Feedforward Neural Networks

4

4.1 Basic Concepts and Terminology for Neural Networks

Backpropagation is the core method of learning for deep learning. But before we can start exploring backpropagation, we must define a number of basic concepts and explain their interactions. Deep learning is machine learning with deep artificial neural networks, and the goal of this chapter explains how shallow neural networks work. We will also refer to shallow neural networks as *simple feedforward neural networks*, although the term itself should be used to refer to any neural network which does not have a feedback connection, not just shallow ones. In this sense, a convolutional neural network is also a feedforward neural network but not a shallow neural network. In general, deep learning consists of fixing the problems which arise when we try to add more layers to a shallow neural network. There are a number of other great books on neural networks. The book [1] offers the reader a rigorous treatment with most of the mathematical details written out, while the book [2] is more geared towards applications, but gives an overview of some connected techniques that we have not explored in this volume such as the Adaline. The book [3] is a great book written by some of the foremost experts in deep learning, and this book can be seen as a natural next step after completing the present volume. One final book we mention, and this book is perhaps the most demanding, is [4]. This is a great book, but it will place serious demands on the reader, and we suggest to tackle it after [3]. There are a number of other excellent books, but we offered here our selection which we believe will best augment the material covered in the present volume.

Any neural network is made of simple basic elements. In the last chapter, we encountered a simple neural network without even knowing it: the logistic regression. A shallow artificial neural network consists of two or three layers, anything more than that is considered deep. Just like a logistic regression, an artificial neural network has an input layer where inputs are stored. Every element which holds an input is called a 'neuron'. The logistic regression then has a single point where all inputs are

S. Skansi, *Introduction to Deep Learning*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-319-73004-2_4

directed, and this is its output (this is also a neuron). The same holds for a simple neural network, but it can have more than one output neuron making the output layer. What is different from logistic regression is that a 'hidden' layer may exist between the input and output layer. Depending on the point of view, we can think of a neural network being a logistic regression with not one but multiple workhorse neurons, and then after them, a final workhorse neuron which 'coordinates' their results, or we could think of it as a logistic regression with a whole layer of workhorse neurons squeezed between the inputs and the old workhorse neuron (which was already present in the logistic regression). Both of these views are useful for developing intuition on neural networks, and keep this in mind in the remainder of this chapter, since we will switch form one view to the other if it becomes convenient.

The structure of a simple three-layer neural network shown in Fig. 4.1. Every neuron of one layer is connected to all neurons of the next layer, but it gets multiplied by a so-called *weight* which determines how much of the quantity from the previous layer is to be transmitted to a given neuron of the next layer. Of course, the weight is not dependent on the initial neuron, but it depends on the initial neuron–destination neuron pair. This means that the link between say neuron N_5 and neuron M_7 has a weight w_k while the link between the neurons N_5 and M_3 has a different weight, w_j. These weights can happen to have the same value by accident, but in most cases, they will have different values.

The flow of information through the neural network goes from the first-layer neurons (input layer), via the second-layer neurons (hidden layer) to the third-layer neurons (output neurons). We return now to Fig. 4.1. The input layer consists of three neurons and each of them can accept one input value, and they are represented by variables x_1, x_2, x_3 (the actual input values will be the values for these variables). Accepting input is the *only* thing the first layer does. Every neuron in the input

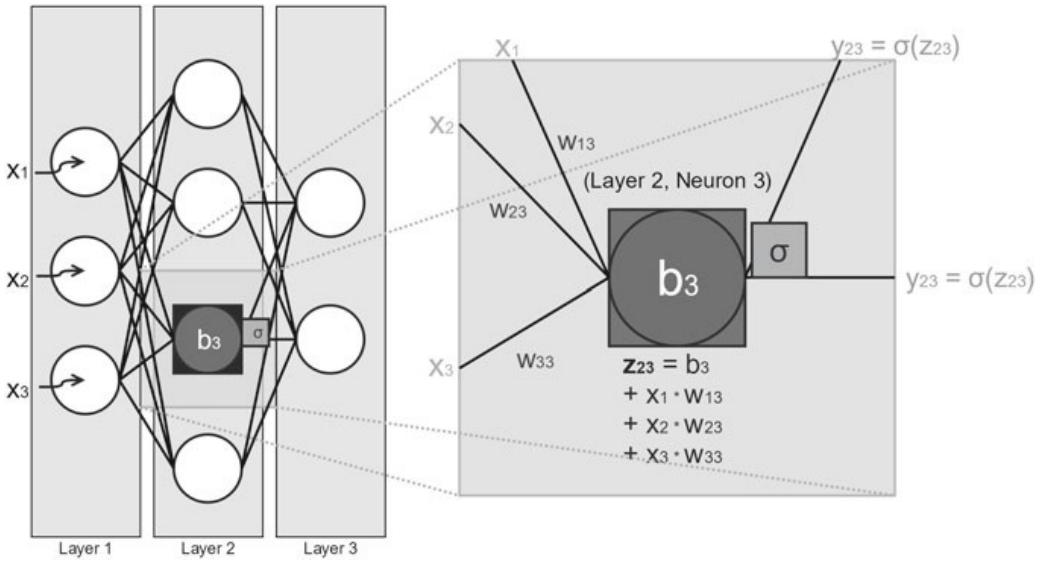

Fig. 4.1 A simple neural network

layer can take a single output. It is possible to have less input values than input neurons (then you can hand 0 to the unused neurons), but the network cannot take in more input values than it has input neurons. Inputs can be represented as a sequence $x_1, x_2, \ldots, x_n$ (which is actually the same as a *row vector*) or as a *column vector* $\mathbf{x} := (x_1, x_2, \ldots, x_n)^\top$. These are different representations of the same data, and we will always choose the representation that makes it easier and faster to compute the operations we might need. In our choice of data representation, we are not constrained by anything else but computational efficiency.

As we already noted, every neuron from the input layer is connected to every neuron from the hidden layer, but neurons of the same layer are not interconnected. Every connection between neuron j in layer k and neuron m in layer n has a weight denoted by w_{jm}^{kn}, and, since it is usually clear from the context which layers are concerned, we may omit the superscript and write simply w_{jm}. The weight regulates how much of the initial value will be forwarded to a given neuron, so if the input is 12 and the weight to the destination neuron is 0.25, the destination will receive the value 3. The weights can decrease the value, but they can also increase it since they are not bound between 0 and 1.

Once again we return to Fig. 4.1 to explain the zoomed neuron on the right-hand side. The zoomed neuron (neuron 3 from layer 2) gets the input which is the sum of the products of the inputs from the previous layer and respective weights. In this case, the inputs are x_3, x_2 and x_3, and the weights are w_{13}, w_{23} and w_{33}. Each neuron has a modifiable value in it, called the *bias*, which is represented here by b_3, and this bias is added to the previous sum. The result of this is called the *logit* and traditionally denoted by z (in our case, z_{23}).

Some simpler models[1] simply give the logit as the output, but most models apply a nonlinear function (also called a *nonlinearity* or *activation function* and represented by 'S' in Fig. 4.1) to the logit to produce the output. The output is traditionally denoted with y (in our case the output of the zoomed neuron is y_{23})[2] The nonlinearity can be generically refered to as $S(x)$ or by the name of the given function. The most common function used is the *sigmoid* or *logistic* function. We have encountered this function before, when it was the main function in logistic regression. The logistic function takes the logit z and returns as its output $\sigma(z) = \frac{1}{1+e^{-z}}$. The logistic function 'squashes' all it receives to a value between 0 and 1, and the intuitive interpretation of its meaning is that it calculates the probability of the output given the input.

A couple of remarks. Different layers may have different nonlinearities which we shall see in the later chapters, but all neurons of the same layer apply the same nonlinearity to its logits. Also, the output of a neuron is the same value in every direction it sends it. Returning to the zoomed neuron in Fig. 4.1, the neuron sends y_{23} in to directions, and both of them are the same value. As a final remark, following Fig. 4.1 again, note that the logits in the next layer will be calculated in the same manner. If we take, for example z_{31}, it will be calculated as $z_{31} = b_{31} + w_{11}^{23} y_{21} +$

[1] These models are called *linear neurons*.

[2] From linear neurons we still want to use the same notation but we set $y_{23} := z_{23}$.

$w_{21}^{23} y_{22} + w_{31}^{23} y_{23} + w_{41}^{23} y_{24}$. The same is done for z_{32}, and then by applying the chosen nonlinearity to z_{31} and z_{32} we obtain the final output.

4.2 Representing Network Components with Vectors and Matrices

Let us recall the general shape of a $m \times n$ matrix (m is the number of rows and n is the number of columns):

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \dots & a_{mn} \end{bmatrix}$$

Suppose we need to define with matrix operations the process sketched in Fig. 4.2.

In Chap. 3 we have seen how to represent the calculations for logistic regression with matrix operators. We follow the same idea here but for simple feedforward neural networks. If we want the input to follow the vertical arrangement as it is in the picture, we can represent it as a column vector, i.e. $\mathbf{x} = (x_1, x_2)^\top$. The Fig. 4.2 also offers us the intermediate values in the network, so we can verify each step of our calculation. As explained in the earlier chapters, if A is a matrix, the matrix entry in the j row and k column is denoted by $A_{j,k}$ or by A_{jk}. If we want to 'switch' the j and k, we need the transpose of the matrix A denoted $A^\top$. So for all entries in the matrices A and $A^\top$ the following holds: A_{jk} has the same value as $A_{kj}^\top$, i.e. $A_{jk} = A_{kj}^\top$. When representing operations in neural networks as vectors and matrices, we want to minimize the use of transpositions (since each one of them has a computational cost), and keep the operations as natural and simple as possible. On the other hand, matrix transposition is not that expensive, and it is sometimes better to keep things intuitive rather than fast. In our case, we will want to represent a weight w which connects the

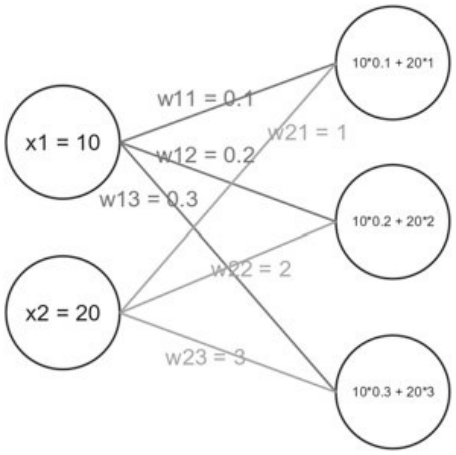

Fig. 4.2 Weights in a network

second neuron in layer 1 and the third neuron in layer 2 with a variable named w_{23}. We see that the index retains information on which neurons in the layers are connected, but one might ask where do we store the information which layers are in question. The answer is very simple, that information is best stored in the matrix name in the program code, e.g. `input_to_hidden_w`. Note that we can call a matrix by its 'mathematical name', e.g. $\mathbf{u}$ or by its 'code name' e.g. `hidden_to_output_w`. So, following Fig. 4.2 we write the weight matrix connecting the two layers as:

$$\begin{bmatrix} w_{11}(=0.1) & w_{12}(=0.2) & w_{13}(=0.3) \\ w_{21}(=1) & w_{22}(=2) & w_{23}(=3) \end{bmatrix}$$

Let us call this matrix $\mathbf{w}$ (we can add subscripts or superscripts to its name). Using matrix multiplication $\mathbf{w}^\top \mathbf{x}$ we get a 3×1 matrix, namely the column vector $\mathbf{z} = (21, 42, 63)^\top$.

With this we have described, alongside the structure of the neurons and connections the forwarding of data through the network which is called *the forward pass*. The forward pass is simply the sum of calculations that happen when the input travels through the neural network. We can view each layer as computing a function. Then, if $\mathbf{x}$ is the input vector, $\mathbf{y}$ is the output vector and f_i, f_h and f_o are the overall functions calculated at each layer, respectively,(products, sums and nonlinearities), we can say that $\mathbf{y} = f_o(f_h(f_i(\mathbf{x})))$. This way of looking at a neural network will be very important when we will address the correction of weights through backpropagation.

