**CHAPTER 11**

**Training Deep Neural Networks**



With Early Release ebooks, you get books in their earliest form— the author’s raw and unedited content as he or she writes—so you can take advantage of these technologies long before the official release of these titles. The following will be Chapter 11 in the final release of the book.



In Chapter 10 we introduced artificial neural networks and trained our first deep neural networks. But they were very shallow nets, with just a few hidden layers. What if you need to tackle a very complex problem, such as detecting hundreds of types of objects in high-resolution images? You may need to train a much deeper DNN, per‐ haps with 10 layers or much more, each containing hundreds of neurons, connected by hundreds of thousands of connections. This would not be a walk in the park:

* First, you would be faced with the tricky *vanishing gradients* problem (or the related *exploding gradients* problem) that affects deep neural networks and makes lower layers very hard to train.
* Second, you might not have enough training data for such a large network, or it might be too costly to label.
* Third, training may be extremely slow.
* Fourth, a model with millions of parameters would severely risk overfitting the training set, especially if there are not enough training instances, or they are too noisy.

In this chapter, we will go through each of these problems in turn and present techni‐ ques to solve them. We will start by explaining the vanishing gradients problem and exploring some of the most popular solutions to this problem. Next, we will look at transfer learning and unsupervised pretraining, which can help you tackle complex



**325**

tasks even when you have little labeled data. Then we will discuss various optimizers that can speed up training large models tremendously compared to plain Gradient Descent. Finally, we will go through a few popular regularization techniques for large neural networks.

With these tools, you will be able to train very deep nets: welcome to Deep Learning!

**Vanishing/Exploding Gradients Problems**

As we discussed in Chapter 10, the backpropagation algorithm works by going from the output layer to the input layer, propagating the error gradient on the way. Once the algorithm has computed the gradient of the cost function with regards to each parameter in the network, it uses these gradients to update each parameter with a Gradient Descent step.

Unfortunately, gradients often get smaller and smaller as the algorithm progresses down to the lower layers. As a result, the Gradient Descent update leaves the lower layer connection weights virtually unchanged, and training never converges to a good solution. This is called the *vanishing gradients* problem. In some cases, the opposite can happen: the gradients can grow bigger and bigger, so many layers get insanely large weight updates and the algorithm diverges. This is the *exploding gradients* prob‐ lem, which is mostly encountered in recurrent neural networks (see ???). More gener‐ ally, deep neural networks suffer from unstable gradients; different layers may learn at widely different speeds.

Although this unfortunate behavior has been empirically observed for quite a while (it was one of the reasons why deep neural networks were mostly abandoned for a long time), it is only around 2010 that significant progress was made in understand‐ ing it. A paper titled [“Understanding the Difficulty of Training Deep Feedforward](https://homl.info/47) [Neural Networks”](https://homl.info/47) by Xavier Glorot and Yoshua Bengio[1](#page2) found a few suspects, includ‐ ing the combination of the popular logistic sigmoid activation function and the weight initialization technique that was most popular at the time, namely random ini‐ tialization using a normal distribution with a mean of 0 and a standard deviation of 1. In short, they showed that with this activation function and this initialization scheme, the variance of the outputs of each layer is much greater than the variance of its inputs. Going forward in the network, the variance keeps increasing after each layer until the activation function saturates at the top layers. This is actually made worse by the fact that the logistic function has a mean of 0.5, not 0 (the hyperbolic tangent function has a mean of 0 and behaves slightly better than the logistic function in deep networks).

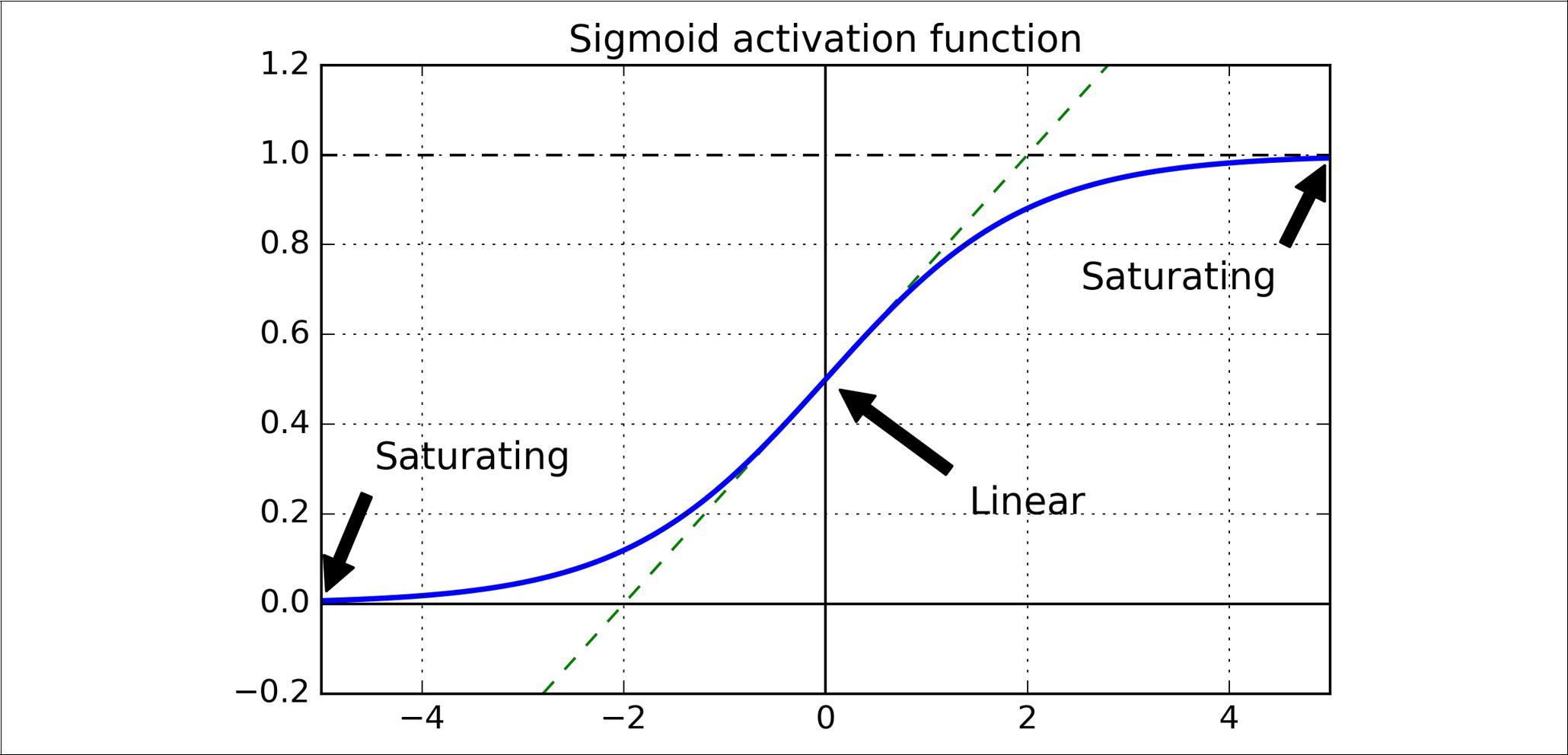


1 “Understanding the Difficulty of Training Deep Feedforward Neural Networks,” X. Glorot, Y Bengio (2010).



**326** **|** **Chapter 11: Training Deep Neural Networks**

Looking at the logistic activation function (see [Figure 11-1](#page3)), you can see that when inputs become large (negative or positive), the function saturates at 0 or 1, with a derivative extremely close to 0. Thus when backpropagation kicks in, it has virtually no gradient to propagate back through the network, and what little gradient exists keeps getting diluted as backpropagation progresses down through the top layers, so there is really nothing left for the lower layers.



*Figure 11-1. Logistic activation function saturation*

**Glorot and He Initialization**

In their paper, Glorot and Bengio propose a way to significantly alleviate this prob‐ lem. We need the signal to flow properly in both directions: in the forward direction when making predictions, and in the reverse direction when backpropagating gradi‐ ents. We don’t want the signal to die out, nor do we want it to explode and saturate. For the signal to flow properly, the authors argue that we need the variance of the outputs of each layer to be equal to the variance of its inputs,[2](#page3) and we also need the gradients to have equal variance before and after flowing through a layer in the reverse direction (please check out the paper if you are interested in the mathematical details). It is actually not possible to guarantee both unless the layer has an equal number of inputs and neurons (these numbers are called the *fan-in* and *fan-out* of the layer), but they proposed a good compromise that has proven to work very well in practice: the connection weights of each layer must be initialized randomly as



1. Here’s an analogy: if you set a microphone amplifier’s knob too close to zero, people won’t hear your voice, but if you set it too close to the max, your voice will be saturated and people won’t understand what you are say‐ ing. Now imagine a chain of such amplifiers: they all need to be set properly in order for your voice to come out loud and clear at the end of the chain. Your voice has to come out of each amplifier at the same amplitude as it came in.



**Vanishing/Exploding Gradients Problems** **|** **327**

described in [Equation 11-1](#page4), where *f an*avg = *f an*in + *f an*out /2. This initialization

strategy is called *Xavier initialization* (after the author’s first name) or *Glorot initiali‐* *zation* (after his last name).

*Equation 11-1. Glorot initialization (when using the logistic activation function)*

Normal distribution with mean 0 and variance *σ*2 = fan1 avg

|  |  |  |  |
| --- | --- | --- | --- |
| Or a uniform distribution between −*r* and + *r*, with *r* = | 3 | |  |
| fan | avg |  |
|  |  |  |



If you just replace *fan*avg with *fan*in in [Equation 11-1](#page4), you get an initialization strategy that was actually already proposed by Yann LeCun in the 1990s, called *LeCun initiali‐* *zation*, which was even recommended in the 1998 book *Neural Networks: Tricks of the Trade* by Genevieve Orr and Klaus-Robert Müller (Springer). It is equivalent toGlorot initialization when *fan*in = *fan*out. It took over a decade for researchers to realize just how important this trick really is. Using Glorot initialization can speed up train‐ ing considerably, and it is one of the tricks that led to the current success of Deep Learning.

Some [papers](https://homl.info/48)[3](#page4) have provided similar strategies for different activation functions. These strategies differ only by the scale of the variance and whether they use *fan*avg or

*fan*in, as shown in[Table 11-1](#page4)(forthe uniform distribution, just compute *r* = 3*σ*2).The initialization strategy for the ReLU activation function (and its variants, includ‐ ing the ELU activation described shortly) is sometimes called *He initialization* (after the last name of its author). The SELU activation function will be explained later in this chapter. It should be used with LeCun initialization (preferably with a normal distribution, as we will see).



*Table 11-1. Initialization parameters for each type of activation function*

|  |  |  |  |
| --- | --- | --- | --- |
| **Initialization** | **Activation functions** | ***σ*² (Normal)** | |
| Glorot | None, Tanh, Logistic, Softmax | 1 | / *fan*avg |
| He | ReLU & variants | 2 | / *fan*in |
| LeCun | SELU | 1 | / *fan*in |

By default, Keras uses Glorot initialization with a uniform distribution. You can change this to He initialization by setting kernel\_initializer="he\_uniform" or ker nel\_initializer="he\_normal" when creating a layer, like this:



1. Such as “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification,” K. He et al. (2015).



**328** **|** **Chapter 11: Training Deep Neural Networks**

keras.layers.Dense(10, activation="relu", kernel\_initializer="he\_normal")

If you want He initialization with a uniform distribution, but based on *fan*avg rather than *fan*in, you can use the VarianceScaling initializer like this:

he\_avg\_init = keras.initializers.VarianceScaling(scale=2., mode='fan\_avg', distribution='uniform')

keras.layers.Dense(10, activation="sigmoid", kernel\_initializer=he\_avg\_init)

**Nonsaturating Activation Functions**

One of the insights in the 2010 paper by Glorot and Bengio was that the vanishing/ exploding gradients problems were in part due to a poor choice of activation func‐ tion. Until then most people had assumed that if Mother Nature had chosen to use roughly sigmoid activation functions in biological neurons, they must be an excellent choice. But it turns out that other activation functions behave much better in deep neural networks, in particular the ReLU activation function, mostly because it does not saturate for positive values (and also because it is quite fast to compute).

Unfortunately, the ReLU activation function is not perfect. It suffers from a problem known as the *dying ReLUs*: during training, some neurons effectively die, meaning they stop outputting anything other than 0. In some cases, you may find that half of your network’s neurons are dead, especially if you used a large learning rate. A neu‐ ron dies when its weights get tweaked in such a way that the weighted sum of its inputs are negative for all instances in the training set. When this happens, it just keeps outputting 0s, and gradient descent does not affect it anymore since the gradi‐ ent of the ReLU function is 0 when its input is negative.[4](#page5)

To solve this problem, you may want to use a variant of the ReLU function, such as the *leaky ReLU*. This function is defined as LeakyReLU*α*(*z*) = max(*αz*, *z*) (see [Figure 11-2](#page6)). The hyperparameter *α* defines how much the function “leaks”: it is the slope of the function for *z* < 0, and is typically set to 0.01. This small slope ensures that leaky ReLUs never die; they can go into a long coma, but they have a chance to eventually wake up. A [2015 paper](https://homl.info/49)[5](#page5) compared several variants of the ReLU activation function and one of its conclusions was that the leaky variants always outperformed the strict ReLU activation function. In fact, setting *α* = 0.2 (huge leak) seemed to result in better performance than *α* = 0.01 (small leak). They also evaluated the *randomized leaky ReLU* (RReLU), where *α* is picked randomly in a given range duringtraining, and it is fixed to an average value during testing. It also performed fairly well and seemed to act as a regularizer (reducing the risk of overfitting the training set).

