side figure is exactly equivalent to the left-hand side, but representationally simpler. With the hidden layer in place, the output is first computed starting from left to right: that is, $\sigma(z_1)$, $\sigma(z_2)$, and $\sigma(z_3)$ are computed as usual and weights for c_1:c_3 are assumed in order to compute the final sigmoid $\sigma(v)$. This would be the first iteration. The output can now be compared to the correct response and the model performance evaluated using the cross-entropy cost function. The goal is now to reduce this error function for which one would first need to incrementally update the weights c_1:c_3 and then work *backwards* to then update the weights b_{11}:b_{23}. This process is termed "backpropagation" for obvious reasons. Backpropagation remains relevant no matter how many hidden layers or output nodes are in question and is fundamental to understanding of how all ANNs work. The actual mechanics of this computation was described with a simple example in Section 4.6 and will not be repeated here. The main difference between the example used in that case and this present one is the computation of the error function.

10.2.4 Classifying More Than 2 Classes: Softmax

Recall that by using logistic regression one is able to address a binary classification problem. However, most real-world classifications require categorization into one of more than two classes. For example, identifying faces, numerals, or objects, and so on. One needs a handy tool to identify which of the several classes a given sample belongs to. Also recall that in the network models discussed so far in this chapter, there was only one output node and the probability that a given sample belonged to a class was obtained. By extension, one could add an output node for every class that needs to be categorized into and the probability that a sample belonged to that particular class (see Fig. 10.10) could simply be computed. This is intuitively how the "softmax" function works.

Softmax does two calculations: it exponentiates the value received at each node of the output layer and then normalizes this value by the sum of the exponentiated values received at all the output nodes. For instance, when the output layer has two nodes (one each for class one and class two), the probability of each class can be expressed as:

$$p(Y = \text{class 1}): \frac{e^{z_1}}{(e^{z_1} + e^{z_2})}$$

$$p(Y = \text{class 2}): \frac{e^{z_2}}{(e^{z_1} + e^{z_2})}$$

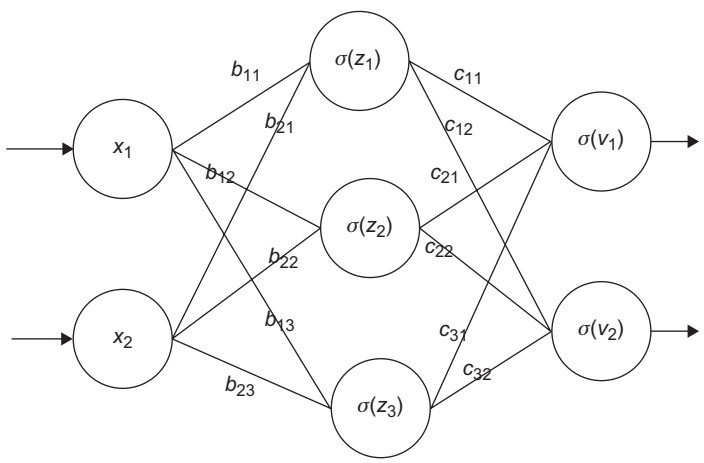

FIGURE 10.10
Softmax output layer in a network.

If one divides the numerator and the denominator of the first class by e^{z_1}, one would get:

$$p(Y = \text{class } 1): \frac{1}{(1 + e^{z2-z1})}$$

which is the same expression as the sigmoid, Eq. (10.3a), if one had only one output node for a binary classification problem (i.e. $z_2 = 0$). In general, for a k-class problem, softmax is computed as:

$$p(Y = k) = \frac{e^{zk}}{\sum_{i=1}^{k} [e^{zi}]} \tag{10.9}$$

Note that what the z_i are has not been stated. In practice they have the same form as in Eq. (10.3a), that is, $z_i = \Sigma b_i x_i$. Another thing to keep in mind is that in logistic regression based on the sigmoid, the thresholding is used or the cut-off value of the output probability to assign the final class—if the value is >0.5 then a "hard maximum" is applied to assign it to one class over the other.

The triple concepts of activation functions, backpropagation, and (calculus based) gradient descent form what may be loosely termed as the "ABCs" of deep learning and remain at the mathematical core of these algorithms. Hopefully this section provided an intuitive understanding of these important concepts using simple networks such as linear and logistic regression models. In combination with the material presented on ANN in Chapter 4, on Classification one should now be in a strong enough position to grasp the extension of these techniques to larger scale networks such as convolutional neural networks.

There is no strict definition of what constitutes a "deep" learning network. A common understanding is that any network with three or more hidden layers between the input and output layers is considered "deep." Based on the math described in the preceding section it should be easy to understand how adding more hidden layers increases the number of weight parameters, b_{ij}. In practice it is not uncommon to have networks where the number of weight parameters (termed *trainable parameters*) run into millions. In the next few sections some typical use cases or applications of deep learning methods will be explored and the practical implementations discussed.

10.2.5 Convolutional Neural Networks

A convolution is a simple mathematical operation between two matrices. Consider a 6×6 matrix A, and a 3×3 matrix B.

$$A = \begin{bmatrix} 10 & 9 & 9 & 8 & 7 & 7 \\ 9 & 8 & 8 & 7 & 6 & 6 \\ 9 & 8 & 8 & 7 & 6 & 6 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

Convolution between ***A*** and ***B***, mathematically denoted as ***A*** (*) ***B***, results in a new matrix, ***C*** whose elements are obtained by a sum-product between the individual elements of ***A*** and ***B***. For the given example,

$$c_{11} = 10 \times 1 + 9 \times 1 + 9 \times 1 + 9 \times 0 + 8 \times 0 + 8 \times 0 + 9 \times -1 + 8 \times -1 + 8 \times -1 = 3$$
$$c_{12} = 9 \times 1 + 8 \times 1 + 8 \times 1 + 8 \times 0 + 8 \times 0 + 7 \times 0 + 8 \times -1 + 8 \times -1 + 7 \times -1 = 3$$

and so on.

