5 Modern training techniques

This chapter covers

- Improving long-term training using a learning rate schedule
- Improving short-term training using optimizers
- Combining learning rate schedules and optimizers to improve any deep model's results
- Tuning network hyperparameters with Optuna

At this point, we have learned the basics of neural networks and three types of architectures: fully connected, convolutional, and recurrent. These networks have been trained with an approach called stochastic gradient descent (SGD), which has been in use since at least the 1960s. Newer improvements to learning the parameters of our network have been invented since then, like momentum and learning rate decay, which can improve *any neural network for any problem* by converging to better solutions in fewer updates. In this chapter, we learn about some of the most successful and widely used variants of SGD in deep learning.

Because there are no constraints on a neural network, they often end up with complex optimization problems with many local minima, as shown figure 5.1. The minima appear to reduce the loss by the most possible, which would mean our model has learned, but the global context reveals that other minima may be better. Not all local minima are "bad"; if a minimum is sufficiently close to the global minimum, we will probably get good results. But some local minima are just not going to give us a model with the predictive accuracy we want. Because gradient descent makes its decisions "locally," once we hit a local minimum, we have no idea how "good" that minimum is or which direction to head if we want to try to find a new one.

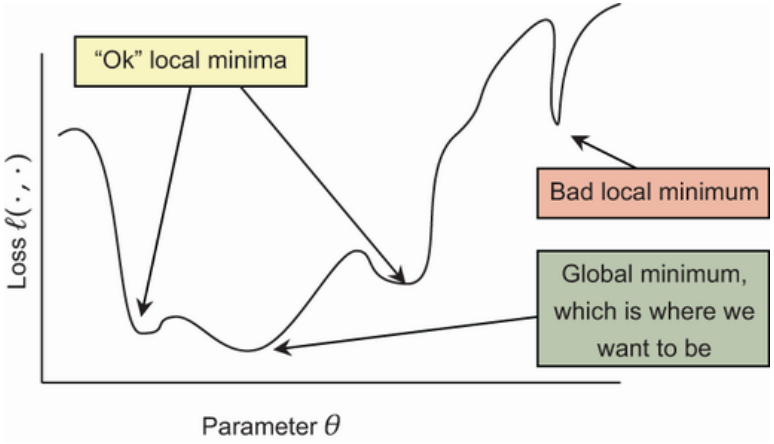


Figure 5.1 Example of a loss landscape that a neural network might encounter. The loss on the y-axis is what we want to minimize, but there are many local minima. There is one global minima that is the best solution, with two local minima that are almost as good—and one bad minima that will not work well.

In this chapter, we learn about *learning rate schedules* and *optimizers* that can help us reach better minima faster, resulting in neural networks that reach higher accuracies in fewer epochs. Again, this applies to *every* network you will ever train and are ubiquitous in modern designs.

To do this, we do a quick review of gradient descent and see how it can be broken up into two parts: the gradient and the learning rate *schedule*. PyTorch has special classes for both of these, so we write a new `train_network` function that will replace the `train_simple_network` function we have seen through the book that incorporates both of these abstractions. These new abstractions will be covered in section 5.1. Then we start filling in the details of these abstractions and how they work, first with the learning rate schedule in section 5.2 followed by gradient update strategies in section 5.3.

There is a *second* optimization problem we have been ignoring, which is how to choose all these hyperparameters like the number of layers, the number of neurons in each layer, and so on. In section 5.4, we are going to learn about how to select hyperparameters using a tool called Optuna. Optuna has special features that make it great at setting hyperparameters even when we have a lot of them and to avoid the full cost of training many different networks to try the parameters.

This chapter is a bit longer because we explain why these newer techniques for using gradients reduce the number of epochs you need to get a good solution, where many other materials just jump right to using these improved methods. This is also important because researchers are constantly working on improved strategies, and the extra work we do to go from the original SGD to modern approaches will help you appreciate the potential for future improvements and to reason about why they may be helping. The Optuna section in particular may seem odd because we will not use this topic in other chapters, as it can get in the way of learning about other techniques. But Optuna and its approach to tuning hyperparameters is a critical real-world skill in deep learning that will help keep you productive and building more accurate solutions.

5.1 Gradient descent in two parts

So far, we have been using and thinking about learning via gradient descent as one monolithic equation and process. We pick a loss function ℓ and a network architecture f, and gradient descent just goes and updates the weights. Because we want to improve how gradient descent works, we first need to recognize and understand its two components. By recognizing that these two components have different behaviors, we can develop strategies for improving each one to get our networks to learn more accurate solutions in fewer epochs. Let's start with a quick review of how gradient descent works.

Remember that everything we do in deep learning works by treating the network as one giant function $f_{\boldsymbol{\theta}}(\boldsymbol{x})$, where we need to use the gradient (∇) with respect to the parameters (Θ) of f to adjust its weights. So we perform

$$\Theta_{t+1} = \Theta_t - \eta \cdot \underbrace{\nabla_{\Theta_t} \ell(f_{\Theta_t}(\boldsymbol{x}), y)}_{\text{the gradient } g^t}$$

This is called *gradient descent* and is the basis for how all modern training of neural networks is done. However, there is room for significant improvement in how we perform this critical step. We use $\boldsymbol{g}^t$ as shorthand for the gradient since we talk about it a *lot* in this section. Notice that we included this ^{t} superscript as if it was part of a sequence. That's because

when we learn, we get a new gradient for every batch of data we process, so our model is learning from a sequence of gradients. We use this to our advantage later.

With this shorthand, it becomes clearer that there are just two parts to this process: the gradient g^t and the learning rate η, as we can see here. Those are the only two parts we can alter to try to improve this:

The new parameters θ_{t+1} are equal to	the old parameters θ_t minus the
learning rate times	the gradient over the batch

$$\theta_{t+1} = \theta_t - \eta \cdot g^t$$

5.1.1 Adding a learning rate schedule

Let's discuss the problems with the earlier update equation. First, there is an issue with the learning rate η. We choose *one* learning rate for *every batch of every epoch of training*. This can be an unreasonable expectation. Consider, as an analogy, a train traveling from one city to another. The train does not travel *full speed* and then instantaneously come to a stop once it reaches the destination. Instead, the train slows down as it approaches the destination. Otherwise, the train would go barreling past the destination.

So, the first type of improvement we can discuss is to alter the learning rate η as a function of how far along we are in the optimization process (t). Let's call this function L and give it an *initial* learning rate η_0 and the current iteration step t as inputs:

$$\underbrace{\Theta_{t+1}}_{\text{new parameters}} = \underbrace{\Theta_t}_{\text{old parameters}} - \underbrace{L(\eta_0, t)}_{\text{learning rate schedule}} \cdot \underbrace{g^t}_{\text{gradient}}$$

We review the details of how we define $L(\eta_0,t)$ shortly. Right now, the important thing is to understand that we have created an abstraction for altering the learning rate, and this abstraction is called a *learning rate schedule*. We can use this to replace L with different schedules based on

our needs or problem. Before we start showing this with code, we need to discuss a second part of the update equation that is tightly coupled with the learning rate schedule $L(\cdot,\cdot)$ in PyTorch.

5.1.2 Adding an optimizer

This part does not have a satisfying name; PyTorch calls it the optimizer, but that also describes the process as a whole. Still, we will use that name since it's the mostcommon—and it's how we *use* the gradient $\boldsymbol{g}^t$. All of the information and learning come from $\boldsymbol{g}^t$; it controls what the network learns and how well it learns. The learning rate η simply controls how quickly we follow that information. But the gradient $\boldsymbol{g}^t$ is only as good as the data we use to train the model. If our data is noisy (and it almost always is), our gradients will also be noisy.

These noisy gradients and how they impact learning are shown in figure 5.2 with three other situations. One is the ideal gradient descent path, which almost never happens. The other two are real problems. Say, for example, that we sometimes get a gradient $\boldsymbol{g}^t$ that is just *too large*. This can happen when we add hundreds of layers to our network and is a common problem called an *exploding gradient*. Mathematically, that would be a situation where $\|\boldsymbol{g}^t\|_2 \to \infty$. If we use that gradient, we may take a far larger step than we ever intended (even with a small η), which can degrade performance or even completely prevent us from learning. The opposite situation occurs too; the gradient could be too small $\|\boldsymbol{g}^t\|_2 \to 0$, causing us to make no progress toward training (even with a large η). This is called a *vanishing* gradient and can occur in almost any architecture but is particularly common when we use the $\tanh(\cdot)$ and $\sigma(\cdot)$ activation functions.[1] When looking at these four situations, remember that optimization also means *minimization*, so the process of going from a high value (red) down to a low value (green) is literally how a neural network "learns."

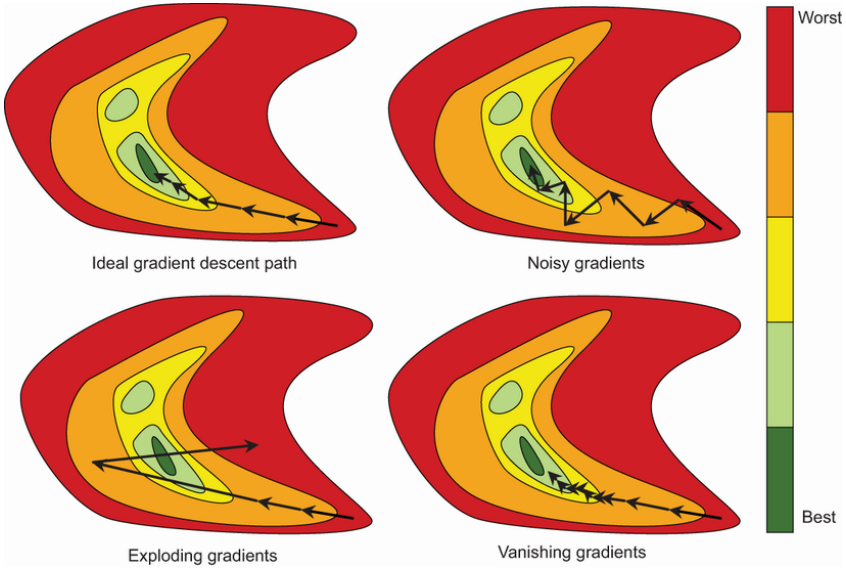


Figure 5.2 A toy example showing three scenarios for gradient descent. Each is a contour plot of a 2D optimization going from red (high values, bad) to dark green (low values, good). The ideal case is at the top left, where each gradient heads exactly toward the solution. The top right shows *noisy* gradients that cause the descent to head in not quite the correct direction, requiring more steps. The bottom left shows exploding gradients, where we begin to take steps that are far too large and head away from the solution. The bottom right shows vanishing gradients, where the gradient becomes so small that we have to take an excessive number of steps to get to the solution.

In these situations, naively using the raw gradient g^t could mislead us. Again, we probably want to introduce an abstraction that takes g^t in as input and do something more clever to avoid these situations. We call that function G that *alters* the gradient to make it better behaved and help accelerate the learning. So now we again update our equation and get:

The new parameters θ_{t+1} are equal to the old parameters θ_t minus the learning rate that has adapted during training (so that we slow down or speed up as needed) multiplied by the gradient over the batch, processed by a function $G(\cdot)$ that tries to exploit any similarities to previous gradients.

$$\theta_{t+1} = \theta_t - L(\eta_0, t) \cdot G(g^t)$$

Now we have a new gradient descent equation. It has all the base components as before, plus some extra abstractions to make it more flexible. These two strategies—adjusting the learning rate and gradient—are ubiq-

uitous in modern deep learning. As such, they both have interfaces in PyTorch. So let's define a new version of our `train_simple_network` function we have been using that allows for these two kinds of improvements. We can then use this new function to compare the effect of the new techniques and continue to use them later.