For a full specification of a neural network we need:

- The number of layers in a network
- The size of the input (recall that this is the same as the number of neurons in the input layer)
- The number of neurons in the hidden layer
- The number of neurons in the output layer
- Initial values for weights
- Initial values for biases

Note that the neurons are not objects. They exist as entries in a matrix, and as such, their number is necessary for specifying the matrices. The weights and biases play a crucial role: the whole point of a neural network is to find a good set of weights and biases, and this is done through training via *backpropagation*, which is the reverse of a forward pass. The idea is to measure the error the network makes when classifying and then modify the weight so that this error becomes very small. The remainder of this chapter will be devoted to backpropagation, but as this is the most important subject in deep learning, we will introduce it slowly and with numerous examples.

4.3 The Perceptron Rule

As we noted before, the learning process in the neurons is simply the modification or update of weights and biases during training with backpropagation. We will explain the backpropagation algorithm shortly. During classification, only the forward pass is made. One of the early learning procedures for artificial neurons is known as *perceptron learning*. The perceptron consisted of a *binary threshold neuron* (also known as binary threshold units) and the perceptron learning rule and altogether looks like a modified logistic regression. Let us formally define the binary threshold neuron:

$$z = b + \sum_i w_i x_i$$

$$y = \begin{cases} 1, z \geq 0 \\ 0, \text{otherwise} \end{cases}$$

Where x_i are the inputs, w_i the weights, b is the bias and z is the logit. The second equation defines the decision, which is usually done with the nonlinearity, but here a binary step function is used instead (hence the name). We take a digression to show that it is possible to absorb the bias as one of the weights, so that we only need a weight update rule. This is displayed in Fig. 4.3: to absorb the bias as a weight, one needs to add an input x_0 with value 1 and the bias is its weight. Note that this is *exactly* the same:

$$z = b + \sum_i w_i x_i = w_0 x_0 (= b) + w_1 x_1 + w_2 x_2 + \ldots$$

According to the above equation, b could either be x_0 or w_0 (the other one must be 1). Since we want to change the bias with learning, and inputs never change, we must treat it as a weight. We call this procedure *bias absorption*.

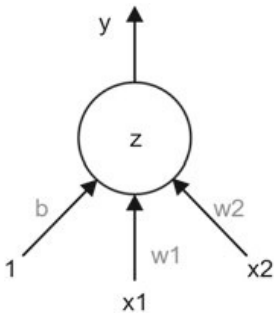

Fig. 4.3 Bias absorption

The perceptron is trained as follows (this is the perceptron learning rule[3]):

1. Choose a training case.
2. If the predicted output matches the output label, do nothing.
3. If the perceptron predicts a 0 and it should have predicted a 1, add the input vector to the weight vector
4. If the perceptron predicts a 1 and it should have predicted a 0, subtract the input vector from the weight vector

As an example, take the input vector to be $\mathbf{x} = (0.3, 0.4)^\top$ and let the bias be $b = 0.5$, the weights $\mathbf{w} = (2, -3)^\top$ and the target[4] $t = 1$. We start by calculating the current classification result:

$$z = b + \sum_i w_i x_i = 0.5 + 2 \cdot 0.3 + (-3) \cdot 0.4 = -0.1$$

As $z < 0$, the output of the perceptron is 0 and should have been 1. This means that we have to use clause (3) from the perceptron rule and add the input vector to the weight vector:

$$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) + (\mathbf{x}, 1) = (2, -3, 0.5) + (0.3, 0.4, 1) = (2.3, -2.6, 1.5)$$

If adding handcrafted features is not a option, the perceptron algorithm is very limited. To see a simple problem that Minsky and Papert exposed in 1969 [5], consider that each classification problem can be understood as a query on the data. This means that we have a property we want the input to satisfy. Machine learning is just a method for defining this complex property in terms of the (numerical) properties present in the input. A query then retrieves all the input points satisfying this property. Suppose we have a dataset consisting of people and their height and weight. To return only those higher than say 175 cm, one would make a query of the form `select * from table where cm>175`. If we, on the other hand, only have jpg files of mugshots with the black and white meter behind the faces, then we would need a classifier to determine the people's height and then sort them accordingly. Note that this classifier would not use numbers, but rather pixels, so it might find people of e.g. 155 cm similar to those of height 175, but not those of 165, since the black and white parts of the background are similar. This means that the machine learning algorithm learns 'similar' in terms of the information representation it is given: what might seem similar in terms of numbers might not be similar in terms of pixels and vice versa. Consider the numbers 6 and 9: visually they are close (just rotate one to get

[3] Formally speaking, all units using the perceptron rule should be called perceptrons, not just binary threshold units.

[4] The target is also called expected value or true label, and it is usually denoted by t.

the other) but numerically they are not. If the representation given to an algorithm is in pixels, and it can be rotated,[5] the algorithm will consider them the same.

When classifying, the machine learning algorithm (and perceptrons are a type of machine learning algorithms) selects some datapoints as belonging to a class and leaves the other out. This means that some of them get the label 1 and some get the label 0, and this learned partitioning hopefully captures the underlying reality: that the datapoints labelled 1 really are 'ones' and the datapoints labelled 0 really are 'zeros'. A classic query in logic and theoretical computer science is called *parity*. This query is done over binary strings of data, and only those with an equal number of ones and zeros are selected and given the label 1. Parity can be relaxed so it considers only strings of length n, then we can formally name it $\texttt{parity}_n(x_0, x_1, \ldots, x_n)$, where each x_i is a single binary digit (or *bit*). $\texttt{parity}_2$ is also called XOR and it is also a logical function called exclusive disjunction. XOR takes two bits and returns 1 if and only if there is the same amount of 1 and 0, and since they are binary strings, this means that there is one 1 and one 0. Note that we can equally use the logical equivalence which has the resulting 0 and 1 exchanged, since they are just names for classes and do not carry much more meaning. So XOR gives the following mapping: $(0, 0) \mapsto 0$, $(0, 1) \mapsto 1$, $(1, 0) \mapsto 1$, $(1, 1) \mapsto 0$.

When we have XOR as the problem (or any instance of parity for that matter), the perceptron is unable to learn to classify the input so that they get the correct labels. This means that a perceptron that has two input neurons (for accepting the two bits for XOR) cannot adjust its two weights to separate the 1 and 0 as they come in the XOR. More formally, if we denote by w_1, w_2 and b the weights and biases of the perceptron, and take the following instance of parity $(0, 0) \mapsto 1$, $(0, 1) \mapsto 0$, $(1, 0) \mapsto 0$ i $(1, 1) \mapsto 1$, we get four inequalities:

1. $w_1 + w_2 \geq b$,
2. $0 \geq b$,
3. $w_1 < b$,
4. $w_2 < b$

The inequality (a) holds since if $(x_1 = 1, x_2 = 1) \mapsto 1$, and we can get 1 as an output only if $w_1x_1 + w_2x_2 = w_1 \cdot 1 + w_2 \cdot 1 = w_1 + w_2$ is greater or equal b, which means $w_1 + w_2 \geq b$.

The inequality (b) holds since if $(x_1 = 0, x_2 = 0) \mapsto 1$, and we can get 1 as an output only if $w_1x_1 + w_2x_2 = w_1 \cdot 0 + w_2 \cdot 0 = 0$ is greater or equal b, which means $0 \geq b$.

The inequality (c) holds since if $(1, 0) \mapsto 0$, then $w_1x_1 + w_2x_2 = w_1 \cdot 1 + w_2 \cdot 0 = w_1$, and for the perceptron to give 0, w_1 has to be less than the bias b, i.e. $w_1 < b$.

[5] As a simple application, think of an image recognition system for security cameras, where one needs to classify numbers seen regardless of their orientation.

The inequality (d) is derived in a similar fashion to (c). By adding (a) and (b) we get $w_1 + w_2 \geq 2b$, and by adding (c) and (d) we get $w_1 + w_2 < 2b$. It is easy to see that the system of inequalities has no solution.

This means that the perceptron, which was claimed to be a contendant for general artificial intelligence could not even learn logical equality. The proposed solution was to make a 'multilayered perceptron'.

4.4 The Delta Rule

The main problem with making the 'multilayered perceptron' is that it is unknown how to extend the perceptron learning rule to work with multiple layers. Since multiple layers are needed, the only option seemed to be to abandon the perceptron rule and use a different rule which is more robust and capable of learning weights accross layer. We already mentioned this rule—*backpropagation*. It was first discovered by Paul Werbos in his PhD thesis [6], but it remained unnoticed. It was discovered for the second time by David Parker in 1981, who tried to get a patent but he subsequently published it in 1985 [7]. The third and the last time it was discovered independently by Yann LeCun in 1985 [8] and by Rumelhart, Hinton and Williams in 1986 [9].

To see what we want to archive, let us consider an example[6] imagine that each day we buy lunch at the nearby supermarket. Every day our meal consists of a piece of chicken, two grilled zucchinis and a scoop of rice. The cashier just gives us the total amount, which varies each day. Suppose that the price of the components does not very over time and that we can weight the food to see how much we have. Note that one meal will not be enough to deduce the prices, since we have three of them[7] and we do not know which component is responsible in what proportion for a total price increase in one euro.

Notice that the price per kilogram is actually similar to the neural network weight. To see this think of how you would find the price per kilogram of the meal components: you make a guess on the prices per kilogram for the components, multiply with the quantity you got today and compare their sum to the price you have actually paid. You will see that you are off by e.g. 6€. Now you must find out which components are 'off'. You could stipulate that each component is off by 2€ and then readjust your stipulated price per kilogram by the 2€ and wait for you next meal to see whether it will be better now. Of course you could have also stipulated that the components are off by 3, 2, 1€ respectively, and either way, you would have to wait for your next meal with your new price per kilograms and try again to see whether you will be off by a lesser or greater amount. Of course, you want to correct your

[6] This is a modified version of an example given by Geoffrey Hinton.

[7] For example, if we only buy chicken, then it would be easy to get the price of the chicken analytically as $total = price \cdot quantity$, and we get $price = \frac{total}{quantity}$.

estimations so that you are off by less and less as the meals pass, and hopefully, this will lead you to a good approximation.

Note that there exists a *true* price per kilogram but we do not know it, and our method is trying to discover it just by measuring how much we miss the total price. There is a certain 'indirectness' in this procedure and this is highly useful and the essence of neural networks. Once, we find our good approximations, we will be able to calculate with appropriate precision the total price of all of our future meals, without needing to find out the actual prices.[8]

Let us work a bit more this example. Each meal has the following general form:

$$total = ppk_{chicken} \cdot quant_{chicken} + ppk_{zucchini} \cdot quant_{zucchini} + ppk_{rice} \cdot quant_{rice}$$

where *total* is the total price, the *quant* is the quantity and the *ppk* is the price per kilogram for each component. Each meal has a total price we know, and the quantities we know. So each meal places a *linear constraint* on the *ppk*-s. But with only this we cannot solve it. If we plug in this formula our initial (or subsequenlty corrected) 'guesstimate'[9] we will get also the predicted value, and by comparing it with the true (target) total value we will also get an error value which will tell us by how much we missed. If after each meal we miss by less, we are doing a great job.

Let us imagine that the true price is $ppk_{chicken} = 10$, $ppk_{zucchini} = 3$, and $ppk_{rice} = 5$. Let us start with a guesstimate of $ppk_{chicken} = 6$, $ppk_{zucchini} = 3$, and $ppk_{rice} = 3$. We know we bought 0.23 kg of chicken, 0.15 kg of zucchini and 0.27 kg of rice and that we paid 3 € in total. By multiplying our guessed prices with the quantities we get 1.38, 0.45 and 0.81, which totals to 2.64, which is 0.35 less than the true price. This value is called the *residual error*, and we want to minimize it over the course of future iterations (meals), so we need to distribute the residual error to the *ppk*-s. We do this simply by changing the *ppk*-s by:

$$\Delta ppk_i = \frac{1}{n} \cdot quant_i(t - y)$$

where $i \in \{chicken, zucchini, rice\}$, n is the cardinality (number of elements) of this set (i.e. 3), $quant_i$ is the quantity of i, t is the total price and y is the predicted total price. This is known as the *delta rule*. When we rewrite this in standard neural network notation it looks like:

$$\Delta w_i = \eta x_i(t - y)$$

[8] In practical terms this might seem far more complicated than simply asking the person serving you lunch the price per kilogram for components, but you can imagine that the person is the soup vendor from the soup kitchen from the TV show Seinfeld (116th episode, or S07E06).