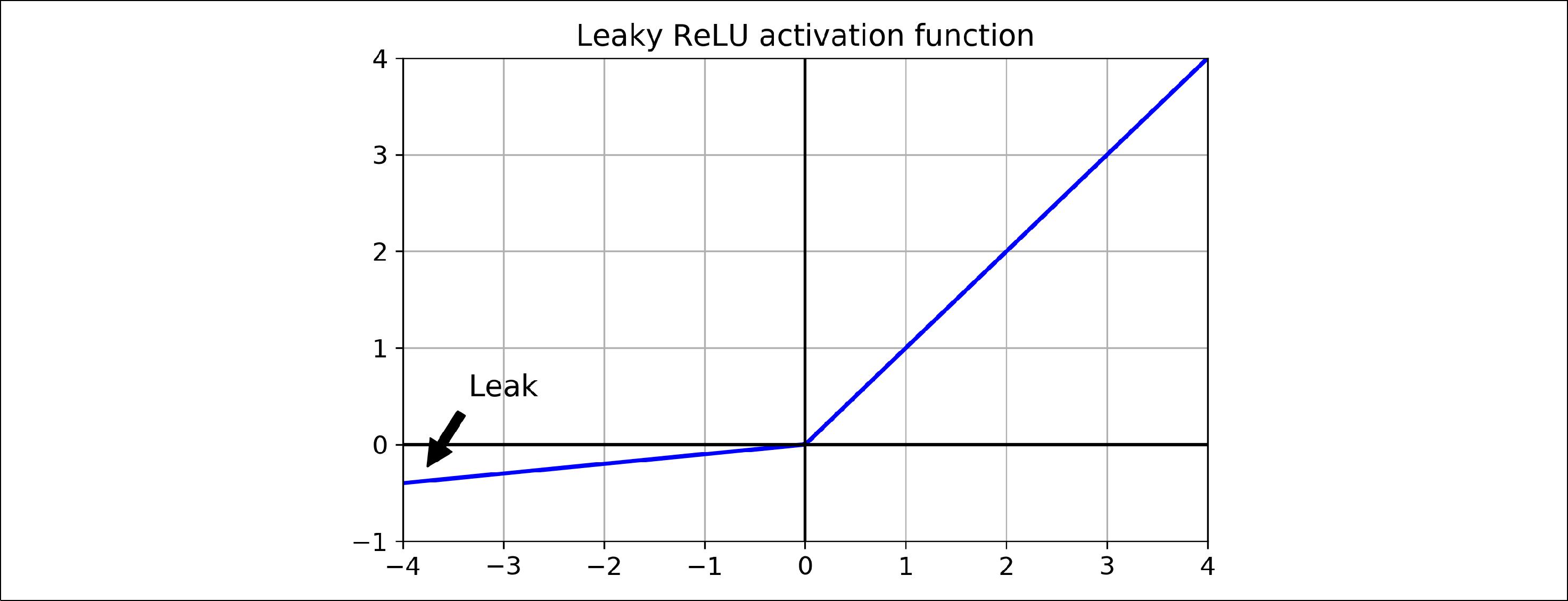


1. Unless it is part of the first hidden layer, a dead neuron may sometimes come back to life: gradient descent may indeed tweak neurons in the layers below in such a way that the weighted sum of the dead neuron’s inputs is positive again.
2. “Empirical Evaluation of Rectified Activations in Convolution Network,” B. Xu et al. (2015).



**Vanishing/Exploding Gradients Problems** **|** **329**

Finally, they also evaluated the *parametric leaky ReLU* (PReLU), where *α* is authorized to be learned during training (instead of being a hyperparameter, it becomes a parameter that can be modified by backpropagation like any other parameter). This was reported to strongly outperform ReLU on large image datasets, but on smaller datasets it runs the risk of overfitting the training set.



*Figure 11-2. Leaky ReLU*

Last but not least, a [2015 paper](https://homl.info/50) by Djork-Arné Clevert et al.[6](#page6) proposed a new activa‐ tion function called the *exponential linear unit* (ELU) that outperformed all the ReLU variants in their experiments: training time was reduced and the neural network per‐ formed better on the test set. It is represented in [Figure 11-3](#page7), and [Equation 11-2](#page6) shows its definition.

*Equation 11-2. ELU activation function*

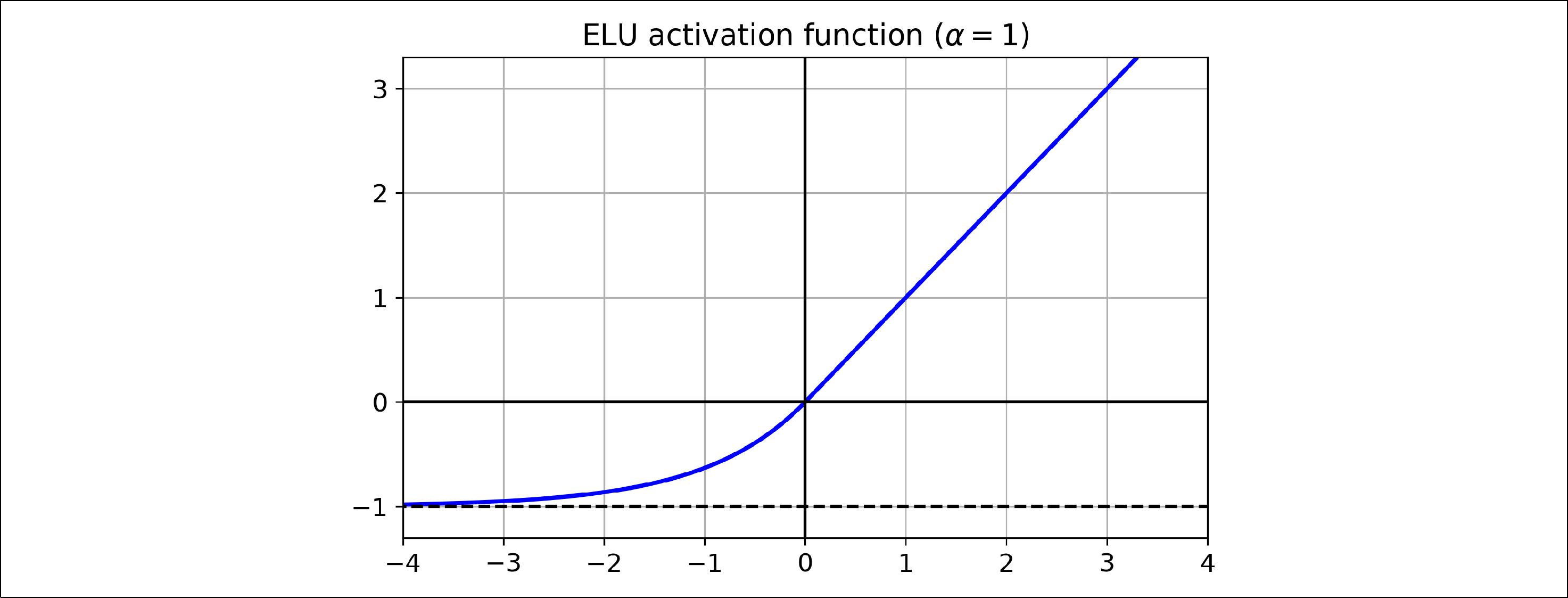
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ELU*α* *z* | = | *α* exp *z* − 1 | if | *z* < 0 |  |
| *z* | if | *z* ≥ 0 |  |



1. “Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs),” D. Clevert, T. Unterthiner, S. Hochreiter (2015).



**330** **|** **Chapter 11: Training Deep Neural Networks**



*Figure 11-3. ELU activation function*

It looks a lot like the ReLU function, with a few major differences:

* First it takes on negative values when *z* < 0, which allows the unit to have an average output closer to 0. This helps alleviate the vanishing gradients problem, as discussed earlier. The hyperparameter *α* defines the value that the ELU func‐ tion approaches when *z* is a large negative number. It is usually set to 1, but you can tweak it like any other hyperparameter if you want.
* Second, it has a nonzero gradient for *z* < 0, which avoids the dead neurons prob‐ lem.
* Third, if *α* is equal to 1 then the function is smooth everywhere, including around *z* = 0, which helps speed up Gradient Descent, since it does not bounce as much left and right of *z* = 0.

The main drawback of the ELU activation function is that it is slower to compute than the ReLU and its variants (due to the use of the exponential function), but dur‐ ing training this is compensated by the faster convergence rate. However, at test time an ELU network will be slower than a ReLU network.

Moreover, in a [2017 paper](https://homl.info/selu)[7](#page7) by Günter Klambauer et al., called “Self-Normalizing Neural Networks”, the authors showed that if you build a neural network composed exclusively of a stack of dense layers, and if all hidden layers use the SELU activation function (which is just a scaled version of the ELU activation function, as its name suggests), then the network will *self-normalize*: the output of each layer will tend to preserve mean 0 and standard deviation 1 during training, which solves the vanish‐ ing/exploding gradients problem. As a result, this activation function often outper‐



7 “Self-Normalizing Neural Networks, " G. Klambauer, T. Unterthiner and A. Mayr (2017).



**Vanishing/Exploding Gradients Problems** **|** **331**

forms other activation functions very significantly for such neural nets (especially deep ones). However, there are a few conditions for self-normalization to happen:

* The input features must be standardized (mean 0 and standard deviation 1).
* Every hidden layer’s weights must also be initialized using LeCun normal initiali‐ zation. In Keras, this means setting kernel\_initializer="lecun\_normal".
* The network’s architecture must be sequential. Unfortunately, if you try to use SELU in non-sequential architectures, such as recurrent networks (see ???) or networks with *skip connections* (i.e., connections that skip layers, such as in wide & deep nets), self-normalization will not be guaranteed, so SELU will not neces‐ sarily outperform other activation functions.
* The paper only guarantees self-normalization if all layers are dense. However, in practice the SELU activation function seems to work great with convolutional neural nets as well (see Chapter 14).



So which activation function should you use for the hidden layers of your deep neural networks? Although your mileage will vary, in general SELU > ELU > leaky ReLU (and its variants) > ReLU > tanh > logistic. If the network’s architecture prevents it from self-normalizing, then ELU may perform better than SELU (since SELU is not smooth at *z* = 0). If you care a lot about runtime latency, then you may prefer leaky ReLU. If you don’t want to tweak yet another hyperparameter, you may just use the default *α* values used by Keras (e.g., 0.3 for the leaky ReLU). If you have spare time and computing power, you can use cross-validation to evaluate other activation functions, in particular RReLU if your network is over‐ fitting, or PReLU if you have a huge training set.

To use the leaky ReLU activation function, you must create a LeakyReLU instance like this:

leaky\_relu = keras.layers.LeakyReLU(alpha=0.2)

layer = keras.layers.Dense(10, activation=leaky\_relu, kernel\_initializer="he\_normal")

For PReLU, just replace LeakyRelu(alpha=0.2) with PReLU(). There is currently no official implementation of RReLU in Keras, but you can fairly easily implement your own (see the exercises at the end of Chapter 12).

For SELU activation, just set activation="selu" and kernel\_initial izer="lecun\_normal" when creating a layer:

layer = keras.layers.Dense(10, activation="selu", kernel\_initializer="lecun\_normal")



**332** **|** **Chapter 11: Training Deep Neural Networks**

**Batch Normalization**

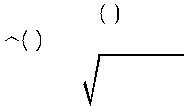
Although using He initialization along with ELU (or any variant of ReLU) can signifi‐ cantly reduce the vanishing/exploding gradients problems at the beginning of train‐ ing, it doesn’t guarantee that they won’t come back during training.

In a [2015 paper](https://homl.info/51),[8](#page9) Sergey Ioffe and Christian Szegedy proposed a technique called *Batch Normalization* (BN) to address the vanishing/exploding gradients problems.The technique consists of adding an operation in the model just before or after the activation function of each hidden layer, simply zero-centering and normalizing each input, then scaling and shifting the result using two new parameter vectors per layer: one for scaling, the other for shifting. In other words, this operation lets the model learn the optimal scale and mean of each of the layer’s inputs. In many cases, if you add a BN layer as the very first layer of your neural network, you do not need to standardize your training set (e.g., using a StandardScaler): the BN layer will do it for you (well, approximately, since it only looks at one batch at a time, and it can also rescale and shift each input feature).

In order to zero-center and normalize the inputs, the algorithm needs to estimate each input’s mean and standard deviation. It does so by evaluating the mean and stan‐ dard deviation of each input over the current mini-batch (hence the name “Batch Normalization”). The whole operation is summarized in [Equation 11-3](#page9).

*Equation 11-3. Batch Normalization algorithm*

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  | *m* | |  |  |  |  |  |
| 1 . | **μ***B*= | 1 | | |  |  | ∑*B* **x** *i* | | | |  |  |
|  |  |  |  |  |  |
|  |  | *mB i* = 1 | | | | | |  |  |  |  |  |
|  |  |  |  |  |  |  | *m* |  |  |  |  |  |
| 2 . | **σ***B*2= |  |  | 1 |  |  | ∑*B* |  | **x** *i* | | − **μ***B* 2 |  |
|  |  |  |  |  |  |  |
|  |  |  |  | *mB i* = 1 | | | | |  |  |  |  |
| 3 . | **x** *i*= |  |  | **x** *i* | |  | − **μ** | *B* |  |  |  |  |
|  |  | **σ** 2+ | | | |  |  |  |  |  |
|  |  |  |  |  |  | *B* | |  |  |  |  |  |

4 . **z***i* = **γ** ⊗ **x***i* + **β**



* **μ***B*is the vector of input means, evaluated over the whole mini-batch*B*(it con‐tains one mean per input).



1. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift,” S. Ioffe and C. Szegedy (2015).



**Vanishing/Exploding Gradients Problems** **|** **333**

* **σ***B*is the vector of input standard deviations, also evaluated over the whole mini-batch (it contains one standard deviation per input).
* *mB* is the number of instances in the mini-batch.
* **x***(i)*is the vector of zero-centered and normalized inputs for instance*i*.
* **γ** is the output scale parameter vector for the layer (it contains one scale parame‐ter per input).
* ⊗ represents element-wise multiplication (each input is multiplied by its corre‐ sponding output scale parameter).
* **β** is the output shift (offset) parameter vector for the layer (it contains one offsetparameter per input). Each input is offset by its corresponding shift parameter.
* *ϵ* is a tiny number to avoid division by zero (typically 10–5). This is called a *smoothing term*.
* **z***(i)*is the output of the BN operation: it is a rescaled and shifted version of theinputs.