The interplay between $L(\cdot,\cdot)$ and $G(\cdot)$

The learning rate schedule and gradient updates ultimately tackle similar goals, so why have both instead of picking one? You can think of them as working on two time scales. The learning rate schedule $L(\cdot,\cdot)$ operates *once per epoch* to adjust the global rate of progress. The gradient update functions $G(\cdot)$ operate on *every* batch, so if you have millions of data points, you may be calling $G(\cdot)$ millions of times but calling $L(\cdot,\cdot)$ at most a few hundred times. So you can think of these things as balancing between long-term and short-term strategies to minimize a function. Like most things in life, it's better to play the balance than focus purely on short- or long-term goals.

5.1.3 Implementing optimizers and schedulers

PyTorch provides two interfaces for us to implement the $L(\cdot,\cdot)$ and $G(\cdot)$ functions we have described. First is the `torch.optim.Optimizer` class that you have already been using. We've been using a naive `SGD` optimizer thus far. We start replacing the `SGD` object, but with other optimizers that use the same interface, so almost no code has to change. The new class is `_LRScheduler`, which has several options for us to choose from. To implement our `train_network` function, we'll make just a few modifications to the `train_simple_network` code. Figure 5.3 shows the high-level process, which has only one new step in yellow, with a slight change to the update step to use $G(\cdot)$ to denote that we can change how gradients are used.

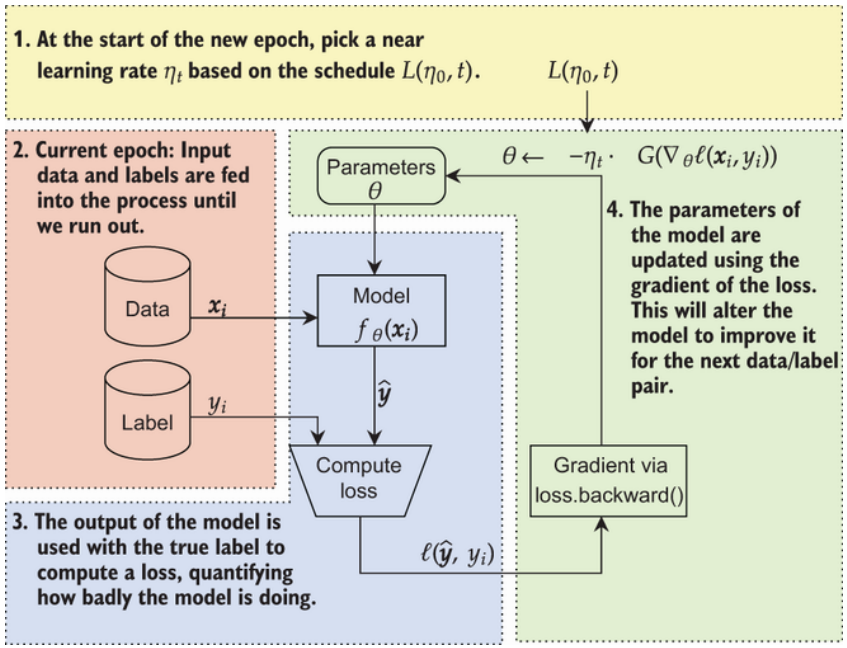


Figure 5.3 Diagram of the new training loop process. The major change is the new step 1 in yellow, showing that we have a learning rate schedule $L(\cdot,\cdot)$. The schedule determines the learning rate η_t used by the process in step 4. Everything else remains the same as before.

Updating the training code

Now let's talk about the three changes we need to make to the `train_simple_network` code. The first thing we change is the function signature so there are two new options available: the optimizer and scheduler. The new signature is shown here, with `optimizer` for $G(\cdot)$ and `lr_schedule` for $L(\cdot,\cdot)$, respectively:

```
def train_network(model, loss_func, train_loader, val_loader=None,
➥ test_loader=None, score_funcs=None, epochs=50, device="cpu",
➥ checkpoint_file=None, optimizer=None, lr_schedule=None):
```

Notice that we set both to have a `None` default value. Specifying a schedule is *always* optional, and you may want to change the schedule used, depending on the task at hand. Some problems take just a few epochs, others take hundreds to thousands, and both of these factors change based on how much data you have. For these reasons, I don't like *requiring* a learning rate schedule to *always* be used. I like to work without one first and then add one based on the problem at hand. However, we must *al-*

ways use some kind of optimizer. So if none is given, we add the following code to use a good default (we get to how it works later in this chapter):

```
if optimizer == None:
optimizer = torch.optim.AdamW(model.parameters()) ❶
```

❶ AdamW is a good default optimizer.

Surprisingly, we are halfway done with the new code. We do not need to make any changes to the `run_epoch` method because it is customary to alter the learning rate after each *epoch* rather than each *batch* (and an epoch is made up of many batches). So after our `run_epoch` function is done, we can add the following code:

```
if lr_schedule is not None:
    if isinstance(lr_schedule, ReduceLROnPlateau):
        lr_schedule.step(val_running_loss)
    else:
        lr_schedule.step() ❶
```

❶ In PyTorch, the convention is to update the learning rate after every epoch.

Again, we learn about `ReduceLROnPlateau` in a moment. It is an idiosyncratic and special member of the learning rate scheduler family that requires an extra argument. Otherwise, we simply call the `step()` function at the end of each epoch, and the learning rate schedule is updated automatically. You may recognize this as the same approach we use for the `Optimizer` class inside of `run_epoch`, which calls `optimizer.step()` at the end of every batch. This is an intentional design choice to make the two closely coupled classes consistent. You can find the complete function definition in the code in the idlmam.py file (**https://github.com/EdwardRaff/Inside-Deep-Learning/blob/main/idlmam.py**).

Using the new training code

That is all it takes to prepare our code for some new and improved learning. Now that we have a new loading function implemented, let's train a

neural network. We use the Fashion-MNIST dataset because it is slightly more challenging while retaining the same size and shape as the original MNIST corpus, which will let us accomplish some testing in a reasonable time.

Here we load Fashion-MNIST, define a multilayer fully connected network, and then train it in a manner equivalent to using our old `train_simple_network` method. To do this, we need slightly more ceremony by specifying the SGD optimizer ourselves:

```
epochs = 50 ❶
B = 256     ❷

train_data = torchvision.datasets.FashionMNIST("./data", train=True,
transform=transforms.ToTensor(), download=True)
test_data = torchvision.datasets.FashionMNIST("./data", train=False,
transform=transforms.ToTensor(), download=True)

train_loader = DataLoader(train_data, batch_size=B, shuffle=True) test_loader = DataLoader(test_dat
```

❶ 50 epochs of training

❷ A respectable average batch size

Now we write some more familiar code, a fully connected network with three hidden layers:

```
D = 28*28    ❶
n = 128      ❷
C = 1        ❸
classes = 10 ❹

fc_model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(D, n),
    nn.Tanh(),
    nn.Linear(n, n),
    nn.Tanh(),
    nn.Linear(n, n),
    nn.Tanh(),
```

```
        nn.Linear(n, classes),
    )
```

❶ How many values are in the input? We use this to help determine the size of subsequent layers.

❷ Hidden layer size

❸ How many channels are in the input?

❹ How many classes are there?

Last, we need to settle on a default starting learning rate η_0. If we do not provide any kind of learning rate schedule $L(\cdot,\cdot)$, then η_0 will be used for every epoch of training. We'll use $\eta_0 = 0.1$, which is more aggressive (read, *large*) than you usually want. I chose this larger value to make it easier to show the impacts of the schedules we can choose from. Under normal use, different optimizers tend to have different preferred defaults, but using $\eta_0 = 0.001$ is usually a good starting point:

```
eta_0 = 0.1
```

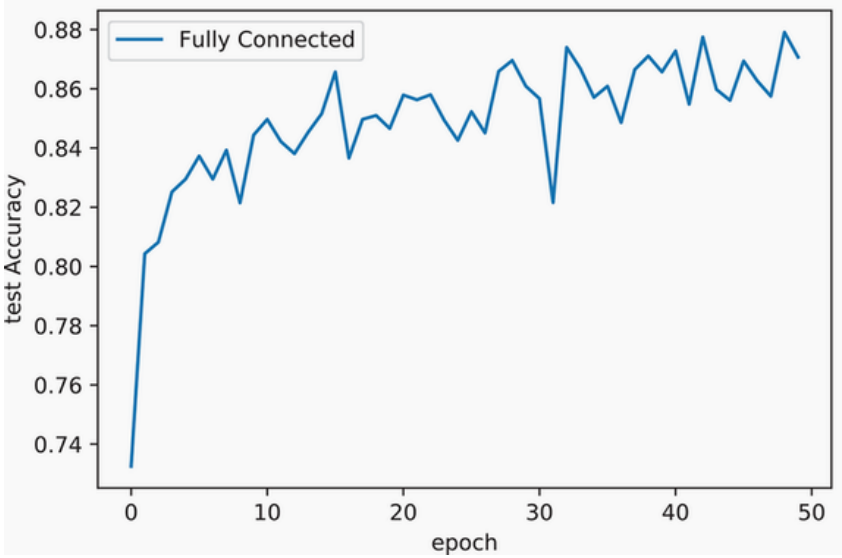
With that in place, we can define a naive `SGD` optimizer the same as we did before. We just need to explicitly call `torch.optim.SGD` and pass it in ourselves, which you can see in the following code. Notice that we set the default learning rate η_0 in the `optimizer`'s constructor. This is the standard process in PyTorch, and any `LRSchedule` object we might use will reach into the `optimizer` object to alter the learning rate:

```
loss_func = nn.CrossEntropyLoss()

fc_results = train_network(fc_model, loss_func, train_loader,
➥ test_loader=test_loader, epochs=epochs,
➥ optimizer=torch.optim.SGD(fc_model.parameters(), lr=eta_0),
➥ score_funcs={'Accuracy': accuracy_score}, device=device)
```

Just like before, we can use seaborn with the return `fc_results` pandas dataframe to quickly plot the results. The following code and output show the kinds of results we get, clearly learning with more training but having

occasional regressions. This is because our learning rate is a bit too aggressive, but it's the kind of behavior you often see on real-world problems with non-aggressive learning rates (like $\eta_0 = 0.001$):

```
    sns.lineplot(x='epoch', y='test Accuracy', data=fc_results, label='Fully
    ➥ Connected')
[12]: <AxesSubplot:xlabel='epoch', ylabel='test Accuracy'>
```



5.2 Learning rate schedules

Now let's talk about ways to implement the learning rate adjustment ($L(\eta_0,t)$) we described earlier. In PyTorch, these are called *learning rate schedulers*, and they take the `optimizer` object as an input because they directly alter the learning rate η used within the `optimizer` object. The high-level approach is shown in figure 5.4. The *only* thing we need to change is the equation used for $L(\eta_0,t)$ to switch between schedules.