It is easier to understand this operation by visualizing as shown in Fig. 10.11. Matrix ***B*** is the lighter shaded one which essentially slides over the larger matrix ***A*** (darker shade) from left (and top) to right (and bottom). At each overlapping position, the corresponding elements of ***A*** and ***B*** are multiplied and all the products are added as indicated in the figure to obtain the corresponding element for ***C***. The resulting output matrix ***C*** will be smaller in dimension than ***A*** but larger than ***B***. So, what is the utility of this

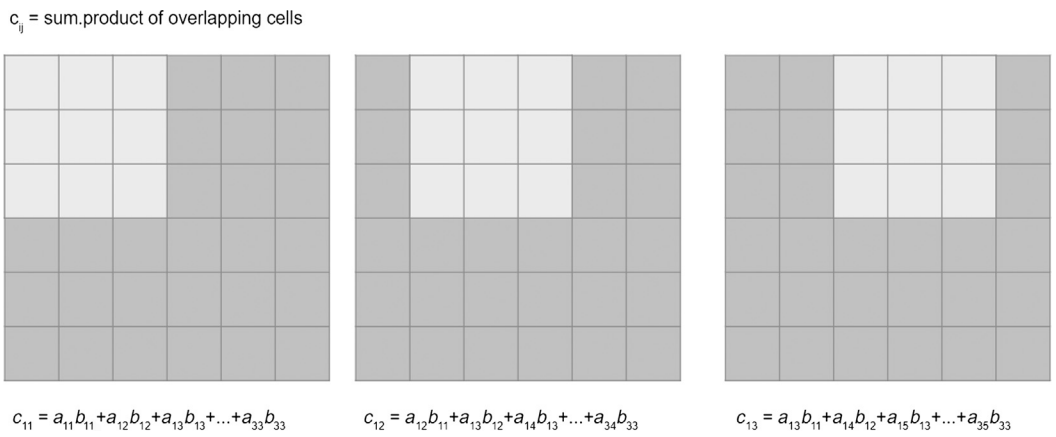

FIGURE 10.11
Computing a convolution.

convolution operation? Matrix $\boldsymbol{A}$ is typically a raw image where each cell in the matrix is a pixel value and matrix $\boldsymbol{B}$ is called a filter (or kernel) which when convolved with the raw image results in a new image that highlights only certain features of the raw image.

As shown in the Fig. 10.12, $\boldsymbol{A}$ and $\boldsymbol{B}$ are basically pixel maps which when convolved together yield another pixel map, $\boldsymbol{C}$. One can see that C accentuates or highlights the horizontal edge in A where the pixels jump from a high value to a low value. In this case, the accent appears to be thick—but in the case of a real image where the matrix sizes are in the order of 1000s (e.g., a 1-megapixel image is approximately a 1000×1000 matrix of pixels)—the accent line will be finer and clearly demarcate or detect any horizontal edge in the picture. Another thing to note about the filter is that when it is flipped by 90 degrees or transposed, that is, use $\boldsymbol{B}^T$ instead of $\boldsymbol{B}$, a convolution performed on a raw image would be able to detect *vertical* edges. By judiciously choosing $\boldsymbol{B}$, it will be possible to identify edges in any orientation.

Thus, convolutions are useful in order to identify and detect basic features in images such as edges. The challenge is of course to determine the right filter for a given image. In the example this was done intuitively—however, machine learning can be used to optimally determine the filter values. Observe that determining the filter was a matter of finding the $3 \times 3 = 9$ values for the matrix $\boldsymbol{B}$ in the example. A couple of things to note: the matrix $\boldsymbol{A}$ is $n_{\text{width}} \times n_{\text{height}}$ in terms of pixels for a grayscale image. Standard color images have three channels: red, green, and blue. Color images are easily handled by considering a 3-dimensional matrix $n_{\text{width}} \times n_{\text{height}} \times n_{\text{channels}}$ which are convolved with as many different filters are there are color channels.

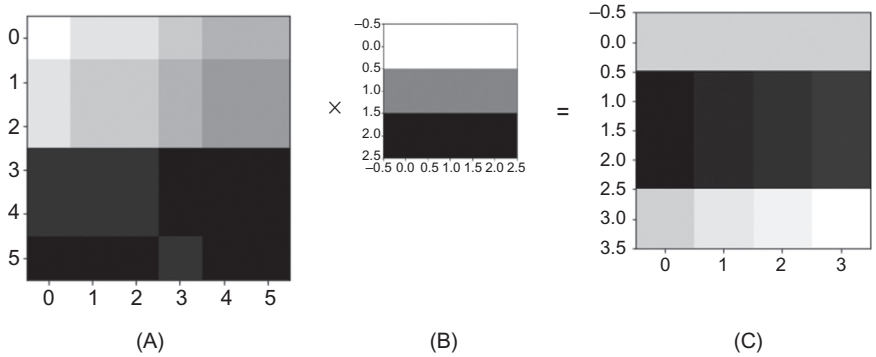

FIGURE 10.12
Effect of performing a convolution on a raw image.

In Fig. 10.12, one can observe that ***C*** is smaller than the raw image ***A***. If the dimensions of ***A*** and ***B*** are known, the dimensions of ***C*** can be determined. For simplicity, assume that $n_{\text{width}} = n_{\text{height}} = n$ which is quite common in real applications. If the filter is also a square matrix of size f, then the output ***C*** is square of dimension $n - f + 1$. In this case $n = 6$, $f = 3$ and, therefore, ***C*** was 4×4.

As the process of convolution reduces the raw image size, it is sometimes useful to enlarge the raw image with dummy pixels so that the original size is retained. This process is called "*padding*" as illustrated in Fig. 10.13. If p is the number of padded pixels, then the output dimension is given by $n + 2p - f + 1$. Thus, in the example, the output of ***C*** will be 6×6 if one unit of padding is added.

Another important consideration in computing convolutions is the "*stride.*" This is the number of pixels the filter is advanced each time to compute the next element of the output. In the examples discussed so far, a stride of 1 has been assumed. Fig. 10.14 illustrates this point. With stride, s, the output dimension can be computed as $(n + 2p - f)/s + 1$.

