Chapter 11. Training Deep Neural Networks

In Chapter 10 you built, trained, and fine-tuned several artificial neural networks using PyTorch. But they were shallow nets with just a few hidden layers. What if you need to tackle a complex problem, such as detecting hundreds of types of objects in high-resolution images? You may need to train a much deeper ANN, perhaps with dozens or even hundreds of layers, each containing hundreds of neurons, linked by hundreds of thousands of connections. Training a deep neural network isn't a walk in the park. Here are some of the problems you could run into:

- You may be faced with the problem of gradients growing ever smaller or larger when flowing backward through the DNN during training. Both of these problems make lower layers very hard to train.
- You might not have enough training data for such a large network, or it might be too costly to label.
- Training may be extremely slow.
- A model with millions of parameters risks severely overfitting the training set, especially if there are not enough training instances or if they are too noisy.

In this chapter we will go through each of these problems and present various techniques to solve them. We will start by exploring the vanishing and exploding gradients problems and some of their most popular solutions, including smart weight initialization, better activation functions, batch-norm, layer-norm, and gradient clipping. Next, we will look at transfer learning and unsupervised pretraining, which can help you tackle complex tasks even when you have little labeled data. Then we will discuss a variety of optimizers that can speed up training large models tremendously. We will also discuss how you can tweak the learning rate during training to speed up training and produce better models. Finally, we will cover a few popular regularization techniques for large neural networks: ℓ_1 and ℓ_2 regularization, dropout, Monte Carlo dropout, and max-norm regularization.

With these tools, you will be able to train all sorts of deep nets. Welcome to *deep* learning!

The Vanishing/Exploding Gradients Problems

As discussed in Chapter 9, the backpropagation algorithm's second phase works by going from the output layer to the input layer, propagating the error gradient along the way. Once the algorithm has computed the gradient of the cost function with regard 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 layers' 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 until layers get insanely large weight updates and the algorithm diverges. This is the *exploding gradients* problem, which surfaces most often in recurrent neural networks (see Chapter 13). More generally, deep neural networks suffer from unstable gradients; different layers may learn at widely different speeds.

This unfortunate behavior was empirically observed long ago, and it was one of the reasons deep neural networks were mostly abandoned in the early 2000s. It wasn't clear what caused the gradients to be so unstable when training a DNN, but some light was shed in a 2010 paper by Xavier Glorot and Yoshua Bengio.[1] The authors found a few suspects, including the combination of the popular sigmoid (logistic) activation function and the weight initialization technique that was most popular at the time (i.e., 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 saturation is actually made worse by the fact that the sigmoid function has a mean of 0.5, not 0 (the hyperbolic tangent function has a mean of 0 and behaves slightly better than the sigmoid function in deep networks).

Looking at the sigmoid activation function (see Figure 11-1), 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 (i.e., the curve is flat at both extremes). 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.

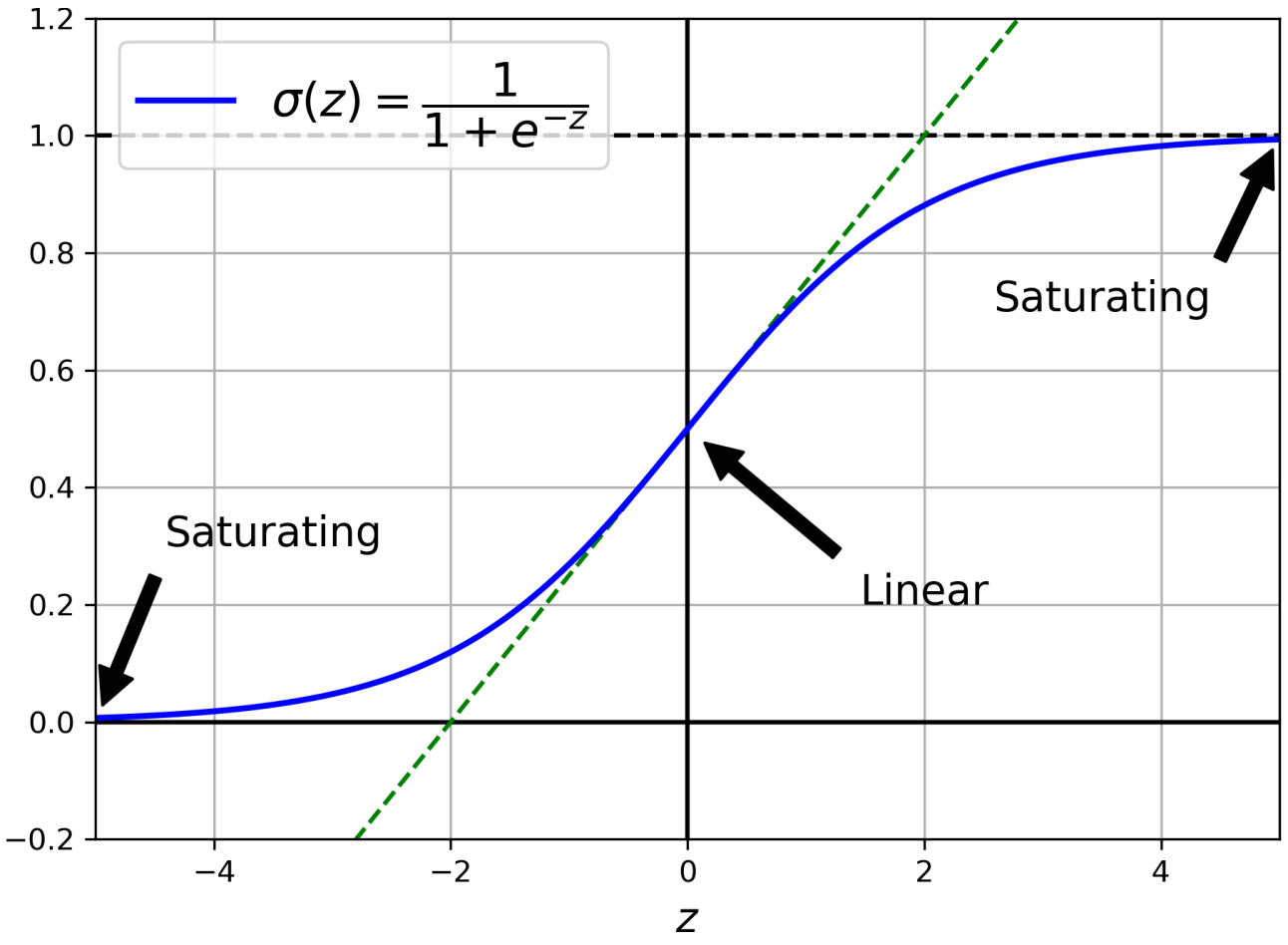

Figure 11-1. Sigmoid activation function saturation

Glorot Initialization and He Initialization

In their paper, Glorot and Bengio proposed a way to significantly alleviate the unstable gradients problem. They pointed out that we need the signal to flow properly in both directions: in the forward direction when making predictions, and in the reverse direction when backpropagating gradients. 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 argued that we need the variance of the outputs of each layer to be equal to the variance of its inputs,[2] and we 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 outputs (these numbers are called the *fan-in* and *fan-out* of the layer), but Glorot and Bengio proposed a good compromise that has proven to work very well in practice: the connection weights of each layer must be initialized randomly, as described in Equation 11-1, where $fan_{\text{avg}} = (fan_{\text{in}} + fan_{\text{out}}) / 2$. This initialization strategy is called *Xavier initialization* or *Glorot initialization*, after the paper's first author.

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

$$\textbf{Normal distribution with mean 0 and variance } \sigma^2 = \frac{1}{fan_{\text{avg}}}$$

$$\textbf{Or a uniform distribution between } -r \textbf{ and } +r\textbf{, with } r = \sqrt{\frac{3}{fan_{\text{avg}}}}$$

If you replace fan_{avg} with fan_{in} in Equation 11-1, you get an initialization strategy that Yann LeCun proposed in the 1990s. He called it *LeCun initialization*. Genevieve Orr and Klaus-Robert Müller even recommended it in their 1998 book *Neural Networks: Tricks of the Trade* (Springer). LeCun initialization is equivalent to Glorot initialization when $fan_{\text{in}} = fan_{\text{out}}$. It took over a decade for researchers to realize how important this trick is. Using Glorot initialization can speed up training considerably, and it is one of the tricks that led to the success of deep learning.

Some papers have provided similar strategies for different activation functions, most notably a 2015 paper by Kaiming He et al.[3] 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 (for the uniform distribution, just use $r = \sqrt{3\sigma^2}$). The initialization strategy proposed for the ReLU activation function and its variants is called *He initialization* or *Kaiming initialization*, after the paper's first author. For SELU, use Yann LeCun's initialization method, preferably with a normal distribution. We will cover all these activation functions shortly.

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

Initialization	Activation functions	σ^2 (Normal)
Xavier Glorot	None, tanh, sigmoid, softmax	$1 / fan_{\text{avg}}$
Kaiming He	ReLU, Leaky ReLU, ELU, GELU, Swish, Mish, SwiGLU, ReLU[2]	$2 / fan_{\text{in}}$

For historical reasons, PyTorch's `nn.Linear` module initializes its weights using Kaiming uniform initialization, except the weights are scaled down by a factor of (and the bias terms are also initialized randomly). Sadly, this is not the optimal scale for any common activation function.[4] One solution is to simply multiply the weights by (i.e., $6^{0.5}$) just after creating the `nn.Linear` layer to get proper Kaiming initialization. To do this, we can update the parameter's `data` attribute. We will also zero out the biases, as there's no benefit in randomly initializing them:

```
import torch
import torch.nn as nn

layer = nn.Linear(40, 10)
layer.weight.data *= 6 ** 0.5  # Kaiming init (or 3 ** 0.5 for LeCun init)
torch.zero_(layer.bias.data)
```

$\sqrt{6}$

This works, but it's clearer and less error-prone to use one of the initialization functions available in the `torch.nn.init` module:

$\sqrt{6}$

```
nn.init.kaiming_uniform_(layer.weight)
nn.init.zeros_(layer.bias)
```

If you want to apply the same initialization method to the weights of every `nn.Linear` layer in a model, you can do so in the model's constructor, after creating each `nn.Linear` layer. Alternatively, you can write a subclass of the `nn.Linear` class and tweak its constructor to initialize the weights as you wish. But arguably the simplest option is to write a little function that takes a module, checks whether it's an instance of the `nn.Linear` class, and if so, applies the desired initialization function to its weights. You can then apply this function to the model and all of its submodules by passing it to the model's `apply()` method. For example:

```
def use_he_init(module):
    if isinstance(module, nn.Linear):
        nn.init.kaiming_uniform_(module.weight)
        nn.init.zeros_(module.bias)

model = nn.Sequential(nn.Linear(50, 40), nn.ReLU(), nn.Linear(40, 1), nn.ReLU())
model.apply(use_he_init)
```

TIP

In a classifier, it's generally a good idea to scale down the weights of the output layer during initialization (e.g., by a factor of 10). Indeed, this will result in smaller logits at the beginning of training, which means they will be closer together, and hence the estimated probabilities will also be closer together. In other words, it encourages the model to be less confident about its predictions when training starts: this will avoid extreme losses and huge gradients that can often make the model's weights bounce around ran

avoid extreme losses and huge gradients that can often make the model's weights bounce around randomly at the start of training, losing time and potentially preventing the model from learning anything.

The `torch.nn.init` module also contains an `orthogonal_()` function which initializes the weights using a random orthogonal matrix, as proposed in a 2014 paper by Andrew Saxe et al.[5] Orthogonal matrices have a number of useful mathematical properties, including the fact that they preserve norms: given an orthogonal matrix **W** and an input vector **x**, the norm of **Wx** is equal to the norm of **x**, and therefore the magnitude of the inputs is preserved in the outputs. When the inputs are standardized, this results in a stable variance through the layer, which prevents the activations and gradients from vanishing or exploding in a deep network (at least at the beginning of training). This initialization technique is much less common than the initialization techniques discussed earlier, but it can work well in recurrent neural nets (Chapter 13) or generative adversarial networks (Chapter 18).

And that's it! Scaling the weights properly will give a deep neural net a much better starting point for training.

Better Activation Functions

One of the insights in the 2010 paper by Glorot and Bengio was that the problems with unstable gradients were in part due to a poor choice of activation function. Until then most people had assumed that if Mother Nature had chosen to use something pretty close to 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 very 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 neuron dies when its weights get tweaked in such a way that the input of the ReLU function (i.e., the weighted sum of the neuron's inputs plus its bias term) is negative for all instances in the training set. When this happens, it just keeps outputting zeros, and gradient descent does not affect it anymore because the gradient of the ReLU function is zero when its input is negative.[6]

To solve this problem, you may want to use a variant of the ReLU function, such as the *leaky ReLU*.

Leaky ReLU

The leaky ReLU activation function is defined as $\text{LeakyReLU}_{\alpha}(z) = \max(\alpha z, z)$ (see Figure 11-2). The hyperparameter α defines how much the function "leaks": it is the slope of the function for $z < 0$. Having a slope for $z < 0$ ensures that leaky ReLUs never actually die; they can go into a long coma, but they have a chance to eventually wake up. A 2015 paper by Bing Xu et al.[7] com-

pared 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 $\alpha = 0.2$ (a huge leak) seemed to result in better performance than $\alpha = 0.01$ (a small leak). The paper also evaluated the *randomized leaky ReLU* (RReLU), where α is picked randomly in a given range during training and is fixed to an average value during testing. RReLU also performed fairly well and seemed to act as a regularizer, reducing the risk of overfitting. Finally, the paper 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. PReLU was reported to strongly outperform ReLU on large image datasets, but on smaller datasets it runs the risk of overfitting the training set.