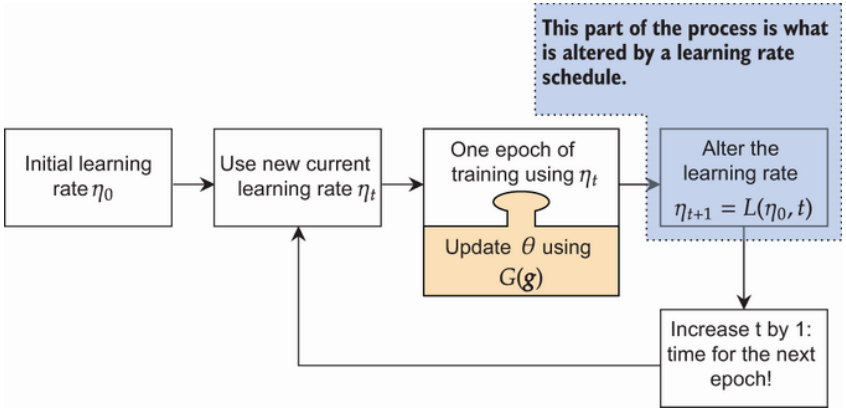


Figure 5.4 General process for every learning rate schedule. We set up before training starts with initial values, do one epoch of training, and then alter η_{t+1}, which is used by the next epoch. The gradient optimizer $G(\cdot)$ is used multiple times per epoch and operates independently of the learning rate schedule.

We will talk about four approaches to adjusting the learning rate that you should be aware of. Each of these schedules has different pros and cons. You may select your schedule to try to stabilize a model that isn't training consistently (accuracy oscillating up and down between epochs), reduce the number of epochs needed to train to save time, or maximize the accuracy of your final model. For most of the content of this book, we are using very few training iterations (10 to 50) to make sure the run in a reasonable amount of time. When you work on a problem in real life, it is common to train for 100 to 500 epochs. The more epochs of training you perform, the bigger the impact a learning rate schedule can have, because there are more opportunities to alter the learning rate. So while we may not use these heavily in the rest of this book, you should still be aware of them.

We first talk about four of the most fundamental learning rate schedules in modern use and the kinds of minimization issues they help resolve, and how. We'll talk about these at an *intuitive* level, as proving them is more math than we want to do in this book. Once we have reviewed the four approaches, we'll do a bake-off to compare their results.

5.2.1 Exponential decay: Smoothing erratic training

The first approach we discuss may not be the most common but is one of the simplest. It is called an *exponential decay rate*. The exponential decay

rate is a good choice if your model behaves erratically, with loss or accuracy increasing and decreasing by large amounts. You can use the exponential decay to help stabilize the training and get a more consistent result. We pick a value $0 < \gamma < 1$ that is multiplied by our learning rate after every epoch. It is defined by the following function, where t represents the current epoch:

$$\eta_t = L_\gamma(\eta_0, t) = \eta_0 \cdot \gamma^t$$

(η_t: current learning rate; η_0: initial learning rate; γ: the decay rate)

PyTorch provides this using the `torch.optim.lr_scheduler.ExponentialLR` class. Because we usually do many epochs, it is important to make sure γ is not set too aggressively: it's a good idea to start with a desired *final* learning rate and call that η_{min}. Then, if you train for a total of T epochs, you can set

$$\gamma = \sqrt[T]{\eta_{min}/\eta_0}$$

to ensure that you pick a value of γ that reaches your desired minimum and results in learning throughout.

For example, say the initial learning rate is $\eta_0 = 0.001$, you want the minimum to be $\eta_{min} = 0.0001$, and you train for $T = 50$ epochs. You need to set $\gamma \approx 0.91201$. The following code simulates this process and shows how to write the code to calculate γ:

```
T=50                                                                    ❶

epochs_input = np.linspace(0, 50, num=50)                               ❷

eta_init = 0.001                                                        ❸
eta_min = 0.0001                                                        ❹

gamma = np.power(eta_min/eta_init,1./T)                                 ❺

effective_learning_rate = eta_init * np.power(gamma, epochs_input)      ❻

sns.lineplot(x=epochs_input, y=eta_init, color='red', label="$\eta_0$")
```

```
ax = sns.lineplot(x=epochs_input, y=effective_learning_rate, color='blue',
➥ label="$\eta_0 \cdot \gamma^t$")
ax.set_xlabel('Epoch')
ax.set_ylabel('Learning Rate')

[13]: Text(0, 0.5, 'Learning rate')
```

❶ Total epochs

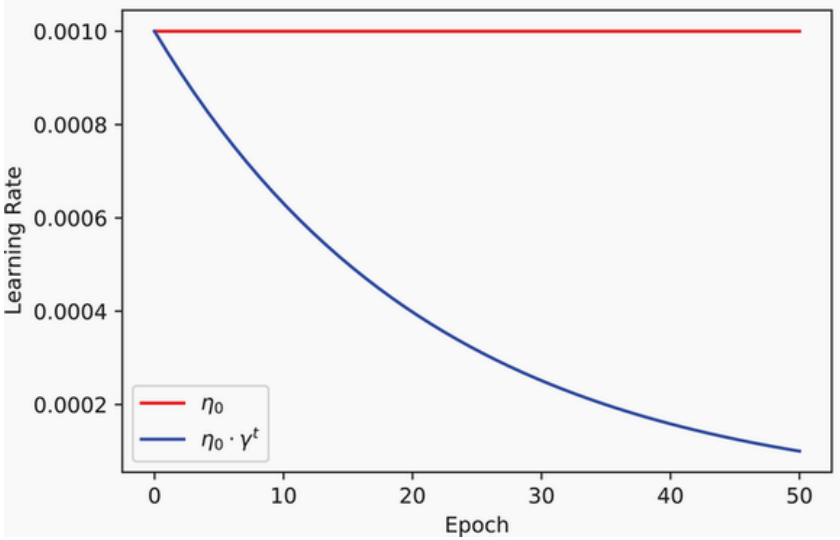
❶ Generates all the t values

❶ Pretend initial learning rate η_0

❶ Pretend desired minimum learning rate η_{min}

❶ Computes the decay rate γ

❶ All the η_t values



The exponential decay rate smoothly and consistently decreases the learning rate every epoch. At a high level, there are many ways to do this. Some people use linear decay ($\eta_0/(\gamma \cdot t)$) instead of exponential ($\eta_0 \cdot \gamma^t$), and a variety of other approaches achieve the same goal; no one can give you a definitive playbook or flowchart to choose between exponential decay

and its related family members. They all follow a similar intuition for why they work, which we will walk through.

In particular, the exponential learning rate schedule helps solve the problem of getting *near* the solution but not quite making it *to* the solution. Figure 5.5 shows how this happens. The black lines show the path that the parameters Θ take as they are altered from one step to the next. The initial weights are random, so we start with a bad set of weights Θ, and initially, updates move us toward the local minima. But as we get closer, we start bouncing around the minimum—sometimes getting closer than a previous step and sometimes moving farther away—which causes the loss to fluctuate up and down.

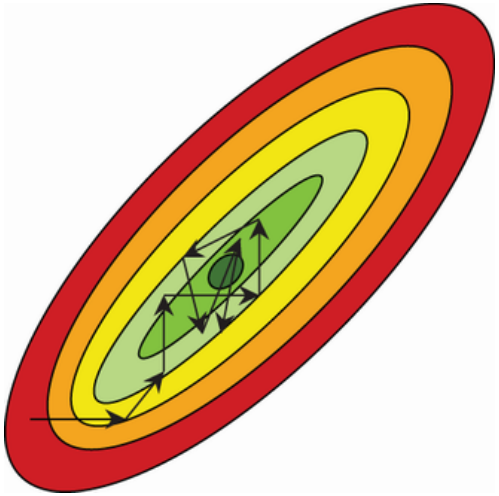


Figure 5.5 Example of the minimization problem that an exponential learning rate helps solve. When you are far from the solution, a large η helps you get to the right *area* faster. But once you are in the right area, η is too large to *reach* the best solution. Instead, it keeps overshooting the local minima (in this case, it's the only minima and so also the global minima).

Returning to our analogy of a train traveling to its destination, going *fast* is great when you are *far* from your goal because it gets you closer faster. But once you are near your destination, it's a good idea to *slow down*. If the station was only 100 feet away and the train was going 100 miles per hour, the train would barrel past the station. You want the train to start slowing down so it can reach a precise location, and that's what the exponential learning rate does; an example is shown in figure 5.6.

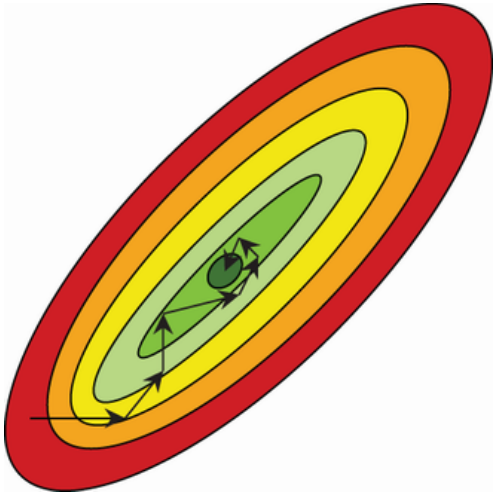


Figure 5.6 Shrinking η at every step causes the optimization to slow down as it gets closer to the destination. This helps it converge to the local minimum instead of bouncing around it.

The trick to using the exponential learning rate well is setting η_{min}. I usually recommend making it 10 to 100 times smaller than η_0. A reduction of 10 to 100× is normal in machine learning and a theme you will see across the schedules we use. The following code shows how we can create the exponential decay schedule by passing in the `optimizer` as an argument at construction. We also use the first line to reset the learned weights to be random so we don't have to keep re-specifying the same neural network over and over again:

```
fc_model.apply(weight_reset)                                          ❶

eta_min = 0.0001                                                      ❷
gamma_expo = (eta_min/eta_0)**(1/epochs)                              ❸
optimizer = torch.optim.SGD(fc_model.parameters(),                    ❹
➥ lr=eta_0)
scheduler = torch.optim.lr_scheduler.                                 ❺
➥ ExponentialLR(optimizer, gamma_expo)

fc_results_expolr = train_network(fc_model, loss_func, train_loader,
➥ test_loader=test_loader, epochs=epochs, optimizer=optimizer,
➥ lr_schedule=scheduler, score_funcs={'Accuracy': accuracy_score},
➥ device=device)
```

❶ Re-randomizes the model weights so we don't need to define the model again

❷ Desired final learning rate η_{min}

❸ Computes γ that results in η_{min}

❹ Sets up the optimizer

❺ Picks a schedule and passes in the optimizer

5.2.2 Step drop adjustment: Better smoothing

The second strategy is an especially popular variant of the exponential decay we just discussed. The *step drop* approach has the same motivation and is also useful to stabilize learning but usually delivers improved accuracy after training. What is the difference between step drop and exponential decay? Instead of constantly adjusting the learning rate ever so slightly, we let it stay fixed for a while and then drop it dramatically just a few times. This is shown in figure 5.7.