So far, it has been taken on intuition that a convolution helps to identify basic image features such as edges. Thus, a filter strides over an image and at each location of the output a sum-product of the corresponding elements can be computed between the image and filter. However, instead of performing a sum-product if one simply used the highest pixel value in the overlapping cells at each compute location (Fig. 10.15), a process called *max pooling* would be obtained. Max pooling can be thought of as an operation that highlights the most dominant feature in an image (e.g., an eye within a face). Max pooling is another feature detection tactic that is widely used for

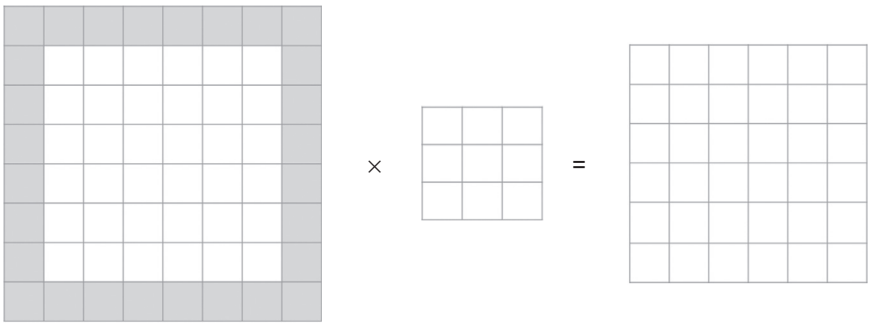

FIGURE 10.13
Padding a convolution.

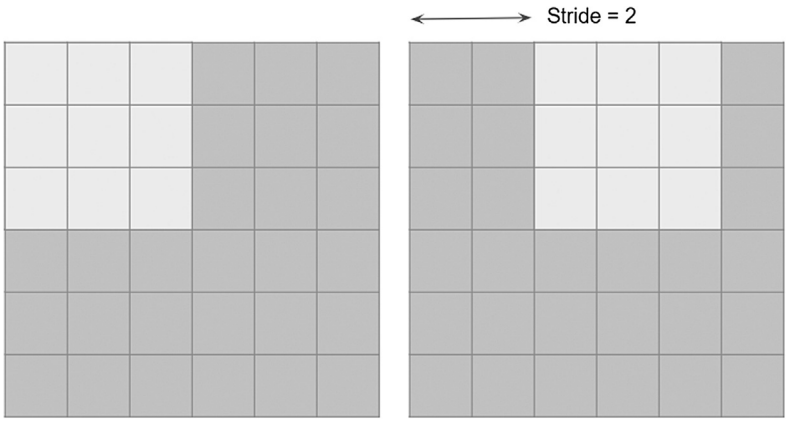

FIGURE 10.14
Visualizing the stride during a convolution.

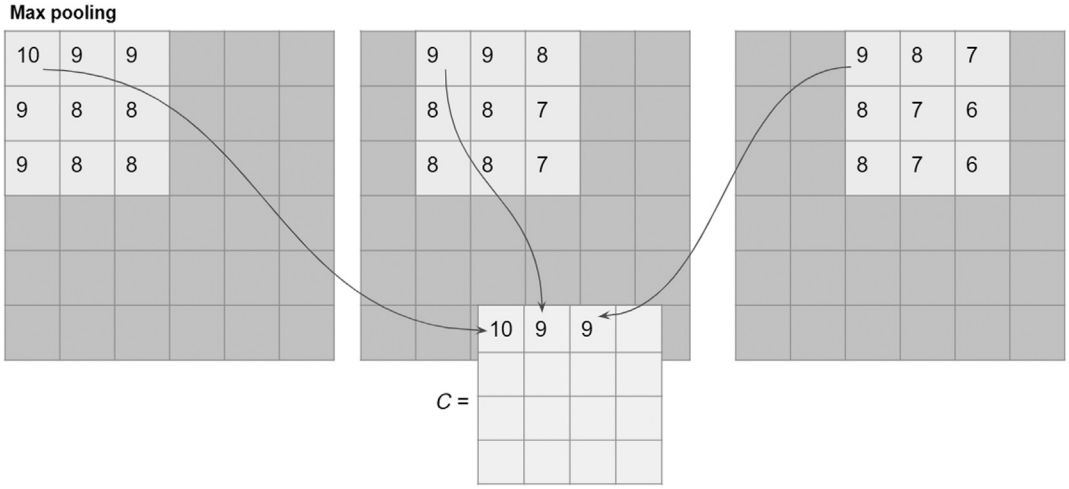

FIGURE 10.15
Max pooling visualized.

image processing. Similar to max pooling, an *average pooling* can also be done to somewhat "airbrush" the raw image, where the values are averaged in the overlapping cells instead of doing a sum-product. The calculation for output dimension after pooling still remains $(n + 2p\text{-}f)/s + 1$.

A convolution is a linear function similar to Eq. **10.1** which is a general form of any neural networkwhere the weights, $\boldsymbol{B}$ are now the pixels of the filter matrix and the inputs, $\boldsymbol{A}$ are the image pixel values. The sum-product

output $\boldsymbol{C}$ is analogous to z in 10.3a . Similar to applying a nonlinear operator to z in 10.3b, the elements of $\boldsymbol{C}$ are typically passed through a RELU nonlinearity. This entire process, as will be summarized, forms one convolutional layer. The output of this convolutional layer is sent to the next layer which can be another convolutional layer (with a different filter) or "flattened" and sent to a regular layer of nodes, called a "Fully Connected" layer which will be described later on.

In the Fig. 10.16, $\boldsymbol{A[0]}$ is the raw image and $\boldsymbol{C}$ is the result of its convolution with the filter $\boldsymbol{B}$. $\boldsymbol{A[1]}$ is the result of adding a bias term to each element of $\boldsymbol{C}$ and passing them through a RELU activation function. $\boldsymbol{B}$ is analogous to a weight matrix b of **10.1**, while C is analogous to $\sigma(z)$ in **10.2b**. The point of making these comparisons is to highlight how convolutions can be used as a part of a deep learning network. In a neural network, backpropagation can be used to compute the elements of the weight matrix, $\boldsymbol{b}$, and a similar process can be applied to determine the elements of the filter matrix $\boldsymbol{B}$.

One additional note: TF and such tools allow one to apply multiple filters in the same layer. So for example, one can let $\boldsymbol{B_1}$ be a horizontal edge detector, $\boldsymbol{B_2}$ be a vertical edge detector, and apply both filters on $\boldsymbol{A[0]}$. The output $\boldsymbol{C}$ can be tensorially represented as a volume, in this example, $\boldsymbol{C}$ will have the dimension of $4 \times 4 \times 2$, which is essentially a stack of two 4×4 matrices, each one is the result of convolution between $\boldsymbol{A}$ and $\boldsymbol{B}_i$ ($\underline{i} = 1,2$). Fig. 10.17 shows how adding multiple filters at a layer will result in a volumetric network.