[9] A guessed estimate. We use this term just to note that for now, we should keep things intuitive an not guess an initial value of, e.g. 12000, 4533233456, 0.0000123, not because it will be impossible to solve it, but because it will need much more steps to assume a form where we could see the regularities appear.

where w_i is a weight, x_i is the input and $t - y$ is the residual error. The η is called the *learning rate*. Its default value should be $\frac{1}{n}$, but there is no constraint placed on it so values like 10 are perfectly ok to use. In practice, however, we want the values for η to be small, and usually of the form 10^{-n}, meaning 0.1, 0.01, etc., but values such as 0.03 or 0.0006 are also used. The learning rate is an example of a *hyperparameter*, which are parameters in the neural network which cannot be learned like regular parameters (like weights and biases) but have to be adjusted by hand. Another example of a hyperparameter is the hidden layer size.

The learning rate controls how much of the residual error is handed down to the individual weights to be updated. The proportional distribution of $\frac{1}{n}$ is not that important if the learning rate is close to that number. For example, if $n = 90$ it is virtually the same if one uses the proportional learning rate of $\frac{1}{90}$ or a learning rate of 0.01. From a practical point of view, it is best to use a learning rate close to the proportional learning rate or smaller. The intuition behind using a smaller learning rate than the proportional is to update the weights only a bit in the right direction. This has two effects: (i) the learning takes longer and (ii) the learning is much more precise. The learning takes longer since with a smaller learning rate each update make only a part of the change needed, and it is more precise since it is much less likely to be overinfluenced by one learning step. We will make this more clear later.

4.5 From the Logistic Neuron to Backpropagation

The delta rule as defined above works for a simple neuron called the *linear neuron*, which is even simpler than the binary threshold unit:

$$y = \sum_i w_i x_i = \mathbf{w}^\top \mathbf{x}$$

To make the delta rule work, we will be needing a function which should measure if we got the result right, and if not, by how much we missed. This is usually called an *error function* or *cost function* and traditionally denoted by $E(x)$ or by $J(x)$. We will be using the *mean squared error*:

$$E = \frac{1}{2} \sum_{n \in train} (t^{(n)} - y^{(n)})^2$$

where the ($t^{(n)}$ denotes the target for the training case n (same for ($y^{(n)}$, but this is the prediction). The training case n is simply a training example, such as a single image or a row in a table. The mean squared error sums the error across all the training cases n, and after that we will update the weights. The natural choice for measuring how far were we from the bullseye would be to use the absolute value as a measure of distance that does not depend on the sign, but the reason behind choosing the square of the difference is that by simply squaring the difference we get a measure similar

to absolute values (albeit larger in magnitude, but this is not a problem, since we want to use it in relative, not absolute terms), but we will get as a bonus some nice properties to work with down the road.

Let us see, how we can derive the delta rule from the SSE to see that they are the same.[10] We start with the above equation defining the mean squared error and differentiate E with respect to w_i and get:

$$\frac{\partial E}{\partial w_i} = \frac{1}{2} \sum_n \frac{\partial y^{(n)}}{\partial w_i} \frac{dE^{(n)}}{dy^{(n)}}$$

The partial derivatives are here just because we have to consider a single w_i and treat all others as constants, but the overall behaviour apart from that is the same as with ordinary derivatives. The above formula tells us a story: it tells us that to find out how E changes with respect to w_i, we must find out how $y^{(n)}$ changes with respect to w_i and how E changes with respect to $y^{(n)}$. This is a nice example of the chain rule of derivations in action. We explored the chain rule in the second chapter but we will give a cheatsheet for derivations shortly so you do not have to go back. Informally speaking, the chain rule is similar to fraction multiplication, and if one recalls that a shallow neural network is a structure of the general form $\mathbf{y} = f_o(f_h(f_i(\mathbf{x})))$, it is easy to see that there will be a lot of places to use the chain rule, especially as we go on to deep learning and add more layers.

We will explain the derivations shortly. The above equation means the weight updates are proportional to the error derivations in all training cases added together:

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} = \sum_n \eta x_i^{(n)} (t^{(n)} - y^{(n)})$$

Let us proceed to the actual derivation. We will be deriving the result for a logistic neuron (also called a *sigmoid* neuron), which we have already presented before, but we will define it once more:

$$z = b + \sum_i w_i x_i$$

$$y = \frac{1}{1 + e^{-z}}$$

Recall that z is the logit. Let us absorb the bias right away, so we do not have to deal with it separately. We will calculate the derivation of the logistic neuron with respect to the weights, and the reader can adapt the procedure to the simpler linear neuron if she likes. As we noted before, the chain rule is your best friend for obtaining derivations, and the 'middle variable' of the chain rule will be the logit. The first

[10] Not in the sense that they are the same formula, but that they refer to the same process and that one can be derived from the other.

part $\frac{\partial z}{\partial w_i}$ which is equal to x_i since $z = \sum_i w_i x_i$ (we absorbed the bias). By the same argument $\frac{\partial z}{\partial x_i} = w_i$.

The derivation of the output with respect to the logit is a simple expression ($\frac{dy}{dz} = y(1-y)$) but is not easy to derive. Let us restate the derivation rules we use[11]

- `LD`: Differentiation is linear, so we can differentiate the summands separately and take out the constant factors: $[f(x)a + g(x)b]' = a \cdot f'(x) + b \cdot g'(x)$
- `Rec`: Reciprocal rule $[\frac{1}{f(x)}]' = -\frac{f'(x)}{f(x)^2}$
- `Const`: Constant rule $c' = 0$
- `ChainExp`: Chain rule for exponents $[e^{f(x)}]' = e^{f(x)} \cdot f'(x)$
- `DerDifVar`: Deriving the differentiation variable $\frac{dy}{dz} z = 1$
- `Exp`: Exponent rule $[f(x)^n]' = n \cdot f(x)^{n-1} \cdot f'(x)$

We can now start deriving $\frac{dy}{dz}$. We start with the definition for y, i.e. with

$$\frac{dy}{dz} \frac{1}{1+e^{-z}}$$

From this expression by application of the `Rec` rule we get

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

From this by applying `LD` we get

$$-\frac{\frac{dy}{dz}1 + \frac{dy}{dz}e^{-z}}{(1+e^{-z})^2}$$

On the first summand in the numerator, we apply `Const` and it becomes 0, and on the second we apply `ChainExp` and it becomes $e^{-z} \cdot \frac{dy}{dz}(-z)$, and so we have

$$-\frac{e^{-z} \cdot \frac{dy}{dz}(-z)}{(1+e^{-z})^2}$$

By applying `LD` to the constant factor -1 implicit with z we get

$$-\frac{-1 \cdot \frac{dy}{dz} z \cdot e^{-z}}{(1+e^{-z})^2}$$

[11] For the sake of easy readability, we deliberately combine Newton and Leibniz notation in the rules, since some of them are more intuitive in one, while some of them are more intuitive in the second. We refer the reader back to Chap. 1 where all the formulations in both notations were given.

which by `DerDifVar` becomes

$$-\frac{-1 \cdot e^{-z}}{(1+e^{-z})^2}$$

We tidy up the signs and get

$$\frac{e^{-z}}{(1+e^{-z})^2}$$

Therefore,

$$\frac{dy}{dz} = \frac{e^{-z}}{(1+e^{-z})^2}$$

Let us factorize the right-hand side in two factors which we will call A and B:

$$\frac{e^{-z}}{(1+e^{-z})^2} = \frac{1}{1+e^{-z}} \cdot \frac{e^{-z}}{1+e^{-z}}$$

It is obvious that $A = y$ from the definition of y. Let us turn our attention to B:

$$\frac{e^{-z}}{1+e^{-z}} = \frac{(1+e^{-z})-1}{1+e^{-z}} = \frac{1+e^{-z}}{1+e^{-z}} - \frac{1}{1+e^{-z}} = 1 - \frac{1}{1+e^{-z}} = 1 - y$$

Therefore $A = y$ and $B = 1 - y$, and $\frac{dy}{dz} = A \cdot B$, from which follows that

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

Since we have $\frac{\partial z}{\partial w_i}$ and $\frac{dy}{dz}$ with the chain rule we get

$$\frac{\partial y}{\partial w_i} = x_i y(1-y)$$

The next thing we need is $\frac{dE}{dy}$.[12] We will be using the same rules for this derivation as we did for $\frac{dy}{dz}$. Recall that $E = \frac{1}{2}(t^{(n)} - y^{(n)})^2$, but we will use the version $E = \frac{1}{2}(t-y)^2$ which is focused on a single target value t and a single prediction y.

Therefore, we need to find

$$\frac{dE}{dy}[\frac{1}{2}(t-y)^2]$$

[12] Strictly speaking, we would need $\frac{\partial E}{\partial y^{(n)}}$ but this generalization is trivial and we chose the simplification since we wanted to improve readability.

By applying `LD` we get

$$\frac{1}{2}\frac{dE}{dy}(t-y)^2$$

By applying `Exp` we get

$$\frac{1}{2}\cdot 2\cdot (t-y)\cdot \frac{dE}{dy}(t-y)$$

Simple cancellation yields

$$(t-y)\cdot \frac{dE}{dy}(t-y)$$

With `LD` we get

$$(t-y)\cdot \frac{dE}{dy}t\cdot \frac{dE}{dy}y$$

Since t is a constant, its derivative is 0 (rule `Const`), and since y is the differentiation variable, its derivative is 1 (`DerDiVar`). By tidying up the expression we get $(t-y)(0-1)$ and finally, $-1\cdot(t-y)$.

Now, we have all the elements for formulating the learning rule for the logistic neuron using the chain rule:

$$\frac{\partial E}{\partial w_i}=\sum_n \frac{\partial y^{(n)}}{\partial w_i}\frac{\partial E}{\partial y^{(n)}}=-\sum_n x_i^{(n)}y^{(n)}(1-y^{(n)})(t^{(n)}-y^{(n)})$$

Note that this is very similar to the delta rule for the linear neuron, but it has also $y^{(n)}(1-y^{(n)})$ extra: this part is the slope of the logistic function.

4.6 Backpropagation

So far we have seen how to use derivatives to learn the weights of a logistic neuron, and without knowing it we have already made excellent progress with understanding backpropagation, since backpropagation is actually the same thing but applied more than once to 'backpropagate' the errors through the layers. The logistic regression (consisting of the input layer and a single logistic neuron), strictly speaking, did not need to use backpropagation, but the weight learning procedure described in the previous section actually *is* a simple backpropagation. As we add layers, we will not have more complex calculations, but just a large number of those calculations. Nevertheless, there are some things to watch out for.

We will write out all the necessary details for backpropagation for the feedforward neural networks, but first, we will build up the intuition behind it. In Chap. 2 we have explained gradient descent, and we will revisit some of the concepts here as

needed. *Backpropagation of errors is basically just gradient descent*. Mathematically speaking, backpropagation is:

$$w_{updated} = w_{old} - \eta \nabla E$$

where w is the weigh, η is the learning rate (for simplicity you can think of it just being 1 for now) and E is the cost function measuring overall performance. We could also write it in computer science notation as a rule that assigns to w a new value:

$$w \leftarrow w - \eta \nabla E$$

This is read as 'the new value of w is w minus $\eta \nabla E$'. This is not circular,[13] since it is formulated as an assignment ($\leftarrow$), not a definition ($=$ or $:=$). This means that first, we calculate the right-hand side, and then we assign to w this new value. Notice that if were to write out this mathematically, we would have a recursive definition.

We may wonder whether we could do weight learning in a more simple manner, without using derivatives and gradient descent.[14] We could try the following approach: select a weight w and modify it a bit and see if that helps. If it does, keep the change. If it makes things worse, then change it in the opposite direction (i.e. instead of adding the small amount from the weight, subtract it). If this makes it better keep the change. If neither change improves the final result, we can conclude that w is perfect as it is and move to the next weight v.

Three problems arise right away. First, the process takes a long time. After the weight change, we need to process at least a couple of training examples for each weight to see if it is better or worse than before. Simply speaking, this is a computational nightmare. Second, by changing the weights individually we will never find out whether a combination of them would work better, e.g. if you change w or v separately (either by adding the small amount or subtracting to one or the other), it might make the classification error worse, but if you were to change them by adding a small amount to both of them it would make things better. The first of these problems will be overcome by using gradient descent,[15] while the second will be only partially resolved. This problem is usually called *local optima*.

The third problem is that near the end of learning, changes will have to be small, and it is possible that the 'small change' our algorithm test will be too large to successfully learn. Backpropagation also has this problem, and it is usually solved by using a dynamic learning rate which gets smaller as the learning progresses.