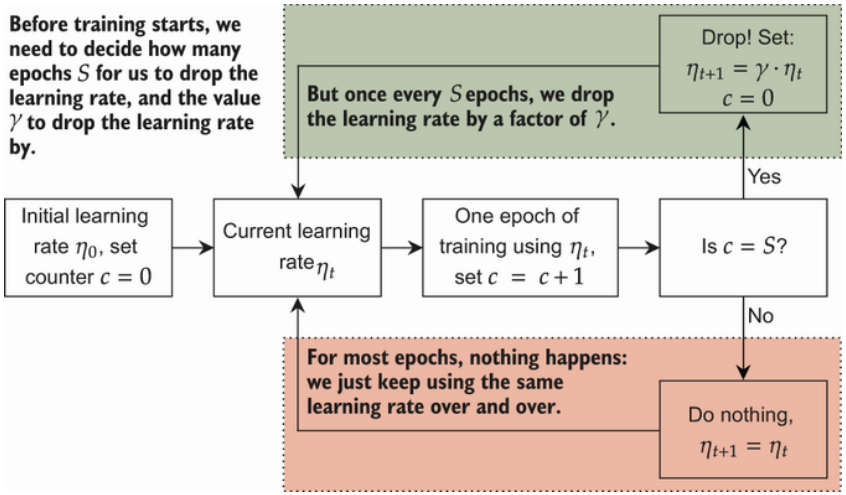
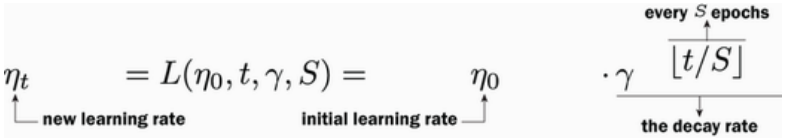


Figure 5.7 The step drop strategy requires us to decide on an initial learning rate, a decay factor γ, and a frequency S. Every S epochs of training, we decrease the learning rate by a factor of γ.

The logic behind this approach is that early in the optimization, we are still far from the solution, so we should go as fast as possible. For the first

S epochs, we head toward the solution at the maximum speed η_0. This is (hopefully) better than exponential decay because the exponential decay starts slowing down *immediately* and is therefore counterproductive (at least in the beginning). Instead, let's keep going at our maximum speed and simply drop the learning rate instantaneously once or twice. That way, we go maximum speed for as long as possible but eventually slow down to converge on the solution.

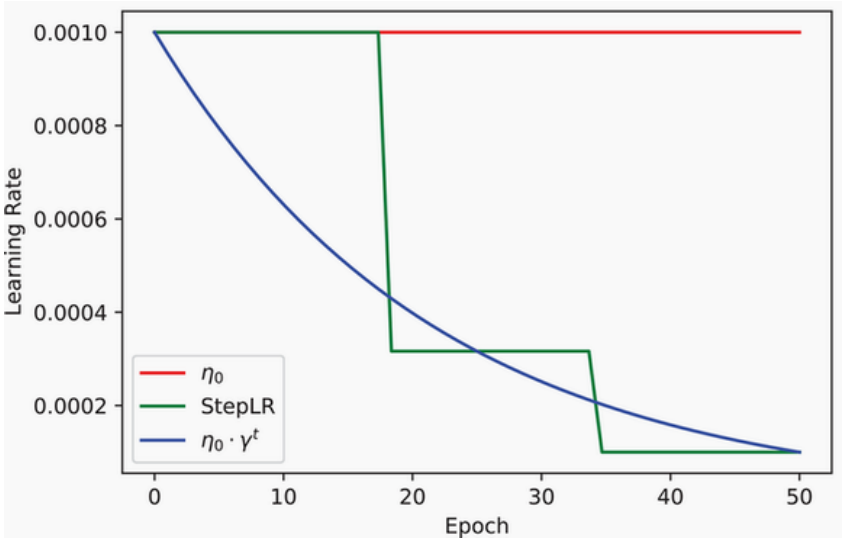
We can also express this strategy in a more mathematical notation. If we want to drop the learning rate every S epochs, we obtain



If you compare this equation to the exponential day, you should notice that they look *almost identical* and are deeply connected. If we set the step drop strategy to drop every epoch ($S = 1$), we get $\gamma^{\lfloor t/1 \rfloor} = \gamma^t$, which is *exactly* exponential decay. So the step drop strategy reduces the frequency at which we decrease the learning rate and balances that out by instead increasing the amount by which we decay.

The same rule of thumb about a 10 to $100\times$ drop from η_0 to η_{min} applies, so we usually set γ to a value in the range of $0.1, 0.5$ and set S such that the learning rate is adjusted only two or three times during training. Again, PyTorch provides this using the `StepLR` class. The following code shows what a StepLR might look like compared to the exponential decay; you can see that it has a higher learning rate for *most* of the epochs but a lower learning rate for longer at the end of training:

```
[15]: Text(0, 0.5, 'Learning Rate')
```



As with all learning rate schedules, we pass in the `optimizer` at construction time. This is shown in the following code, where we train with four drops in learning rate, each by a factor of $\gamma = 0.3$. The last drop happens in the last few epochs, and we have far fewer total epochs, which means the first three are much more important. Although four drops equal a $1/0.3^4 \approx 123\times$ reduction in the learning rate, most of the training happens with a $1/0.3^3 \approx 37\times$ drop (or less):

```
fc_model.apply(weight_reset)

optimizer = torch.optim.SGD(fc_model.parameters(), lr=eta_0)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, ❶
epochs//4, gamma=0.3)

fc_results_steplr = train_network(fc_model, loss_func, train_loader,
➥ test_loader=test_loader, epochs=epochs, optimizer=optimizer,
➥ lr_schedule=scheduler, score_funcs={'Accuracy': accuracy_score},
➥ device=device)
```

❶ Tells it to step down by a factor f γ every epochs //4 so this happens four times total.

5.2.3 Cosine annealing: Greater accuracy but less stability

The next learning rate is odd but effective: *cosine annealing*. Cosine annealing has a different logic and strategy than exponential decay and the step learning rate. The latter two only decrease the learning rate, but cosine annealing decreases and increases the learning rate. This approach is very effective for getting the best possible results but does not provide the same degree of stabilization, so it might not work on poorly behaved datasets and networks.

Cosine annealing also has an initial learning rate η_0 and a minimum rate η_{min}; the difference is that we alternate between the minimum and maximum learning rates. The math follows this equation, where T_{max} is the number of epochs between cycles:

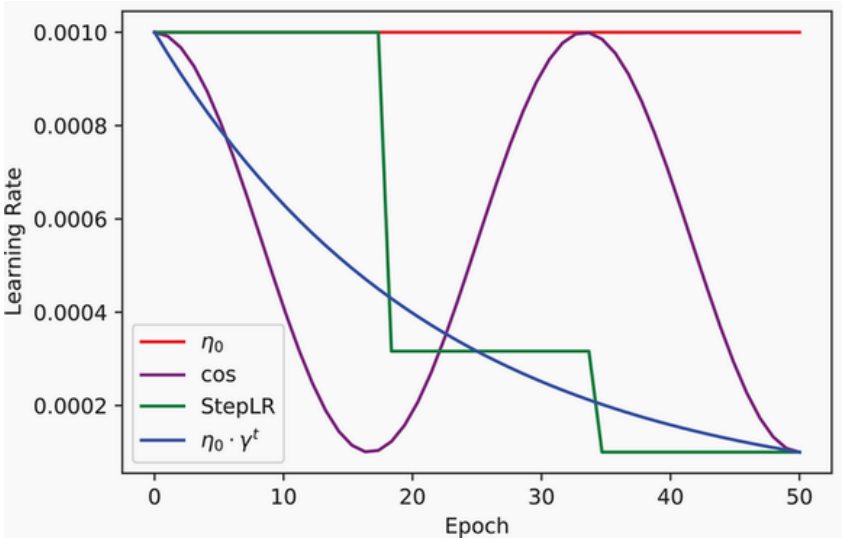
$$\underbrace{\eta_t}_{\text{current learning rate}} = \underbrace{\eta_{min}}_{\text{minimum learning rate}} + \underbrace{(\eta_0 - \eta_{min})}_{\text{gap between min and max rates}} \cdot \underbrace{\frac{1}{2}\left(1 + \cos\left(\frac{t}{T_{max}} \cdot \pi\right)\right)}_{\text{oscillate between min and max}}$$

The cos term fluctuates up and down like the cosine function normally does, and we rescale its cosine to have a maximum at η_0 instead of 1 and a minimum at η_{min} instead of –1. PyTorch provides this with the `CosineAnnealingLR` class. The value T_{max} becomes a new hyperparameter for the model. I like to set T_{max} so we have 10 to 50 oscillation dips total, and we want to always end on a dip (to end by slowing down for the destination). For example, if we want S dips, we use $T_{max} = T/(S \cdot 2-1)$. The following code shows two dips of oscillation in the cosine schedule by setting $T_{max} = T/(2 \cdot 2-1)$:

```
cos_lr = eta_min +
0.5*(eta_init-eta_min)*(1+np.cos(epochs_input/(T/3.0)*np.pi)) ❶
sns.lineplot(x=epochs_input, y=eta_init, color='red', label="$\eta_0$")
sns.lineplot(x=epochs_input, y=cos_lr, color='purple', label="$\cos$")
sns.lineplot(x=epochs_input, y=[eta_init]*18+[eta_init/3.16]*16 + *16, color='green',
➥ label="StepLR") % ax = sns.lineplot(x=epochs_input,     y=effective_learning_rate, color='blu
➥ label="$\eta_0 \cdot \gamma^t$")
ax.set_xlabel('Epoch')
ax.set_ylabel('Learning Rate')
```

```
[17]: Text(0, 0.5, 'Learning Rate')
```

❶ Computes the cosine schedule η_t for every value of t



At first glance, this cosine schedule does not make sense. Why would we want to fluctuate the learning rate up and down? It makes more sense when we remember that neural networks are not *convex*. A convex function has only one local minimum, and *every* gradient leads us to the optimal solution. But neural networks are non-convex and can have many local minima, which may not be a good solution. If our model first heads towards one of these local minima, and we decrease only the learning rate, we may get stuck in this sub-optimal area.

Figure 5.8 shows what can go wrong when training a neural network that has multiple minima. The network starts in a bad position (because initial weights Θ are random) and makes progress toward a local minimum. It's an OK solution, but a better one exists nearby. Optimization is hard, and we have no way of knowing how good our minima are or how many exist; so we end up stuck in a sub-optimal position.

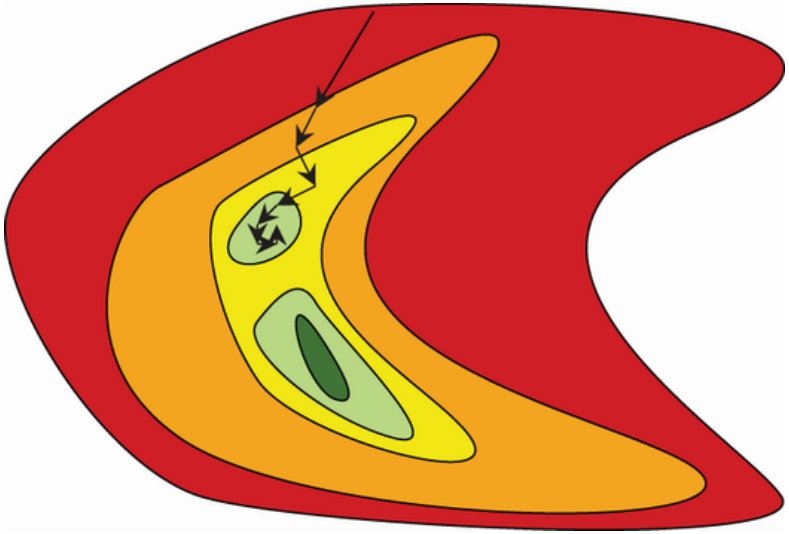


Figure 5.8 If we get an unlucky starting point (because it leads to a sub-optimal minimum), our search may take us to the sub-optimal minimum (light green). If we shrink the learning rate, our search may get stuck. The only way to escape is to increase the learning rate.