In order to determine the filter elements, remember that backpropagation will be used. So, a cost-function will need to be computed, such as the one in **10.3**, and minimize/maximize it using gradient descent. The cost function is now dependent on $3 \times 3 \times 2 = 18$ parameters in this example. In order to

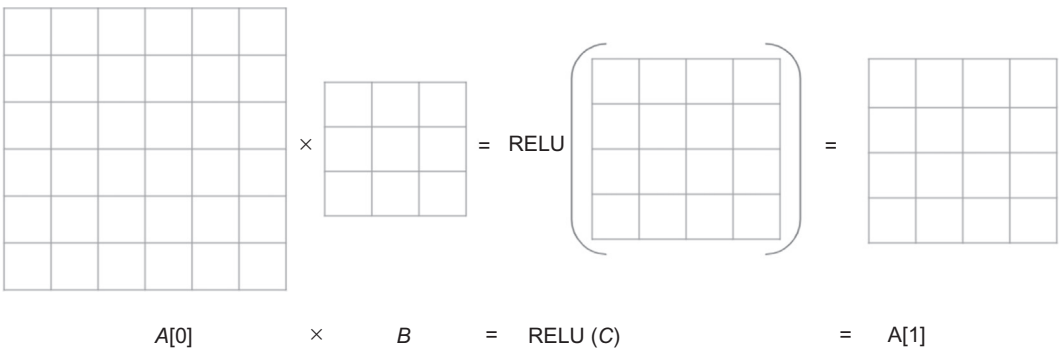

FIGURE 10.16
Combining a convolution with an activation function.

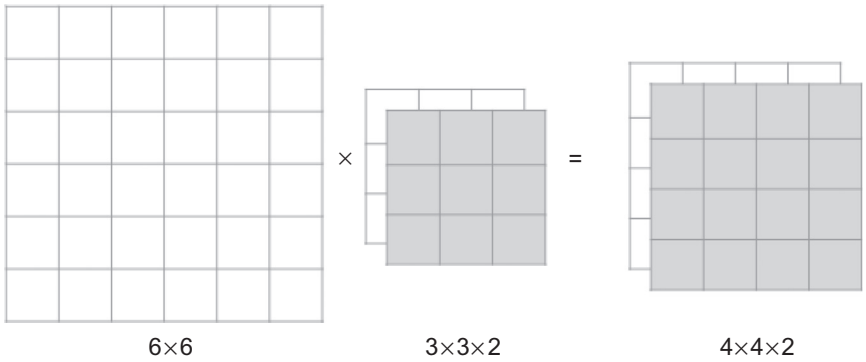

FIGURE 10.17
Multiple filters of convolution.

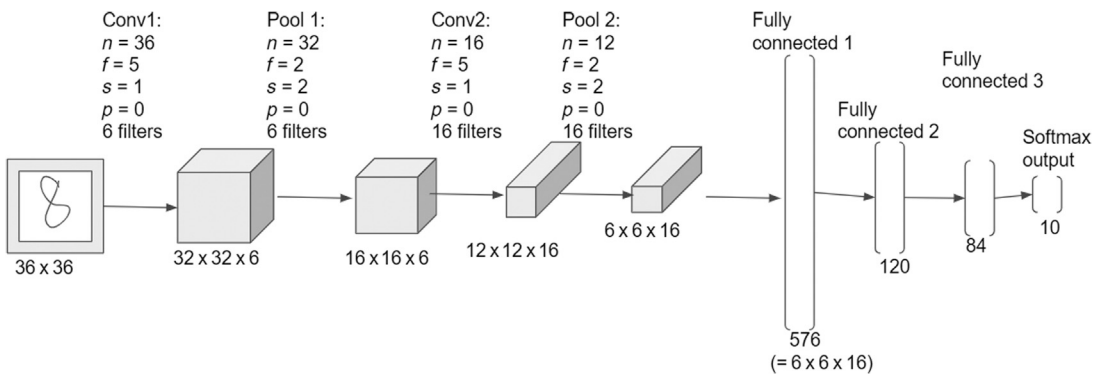

FIGURE 10.18
Architecture of a typical CNN model.

completely define the cost-function one can "flatten" the output $4 \times 4 \times 2$ matrix into $32(=4 \times 4 \times 2)$ nodes and connect each node to a logistic regression output node or softmax output nodes.

Fig. 10.18 shows a classic CNN architecture introduced by LeCun (1989). It consists of several convolutional layers interspersed with max pool layers and finally followed by what are known as fully connected layers where the last convolution output matrix is flattened into its constituent elements and passed through a few hidden layers before terminating at a softmax output layer. This was one of the first CNNs and was used to recognize handwritten digits.

CNNs are highly capable deep learning networks which function highly efficiently because of a couple of reasons. The feature detection layers (such as

Conv1, Conv2, etc.,) are computationally quite quick because there are few parameters to train (e.g., each Conv1 filter is 5×5 which yields $25 + 1$ for bias = 26 times 6 filters = 156 parameters) and not every parameter in the output matrix is connected to every other parameter of the next layer as in a typical neural network (as happens in the fully connected layers down the network). Thus the fully connected layers FC1 and FC2 have 576 times 120 = 69,120 parameters to train. Because of their flexibility and computational efficiency, CNNs are now one of the most common deep learning techniques in practical applications.

Layers are the high-level building blocks in deep learning. As observed in Fig. 10.18, there are several convolutional layers and several fully connected layers (also called "*Dense*" layers) Each layer receives the inputs from the previous layer, applies the weights and aggregates with an activation function. A couple of other key concepts that are in usage in deep learning are summarized here.

10.2.6 Dense Layer

A dense or fully connected layer is one where all the nodes in the prior layer are connected to all the nodes in the next layer. Several layers of dense layers form different levels of representation in the data. (Fig. 10.19).

10.2.7 Dropout Layer

A dropout layer helps to prevent model overfitting by dropping the nodes randomly in the layer. The probability of dropping a node is controlled by a factor which ranges from 0 to 1. A dropout factor closer to one drops more nodes from the layer. This is a form of regularization that reduces the complexity of the model. This concept is depicted in Fig. 10.20

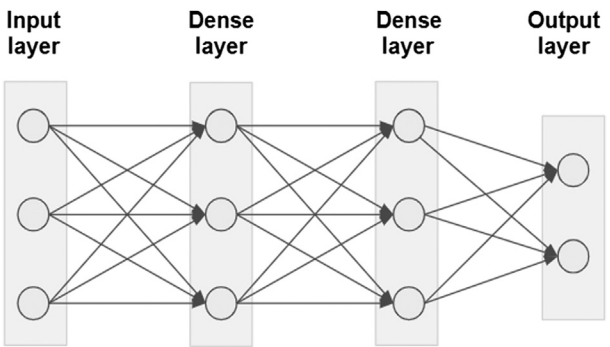

FIGURE 10.19
Illustrating a dense or fully connected layer.