---

[13] A definition is circular if the same term occurs in both the *definiendum* (what is being defined) and *definiens* (with which it is defined), i.e. on both sides of $=$ (or more precisely of $:=$) and in our case this term could be w. A recursive definition has the same term on both sides, but on the defining side (definiens) it has to be 'smaller' so that one could resolve the definition by going back to the starting point.

[14] If you recall, the perceptron rule also qualifies as a 'simpler' way of learning weights, but it had the major drawback that it cannot be generalized to multiple layers.

[15] Although it must be said that the whole field of deep learning is centered around overcoming the problems with gradient descent that arise when using it in deep networks.

If we formalize this approach we will get a method called *finite difference approximation*[16]:

1. Each weight w_i, $1 \leq i \leq k$ is adjusted by adding to it a small constant ε (e.g. whose value is 10^{-6}) and the overall error (with only w_i changed) is evaluated (we will denote this by E_i^+)
2. Change back the weight to its initial value w_i and subtract ε from it and reevaluate the error (this will be E_i^-)
3. Do this for all weights w_j, $\leq j \leq k$
4. Once finished, the new weights will be set to $w_i \leftarrow w_i - \frac{E_i^+ - E_i^-}{2\varepsilon}$

The finite difference approximation does a good job in approximating the gradient, and nothing more than elementary arithmetic is used. If we recall what a derivative is and how it is defined from Chap. 2, the finite difference approximation makes sense even in terms of the 'meaning' of the procedure. This method can be used to build up the intuition how weight learning proceeds in full backpropagation. However, most current libraries which have tools for automatic differentiation perform gradient descent in a fraction of the time it would take to compute the finite difference approximation. Performance issues aside, the finite difference approximation would indeed work in a feedforward neural network.

Now, we turn to backpropagation. Let us examine what happens in the hidden layer of the feedforward neural network. We start with randomly initialized weights and biases, multiply them with the inputs, add them together, and take them through the logistic regression which "flattens" them to a value between 0 and 1, and we do that one more time. At the end, we get a value between 0 and 1 from the logistic neuron in the output layer. We can say that everything above 0.5 is 1 and below is 0. But the problem is that if the network gives a 0.67 and the output should have been 0, we know only the error the network produced (the function E), and we should use this. More precisely, we want to measure how E changes when the w_i change, which means that we want to find the derivative of E with regard to the activities of the hidden layer. We want to find all the derivatives at the same time, and for this, we use vector and matrix notations and, consequently, the gradient. Once we have the derivatives of E with regard to the hidden layer activity, we will easily compute the changes for the weights themselves.

We will address the procedure illustrated in Fig. 4.4. To keep the exposition as clear as possible, we will use only two indices, as if each layer has only one neuron. In the following section, we shall expand this to a fully functional feedforward neural network. As illustrated in Fig. 4.4 we will use the subscripts o for the output layer and h for the hidden layer. Recall that z is the logit, i.e. everything except the application of the nonlinearity.

[16]Cf. G. Hinton's Coursera course, where this method is elaborated.

As we have

$$E = \frac{1}{2} \sum_{o \in Output} (t_o - y_o)^2$$

the first thing we need to do is turn the difference between the output and the target value into an error derivation. We have done this already in the previous sections of this chapter:

$$\frac{\partial E}{\partial y_o} = -(t_o - y_o)$$

Now, we need to reformulate the error derivative with regard to y_o into an error derivative with regard to z_o. For this, we use the chain rule:

$$\frac{\partial E}{\partial z_o} = \frac{\partial y_o}{\partial z_o} \frac{\partial E}{\partial y_o} = y_o(1 - y_o) \frac{\partial E}{\partial y_o}$$

Now we can calculate the error derivative with respect to y_h:

$$\frac{\partial E}{\partial y_h} = \sum_o \frac{dz_o}{dy_h} \frac{\partial E}{\partial z_o} = \sum_o w_{ho} \frac{\partial E}{\partial z_o}$$

These steps we made from $\frac{\partial E}{\partial y_o}$ to $\frac{\partial E}{\partial y_h}$ are the heart of backpropagation. Notice that now we can repeat this to go through as many layers as we want. There will be a catch though, but for now is all good. A few remarks about the above equation. From the previous section, when we addressed the logistic neuron we know that $\frac{dz_o}{dy_h} = w_{ho}$. Once, we have $\frac{\partial E}{\partial z_o}$ it is very simple to get the error derivative with regard to the weights:

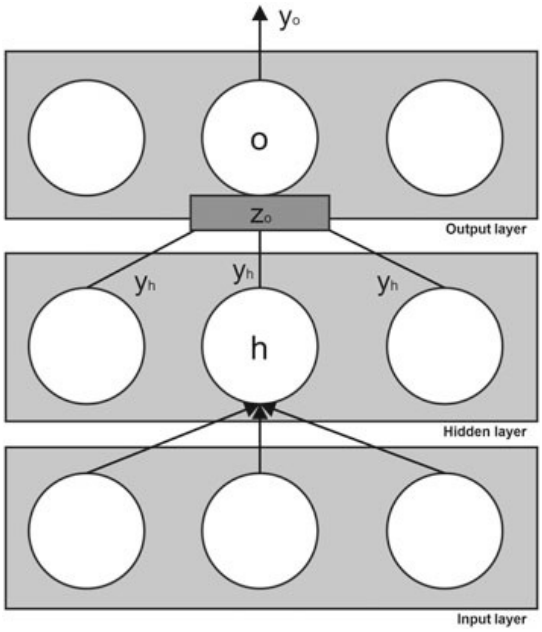

Fig. 4.4 Backpropagation

$$\frac{\partial E}{\partial w_{ho}} = \frac{\partial z_o}{\partial w_{ho}} \frac{\partial E}{\partial z_o} = y_i \frac{\partial E}{\partial z_j}$$

The rule for updating weights is quite straightforward, and we call it the *general weight update rule*:

$$w_i^{new} = w_i^{old} + (-1)\eta \frac{\partial E}{\partial w_i^{old}}$$

The η is the learning rate and the factor -1 is here to make sure we go towards minimizing E, otherwise we would be maximizing it. We can also state it in vector notation[17] to get rid of the indices:

$$\mathbf{w}^{new} = \mathbf{w}^{old} - \eta \nabla E$$

Informally speaking, the learning rate controls by how much we should update. There are a couple of possibilities (we will discuss the learning rate in more detail later):

1. Fixed learning rate
2. Adaptable global learning rate
3. Adaptable learning rate for each connection

We will address these issues in more detail later, but before that, we will show a detailed calculation for error backpropagation in a simple neural network, and in the next section, we will code the network. The remainder of this chapter is probably the most important part of the whole book, so be sure to go through all the details.

Let us see a working example[18] of a simple and shallow feedforward neural network. The network is represented in Fig. 4.5. Using the notation, the starting weights and the inputs specified in the image, we will calculate all the intricacies of the forward pass and backpropagation for this network. Notice the enlarged neuron D. We have used this to illustrate, where the logit z_D is and how it becomes the output of D (y_D) by applying to it the logistic function σ.

We will assume (as we did previously) that all the neurons have a logistic activation function. So we need to do a forward pass, a backpropagation, and a second forward pass to see the decrease in the error. Let us briefly comment on the network itself. Our network has three layers, with the input and hidden layers consisting of two neurons, and the output error which consists of one neuron. We have denoted the layers with capital letters, but we have skipped the letter E to avoid confusing it with the error function, so we have neurons named A, B, C, D and F. This is not usual.

[17] We must then use the gradient, not individual partial derivatives.

[18] This is a modified version of the example by Matt Mazur available at https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/.

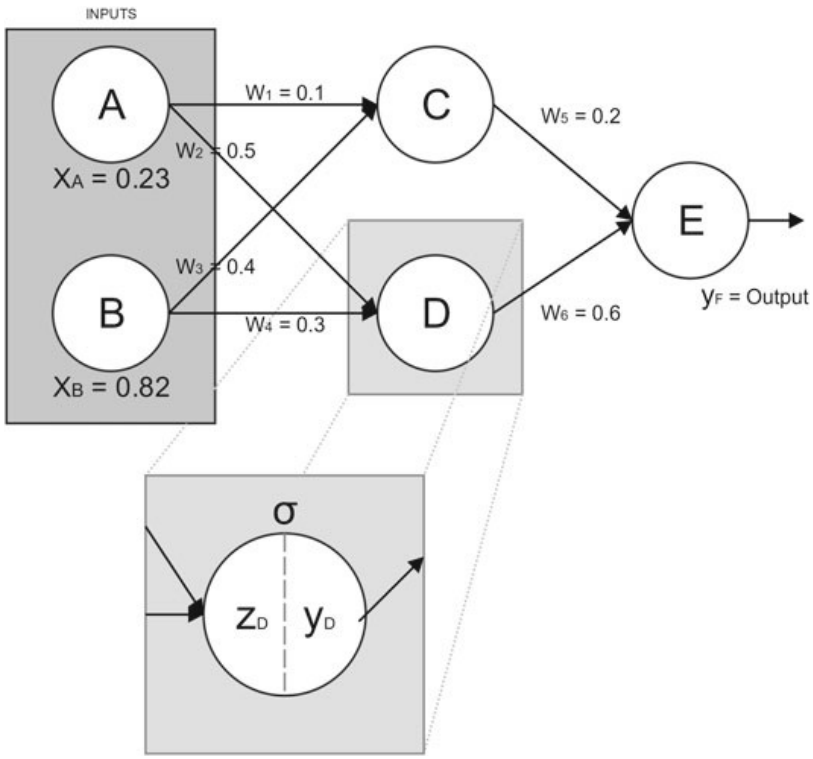

Fig. 4.5 Backpropagation in a complete simple neural network

The usual procedure is to name them by referring to the layer and neuron in the layer, e.g. 'third neuron in the first layer' or '1, 3'. The input layer takes in two inputs, the neuron A takes in $x_A = 0.23$ and the neuron B takes in $x_B = 0.82$. The target for this training case (consisting of x_A and x_B) will be 1. As we noted earlier, the hidden and output layers have the logistic activation function (also called *logistic nonlinearity*), which is defined as $\sigma(z) = \frac{1}{1+e^{-z}}$.

We start by computing the forward pass. The first step is to calculate the outputs of C and D, which are referred to as y_C and y_D, respectably:

$$y_C = \sigma(0.23 \cdot 0.1 + 0.82 \cdot 0.4) = \sigma(0.351) = 0.5868$$

$$y_D = \sigma(0.23 \cdot 0.5 + 0.82 \cdot 0.3) = \sigma(0.361) = 0.5892$$

And now we use y_C and y_D as inputs to the neuron F which will give us the final result:

$$y_F = \sigma(0.5868 \cdot 0.2 + 0.5892 \cdot 0.6) = \sigma(0.4708) = 0.6155$$

Now, we need to calculate the output error. Recall that we are using the mean squared error function, i.e. $E = \frac{1}{2}(t - y)^2$. So we plug in the target (1) and output (0.6155) and get:

$$E = \frac{1}{2}(t - y)^2 = \frac{1}{2}(1 - 0.6155)^2 = 0.0739$$

Now we are all set to calculate the derivatives. We will explain how to calculate w_5 and w_3 but all other weights are calculated with the same procedure. As backpropagation proceeds in the opposite direction that the forward pass, calculating w_5 is easier and we will do that first. We need to know how the change in w_5 affects E and we want to take those changes which minimize E. As noted earlier, the chain rule for derivatives will do most of the work for us. Let us rewrite what we need to calculate:

$$\frac{\partial E}{\partial w_5} = \frac{\partial E}{\partial y_F} \cdot \frac{\partial y_F}{\partial z_F} \cdot \frac{\partial z_F}{\partial w_5}$$

We have found the derivatives for all of these in the previous sections so we will not repeat their derivations. Note that we need to use partial derivatives because every derivation is made with respect to an indexed term. Also, note that the vector containing all partial derivatives (for all indices i) is the gradient. Let us address $\frac{\partial E}{\partial y_F}$ now. As we have seen earlier:

$$\frac{\partial E}{\partial y_F} = -(t - y_F)$$

In our case that means:

$$\frac{\partial E}{\partial y_F} = -(1 - 0.6155) = -0.3844$$

Now we address $\frac{\partial y_F}{\partial z_F}$. We know that this is equal to $y_F(1 - y_F)$. In our case this means:

$$\frac{\partial y_F}{\partial z_F} = y_F(1 - y_F) = 0.6155(1 - 0.6155) = 0.2365$$