So during training, BN just standardizes its inputs then rescales and offsets them. Good! What about at test time? Well it is not that simple. Indeed, we may need to make predictions for individual instances rather than for batches of instances: in this case, we will have no way to compute each input’s mean and standard deviation. Moreover, even if we do have a batch of instances, it may be too small, or the instan‐ ces may not be independent and identically distributed (IID), so computing statistics over the batch instances would be unreliable (during training, the batches should not be too small, if possible more than 30 instances, and all instances should be IID, as we saw in Chapter 4). One solution could be to wait until the end of training, then run the whole training set through the neural network, and compute the mean and stan‐ dard deviation of each input of the BN layer. These “final” input means and standard deviations can then be used instead of the batch input means and standard deviations when making predictions. However, it is often preferred to estimate these final statis‐ tics during training using a moving average of the layer’s input means and standard deviations. To sum up, four parameter vectors are learned in each batch-normalized layer: **γ** (the ouput scale vector) and **β** (the output offset vector) are learned through regular backpropagation, and **μ** (the final input mean vector), and **σ** (the final input standard deviation vector) are estimated using an exponential moving average. Note that **μ** and **σ** are estimated during training, but they are not used at all during train‐ ing, only after training (to replace the batch input means and standard deviations in [Equation 11-3](#page9)).

The authors demonstrated that this technique considerably improved all the deep neural networks they experimented with, leading to a huge improvement in the ImageNet classification task (ImageNet is a large database of images classified into many classes and commonly used to evaluate computer vision systems). The vanish‐



**334** **|** **Chapter 11: Training Deep Neural Networks**

ing gradients problem was strongly reduced, to the point that they could use saturat‐ ing activation functions such as the tanh and even the logistic activation function. The networks were also much less sensitive to the weight initialization. They were able to use much larger learning rates, significantly speeding up the learning process. Specifically, they note that “Applied to a state-of-the-art image classification model, Batch Normalization achieves the same accuracy with 14 times fewer training steps, and beats the original model by a significant margin. […] Using an ensemble of batch-normalized networks, we improve upon the best published result on ImageNet classification: reaching 4.9% top-5 validation error (and 4.8% test error), exceeding the accuracy of human raters.” Finally, like a gift that keeps on giving, Batch Normal‐ ization also acts like a regularizer, reducing the need for other regularization techni‐ ques (such as dropout, described later in this chapter).

Batch Normalization does, however, add some complexity to the model (although it can remove the need for normalizing the input data, as we discussed earlier). More‐ over, there is a runtime penalty: the neural network makes slower predictions due to the extra computations required at each layer. So if you need predictions to be lightning-fast, you may want to check how well plain ELU + He initialization perform before playing with Batch Normalization.

You may find that training is rather slow, because each epoch takes much more time when you use batch normalization. However, this is usually counterbalanced by the fact that convergence is much faster with BN, so it will take fewer epochs to reach the same per‐ formance. All in all, *wall time* will usually be smaller (this is the time measured by the clock on your wall).



**Implementing Batch Normalization with Keras**

As with most things with Keras, implementing Batch Normalization is quite simple. Just add a BatchNormalization layer before or after each hidden layer’s activation function, and optionally add a BN layer as well as the first layer in your model. For example, this model applies BN after every hidden layer and as the first layer in the model (after flattening the input images):

model = keras.models.Sequential([ keras.layers.Flatten(input\_shape=[28, 28]), keras.layers.BatchNormalization(),

keras.layers.Dense(300, activation="elu", kernel\_initializer="he\_normal"), keras.layers.BatchNormalization(),

keras.layers.Dense(100, activation="elu", kernel\_initializer="he\_normal"), keras.layers.BatchNormalization(),

keras.layers.Dense(10, activation="softmax")

])



**Vanishing/Exploding Gradients Problems** **|** **335**

That’s all! In this tiny example with just two hidden layers, it’s unlikely that Batch Normalization will have a very positive impact, but for deeper networks it can make a tremendous difference.

Let’s zoom in a bit. If you display the model summary, you can see that each BN layer adds 4 parameters per input: **γ**, **β**, **μ** and **σ** (for example, the first BN layer adds 3136 parameters, which is 4 times 784). The last two parameters, **μ** and **σ**, are the moving averages, they are not affected by backpropagation, so Keras calls them “Non-trainable”[9](#page12) (if you count the total number of BN parameters, 3136 + 1200 + 400, and divide by two, you get 2,368, which is the total number of non-trainable params in this model).

**>>>** model.summary()Model: "sequential\_3"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

flatten\_3 (Flatten) (None, 784) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |
| --- | --- |
| batch\_normalization\_v2 (Batc (None, 784) | 3136 |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_50 (Dense) (None, 300) 235500

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |
| --- | --- |
| batch\_normalization\_v2\_1 (Ba (None, 300) | 1200 |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_51 (Dense) (None, 100) 30100

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |
| --- | --- |
| batch\_normalization\_v2\_2 (Ba (None, 100) | 400 |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_52 (Dense) (None, 10) 1010

=================================================================

Total params: 271,346 Trainable params: 268,978 Non-trainable params: 2,368

Let’s look at the parameters of the first BN layer. Two are trainable (by backprop), and two are not:

**>>>** [(var.name,var.trainable) **for** var **in** model.layers[1].variables][('batch\_normalization\_v2/gamma:0', True), ('batch\_normalization\_v2/beta:0', True), ('batch\_normalization\_v2/moving\_mean:0', False), ('batch\_normalization\_v2/moving\_variance:0', False)]

Now when you create a BN layer in Keras, it also creates two operations that will be called by Keras at each iteration during training. These operations will update the



1. However, they are estimated during training, based on the training data, so arguably they *are* trainable. In Keras, “Non-trainable” really means “untouched by backpropagation”.



**336** **|** **Chapter 11: Training Deep Neural Networks**

moving averages. Since we are using the TensorFlow backend, these operations are TensorFlow operations (we will discuss TF operations in Chapter 12).

**>>>** model.layers[1].updates

[<tf.Operation 'cond\_2/Identity' type=Identity>, <tf.Operation 'cond\_3/Identity' type=Identity>]

The authors of the BN paper argued in favor of adding the BN layers before the acti‐ vation functions, rather than after (as we just did). There is some debate about this, as it seems to depend on the task. So that’s one more thing you can experiment with to see which option works best on your dataset. To add the BN layers before the activa‐ tion functions, we must remove the activation function from the hidden layers, and add them as separate layers after the BN layers. Moreover, since a Batch Normaliza‐ tion layer includes one offset parameter per input, you can remove the bias term from the previous layer (just pass use\_bias=False when creating it):

model = keras.models.Sequential([ keras.layers.Flatten(input\_shape=[28, 28]), keras.layers.BatchNormalization(),

keras.layers.Dense(300, kernel\_initializer="he\_normal", use\_bias=False), keras.layers.BatchNormalization(),

keras.layers.Activation("elu"),

keras.layers.Dense(100, kernel\_initializer="he\_normal", use\_bias=False), keras.layers.Activation("elu"),

keras.layers.BatchNormalization(), keras.layers.Dense(10, activation="softmax")

])

The BatchNormalization class has quite a few hyperparameters you can tweak. The defaults will usually be fine, but you may occasionally need to tweak the momentum. This hyperparameter is used when updating the exponential moving averages: given a new value **v** (i.e., a new vector of input means or standard deviations computed over the current batch), the running average  is updated using the following equation:

**v**  **v** × momentum + **v** × 1 − momentum



A good momentum value is typically close to 1—for example, 0.9, 0.99, or 0.999 (you want more 9s for larger datasets and smaller mini-batches).

Another important hyperparameter is axis: it determines which axis should be nor‐ malized. It defaults to –1, meaning that by default it will normalize the last axis (using the means and standard deviations computed across the *other* axes). For example, when the input batch is 2D (i.e., the batch shape is [batch size, features]), this means that each input feature will be normalized based on the mean and standard deviation computed across all the instances in the batch. For example, the first BN layer in the previous code example will independently normalize (and rescale and shift) each of the 784 input features. However, if we move the first BN layer before the Flatten



**Vanishing/Exploding Gradients Problems** **|** **337**

layer, then the input batches will be 3D, with shape [batch size, height, width], there‐ fore the BN layer will compute 28 means and 28 standard deviations (one per column of pixels, computed across all instances in the batch, and all rows in the column), and it will normalize all pixels in a given column using the same mean and standard devi‐ ation. There will also be just 28 scale parameters and 28 shift parameters. If instead you still want to treat each of the 784 pixels independently, then you should set

axis=[1, 2].

Notice that the BN layer does not perform the same computation during training and after training: it uses batch statistics during training, and the “final” statistics after training (i.e., the final value of the moving averages). Let’s take a peek at the source code of this class to see how this is handled:

**class BatchNormalization**(Layer):[...]

**def** call(self, inputs, training=None): **if** training **is** None:

training = keras.backend.learning\_phase() [...]

The call() method is the one that actually performs the computations, and as you can see it has an extra training argument: if it is None it falls back to keras.back end.learning\_phase(), which returns 1 during training (the fit() method ensures that). Otherwise, it returns 0. If you ever need to write a custom layer, and it needs to behave differently during training and testing, simply use the same pattern (we will discuss custom layers in Chapter 12).

Batch Normalization has become one of the most used layers in deep neural net‐ works, to the point that it is often omitted in the diagrams, as it is assumed that BN is added after every layer. However, a very recent [paper](https://homl.info/fixup)[10](#page14) by Hongyi Zhang et al. may well change this: the authors show that by using a novel fixed-update (fixup) weight initialization technique, they manage to train a very deep neural network (10,000 lay‐ ers!) without BN, achieving state-of-the-art performance on complex image classifi‐ cation tasks.

**Gradient Clipping**

Another popular technique to lessen the exploding gradients problem is to simply clip the gradients during backpropagation so that they never exceed some threshold. This is called [*Gradient Clipping*](https://homl.info/52).[11](#page14) This technique is most often used in recurrent neu‐



1. “Fixup Initialization: Residual Learning Without Normalization,” Hongyi Zhang, Yann N. Dauphin, Tengyu Ma (2019).
2. “On the difficulty of training recurrent neural networks,” R. Pascanu et al. (2013).



**338** **|** **Chapter 11: Training Deep Neural Networks**

ral networks, as Batch Normalization is tricky to use in RNNs, as we will see in ???. For other types of networks, BN is usually sufficient.

In Keras, implementing Gradient Clipping is just a matter of setting the clipvalue or clipnorm argument when creating an optimizer. For example:

optimizer = keras.optimizers.SGD(clipvalue=1.0) model.compile(loss="mse", optimizer=optimizer)

This will clip every component of the gradient vector to a value between –1.0 and 1.0. This means that all the partial derivatives of the loss (with regards to each and every trainable parameter) will be clipped between –1.0 and 1.0. The threshold is a hyper‐ parameter you can tune. Note that it may change the orientation of the gradient vec‐ tor: for example, if the original gradient vector is [0.9, 100.0], it points mostly in the direction of the second axis, but once you clip it by value, you get [0.9, 1.0], which points roughly in the diagonal between the two axes. In practice however, this approach works well. If you want to ensure that Gradient Clipping does not change the direction of the gradient vector, you should clip by norm by setting clipnorm instead of clipvalue. This will clip the whole gradient if its ℓ2 norm is greater than the threshold you picked. For example, if you set clipnorm=1.0, then the vector [0.9, 100.0] will be clipped to [0.00899964, 0.9999595], preserving its orientation, but almost eliminating the first component. If you observe that the gradients explode during training (you can track the size of the gradients using TensorBoard), you may want to try both clipping by value and clipping by norm, with different threshold, and see which option performs best on the validation set.

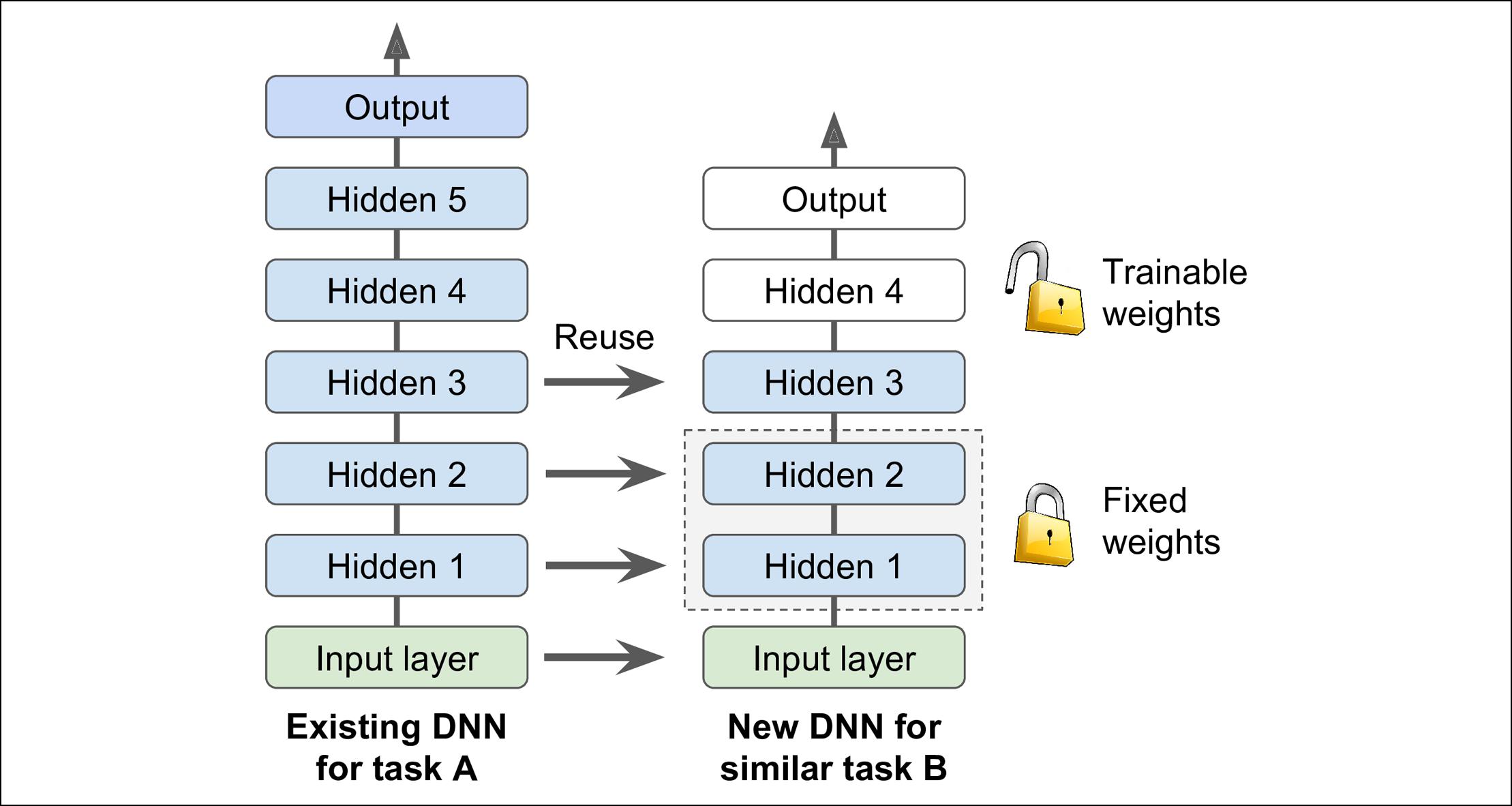
**Reusing Pretrained Layers**

It is generally not a good idea to train a very large DNN from scratch: instead, you should always try to find an existing neural network that accomplishes a similar task to the one you are trying to tackle (we will discuss how to find them in Chapter 14), then just reuse the lower layers of this network: this is called *transfer learning*. It will not only speed up training considerably, but will also require much less training data.