There is a sliver of hope, though. Through experimentation (and hard math that we aren't going to look at), there is a common phenomenon that sub-optimal local minima exist on the way toward a better minimum (figure 5.8 shows this). So if we shrink and then later increase the learning rate η, we can give our model a chance to escape the current local minimum and find a new alternative minimum. The larger learning rate hopefully gets us to a new and better area, and the decay again allows us to hone in on a more refined solution. Figure 5.9 shows how this works with cosine annealing.

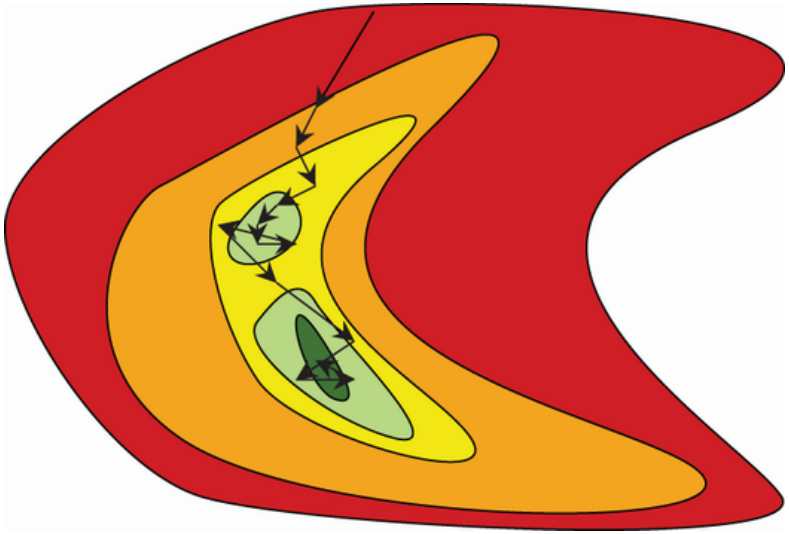


Figure 5.9 Gradient descent first takes us to the sub-optimal minimum, but as the learning ratein-creases again, we bounce out of that area and toward a better solution. The search continues to bounce around the better solution until the cosine schedule again decreases the learning rate, al-lowing the model to converge toward a more precise answer.

You may wonder what stops us from bouncing out of a better solution into a worse one. Technically, nothing stops that from happening. However, researchers have developed a lot of theory about neural net-works that gives us some confidence that bouncing out of a good solution is unlikely. The gist of these results is that gradient descent likes to find wide basins as good solutions, and it would be difficult for the optimiza-tion to bounce out because the solution is wide. This gives us a little more confidence that this crazy cosine schedule is a good idea, and empirically it has performed well on a number of tasks (image classification, natural language processing, and many more). In fact, cosine annealing has been so successful that there have been dozens of proposed alternatives.

The code to implement this approach again is very similar to the previous learning rate schedules. This example uses `epochs//3` for the T_{max} value, which means it performs two dips. I always stick with an odd number for the divisor of epochs so the learning ends with a dip down to a small learning rate. It is also important that the number of dips be no more than one-quarter the number of epochs—otherwise, the learning rate will fluctuate too much from one epoch to the next. Here's the code:

```
fc_model.apply(weight_reset)

optimizer = torch.optim.SGD(fc_model.parameters(), lr=eta_0)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, epochs//3,
➥ eta_min=0.0001) ❶

fc_results_coslr = train_network(fc_model, loss_func, train_loader,
➥ test_loader=test_loader, epochs=epochs, optimizer=optimizer,
➥ lr_schedule=scheduler, score_funcs={'Accuracy': accuracy_score},
➥ device=device)
```

❶ Tells the cosine to go down, then up, and then down (that's three). If we were doing more than 10 epochs, I would push this higher.

Many friends and flavors of cosine annealing

Cosine annealing is an effective learning rate schedule on its own, but its original design was for a slightly different purpose. You may recall from your prior work or training in machine learning that *ensembling* is a great way to improve predictiveaccuracy on a task: train a bunch of models, and average your predictions to get a final, better, answer. But training 20 to 100 neural networks just to average themtogether is very expensive. That's where cosine annealing comes in. Instead of training 20 models, train *1* model with cosine annealing. Every time the learning rate bottoms out in a dip, take a copy of those weights and use that as one of your models. So if you want an ensemble of 20 networks and you are doing T epochs of training, you use $T_{max} = T/(20 \cdot 2 - 1)$. This results in exactly 20 dips in the learning rate.

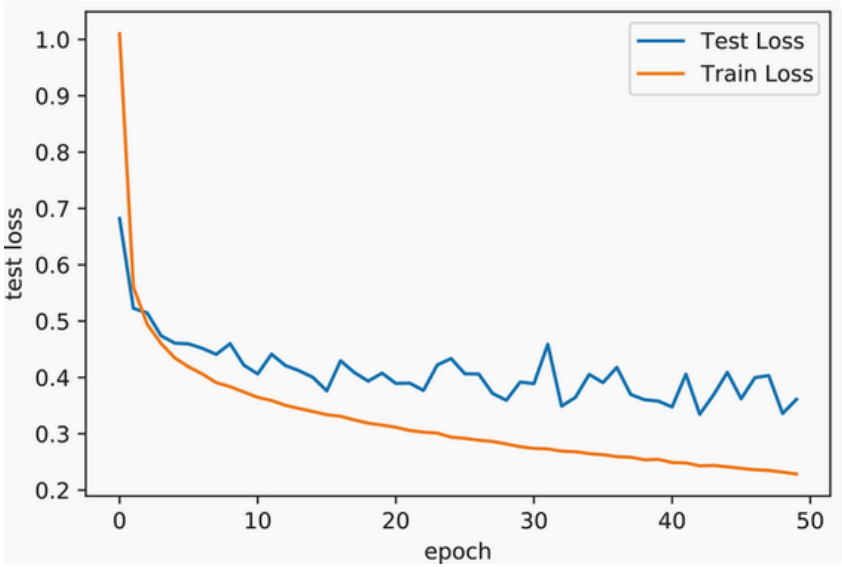
Since PyTorch 1.6, an even fancier version of this idea called *stochastic weight averaging* (SWA) is built in (**https://pytorch.org/docs/stable/optim.html#stochastic-weight-averaging**). With SWA, you can *average* the parameters Θ from each dip, giving you *one* model that is closer in accuracy to an ensemble of models. This is a rare best of both worlds, where you get the benefits of ensembling at the cost and storage of just one model.

5.2.4 Validation plateau: Data-based adjustments

None of the learning rate schedules we have talked about so far rely on external information. All you need to know is an initial learning rate η_0, a minimum rate η_{min}, and not much else. None of them use information about *how well the learning is going*. But what information do we have available to use, and how should we use it? The primary information we have is the loss ℓ for each epoch of training, and we can add it into our approach to try to maximize the accuracy of our final model. This is what the plateau-based strategy does, and it will often get you the best possible accuracy for your final model. Let's look at the training and testing set loss from the baseline network `fc_model`:

```
sns.lineplot(x='epoch', y='test loss',
➥ data=fc_results, label='Test Loss')
sns.lineplot(x='epoch', y='train loss',
➥ data=fc_results, label='Train Loss')
```

```
[19]: <AxesSubplot:xlabel='epoch', ylabel='test loss'>
```



The training loss consistently heads down, which is normal since neural networks rarely underfit the data. When our model trains on a specific set of data and sees the same data over and over (that's what an epoch is:

seeing all the data once), the model gets *unreasonably* good at predicting the right answer for the training data. That's classic overfitting, which occurs to different degrees depending on how big your network is. We expect the training loss to always go down, so it isn't a good source of information about how well the learning is going. But if we look at the test loss, we see that it does *not* consistently get better with each epoch. The test loss begins to stabilize or plateau after a certain point.

If the test loss has stabilized, that would be a good time to reduce the learning rate.[2] If we are already at an optimal location, reducing the learning rate will not hurt anything. If we are bouncing around a better solution, reducing the learning rate can help us improve based on the same logic we used to justify exponential decay. But now we are reducing the learning rate *when doing so appears necessary based on the data*, rather than based on a fixed arbitrary schedule.

This is the idea behind the *reduce learning rate on plateau* schedule, implemented by the `ReduceLROnPlateau` class in PyTorch. This is also why the `step` function needs the last validation loss passed in as an argument in our code, which looks like `lr_schedule.step(results["val loss"][-1])`. That way, the `ReduceLROnPlateau` class can compare the current loss to the recent history of losses.

`ReduceLROnPlateau` has three primary arguments to control how well it works. First is `patience`, which tells us how many epochs of no improvement we want to see before reducing the learning rate. A value of 10 is common, as you don't want to drop the learning rate prematurely. Instead, you want some consistent evidence that no more progress is being made before changing speed.

The second argument is `threshold`, which defines what counts as not improving. A threshold of exactly 0 says that we are improving if there is *any* amount lower than the previous best loss in the past `patience` epochs. That might be *too* pedantic because the loss could be decreasing by tiny but meaningless amounts every epoch. If we use a threshold of 0.01, we are saying that the loss needs to decrease by more than 1% to count as an improvement. But that could also be a little too loose: when you train for hundreds of epochs, you could be making slow but steady

progress, and *reducing* the learning rate is unlikely to speed up convergence. It can take some work to setting this *just* right, so we stick the default value of `0.0001`, but this is a hyperparameter worth playing with if you want to squeeze out the maximum possible performance.

The last parameter is the `factor` by which we want to drop the learning rate η every time we determine that we have hit a plateau. This works the same way γ does for the `StepLR` class. Again, values in the range of 0.1 to 0.5 are all reasonable choices.

But we need to recognize one critical thing before using the `ReduceLROnPlateau` schedule: you can't use the test data to choose when to alter the learning rate. Doing so *will* cause overfitting because you are using information about the test data to make decisions and then using the test data to evaluate your results. Instead, you need to have training, validation, and testing splits. Our normal code has been using validation as the test data, which was fine because we were not *looking* at the validation data to make decisions while training. The process of `ReduceLROnPlateau` with this important detail is summarized in figure 5.10.

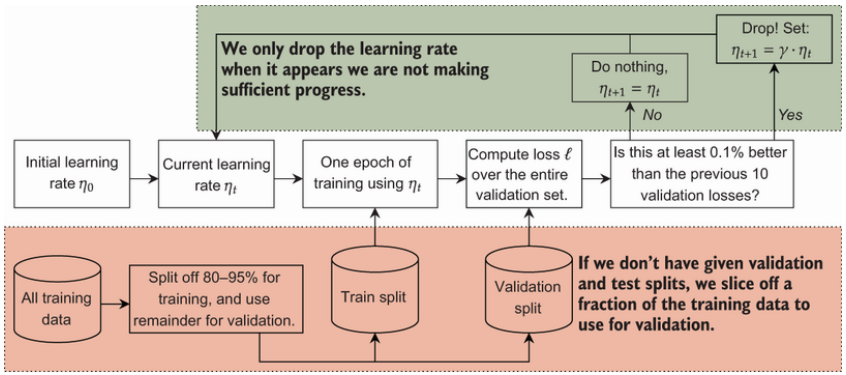


Figure 5.10 Plateau-based learning rate schedule strategy. The bottom part, in red, shows us slicing off a portion of the training data to use as our validation set. Doing this, or having a special validation set along with a test set, is necessary to avoid overfitting. The upper part, in green, shows how we reduce only the learning rate η when our validation loss is not reducing.