The only thing left to calculate is $\frac{\partial z_F}{\partial w_5}$. Remember that:

$$z_F = y_C \cdot w_5 + y_D \cdot w_6$$

By using the rules of differentiation (derivatives of constants (w_6 is treated like a constant) and differentiating the differentiation variable) we get:

$$\frac{\partial z_F}{\partial w_5} = y_C \cdot 1 + y_D \cdot 0 = y_C = 0.5868$$

We take these values back to the chain rule and get:

$$\frac{\partial E}{\partial w_5} = \frac{\partial E}{\partial y_F} \cdot \frac{\partial y_F}{\partial z_F} \cdot \frac{\partial z_F}{\partial w_5} = -0.3844 \cdot 0.2365 \cdot 0.5868 = -0.0533$$

We repeat the same process[19] to get $\frac{\partial E}{\partial w_6} = -0.0535$. Now, all we have to do is use these values in the general weight update rule[20] (we use a learning rate, $\eta = 0.7$):

$$w_5^{new} = w_5^{old} - \eta \frac{\partial E}{\partial w_5} = 0.2 - (0.7 \cdot 0.0533) = 0.2373$$

$$w_6^{new} = 0.6374$$

Now we can continue to the next layer. But an important note first. We will be needing a value for w_5 and w_6 to find the derivatives of w_1, w_2, w_3 and w_4, and we will be using the old values, not the updated ones. We will update the whole network when we will have all the updated weights. We proceed to the hidden layer. What we need to now is to find the update for w_3. Notice that to get from the output neuron F to w_3 we need to go across C, so we will be using:

$$\frac{\partial E}{\partial w_3} = \frac{\partial E}{\partial y_C} \cdot \frac{\partial y_C}{\partial z_C} \cdot \frac{\partial z_C}{\partial w_3}$$

The process will be similar to $\frac{\partial E}{\partial w_3}$, but with a couple of modifications. We start with:

$$\frac{\partial E}{\partial y_C} = \frac{\partial z_F}{\partial y_C} \frac{\partial E}{\partial z_F} = w_5 \frac{\partial E}{\partial z_F} = w_5 \frac{\partial y_F}{\partial z_F} \cdot \frac{\partial E}{\partial y_F} = 0.2 \cdot 0.2365 \cdot (-0.3844) = 0.2 \cdot (-0.0909) = -0.0181$$

Now we need $\frac{\partial y_C}{\partial z_C}$:

$$\frac{\partial y_C}{\partial z_C} = y_C(1 - y_C) = 0.5868 \cdot (1 - 0.5868) = 0.2424$$

And we also need $\frac{\partial z_C}{\partial w_3}$. Recall that $z_C = x_1 \cdot w_1 + x_2 \cdot w_2$, and therefore:

$$\frac{\partial z_C}{\partial w_3} = x_1 \cdot 0 + x_2 \cdot 1 = x_2 = 0.82$$

Now we have:

$$\frac{\partial E}{\partial w_3} = \frac{\partial E}{\partial y_C} \cdot \frac{\partial y_C}{\partial z_C} \cdot \frac{\partial z_C}{\partial w_3} = -0.0181 \cdot 0.2424 \cdot 0.82 = -0.0035$$

[19] The only difference is the step for $\frac{\partial z_F}{\partial w_5}$, where there is a 0 now for w_5 and a 1 for w_6.

[20] Which we discussed earlier, but we will restate it here: $w_k^{new} = w_k^{old} - \eta \frac{\partial E}{\partial w_k}$.

Using the general weight update rule we have:

$$w_3^{new} = 0.4 - (0.7 \cdot (-0.0035)) = 0.4024$$

We use the same steps (across C) to find $w_1^{new} = 0.1007$. To get w_2^{new} and w_4^{new} we need to go across D. Therefore we need:

$$\frac{\partial E}{\partial w_3} = \frac{\partial E}{\partial y_D} \cdot \frac{\partial y_D}{\partial z_D} \cdot \frac{\partial z_D}{\partial w_3}$$

But we know the procedure, so:

$$\frac{\partial E}{\partial y_D} = w_6 \cdot \frac{\partial E}{\partial z_F} = 0.6 \cdot (-0.0909) = -0.0545$$

$$\frac{\partial y_C}{\partial z_C} = y_D(1 - y_D) = 0.5892(1 - 0.5892) = 0.2420$$

And:

$$\frac{\partial z_D}{\partial w_2} = 0.23$$

$$\frac{\partial z_D}{\partial w_4} = 0.82$$

Finally, we have (remember we have the 0.7 learning rate):

$$w_2^{new} = 0.5 - 0.7 \cdot (-0.0545 \cdot 0.2420 \cdot 0.23) = 0.502$$

$$w_4^{new} = 0.3 - 0.7 \cdot (-0.0545 \cdot 0.2420 \cdot 0.82) = 0.307$$

And we are done. To recap, we have:

- $w_1^{new} = 0.1007$
- $w_2^{new} = 0.502$
- $w_3^{new} = 0.4024$
- $w_4^{new} = 0.307$
- $w_5^{new} = 0.2373$
- $w_6^{new} = 0.6374$
- $E^{old} = 0.0739$

We can now make another forward pass with the new weights to make sure that the error has decreased:

$$y_C^{new} = \sigma(0.23 \cdot 0.1007 + 0.82 \cdot 0.4024) = \sigma(0.3531) = 0.5873$$

$$y_D^{new} = 0.5907$$

$$y_F^{new} = \sigma(0.5873 \cdot 0.2373 + 0.5907 \cdot 0.6374) = \sigma(0.5158) = 0.6261$$

$$E^{new} = \frac{1}{2}(1 - 0.6261)^2 = 0.0699$$

Which shows that the error has decreased. Note that we have processed only one training sample, i.e. the input vector (0.23, 0.82). It is possible to use multiple training samples to generate the error and find the gradients (mini-batch training[21]), and we can do this a number of times and each repetition is called an *iteration*. Iterations are sometimes erroneously called *epochs*. The two terms are very similar and we can consider them synonyms for now, but quite soon we will need to delineate the difference, and we will do this in the next chapter.

An alternative to this would be to update the weights after every single training example.[22] This is called *online learning*. In online learning, we process a single input vector (training sample) per iteration. We will discuss this in the next chapter in more detail.

In the remainder of this chapter, we will integrate all the ideas we have presented so far in a fully functional feedforward neural network, written in Python code. This example will be fully functional Python 3.x code, but we will write out some things that could be better left for a Python module to do.

Technically speaking, in anything but the most basic setting, we shall not use the SSE, but its variant, the *mean squared error* (MSE). This is because we need to be able to rewrite the cost function as the average of the cost functions SSE_x for individual training samples x, and we therefore define $MSE := \frac{1}{n}\sum_x SSE_x$.

4.7 A Complete Feedforward Neural Network

Let us see a complete feedforward neural network which does a simple classification. The scenario is that we have a webshop selling books and other stuff, and we want to know whether a customer will abandon a shopping basket at checkout. This is why we are making a neural network to predict it. For simplicity, all the data is just numbers. Open a new text file, rename it to `data.csv` and write in the following:

[21] Or full-batch if we use the whole training set.

[22] Which is equal to using a mini-batch of size 1.

```
includes_a_book,purchase_after_21,total,user_action
1,1,13.43,1
1,0,23.45,1
0,0,45.56,0
1,1,56.43,0
1,0,44.44,0
1,1,667.65,1
1,0,56.66,0
0,1,43.44,1
0,0,4.98,1
1,0,43.33,0
```

This will be our dataset. You can actually substitute this for anything, and as long as values are numbers, it will still work. The target is the `user_action` column, and we take 1 to mean that the purchase was successful, and 0 to mean that the user has abandoned the basket. Notice that we are talking about abandoning a shopping basket, but we could have put anything in, from images of dogs to bags of words. You should also make another CSV file named `new_data.csv` that has the same structure as `data.csv`, but without the last column (`user_action`). For example:

```
includes_a_book,purchase_after_21,total
1,0,73.75
0,0,64.97
1,0,3.78
```

Now let is continue to the Python code file. All the code in the remainder of this section should be placed in a single file, you can name it `ffnn.py`, and placed in the same folder as `data.csv` and `new_data.csv`. The first part of the code contains the import statements:

```
import pandas as pd
import numpy as np
from keras.models import Sequential
from keras.layers.core import Dense
TARGET_VARIABLE ="user_action"
TRAIN_TEST_SPLIT=0.5
HIDDEN_LAYER_SIZE=30
raw_data = pd.read_csv("data.csv")
```

The first four lines are just imports, the next three are hyperparameters. The `TARGET_VARIABLE` tells Python what is the target variable we wish to predict. The last line opens the file `data.csv`. Now we must make the train–test split. We have a hyperparameter that currently leaves 0.5 of the datapoints in the training set, but you can change this hyperparameter to something else. Just be careful since we have a tiny dataset which might cause some problems if the split is something like 0.95. The code for the train–test split is:

```
mask = np.random.rand(len(raw_data)) < TRAIN_TEST_SPLIT
tr_dataset = raw_data[mask]
te_dataset = raw_data[~mask]
```

The first line here defines a random sampling of the data to be used to get the train–test split and the next two lines select the appropriate sub-dataframes from the original Pandas dataframe (a dataframe is a table-like object, very similar to an Numpy array but Pandas focuses on easy reshaping and splitting, while Numpy focuses on fast computation). The next lines split both the train and test dataframes into labels and data, and then convert them into Numpy arrays, since Keras needs Numpy arrays to work. The process is relatively painless:

```
tr_data = np.array(raw_data.drop(TARGET_VARIABLE,
axis=1))
tr_labels = np.array(raw_data[[TARGET_VARIABLE]])
te_data = np.array(te_dataset.drop(TARGET_VARIABLE,
axis=1))
te_labels = np.array(te_dataset[[TARGET_VARIABLE]])
```

Now, we move to the Keras specification of a neural network model, and its compilation and training (fitting). We need to compile the model since we want Keras to fill in the nasty details and create arrays of appropriate dimensionality of the weight and bias matrices:

```
ffnn = Sequential()
ffnn.add(Dense(HIDDEN_LAYER_SIZE, input_shape=(3,),
activation="sigmoid"))
ffnn.add(Dense(1, activation="sigmoid"))
ffnn.compile(loss="mean_squared_error", optimizer=
"sgd", metrics=['accuracy'])
ffnn.fit(tr_data, tr_labels, epochs=150, batch_size=2,
verbose=1)
```

The first line initializes a new sequential model in a variable called `ffnn`. The second line specifies both the input layer (to accept 3D vectors as single data inputs), and the hidden layer size which is specified at the beginning of the file in the variable `HIDDEN_LAYER_SIZE`. The third line will take the hidden layer size from the previous layer (Keras does this automatically), and create an output layer with one neuron. All neurons will be having sigmoid or logistic activation functions. The fourth line specifies the error function (MSE), the optimizer (stochastic gradient descent) and which metrics to calculate. It also compiles the model, which means that it will assemble all the other stuff that Python needs from what we have specified. The last line trains the neural network on `tr_data`, using `tr_labels`, for 150 epochs, taking two samples in a mini-batch. `verbose=1` means that it will print the accuracy and loss after each epoch of training. Now we can continue to analyze the results on the test set:

```
metrics = ffnn.evaluate(te_data, te_labels, verbose=1)
print("%s: %.2f%%" % (ffnn.metrics_names[1],
metrics[1]*100))
```

The first line evaluates the model on `te_data` using `te_labels` and the second prints out accuracy as a formatted string. Next, we take in the `new_data.csv` file which simulates new data on our website and we try to predict what will happen using the `ffnn` trained model:

```
new_data = np.array(pd.read_csv("new_data.csv"))
results = ffnn.predict(new_data)
print(results)
```

References

1. M. Hassoun, *Fundamentals of Artificial Neural Networks* (MIT Press, Cambridge, 2003)
2. I.N. da Silva, D.H. Spatti, R.A. Flauzino, L.H.B. Liboni, S.F. dos Reis Alves, *Artificial Neural Networks: A Practical Course* (Springer, New York, 2017)
3. I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning* (MIT Press, Cambridge, 2016)
4. G. Montavon, G. Orr, K.R. Müller, *Neural Networks: Tricks of the Trade* (Springer, New York, 2012)
5. M. Minsky, S. Papert, *Perceptrons: An Introduction to Computational Geometry* (MIT Press, Cambridge, 1969)
6. P.J. Werbos, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences* (Harvard University, Cambridge, 1975)
7. D.B. Parker, Learning-logic. Technical Report-47 (MIT Center for Computational Research in Economics and Management Science, Cambridge, 1985)
8. Y. LeCun, Une procédure d'apprentissage pour réseau a seuil asymmetrique. Proc. Cogn. **85**, 599–604 (1985)
9. D.E. Rumelhart, G.E. Hinton, R.J. Williams, Learning internal representations by error propagation. Parallel Distrib. Process. **1**, 318–362 (1986)

Modifications and Extensions to a Feed-Forward Neural Network

5

5.1 The Idea of Regularization

Let us recall the ideas of variance and bias. If we have two classes (denoted by X and O) in a 2D space and the classifier draws a very straight line we have a classifier with a high bias. This line will generalize well, meaning that the classification error for the new points (test error) will be very similar to the classification error for the old points (training error). This is great, but the problem is that the error will be too large in the first place. This is called *underfitting*. On the other hand, if we have a classifier that draws an intricate line to include every X and none of the Os, then we have high variance (and low bias), which is called *overfitting*. In this case, we will have a relatively low training error a much larger testing error.