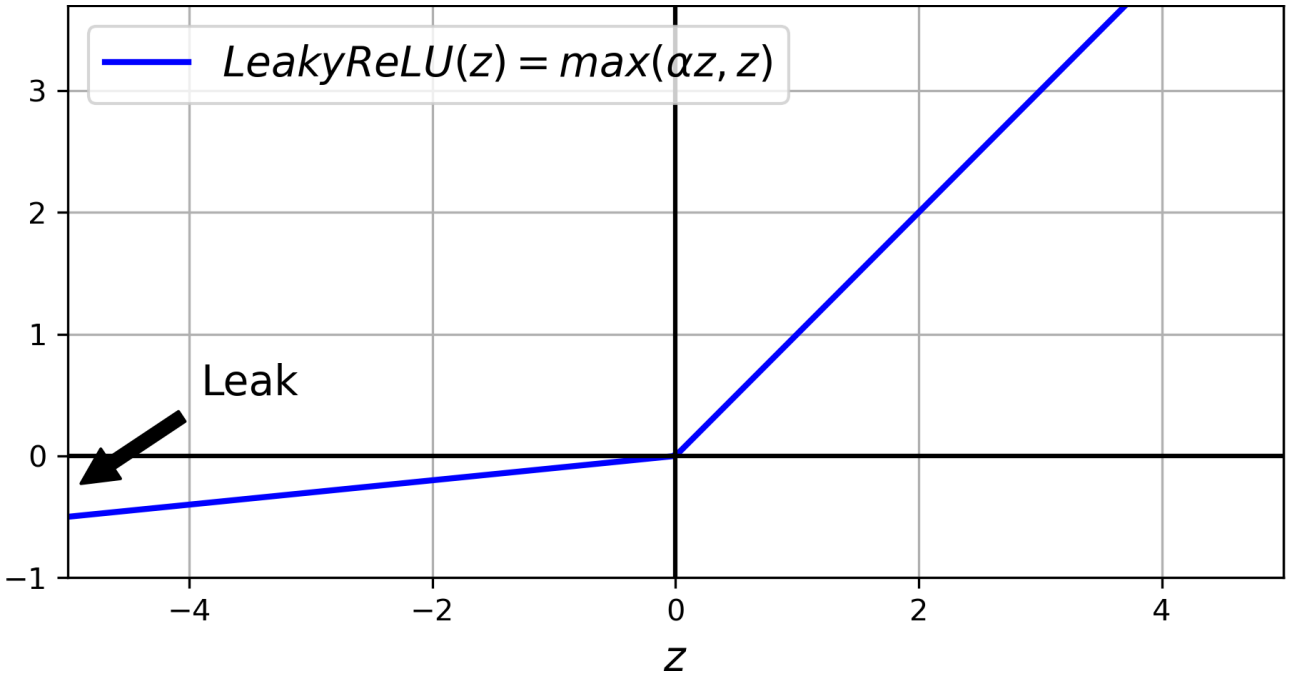

Figure 11-2. Leaky ReLU: like ReLU, but with a small slope for negative values

As you might expect, PyTorch includes modules for each of these activation functions: `nn.LeakyReLU`, `nn.RReLU`, and `nn.PReLU`. Just like for other ReLU variants, you should use these along with Kaiming initialization, but the variance should be slightly smaller due to the negative slope: it should be scaled down by a factor of $1 + \alpha^2$. PyTorch supports this: you can pass the α hyperparameter to the `kaiming_uniform_()` and `kaiming_normal_()` functions, along with `nonlinearity="leaky_relu"` to get the appropriately adjusted Kaiming initialization:

```
alpha = 0.2
model = nn.Sequential(nn.Linear(50, 40), nn.LeakyReLU(negative_slope=alpha))
nn.init.kaiming_uniform_(model[0].weight, alpha, nonlinearity="leaky_relu")
```

ReLU, leaky ReLU, and PReLU all suffer from the fact that they are not smooth functions: their slopes abruptly change at $z = 0$. As we saw in Chapter 4 when we discussed lasso, this sort of discontinuity in the derivatives can make gradient descent bounce around the optimum and slow down convergence. So now we will look at some smooth variants of the ReLU activation function, starting with ELU and SELU.

ELU and SELU

In 2015, a paper by Djork-Arné Clevert et al.[8] proposed a new activation function, called the *exponential linear unit* (ELU), that outperformed all the ReLU variants in the authors' experiments: training time was reduced, and the neural network performed better on the test set. Equation 11-2 shows this activation function's definition.

Equation 11-2. ELU activation function

The ELU activation function looks a lot like the ReLU function (see Figure 11-3), with a few major differences:

- It takes on negative values when $z < 0$, which allows the unit to have an average output closer to 0 and helps alleviate the vanishing gradients problem. The hyperparameter α defines the opposite of the value that the ELU function approaches when z is a large negative number. It is usually set to 1, but you can tweak it like any other hyperparameter.
- It has a nonzero gradient for $z < 0$, which avoids the dead neurons problem.
- 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 to the left and right of $z = 0$.

Using ELU with PyTorch is as easy as using the `nn.ELU` module, along with Kaiming initialization. The main drawback of the ELU activation function is that it is slower to compute than the ReLU function and its variants (due to the use of the exponential function). Its faster convergence rate during training may compensate for that slow computation, but still, at test time an ELU network will be a bit slower than a ReLU network.

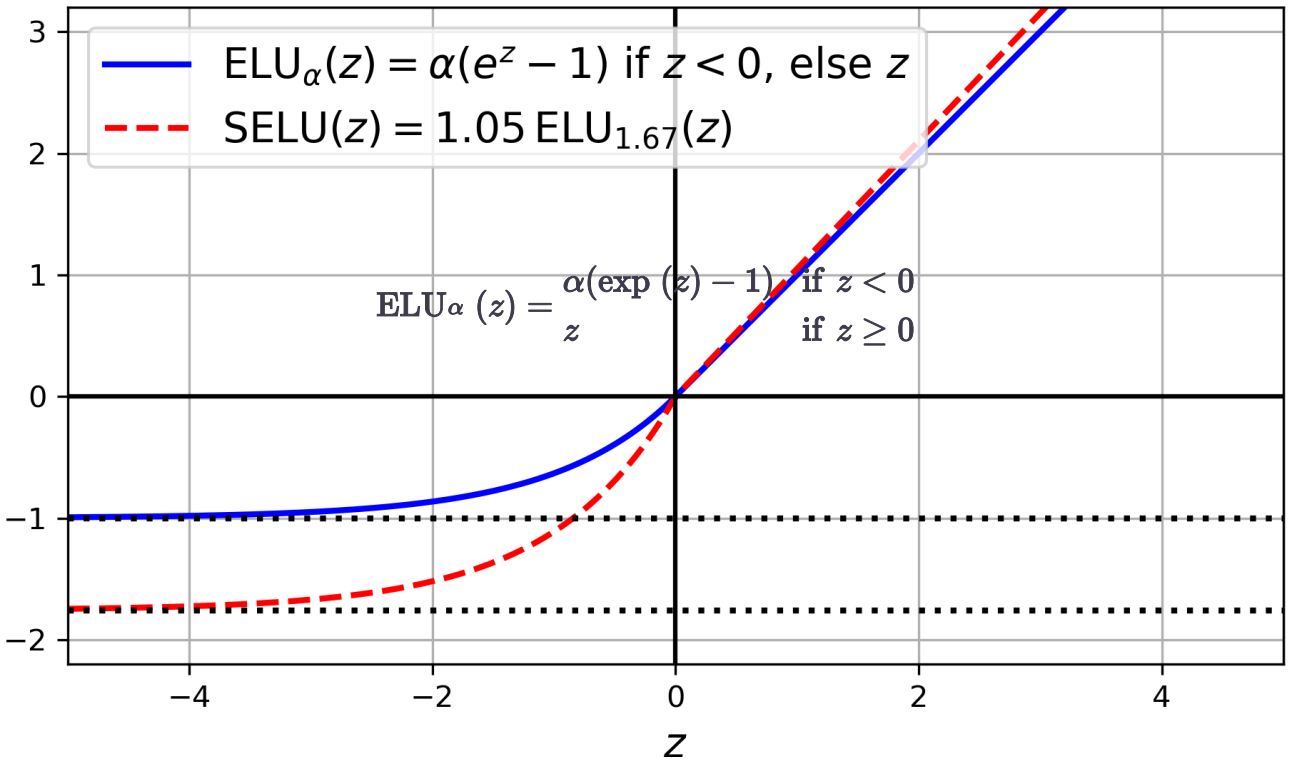

Figure 11-3. ELU and SELU activation functions

Not long after, a 2017 paper by Günter Klambauer et al.[9] introduced the *scaled ELU* (SELU) activation function: as its name suggests, it is a scaled variant of the ELU activation function (about 1.05 times ELU, using $\alpha \approx 1.67$). The authors showed that if you build a neural network composed exclusively of a stack of dense layers (i.e., an MLP), and if all hidden layers use the SELU activation function, then the network will *self-normalize*: the output of each layer will tend to preserve a mean of 0 and a standard deviation of 1 during training, which solves the vanishing/exploding gradients problem. As a result, the SELU activation function may outperform other activation functions for MLPs, especially deep ones. To use it with PyTorch, just use `nn.SELU`. There are, however, a few conditions for self-normalization to happen (see the paper for the mathematical justification):

- The input features must be standardized: mean 0 and standard deviation 1.
- Every hidden layer's weights must be initialized using LeCun normal initialization.
- The self-normalizing property is only guaranteed with plain MLPs. If you try to use SELU in other architectures, like recurrent networks (see Chapter 13) or networks with *skip connections* (i.e., connections that skip layers, such as in Wide & Deep neural networks), it will probably not outperform ELU.
- You cannot use regularization techniques like ℓ_1 or ℓ_2 regularization, batch-norm, layer-norm, max-norm, or regular dropout (these are discussed later in this chapter).

These are significant constraints, so despite its promises, SELU did not gain a lot of traction. Moreover, other activation functions seem to outperform it quite consistently on most tasks. Let's look at some of the most popular ones.

GELU, Swish, SwiGLU, Mish, and RELU[2]

The *Gaussian Error Linear Unit* (*GELU*) was introduced in a 2016 paper by Dan Hendrycks and Kevin Gimpel.[10] Once again, you can think of it as a smooth variant of the ReLU activation function. Its definition is given in Equation 11-3, where Φ is the standard Gaussian cumulative distribution function (CDF): $\Phi(z)$ corresponds to the probability that a value sampled randomly from a normal distribution of mean 0 and variance 1 is lower than z.

Equation 11-3. GELU activation function

As you can see in Figure 11-4, GELU resembles ReLU: it approaches 0 when its input z is very negative, and it approaches z when z is very positive. However, whereas all the activation functions we've discussed so far were both convex and monotonic,[11] the GELU activation function is neither: from left to right, it starts by going straight, then it wiggles down, reaches a low point around –0.17 (near $z \approx -0.75$), and finally bounces up and ends up going straight toward the top right. This fairly complex shape and the fact that it has a curvature at every point may explain why it works so well, especially for complex tasks: gradient descent may find it easier to fit complex patterns. In practice, it often outperforms every other activation function discussed so far. However, it is a bit more computationally intensive, and the performance boost it provides is not always sufficient to justify the extra cost. That said, it is possible to show that

it provides is not always sufficient to justify the extra cost. That said, it is possible to show that it is approximately equal to $z\sigma(1.702\,z)$, where σ is the sigmoid function: using this approximation also works very well, and it has the advantage of being much faster to compute.

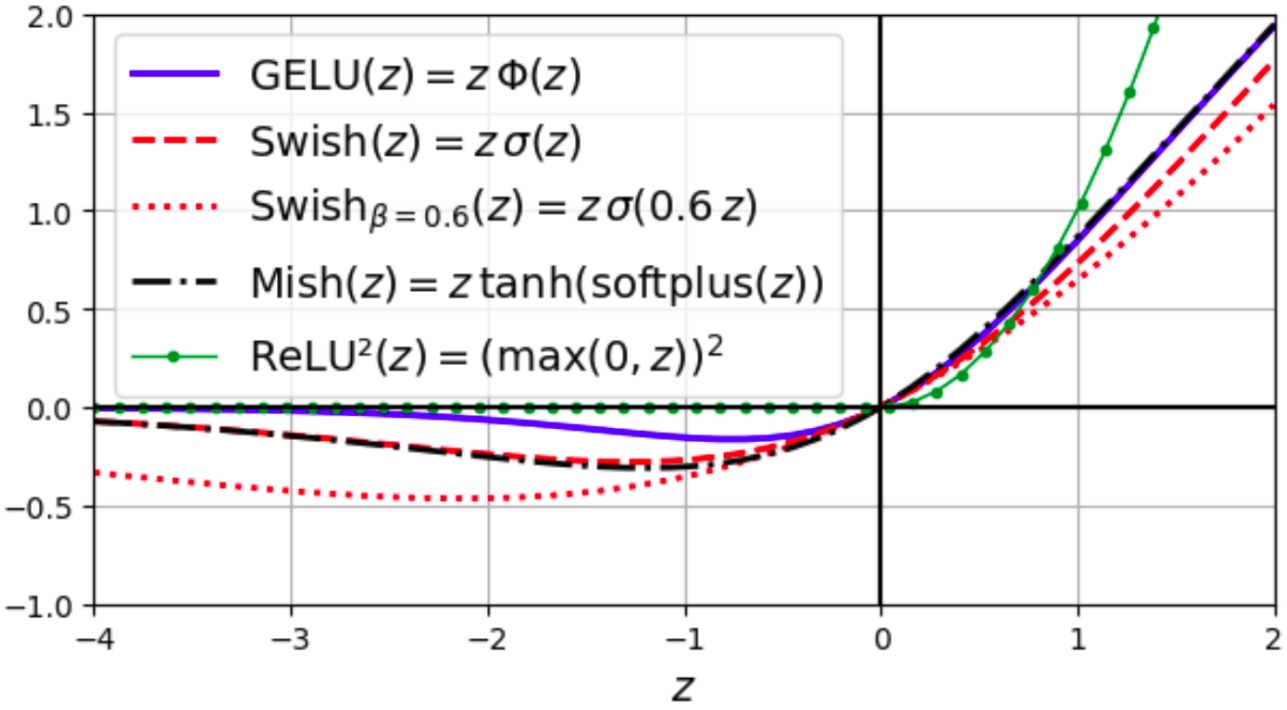

Figure 11-4. GELU, Swish, parametrized Swish, Mish, and ReLU^2 activation functions

The GELU paper also introduced the *sigmoid linear unit* (SiLU) activation function, which is equal to $z\sigma(z)$, but it was outperformed by GELU in the authors' tests. Interestingly, a 2017 paper by Prajit Ramachandran et al.[12] rediscovered the SiLU function by automatically searching for good activation functions. The authors named it *Swish*, and the name caught on. In their paper, Swish outperformed every other function, including GELU. Ramachandran et al. later generalized Swish by adding an extra scalar hyperparameter β to scale the sigmoid function's input. The generalized Swish function is $\text{Swish}_\beta(z) = z\sigma(\beta z)$, so GELU is approximately equal to the generalized Swish function using $\beta = 1.702$. You can tune β like any other hyperparameter. Alternatively, it's also possible to make β trainable and let gradient descent optimize it (a bit like PReLU): there is typically a single trainable β parameter for the whole model, or just one per layer, to keep the model efficient and avoid overfitting.

A popular Swish variant is *SwiGLU*:[13] the inputs go through the Swish activation function, and in parallel through a linear layer, then both outputs are multiplied itemwise. That's $\text{SwiGLU}(\mathbf{z}) = \text{Swish}_\beta(\mathbf{z}) \otimes \text{Linear}(\mathbf{z})$. This is often implemented by doubling the output dimensions of the previous linear layer, then splitting the outputs in two along the feature dimension to get $\mathbf{z}_1$ and $\mathbf{z}_2$, and finally applying: $\text{SwiGLU}_\beta(\mathbf{z}) = \text{Swish}_\beta(\mathbf{z}_1) \otimes \mathbf{z}_2$. This is a variant of the *gated linear unit (GLU)*[14] introduced by Facebook researchers in 2016. The itemwise multiplication gives the model more expressive power, allowing it to learn when to turn off (i.e., multiply by 0) or amplify specific features: this is called a *gating mechanism*. SwiGLU is very common in modern transformers (see Chapter 15).

Another GELU-like activation function is *Mish*, which was introduced in a 2019 paper by Diganta Misra.[15] It is defined as $\text{mish}(z) = z\tanh(\text{softplus}(z))$, where $\text{softplus}(z) = \log(1 + \exp(z))$. Just like GELU and Swish, it is a smooth, nonconvex, and nonmonotonic variant of ReLU, and once again the author ran many experiments and found that Mish generally outperformed

once again the author ran many experiments and found that Mish generally outperformed other activation functions—even Swish and GELU, by a tiny margin. Figure 11-4 shows GELU, Swish (both with the default $\beta = 1$ and with $\beta = 0.6$), and lastly Mish. As you can see, Mish overlaps almost perfectly with Swish when z is negative, and almost perfectly with GELU when z is positive.

Lastly, in 2021, Google researchers ran an automated architecture search to improve large transformers, and the search found a very simple yet effective activation function: ReLU^2.[2][16] As its name suggests, it's simply ReLU squared: $\text{ReLU}^2(z) = (\max(0, z))^2$. It has all the qualities of ReLU (simplicity, computational efficiency, sparse output, no saturation on the positive side) but it also has smooth gradients at $z = 0$, and it often outperforms other activation functions, especially for sparse models. However, training can be less stable, in part because of its increased sensitivity to outliers and dying ReLUs.