For example, suppose that you have access to a DNN that was trained to classify pic‐ tures into 100 different categories, including animals, plants, vehicles, and everyday objects. You now want to train a DNN to classify specific types of vehicles. These tasks are very similar, even partly overlapping, so you should try to reuse parts of the first network (see [Figure 11-4](#page16)).



**Reusing Pretrained Layers** **|** **339**



*Figure 11-4. Reusing pretrained layers*

If the input pictures of your new task don’t have the same size as the ones used in the original task, you will usually have to add a preprocessing step to resize them to the size expected by the origi‐ nal model. More generally, transfer learning will work best when the inputs have similar low-level features.



The output layer of the original model should usually be replaced since it is most likely not useful at all for the new task, and it may not even have the right number of outputs for the new task.

Similarly, the upper hidden layers of the original model are less likely to be as useful as the lower layers, since the high-level features that are most useful for the new task may differ significantly from the ones that were most useful for the original task. You want to find the right number of layers to reuse.



The more similar the tasks are, the more layers you want to reuse (starting with the lower layers). For very similar tasks, you can try keeping all the hidden layers and just replace the output layer.

Try freezing all the reused layers first (i.e., make their weights non-trainable, so gradi‐ ent descent won’t modify them), then train your model and see how it performs. Then try unfreezing one or two of the top hidden layers to let backpropagation tweak them and see if performance improves. The more training data you have, the more



**340** **|** **Chapter 11: Training Deep Neural Networks**

layers you can unfreeze. It is also useful to reduce the learning rate when you unfreeze reused layers: this will avoid wrecking their fine-tuned weights.

If you still cannot get good performance, and you have little training data, try drop‐ ping the top hidden layer(s) and freeze all remaining hidden layers again. You can iterate until you find the right number of layers to reuse. If you have plenty of train‐ ing data, you may try replacing the top hidden layers instead of dropping them, and even add more hidden layers.

**Transfer Learning With Keras**

Let’s look at an example. Suppose the fashion MNIST dataset only contained 8 classes, for example all classes except for sandals and shirts. Someone built and trained a Keras model on that set and got reasonably good performance (>90% accuracy). Let’s call this model A. You now want to tackle a different task: you have images of sandals and shirts, and you want to train a binary classifier (positive=shirts, negative=san‐ dals). However, your dataset is quite small, you only have 200 labeled images. When you train a new model for this task (let’s call it model B), with the same architecture as model A, it performs reasonably well (97.2% accuracy), but since it’s a much easier task (there are just 2 classes), you were hoping for more. While drinking your morn‐ ing coffee, you realize that your task is quite similar to task A, so perhaps transfer learning can help? Let’s find out!

First, you need to load model A, and create a new model based on the model A’s lay‐ ers. Let’s reuse all layers except for the output layer:

model\_A = keras.models.load\_model("my\_model\_A.h5") model\_B\_on\_A = keras.models.Sequential(model\_A.layers[:-1]) model\_B\_on\_A.add(keras.layers.Dense(1, activation="sigmoid"))

Note that model\_A and model\_B\_on\_A now share some layers. When you train model\_B\_on\_A, it will also affect model\_A. If you want to avoid that, you need to clone model\_A before you reuse its layers. To do this, you must clone model A’s architecture, then copy its weights (since clone\_model() does not clone the weights):

model\_A\_clone = keras.models.clone\_model(model\_A) model\_A\_clone.set\_weights(model\_A.get\_weights())

Now we could just train model\_B\_on\_A for task B, but since the new output layer was initialized randomly, it will make large errors, at least during the first few epochs, so there will be large error gradients that may wreck the reused weights. To avoid this, one approach is to freeze the reused layers during the first few epochs, giving the new layer some time to learn reasonable weights. To do this, simply set every layer’s train able attribute to False and compile the model:

**for** layer **in** model\_B\_on\_A.layers[:-1]: layer.trainable = False



**Reusing Pretrained Layers** **|** **341**

model\_B\_on\_A.compile(loss="binary\_crossentropy", optimizer="sgd", metrics=["accuracy"])

You must always compile your model after you freeze or unfreeze layers.



Next, we can train the model for a few epochs, then unfreeze the reused layers (which requires compiling the model again) and continue training to fine-tune the reused layers for task B. After unfreezing the reused layers, it is usually a good idea to reduce the learning rate, once again to avoid damaging the reused weights:

history = model\_B\_on\_A.fit(X\_train\_B, y\_train\_B, epochs=4, validation\_data=(X\_valid\_B, y\_valid\_B))

**for** layer **in** model\_B\_on\_A.layers[:-1]: layer.trainable = True

optimizer = keras.optimizers.SGD(lr=1e-4) *# the default lr is 1e-3* model\_B\_on\_A.compile(loss="binary\_crossentropy", optimizer=optimizer,

metrics=["accuracy"])

history = model\_B\_on\_A.fit(X\_train\_B, y\_train\_B, epochs=16, validation\_data=(X\_valid\_B, y\_valid\_B))

So, what’s the final verdict? Well this model’s test accuracy is 99.25%, which means that transfer learning reduced the error rate from 2.8% down to almost 0.7%! That’s a factor of 4!

**>>>** model\_B\_on\_A.evaluate(X\_test\_B,y\_test\_B)[0.06887910133600235, 0.9925]

Are you convinced? Well you shouldn’t be: I cheated! :) I tried many configurations until I found one that demonstrated a strong improvement. If you try to change the classes or the random seed, you will see that the improvement generally drops, or even vanishes or reverses. What I did is called “torturing the data until it confesses”. When a paper just looks too positive, you should be suspicious: perhaps the flashy new technique does not help much (in fact, it may even degrade performance), but the authors tried many variants and reported only the best results (which may be due to shear luck), without mentioning how many failures they encountered on the way. Most of the time, this is not malicious at all, but it is part of the reason why so many results in Science can never be reproduced.

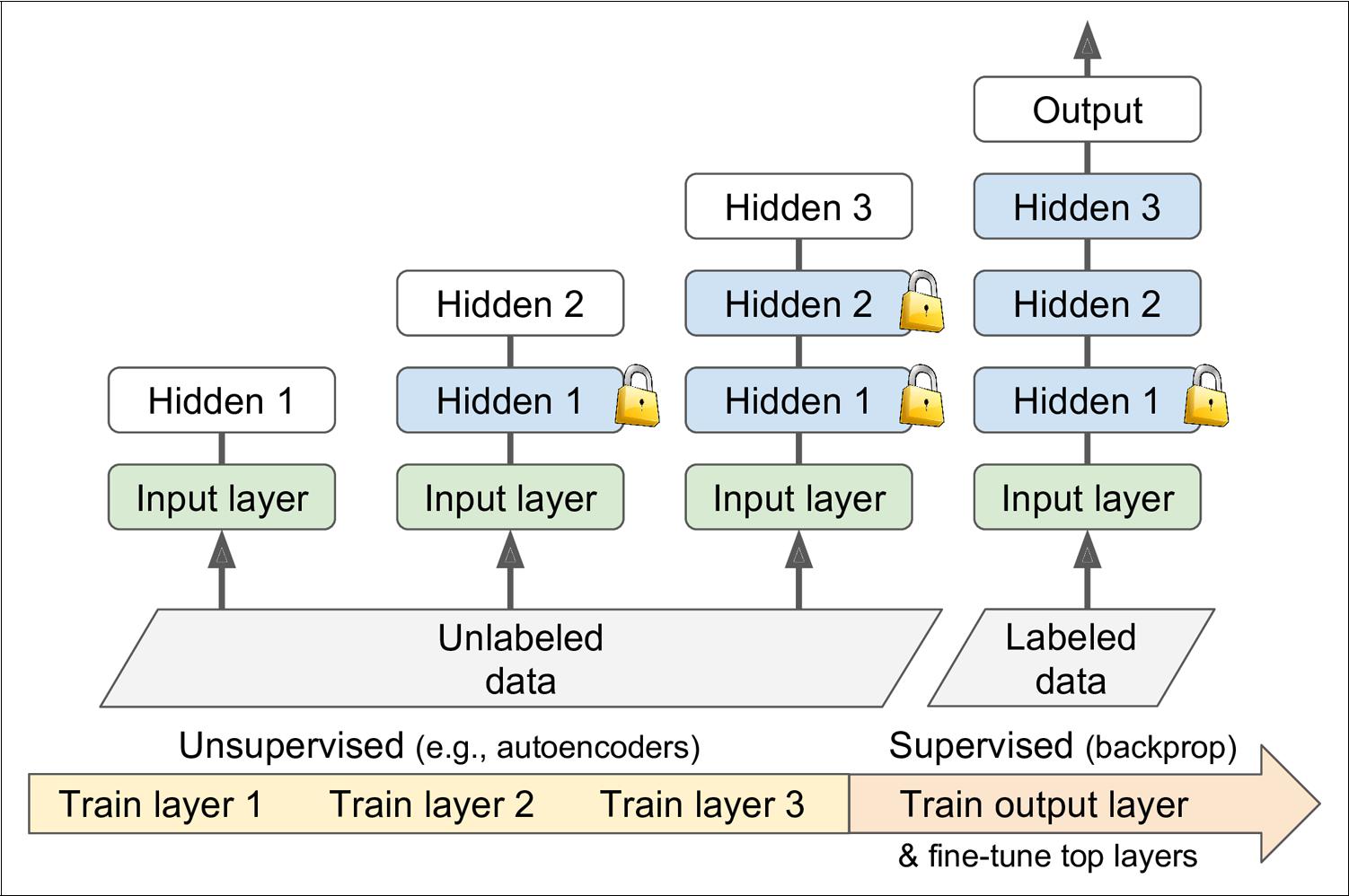
So why did I cheat? Well it turns out that transfer learning does not work very well with small dense networks: it works best with deep convolutional neural networks, so we will revisit transfer learning in Chapter 14, using the same techniques (and this time there will be no cheating, I promise!).



**342** **|** **Chapter 11: Training Deep Neural Networks**

**Unsupervised Pretraining**

Suppose you want to tackle a complex task for which you don’t have much labeled training data, but unfortunately you cannot find a model trained on a similar task. Don’t lose all hope! First, you should of course try to gather more labeled training data, but if this is too hard or too expensive, you may still be able to perform *unsuper‐* *vised pretraining* (see[Figure 11-5](#page19)). It is often rather cheap to gather unlabeled train‐ing examples, but quite expensive to label them. If you can gather plenty of unlabeled training data, you can try to train the layers one by one, starting with the lowest layer and then going up, using an unsupervised feature detector algorithm such as *Restric‐* *ted Boltzmann Machines* (RBMs; see???) or autoencoders (see???). Each layer istrained on the output of the previously trained layers (all layers except the one being trained are frozen). Once all layers have been trained this way, you can add the output layer for your task, and fine-tune the final network using supervised learning (i.e., with the labeled training examples). At this point, you can unfreeze all the pretrained layers, or just some of the upper ones.



*Figure 11-5. Unsupervised pretraining*

This is a rather long and tedious process, but it often works well; in fact, it is this technique that Geoffrey Hinton and his team used in 2006 and which led to the revival of neural networks and the success of Deep Learning. Until 2010, unsuper‐ vised pretraining (typically using RBMs) was the norm for deep nets, and it was only after the vanishing gradients problem was alleviated that it became much more com‐



**Reusing Pretrained Layers** **|** **343**

mon to train DNNs purely using supervised learning. However, unsupervised pre‐ training (today typically using autoencoders rather than RBMs) is still a good option when you have a complex task to solve, no similar model you can reuse, and little labeled training data but plenty of unlabeled training data.

**Pretraining on an Auxiliary Task**

If you do not have much labeled training data, one last option is to train a first neural network on an auxiliary task for which you can easily obtain or generate labeled training data, then reuse the lower layers of that network for your actual task. The first neural network’s lower layers will learn feature detectors that will likely be reusa‐ ble by the second neural network.

For example, if you want to build a system to recognize faces, you may only have a few pictures of each individual—clearly not enough to train a good classifier. Gather‐ ing hundreds of pictures of each person would not be practical. However, you could gather a lot of pictures of random people on the web and train a first neural network to detect whether or not two different pictures feature the same person. Such a net‐ work would learn good feature detectors for faces, so reusing its lower layers would allow you to train a good face classifier using little training data.

For *natural language processing* (NLP) applications, you can easily download millions of text documents and automatically generate labeled data from it. For example, you could randomly mask out some words and train a model to predict what the missing words are (e.g., it should predict that the missing word in the sentence “What \_\_\_

you saying?” is probably “are” or “were”). If you can train a model to reach good per‐ formance on this task, then it will already know quite a lot about language, and you can certainly reuse it for your actual task, and fine-tune it on your labeled data (we will discuss more pretraining tasks in ???).

*Self-supervised learning* is when you automatically generate thelabels from the data itself, then you train a model on the resulting “labeled” dataset using supervised learning techniques. Since this approach requires no human labeling whatsoever, it is best classi‐ fied as a form of unsupervised learning.