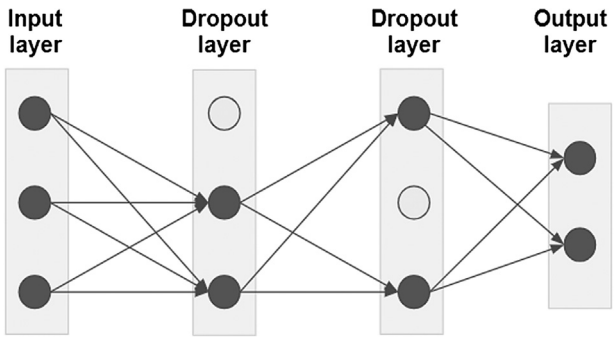

FIGURE 10.20
Illustrating a dense or fully connected layer.

10.2.8 Recurrent Neural Networks

The field of deep learning has exploded in the last decade due to a variety of reasons outlined in the earlier sections. This chapter provided an intuition into one of the most common deep learning methodologies: CNN. These are but one representation of a deep neural network architecture. The other prominent deep learning method in widespread use is called recurrent neural network (RNN). RNNs find application in any situation where the data have a temporal component. Prominent examples are time series coming from financial data or sensor data, language related data such as analyzing the sentiment from a series of words which make up a sentence, entity recognition within a sentence, translating a series of words from one language to another and so on. In each of these instances, the core of the network still consists of a processing node coupled with an activation function as depicted in Fig. 10.1.

Suppose the time series problem is a text entity recognition. So, here is a set of training examples consisting of sentences from which one can identify certain named entities in each sentence such as proper nouns, places, dates, and so on. The training set, thus, looks like:

Sample	$x^{<1>}$	$x^{<2>}$	$x^{<3>}$	$x^{<4>}$
1	This	is	an	egg
2	I	love	scrambled	eggs
3	Do	you	like	omelettes?
4	Green	eggs	and	Ham
5	My	name	is	Sam
...	...	...	...	...

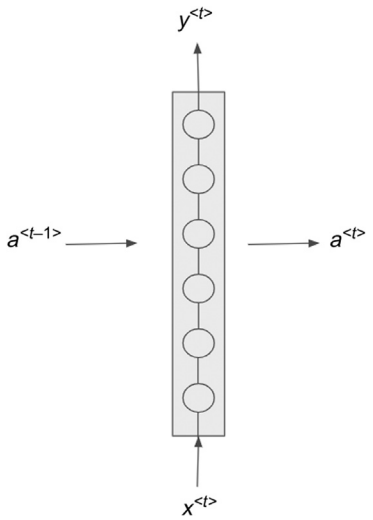

FIGURE 10.21
A basic representation of RNN. *RNN*, Recurrent neural network.

The objective here is to predict that $y^{<j>}$ is a named entity such as a proper noun. So, $y^{<4>}$ would be 1, whereas $y^{<1,2,3>}$ are 0's. The idea behind an RNN is to train a network by passing the training data through it in a sequence (where each example is an ordered sequence). In the schematic in Fig. 10.21, $x^{<t>}$ are the inputs where $<t>$ indicates the position in the sequence. Note that there are as many sequences as there are samples. $y^{<t>}$ are the predictions which are made for each position based on the training data. What does the training accomplish? It will determine the set of weights of this (vertically depicted) network, b_x which in a linear combination with $x_i^{<t>}$ and passed through a nonlinear activation (typically tanh), produces an activation matrix $a^{<t>}$. So:

$$a^{<t>} = g(b_x x^{<t>})$$

However, RNNs also use the value of the activation from the previous time step (or previous word in the sequence) because typically in most sequences—such as sentences—the prediction of a next word is usually dependent on the previous word or set of words. For example, the previous words "My", "name", and "is" would almost certainly make the next word a proper noun (so $y = 1$). This information is helpful in strengthening the prediction. Therefore, the value of the activation matrix can be modified by adding the previous steps' activation multiplied by another coefficient, b_a:

$$a^{<t>} = g(b_a a^{<t-1>} + b_x x^{<t>})$$

Finally, the prediction itself for the position $<t>$ is given by:

$$y^{<t>} = g(b_y a^{<t>})$$

where b_y is another set of coefficients. All the coefficients are obtained during the learning process by using the standard process of backpropagation. Because of the temporal nature of data, RNNs typically do not have structures that are as deep as in CNNs. It is not common to see more than 4–5 layers which are all temporally connected.

10.2.9 Autoencoders

So far, deep learning has been discussed in the context of supervised learning where an explicit labeled output dataset was used to train a model. Deep learning can also be used in an unsupervised context and this is where Autoencoders are useful.

An autoencoder is deep learnings answer to dimensionality reduction (which is introduced in chapter 14 on Feature Selection). The idea is pretty simple: transform the input through a series of hidden layers but ensure that the final output layer is the same dimension as the input layer. However, the intervening hidden layers have progressively smaller number of nodes (and hence, reduce the dimension of the input matrix). If the output matches or encodes the input closely, then the nodes of the smallest hidden layer can be taken as a valid dimension reduced dataset.

This concept is illustrated in Fig. 10.22.

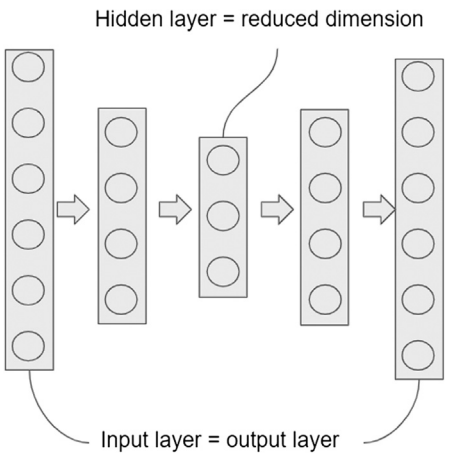

FIGURE 10.22
Concept of dropout.