Let us take and abstract example. Imagine that we have the task of finding orcas among other animals. Then our classifier should be able to locate orcas by using the properties that are common to *all* orcas but not present in other animals. Notice that when we said 'all' we wanted to make sure we are identifying the species, not a subgroup of the specie: e.g. having a blue tag on the tail might be something that some orcas have, but we want to catch only those things that all orcas have and no other animal has. A 'species' in general is called a *type* (e.g. orcas), whereas an individual is called a *token* (e.g. the orca Shamu). We want to find a property that defines the type we are trying to classify. We call such a property a *necessary property*. In case of orcas this might be simply the property (or query):

$$\texttt{orca}(x) := \texttt{mammal}(x) \wedge \texttt{livesInOcean}(x) \wedge \texttt{blackAndWhite}(x)$$

But, sometimes it is not that easy to find such a property. Trying to find such a property is what a supervised machine learning algorithm does. So the problem might be rephrased as trying to find a complex property which defines a type as best as possible (by trying to include the biggest possible number of tokens and try

S. Skansi, *Introduction to Deep Learning*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-319-73004-2_5

to include only the relevant tokens in the definition). Therefore, overfitting can be understood in another way: our classifier is so good that we are not only capturing the necessary properties from our training examples, but also the non-necessary or *accidental properties*. So, we would like to capture all the properties which we need, but we want something to help us stop when we begin including the non-necessary properties.

Underfitting and overfitting are the two extremes. Empirically speaking, we can really go from high bias and low variance to high variance and low bias. Want to stop at a point in between, and we want this point to have better-than-average generalization capabilities (inherited from the higher bias), and a good fit to the data (inherited from high variance). How to find this 'sweet spot' is the art of machine learning, and the received wisdom in the machine learning community will insist it is best to find this by hand. But it is not impossible to automate, and deep learning, wanting to become a contender for artificial intelligence, will automate as much as possible. There is one approach which tries to automate our intuitions about overfitting, and this idea is called *regularization*.

Why are we talking about overfitting and not underfitting? Remember that if have a very high bias we will end up with a linear classifier, and linear classifiers cannot solve the XOR or similar simple problems. What we want then is to significantly lower the bias until we have reached the point after which we are overfitting. In the context of deep learning, after we have added a layer to logistic regression, we have said farewell to high bias and sailed away towards the shores of high variance. This sounds very nice, but how can we stop in time? How can we prevent overfitting. The idea of regularization is to add a regularization parameter to the error function E, so we will have

$$E^{improved} := E^{original} + RegularizationTerm$$

Before continuing to the formal definitions, let us see how we can develop a visual intuition on what regularization does (Fig. 5.1).

The left-hand side of the image depicts the classical various choices of hyperplanes we usually have (bias, variance, etc.). If we add a regularization term, the effect will be that the error function will not be able to pinpoint the datapoints *exactly*, and the

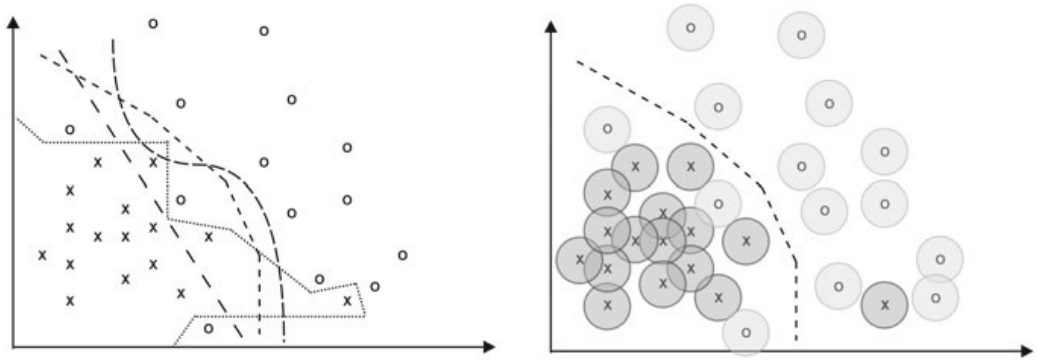

Fig. 5.1 Intuition about regularization

effect will be similar to the points becoming actually little circles. In this way, some of the choices for the hyperplane will simply become impossible, and the one that are left will be the ones that have a good "neutral zone" between Xs and Os. This is not the exact explanation of regularization (we will get to that shortly) but an intuition which is useful for informal reasoning about what regularization does and how it behaves.

5.2 L_1 and L_2 Regularization

As we have noted earlier, regularization means adding a term to the error function, so we have:

$$E^{improved} := E^{original} + RegularizationTerm$$

As one might guess, adding different regularization terms give rise to different regularization techniques. In this book, we will address the two most common types of regularization, L_1 and L_2 regularization. We will start with L_2 regularization and explore it in detail, since it is more useful in practice and it is also easier to grasp the connections with vector spaces and the intuition we developed in the previous section. Afterwards we will turn briefly to L_1 and later in this chapter we will address dropout which is a very useful technique unique to neural networks and has effects similar to regularization.

L_2 regularization is known under many names, 'weight decay', 'ridge regression', and 'Tikhonov regularization'. L_2 regularization was first formulated by the Soviet mathematician Andrey Tikhonov in 1943 [1], and was further refined in his paper [2]. The idea of L_2 regularization is to use the L_2 or *Euclidean norm* for the regularization term.

The L_2 norm of a vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ is simply $\sqrt{x_1^2 + x_2^2 + \ldots + x_n^2}$. The L_2 norm of the vector $\mathbf{x}$ can be denoted by $L_2(\mathbf{x})$ or, more commonly, by $||\mathbf{x}||_2$. The vector used is the weights of the final layer, but a version using all weights in the network can also be used (but in that case, our intuition will be off). So now we can rewrite the *preliminary* L_2-regularized error function as:

$$E^{improved} := E^{original} + ||\mathbf{w}||_2$$

But, in the machine learning community, we usually do not use the square root, so instead of $||\mathbf{w}||_2$ we will use the square of the L_2 norm, i.e. $(||\mathbf{w}||_2)^2 = ||\mathbf{w}||_2^2$ which is actually just $\sum_w w^2$. We will also want to add a hyperparameter to be able to adjust how much of the regularization we want to use (called the *regularization parameter* or *regularization rate*, and denoted by λ), and divide it by the size of the

batch used (to account for the fact that we want it to be proportional), so the final L_2-regularized error function is:

$$E^{improved} := E^{original} + \frac{\lambda}{n}||\mathbf{w}||_2^2 = E^{original} + \frac{\lambda}{n}\sum_{w_i in w_o} w_i^2$$

Let us work a bit on the explanation[1] what L_2 regularization does. The intuition is that during the learning procedure, smaller weights will be preferred, but larger weights will be considered if the overall decrease in error is significant. This explains why it is called 'weight decay'. The choice of λ determines how much will small weights be preferred (when λ is large, the preference for small weights will be great). Let us work through a simple derivation. We start with our regularized error function:

$$E^{new} = E^{old} + \frac{\lambda}{n}\sum_{w} w^2$$

By taking the partial derivatives of this equation we get:

$$\frac{\partial E^{new}}{\partial w} = \frac{\partial E^{old}}{\partial w} + \frac{\lambda}{n}w$$

Taking this back to the general weight update rule we get:

$$w^{new} = w^{old} - \eta \cdot (\frac{\partial E^{old}}{\partial w} + \frac{\lambda}{n}w)$$

One might wonder whether this would actually make the weights converge to 0, but this is not the case, since the first component $\frac{\partial E^{old}}{\partial w}$ will increase the weights if the reduction in error (this part controls the unregularized error) is significant.

We can now proceed to briefly sketch L_1 regularization. L_1 regularization, also known as 'lasso' or 'basis pursuit denoising' was first proposed by Robert Tibshirani in 1996 [4]. L_1 regularization uses the absolute value instead of the squares:

$$E^{improved} := E^{original} + \frac{\lambda}{n}||\mathbf{w}||_1 = E^{original} + \frac{\lambda}{n}\sum_{w_i in w_o} |w_i|$$

Let us compare the two regularizations to expose their peculiarities. For most classification and prediction problems, L_2 is better. However, there are certain tasks where L_1 excels [5]. The problems where L_1 is superior are those that contain a lot of irrelevant data. This might be either very noisy data, or features that are not informative, but it can also be sparse data (where most features are irrelevant because

[1] We will be using a modification of the explanation offered by [3]. Note that this book is available online at http://neuralnetworksanddeeplearning.com.

they are missing). This means that there are a number of useful applications of L_1 regularization in signal processing (e.g. [6]) and robotics (e.g. [7]).

Let us try to develop an intuition behind the two regularizations. The L_2 regularization tries to push down the square of the weights (which does not increase linearly as the weights increase), whereas L_1 is concerned with absolute values which is linear, and therefore L_2 will quickly penalize large weights (it tends to concentrate on them). L_1 regularization will make much more weights slightly smaller, which usually results in many weights coming close to 0. To simplify the matter completely, take the plots of the graphs $f(x) = x^2$ and $g(x) = |x|$. Imagine that those plots are physical surfaces like bowls. Now imagine putting some points in the graphs (which correspond to the weights) and adding 'gravity', so that they behave like physical objects (tiny marbles). The 'gravity' corresponds to gradient descent, since it is a move towards the minimum (just like gravity would push to a minimum in a physical system). Imagine that there is also friction, which corresponds to the idea that E does not care anymore about the weights that are already very close to the minimum. In the case of $f(x)$, we will have a number of points around the point $(0, 0)$, but a bit dispersed, whereas in $g(x)$ they would be very tightly packed around the $(0, 0)$ point. We should also note that two vectors can have the same L_1 norm but different L_2 norms. Take $\mathbf{v}_1 = (0.5, 0.5)$ and $\mathbf{v}_2 = (-1, 0)$. Then $||\mathbf{v}_1||_1 = |0.5| + |0.5| = 1$ and $||\mathbf{v}_2||_1 = |-1| + |0| = 1$, but $||\mathbf{v}_1||_2 = \sqrt{0.5^2 + 0.5^2} = \frac{1}{\sqrt{2}}$ and $||\mathbf{v}_2||_2 = \sqrt{1^2 + 0^2} = 1$.

5.3 Learning Rate, Momentum and Dropout

In this section, we will revisit the idea of the learning rate. The learning rate is an example of a *hyperparameter*. The name is quite unusual, but there is actually a simple reason behind it. Every neural network is actually a function which assigns to a given input vector (input) a class label (output). The way the neural network does this is via the operations it performs and the parameters it is given. Operations include the logistic function, matrix multiplication, etc., while the parameters are all numbers which are not inputs, viz. weights and biases. We know that the biases are simply weights and that the neural network finds a good set of weights by backpropagationg the errors it registers. Since operations are always the same, this means that all of the learning done by a neural network is actually a search for a good set of weight, or in other words, it is simply adjusting its parameters. There is nothing more to it, no magic, just weight adjusting. Now that this is clear, it is easy to say what a hyperparameter is. A hyperparameter is any number used in the neural network which cannot be learned by the network. An example would be the learning rate or the number of neurons in the hidden layer.