The following code shows how to do this properly. First, we use the `random_split` class to create an 80/20 split of the *training* data. The 20% split will become our validation set, which `ReduceLROnPlateau` uses to check whether we should drop the learning rate; and the 80% is used to

train the model parameters Θ. Pay attention to the call to *train_network*, where we use the `train_loader`, `val_loader`, and `test_loader` to set each of the three components properly. In the results of this model, we need to make sure we are looking at the *test* results and not the *val* results. Other than these careful precautions, this code is not much different from before:

```
fc_model.apply(weight_reset)                                                    ❶

train_sub_set, val_sub_set = torch.utils.data.random_split(train_data,
➥ [int(len(train_data)*0.8),                                                   ❷
➥ int(len(train_data)*0.2)])

train_sub_loader = DataLoader(train_sub_set,
➥ batch_size=B, shuffle=True)
val_sub_loader = DataLoader(val_sub_set, batch_size=B)                          ❸
optimizer =torch.optim.SGD(fc_model.parameters(), lr=eta_0)

scheduler =torch.optim.lr_scheduler.ReduceLROnPlateau(                          ❹
➥ optimizer, mode='min', factor=0.2, patience=10)

fc_results_plateau = train_network(fc_model, loss_func, train_loader,           ❺
➥ val_loader=val_sub_loader, test_loader=test_loader, epochs=epochs,
➥ optimizer=optimizer, lr_schedule=scheduler, score_funcs={'Accuracy':
➥ accuracy_score}, device=device)
```

❶ Resets the weights again so we don't have to define a new model

❷ Creates training and validation subsets since we do not have explicit validation and test sets

❸ Creates loaders for the train and validation subsets. Our test loader stays the same—never alter your test data!

❹ Sets up our plateau schedule using gamma=0.2

❺ Train up the model!

Don't shoot yourself in the foot!

Plateau-based adjustments to the learning rate are a *very* popular and successful strategy. Using information about how well the current model appears to be doing gets the best results on many problems. *But* you should not blindly use it in all circumstances. There are two particular cases where `ReduceLROnPlateau` may perform poorly or even mislead you into an overconfident result.

First is the case where you don't have much data. The `ReduceLROnPlateau` strategy requires training and validation sets to work and thus reduces the amount of data you have for learning the model parameters. If you have only 100 training points, it may be painful to use 10 to 30% for validation. You need enough data both to estimate the parameters Θ and tell whether the learning has plateaued—and if you don't have a lot of data, that may be too tall an order to fill.

The second case is when your data strongly violates the identically and independently distributed (IID) assumption. For example, if your data includes multiple samples from the same person, or events that depend on the date they occur (e.g., if you want to predict the weather, you can't have data from the future!), naively applying a random split to create the validation set can lead to poor results. In this situation, you need to make sure your training, validation, and testing splits don't have any accidental leakage from one to the other. Using the weather example, you might want to make sure your training split has data from *only* the years 1980 through 2004, validation data 2005 through 2010, and test data 2011 through 2016. This way, no accidental time traveling will occur, which would be a big IID violation.

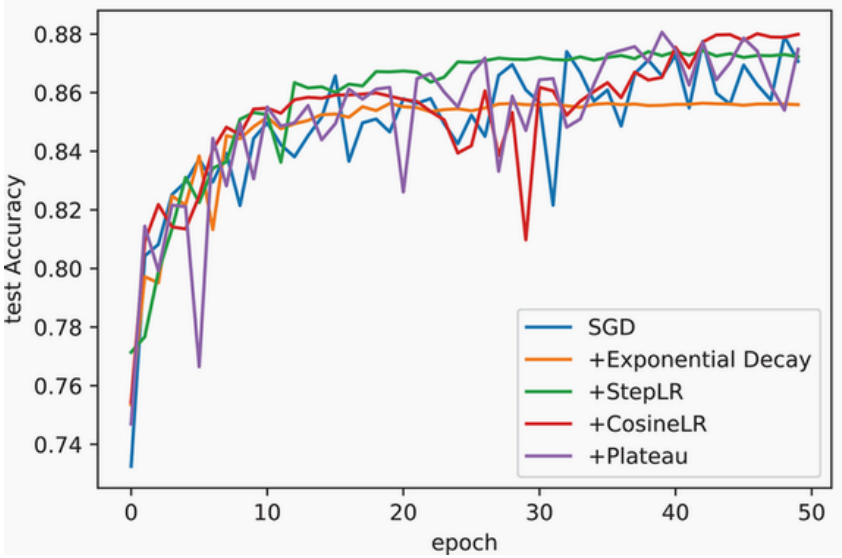
5.2.5 Comparing the schedules

Now that we have trained four common learning rate schedules, we can compare their results and see which performed best. The results are shown in the following plot, with the test accuracy on the y-axis. Some trends are quickly obvious and worth talking about. The vanilla SGD does fine but constantly fluctuates up and down from epoch to epoch. Every learning rate schedule we are looking at provides *some* kind of benefit:

```
sns.lineplot(x='epoch', y='test Accuracy', data=fc_results, label='SGD')
sns.lineplot(x='epoch', y='test Accuracy',
➥ data=fc_results_expolr, label='+Exponential Decay')
```

```
sns.lineplot(x='epoch', y='test Accuracy',
➥ data=fc_results_steplr, label='+StepLR')
sns.lineplot(x='epoch', y='test Accuracy',
➥ data=fc_results_coslr, label='+CosineLR')
sns.lineplot(x='epoch', y='test Accuracy',
➥ data=fc_results_plateau, label='+Plateau')
```

```
[22]: AxesSubplot:xlabel='epoch', ylabel='test Accuracy'>
```



Exponential decay has the benefit of being *smooth and consistent*: from epoch to epoch, it has almost identical behavior. Unfortunately, it trends toward the lower side of accuracy and does a little worse than naive SGD. But consistency has value, and if we ended one epoch sooner, SGD would perform worse.

The `StepLR` schedule is very similar to the exponential one in behavior, with a bit more asperity in the beginning as it goes fast before slowing down and stabilizing later in training. This ends up on the higher end of SGD in performance.

The cosine schedule ultimately does even better, reaching the highest accuracy of naive SGD, `StepLR`, or exponential decay. About 27 epochs in, performance drops suddenly as the learning rate ramps back up; then performance restabilizes at a higher accuracy when the learning rate de-

cays again. This is why I recommend setting the cosine approach so that it ends on a dip.

The Plateau approach also does a good job, and happens to just get second place in this set of tests. With more epochs, the Plateau would self-stabilize by dropping the learning rate further. Using feedback from how the model is actually doing, rather than assuming behavior, can help get every last bit of accuracy out of your model.

Table 5.1 summarizes the pros and cons and when you might want to use each method. Unfortunately, no one size fits all, and each will have cases where it doesn't pan out. That depends heavily on your data, and you won't know until you train the data and find out. This is why I like to start with no learning rate schedule and then add one of these four based on what happens.

Table 5.1 Flow `control` statements

	Exponential decay	StepLR	CosineLR	Plateau
Pros	Very consistentresults. Little tuningneeded.	Consistent results. Little tuning needed. Often improves accuracy.	Often improves accuracy. Advanced versions can significantly improve accuracy.	Often improves accuracy significantly and reduces epochs of training.
Cons	Can modestlyreduce final accuracy.	Most helpful when training for 100+ epochs.	Requires someparameter tuning; doesn't always work.	Not all data is used for training. More risk of overfitting.
When to use	Training the same model, but with different initial weights, can give wildly different results. Exponential decay can help fix stabilize training.	You want toimprove your final model's accuracy with minimal work.	You want to squeeze out maximum performance without too much extra work, and ideally you have enough time to do a few extra training runs.	You want to squeeze out maximum performance without too much extra work, and ideally you have a lot of data.

Despite how well the plateau approach does, we won't use it much in this book. Part of the reason is that it requires extra code that can be distracting when we are trying to focus on new concepts. Also, sometimes we will set up learning problems in a *specific* way so you see real-world behavior for problems that take only a few minutes to run instead of hours or more, and the plateau approach makes it harder to set up some of these scenarios. But these initial results show that you should keep it in mind as a powerful tool in your arsenal.

5.3 Making better use of gradients

We now know several ways to alter the learning rate η to improve how quickly our model learns and allow it to learn more accurate solutions. We did this by defining different learning rate schedules $L(\cdot,\cdot)$, which look at the long term. Think of $L(\cdot,\cdot)$ as setting the desired speed for the journey. But detours, potholes, and other obstacles in the short term may require a change in speed. This is why we also want to modify the gradient $\boldsymbol{g}$ with a function $G(\cdot)$. We focus on gradient update schemes on their own first and then combine them with learning rate schedules to get improved results. First, we talk about some broad motivations and then dive into the most common approaches. All three approaches are one-line changes to your code with PyTorch (they are built in) and can dramatically improve the accuracy of your models.

Remember that we are using $\boldsymbol{g}^t$ to indicate the tth gradient that we are receiving to update the network. Because we are doing stochastic gradient descent in batches, this is a noisy gradient. Even with the noise in $\boldsymbol{g}^t$, there is valuable information in $\boldsymbol{g}^t$ that we are not fully utilizing. Consider the jth parameter's gradient $\boldsymbol{g}_j^t$. What if it's consistently almost the same value every time? Mathematically, that would be a situation where

$$\boldsymbol{g}_j^t \approx \boldsymbol{g}_j^{t-1} \approx \boldsymbol{g}_j^{t-2} \approx \boldsymbol{g}_j^{t-3} \approx \dots$$

This tells us the jth parameter needs to be moved in the same direction each time. Figure 5.11 shows such a situation.

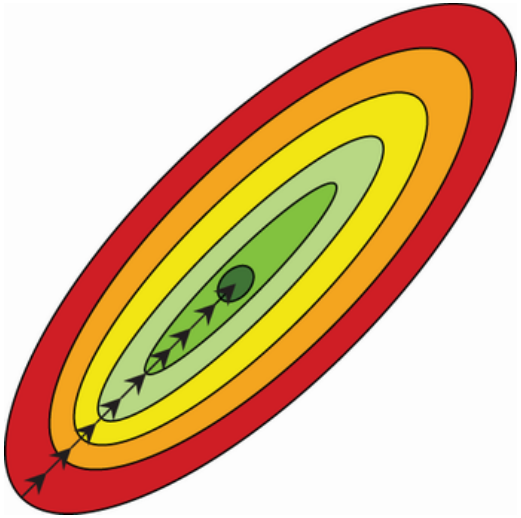


Figure 5.11 Example where the gradient value could consistently return the same value over and over. We are heading on a direct path toward the solution, but we could get there faster if we used a larger learning rate.

We don't necessarily want to increase the learning rate η, because other dimensions may have different behavior. Every network parameter gets its own gradient value, and networks can have millions of parameters. So if the gradient at index j is consistently the same, maybe we should be increasing how far we step (i.e., increase learning rate η) for *just* index j. We would need to ensure that this was done only with respect to index j, because the gradient for a different index i might not be as consistent.

So, we would like to have a global learning rate η and individualized learning rates η_j for all of the parameters. While the global learning rate stays fixed (unless we use a learning rate schedule), we let some algorithms adjust each of the η_j values to try to improve convergence. Most gradient update schemes work by rescaling the gradient $\boldsymbol{g}^t$, which is equivalent to giving each parameter its own personalized learning rate η_j.