TIP

So, which activation function should you use for the hidden layers of your deep neural networks? ReLU remains a good default for most tasks: it's often just as good as the more sophisticated activation functions, plus it's very fast to compute, and many libraries and hardware accelerators provide ReLU-specific optimizations. However, Swish is probably a better default for complex tasks, and you can even try parametrized Swish with a learnable β parameter for the most complex tasks. Mish and SwiGLU may give you slightly better results, but they require a bit more compute. If you care a lot about runtime latency, then you may prefer leaky ReLU, or parametrized leaky ReLU for complex tasks, or even ReLU^2, especially for sparse models.

PyTorch supports GELU, Mish, and Swish out of the box (using `nn.GELU`, `nn.Mish`, and `nn.SiLU`, respectively). To implement SwiGLU, double the previous linear layer's output dimension, then use `z1, z2 = z.chunk(2, dim=-1)` to split its output in two, and compute `F.silu(beta * z1) * z2` (where `F` is `torch.nn.functional`). For ReLU^2, simply compute `F.relu(z).square()`. PyTorch also includes simplified and approximated versions of several activation functions, which are much faster to compute and often more stable during training. These simplified versions have names starting with "Hard", such as `nn.Hardsigmoid`, `nn.Hardtanh`, and `nn.Hardswish`, and they are often used on mobile devices.

That's all for activation functions! Now, let's look at a completely different way to solve the unstable gradients problem: batch normalization.

Batch Normalization

Although using Kaiming initialization along with ReLU (or any of its variants) can significantly reduce the danger of the vanishing/exploding gradients problems at the beginning of training, it doesn't guarantee that they won't come back during training.

In a 2015 paper,[17] Sergey Ioffe and Christian Szegedy proposed a technique called *batch normalization* (BN) that addresses these problems. The technique consists of adding an operation

in the model just before or after the activation function of each hidden layer. This operation simply zero-centers and normalizes each input, then scales and shifts the result using two new parameter vectors per layer: one for scaling, the other for shifting. In other words, the 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 (no need for `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 standard deviation of the input over the current mini-batch (hence the name "batch normalization"). The whole operation is summarized step by step in Equation 11-4.

Equation 11-4. Batch normalization algorithm

In this algorithm:

- $\boldsymbol{\mu}_B$ is the vector of input means, evaluated over the whole mini-batch B (it contains one mean per input).
- m_B is the number of instances in the mini-batch.
- $\mathbf{x}^{(i)}$ is the input vector of the batch norm layer for instance i.
- $\boldsymbol{\sigma}_B$ is the vector of input standard deviations, also evaluated over the whole mini-batch (it contains one standard deviation per input).
- $^{(i)}$ is the vector of zero-centered and normalized inputs for instance i.
- ε is a tiny number that avoids division by zero and ensures the gradients don't grow too large (typically 10^{-5}). This is called a *smoothing term*.
- $\boldsymbol{\gamma}$ is the output scale parameter vector for the layer (it contains one scale parameter per input).
- $\otimes$ represents element-wise multiplication (each input is multiplied by its corresponding output scale parameter).
- $\boldsymbol{\beta}$ is the output shift (offset) parameter vector for the layer (it contains one shift parameter per input). Each input is offset by its corresponding shift parameter.
- $\mathbf{z}^{(i)}$ is the output of the BN operation. It is a rescaled and shifted version of the inputs.

So during training, BN standardizes its inputs, then rescales and offsets them. Good! What about at test time? Well, it's not that simple. Indeed, we may need to make predictions for indi-

vidual instances rather than for batches of instances: in this case, we will have no way to compute each input's standard deviation. Moreover, even if we do have a batch of instances, it may be too small, or the instances may not be independent and identically distributed, so computing statistics over the batch instances would be unreliable. One solution is to wait until the end of training, then run the whole training set through the neural network and compute the mean and standard 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, most implementations of batch norm estimate these final statistics during training by using a moving average of the layer's batch input means and variances. This is what PyTorch does automatically when you use its batch norm layers, such as `nn.BatchNorm1d` (which we will discuss in the next section). To sum up, four parameter vectors are learned in each batch norm layer: $\boldsymbol{\gamma}$ (the output scale vector) and $\boldsymbol{\beta}$ (the output offset vector) are learned through regular backpropagation, and $\boldsymbol{\mu}$ (the final input mean vector) and $\boldsymbol{\sigma}^2$ (the final input variance vector) are estimated using an exponential moving average. Note that $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}^2$ are estimated during training, but they are used only after training, once you switch the model to evaluation mode using `model.eval()`: $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}^2$ then replace $\boldsymbol{\mu}_{\boldsymbol{B}}$ and $\boldsymbol{\sigma}_{\boldsymbol{B}}^2$ in Equation 11-4.

Ioffe and Szegedy demonstrated that batch norm 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, commonly used to evaluate computer vision systems). The vanishing gradients problem was strongly reduced, to the point that they could use saturating activation functions such as tanh and even sigmoid. The networks were also much less sensitive to the weight initialization. The authors 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 norm 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 norm acts like a regularizer, reducing the need for other regularization techniques (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 discussed earlier). Moreover, there is a runtime penalty: the neural network makes slower predictions due to the extra computations required at each layer. Fortunately, it's often possible to fuse the BN layer with the previous layer after training, thereby avoiding the runtime penalty. This is done by updating the previous layer's weights and biases so that it directly produces outputs of the appropriate scale and offset. For example, if the previous layer computes $\mathbf{XW} + \mathbf{b}$, then the BN layer will compute $\boldsymbol{\gamma} \otimes (\mathbf{XW} + \mathbf{b} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta}$ (ignoring the smoothing term ε in the denominator). If we define $\mathbf{W}' = \boldsymbol{\gamma} \otimes \mathbf{W} / \boldsymbol{\sigma}$ and $\mathbf{b}' = \boldsymbol{\gamma} \otimes (\mathbf{b} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta}$, the equation simplifies to $\mathbf{XW}' + \mathbf{b}'$. So, if we replace the

previous layer's weights and biases (**W** and **b**) with the updated weights and biases (**W**' and **b**'), we can get rid of the BN layer. This is one of the optimizations performed by `optimize_for_inference()` (see Chapter 10).

NOTE

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

Implementing batch norm with PyTorch

As with most things with PyTorch, implementing batch norm is straightforward and intuitive. Just add a `nn.BatchNorm1d` layer before or after each hidden layer's activation function, and specify the number of inputs of each BN layer. You may also add a BN layer as the first layer in your model, which removes the need to standardize the inputs manually. For example, let's create a Fashion MNIST image classifier (similar to the one we built in Chapter 10) using BN as the first layer in the model (after flattening the input images), then again after each hidden layer:

```
model = nn.Sequential(
    nn.Flatten(),
    nn.BatchNorm1d(1 * 28 * 28),
    nn.Linear(1 * 28 * 28, 300),
    nn.ReLU(),
    nn.BatchNorm1d(300),
    nn.Linear(300, 100),
    nn.ReLU(),
    nn.BatchNorm1d(100),
    nn.Linear(100, 10)
)
```

You can now train the model normally (as you learned in Chapter 10), and that's it! In this tiny example with just two hidden layers, batch norm is unlikely to have a large impact, but for deeper networks it can make a tremendous difference.

WARNING

Since batch norm behaves differently during training and during evaluation, it's critical to switch to training mode during training (using `model.train()`), and switch to evaluation mode during evaluation (using `model.eval()`). Forgetting to do so is one of the most common mistakes.

If you look at the parameters of the first BN layer, you will find two: `weight` and `bias`, which correspond to γ and β in Equation 11-4:

```
>>> dict(model[1].named_parameters()).keys()
dict_keys(['weight', 'bias'])
```

And if you look at the buffers of this same BN layer, you will find three: `running_mean`, `running_var`, and `num_batches_tracked`. The first two correspond to the running means μ and σ^2 discussed earlier, and `num_batches_tracked` simply counts the number of batches seen during training:

```
>>> dict(model[1].named_buffers()).keys()
dict_keys(['running_mean', 'running_var', 'num_batches_tracked'])
```

The authors of the BN paper argued in favor of adding the BN layers before the activation functions, rather than after (as we just did). There is some debate about this, and it seems to depend on the task, so you can experiment with this to see which option works best on your dataset. If you move the BN layers before the activation functions, you can also remove the bias term from the previous `nn.Linear` layers by setting their `bias` hyperparameter to `False`. Indeed, a batch norm layer already includes one bias term per input. You can also drop the first BN layer to avoid sandwiching the first hidden layer between two BN layers, but this means you should normalize the training set before training. The updated code looks like this:

```
model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(1 * 28 * 28, 300, bias=False),
    nn.BatchNorm1d(300),
    nn.ReLU(),
    nn.Linear(300, 100, bias=False),
    nn.BatchNorm1d(100),
    nn.ReLU(),
    nn.Linear(100, 10)
)
```

The `nn.BatchNorm1d` class has 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 by the `BatchNorm1d` layer when it updates the exponential moving averages; given a new value $\mathbf{v}$ (i.e., a new vector of input means or variances computed over the current batch), the layer updates the running average using the following equation:

A good momentum value is typically close to 0; for example, 0.01 or 0.001. You want more 0s for smaller mini-batches, and fewer for larger mini-batches. The default is 0.1, which is good for large batch sizes, but not great for small batch sizes such as 32 or 64.

WARNING

When people talk about “momentum” in the context of a running mean, they usually refer to the weight of the current running mean in the update equation. Sadly, for historical reasons, PyTorch uses the opposite meaning in the BN layers. However, other parts of PyTorch use the conventional meaning (e.g., in optimizers), so don’t get confused.

Batch norm 1D, 2D, and 3D

In the previous examples, we flattened the input images before sending them through the first `nn.BatchNorm1d` layer. This is because a `nn.BatchNorm1d` layer works on batches of shape `[batch_size, num_features]` (just like the `nn.Linear` layer does), so you would get an error if you moved it before the `nn.Flatten` layer.

However, you could use a `nn.BatchNorm2d` layer before the `nn.Flatten` layer: indeed, it expects its inputs to be image batches of shape `[batch_size, channels, height, width]`, and it computes the batch mean and variance across both the batch dimension (dimension 0) and the spatial dimensions (dimensions 2 and 3). This means that all pixels in the same batch and channel get normalized using the same mean and variance: the `nn.BatchNorm2d` layer only has one weight per channel and one bias per channel (e.g., three weights and three bias terms for color images with three channels for red, green, and blue). This generally works better when dealing with image datasets.

There’s also a `nn.BatchNorm3d` layer which expects batches of shape `[batch_size, channels, depth, height, width]`: this is useful for datasets of 3D images, such as CT scans.

The `nn.BatchNorm1d` layer can also work on batches of sequences. The convention in PyTorch is to represent batches of sequences as 3D tensors of shape `[batch_size, sequence_length, num_features]`. For example, suppose you work on particle physics and you have a dataset of particle trajectories, where each trajectory is composed of a sequence of 100 points in 3D space, then a batch of 32 trajectories will have a shape of `[32, 100, 3]`. However, the `nn.BatchNorm1d` layer expects the shape to be `[batch_size, num_features, sequence_length]`, and it computes the batch mean and variance across the first and last dimensions to get one mean and variance per feature. So you must permute the last two dimensions of the data using `X.permute(0, 2, 1)` before letting it go through the `nn.BatchNorm1d` layer. We will discuss sequences further in Chapter 13.

Batch normalization has become one of the most-used layers in deep neural networks, especially deep convolutional neural networks discussed in Chapter 12, to the point that it is often omitted in the architecture diagrams: it is assumed that BN is added after every layer. That said, it is not perfect. In particular, the computed statistics for an instance are biased by the other samples in a batch, which may reduce performance (especially for small batch sizes). Moreover, BN struggles with some architectures, such as recurrent nets, as we will see in Chapter 13. For these reasons, batch-norm is more and more often replaced by layer-norm.

Layer Normalization

Layer normalization (LN) is very similar to batch norm, but instead of normalizing across the batch dimension, LN normalizes across the feature dimensions. This simple idea was introduced by Jimmy Lei Ba et al. in a 2016 paper,[18] and initially applied mostly to recurrent nets. However, in recent years it has been successfully applied to many other architectures, such as convolutional nets, transformers, diffusion nets, and more.

One advantage is that LN can compute the required statistics on the fly, at each time step, independently for each instance. This also means that it behaves the same way during training and testing (as opposed to BN), and it does not need to use exponential moving averages to estimate the feature statistics across all instances in the training set, like BN does. Lastly, LN learns a scale and an offset parameter for each input feature, just like BN does.

PyTorch includes a `nn.LayerNorm` module. To create an instance, you must simply indicate the size of the dimensions that you want to normalize over. These must be the last dimension(s) of the inputs. For example, if the inputs are batches of 100 × 200 RGB images of shape `[3, 100, 200]`, and you want to normalize each image over each of the three color channels separately, you would use the following `nn.LayerNorm` module:

```
inputs = torch.randn(32, 3, 100, 200)  # a batch of random RGB images
layer_norm = nn.LayerNorm([100, 200])
result = layer_norm(inputs)  # normalizes over the last two dimensions
```

The following code produces the same result:

```
means = inputs.mean(dim=[2, 3], keepdim=True)  # shape: [32, 3, 1, 1]
vars_ = inputs.var(dim=[2, 3], keepdim=True, unbiased=False)  # shape: same
stds = torch.sqrt(vars_ + layer_norm.eps)  # eps is a smoothing term (1e-5)
result = layer_norm.weight * (inputs - means) / stds + layer_norm.bias
# result shape: [32, 3, 100, 200]
```

However, most computer vision architectures that use LN normalize over all channels at once. For this, you must include the size of the channels dimension when creating the `nn.LayerNorm` module:

```
layer_norm = nn.LayerNorm([3, 100, 200])
result = layer_norm(inputs)  # normalizes over the last three dimensions
```

And that's all there is to it! Now let's look at one last technique to stabilize gradients during training: gradient clipping.

Gradient Clipping

Another technique to mitigate the exploding gradients problem is to clip the gradients during backpropagation so that they never exceed some threshold. This is called *gradient clipping*.[19] This technique is generally used in recurrent neural networks, where using batch norm is tricky (as you will see in Chapter 13).

In PyTorch, gradient clipping is generally implemented by calling either `torch.nn.utils.clip_grad_norm_()` or `torch.nn.utils.clip_grad_value_()` at each iteration during training, right after the gradients are computed (i.e., after `loss.backward()`). Both functions take as a first argument the list of model parameters whose gradients must be clipped—typically all of them (`model.parameters()`). The `clip_grad_norm_()` function clips each gradient vector's norm if it exceeds the given `max_norm` argument. This is a hyperparameter you can tune (a typical default value is 1.0). The `clip_grad_value_()` function independently clips the individual components of the gradient vector between `-clip_value` and `+clip_value`, where `clip_value` is a hyperparameter you can tune. For example, this training loop clips the norm of each gradient vector to 1.0:

```
for epoch in range(n_epochs):
    for X_batch, y_batch in train_loader:
        X_batch, y_batch = X_batch.to(device), y_batch.to(device)
        y_pred = model(X_batch)
        loss = loss_fn(y_pred, y_batch)
        loss.backward()
        nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()
        optimizer.zero_grad()
```

Note that `clip_grad_value_()` will change the orientation of the gradient vector when its components are clipped. For instance, if the original gradient vector is `[0.9, 100.0]`, it points mostly in the direction of the second dimension; but once you clip it by value, you get `[0.9, 1.0]`, which points roughly at the diagonal between the two axes. Despite this reorientation, this approach actually works quite well in practice. If you clipped the same vector by norm, the result would be `[0.00899964, 0.9999595]`: this would preserve the vector's orientation, but almost eliminate the first component. The best clipping function to use depends on the dataset.