This means that learning cannot adjust hyperparameters, and they have to be adjusted manually. Here machine learning leans heavily towards art, since there is no scientific way to do it, it is more a matter of intuition and experience. But despite the fact that finding a good set of hyperparameters is not easy, there is a standard

procedure how to do it. To do this, we must revisit the idea of splitting the data set in a training set and a testing set. Suppose we have kept 10% of the datapoints for testing, and the rest we wanted to use as the training set. Now we will take another 10% of datapoints from the training set and call it a *validation set*. So we will have 80% of the datapoints in the training set for training, 10% we use for a validation set, and 10% we keep for a test set. The idea is to train on the train set with a given set of hyperparameters and test it on the validation set. If we are not happy, we re-train the network and test the validation set again. We do this until we get a good classification. Then, and only then we test on the test set to see how it is doing.

Remember that a low train error and a high test error is a sign of overfitting. When we are just training and testing (with no hyperparameter tuning), this is a good rule to stick to. But if we are tuning hyperparameter, we might get overfitting to both the training and validation set, since we are changing the hyperparameters *until* we get a small error on the validation set. If the errors can become misleadingly small since the classifier learns the noise of the training set, and we manually change the hyperparameters to suit the noise of the validation set. If, after this, there is proportionately small error on the test set, we have a winner, otherwise it is back to the drawing board. Of course, it is possible to alter the sizes of the train, validation and test sets, but these are the standard starting values (80%, 10% and 10% respectively).

We return to the learning rate. The idea of including a learning rate was first explicitly proposed in [8]. As we have seen in the last chapter, the learning rate controls how much of the update we want, since the learning rate is part of the general weight update rule, i.e. it comes into play in the very end of backpropagation. Before turning to the types of the learning rate, let us explore why the learning rate is important in an abstract setting.[2] We will construct an abstract model of learning by generalizing the idea with the parabola we proposed in the previous section. We need to expand this to three dimensions just so we can have more than one way to move. The overall shape of the 3D surface we will be using is like a bowl (Fig. 5.2).

Its lateral view is given by the axes x and y (we do not see z). Seen from the top (axes x and z visible, axis y not visible), it looks like a circle or ellipse. When we 'drop' a point at (x_k, z_k), it will get the value y_k from the curve at the coordinates (x_k, z_k). In other words, it will be as if we drop the point and it falls towards the bowl and stops as soon as it meets the surface of the bowl (imagine that our point is a sticky object, like a chewing gum). We drop it at a precise (x_k, z_k) (this is the 'top view'), we do not know the final 'height' of the sticky object, but we will measure it when it falls to the side of the bowl.

The gradient is like gravity, and it tries to minimize y. If we want to continue our analogy, we must make a couple of changes to the physical world: (i) we will not have sticky objects all the time (we needed them to explain how can we get the y of a point if we only have (x, z)), but little marbles which turn to sticky objects when they have finished their move (or you may think that they 'freeze'), (b) there is no

[2] We take the idea for this abstraction from Geoffrey Hinton's courses.

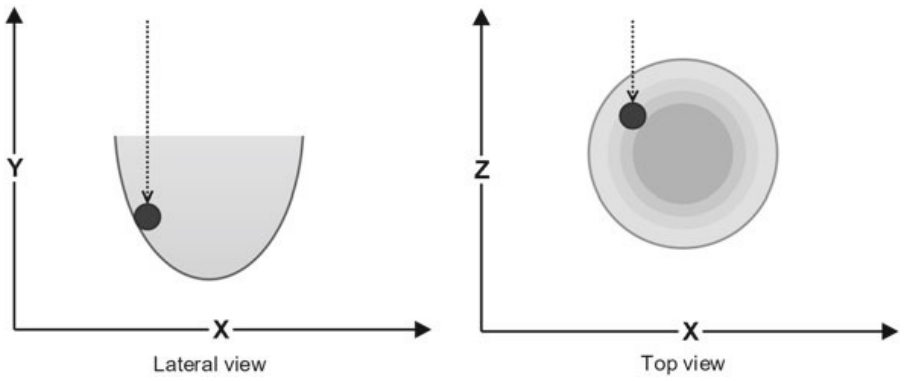

Fig. 5.2 Gradient bowl

friction or inertia and, perhaps the most counterintuitive, (c) our gravity is similar to physical gravity but different.

Let us explain (c) in more detail. Suppose we are looking from the top, so we see only axes x and z and we drop a marble. We want our gravity to behave like physical gravity in the sense that it will automatically generate the direction the marble has to move (looking from the top, the x and z view) so that it moves along the curvature of the bowl which is, hopefully, the direction of the bottom of the bowl (the global minimum value for y).

We want it to be different to physical gravity so that the amount of movement in this direction is not determined by the exact position of the minimum for y, i.e. it does not settle in the bottom but may move on the other side of the bowl (and remains there as if it became a sticky object again). We leave the amount of movement unspecified at the moment, but assume it is rarely the *exact* amount needed to reach the actual minimum: sometimes it is a bit more and it overshoots and sometimes is a bit less and it fails to reach it. One very important point has to be made here: the curvature 'points' at the minimum, but we are following the curvature at the point we currently are, and not the minimum. In a sense, the marble is extremely 'short-sighted' (marbles usually are): it sees only the current curvature and moves along it. We will know we have found the minimum when we have the curvature of 0. Note that in our example we have an 'idealized bowl', which has only one point where the curvature is 0, and that is the global minimum for y. Imagine how many more complex surfaces there might be where we cannot say that the point of curvature 0 is the global minimum, but also note that if we could have a transformation which transforms any of these complex surfaces into our bowl we would have a perfect learning algorithm.

Also, we want to add a bit of imprecision, so imagine that the direction of our gravity is the 'general direction' of the curvature of the bowl—sometimes a bit to the left, sometimes a bit to the right of the minimum, but only on rare occasions follows precisely the curvature.

Now we have the perfect setting for explaining learning in the abstract sense. Each epoch of learning is one move (of some amount) in the 'general direction' of the curvature of the bowl, and after it is done, it sticks where it is. The second epoch

'unfreezes' the situation, and again the general direction towards of the curvature is followed. this second move might either be the continuation of the first, or a move in an almost opposite direction if the marble overshot the minimum (bottom). The process can continue indefinitely, but after a number of epochs the moves will be really small and insignificant, so we can either stop after a predetermined number of epochs or when the improvement is not significant.[3]

Now let us return to the learning rate. The learning rate controls how much of the amount of movement we are going to take. A learning rate of 1 means to make the whole move, and a learning rate of 0.1 means to make only 10% of the move. As mentioned earlier, we can have a global learning rate or parametrized learning rate so that it changes according to certain conditions we specify such as the number of epochs so far, or some other parameter.

Let us return a bit to our bowl. So far we had a round bowl, but imagine we have a shallow bowl of the shape of an elongated ellipse (Fig. 5.3). If we drop the marble near the narrow middle, we will have almost the same situation as before. But if we drop it on the marble at the top left portion, it will move along a very shallow curvature and it will take a very large number of epochs to find its way towards the bottom of the bowl. The learning rate can help here. If we take only a fraction of the move, the direction of the curvature for the next move will be considerably better than if we move from one edge of a shallow and elongated bowl to the opposing edge. It will make smaller steps but it will find a good direction much more quickly.

This leaves us with discussing the typical values for the learning rate η. The values most often used are 0.1, 0.01, 0.001, and so on. Values like 0.03 will simply get lost and behave very similarly to the closest logarithm, which is 0.01 in case of 0.03.[4] The learning rate is a hyperparameter, and like all hyperparameters it has to be tuned on the validation set. So, our suggestion is to try with some of the standard values for a given hyperparameter and then see how it behaves and modify it accordingly.

We turn our attention now to an idea similar to the learning rate, but different called *momentum*, also called *inertia*. Informally speaking, the learning rate controls

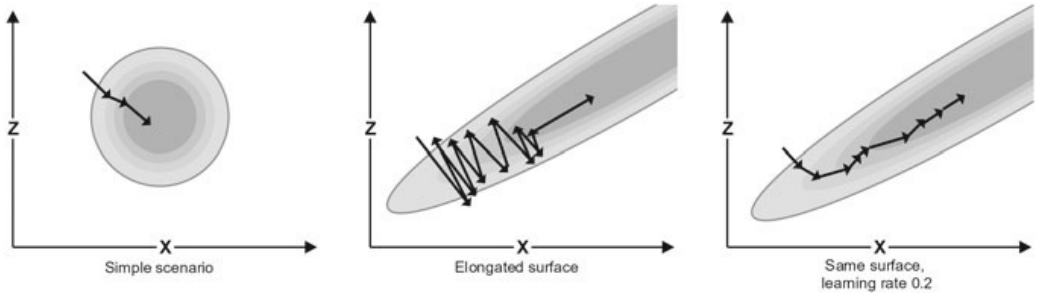

Fig. 5.3 Learning rate

[3] This is actually also a technique which is used to prevent overfitting called *early stopping*.

[4] You can use the learning rate to force a gradient explosion, so if you want to see gradient explosion for yourself try with an η value of 5 or 10.

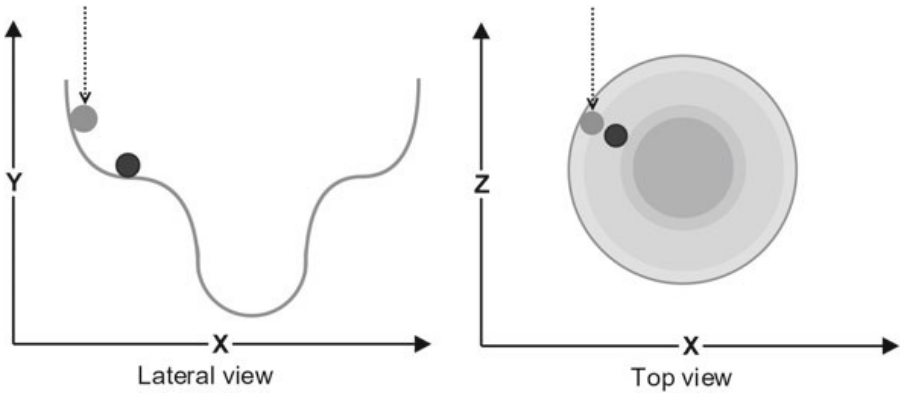

Fig. 5.4 Local minimum

how much of the move to keep in the present step, while momentum controls how much of the move from the previous step to keep in the current step. The problem which momentum tries to solve is the problem of *local minima*. Let us return to our idea with the bowl but now let us modify the bowl to have local minima. You can see the lateral view in Fig. 5.4. Notice that the learning rate was concerned with the 'top' view whereas the momentum addresses problems with the 'lateral' view.

The marble falls down as usual (depicted as grey in the image) and continues along the curvature, and stops when the curvature is 0 (depicted by black in the image). But the problem is that the curvature 0 is not necessarily the global minimum, it is only local. If it were a physical system, the marble would have momentum and it would fall over the local minimum to a global minimum, there it would go back and forth a bit and then it would settle. Momentum in neural networks is just the formalization of this idea. Momentum, like the learning rate is added to the general weight update rule:

$$w_i^{new} = w_i^{old} - \eta \frac{\partial E}{\partial w_i^{old}} + \mu(|w_i^{old} - w_i^{older}|)$$

Where w_i^{new} is the current weight to be computed, w_i^{old} is the previous value of the weight and w_i^{older} was the value of the weight before that. μ is the *momentum rate* and ranges from 0 to 1. It directly controls how much of the previous change in weight we will keep in this iteration. A typical value for μ is 0.9, and should be adjusted usually to a value between 0.10 and 0.99. Momentum is as old as the last discovery of backpropagation, and it was first published in the same paper by Rumelhart, Hinton and Williams [9].