10.2.10 Related AI Models

Two other algorithms will briefly be mentioned which fall more into the domain of AI rather than the straightforward function approximation objective used so far. However, many researchers and experts tend to consider these as newer applications of deep learning because deep networks may be used as part of the algorithms.

Reinforcement Learning (RL) is an agent-based goal seeking technique where an (AI) agent tries to determine the best action to take in a given environment depending on a reward. The agent has access to data which correspond to the various states in an environment and a label for each action. A deep learning network may be used to take in an observation or state-array and output probabilities for each action (or label). The most popular implementation of RL is Google's alpha-go AI which defeated a top-ranked human Go player. Practical applications of RL include route optimization strategies for a self-driving vehicle, for example. Most such applications are experimental as of this publication.

Generative adversarial network (GAN) are at the cutting edge of deep learning implementations—they were first introduced in 2014 and are yet to find mainstream applications. GANs are proposed to be used to generate new samples similar to the data they were originally trained on. For example, creating new "photographs of faces" after being trained on a large set of facial recognition data. GANs consist of two parts: A Generator and a Discriminator. Both of these are deep neural networks, the generator "mashes" up new samples while the discriminator evaluates if the new samples are "valid" or not. One can think of the analogy of counterfeiting currency. A counterfeiter tries to pass off poorly forged notes at first to be rejected by say, a vendor. The counterfeiter learns from this experience and gets increasingly sophisticated in his forgery until such time that the vendor can no longer discriminate between a forged note and a real one and, thereby, accepts the new note as a valid sample of the known distribution.

10.3 HOW TO IMPLEMENT

Deep-learning architecture in RapidMiner can be implemented by a couple of different paths. The simple artificial neural networks with multiple hidden layers can be implemented by the *Neural Net* operator introduced in Chapter 4, Classification, Artificial Neural Network. The operator parameter can be configured to include multiple hidden layers and nodes within each

layer. By default, the layer configuration is *dense*. This operator lacks the variety of different layer designs that distinguishes deep-learning architecture from simple Neural Networks.

The *Keras* extension for RapidMiner offers a set of operators specific for deep learning. It utilizes the Keras neural network library for Python. Keras is designed to run on top of popular deep learning frameworks like TensorFlow and Microsoft Cognitive Toolkit. The *Keras* extension in RapidMiner enables a top-level, visual, deep-learning process along with data science preprocessing and postprocessing. The Keras extension requires Python and related libraries to be installed.[14] The modeling and execution of the deep-learning process in production application requires machines running GPUs as computation using normal machines can be time consuming. The following implementation uses Keras extension operators and can be run on general-purpose machines[15].

Handwritten Image Recognition

Optical character recognition is an image recognition technique where handwritten or machine-written characters are recognized by computers. A deep learning-based (convolutional neural network) numeric character recognition model is developed in this section. As with any deep-learning model, the learner needs plenty of training data. In this case, a large number of labeled handwritten images are needed to develop a deep learning model. The MNIST database[16] (Modified National Institute of Standards and Technology database) is a large database of labeled handwritten digits used commonly for training image processing systems. The dataset consists of 70,000 images of handwritten digits (0,1,2,...,9). Fig 10.23 shows sample training images for the digits 2, 8, and 4.

The complete RapidMiner process for implementing handwritten image recognition on the MNIST dataset is shown in Fig. 10.24.[17] The process has a Execute Python operator to convert the 28 x 28 image pixels to a data frame. The Keras Model contains a deep-learning model with several convolutional and dense layers. The model was applied to the test dataset and performance was evaluated. The dataset and the process are available at www.IntroDataScience.com

---

[14] https://community.rapidminer.com/t5/RapidMiner-Studio-Knowledge-Base/Keras-Deep-Learning-extension/ta-p/40839.

[15] Note that on CPU the runtime is likely to be 100x to 1000x slower than on GPU. For this example, the run time was 37 hours on a 2.5 GHz core i7.

[16] MNIST Database - http://yann.lecun.com/exdb/mnist/.

[17] This implementation example was generously provided by Dr. Jacob Cybulski of Deakin University, Australia.

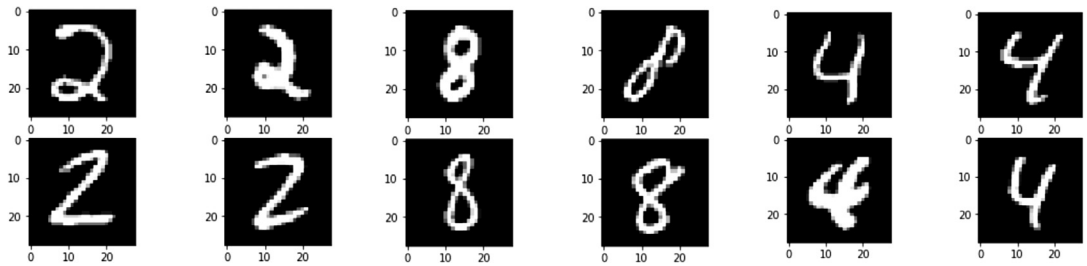

FIGURE 10.23
Three different handwritten samples of the digits 2,8, and 4 from the MNIST dataset.