Reusing Pretrained Layers

It is generally not a good idea to train a very large DNN from scratch without first trying to find an existing neural network that accomplishes a similar task to the one you are trying to tackle (I will discuss how to find them in Chapter 12). If you find such a neural network, then you can generally reuse most of its layers, except for the top ones. This technique is called *transfer learning*. It will not only speed up training considerably, but also require significantly less training data.

Suppose you have access to a DNN that was trained to classify pictures into one hundred different categories, including animals, plants, vehicles, and everyday objects, and 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 11-5).

NOTE

If the input pictures for 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 original model. More generally, transfer learning will work best when the inputs have similar low-level features. For example, a neural net trained on regular pictures taken from mobile phones will help with many other tasks on mobile phone pictures, but it will likely not help at all on satellite images or medical images.

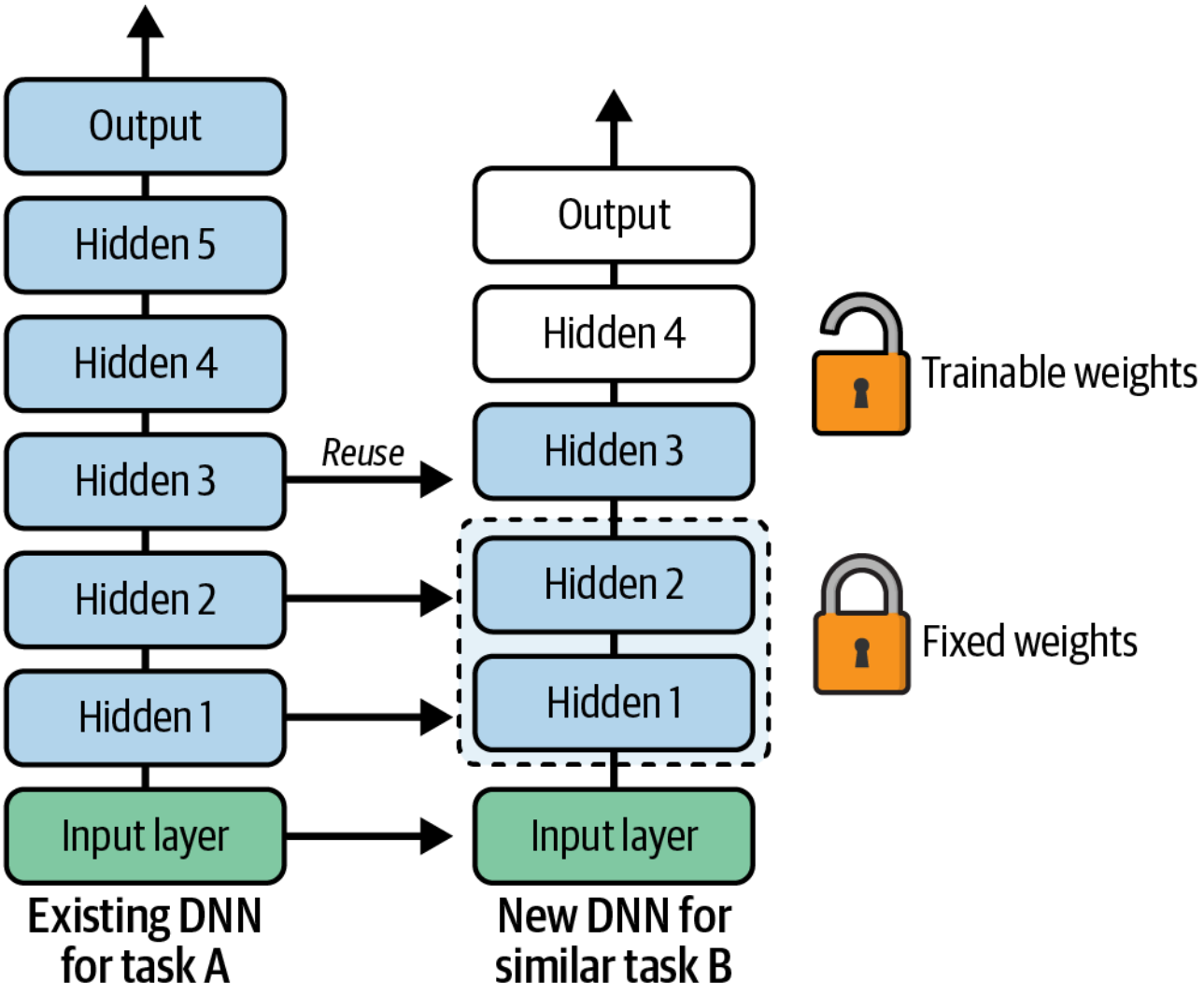

Figure 11-5. Reusing pretrained layers

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

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.

TIP

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

Try freezing all the reused layers first (i.e., make their parameters nontrainable by setting `requires_grad` to `False` so that gradient descent won't modify them and they will remain fixed), 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 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 dropping the top hidden layer(s) and freezing all the remaining hidden layers again. You can iterate until you find the right number of layers to reuse. If you have plenty of training data, you may try replacing the top hidden layers instead of dropping them, and even adding more hidden layers.

Transfer Learning with PyTorch

Let's look at an example. Suppose the Fashion MNIST dataset only contained eight classes—for example, all the classes except for Pullover and T-shirt/top. Someone built and trained a PyTorch model on that set and got reasonably good performance (~92% accuracy). Let's call this model A. You now want to tackle a different task: you have images of T-shirts and pullovers, and you want to train a binary classifier: positive for T-shirt/top, negative for Pullover. Your dataset is tiny; you only have 20 labeled images! When you train a new model for this task (let's call it model B) with the same architecture as model A, you get 71.6% test accuracy. While drinking your morning coffee, you realize that your task is quite similar to task A, so perhaps transfer learning can help? Let's find out!

First, let's look at model A:

```
torch.manual_seed(42)

model_A = nn.Sequential(
    nn.Flatten(),
    nn.Linear(1 * 28 * 28, 100),
    nn.ReLU(),
    nn.Linear(100, 100),
    nn.ReLU(),
    nn.Linear(100, 100),
    nn.ReLU(),
    nn.Linear(100, 8)
)
[...]  # train this model or load pretrained weights
```

We can now reuse the layers we want, for example, all layers except for the output layer:

```
import copy

torch.manual_seed(42)
reused_layers = copy.deepcopy(model_A[:-1])
model_B_on_A = nn.Sequential(
    *reused_layers,
    nn.Linear(100, 1)  # new output layer for task B
).to(device)
```

In this code, we use Python's `copy.deepcopy()` function to copy all the modules in the `nn.Sequential` module (along with all their data and submodules), except for the last layer. Since we're making a deep copy, all the submodules are copied as well. Then we create `model_B_on_A`, which is a `nn.Sequential` model based on the reused layers of model A, plus a new output layer for task B: it has a single output since task B is binary classification.

You could start training `model_B_on_A` for task B now, 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:

```
for layer in model_B_on_A[:-1]:
    for param in layer.parameters():
        param.requires_grad = False
```

Now you can train `model_B_on_A`. But don't forget that task B is binary classification, so you must switch the loss to `nn.BCEWithLogitsLoss` (or to `nn.BCELoss` if you prefer to add a `nn.Sigmoid` activation function on the output layer), as we discussed in Chapter 10. Also, if you are using `torchmetrics`, make sure to set `task="binary"` when creating the `Accuracy` metric:

```
xentropy = nn.BCEWithLogitsLoss()
accuracy = torchmetrics.Accuracy(task="binary").to(device)
[...]  # train model_B_on_A
```

After you have trained the model for a few epochs, you can unfreeze the reused layers (setting `param.requires_grad = True` for all parameters), reduce the learning rate, and continue training to fine-tune the reused layers for task B.

So, what's the final verdict? Well, this model's test accuracy is 92.5%, which is much better than the 71.6% accuracy we reached without pretraining!

Are you convinced? Well, you shouldn't be: I cheated! I tried many configurations until I found

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 actually 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 sheer luck—without mentioning how many failures they encountered along the way: that's called *p-hacking*. Most of the time, this is not malicious, but it is part of the reason why so many results in science can never be reproduced.

But why did I cheat? It turns out that transfer learning does not work very well with small dense networks, presumably because small networks learn few patterns, and dense networks learn very specific patterns, which are unlikely to be useful for other tasks. Transfer learning works best with deep convolutional neural networks and with Transformer architectures. We will revisit transfer learning in Chapter 12 and Chapter 15, using the techniques we just discussed (and this time it will work fine without cheating, I promise!).

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 hope! First, you should try to gather more labeled training data, but if you can't, you may still be able to perform *unsupervised pretraining* (see Figure 11-6). Indeed, it is often cheap to gather unlabeled training examples, but expensive to label them. If you can gather plenty of unlabeled training data, you can try to use it to train an unsupervised model, such as an autoencoder (see Chapter 18). Then you can reuse the lower layers of the autoencoder, add the output layer for your task on top, and fine-tune the final network using supervised learning (i.e., with the labeled training examples).

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, unsupervised pretraining—typically with restricted Boltzmann machines (RBMs; see the notebook at *https://homl.info/extra-anns*)—was the norm for deep nets, and only after the vanishing gradients problem was alleviated did it become much more common to train DNNs purely using supervised learning. Unsupervised pretraining (today typically using autoencoders or diffusion models 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.

Note that in the early days of deep learning it was difficult to train deep models, so people would use a technique called *greedy layer-wise pretraining* (depicted in Figure 11-6). They would first train an unsupervised model with a single layer, typically an RBM, then they would freeze that layer and add another one on top of it, then train the model again (effectively just training the new layer), then freeze the new layer and add another layer on top of it, train the model again, and so on. Nowadays, things are much simpler: people generally train the full unsupervised model in one shot and use models such as autoencoders or diffusion models rather than RBMs.

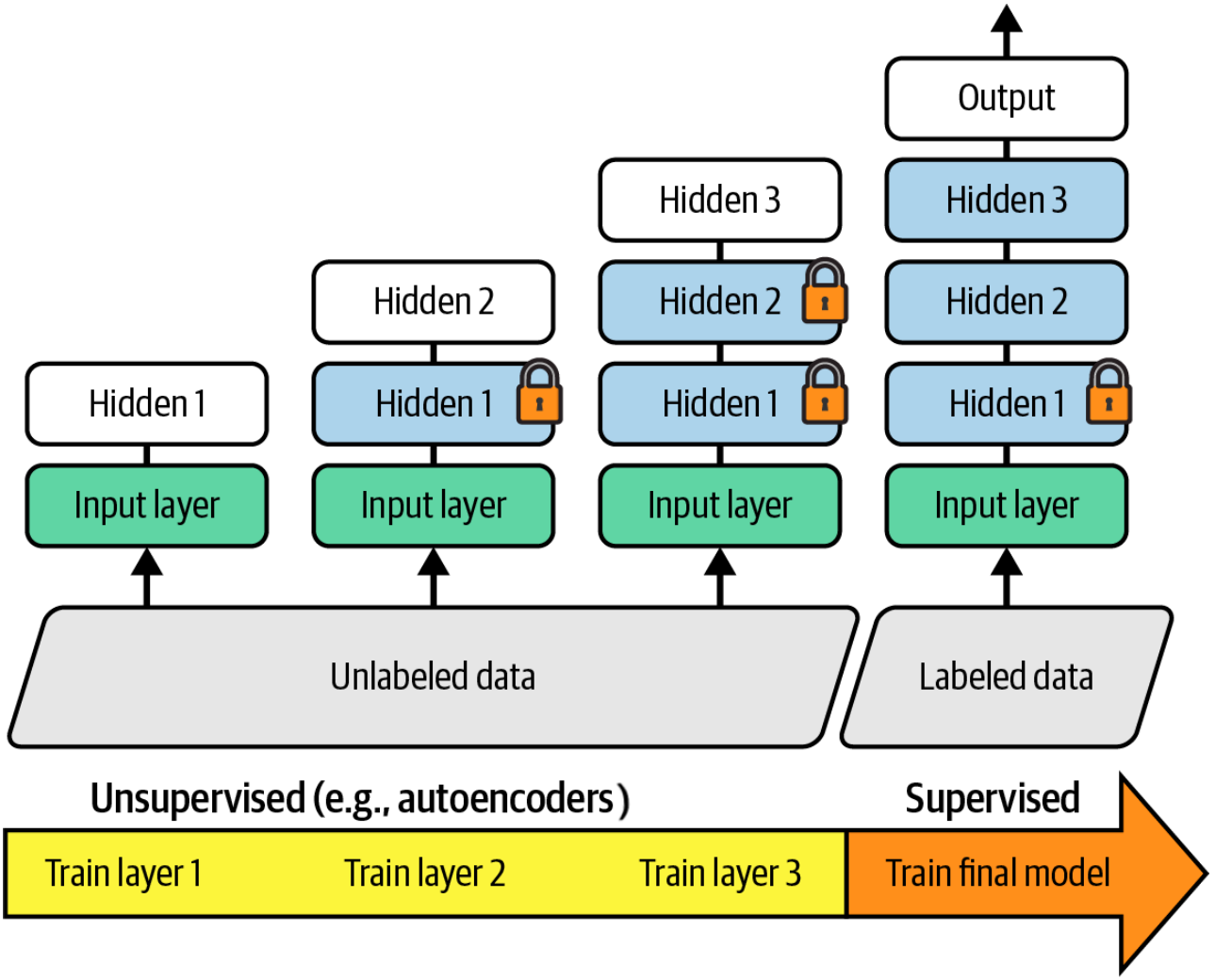

Figure 11-6. Greedy layer-wise pretraining used in the early days of deep learning; nowadays the unsupervised part is typically done in one shot on all the data rather than one layer at a time

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 reusable 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. Gathering hundreds of pictures of each person would not be practical. You could, however, use a public dataset containing millions of pictures of people (such as VGGFace2) and train a first neural network to detect whether two different pictures feature the same person. Such a network would learn good feature detectors for faces, so reusing its lower layers would allow you to train a good face classifier that uses little training data.

WARNING

You could also just scrape pictures of random people from the web, but this would probably be illegal. Firstly, photos are usually copyrighted by their creators, and websites like Instagram or Facebook enforce these copyright protections through their terms of service, which prohibit scraping and unauthorized use. Secondly, over 40 countries require explicit consent for collecting and processing personal data, in-

cluding facial images.

For natural language processing (NLP) applications, you can download a corpus of 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 performance 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 (this is basically how large language models are trained and fine-tuned, as we will see in Chapter 15).

NOTE

Self-supervised learning is when you automatically generate the labels from the data itself, as in the text-masking example, then you train a model on the resulting "labeled" dataset using supervised learning techniques.

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 initialization strategy for the connection weights, using a good activation function, using batch-norm or layer-norm, and reusing parts of a pretrained network (possibly built for 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 we will present the most popular optimization algorithms: momentum, Nesterov accelerated gradient, AdaGrad, RMSProp, and finally, Adam and its variants.

Momentum

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 core idea behind *momentum optimization*, proposed by Boris Polyak in 1964.[20] In contrast, regular gradient descent will take small steps when the slope is gentle and big steps when the slope is steep, but it will never pick up speed. As a result, regular gradient descent is generally much slower to reach the minimum than momentum optimization.

As we saw in Chapter 4, gradient descent updates the weights $\boldsymbol{\theta}$ by directly subtracting the gradient of the cost function $J(\boldsymbol{\theta})$ with regard to the weights ($\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$) multiplied by the learning rate η. The equation is $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. 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** (multiplied by the learning rate η), and it updates the weights by adding this momentum vector (see Equation 11-5). In other words, the gradient is used as a force learning to an acceleration, not as a speed. To simulate some sort of friction mechanism and prevent the momentum from growing too large, the algorithm introduces a new hyperparameter β, called the *momentum coefficient*, which must be set between 0 (high friction) and 1 (no friction). A typical momentum value is 0.9.