5.3.1 SGD with momentum: Adapting to gradient consistency

We want to increase the learning rate if the gradient for parameter j consistently heads in the same direction. We can describe this as wanting the gradient to build momentum. If the gradient in one direction keeps returning similar values, it should start taking bigger and bigger steps in

the same direction. Figure 5.12 shows an example problem that momentum can help fix.

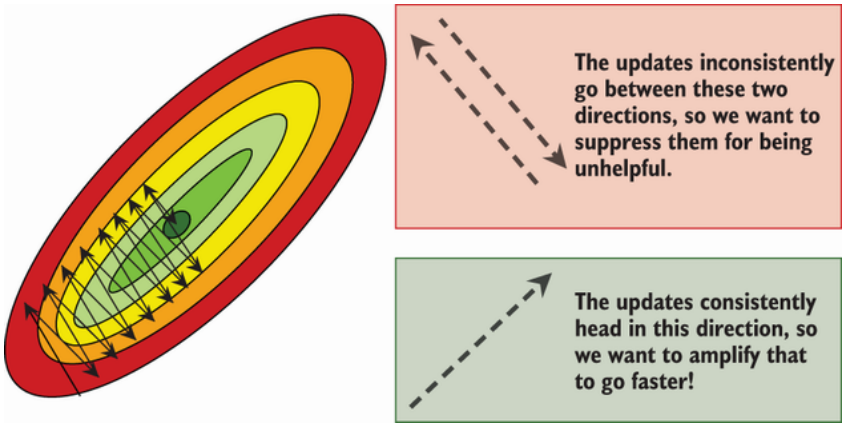


Figure 5.12 The naive optimization is almost stuck oscillating between the top-left and bottom-right directions, but it is slowly and consistently making progress in the top-right direction that leads to the solution. If we could build some momentum in that direction, we would not need so many steps.

This momentum strategy was one of the first developed for improving SGD and solving the problem shown in the figure and is still widely used today. Let's talk about the math of how it works.

Our normal gradient update equation, again, looks like this:

$$\Theta_{t+1} = \Theta_t - \eta \cdot g^t$$

The initial version of the momentum idea is to have a momentum weight μ, which has some value in the range $(0,1)$ (i.e., $\mu \in (0,1)$).[3] Next, we add a velocity term called v, which accumulates our momentum. So our velocity v contains a fraction (μ) of our previous gradient step, as detailed in the following equation:

$$\underbrace{v^{t+1}}_{\text{The new momentum}} = \underbrace{\mu \cdot v^t}_{\text{a fraction of the current momentum}} + \underbrace{\eta \cdot g^t}_{\text{The current direction}}$$

Because our velocity depends on our previous velocity, it takes into account all of our previous gradient updates. Next, we simply use v in place of g to perform the gradient update:

$$\Theta^{t+1} = \Theta^t - v^{t+1}$$

The amazing shrinking history

The equation for momentum relies heavily on the fact that μ is a value smaller than 1. This is because it has a shrinking effect on old gradients. One way to look at this is to write out what happens to the equations as we repeatedly apply momentum. Let's write this out in detail.

The following table shows what happens to the velocity v as we keep applying momentum μ. The left-most column is the new value of Θ, and we start with Θ_1 as the initial random weights. There is no previous velocity v yet, so nothing happens. Starting with the second round, we can expand the velocity term, as shown on the right.

$$\Theta_2 = \Theta_1 - \underbrace{(\eta \cdot g_1)}_{\text{this is the new momentum } v_1}$$

$$\Theta_3 = \Theta_2 - \underbrace{(\mu \cdot v_2 + \eta \cdot g_2)}_{v_3} = \Theta_2 - \underbrace{\left(\overbrace{\mu \cdot \eta \cdot g_1}^{\text{This is } \mu \cdot v_2} + \eta \cdot g_2\right)}_{v_3}$$

$$\Theta_4 = \Theta_3 - \underbrace{(\mu \cdot v_3 + \eta \cdot g_3)}_{v_4} = \Theta_3 - \underbrace{\left(\overbrace{\mu^2 \cdot \eta \cdot g_1 + \mu \cdot \eta \cdot g_2}^{\text{This is } \mu \cdot v_3} + \eta \cdot g_3\right)}_{v_4}$$

$$\Theta_5 = \Theta_4 - \underbrace{(\mu \cdot v_4 + \eta \cdot g_4)}_{v_5} = \Theta_4 - \underbrace{\left(\overbrace{\mu^3 \cdot \eta \cdot g_1 + \mu^2 \cdot \eta \cdot g_2 + \mu \cdot \eta \cdot g_3}^{\text{This is } \mu \cdot v_4} + \eta \cdot g_4\right)}_{v_5}$$

Notice that every time we update with momentum, *every previous gradient* contributes to the current update! Also notice that the exponents above μ get larger every time we apply another round of momentum. This makes the contributions of very old gradients become essentially zero as we keep updating. If we use a standard momentum of $\mu = 0.9$, old gradients contribute less than 0.01% of their value after just 88 updates. Since we often do thousands to hundreds of thousands of updates per epoch, this is a very short-term effect.

You may look at the previous equations and be concerned: if we are taking into account all of our old gradients, what if some of them are no longer useful? That's why we keep the momentum term $\mu \in (0,1)$. If we are currently at update t, and we think about the past gradient from k steps

ago, its contribution is $\mu^k \boldsymbol{g}^t$, which quickly becomes small. If we have $\mu = 0.9$, the gradient 40 steps ago is contributing a weight of only $0.9^{40} \approx 0.01$ to the current velocity $\boldsymbol{v}$. The value μ helps us forget past gradients so that learning can adapt and is able to grow larger if we are always heading in the same direction.

This strategy is a simple way to significantly improve your model's accuracy and training time. It can solve both problems we have looked at; figure 5.13 shows the solutions.

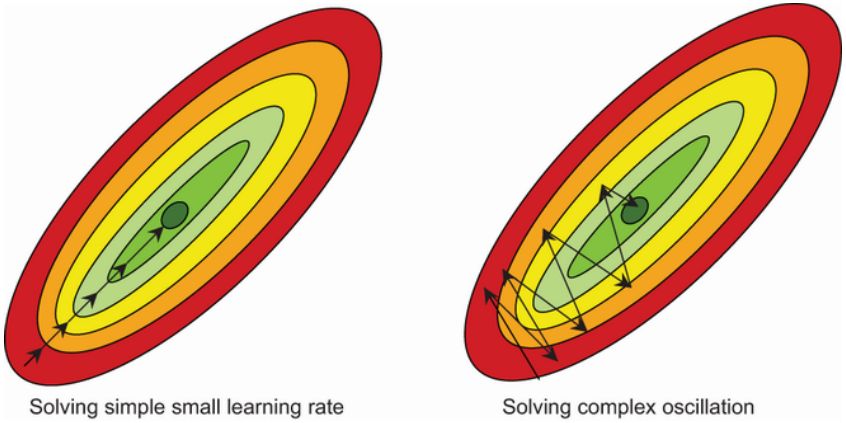


Figure 5.13 On the left, momentum solves the problem of a small learning rate. The direction is consistent, so the momentum builds and increases the effective learning rate. On the right, the momentum builds consistently—but initially, slowly—in the top-right direction. The oscillation may continue, but fewer steps are needed to reach the minimum.

If you want to get the *best possible* accuracy from your network, SGD with some form of momentum is still considered one of the best options. The downside is that finding the combination of μ and η values that gives the *absolute best results* is not easy and requires training many models. We talk about smarter ways to search for μ and η in the last section of this chapter.

Nesterov momentum: Adapting to changing consistency

A second flavor of momentum exists and is worth mentioning, as it often performs better in practice. This version is called *Nesterov momentum*.

In normal momentum, we take the gradient with respect to the current weights (that is, $\boldsymbol{g}^t$) and then move in the direction of our gradient and ve-

locity combined. If we have built up a *lot* of momentum, it can be difficult to make turns during the optimization. Let's look at a toy example of what can go wrong in figure 5.14.

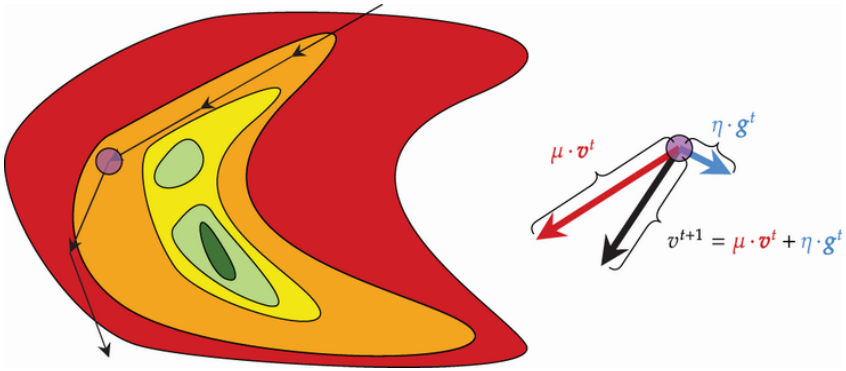
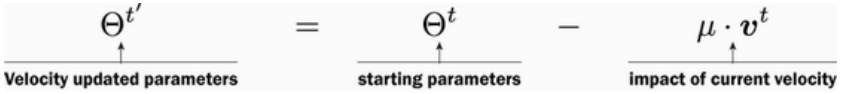


Figure 5.14 Imagine that it has taken a long time to reach this point, so the optimizer has built up a large momentum (i.e., $\|v^t\|_2 > 0$). When we reach the purple point, the momentum causes us to swerve *around* the desired area because the momentum, even after decay, is larger than the gradient. Since it is larger, it carries the weights more in the same direction rather than a new direction.

Technically, this problem shows momentum behaving correctly: the momentum is carrying Θ_{t+1} in a consistent direction. It's just not a good idea anymore, and it will take a few iterations for the momentum to be corrected and start heading toward the actual solution.

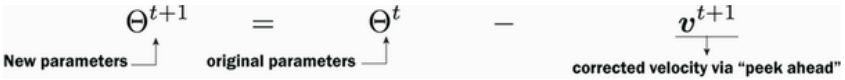
In Nesterov momentum, we instead decide to be patient. Normal momentum at step t immediately calculates the new gradient g^t and adds the velocity v^t. Nesterov first acts on the velocity and *then* calculates a new gradient *after* letting the momentum move us. This way, if we are moving in the *wrong* direction, Nesterov will likely have a larger gradient to push us back in the correct direction sooner. This looks like the following sequence of equations, where I'm using t' to denote the patient steps.

First we compute $\Theta^{t'}$ based only on the previous velocity. We don't look at the new data yet, meaning this could be a bad direction if things are changing (or it could be good—we don't know because we haven't looked at the data):



Second, we finally look at the new data x using our modified weights $\Theta^{t'}$. This is like peeking at the near future so we can change or correct our answer about how we want to update the parameters. If the velocity was in the correct direction, nothing really changes. But if the velocity was going to take us in a bad direction, we can change course and counteract the effect more quickly:

Finally, we modify $\Theta^{t'}$ again using the new velocity that contains the fresh gradient, giving us the ultimate updated weights Θ^{t+1}:



To recap: the first equation alters the parameters Θ based on *only* the current velocity (no new batch and no new information). The second equation computes a *new* velocity using the old velocity and the gradient *after* having altered Θ from the first equation. Finally, the third equation takes a step based on these results. While this looks like extra work, there is a clever way to organize it so that it takes exactly the same amount of time as normal momentum (which we won't go into because it's a little confusing). Figure 5.15 shows how Nesterov momentum affects the previous example.