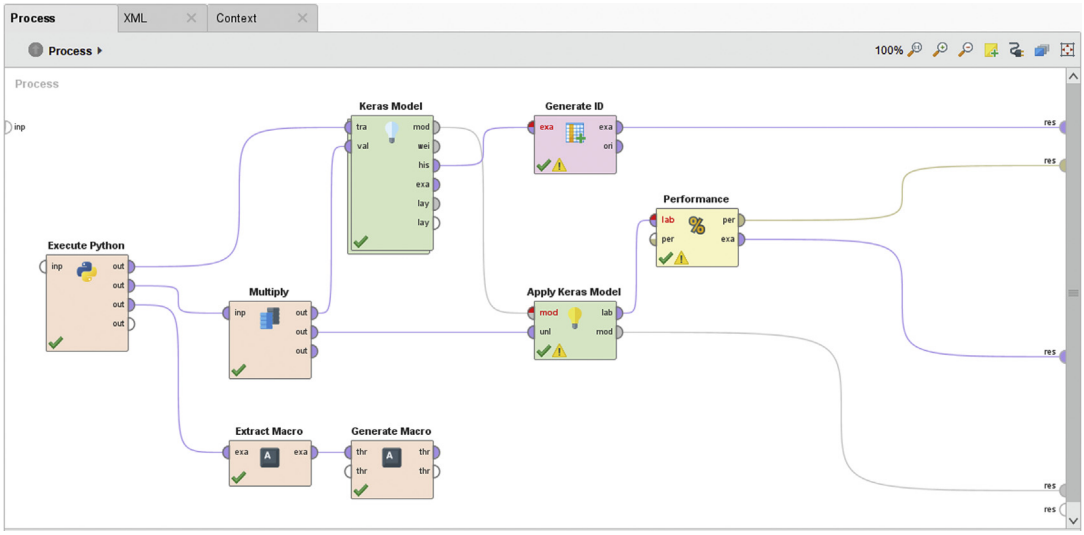

FIGURE 10.24
RapidMiner process for Deep learning

Step 1: Dataset Preparation

The key challenge in implementing this well-known model using RapidMiner is in preparing the dataset. RapidMiner expects the data in the form of a standard data frame (a table of rows as samples and columns as attributes) organized into rows and columns and in its current version (as of this publication) cannot use the raw image data. The raw data consist of 70,000 images which are 28 x 28 pixels. This needs to be transformed into a Pandas data frame which is 70,000 rows (or samples) by 784 columns (pixel values) and then split up into 60,000 sample training and 10,000 sample test sets. The 28 x 28 (pixels) yields 784 pixel values. The first

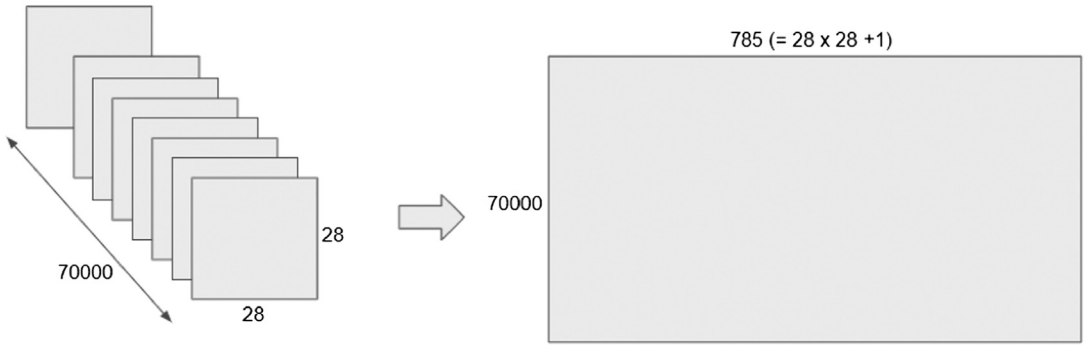

FIGURE 10.25
Reshaping the MNIST image tensor into a data frame.

operator in this process is a Python script executor which takes the raw data and transforms it. In the Execute Python operator (Note: this is very similar to the Execute R operator discussed in Chapter 15: Getting started with RapidMiner, and works analogously), the data is read from the MNIST dataset, its shape saved, the pixel data transformed into floats from integers, and finally both training and test "x" and "y" vectors are "unfolded" and merged into Pandas data frames with the "y" column defined as a "label" (using rm_metadata attribute). Fig. 10.25 shows graphically what the Execute Python block accomplishes. The shape information is alsoreturned as a data frame, so that it can later be used to set image sizes and shapes in the convolutional nets. The results from the Execute Python block consists of three data frames: (i) a training data frame 60,000 rows x 785 columns; (ii) test data frame 10,000 x 785; and (iii) and a shape information data frame which stores information about the data (each image is 28 x 28 x 1 tensor, the 1 refers to the channel). After flattening the 28 x 28 image we have 784 columns and we add one more column to contain the "label" information (i.e., which digit is encoded by the 784 columns).

Step 2: Modeling using the Keras Model

The rest of process is very straightforward after the image data is transformed. It passes the training data through the Keras Model and applies the model on the test dataset (Apply Keras Model) and then checks the model performance. The Extract Macro and Generate Macro operators pull the shape information into variables called img_shape, img_row, img_cols, and img_channels which are the used to reshape the data frame into a tensor that the Keras Model operator can then use. The main modeling operator for

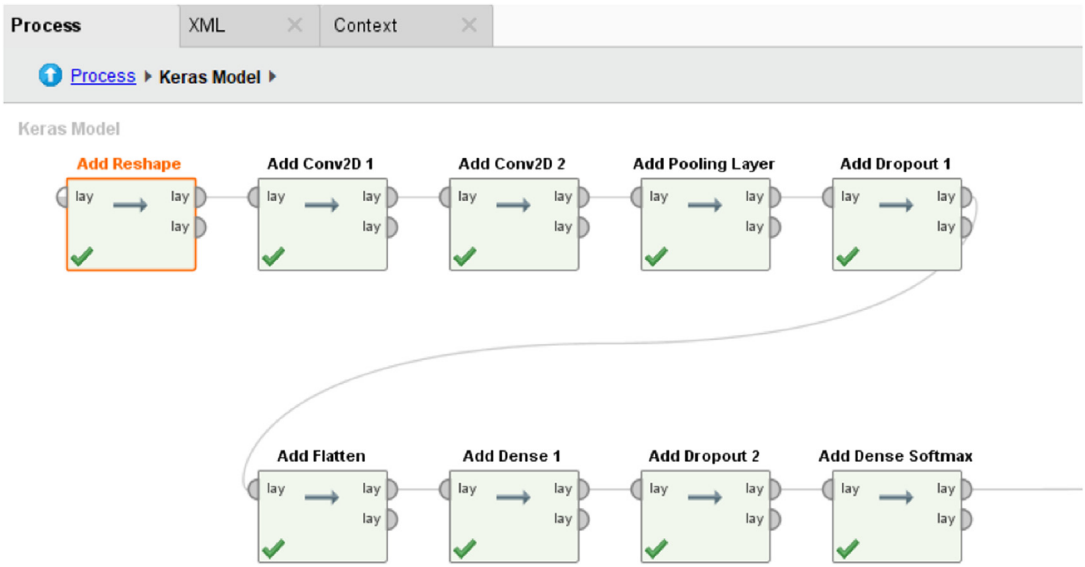

FIGURE 10.26
Deep learning subprocess for keras operators.

deep learning is the Keras Model operator under Keras > Modelling folder. The Keras operator is a meta operator which has a subprocess of deep-learning layers within it. The main parameters for the Keras modeling operator chosen are: Input shape: [img_size (=img_rows*img_cols*img_channels)], Loss: categorical_crossentropy, Optimizer: Adadelta, Learning rate, Epochs, batch_size are initially set at 1.0, 12 and 128 respectively. These can be adjusted later for optimal performance.