Equation 11-5. Momentum algorithm

You can 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 $1 / (1 - \beta)$ (ignoring the sign). For example, if $\beta = 0.9$, then the terminal velocity is equal to 10 times the gradient times the learning rate, so momentum optimization ends up going 10 times faster than gradient descent! In practice, the gradients are not constant, so the speedup is not always as dramatic, but momentum optimization can escape from plateaus much faster than regular gradient descent. 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 valley. 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-norm or layer-norm, the upper layers will often end up having inputs with very different scales, so using momentum optimization helps a lot. It can also help roll past local optima.

NOTE

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's good to have a bit of friction in the system: it reduces these oscillations and thus speeds up convergence.

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

```
optimizer = torch.optim.SGD(model.parameters(), momentum=0.9, lr=0.05)
```

The one drawback of momentum optimization is that it adds yet another hyperparameter 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,[21] is almost

always faster than regular momentum optimization. The *Nesterov accelerated gradient* (NAG) method, also known as *Nesterov momentum optimization*, measures the gradient of the cost function not at the local position $\mathbf{\theta}$ but slightly ahead in the direction of the momentum, at $\mathbf{\theta} + \beta\mathbf{m}$ (see Equation 11-6).

Equation 11-6. Nesterov accelerated gradient algorithm

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 the gradient at the original position, as you can see in Figure 11-7 (where ∇_1 represents the gradient of the cost function measured at the starting point $\mathbf{\theta}$, and ∇_2 represents the gradient at the point located at $\mathbf{\theta} + \beta\mathbf{m}$).

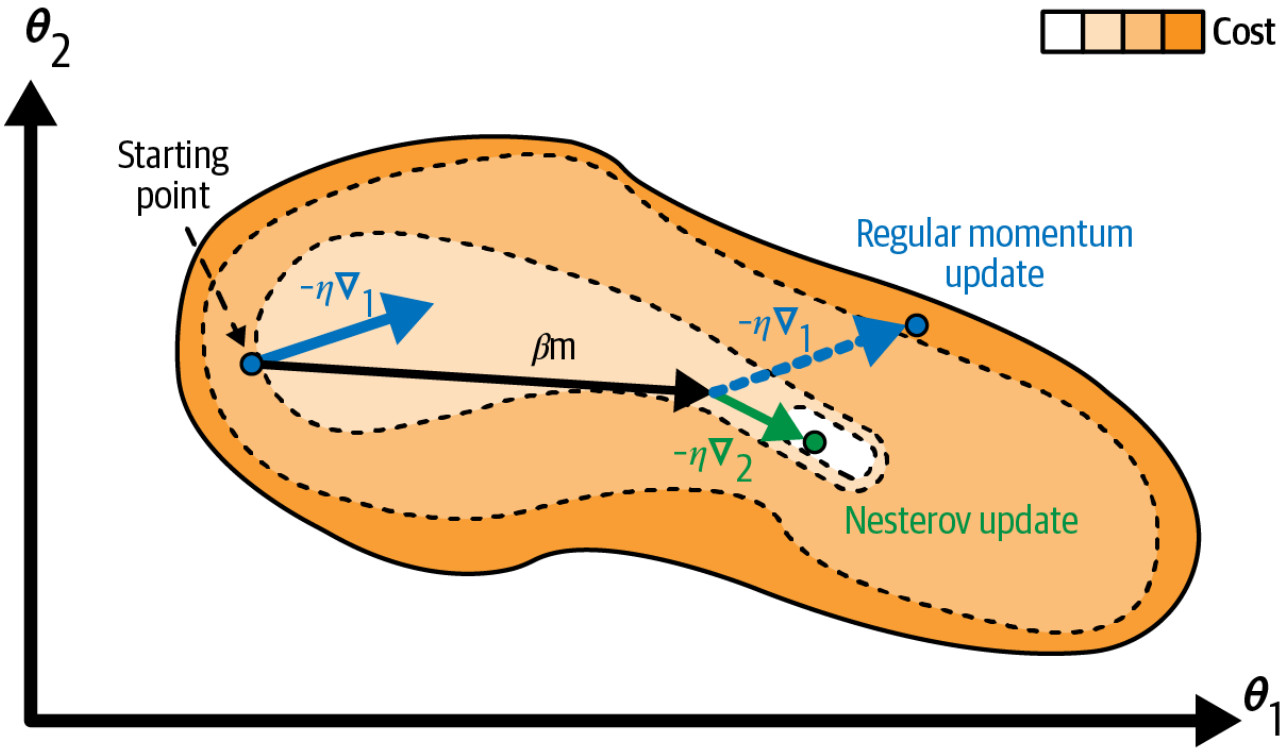

Figure 11-7. Regular versus Nesterov momentum optimization: the former applies the gradients computed before the momentum step, while the latter applies the gradients computed after

As you can see, the Nesterov update ends up closer to the optimum. After a while, these small improvements add up and NAG ends up being significantly faster than regular momentum optimization. Moreover, note that when the momentum pushes the weights across a valley, ∇_1 continues to push farther across the valley, while ∇_2 pushes back toward the bottom of the valley. This helps reduce oscillations and thus NAG converges faster.

To use NAG, simply set `nesterov=True` when creating the `SGD` optimizer:

```
optimizer = torch.optim.SGD(model.parameters(),
                            momentum=0.9, nesterov=True, lr=0.05)
```

AdaGrad

AdaGrad

Consider the elongated bowl problem again: gradient descent starts by quickly going down the steepest slope, which does not point straight toward the global optimum, then it very slowly goes down to the bottom of the valley. It would be nice if the algorithm could correct its direction earlier to point a bit more toward the global optimum. The *AdaGrad* algorithm[22] achieves this correction by scaling down the gradient vector along the steepest dimensions (see Equation 11-7).

Equation 11-7. AdaGrad algorithm

The first step accumulates the square of the gradients into the vector $\mathbf{s}$ (recall that the $\otimes$ symbol represents the element-wise multiplication). This vectorized form is equivalent to computing $s_i \leftarrow s_i + (\partial J(\boldsymbol{\theta}) / \partial \theta_i)^2$ for each element s_i of the vector $\mathbf{s}$; in other words, each s_i accumulates the squares of the partial derivative of the cost function with regard to parameter θ_i. If the cost function is steep along the i^{th} dimension, then s_i 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 $\oslash$ symbol represents the element-wise division, the square root is also computed element-wise, and ε is a smoothing term to avoid division by zero, typically set to 10^{-10}). This vectorized form is equivalent to simultaneously computing for all parameters θ_i.

In short, this algorithm decays the learning rate, but it does so faster for steep dimensions 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 11-8). One additional benefit is that it requires much less tuning of the learning rate hyperparameter η.

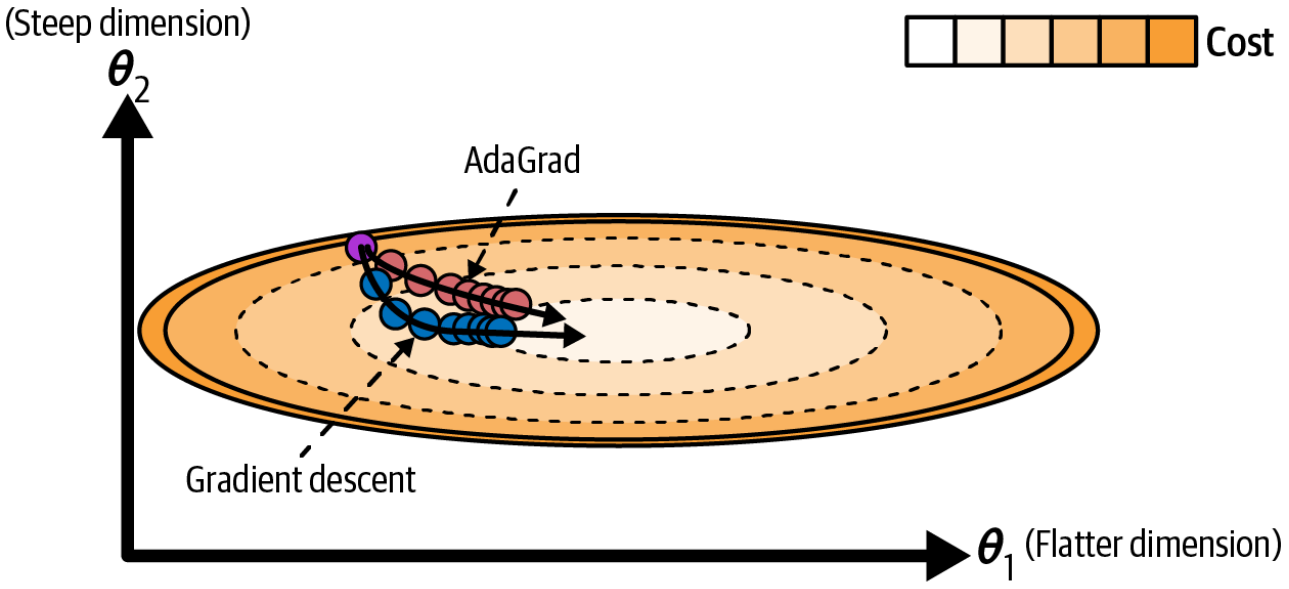

Figure 11-8. AdaGrad versus gradient descent: the former can correct its direction earlier to point to the optimum

AdaGrad frequently performs well for simple quadratic problems, but it often stops too early

AdaGrad frequently performs well for simple quadratic problems, but 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 PyTorch 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). Still, understanding AdaGrad is helpful to comprehend the other adaptive learning rate optimizers.

RMSProp

As we've seen, AdaGrad runs the risk of slowing down a bit too fast and never converging to the global optimum. The *RMSProp* algorithm[23] fixes this by accumulating only the gradients from the most recent iterations, as opposed to all the gradients since the beginning of training. It does so by using exponential decay in the first step (see Equation 11-8).

Equation 11-8. RMSProp algorithm

The decay rate α 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, PyTorch has an `RMSprop` optimizer:

```
optimizer = torch.optim.RMSprop(model.parameters(), alpha=0.9, lr=0.05)
```

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

Adam

Adam,[24] which stands for *adaptive moment estimation*, combines the ideas of momentum 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 11-9). 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.

Equation 11-9. Adam algorithm

In this equation, 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: β_1 corresponds to β in momentum optimization, and β_2 corresponds to α in 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 $1 - \beta_1$ 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 scaling 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^{-8}. These are the default values for the `Adam` class. Here is how to create an Adam optimizer using PyTorch:

```
optimizer = torch.optim.Adam(model.parameters(), betas=(0.9, 0.999), lr=0.05)
```

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 $\eta = 0.001$, making Adam even easier to use than gradient descent.

TIP

If you are starting to feel overwhelmed by all these different techniques and are 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, three variants of Adam are worth mentioning: AdaMax, NAdam, and AdamW.

AdaMax

The Adam paper also introduced AdaMax. Notice that in step 2 of Equation 11-9, Adam accumulates the squares of the gradients in **s** (with a greater weight for more recent gradients). In step 5, if we ignore ε and steps 3 and 4 (which are technical details anyway), Adam scales down the parameter updates by the square root of **s**. In short, Adam scales down the parameter updates by the ℓ_2 norm of the time-decayed gradients (recall that the ℓ_2 norm is the square root of the sum of squares).

AdaMax replaces the ℓ_2 norm with the ℓ_∞ norm (a fancy way of saying the max). Specifically, it replaces step 2 in Equation 11-9 with , it drops step 4, and in step 5 it scales down the gradient updates by a factor of **s**, which is the max of the absolute

value of the time-decayed gradients.

In practice, this can make AdaMax more stable than Adam, but it really depends on the dataset, and in general Adam performs better. So, this is just one more optimizer you can try if you experience problems with Adam on some task.

NAdam

NAdam optimization is Adam optimization plus the Nesterov trick, so it will often converge slightly faster than Adam. In his report introducing this technique,[25] the researcher Timothy Dozat compares many different optimizers on various tasks and finds that NAdam generally outperforms Adam but is sometimes outperformed by RMSProp.

AdamW

AdamW[26] is a variant of Adam that integrates a regularization technique called *weight decay*. Weight decay reduces the size of the model's weights at each training iteration by multiplying them by a decay factor such as 0.99. This may remind you of ℓ_2 regularization (introduced in Chapter 4), which also aims to keep the weights small, and indeed it can be shown mathematically that ℓ_2 regularization is equivalent to weight decay when using SGD. However, when using Adam or its variants, ℓ_2 regularization and weight decay are *not* equivalent: in practice, combining Adam with ℓ_2 regularization results in models that often don't generalize as well as those produced by SGD. AdamW fixes this issue by properly combining Adam with weight decay.

WARNING

Adaptive optimization methods (including RMSProp, Adam, AdaMax, NAdam, and AdamW optimization) are often great, converging fast to a good solution. However, a 2017 paper[27] by Ashia C. Wilson et al. showed that they can lead to solutions that generalize poorly on some datasets. So when you are disappointed by your model's performance, try using NAG instead: your dataset may just be allergic to adaptive gradients.

To use NAdam, AdaMax, or AdamW in PyTorch, replace `torch.optim.Adam` with `torch.optim.NAdam`, `torch.optim.Adamax`, or `torch.optim.AdamW`. For AdamW, you probably want to tune the `weight_decay` hyperparameter.

All the optimization techniques discussed so far only rely on the *first-order partial derivatives* (*Jacobians*, which measure the slope of the loss function along each axis). The optimization literature also contains amazing algorithms based on the *second-order partial derivatives* (the *Hessians*, which are the partial derivatives of the Jacobians, measuring how each Jacobian changes along each axis; in other words, measuring the loss function's curvature).

Unfortunately, these Hessian-based algorithms are hard to apply directly to deep neural networks because there are n^2 second-order derivatives per output (where n is the number of pa-

rameters), as opposed to just n first-order derivatives per output. Since DNNs typically have hundreds of thousands of parameters or more, the second-order optimization algorithms often don't even fit in memory, and even when they do, computing the *Hessian matrix* is just too slow.[28]

Luckily, it is possible to use stochastic methods that can efficiently approximate second-order information. One such algorithm is Shampoo,[29] which uses accumulated gradient information to approximate the second-order terms, similar to how Adam accumulates first-order statistics. It is not included in the PyTorch library, but you can get it in the PyTorch-Optimizer library (`pip install torch_optimizer`).

TRAINING SPARSE MODELS

All the optimization algorithms we just discussed 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 way to achieve this is to train the model as usual, then get rid of the tiny weights (set them to zero) using `torch.nn.prune.l1_unstructured()`. Or you can get rid of entire neurons, channels, or layers, not just individual weights, using `torch.nn.prune.ln_structured()`, or other functions in the `torch.nn.prune` package.

You should generally also apply fairly strong sparsity inducing regularization during training, such as ℓ_1 regularization (you'll see how later in this chapter), since it pushes the optimizer to zero out as many weights as it can (see "Lasso Regression"). You can also try scaling down random weights during initialization to encourage sparsity.

Table 11-2 compares all the optimizers we've discussed so far (⭐ is bad, ⭐⭐ is average, and ⭐⭐⭐ is good).