**Faster Optimizers**

Training a very large deep neural network can be painfully slow. So far we have seen four ways to speed up training (and reach a better solution): applying a good initiali‐ zation strategy for the connection weights, using a good activation function, using Batch Normalization, and reusing parts of a pretrained network (possibly built on an auxiliary task or using unsupervised learning). Another huge speed boost comes from using a faster optimizer than the regular Gradient Descent optimizer. In this section



**344** **|** **Chapter 11: Training Deep Neural Networks**

we will present the most popular ones: Momentum optimization, Nesterov Acceler‐ ated Gradient, AdaGrad, RMSProp, and finally Adam and Nadam optimization.

**Momentum Optimization**

Imagine a bowling ball rolling down a gentle slope on a smooth surface: it will start out slowly, but it will quickly pick up momentum until it eventually reaches terminal velocity (if there is some friction or air resistance). This is the very simple idea behind *Momentum optimization*,[proposed by Boris Polyak in 1964](https://homl.info/54).[12](#page21)Incontrast, regularGradient Descent will simply take small regular steps down the slope, so it will take much more time to reach the bottom.

Recall that Gradient Descent simply updates the weights **θ** by directly subtracting the gradient of the cost function *J*(**θ**) with regards to the weights (∇**θ***J*(**θ**)) multiplied by the learning rate *η*. The equation is: **θ** ← **θ** – *η*∇**θ***J*(**θ**). It does not care about what the earlier gradients were. If the local gradient is tiny, it goes very slowly.

Momentum optimization cares a great deal about what previous gradients were: at each iteration, it subtracts the local gradient from the *momentum vector* **m** (multi‐ plied by the learning rate *η*), and it updates the weights by simply adding this momentum vector (see [Equation 11-4](#page21)). In other words, the gradient is used for accel‐ eration, not for speed. To simulate some sort of friction mechanism and prevent the momentum from growing too large, the algorithm introduces a new hyperparameter *β*, simply called the *momentum*, which must be set between 0 (high friction) and 1(no friction). A typical momentum value is 0.9.

*Equation 11-4. Momentum algorithm*

1 . **m**  *β***m**− *η*∇**θ***J***θ**

2 . **θ**  **θ** + **m**

You can easily verify that if the gradient remains constant, the terminal velocity (i.e., the maximum size of the weight updates) is equal to that gradient multiplied by the

learning rate *η* multiplied by 1 −1 *β* (ignoring the sign). For example, if *β* = 0.9, then the

terminal velocity is equal to 10 times the gradient times the learning rate, so Momen‐ tum optimization ends up going 10 times faster than Gradient Descent! This allows Momentum optimization to escape from plateaus much faster than Gradient Descent. In particular, we saw in Chapter 4 that when the inputs have very different scales the cost function will look like an elongated bowl (see Figure 4-7). Gradient Descent goes down the steep slope quite fast, but then it takes a very long time to go down the val‐



12 “Some methods of speeding up the convergence of iteration methods,” B. Polyak (1964).



**Faster Optimizers** **|** **345**

ley. In contrast, Momentum optimization will roll down the valley faster and faster until it reaches the bottom (the optimum). In deep neural networks that don’t use Batch Normalization, the upper layers will often end up having inputs with very dif‐ ferent scales, so using Momentum optimization helps a lot. It can also help roll past local optima.

Due to the momentum, the optimizer may overshoot a bit, then come back, overshoot again, and oscillate like this many times before stabilizing at the minimum. This is one of the reasons why it is good to have a bit of friction in the system: it gets rid of these oscillations and thus speeds up convergence.



Implementing Momentum optimization in Keras is a no-brainer: just use the SGD optimizer and set its momentum hyperparameter, then lie back and profit!

optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9)

The one drawback of Momentum optimization is that it adds yet another hyperpara‐ meter to tune. However, the momentum value of 0.9 usually works well in practice and almost always goes faster than regular Gradient Descent.

**Nesterov Accelerated Gradient**

One small variant to Momentum optimization, proposed by [Yurii Nesterov in 1983](https://homl.info/55),[13](#page22) is almost always faster than vanilla Momentum optimization. The idea of *Nesterov* *Momentum optimization*, or *Nesterov Accelerated Gradient* (NAG), is to measure thegradient of the cost function not at the local position but slightly ahead in the direc‐ tion of the momentum (see [Equation 11-5](#page22)). The only difference from vanilla Momentum optimization is that the gradient is measured at **θ** + *β***m** rather than at **θ**.

*Equation 11-5. Nesterov Accelerated Gradient algorithm*

1 . **m**  *β***m**− *η*∇**θ***J***θ** +*β***m**

2 . **θ**  **θ** + **m**

This small tweak works because in general the momentum vector will be pointing in the right direction (i.e., toward the optimum), so it will be slightly more accurate to use the gradient measured a bit farther in that direction rather than using the gradi‐ ent at the original position, as you can see in [Figure 11-6](#page23) (where ∇1 represents the gradient of the cost function measured at the starting point **θ**, and ∇2 represents the



1. “A Method for Unconstrained Convex Minimization Problem with the Rate of Convergence O(1/k2),” Yurii Nesterov (1983).

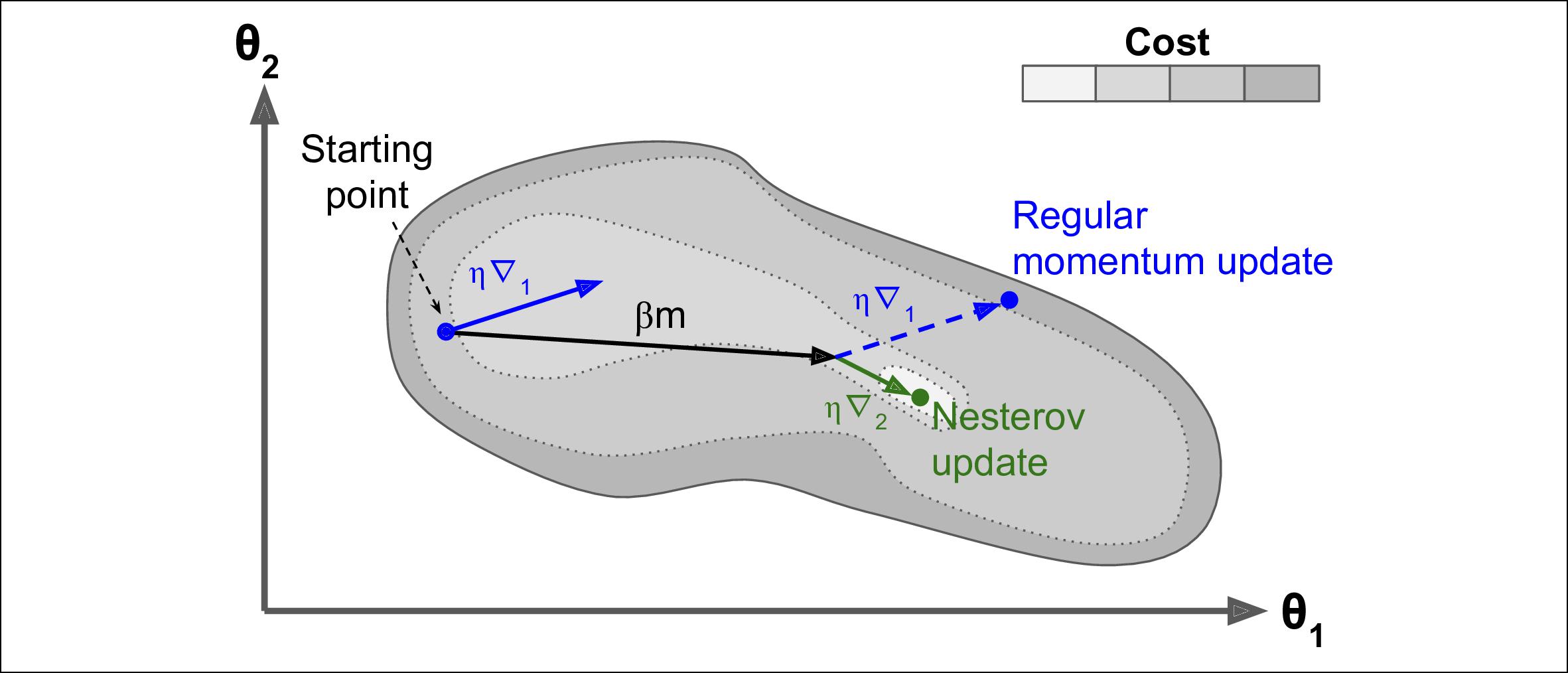


**346** **|** **Chapter 11: Training Deep Neural Networks**

gradient at the point located at **θ** + *β***m**). As you can see, the Nesterov update ends up slightly closer to the optimum. After a while, these small improvements add up and NAG ends up being significantly faster than regular Momentum optimization. More‐ over, note that when the momentum pushes the weights across a valley, ∇1 continues to push further across the valley, while ∇2 pushes back toward the bottom of the val‐ ley. This helps reduce oscillations and thus converges faster.

NAG will almost always speed up training compared to regular Momentum optimi‐ zation. To use it, simply set nesterov=True when creating the SGD optimizer:

optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9, nesterov=True)



*Figure 11-6. Regular versus Nesterov Momentum optimization*

**AdaGrad**

Consider the elongated bowl problem again: Gradient Descent starts by quickly going down the steepest slope, then slowly goes down the bottom of the valley. It would be nice if the algorithm could detect this early on and correct its direction to point a bit more toward the global optimum.

The [*AdaGrad* algorithm](https://homl.info/56)[14](#page23) achieves this by scaling down the gradient vector along the steepest dimensions (see [Equation 11-6](#page23)):

*Equation 11-6. AdaGrad algorithm*

1 . **s**  **s** +∇**θ***J***θ** ⊗ ∇**θ***J***θ**

2 . **θ**  **θ** −*η*∇**θ***J***θ** ⊘ **s** +



14 “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization,” J. Duchi et al. (2011).



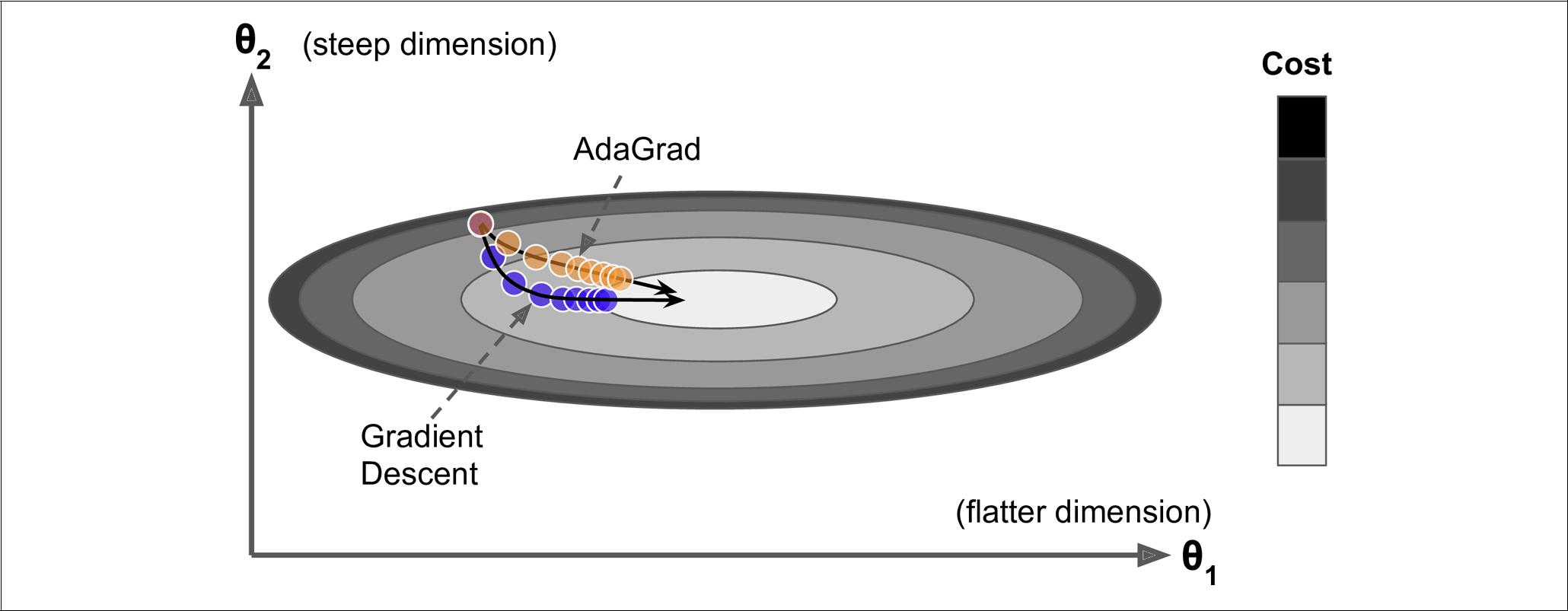
**Faster Optimizers** **|** **347**

The first step accumulates the square of the gradients into the vector **s** (recall that the ⊗ symbol represents the element-wise multiplication). This vectorized form is equiv‐ alent to computing *si* ← *si* + (∂ *J*(**θ**) / ∂ *θi*)2 for each element *si* of the vector **s**; in other words, each *si* accumulates the squares of the partial derivative of the cost function with regards to parameter *θi*. If the cost function is steep along the ith dimension, then *si* will get larger and larger at each iteration.

The second step is almost identical to Gradient Descent, but with one big difference: the gradient vector is scaled down by a factor of + (the ⊘ symbol represents the element-wise division, and ϵ is a smoothing term to avoid division by zero, typically

|  |  |  |
| --- | --- | --- |
| set | to 10–10). This | vectorized form is equivalent to computing |
| *θi* | *θi* − *η* ∂*J* **θ**/ ∂*θi*/ *si* + | for all parameters *θi* (simultaneously). |

In short, this algorithm decays the learning rate, but it does so faster for steep dimen‐ sions than for dimensions with gentler slopes. This is called an *adaptive learning rate*. It helps point the resulting updates more directly toward the global optimum (see [Figure 11-7](#page24)). One additional benefit is that it requires much less tuning of the learn‐ ing rate hyperparameter *η*.



*Figure 11-7. AdaGrad versus Gradient Descent*

AdaGrad often performs well for simple quadratic problems, but unfortunately it often stops too early when training neural networks. The learning rate gets scaled down so much that the algorithm ends up stopping entirely before reaching the global optimum. So even though Keras has an Adagrad optimizer, you should not use it to train deep neural networks (it may be efficient for simpler tasks such as Linear Regression, though). However, understanding Adagrad is helpful to grasp the other adaptive learning rate optimizers.