The architecture of the Keras deep learning network is shown in Fig. 10.26. The main thing to note is that we need to include an extra initial step (Add reshape) to fold the data back into its original form (28 x 28 x 1) based on the size and shape information passed into RapidMiner. There are two Conv2D layers: the first uses 32 filters ($n = 32$) and the second uses 64 filters. All filters are 3 x 3 (i.e., $f = 3$) and stride ($s = 1$). MaxPooling (with $f = 2$ and $s = 2$) and a Dropout layer which randomly eliminates 25% of the nodes (see Fig. 10.20), complete the CNN portion of the network before a flattened layer converts the 2D images into a column of 9216 nodes. This is connected to a 128-node layer, and another Dropout layer (50%) is applied before terminating in a 10-unit softmax output layer. As an exercise, the reader is encouraged to write this process as a schematic similar to Fig. 10.18 to make sure the internal workings of the network are clearer. As you move from one layer to the next, Keras

KerasModelIOObject

Layer (type)	Output Shape	Param #
reshape_1 (Reshape)	(None, 28, 28, 1)	0
conv2d_1 (Conv2D)	(None, 26, 26, 32)	320
conv2d_2 (Conv2D)	(None, 24, 24, 64)	18496
max_pooling2d_1 (MaxPooling2	(None, 12, 12, 64)	0
dropout_1 (Dropout)	(None, 12, 12, 64)	0
flatten_1 (Flatten)	(None, 9216)	0
dense_1 (Dense)	(None, 128)	1179776
dropout_2 (Dropout)	(None, 128)	0
dense_2 (Dense)	(None, 10)	1290

Total params: 1,199,882
Trainable params: 1,199,882
Non-trainable params: 0

FIGURE 10.27
A summary of the deep-learning model and its constituent layers.

automatically calculates the output size (i.e., number of output nodes for that layer). The user must make sure that the output of the previous layer is compatible with the input of the next layer by specifying the correct input_size. The reader should also try to calculate the number of weights or parameters that need to be learned at each layer. Fig. 10.27 shows the model summary. As seen, this model has more than one million weights (called "Trainable params") that need to be trained or learned making it a truly deep-learning model. Observe that the largest number of weights occur in the first dense layer.

Step 3: Applying the Keras Model

The model developed by the Keras modeling operator can be applied on the test dataset using Apply Keras model under Keras > Scoring folder. This

accuracy: 99.04%

	true 7	true 2	true 1	true 0	true 4	true 9	true 5	true 6	true 3	true 8	class precision
pred. 7	1015	4	0	1	0	0	0	0	0	1	99.41%
pred. 2	5	1023	1	0	0	0	0	0	2	2	99.03%
pred. 1	2	1	1130	0	0	1	0	2	0	0	99.47%
pred. 0	1	1	0	975	0	1	3	4	0	3	98.68%
pred. 4	0	1	0	0	969	3	0	1	0	0	99.49%
pred. 9	2	0	0	0	6	994	0	0	0	4	98.81%
pred. 5	0	0	0	1	0	5	881	1	1	1	98.99%
pred. 6	0	0	2	2	5	0	3	949	0	0	98.75%
pred. 3	2	1	2	0	0	3	5	0	1006	1	98.63%
pred. 8	1	1	0	1	2	2	0	1	1	962	99.07%
class recall	98.74%	99.13%	99.56%	99.49%	98.68%	98.51%	98.77%	99.06%	99.60%	98.77%	

FIGURE 10.28
Comparing the deep learning predicted values to actual values.

operator is similar to other Apply Model operators. The input for the operator is the test dataset with 10,000 labeled image data frame and the model. The output is the scored dataset. The scored output is then connected to the Performance (Classification) operator to compare the deep learning-based digits recognition with the actual value of the label.

Step 4: Results

Fig. 10.28 shows the results of the prediction performed by the CNN-based deep-learning model. As seen from the confusion matrix, the performance measured via recall, precision, or overall accuracy is around 99% across all the digits, indicating that the model performs exceptionally well. This implementation shows both the power and the complexity of the deep learning process.

10.4 CONCLUSION

Deep learning is a fast-growing field of research and its applications run the gamut of structured and unstructured data (text, voice, images, video, and others). Each day brings new variants of the deep architectures and with it, new applications. It is difficult to do justice to this field within the scope of a broad data science focused book. This chapter provided a brief high-level overview of the collection of techniques known as deep learning. An overview on how the majority of today's AI applications are supported solely by deep learning was provided. The fundamental similarity between deep learning and function fitting methods such as multiple linear regression and logistic regression were demonstrated. The essential differences between deep learning and traditional machine learning was also discussed. Some time was then spent detailing one of the most common deep learning techniques—convolutional neural networks and how it can be implemented using

Rapidminer was discussed. Finally, some of the other deep learning techniques were highlighted which are rapidly gaining ground in research and industry.

References

Explainable Artificial Intelligence. (2016). *Broad agency announcement*. Defense Advanced Research Projects Agency. <https://www.darpa.mil/program/explainable-artificial-intelligence>.

Hinton, G. E., & Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *Science, 313*, 504–507.

Le Cun, Y. (1989). Generalization and network design Strategies. *University of Toronto technical report CRG-TR-89-4*.

Marczyk, J., Hoffman, R., Krishnaswamy, P. (1999). *Uncertainty management in automotive crash: From analysis to simulation*. <https://pdfs.semanticscholar.org/2f26/c851cab16ee20925-c4e556eff5198d92ef3c.pdf>.

Minsky, M., and Papert, S (1969). *Perceptrons: an introduction to computational geometry*. Cambridge, Massachusetts: MIT Press.

Rosenblatt, F. (1957) The perceptron: A Probabilistic model for Visual Perception, Procs. of the 15th International Congress of Psychology, North Holland, 290-297.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature, 323*(9), 533–536.

Somers, J. (2017). Is AI riding a one-trick pony? MIT Technology Review, September 29, 2017. <https://www.technologyreview.com/s/608911/is-ai-riding-a-one-trick-pony/>.

Wissner-Gross, A. (2016). Datasets over algorithms. Edge <https://www.edge.org/response-detail/26587>.