Table 11-2. Optimizer comparison

Class	Convergence speed	Convergence quality
`SGD`	⭐	⭐⭐⭐
`SGD(momentum=...)`	⭐⭐	⭐⭐⭐
`SGD(momentum=..., nesterov=True)`	⭐⭐	⭐⭐⭐
`Adagrad`	⭐⭐⭐	⭐ (stops too early)
`RMSprop`	⭐⭐⭐	⭐⭐ or ⭐⭐⭐
`Adam`	⭐⭐⭐	⭐⭐ or ⭐⭐⭐
`AdaMax`	⭐⭐⭐	⭐⭐ or ⭐⭐⭐
`NAdam`	⭐⭐⭐	⭐⭐ or ⭐⭐⭐

`AdamW`	⭐⭐⭐	⭐⭐ or ⭐⭐⭐

Learning Rate Scheduling

Finding a good learning rate is very important. If you set it too high, training will diverge (as discussed in "Gradient Descent"). If you set it too low, then training will be painfully slow, and it may also get stuck in a local optimum and produce a suboptimal model. If you set the learning rate fairly high (but not high enough to diverge), then training will often make rapid progress at first, but it will end up dancing around the optimum toward the end of training and thereby produce a suboptimal model. If you find a really good learning rate, you can end up with an excellent model, but training will generally be a bit too slow. Luckily, you can do better than a constant learning rate. In particular, it's a good idea to start with a fairly high learning rate and then reduce it toward the end of training (or whenever progress stops): this ensures that training starts fast, while also allowing backprop to settle down toward the end to really fine-tune the model parameters (see Figure 11-9).

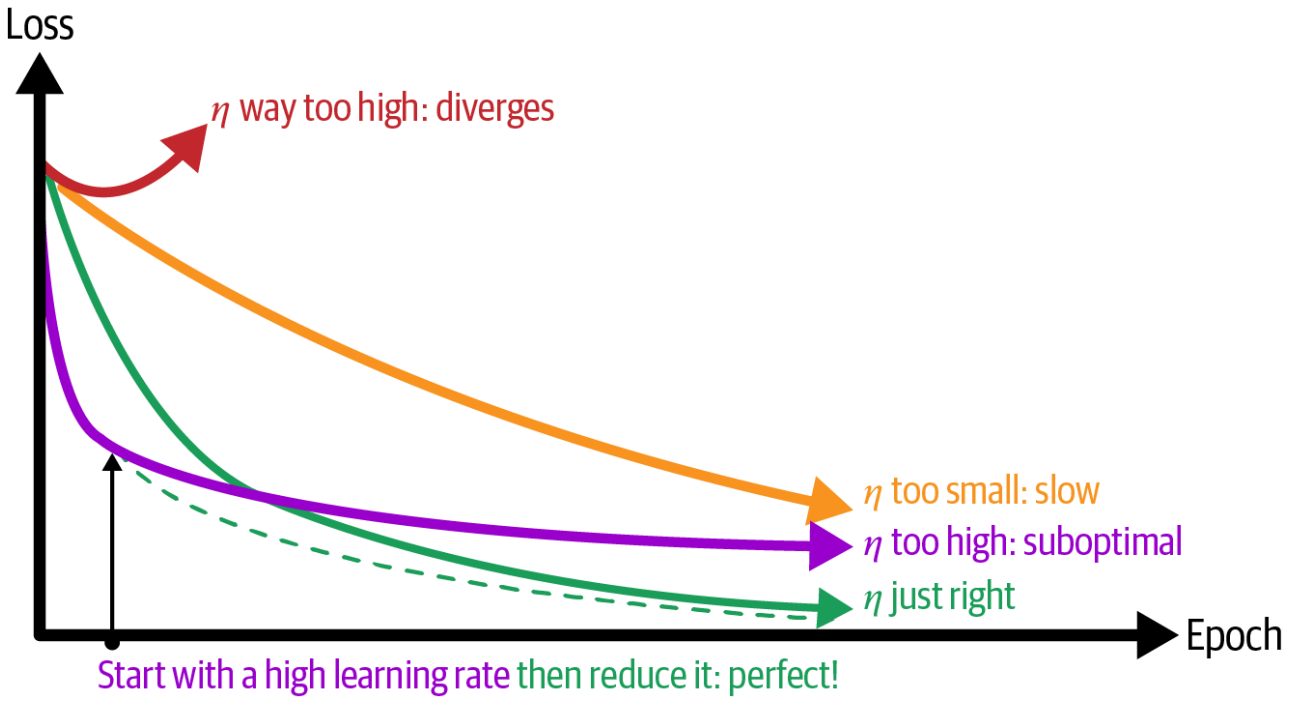

Figure 11-9. Learning curves for various learning rates η

There are various other strategies to tweak the learning rate during training. These are called *learning schedules* (I briefly introduced this concept in Chapter 4). The `torch.optim.lr_scheduler` module provides several implementations of common learning schedules. Let's look at the most important ones, starting with exponential scheduling.

Exponential Scheduling

The `ExponentialLR` class implements *exponential scheduling*, whereby the learning rate is multiplied by a constant factor `gamma` at some regular interval, typically at every epoch. As a result, after the n^{th} epoch, the learning rate will be equal to the initial learning rate times `gamma` to the power of n. This factor `gamma` is yet another hyperparameter you can tune. In general, you will want to set `gamma` to a value lower than 1, but fairly close to 1 to avoid de-

creasing the learning rate too fast. For example, if `gamma` is set to 0.9, then after 10 epochs the learning rate will be about 35% of the initial learning rate, and after 20 epochs it will be about 12%.

The `ExponentialLR` constructor expects at least two arguments: the optimizer whose learning rate will be tweaked during training, and the factor `gamma`:

```
model = [...]  # build the model
optimizer = torch.optim.SGD(model.parameters(), lr=0.05)  # or any other optim.
scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.9)
```

Next, you must update the training loop to call `scheduler.step()` at the end of each epoch to tweak the optimizer's learning rate:

```
for epoch in range(n_epochs):
    for X_batch, y_batch in train_loader:
        [...]  # the rest of the training loop remains unchanged

    scheduler.step()
```

TIP

If you interrupt training and you later want to resume it where you left off, you should set the `last_epoch` argument of the scheduler's constructor to the last epoch you ran (zero-indexed). The default is –1, which makes the scheduler start training from scratch.

Cosine Annealing

Instead of decreasing the learning rate exponentially, you can use the cosine function to go from the maximum learning rate $\eta_{\mathbf{max}}$ at the start of training, down to the minimum learning rate $\eta_{\mathbf{min}}$ at the end. This is called *cosine annealing*. Compared to exponential scheduling, cosine annealing ensures that the learning rate remains fairly high during most of training, while getting closer to the minimum near the end (see Figure 11-10). All in all, cosine annealing generally performs better. The learning rate at epoch t (zero-indexed) is given by Equation 11-10, where $T_{\mathbf{max}}$ is the maximum number of epochs.

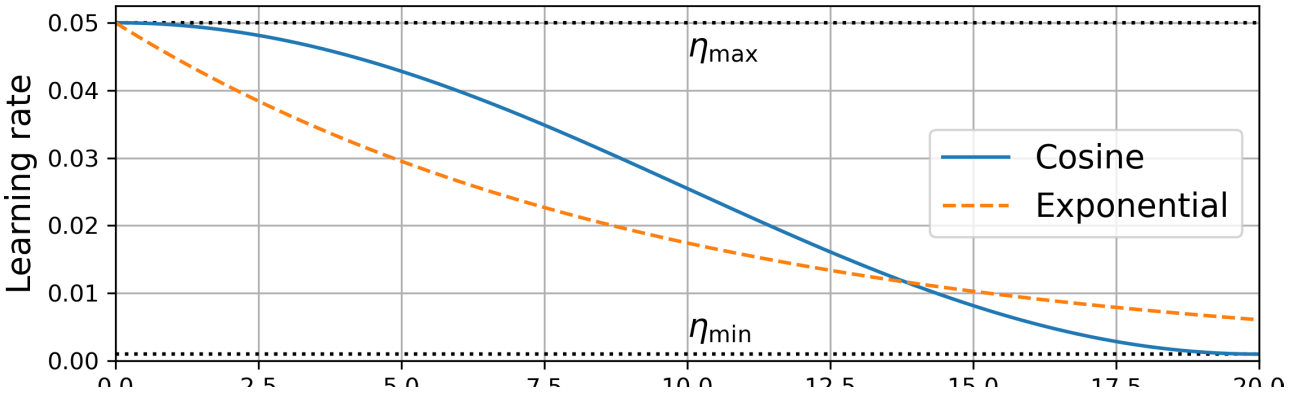

Figure 11-10. Cosine annealing learning schedule

Equation 11-10. Cosine annealing equation

PyTorch includes the `CosineAnnealingLR` scheduler, which you can create as follows (`T_max` is $T_{\mathbf{max}}$ and `eta_min` is $\eta_{\mathbf{min}}$). You can then use it just like the `ExponentialLR` scheduler:

```
cosine_scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(
    optimizer, T_max=20, eta_min=0.001)
```

One problem with cosine annealing is that you have to set two new hyperparameters, $T_{\mathbf{max}}$ and $\eta_{\mathbf{min}}$, and it's not easy to know in advance how many epochs to train and when to stop decreasing the learning rate. This is why I generally prefer to use the performance scheduling technique.

Performance Scheduling

Performance scheduling, also called *adaptive scheduling*, is implemented by PyTorch's `ReduceLROnPlateau` scheduler: it keeps track of a given metric during training—typically the validation loss—and if this metric stops improving for some time, it multiplies the learning rate by some factor. This scheduler has quite a few hyperparameters, but the default values work well for most of them. You may occasionnally need to tweak the following (see the documentation for information on the other hyperparameters):

`mode`

If the tracked metric must be maximized (such as the validation accuracy), then you must set the `mode` to `'max'`. The default is `'min'`, which is fine if the tracked metric must be minimized (such as the validation loss).

`patience`

The number of consecutive steps (typically epochs) to wait for improvement in the monitored metric before reducing the learning rate. It defaults to 10, which is generally fine. If each epoch is very long, then you may want to reduce this value.

`factor`

The factor by which the learning rate will be multiplied whenever the monitored metric fails to improve for too long. It defaults to 0.1, again a reasonable default, but perhaps a bit small in some cases.

For example, let's implement performance scheduling based on the validation accuracy (i.e.,

which we want to maximize):

```
[...]  # build the model and optimizer
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
  optimizer, mode="max", patience=2, factor=0.1)
```

The training loop needs to be tweaked again because we must evaluate the desired metric at each epoch (in this example, we are using the `evaluate_tm()` function that we defined in Chapter 10), and we must then pass the result to the scheduler's `step()` method:

```
metric = torchmetrics.Accuracy(task="multiclass", num_classes=10).to(device)
for epoch in range(n_epochs):
    for X_batch, y_batch in train_loader:
        [...]  # the rest of the training loop remains unchanged
    val_metric = evaluate_tm(model, valid_loader, metric).item()
    scheduler.step(val_metric)
```

Warming Up the Learning Rate

So far, we have always started training with the maximum learning rate. However, this can sometimes cause gradient descent to bounce around randomly at the beginning of training, neither exploding nor making any significant progress. This typically happens with sensitive models, such as recurrent neural networks (Chapter 13), or when using a very large batch size. In such cases, one solution is to "warm up" the learning rate, starting close to zero and gradually increasing the learning rate over a few epochs, up to the maximum learning rate. During this warm-up phase, gradient descent has time to stabilize into a better region of the loss landscape, where it can then make quick progress using a high learning rate.

Why does this work? Well, the loss landscape sometimes resembles the Himalayas: it's very high up and full of gigantic spikes. If you start with a high learning rate, you might jump from one mountain peak to the next for a very long time. If instead you start with a small learning rate, you will just walk down the mountain and valleys and escape the spiky mountain range altogether until you reach flatter lands. From then on, you can use a large learning rate for the rest of your journey, slowing down only toward the end.

A common way to implement learning rate warm up using PyTorch is to use a `LinearLR` scheduler to increase the learning rate linearly over a few epochs. For example, the following scheduler will increase the learning rate from 10% to 100% of the optimizer's original learning rate over 3 epochs (i.e., 10% during the first epoch, 40% during the second epoch, 70% during the third epoch, and 100% after that):

```
warmup_scheduler = torch.optim.lr_scheduler.LinearLR(
    optimizer, start_factor=0.1, end_factor=1.0, total_iters=3)
```

If you would like more flexibility, you can write your own custom function and wrap it in a `LambdaLR` scheduler. For example, the following scheduler is equivalent to the `LinearLR` scheduler we just defined:

```
warmup_scheduler = torch.optim.lr_scheduler.LambdaLR(
    optimizer,
    lambda epoch: (min(epoch, 3) / 3) * (1.0 - 0.1) + 0.1)
```

You must then insert `warmup_scheduler.step()` at the beginning of each epoch, and make sure you deactivate the scheduler(s) you are using for the rest of training during the warm-up phase. And that's all!

```
for epoch in range(n_epochs):
    warmup_scheduler.step()
    for X_batch, y_batch in train_loader:
        [...]  # the rest of the training loop is unchanged
    if epoch >= 3:  # deactivate other scheduler(s) during warmup
        scheduler.step(val_metric)
```

In short, you pretty much always want to cool down the learning rate at the end of training, and you may also want to warm it up at the beginning if gradient descent needs a bit of help getting started. But are there any cases where you may want to tweak the learning rate in the middle of training? Well yes, there are, for example, if gradient descent gets stuck in a local optimum or a high plateau. Gradient descent could remain stuck here for a long time, or even forever. Luckily, there's a way to escape this trap: just increase the learning rate for a little while.

You could spend your time staring at the learning curves during training, and manually interrupting it to tweak the learning rate when needed, but you probably have better things to do. Alternatively, you could implement a custom scheduler that monitors the validation metric—much like the `ReduceLROnPlateau` scheduler—and increases the learning rate for a while if the validation metric is stuck in a bad plateau. For this, you could subclass the `LRScheduler` base class. This is beyond the scope of this book, but you can take inspiration from the `ReduceLROnPlateau` scheduler's source code (and get a little bit of help from your favorite AI assistant). But a much simpler option is to use the cosine annealing with warm restarts learning schedule. Let's look at it now.

Cosine Annealing with Warm Restarts

Cosine annealing with warm restarts was introduced in a 2016 paper by Ilya Loshchilov and Frank Hutter.[30] This schedule just repeats the cosine annealing schedule over and over again. Since the learning rate regularly shoots back up, this schedule allows gradient descent to escape local optima and plateaus automatically. The authors recommend starting with a fairly short round of cosine annealing, but then doubling $T_{\mathbf{max}}$ after each round (see Figure 11-11). This allows gradient descent to do a lot of quick explorations at the start of training, while also taking the time to properly optimize the model later during training, possibly escaping a

plateau or two along the way.