**348** **|** **Chapter 11: Training Deep Neural Networks**

**RMSProp**

Although AdaGrad slows down a bit too fast and ends up never converging to the global optimum, the *RMSProp* algorithm[15](#page25) fixes this by accumulating only the gradi‐ ents from the most recent iterations (as opposed to all the gradients since the begin‐ ning of training). It does so by using exponential decay in the first step (see [Equation](#page25) [11-7](#page25)).

*Equation 11-7. RMSProp algorithm*

1 . **s**  *β***s**+ 1 − *β*∇**θ***J***θ** ⊗ ∇**θ***J***θ**

2 . **θ**  **θ** −*η*∇**θ***J***θ** ⊘ **s** +



The decay rate *β* is typically set to 0.9. Yes, it is once again a new hyperparameter, but this default value often works well, so you may not need to tune it at all.

As you might expect, Keras has an RMSProp optimizer:

optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9)

Except on very simple problems, this optimizer almost always performs much better than AdaGrad. In fact, it was the preferred optimization algorithm of many research‐ ers until Adam optimization came around.

**Adam and Nadam Optimization**

[*Adam*](https://homl.info/59),[16](#page25)whichstands for *adaptive moment estimation*, combines the ideas of Momen‐tum optimization and RMSProp: just like Momentum optimization it keeps track of an exponentially decaying average of past gradients, and just like RMSProp it keeps track of an exponentially decaying average of past squared gradients (see [Equation](#page26) [11-8](#page26)).[17](#page25)



1. This algorithm was created by Geoffrey Hinton and Tijmen Tieleman in 2012, and presented by Geoffrey Hinton in his Coursera class on neural networks (slides: [*https://homl.info/57*](https://homl.info/57); video: [*https://homl.info/58*](https://homl.info/58)). Amusingly, since the authors did not write a paper to describe it, researchers often cite “slide 29 in lecture 6” in their papers.
2. “Adam: A Method for Stochastic Optimization,” D. Kingma, J. Ba (2015).
3. These are estimations of the mean and (uncentered) variance of the gradients. The mean is often called the *first moment*, while the variance is often called the *second moment*, hence the name of the algorithm.
4. 

**Faster Optimizers** **|** **349**

*Equation 11-8. Adam algorithm*

1 . **m**  *β*1**m**− 1 − *β*1∇**θ***J***θ**

2 . **s**  *β*2**s**+ 1 − *β*2∇**θ***J***θ** ⊗ ∇**θ***J***θ**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 3 . | **m** |  | **m** |  |  |  |  |
|  | 1 − *β* | *t* | | |  |
|  |  |  |  |
|  |  | 1 | | |  |  |  |
| 4 . | **s** |  | **s** |  |  |  |  |
| 1 − *β* | | *t* | | |  |
|  |  |  |
|  |  | 2 | |  |  |  |  |
| 5 . | **θ θ** +*η* **m** ⊘ **s** + | | | | | |  |



• *t* represents the iteration number (starting at 1).

If you just look at steps 1, 2, and 5, you will notice Adam’s close similarity to both Momentum optimization and RMSProp. The only difference is that step 1 computes an exponentially decaying average rather than an exponentially decaying sum, but these are actually equivalent except for a constant factor (the decaying average is just 1 – *β*1 times the decaying sum). Steps 3 and 4 are somewhat of a technical detail: since **m** and **s** are initialized at 0, they will be biased toward 0 at the beginning of training,so these two steps will help boost **m** and **s** at the beginning of training.

The momentum decay hyperparameter *β*1 is typically initialized to 0.9, while the scal‐ ing decay hyperparameter *β*2 is often initialized to 0.999. As earlier, the smoothing term *ϵ* is usually initialized to a tiny number such as 10–7. These are the default values for the Adam class (to be precise, epsilon defaults to None, which tells Keras to use keras.backend.epsilon(), which defaults to 10–7; you can change it using

keras.backend.set\_epsilon()).

optimizer = keras.optimizers.Adam(lr=0.001, beta\_1=0.9, beta\_2=0.999)

Since Adam is an adaptive learning rate algorithm (like AdaGrad and RMSProp), it requires less tuning of the learning rate hyperparameter *η*. You can often use the default value *η* = 0.001, making Adam even easier to use than Gradient Descent.



If you are starting to feel overwhelmed by all these different techni‐ ques, and wondering how to choose the right ones for your task, don’t worry: some practical guidelines are provided at the end of this chapter.

Finally, two variants of Adam are worth mentioning:

1. 

**350** **|** **Chapter 11: Training Deep Neural Networks**

* Adamax, introduced in the same paper as Adam: notice that in step 2 of [Equation](#page26) [11-8](#page26), Adam accumulates the squares of the gradients in **s** (with a greater weight for more recent weights). In step 5, if we ignore *ϵ* and steps 3 and 4 (which are technical details anyway), Adam just scales down the parameter updates by the square root of **s**. In short, Adam scales down the parameter updates by the ℓ2 norm of the time-decayed gradients (recall that the ℓ2 norm is the square root of the sum of squares). Adamax just replaces the ℓ2 norm with the ℓ∞ norm (a fancy way of saying the max). Specifically, it replaces step 2 in [Equation 11-8](#page26) with

 max *β*2,∇*θJ**θ*, it drops step 4, and in step 5 it scales down the gradient updates by a factor of **s**, which is just the max of the time-decayed gradients. In practice, this can make Adamax more stable than Adam, but this really depends on the dataset, and in general Adam actually performs better. So it’s just one more optimizer you can try if you experience problems with Adam on some task.

* [Nadam optimization](https://homl.info/nadam)[18](#page27) is more important: it is simply Adam optimization plus the Nesterov trick, so it will often converge slightly faster than Adam. In his report, Timothy Dozat compares many different optimizers on various tasks, and finds that Nadam generally outperforms Adam, but is sometimes outperformed by RMSProp.

Adaptive optimization methods (including RMSProp, Adam and Nadam optimization) are often great, converging fast to a good sol‐ ution. However, a [2017 paper](https://homl.info/60)[19](#page27) by Ashia C. Wilson et al. showed that they can lead to solutions that generalize poorly on some data‐ sets. So when you are disappointed by your model’s performance, try using plain Nesterov Accelerated Gradient instead: your dataset may just be allergic to adaptive gradients. Also check out the latest research, it is moving fast (e.g., AdaBound).



All the optimization techniques discussed so far only rely on the *first-order partial* *derivatives* (*Jacobians*). The optimization literature contains amazing algorithmsbased on the *second-order partial derivatives* (the *Hessians*, which are the partial derivatives of the Jacobians). Unfortunately, these algorithms are very hard to apply to deep neural networks because there are *n*2 Hessians per output (where *n* is the number of parameters), as opposed to just *n* Jacobians per output. Since DNNs typi‐ cally have tens of thousands of parameters, the second-order optimization algorithms



1. “Incorporating Nesterov Momentum into Adam,” Timothy Dozat (2015).
2. “The Marginal Value of Adaptive Gradient Methods in Machine Learning,” A. C. Wilson et al. (2017).



**Faster Optimizers** **|** **351**

often don’t even fit in memory, and even when they do, computing the Hessians is just too slow.



**Training Sparse Models**

All the optimization algorithms just presented produce dense models, meaning that most parameters will be nonzero. If you need a blazingly fast model at runtime, or if you need it to take up less memory, you may prefer to end up with a sparse model instead.

One trivial way to achieve this is to train the model as usual, then get rid of the tiny weights (set them to 0). However, this will typically not lead to a very sparse model, and it may degrade the model’s performance.

A better option is to apply strong ℓ1 regularization during training, as it pushes the optimizer to zero out as many weights as it can (as discussed in Chapter 4 about Lasso Regression).

However, in some cases these techniques may remain insufficient. One last option is to apply *Dual Averaging*, often called *Follow The Regularized Leader* (FTRL), a [techni‐](https://homl.info/61) [que proposed by Yurii Nesterov](https://homl.info/61).[20](#page28) When used with ℓ1 regularization, this technique often leads to very sparse models. Keras implements a variant of FTRL called [*FTRL-Proximal*](https://homl.info/62)[21](#page28)intheFTRLoptimizer.

**Learning Rate Scheduling**

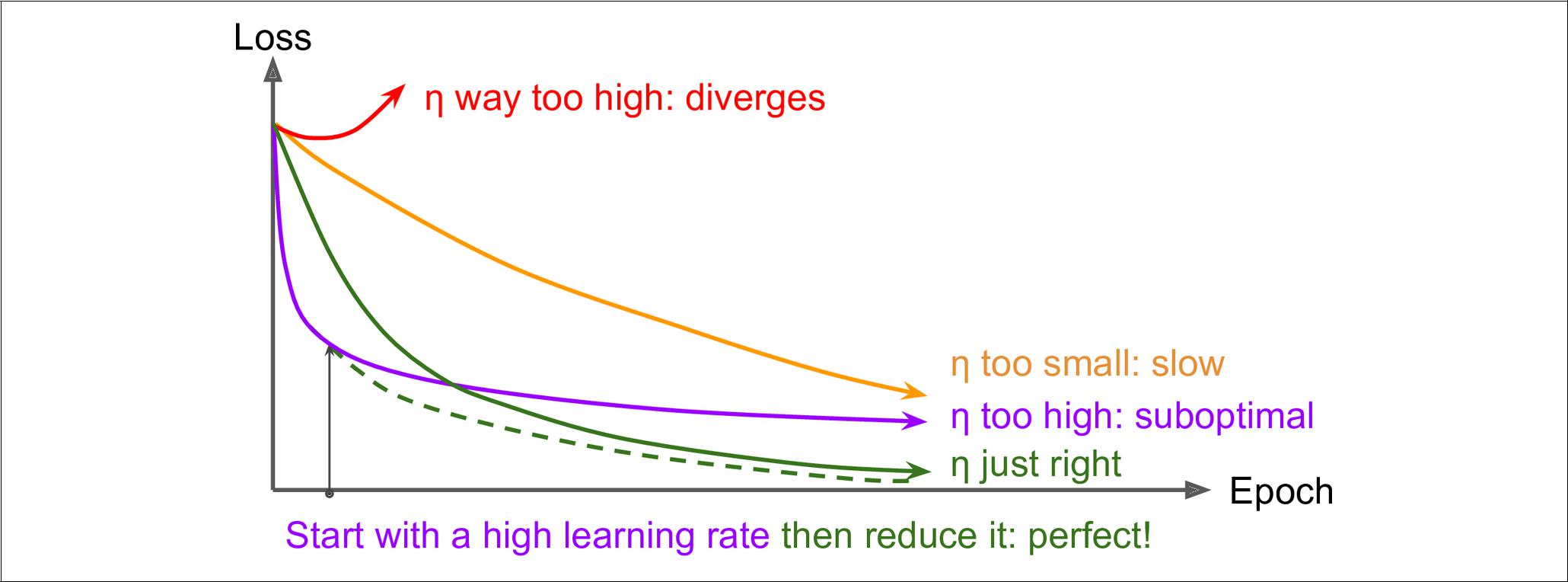
Finding a good learning rate can be tricky. If you set it way too high, training may actually diverge (as we discussed in Chapter 4). If you set it too low, training will eventually converge to the optimum, but it will take a very long time. If you set it slightly too high, it will make progress very quickly at first, but it will end up dancing around the optimum, never really settling down. If you have a limited computing budget, you may have to interrupt training before it has converged properly, yielding a suboptimal solution (see [Figure 11-8](#page29)).



1. “Primal-Dual Subgradient Methods for Convex Problems,” Yurii Nesterov (2005).
2. “Ad Click Prediction: a View from the Trenches,” H. McMahan et al. (2013).



**352** **|** **Chapter 11: Training Deep Neural Networks**



*Figure 11-8. Learning curves for various learning rates η*

As we discussed in Chapter 10, one approach is to start with a large learning rate, and divide it by 3 until the training algorithm stops diverging. You will not be too far from the optimal learning rate, which will learn quickly and converge to good solu‐ tion.

However, you can do better than a constant learning rate: if you start with a high learning rate and then reduce it once it stops making fast progress, you can reach a good solution faster than with the optimal constant learning rate. There are many dif‐ ferent strategies to reduce the learning rate during training. These strategies are called *learning schedules* (we briefly introduced this concept inChapter 4), the most com‐mon of which are:

*Power scheduling*

Set the learning rate to a function of the iteration number *t*: *η*(*t*) = *η*0 / (1 + *t*/*k*)*c*. The initial learning rate *η*0, the power *c* (typically set to 1) and the steps *s* are hyperparameters. The learning rate drops at each step, and after *s* steps it is down to *η*0 / 2. After *s* more steps, it is down to *η*0 / 3. Then down to *η*0 / 4, then *η*0 / 5, and so on. As you can see, this schedule first drops quickly, then more and more slowly. Of course, this requires tuning *η*0, *s* (and possibly *c*).

*Exponential scheduling*

Set the learning rate to: *η*(*t*) = *η*0 0.1*t/s*. The learning rate will gradually drop by a factor of 10 every *s* steps. While power scheduling reduces the learning rate more and more slowly, exponential scheduling keeps slashing it by a factor of *10* every *s* steps.

*Piecewise constant scheduling*

Use a constant learning rate for a number of epochs (e.g., *η*0 = 0.1 for 5 epochs), then a smaller learning rate for another number of epochs (e.g., *η*1 = 0.001 for 50 epochs), and so on. Although this solution can work very well, it requires fid‐



**Faster Optimizers** **|** **353**

dling around to figure out the right sequence of learning rates, and how long to use each of them.

*Performance scheduling*

Measure the validation error every *N* steps (just like for early stopping) and reduce the learning rate by a factor of *λ* when the error stops dropping.

A [2013 paper](https://homl.info/63) [22](#page30) by Andrew Senior et al. compared the performance of some of the most popular learning schedules when training deep neural networks for speech rec‐ ognition using Momentum optimization. The authors concluded that, in this setting, both performance scheduling and exponential scheduling performed well. They favored exponential scheduling because it was easy to tune and it converged slightly faster to the optimal solution (they also mentioned that it was easier to implement than performance scheduling, but in Keras both options are easy).

