5 Modifications and Extensions to a Feed-Forward Neural Network

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):

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

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

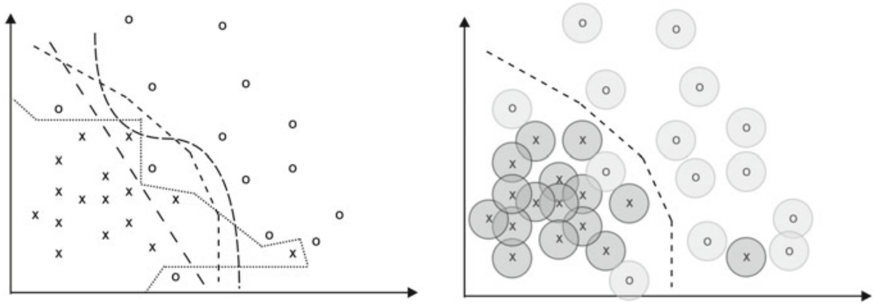

Fig. 5.1 Intuition about regularization

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.

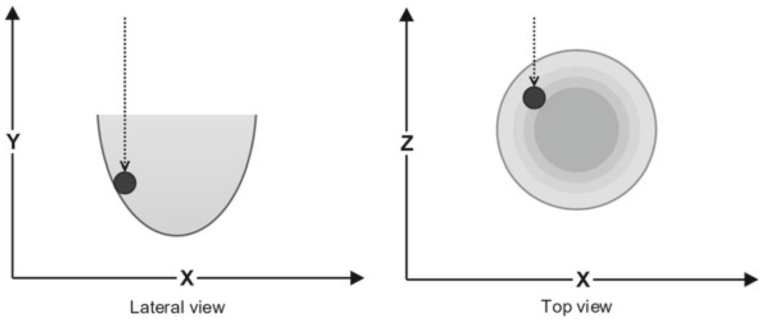

Fig. 5.2 Gradient bowl

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

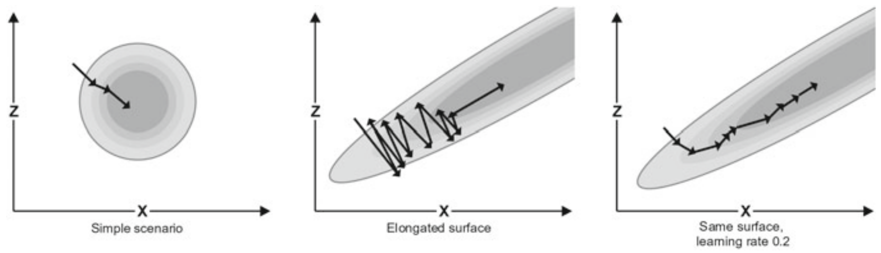

Fig. 5.3 Learning rate

[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.

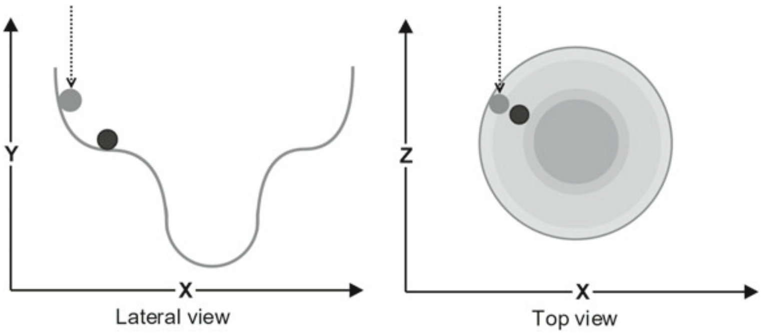

Fig. 5.4 Local minimum

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

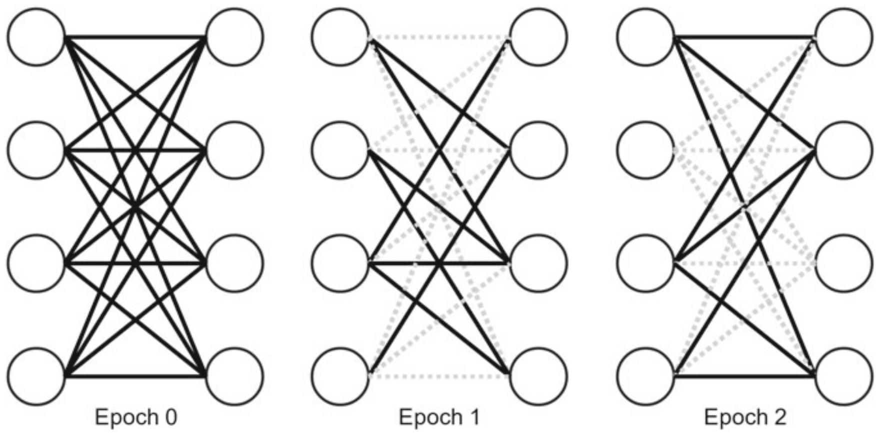

Fig. 5.5 Dropout with $\pi = 0.5$

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)