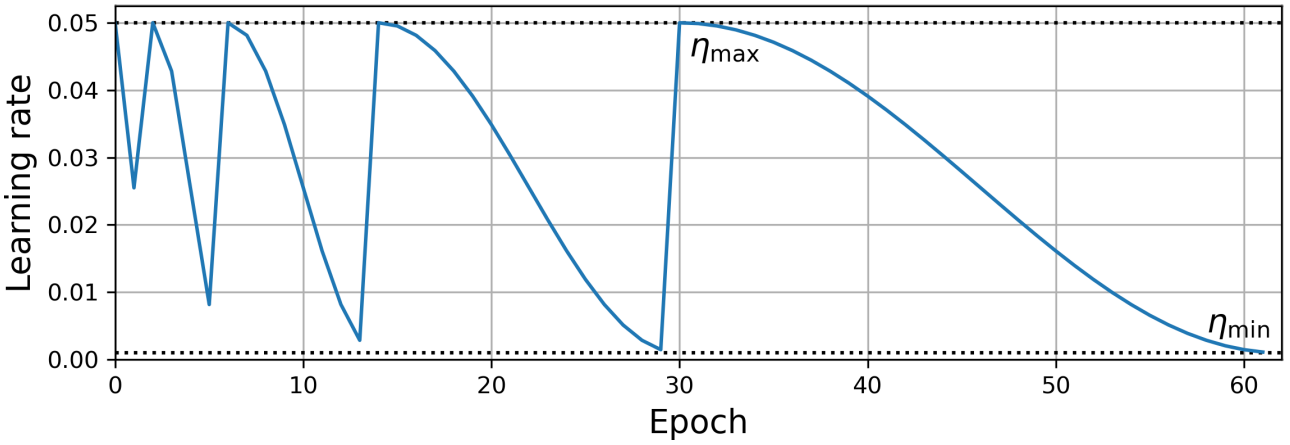

Figure 11-11. Cosine annealing with warm restarts

Conveniently, PyTorch includes a `CosineAnnealingWarmRestarts` scheduler. You must set `T_0`, which is the value of T_{max} for the first round of cosine annealing. You may also set `T_mult` to 2 if you want to double T_{max} at each round (the default is 1, meaning T_{max} stays constant and all rounds have the same length). Finally, you can set `eta_min` (it defaults to 0):

```
cosine_repeat_scheduler = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(
    optimizer, T_0=2, T_mult=2, eta_min=0.001)
```

1cycle Scheduling

Yet another popular learning schedule is *1cycle*, introduced in a 2018 paper by Leslie Smith.[31] It starts by warming up the learning rate, starting at η_0 and growing linearly up to η_1 halfway through training. Then it decreases the learning rate linearly down to η_0 again during the second half of training, finishing the last few epochs by dropping the rate down by several orders of magnitude (still linearly). The maximum learning rate η_1 is chosen using the same approach we used to find the optimal learning rate, and the initial learning rate η_0 is usually 10 times lower. When using a momentum, we start with a high momentum first (e.g., 0.95), then drop it down to a lower momentum during the first half of training (e.g., down to 0.85, linearly), and then bring it back up to the maximum value (e.g., 0.95) during the second half of training, finishing the last few epochs with that maximum value. Smith did many experiments showing that this approach was often able to speed up training considerably and reach better performance. For example, on the popular CIFAR10 image dataset, this approach reached 91.9% validation accuracy in just 100 epochs, compared to 90.3% accuracy in 800 epochs through a standard approach (using the same neural network architecture). This feat was dubbed *super-convergence*. PyTorch implements this schedule in the `OneCycleLR` scheduler.

TIP

If you are not sure which learning schedule to use, 1cycle can be a good default, but I tend to have more luck with performance scheduling. If you run into instabilities at the start of training, try adding learning rate warm-up. And if training gets stuck on plateaus, try cosine annealing with warm restarts.

We have now covered the most popular learning schedules, but PyTorch offers a few extra schedulers (e.g., a polynomial scheduler, a cyclic scheduler, a scheduler that makes it easy to chain other schedulers, and a few more), so make sure to check out the documentation.

Now let's move on to one final topic before we complete this chapter on deep learning training techniques: regularization. Deep learning is highly prone to overfitting, so regularization is key!

Avoiding Overfitting 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 typically have tens of thousands of parameters, sometimes even millions or billions. This gives them an incredible amount of freedom and means they can fit a huge variety of complex datasets. But this great flexibility also makes the network prone to overfitting the training set. Regularization is often needed to prevent this.

We already implemented one of the best regularization techniques in Chapter 4: early stopping. Moreover, even though batch-norm and layer-norm were designed to solve the unstable gradients problems, they also act like pretty good regularizers. In this section we will examine other popular regularization techniques for neural networks: ℓ_1 and ℓ_2 regularization, dropout, MC dropout, and max-norm regularization.

ℓ_1 and ℓ_2 Regularization

Just like you did in Chapter 4 for simple linear models, you can use ℓ_2 regularization to constrain a neural network's connection weights, and/or ℓ_1 regularization if you want a sparse model (with many weights equal to 0). As we saw earlier (when discussing the AdamW optimizer), ℓ_2 regularization is mathematically equivalent to weight decay when using an `SGD` optimizer (with or without momentum), so if that's the case you can implement ℓ_2 regularization by simply setting the optimizer's `weight_decay` argument. For example, here is how to apply ℓ_2 regularization to the connection weights of a PyTorch model trained using `SGD`, with a regularization factor of 10^{-4}:

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.05, weight_decay=1e-4)
[...]  # use the optimizer normally during training
```

If instead you are using an Adam optimizer, you should switch to AdamW and set the `weight_decay` argument. This is not exactly equivalent to ℓ_2 regularization, but as we saw earlier it's pretty close and it works better.

Note that weight decay is applied to every model parameter, including bias terms, and even parameters of batch-norm and layer-norm layers. Generally that's not a big deal, but penalizing these parameters does not contribute much to regularization and it may sometimes negatively impact training performance. So how can we apply weight decay to some model parameters and not others? One approach is to implement ℓ_2 regularization manually, without relying on the optimizer's weight decay feature. For this, you must tweak the training loop to manually compute the ℓ_2 loss based only on the parameters you want, and add this ℓ_2 loss to the main loss:

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.05)
params_to_regularize = [
    param for name, param in model.named_parameters()
    if not "bias" in name and not "bn" in name]
for epoch in range(n_epochs):
    for X_batch, y_batch in train_loader:
        [...]  # the rest of the training loop is unchanged
        main_loss = loss_fn(y_pred, y_batch)
        l2_loss = sum(param.pow(2.0).sum() for param in params_to_regularize)
        loss = main_loss + 1e-4 * l2_loss
        [...]
```

Another approach is to use PyTorch's *parameter groups* feature, which lets the optimizer apply different hyperparameters to different groups of model parameters. So far, we have always created optimizers by passing them the full list of model parameters: PyTorch automatically put them all in a single parameter group, sharing the same hyperparameters. Instead, we can pass a list of dictionaries to the optimizer, each with a `"params"` entry containing a list of parameters, and (optionally) some hyperparameter key/value pairs specific to this group of parameters. The group-specific hyperparameters take precedence over the optimizer's global hyperparameters. For example, let's create an optimizer with two parameter groups: the first group will contain all the parameters we want to regularize and it will use weight decay, while the second group will contain all the bias terms and BN parameters, and it will not use weight decay at all.

```
params_bias_and_bn = [
    param for name, param in model.named_parameters()
    if "bias" in name or "bn" in name]
optimizer = torch.optim.SGD([
    {"params": params_to_regularize, "weight_decay": 1e-4},
    {"params": params_bias_and_bn},
    ], lr=0.05)
[...]  # use the optimizer normally during training
```

TIP

Parameter groups also allow you to apply different learning rates to different parts of your model. This is most common for transfer learning, when you want new layers to be updated faster than reused ones.

Now how about ℓ_1 regularization? Well unfortunately PyTorch does not provide any helper for this, so you need to implement it manually, much like we did for ℓ_2 regularization. This means tweaking the training loop to compute the ℓ_1 loss and adding it to the main loss:

```
l1_loss = sum(param.abs().sum() for param in params_to_regularize)
loss = main_loss + 1e-4 * l1_loss
```

That's all there is to it! Now let's move on to Dropout, which is one of the most popular regularization techniques for deep neural networks.

Dropout

Dropout was proposed in a paper[32] by Geoffrey Hinton et al. in 2012 and further detailed in a 2014 paper[33] by Nitish Srivastava et al., and it has proven to be highly successful: many state-of-the-art neural networks use dropout, as it gives them a 1%–2% accuracy boost. 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 11-12). The hyperparameter *p* is called the *dropout rate*, and it is typically set between 10% and 50%: closer to 20%–30% in recurrent neural nets (see Chapter 13), and closer to 40%–50% in convolutional neural networks (see Chapter 12). After training, neurons don't get dropped anymore. And that's all (except for a technical detail we will discuss shortly).

It's surprising at first that this destructive technique works at all. Would a company perform better if its employees were told to toss a coin every morning to decide whether to go to work? Well, who knows; perhaps it would! The company would be forced to adapt its organization; it could not rely on any single person to work 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 companies, 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 all 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 better.

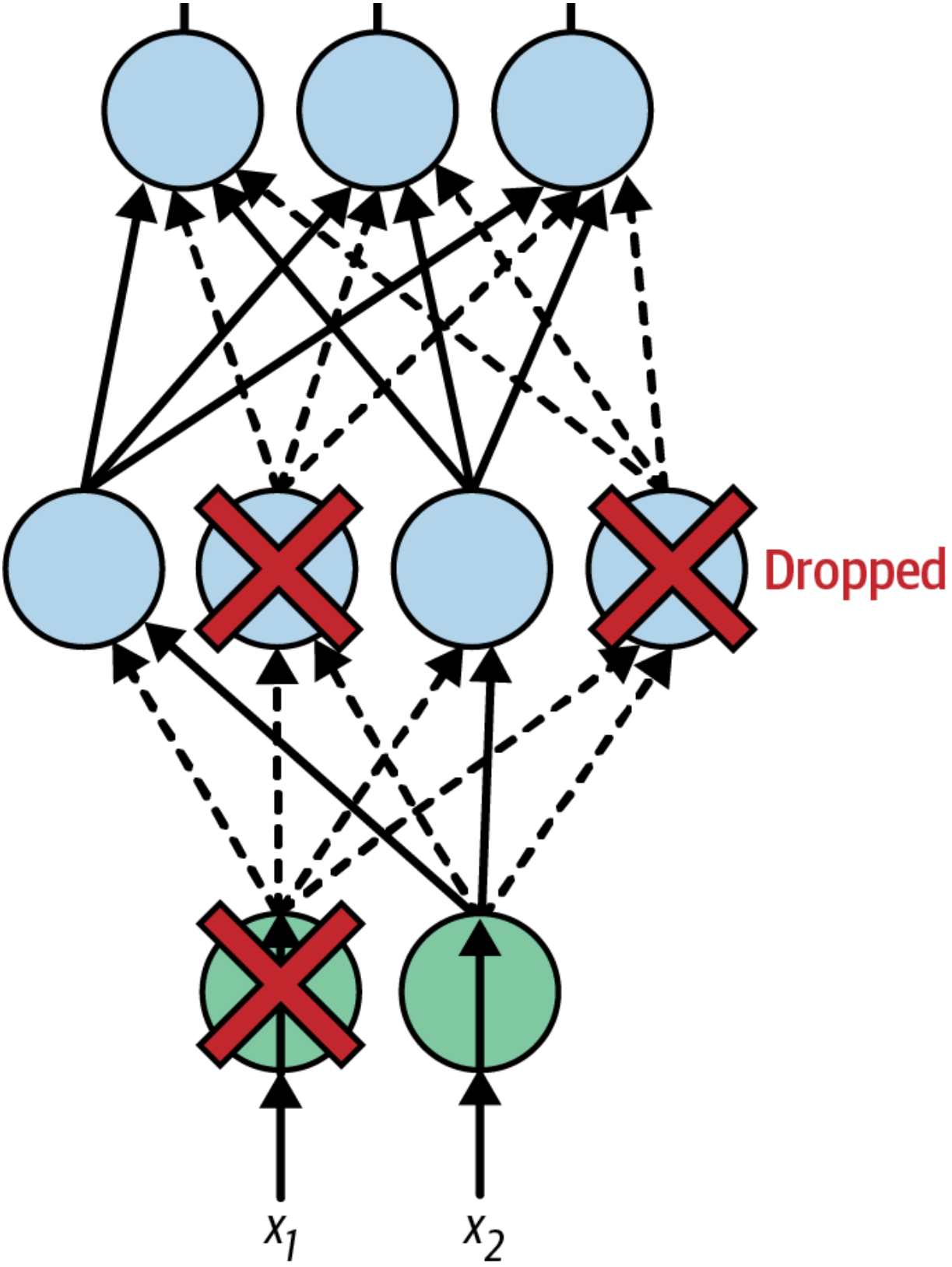

Figure 11-12. With dropout regularization, at each training iteration a random subset of all neurons in one or more layers—except the output layer—are "dropped out"; these neurons output 0 at this iteration (represented by the dashed arrows)

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 are a total of 2^N possible networks (where N is the total number of droppable 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 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 because 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.

TIP

Higher layers, which learn more complex feature combinations, benefit more from dropout because they are more prone to overfitting. So you can usually apply dropout only to the neurons of the top hidden layers (e.g., one to three hidden layers). However, you should avoid dropping the output neurons, as this would be like changing the task during training: it wouldn't help.

There is one small but important technical detail. Suppose p = 75%: on average only 25% of all neurons are active at each step during training. This means that after training, each neuron receives four times more inputs than during training, on average. This discrepancy is so large that the model is unlikely to work well. To avoid this issue, a simple solution is to multiply the inputs by 4 during training, which is the same as dividing them by 25%. More generally, we need to divide the inputs by the *keep probability* (1 – p) during training.

To implement dropout using PyTorch, you can use the `nn.Dropout` layer. It's important to switch to training mode during training, and to evaluation mode during evaluation (just like for batch norm). In training mode, the layer randomly drops some inputs (setting them to 0) and divides the remaining inputs by the keep probability. In evaluation mode, it does nothing at all; it just passes the inputs to the next layer. The following code applies dropout regularization before every `nn.Linear` layer, using a dropout rate of 0.2:

```
model = nn.Sequential(
    nn.Flatten(),
    nn.Dropout(p=0.2), nn.Linear(1 * 28 * 28, 100), nn.ReLU(),
    nn.Dropout(p=0.2), nn.Linear(100, 100), nn.ReLU(),
    nn.Dropout(p=0.2), nn.Linear(100, 100), nn.ReLU(),
    nn.Dropout(p=0.2), nn.Linear(100, 10)
).to(device)
```

WARNING

Since dropout is only active during training, comparing the training loss and the validation loss 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).

If you observe that the model is overfitting, you can increase the dropout rate. Conversely, 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 apply dropout to the last few hidden layers, so you may want to try this if full dropout is too strong.

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

TIP

If you want to regularize a self-normalizing network based on the SELU activation function (as discussed earlier), you should use *alpha dropout*: 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. PyTorch implements it in the `nn.AlphaDropout` layer.

Monte Carlo (MC) Dropout

In 2016, a paper[34] by Yarin Gal and Zoubin Ghahramani added a few more good reasons to use dropout:

- First, the paper established a profound connection between dropout networks (i.e., neural networks containing `Dropout` layers) and approximate Bayesian inference,[35] giving dropout a solid mathematical justification.
- Second, the authors introduced 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. It also provides a much better measure of the model's uncertainty, and it can be implemented in just a few lines of code.

This description of MC dropout sounds like some "one weird trick" clickbait, so let me explain: it is just like regular dropout, except it is active not only during training, but also during evaluation. This means that the predictions are always a bit random (hence the name Monte Carlo). But instead of making a single prediction, you make many predictions and average them out. It turns out that this produces better predictions than the original model.