Implementing power scheduling in Keras is the easiest option: just set the decay hyperparameter when creating an optimizer. The decay is the inverse of *s* (the num‐ ber of steps it takes to divide the learning rate by one more unit), and Keras assumes that *c* is equal to 1. For example:

optimizer = keras.optimizers.SGD(lr=0.01, decay=1e-4)

Exponential scheduling and piecewise scheduling are quite simple too. You first need to define a function that takes the current epoch and returns the learning rate. For example, let’s implement exponential scheduling:

**def** exponential\_decay\_fn(epoch): **return** 0.01\*0.1\*\*(epoch/20)

If you do not want to hard-code *η*0 and *s*, you can create a function that returns a configured function:

**def** exponential\_decay(lr0,s):

**def** exponential\_decay\_fn(epoch): **return** lr0\*0.1\*\*(epoch/s)

**return** exponential\_decay\_fn

exponential\_decay\_fn = exponential\_decay(lr0=0.01, s=20)

Next, just create a LearningRateScheduler callback, giving it the schedule function, and pass this callback to the fit() method:

lr\_scheduler = keras.callbacks.LearningRateScheduler(exponential\_decay\_fn) history = model.fit(X\_train\_scaled, y\_train, [...], callbacks=[lr\_scheduler])



1. “An Empirical Study of Learning Rates in Deep Neural Networks for Speech Recognition,” A. Senior et al. (2013).



**354** **|** **Chapter 11: Training Deep Neural Networks**

The LearningRateScheduler will update the optimizer’s learning\_rate attribute at the beginning of each epoch. Updating the learning rate just once per epoch is usually enough, but if you want it to be updated more often, for example at every step, you need to write your own callback (see the notebook for an example). This can make sense if there are many steps per epoch.

The schedule function can optionally take the current learning rate as a second argu‐ ment. For example, the following schedule function just multiplies the previous learning rate by 0.1&1/20, which results in the same exponential decay (except the decay now starts at the beginning of epoch 0 instead of 1). This implementation relies on the optimizer’s initial learning rate (contrary to the previous implementation), so make sure to set it appropriately.

**def** exponential\_decay\_fn(epoch, lr): **return** lr\*0.1\*\*(1/20)

When you save a model, the optimizer and its learning rate get saved along with it. This means that with this new schedule function, you could just load a trained model and continue training where it left off, no problem. However, things are not so simple if your schedule function uses the epoch argument: indeed, the epoch does not get saved, and it gets reset to 0 every time you call the fit() method. This could lead to a very large learning rate when you continue training a model where it left off, which would likely damage your model’s weights. One solution is to manually set the fit() method’s initial\_epoch argument so the epoch starts at the right value.

For piecewise constant scheduling, you can use a schedule function like the following one (as earlier, you can define a more general function if you want, see the notebook for an example), then create a LearningRateScheduler callback with this function and pass it to the fit() method, just like we did for exponential scheduling:

**def** piecewise\_constant\_fn(epoch): **if** epoch<5:

**return** 0.01 **elif** epoch<15: **return** 0.005

**else**:

**return** 0.001

For performance scheduling, simply use the ReduceLROnPlateau callback. For exam‐ ple, if you pass the following callback to the fit() method, it will multiply the learn‐ ing rate by 0.5 whenever the best validation loss does not improve for 5 consecutive epochs (other options are available, please check the documentation for more details):

lr\_scheduler = keras.callbacks.ReduceLROnPlateau(factor=0.5, patience=5)

Lastly, tf.keras offers an alternative way to implement learning rate scheduling: just define the learning rate using one of the schedules available in keras.optimiz



**Faster Optimizers** **|** **355**

ers.schedules, then pass this learning rate to any optimizer. This approach updates the learning rate at each step rather than at each epoch. For example, here is how to implement the same exponential schedule as earlier:

s = 20 \* len(X\_train) // 32 *# number of steps in 20 epochs (batch size = 32)* learning\_rate = keras.optimizers.schedules.ExponentialDecay(0.01, s, 0.1) optimizer = keras.optimizers.SGD(learning\_rate)

This is nice and simple, plus when you save the model, the learning rate and its schedule (including its state) get saved as well. However, this approach is not part of the Keras API, it is specific to tf.keras.

To sum up, exponential decay or performance scheduling can considerably speed up convergence, so give them a try!

**Avoiding Over€tting Through Regularization**

With four parameters I can fit an elephant and with five I can make him wiggle his trunk.

—John von Neumann, *cited by Enrico Fermi in Nature 427*

With thousands of parameters you can fit the whole zoo. Deep neural networks typi‐ cally have tens of thousands of parameters, sometimes even millions. With so many parameters, the network has an incredible amount of freedom and can fit a huge vari‐ ety of complex datasets. But this great flexibility also means that it is prone to overfit‐ ting the training set. We need regularization.

We already implemented one of the best regularization techniques in Chapter 10: early stopping. Moreover, even though Batch Normalization was designed to solve the vanishing/exploding gradients problems, is also acts like a pretty good regularizer. In this section we will present other popular regularization techniques for neural net‐ works: ℓ1 and ℓ2 regularization, dropout and max-norm regularization.

**ℓ1 and ℓ2 Regularization**

Just like you did in Chapter 4 for simple linear models, you can use ℓ1 and ℓ2 regulari‐ zation to constrain a neural network’s connection weights (but typically not its bia‐ ses). Here is how to apply ℓ2 regularization to a Keras layer’s connection weights, using a regularization factor of 0.01:

layer = keras.layers.Dense(100, activation="elu", kernel\_initializer="he\_normal", kernel\_regularizer=keras.regularizers.l2(0.01))

The l2() function returns a regularizer that will be called to compute the regulariza‐ tion loss, at each step during training. This regularization loss is then added to the final loss. As you might expect, you can just use keras.regularizers.l1() if you



**356** **|** **Chapter 11: Training Deep Neural Networks**

want ℓ1 regularization, and if you want both ℓ1 and ℓ2 regularization, use keras.regu larizers.l1\_l2() (specifying both regularization factors).

Since you will typically want to apply the same regularizer to all layers in your net‐ work, as well as the same activation function and the same initialization strategy in all hidden layers, you may find yourself repeating the same arguments over and over. This makes it ugly and error-prone. To avoid this, you can try refactoring your code to use loops. Another option is to use Python’s functools.partial() function: it lets you create a thin wrapper for any callable, with some default argument values. For example:

**from functools import** partial

RegularizedDense = partial(keras.layers.Dense, activation="elu", kernel\_initializer="he\_normal",

kernel\_regularizer=keras.regularizers.l2(0.01))

model = keras.models.Sequential([ keras.layers.Flatten(input\_shape=[28, 28]), RegularizedDense(300), RegularizedDense(100), RegularizedDense(10, activation="softmax",

kernel\_initializer="glorot\_uniform")

])

**Dropout**

*Dropout* is one of the most popular regularization techniques for deep neural net‐works. It was [proposed](https://homl.info/64)[23](#page33) by Geoffrey Hinton in 2012 and further detailed in a [paper](https://homl.info/65)[24](#page33) by Nitish Srivastava et al., and it has proven to be highly successful: even the state-of-the-art neural networks got a 1–2% accuracy boost simply by adding dropout. This may not sound like a lot, but when a model already has 95% accuracy, getting a 2% accuracy boost means dropping the error rate by almost 40% (going from 5% error to roughly 3%).

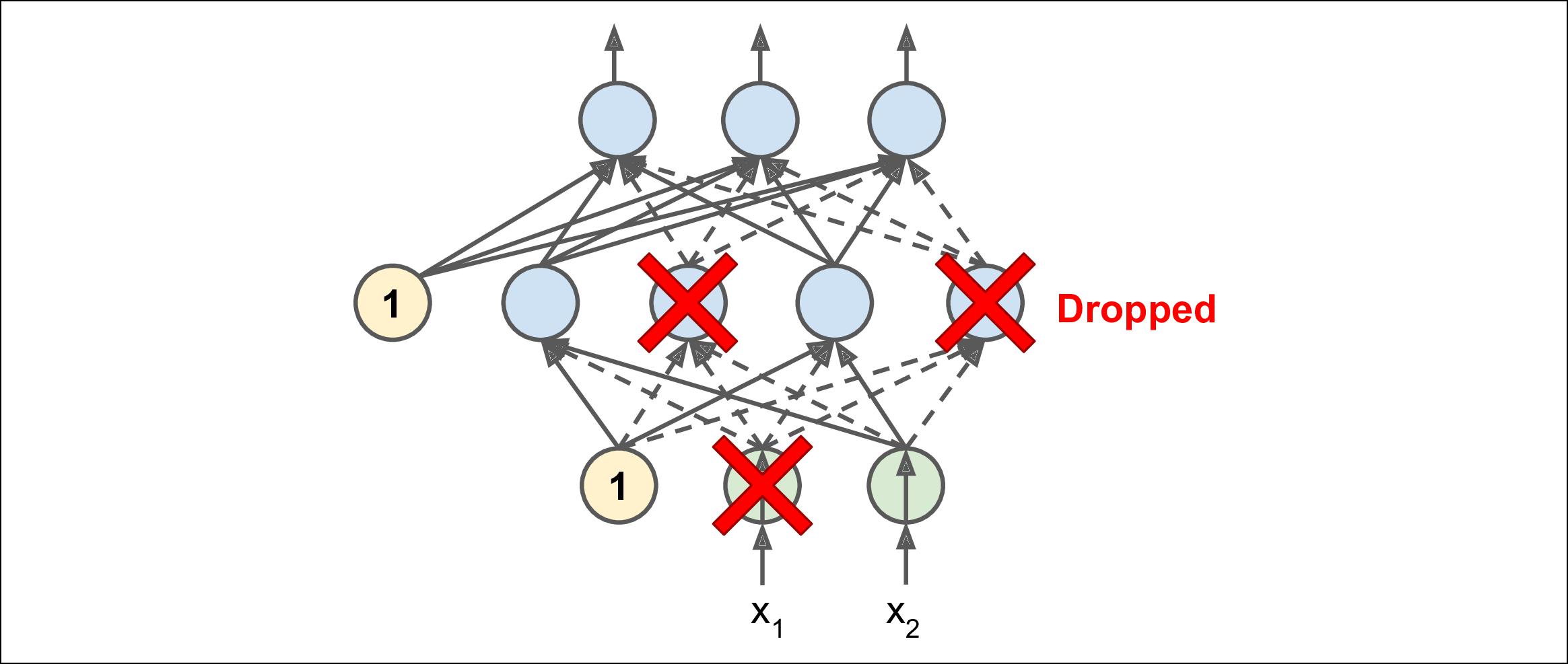
It is a fairly simple algorithm: at every training step, every neuron (including the input neurons, but always excluding the output neurons) has a probability *p* of being temporarily “dropped out,” meaning it will be entirely ignored during this training step, but it may be active during the next step (see [Figure 11-9](#page34)). The hyperparameter *p* is called the *dropout rate*, and it is typically set to 50%. After training, neurons don’tget dropped anymore. And that’s all (except for a technical detail we will discuss momentarily).



1. “Improving neural networks by preventing co-adaptation of feature detectors,” G. Hinton et al. (2012).
2. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting,” N. Srivastava et al. (2014).



**Avoiding Over€tting Through Regularization** **|** **357**



*Figure 11-9. Dropout regularization*

It is quite surprising at first that this rather brutal technique works at all. Would a company perform better if its employees were told to toss a coin every morning to decide whether or not to go to work? Well, who knows; perhaps it would! The com‐ pany would obviously be forced to adapt its organization; it could not rely on any sin‐ gle person to fill in the coffee machine or perform any other critical tasks, so this expertise would have to be spread across several people. Employees would have to learn to cooperate with many of their coworkers, not just a handful of them. The company would become much more resilient. If one person quit, it wouldn’t make much of a difference. It’s unclear whether this idea would actually work for compa‐ nies, but it certainly does for neural networks. Neurons trained with dropout cannot co-adapt with their neighboring neurons; they have to be as useful as possible on their own. They also cannot rely excessively on just a few input neurons; they must pay attention to each of their input neurons. They end up being less sensitive to slight changes in the inputs. In the end you get a more robust network that generalizes bet‐ ter.

Another way to understand the power of dropout is to realize that a unique neural network is generated at each training step. Since each neuron can be either present or absent, there is a total of 2*N* possible networks (where *N* is the total number of drop‐ pable neurons). This is such a huge number that it is virtually impossible for the same neural network to be sampled twice. Once you have run a 10,000 training steps, you have essentially trained 10,000 different neural networks (each with just one training instance). These neural networks are obviously not independent since they share many of their weights, but they are nevertheless all different. The resulting neural network can be seen as an averaging ensemble of all these smaller neural networks.

There is one small but important technical detail. Suppose *p* = 50%, in which case during testing a neuron will be connected to twice as many input neurons as it was (on average) during training. To compensate for this fact, we need to multiply each



**358** **|** **Chapter 11: Training Deep Neural Networks**

neuron’s input connection weights by 0.5 after training. If we don’t, each neuron will get a total input signal roughly twice as large as what the network was trained on, and it is unlikely to perform well. More generally, we need to multiply each input connec‐ tion weight by the *keep probability* (1 – *p*) after training. Alternatively, we can divide each neuron’s output by the keep probability during training (these alternatives are not perfectly equivalent, but they work equally well).

To implement dropout using Keras, you can use the keras.layers.Dropout layer. During training, it randomly drops some inputs (setting them to 0) and divides the remaining inputs by the keep probability. After training, it does nothing at all, it just passes the inputs to the next layer. For example, the following code applies dropout regularization before every Dense layer, using a dropout rate of 0.2:

model = keras.models.Sequential([ keras.layers.Flatten(input\_shape=[28, 28]), keras.layers.Dropout(rate=0.2),

keras.layers.Dense(300, activation="elu", kernel\_initializer="he\_normal"), keras.layers.Dropout(rate=0.2),

keras.layers.Dense(100, activation="elu", kernel\_initializer="he\_normal"), keras.layers.Dropout(rate=0.2),

keras.layers.Dense(10, activation="softmax")

])

Since dropout is only active during training, the training loss is penalized compared to the validation loss, so comparing the two can be misleading. In particular, a model may be overfitting the training set and yet have similar training and validation losses. So make sure to evaluate the training loss without dropout (e.g., after training). Alternatively, you can call the fit() method inside a

with keras.backend.learning\_phase\_scope(1) block: this will force dropout to be active during both training and validation.[25](#page35)

If you observe that the model is overfitting, you can increase the dropout rate. Con‐ versely, you should try decreasing the dropout rate if the model underfits the training set. It can also help to increase the dropout rate for large layers, and reduce it for small ones. Moreover, many state-of-the-art architectures only use dropout after the last hidden layer, so you may want to try this if full dropout is too strong.