There is one final interesting technique for improving the way neural networks learn and reduce overfitting, named *dropout*. We have chosen to define regularization as adding a regularization term to the cost function, and according to this definition dropout is not regularization, but it does lower the gap between the training error and the testing error, and consequently it reduces overfitting. One could define regularization to be any technique that reduces this spread, and then dropout would be a regularization technique. One could call dropout a 'structural regularization' and

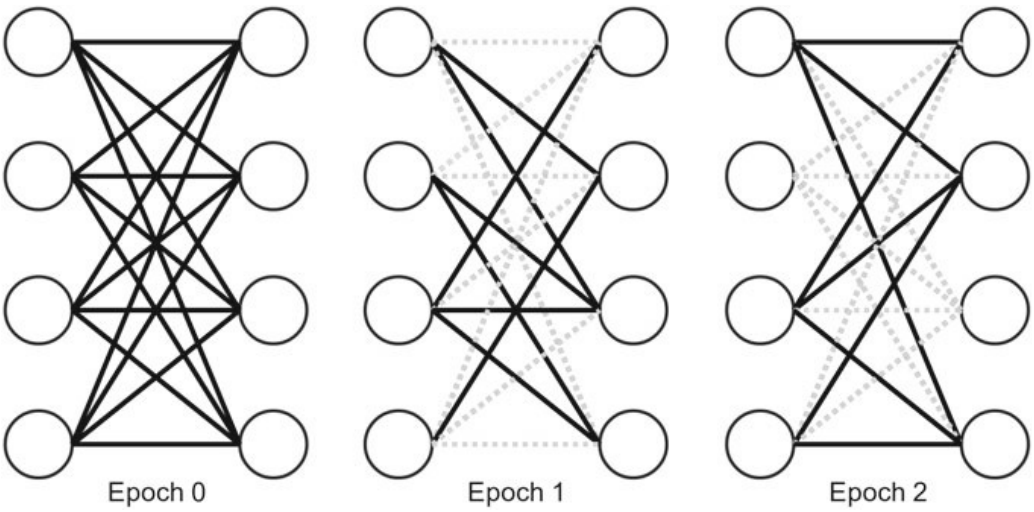

Fig. 5.5 Dropout with $\pi = 0.5$

the L_1 and L_2 regularizations 'numerical regularizations', but this is not standard terminology and we will not be using it.

Dropout was first explained in [10], but one could find more details about it in [11] and especially [12]. Dropout is a surprisingly simple technique. We add a dropout parameter π ranging from 0 to 1 (to be interpreted as a probability), and in each epoch every weight is set to zero with a probability of π (Fig. 5.5). Returning to the general weight update rule (where we need a w_k^{old} for calculating the weight updates), if in epoch n the weight w_k was set to zero, the w_k^{old} for epoch $n+1$ will be the w_k from epoch $n-1$. Dropout forces the network to learn redundancies so it is better in isolating the necessary properties of the dataset. A typical value for π is 0.2, but like all other hyperparameters it has to be tuned on the validation set.

5.4 Stochastic Gradient Descent and Online Learning

So far in this book, we have been a bit clumsy with one important question[5]: how does backpropagation work from a 'bird's-eye view'. We have been avoiding this question to avoid confusion until we had enough conceptual understanding to address it, and now we know enough to state it clearly. Backpropagation in the neural network works in the following way: we take one training sample at a time and pass it through the network and record the squared error for each. Then we use it to calculate the mean (squared) error. Once we have the mean squared error, we backpropagate it using gradient descent to find a better set of weights. Once we are done, we have

[5] We have been clumsy around several things, and this section is intended to redefine them a bit to make them more precise.

finished one epoch of training. We may do this for as many epochs we want. Usually, we want to continue either for a fixed number of epochs or stop it if it does not help with decreasing the error anymore.

What we have used when explaining backpropagation was a training set of size 1 (a single example). If this is the whole training set (a weirdly small training set), this would be an example of (full) gradient descent (also called *full-batch learning*). We could however think of it as being a subset of the training set. When using a randomly selected subset of from the training set of the size n, we say we use *stochastic gradient descent* or *minibatch learning* (with batch size n). Learning with a minibatch of size 1 is called *online learning*. Online learning can be either 'stationary' with fixed training set and then selecting randomly[6] one by one, or simply giving new training samples as they come along.[7] So we could think of our example backpropagation from the last chapter as an instance of online learning.

Now we are also in position to introduce a terminological finesse we have been neglecting until now. An *epoch* is one complete forward and backward pass over the *whole* training set. If we divide the training set of the size 10000 in 10 minibatches,[8] then one forward and one backward pass over a batch is called one *iteration*, and ten iterations (the size of the minibatch) is one *epoch*. This will hold only if the samples are divided as we stated in the footnote. If we use a random selection of training samples for the minibatch, then ten iterations will not make one epoch. If, on the other hand, we shuffle the training set and then divide it, then ten iterations will make one epoch and the forces fighting for order in the universe will be triumphant once more.

Stochastic gradient descent is usually much quicker to converge, since by random sampling we can get a good estimate of the overall gradient, but if the minimum is not pronounced (the bowl is too shallow) it tends to compound the problems we have seen previously in Fig. 5.3 (the middle part) in the previous section. The intuitive reason behind it is that when we have a shallow curvature and sample the surface randomly we will be prone to losing the little amount of information about the curvature that we had in the beginning. In such cases, full gradient descent couple with momentum might be a good choice.

[6] We could use also a non-random selection. One of the most interesting ideas here is that of learning the simplest instances first and then proceeding to the more tricky ones, and this approach is called *curriculum learning*. For more on this see [13].

[7] This is similar to *reinforcement learning*, which is, along with supervised and unsupervised learning one of the three main areas of machine learning, but we have decided against including it in this volume, since it falls outside of the the idea of a *first* introduction to deep learning. If the reader wishes to learn more, we refer her to [14].

[8] Suppose for the sake of clarification it is non-randomly divided: the first batch contains training samples 1 to 1000, the second 1001 to 2000, etc.

5.5 Problems for Multiple Hidden Layers: Vanishing and Exploding Gradients

Let us return to the calculation of the fully functional feed-forward neural network from the last chapter. Remember it was a neural network with the configuration (2, 2, 1), meaning it has two input neurons, two hidden neurons[9] and one output neuron. Let us revisit the weight updates we calculated:

- $w_1^{old} = 0.1$, $w_1^{new} = 0.1007$
- $w_2^{old} = 0.5$, $w_2^{new} = 0.502$
- $w_3^{old} = 0.4$, $w_3^{new} = 0.4024$
- $w_4^{old} = 0.3$, $w_4^{new} = 0.307$
- $w_5^{old} = 0.2$, $w_5^{new} = 0.2373$
- $w_6^{old} = 0.6$, $w_6^{new} = 0.6374$

Just by looking at the amount of the weight update you might notice that two weights have been updated with a significantly larger amount than the other weights. These two weights (w_5 and w_6) are the weights connecting the output layer with the hidden layer. The rest of the weights connect the input layer with the hidden layer. But why are they larger? The reason is that we had to backpropagate through few layers, and they remained larger: backpropagation is, structurally speaking, just the chain rule. The chain rule is just multiplication of derivatives. And, derivatives of everything we needed[10] have values between 0 and 1. So, by adding layers through which we had to backpropagate, we needed to multiply more and more 0 to 1 numbers, and this generally tends to become very small very quickly. And this is without regularization, with regularization it would be even worse, since it would prefer small weights at all times (since the weight updates would be small because of the derivatives, there would be little chance of the unregularized part to increase the weights). This phenomena is called *vanishing gradient*.

We could try to circumvent this problem by initializing the weights to a very large value and hope that backpropagation will just chip them to the correct value.[11] In this case, we might get a very large gradient which would also hinder learning since a step in the direction of the gradient would be the right direction but the magnitude of the step would take us farther away from the solution than we were before the step. The moral of the story is that usually the problem is the vanishing gradient, but

[9] A single hidden layer with two neurons in it. It it was (3, 2, 4, 1) we would know it has two hidden layer, the first one with two neurons and the second one with four.

[10] Ok, we have used the adjusted the values to make this statement true. Several of the derivatives we need will become a value between 0 and 1 soon, but it the sigmoid derivatives are mathematically bound between 0 and 1, and if we have many layers (e.g. 8), the sigmoid derivatives would dominate backpropagation.

[11] If the regular approach was something like making a clay statue (removing clay, but sometimes adding), the intuition behind initializing the weights to large values would be taking a block of stone or wood and start chipping away pieces.

if we change radically our approach we would be blown in the opposite direction which is even worse. Gradient descent, as a method, is simply too unstable if we use many layers through which we need to backpropagate.

To put the importance of the the vanishing gradient problem, we must note that the vanishing gradient is *the* problem to which deep learning is the solution. What truly defines deep learning are the techniques which make possible to stack many layers and yet avoid the vanishing gradient problem. Some deep learning techniques deal with the problem head on (LSTM), while some are trying to circumvent it (convolutional neural networks), some are using different connections than simple neural networks (Hopfield networks), some are hacking the solution (residual connections), while some have been using weird neural network phenomena to gain the upper hand (autoencoders). The rest of this book is devoted to these techniques and architectures. Historically speaking, the vanishing gradient was first identified by Sepp Hochreiter in 1991 in his diploma thesis [15]. His thesis advisor was Jürgen Schmidhuber, and the two will develop one of the most influential recurrent neural network architectures (LSTM) in 1997 [16], which we will explore in detail in the following chapters. An interesting paper by the same authors which brings more detail to the discussion of the vanishing gradient is [17].

We make a final remark before continuing to the second part of this book. We have chosen what we believe to be the most popular and influential neural architectures, but there are many more and many more will be discovered. The aim of this book is not to provide a comprehensive view of everything there is or will be, but to help the reader acquire the knowledge and intuition needed to pursue research-level deep learning papers and monographs. This is not a final tome about deep learning, but a first introduction which is necessarily incomplete. We made a serious effort to include a range of neural architectures which will demonstrate to the reader the vast richness and fulfilling diversity of this amazing field of cognitive science and artificial intelligence.

References

1. A.N. Tikhonov, On the stability of inverse problems. Dokl. Akad. Nauk SSSR **39**(5), 195–198 (1943)
2. A.N. Tikhonov, Solution of incorrectly formulated problems and the regularization method. Sov. Math. **4**, 1035–1038 (1963)
3. M.A. Nielsen, *Neural Networks and Deep Learning* (Determination Press, 2015)
4. R. Tibshirani, Regression shrinkage and selection via the lasso. J. Roy. Stat. Soc. Ser B (Methodol.) **58**(1), 267–288 (1996)
5. A. Ng, Feature selection, L1 versus L2 regularization, and rotational invariance, in *Proceedings of the International Conference on Machine Learning* (2004)
6. D.L. Donoho, Compressed sensing. IEEE Trans. Inf. Theory **52**(4), 1289–1306 (2006)
7. E.J. Candes, J. Romberg, T. Tao, Robust uncertainty principles: exact signal reconstruction from highly incomplete frequency information. IEEE Trans. Inf. Theory **52**(2), 489–509 (2006)

8. J. Wen, J.L. Zhao, S.W. Luo, Z. Han, The improvements of BP neural network learning algorithm, in *Proceedings of 5th International Conference on Signal Processing* (IEEE Press, 2000), pp. 1647–1649
9. D.E. Rumelhart, G.E. Hinton, R.J. Williams, Learning internal representations by error propagation. Parallel Distrib. Process. **1**, 318–362 (1986)
10. G.E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, Improving neural networks by preventing co-adaptation of feature detectors (2012)
11. G.E. Dahl, T.N. Sainath, G.E. Hinton, Improving deep neural networks for LVCSR using rectified linear units and dropout, in *IEEE International Conference on Acoustic Speech and Signal Processing* (IEEE Press, 2013), pp. 8609–8613
12. N. Srivastava, G.E. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, Dropout: a simple way to prevent neural networks from overfitting. J. Mach. Learn. Res. **15**, 1929–1958 (2014)
13. Y. Bengio, J. Louradour, R. Collobert, J. Weston, Curriculum learning, in *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, New York, NY, USA*, (ACM, 2009), pp. 41–48
14. R.S. Sutton, A.G. Barto, *Reinforcement Learning: An Introduction* (MIT Press, Cambridge, 1998)
15. S. Hochreiter, Untersuchungen zu dynamischen neuronalen Netzen, Diploma thesis, Technische Universität Munich, 1991
16. S. Hochreiter, J. Schmidhuber, Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)
17. S. Hochreiter, Y. Bengio, P. Frasconi, J. Schmidhuber, Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, in *A Field Guide to Dynamical Recurrent Neural Networks*, ed. by S.C. Kremer, J.F. Kolen (IEEE Press, 2001)