Below is a full implementation of MC dropout, using the model we trained in the previous section to make predictions for a batch of images:

```
model.eval()
for module in model.modules():
    if isinstance(module, nn.Dropout):
        module.train()

X_new = [...]  # some new images, e.g., the first 3 images of the test set
X_new = X_new.to(device)

torch.manual_seed(42)
with torch.no_grad():
    X_new_repeated = X_new.repeat_interleave(100, dim=0)
    y_logits_all = model(X_new_repeated).reshape(3, 100, 10)
    y_probas_all = torch.nn.functional.softmax(y_logits_all, dim=-1)
    y_probas = y_probas_all.mean(dim=1)
```

Let's go through this code:

- First, we switch the model to evaluation mode as we always do before making predictions, but this time we immediately switch all the dropout layers back to training mode, so they will behave just like during training (i.e., randomly dropping out some of their inputs). In other words, we convert the dropout layers to MC dropout layers.
- Next we load a new batch of images `X_new`, and we move it to the GPU. In this example, let's assume `X_new` contains three images.
- We then use the `repeat_interleave()` method to create a batch containing 100 copies of each image in `X_new`. The images are repeated along the first dimension (`dim=0`) so `X_new_repeated` has a shape of `[300, 1, 28, 28]`.
- Next, we pass this big batch to the model, which predicts 10 logits per image, as usual. This tensor's shape is `[300, 10]`, but we reshape it to `[3, 100, 10]` to group the predictions for each image. Remember that the dropout layers are active, which means that there's some variability across the predictions, even for copies of the same image.
- Then we convert these logits to estimated probabilities using the softmax function.
- Lastly, we compute the mean over the second dimension (`dim=1`) to get the average estimated probability for each class and each image, across all 100 predictions. The result is a tensor of shape `[3, 10]`. These are our final predictions:[36]

```
>>> y_probas.cpu().numpy().round(2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.03, 0.  , 0.97],
       [0.99, 0.  , 0.  , 0.  , 0.  , 0.01, 0.  , 0.  , 0.  ],
       [0.5 , 0.03, 0.03, 0.22, 0.02, 0.  , 0.2 , 0.  , 0.  , 0.  ]],
      dtype=float32)
```

WARNING

Rather than converting the logits to probabilities and then computing the mean probabilities, you may be tempted to do the reverse: first average over the logits and *then* convert the mean logits to probabilities. This is faster but it does not properly reflect the model's uncertainty, so it tends to produce overconfident models.

MC dropout tends to improve the reliability of the model's probability estimates. This means that it's less likely to be confidently wrong, making it safer (you don't want a self-driving car confidently ignoring a stop sign). It's also useful when you're interested in the top *k* classes, not just the most likely. Additionally, you can take a look at the standard deviation of each class probability:

```
>>> y_std = y_probas_all.std(dim=1)
>>> y_std.cpu().numpy().round(2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.06, 0.  , 0.06],
       [0.02, 0.  , 0.  , 0.  , 0.  , 0.02, 0.  , 0.  , 0.  ],
       [0.16, 0.03, 0.04, 0.15, 0.03, 0.  , 0.09, 0.  , 0.  , 0.  ]],
      dtype=float32)
```

There's a standard deviation of 0.06 for the probability estimate of class 9 (ankle boot) for the first image. This adds a grain of salt to the estimated probability of 97% for this class: in fact, the model is really saying "mmh, I'm guessing over 90%". If you were building a risk-sensitive system (e.g., a medical or financial system), you may want to consider only the predictions with both a high estimated probability *and* a low standard deviation.

NOTE

The number of Monte Carlo samples you use (100 in this example) is a hyperparameter you can tweak. The higher it is, the more accurate the predictions and their uncertainty estimates are, but also the slower the predictions are. Moreover, above a certain number of samples, you will notice little improvement. Your job is to find the right trade-off among latency, throughput, and accuracy, depending on your application.

If you want to train an MC dropout model from scratch rather than reuse an existing dropout model, you should probably use a custom `McDropout` module rather than using `nn.Dropout` and hacking around with `train()` and `eval()`, as this is a bit brittle (e.g., it won't play nicely with the evaluation function). Here is a three-line implementation:

```
class McDropout(nn.Dropout):
    def forward(self, input):
        return F.dropout(input, self.p, training=True)
```

In short, MC dropout is a great technique that boosts dropout models and provides better uncertainty estimates. And of course, since it is just regular dropout during training, it also acts like a regularizer.

Max-Norm Regularization

Another fairly popular regularization technique for neural networks is called *max-norm regularization*: for each neuron, it constrains the weights $\mathbf{w}$ of the incoming connections such that $\| \mathbf{w} \|_2 \leq r$, where r is the max-norm hyperparameter and $\| \cdot \|_2$ is the ℓ_2 norm.

Reducing r increases the amount of regularization and helps reduce overfitting. Max-norm regularization can also help alleviate the unstable gradients problems (if you are not using batch-norm or layer-norm).

Rather than adding a regularization loss term to the overall loss function, max-norm regularization is typically implemented by computing $\| \mathbf{w} \|_2$ after each training step and rescaling $\mathbf{w}$ if needed ($\mathbf{w} \leftarrow \mathbf{w}\, r / \| \mathbf{w} \|_2$). Here's a common way to implement this in PyTorch:

```
def apply_max_norm(model, max_norm=2, epsilon=1e-8, dim=1):
    with torch.no_grad():
        for name, param in model.named_parameters():
            if 'bias' not in name:
```

```
                actual_norm = param.norm(p=2, dim=dim, keepdim=True)
                target_norm = torch.clamp(actual_norm, 0, max_norm)
                param *= target_norm / (epsilon + actual_norm)
```

This function iterates through all of the model's weight matrices (i.e., all parameters except for the bias terms), and for each one of them it uses the `norm()` method to compute the ℓ_2 norm of each row (`dim=1`). A `nn.Linear` layer has weights of shape [*number of neurons*, *number of inputs*], so using `dim=1` means that we will get one norm per neuron, as desired. Then the function uses `torch.clamp()` to compute the target norm for each neuron's weights: this creates a copy of the `actual_norm` tensor, except that all values greater than `max_norm` are replaced by `max_norm` (this corresponds to *r* in the previous equation). Lastly, we rescale the weight matrix so that each column ends up with the target norm. Note that the smoothing term `epsilon` is used to avoid division by zero in case some columns have a norm equal to zero.

Next, all you need to do is call `apply_max_norm(model)` in the training loop, right after calling the optimizer's `step()` method. And of course you probably want to fine-tune the `max_norm` hyperparameter.

TIP

When using max-norm with layers other than `nn.Linear`, you may need to tweak the `dim` argument. For example, when using convolutional layers (see Chapter 12), you generally want to set `dim=[1, 2, 3]` to limit the norm of each convolutional kernel.

Practical Guidelines

In this chapter we have covered a wide range of techniques, and you may be wondering which ones you should use. This depends on the task, and there is no clear consensus yet, but I have found the configuration in Table 11-3 to work fine in most cases, without requiring much hyperparameter tuning. That said, please do not consider these defaults as hard rules!

Table 11-3. Default DNN configuration

Hyperparameter	Default value
Kernel initializer	He initialization
Activation function	ReLU if shallow; Swish if deep
Normalization	None if shallow; batch-norm or layer-norm if deep
Regularization	Early stopping; weight decay if needed
Optimizer	Nesterov accelerated gradients or AdamW
Learning rate schedule	Performance scheduling or 1cycle

If the network is a simple stack of dense layers, then it can self-normalize, and you should use

If the network is a simple stack of dense layers, then it can self-normalize, and you should use the configuration in Table 11-4 instead (don't forget to normalize the input features!).

Table 11-4. DNN configuration for a self-normalizing net

Hyperparameter	Default value
Kernel initializer	LeCun initialization
Activation function	SELU
Normalization	None (self-normalization)
Regularization	Alpha dropout if needed
Optimizer	Nesterov accelerated gradients
Learning rate schedule	Performance scheduling or 1cycle

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 use pretraining on an auxiliary task if you have a lot of labeled data for a similar task.

While the previous guidelines should cover most cases, there are some exceptions:

- If you need a sparse model, you can use ℓ_1 regularization. You can also try zeroing out the smallest weights after training (for example, using the `torch.nn.prune.l1_unstructured()` function). This will break self-normalization, so you should use the default configuration in this case.
- If you need a low-latency model (one that performs lightning-fast predictions), you may need to use fewer layers; use a fast activation function such as `nn.ReLU`, `nn.LeakyReLU`, or `nn.Hardswish`; and fold the batch-norm and layer-norm layers into the previous layers after training. Having a sparse model will also help. Finally, you may want to reduce the float precision from 32 bits to 16 or even 8 bits. Appendix B covers several techniques to make models smaller and faster, including reduced precision models, mixed precision models, and quantization.
- 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.

Over the last three chapters, we have learned what artificial neural nets are, how to build and train them using Scikit-Learn and PyTorch, and a variety of techniques that make it possible to train deep and complex nets. In the next chapter, all of this will come together as we dive into one of the most important applications of deep learning: computer vision.

Exercises

1. What is the problem that Glorot initialization and He initialization aim to fix?
2. Is it OK to initialize all the weights to the same value as long as that value is selected ran-

domly using He initialization?

3. Is it OK to initialize the bias terms to 0?
4. In which cases would you want to use each of the activation functions we discussed in this chapter?
5. What may happen if you set the `momentum` hyperparameter too close to 1 (e.g., 0.99999) when using an `SGD` optimizer?
6. Name three ways you can produce a sparse model.
7. Does dropout slow down training? Does it slow down inference (i.e., making predictions on new instances)? What about MC dropout?
8. Practice training a deep neural network on the CIFAR10 image dataset:
   a. Load CIFAR10 just like you loaded the FashionMNIST dataset in Chapter 10, but using `torchvision.datasets.CIFAR10` instead of `FashionMNIST`. The dataset is composed of 60,000 32 × 32–pixel color images (50,000 for training, 10,000 for testing) with 10 classes.
   b. Build a DNN with 20 hidden layers of 100 neurons each (that's too many, but it's the point of this exercise). Use He initialization and the Swish activation function (using `nn.SiLU`). Since this is a classification task, you will need an output layer with one neuron per class.
   c. Using NAdam optimization and early stopping, train the network on the CIFAR10 dataset. Remember to search for the right learning rate each time you change the model's architecture or hyperparameters.
   d. Now try adding batch-norm and compare the learning curves: is it converging faster than before? Does it produce a better model? How does it affect training speed?
   e. Try replacing batch-norm with SELU, and make the necessary adjustments to ensure the network self-normalizes (i.e., standardize the input features, use LeCun normal initialization, make sure the DNN contains only a sequence of dense layers, etc.).
   f. Try regularizing the model with alpha dropout. Then, without retraining your model, see if you can achieve better accuracy using MC dropout.
   g. Retrain your model using 1cycle scheduling and see if it improves training speed and model accuracy.

Solutions to these exercises are available at the end of this chapter's notebook, at *https://homl.info/colab-p*.

[1] Xavier Glorot and Yoshua Bengio, "Understanding the Difficulty of Training Deep Feedforward Neural Networks", *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics* (2010): 249–256.

[2] Here's an analogy: if you set a microphone amplifier's volume 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 saying. 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.

[3] Kaiming He et al., "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet

Classification," *Proceedings of the 2015 IEEE International Conference on Computer Vision* (2015): 1026–1034.

4 A PyTorch issue (#18182) has been open since 2019 to update the weight initialization to use the current best practices.

5 Andrew Saxe et al., "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks," ICLR (2014).

6 A dead neuron may come back to life if its inputs evolve over time and eventually return within a range where the ReLU activation function gets a positive input again. For example, this may happen if gradient descent tweaks the neurons in the layers below the dead neuron.

7 Bing Xu et al., "Empirical Evaluation of Rectified Activations in Convolutional Network," arXiv preprint arXiv:1505.00853 (2015).

8 Djork-Arné Clevert et al., "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)," *Proceedings of the International Conference on Learning Representations*, arXiv preprint (2015).

9 Günter Klambauer et al., "Self-Normalizing Neural Networks", *Proceedings of the 31st International Conference on Neural Information Processing Systems* (2017): 972–981.

10 Dan Hendrycks and Kevin Gimpel, "Gaussian Error Linear Units (GELUs)", arXiv preprint arXiv:1606.08415 (2016).

11 A function is convex if the line segment between any two points on the curve never lies below the curve. A monotonic function only increases, or only decreases.

12 Prajit Ramachandran et al., "Searching for Activation Functions", arXiv preprint arXiv:1710.05941 (2017).

13 Noam Shazeer, "GLU Variants Improve Transformer", arXiv preprint arXiv:2002.05202 (2020).

14 Yann Dauphin et al., "Language Modeling with Gated Convolutional Networks", arXiv preprint arXiv:1612.08083 (2016).

15 Diganta Misra, "Mish: A Self Regularized Non-Monotonic Activation Function", arXiv preprint arXiv:1908.08681 (2019).

16 So et al., "Primer: Searching for Efficient Transformers for Language Modeling", arXiv preprint arXiv:2109.08668 (2021).

17 Sergey Ioffe and Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", *Proceedings of the 32nd International Conference on Machine Learning* (2015): 448–456.

18 Jimmy Lei Ba et al., "Layer Normalization", arXiv preprint arXiv:1607.06450 (2016).

19 Razvan Pascanu et al., "On the Difficulty of Training Recurrent Neural Networks", *Proceedings of the 30th International Conference on Machine Learning* (2013): 1310–1318.

20 Boris T. Polyak, "Some Methods of Speeding Up the Convergence of Iteration Methods", *USSR*

Computational Mathematics and Mathematical Physics 4, no. 5 (1964): 1–17.

[21] Yurii Nesterov, "A Method for Unconstrained Convex Minimization Problem with the Rate of Convergence $O(1/k^2)$," *Doklady AN USSR* 269 (1983): 543–547.

[22] John Duchi et al., "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization", *Journal of Machine Learning Research* 12 (2011): 2121–2159.

[23] 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*, video: *https://homl.info/58*). Amusingly, since the authors did not write a paper to describe the algorithm, researchers often cite "slide 29 in lecture 6e" in their papers.

[24] Diederik P. Kingma and Jimmy Ba, "Adam: A Method for Stochastic Optimization", arXiv preprint arXiv:1412.6980 (2014).

[25] Timothy Dozat, "Incorporating Nesterov Momentum into Adam", (2016).

[26] Ilya Loshchilov, and Frank Hutter, "Decoupled Weight Decay Regularization", arXiv preprint arXiv:1711.05101 (2017).

[27] Ashia C. Wilson et al., "The Marginal Value of Adaptive Gradient Methods in Machine Learning", *Advances in Neural Information Processing Systems* 30 (2017): 4148–4158.

[28] The *Jacobian matrix* contains all the first-order partial derivatives of a function with multiple parameters and multiple outputs: one column per parameter, and one row per output. When training a neural net with gradient descent, there's a single output—the loss—so the matrix contains a single row, and there's one column per model parameter, so it's a $1 \times n$ matrix. The *Hessian matrix* contains all the second-order derivatives of a single-output function with multiple parameters: for each model parameter it contains one row and one column, so it's an $n \times n$ matrix. The informal names *Jacobians* and *Hessians* refer to the elements of these matrices.

[29] V. Gupta et al., "Shampoo: Preconditioned Stochastic Tensor Optimization", arXiv preprint arXiv:1802.09568 (2018).

[30] Ilya Loshchilov and Frank Hutter, "SGDR: Stochastic Gradient Descent With Warm Restarts", arXiv preprint arXiv:1608.03983 (2016).

[31] Leslie N. Smith, "A Disciplined Approach to Neural Network Hyper-Parameters: Part 1—Learning Rate, Batch Size, Momentum, and Weight Decay", arXiv preprint arXiv:1803.09820 (2018).

[32] Geoffrey E. Hinton et al., "Improving Neural Networks by Preventing Co-Adaptation of Feature Detectors", arXiv preprint arXiv:1207.0580 (2012).

[33] Nitish Srivastava et al., "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", *Journal of Machine Learning Research* 15 (2014): 1929–1958.

[34] Yarin Gal and Zoubin Ghahramani, "Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning", *Proceedings of the 33rd International Conference on Machine Learning* (2016): 1050–1059.

[35] 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*.

[36] We must move the data to the CPU using the `cpu()` method before converting it to a NumPy array. We need a NumPy array to call the `round()` method with a number of decimals, since the tensor's `round()` method sadly does not have this feature.