Dropout does tend to significantly slow down convergence, but it usually results in a much better model when tuned properly. So, it is generally well worth the extra time and effort.



1. This is specific to tf.keras, so you may prefer to use keras.backend.set\_learning\_phase(1) before calling the fit() method (and set it back to 0 right after).



**Avoiding Over€tting Through Regularization** **|** **359**

If you want to regularize a self-normalizing network based on the SELU activation function (as discussed earlier), you should use AlphaDropout: this is a variant of dropout that preserves the mean and standard deviation of its inputs (it was introduced in the same paper as SELU, as regular dropout would break self-normalization).

**Monte-Carlo (MC) Dropout**

In 2016, a [paper](https://homl.info/mcdropout)[26](#page36) by Yarin Gal and Zoubin Ghahramani added more good reasons to use dropout:

* First, the paper establishes a profound connection between dropout networks (i.e., neural networks containing a dropout layer before every weight layer) and approximate Bayesian inference[27](#page36), giving dropout a solid mathematical justifica‐ tion.
* Second, they introduce a powerful technique called *MC Dropout*, which can boost the performance of any trained dropout model, without having to retrain it or even modify it at all!
* Moreover, MC Dropout also provides a much better measure of the model’s uncertainty.
* Finally, it is also amazingly simple to implement. If this all sounds like a “one weird trick” advertisement, then take a look at the following code. It is the full implementation of *MC Dropout*, boosting the dropout model we trained earlier, without retraining it:

**with** keras.backend.learning\_phase\_scope(1):*# force training mode = dropout on*y\_probas = np.stack([model.predict(X\_test\_scaled)

**for** sample **in** range(100)])y\_proba = y\_probas.mean(axis=0)

We first force training mode on, using a learning\_phase\_scope(1) context. This turns dropout on within the with block. Then we make 100 predictions over the test set, and we stack them. Since dropout is on, all predictions will be different. Recall that predict() returns a matrix with one row per instance, and one column per class. Since there are 10,000 instances in the test set, and 10 classes, this is a matrix of shape [10000, 10]. We stack 100 such matrices, so y\_probas is an array of shape [100, 10000, 10]. Once we average over the first dimension (axis=0), we get y\_proba, an array of shape [10000, 10], like we would get with a single prediction. That’s all! Averaging



1. “Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning,” Y. Gal and Z. Ghahramani (2016).
2. Specifically, they show that training a dropout network is mathematically equivalent to approximate Bayesian inference in a specific type of probabilistic model called a *deep Gaussian Process*.



**360** **|** **Chapter 11: Training Deep Neural Networks**

over multiple predictions with dropout on gives us a Monte Carlo estimate that is generally more reliable than the result of a single prediction with dropout off. For example, let’s look at the model’s prediction for the first instance in the test set, with dropout off:

**>>>** np.round(model.predict(X\_test\_scaled[:1]),2)

array([[0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.01, 0. , 0.99]], dtype=float32)

The model seems almost certain that this image belongs to class 9 (ankle boot). Should you trust it? Is there really so little room for doubt? Compare this with the predictions made when dropout is activated:

**>>>** np.round(y\_probas[:, :1],2)

array([[[0. , 0. , 0. , 0. , 0. , 0.14, 0. , 0.17, 0. , 0.68]], [[0. , 0. , 0. , 0. , 0. , 0.16, 0. , 0.2 , 0. , 0.64]], [[0. , 0. , 0. , 0. , 0. , 0.02, 0. , 0.01, 0. , 0.97]], [...]

This tells a very different story: apparently, when we activate dropout, the model is not sure anymore. It still seems to prefer class 9, but sometimes it hesitates with classes 5 (sandal) and 7 (sneaker), which makes sense given they’re all footwear. Once we average over the first dimension, we get the following MC dropout predictions:

**>>>** np.round(y\_proba[:1],2)

array([[0. , 0. , 0. , 0. , 0. , 0.22, 0. , 0.16, 0. , 0.62]], dtype=float32)

The model still thinks this image belongs to class 9, but only with a 62% confidence, which seems much more reasonable than 99%. Plus it’s useful to know exactly which other classes it thinks are likely. And you can also take a look at the [standard devia‐](https://xkcd.com/2110) [tion of the probability estimates](https://xkcd.com/2110):

* y\_std = y\_probas.std(axis=0)
* np.round(y\_std[:1], 2)

array([[0. , 0. , 0. , 0. , 0. , 0.28, 0. , 0.21, 0.02, 0.32]], dtype=float32)

Apparently there’s quite a lot of variance in the probability estimates: if you were building a risk-sensitive system (e.g., a medical or financial system), you should prob‐ ably treat such an uncertain prediction with extreme caution. You definitely would not treat it like a 99% confident prediction. Moreover, the model’s accuracy got a small boost from 86.8 to 86.9:

* accuracy = np.sum(y\_pred == y\_test) / len(y\_test)
* accuracy

0.8694



**Avoiding Over€tting Through Regularization** **|** **361**

The number of Monte Carlo samples you use (100 in this example) is a hyperparameter you can tweak. The higher it is, the more accu‐ rate the predictions and their uncertainty estimates will be. How‐ ever, it you double it, inference time will also be doubled. Moreover, above a certain number of samples, you will notice little improvement. So your job is to find the right tradeoff between latency and accuracy, depending on your application.



If your model contains other layers that behave in a special way during training (such as Batch Normalization layers), then you should not force training mode like we just did. Instead, you should replace the Dropout layers with the following MCDropout class:

**class MCDropout**(keras.layers.Dropout): **def** call(self,inputs):

**return** super().call(inputs,training=True)

We just sublass the Dropout layer and override the call() method to force its train ing argument to True (see Chapter 12). Similarly, you could define an MCAlphaDrop out class by subclassing AlphaDropout instead. If you are creating a model from scratch, it’s just a matter of using MCDropout rather than Dropout. But if you have a model that was already trained using Dropout, you need to create a new model, iden‐ tical to the existing model except replacing the Dropout layers with MCDropout, then copy the existing model’s weights to your new model.

In short, MC Dropout is a fantastic technique that boosts dropout models and pro‐ vides better uncertainty estimates. And of course, since it is just regular dropout dur‐ ing training, it also acts like a regularizer.

**Max-Norm Regularization**

Another regularization technique that is quite popular for neural networks is called *max-norm regularization*: for each neuron, it constrains the weights**w**of the incom‐ing connections such that ∥ \*w\* ∥2 ≤ \_r\_, where *r* is the max-norm hyperparameter and ∥ · ∥2 is the ℓ2 norm.

Max-norm regularization does not add a regularization loss term to the overall loss

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function. Instead, it is typically implemented by computing ∥**w**∥2 after each training | | | |  |
| step and clipping **w** if needed (**w w** | *r* |  | ). |  |
| **w** | 2 |  |
|  |  |  |



Reducing *r* increases the amount of regularization and helps reduce overfitting. Max-norm regularization can also help alleviate the vanishing/exploding gradients prob‐ lems (if you are not using Batch Normalization).



**362** **|** **Chapter 11: Training Deep Neural Networks**

To implement max-norm regularization in Keras, just set every hidden layer’s ker nel\_constraint argument to a max\_norm() constraint, with the appropriate max value, for example:

keras.layers.Dense(100, activation="elu", kernel\_initializer="he\_normal", kernel\_constraint=keras.constraints.max\_norm(1.))

After each training iteration, the model’s fit() method will call the object returned by max\_norm(), passing it the layer’s weights and getting clipped weights in return, which then replace the layer’s weights. As we will see in Chapter 12, you can define your own custom constraint function if you ever need to, and use it as the ker nel\_constraint. You can also constrain the bias terms by setting the bias\_con straint argument.

The max\_norm() function has an axis argument that defaults to 0. A Dense layer usu‐ ally has weights of shape [number of inputs, number of neurons], so using axis=0 means that the max norm constraint will apply independently to each neuron’s weight vector. If you want to use max-norm with convolutional layers (see Chapter 14), make sure to set the max\_norm() constraint’s axis argument appropriately (usually

axis=[0, 1, 2]).

**Summary and Practical Guidelines**

In this chapter, we have covered a wide range of techniques and you may be wonder‐ ing which ones you should use. The configuration in [Table 11-2](#page39) will work fine in most cases, without requiring much hyperparameter tuning.

*Table 11-2. Default DNN configuration*

|  |  |
| --- | --- |
| **Hyperparameter** | **Default value** |
| Kernel initializer: | LeCun initialization |
| Activation function: | SELU |
| Normalization: | None (self-normalization) |
| Regularization: | Early stopping |
| Optimizer: | Nadam |
| Learning rate schedule: | Performance scheduling |
|  |  |

Don’t forget to standardize the input features! Of course, you should also try to reuse parts of a pretrained neural network if you can find one that solves a similar problem, or use unsupervised pretraining if you have a lot of unlabeled data, or pretraining on an auxiliary task if you have a lot of labeled data for a similar task.

The default configuration in [Table 11-2](#page39) may need to be tweaked:



**Summary and Practical Guidelines** **|** **363**

* If your model self-normalizes:

— If it overfits the training set, then you should add alpha dropout (and always use early stopping as well). Do not use other regularization methods, or else they would break self-normalization.

* If your model cannot self-normalize (e.g., it is a recurrent net or it contains skip connections):

— You can try using ELU (or another activation function) instead of SELU, it may perform better. Make sure to change the initialization method accord‐ ingly (e.g., He init for ELU or ReLU).

— If it is a deep network, you should use Batch Normalization after every hidden layer. If it overfits the training set, you can also try using max-norm or ℓ2 reg‐ ularization.

* If you need a sparse model, you can use ℓ1 regularization (and optionally zero out the tiny weights after training). If you need an even sparser model, you can try using FTRL instead of Nadam optimization, along with ℓ1 regularization. In any case, this will break self-normalization, so you will need to switch to BN if your model is deep.
* If you need a low-latency model (one that performs lightning-fast predictions), you may need to use less layers, avoid Batch Normalization, and possibly replace the SELU activation function with the leaky ReLU. Having a sparse model will also help. You may also want to reduce the float precision from 32-bits to 16-bit (or even 8-bits) (see ???).
* If you are building a risk-sensitive application, or inference latency is not very important in your application, you can use MC Dropout to boost performance and get more reliable probability estimates, along with uncertainty estimates.

With these guidelines, you are now ready to train very deep nets! I hope you are now convinced that you can go a very long way using just Keras. However, there may come a time when you need to have even more control, for example to write a custom loss function or to tweak the training algorithm. For such cases, you will need to use TensorFlow’s lower-level API, as we will see in the next chapter.

**Exercises**

1. Is it okay to initialize all the weights to the same value as long as that value is selected randomly using He initialization?
2. Is it okay to initialize the bias terms to 0?
3. Name three advantages of the SELU activation function over ReLU.



**364** **|** **Chapter 11: Training Deep Neural Networks**

1. In which cases would you want to use each of the following activation functions: SELU, leaky ReLU (and its variants), ReLU, tanh, logistic, and softmax?
2. What may happen if you set the momentum hyperparameter too close to 1 (e.g., 0.99999) when using an SGD optimizer?
3. Name three ways you can produce a sparse model.
4. Does dropout slow down training? Does it slow down inference (i.e., making predictions on new instances)? What are about MC dropout?
5. Deep Learning.
   1. Build a DNN with five hidden layers of 100 neurons each, He initialization, and the ELU activation function.
   2. Using Adam optimization and early stopping, try training it on MNIST but only on digits 0 to 4, as we will use transfer learning for digits 5 to 9 in the next exercise. You will need a softmax output layer with five neurons, and as always make sure to save checkpoints at regular intervals and save the final model so you can reuse it later.
   3. Tune the hyperparameters using cross-validation and see what precision you can achieve.
   4. Now try adding Batch Normalization and compare the learning curves: is it converging faster than before? Does it produce a better model?
   5. Is the model overfitting the training set? Try adding dropout to every layer and try again. Does it help?
6. Transfer learning.
   1. Create a new DNN that reuses all the pretrained hidden layers of the previous model, freezes them, and replaces the softmax output layer with a new one.
   2. Train this new DNN on digits 5 to 9, using only 100 images per digit, and time how long it takes. Despite this small number of examples, can you achieve high precision?
   3. Try caching the frozen layers, and train the model again: how much faster is it now?
   4. Try again reusing just four hidden layers instead of five. Can you achieve a higher precision?
   5. Now unfreeze the top two hidden layers and continue training: can you get the model to perform even better?
7. Pretraining on an auxiliary task.
   1. In this exercise you will build a DNN that compares two MNIST digit images and predicts whether they represent the same digit or not. Then you will reuse the lower layers of this network to train an MNIST classifier using very little



**Exercises** **|** **365**

training data. Start by building two DNNs (let’s call them DNN A and B), both similar to the one you built earlier but without the output layer: each DNN should have five hidden layers of 100 neurons each, He initialization, and ELU activation. Next, add one more hidden layer with 10 units on top of both DNNs. To do this, you should use a keras.layers.Concatenate layer to con‐ catenate the outputs of both DNNs for each instance, then feed the result to the hidden layer. Finally, add an output layer with a single neuron using the logistic activation function.

1. Split the MNIST training set in two sets: split #1 should containing 55,000 images, and split #2 should contain contain 5,000 images. Create a function that generates a training batch where each instance is a pair of MNIST images picked from split #1. Half of the training instances should be pairs of images that belong to the same class, while the other half should be images from dif‐ ferent classes. For each pair, the training label should be 0 if the images are from the same class, or 1 if they are from different classes.
2. Train the DNN on this training set. For each image pair, you can simultane‐ ously feed the first image to DNN A and the second image to DNN B. The whole network will gradually learn to tell whether two images belong to the same class or not.
3. Now create a new DNN by reusing and freezing the hidden layers of DNN A and adding a softmax output layer on top with 10 neurons. Train this network on split #2 and see if you can achieve high performance despite having only 500 images per class.

Solutions to these exercises are available in ???.



**366** **|** **Chapter 11: Training Deep Neural Networks**