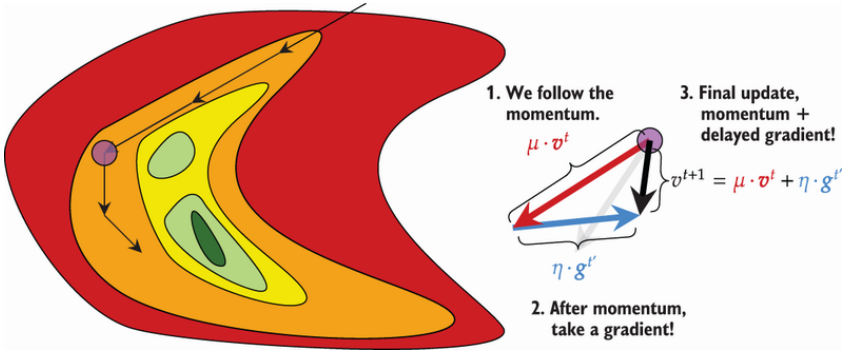


Figure 5.15 At the same purple point, the right side shows how the update is calculated. Instead of computing g^t at the starting point, we first follow the momentum, which is in the *wrong* direction. This causes the gradient at this new location to be larger back in the *correct* direction. Combined, we get a smaller step that is closer to the solution. The original standard momentum result is shown faded in black.

Let's reason through another scenario to exercise our mental model of what is happening. Try drawing a diagram of this yourself to help you follow along.

Consider the standard momentum case where our momentum has been useful, allowing us to head in the correct direction, and assume that we have just reached our goal. We are at the function's minimum, and the problem is solved. But, *we don't ever know for sure that we are at the minimum*, so we run the next step of the optimization process. Because of the momentum we have built up, we are about to roll past the minimum. In normal momentum, we compute the gradient—and if we are at the solution, the gradient $g^t = 0$, so there is no change. Then we add the velocity v^t, and it carries us away.

Now, think about this scenario under Nesterov momentum. We are at the solution, and first we follow the velocity, pushing us away from the goal. Then we calculate the gradient, which recognizes that we need to head back in the opposite direction and thus move toward the goal. When we add these two together, they almost cancel out (we take a smaller step forward or backward, depending on which has a larger magnitude, g^t or v^t). In one step, we have started to change the direction in which our optimizer is headed, whereas normal momentum would take two steps.

This is the intuition and reasoning for why Nesterov momentum is usually the preferred version of momentum. Now that we've talked about the

ideas, we can turn them into code. The `SGD` class simply requires us to set the `momentum` flag to a nonzero value if we would like momentum and `nesterov=True` if we want it to be Nesterov momentum. The following code trains our network with both versions.

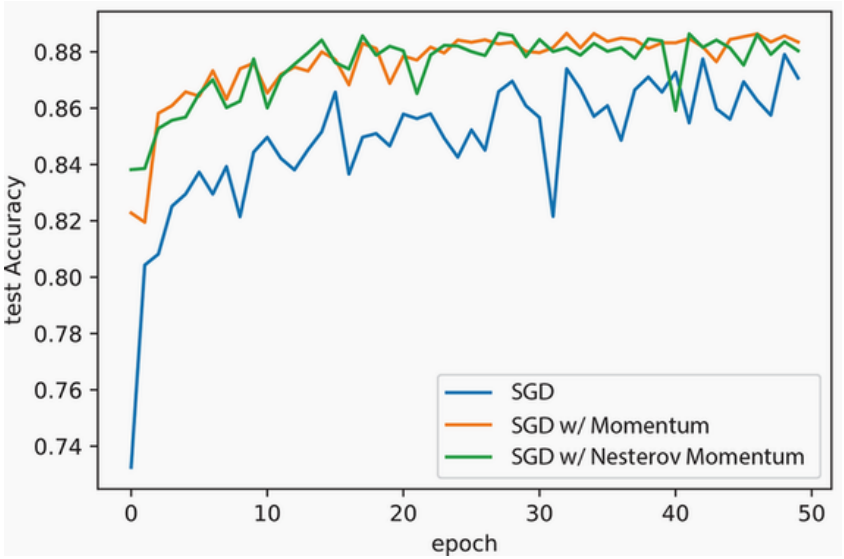
```
fc_model.apply(weight_reset)

optimizer = torch.optim.SGD(fc_model.parameters(), lr=eta_0,
➥ momentum=0.9, nesterov=False)

fc_results_momentum = train_network(fc_model, loss_func, train_loader,
➥ test_loader=test_loader, epochs=epochs, optimizer=optimizer,
➥ score_funcs={'Accuracy': accuracy_score}, device=device)

fc_model.apply(weight_reset)

optimizer = torch.optim.SGD(fc_model.parameters(), lr=eta_0,
➥ momentum=0.9, nesterov=True)

fc_results_Nesterov = train_network(fc_model, loss_func, train_loader,
➥ test_loader=test_loader, epochs=epochs, optimizer=optimizer,
➥ score_funcs={'Accuracy': accuracy_score}, device=device)
```

Comparing SGD with momentums

Here we plot the results from vanilla SGD and both flavors of momentum. Both momentum versions perform dramatically better, learning faster and giving more accurate solutions. But you won't usually see one version of momentum perform dramatically better or worse than the other. While Nesterov does solve a real problem, normal momentum will *eventually* correct itself after more gradient updates. Still, I prefer to use the Nesterov flavor because in my experience, if one momentum is better by a nontrivial amount, it is usually Nesterov:

```
sns.lineplot(x='epoch', y='test Accuracy', data=fc_results, label='SGD')
sns.lineplot(x='epoch', y='test Accuracy', data=fc_results_momentum,
➥ label='SGD w/ Momentum') sns.lineplot(x='epoch', y='test Accuracy', data=fc_results_nestrov,
➥ label='SGD w/ Nesterov Momentum')
```

```
[25]: <AxesSubplot:xlabel='epoch', ylabel='test Accuracy'>
```



5.3.2 Adam: Adding variance to momentum

One of the currently most popular optimization techniques is called *adaptive moment estimation*, or Adam for short. Adam is strongly related to the SGD with momentum that we just described. Adam is my current favorite approach because it has default parameters that just work, so I don't have to spend time tuning them. This is not true for SGD with momentum (normal or Nesterov flavored). We have to rename a few terms to make the math consistent with how you will read about it elsewhere: the velocity $\boldsymbol{v}$ becomes $\boldsymbol{m}$, and the momentum weight μ becomes β_1. Now we can describe the first step of Adam, which has one major change shown in red:

$$\underbrace{\boldsymbol{m}^t}_{\text{new velocity}} = \underbrace{\beta_1}_{\text{momentum}} \cdot \underbrace{\boldsymbol{m}^{t-1}}_{\text{old velocity}} + \underbrace{(1-\beta_1) \cdot \boldsymbol{g}^t}_{\text{the gradient}}$$

This describes the momentum update equation, but we are now down-weighting the *current* gradient $\boldsymbol{g}^t$ by $(1-\beta_1)$. This makes $\boldsymbol{m}^t$ a *weighted average* between the previous velocity and the current gradient. More specifically, because we added the $(1-\beta_1)$ term, this is now called an *exponential moving average*. It's called *exponential* because the gradient from z steps ago $\boldsymbol{g}^{t-z}$ has an exponentiated contribution of $\beta_1{}^z$, and it's a *moving average* because it is computing a kind of weighted average where most

of the weight is on the most recent items, so the average moves with the most recent data.

Since we are now discussing momentum as an *average* or *mean* gradient over multiple updates, we can talk about the *variance* of the gradient as well. If one parameter g_j^t has high variance, we probably don't want to let it contribute too much to the momentum because it will likely change again. If a parameter g_j^t has *low* variance, it's a reliable direction to head in, so we should give it *more* weight in the momentum calculation.

To implement this idea, we can calculate information about the variance over time by looking at squared gradient values. Normally, for variance, we would subtract the mean before squaring, but that is hard to do in this scenario. So we use the squared values as an *approximation* of the variance and keep in mind that it does not provide perfect information. Using $\odot$ to denote the elementwise multiplication between two vectors ($\boldsymbol{a} \odot \boldsymbol{b} = [a_1 \cdot b_1, a_2 \cdot b_2, \ldots]$) leads to this equation for variance of velocity $\boldsymbol{v}$:

$$\underbrace{\boldsymbol{v}^t}_{\text{new variance velocity}} = \underbrace{\beta_2}_{\text{variance momentum}} \cdot \underbrace{\boldsymbol{v}^{t-1}}_{\text{old variance velocity}} + (1-\beta_2) \cdot \underbrace{\boldsymbol{g}^t \odot \boldsymbol{g}^t}_{\text{approximate variance}}$$

We save $\boldsymbol{m}^t$ and $\boldsymbol{v}^t$, but we perform one more alteration before using them. Why? Because when we are early in the optimization process, meaning t is a small value, $\boldsymbol{m}^t$ and $\boldsymbol{v}^t$ give us biased estimates of the mean and variance. They are biased because they are initialized to zero (i.e., $\boldsymbol{m}^0 = \boldsymbol{v}^0 = \vec{0}$), so the early estimates will be *too small* if we use them naively.

Think about when $t = 1$. In that case, the value of $\boldsymbol{m}^1 = (1-\beta_1) \cdot \boldsymbol{g}$. The true average is just $\boldsymbol{g}$, but instead we have discounted it by a factor of $1 - \beta_1$. To fix this, we just adjust as so:

$$\hat{m} = \frac{m^t}{1 - \beta_1^t}$$

$$\hat{v} = \frac{v^t}{1 - \beta_2^t}$$

Now we have $\hat{m}$ and $\hat{v}$, which give us the unbiased estimates of the mean and variance, respectively, and we use them together to update our weights Θ:

The new parameters θ_{t+1} are equal to the old parameters θ_t minus the learning rate times momentum and gradient (what currently looks best) but adjust how much we trust the direction by how consistently we've been going in the same direction over time.

$$\theta_{t+1} = \theta_t - \eta \cdot \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon}$$

What is going on here? The numerator $\eta \cdot \hat{m}$ is computing the SGD with momentum term, but now we are normalizing each dimension by the variance. That way, if a parameter has naturally large swings in its value, we won't adjust as quickly to a new large swing—because it usually is quite noisy. If a parameter usually has a small variance and is very consistent, we adapt quickly to any observed change. The ϵ term is a very small value so we never end up dividing by zero.

That gives you the intuition behind Adam. The original authors proposed using the following values: $\eta = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$.

Personally, I recommend using Adam or one of its descendants as the default choice for any optimization problem. Why? Because it's an optimizer that, using the default values, usually performs well with no further alteration. It may not get you the *best possible* performance, and you can often improve results by adjusting the parameters, but the defaults usually work well. And if they don't work, other parameter settings usually won't, either.

Most optimizers don't share this property, or not to the degree that Adam exhibits it. For example, SGD is very sensitive to the momentum and learning rate terms, and you usually need to do some tuning to get good performance out of normal SGD. Since Adam does not *require* this finicky tuning, you don't have to do as much experimentation to figure out what works. Thus, you can save a detailed optimization process until after you have settled on your final architecture and are ready to squeeze out every last drop of accuracy. Ultimately, this makes Adam a great time saver: spend the time on your architecture, and leave the optimizer changes for the end.

Other flavors of Adam

The original paper for the Adam algorithm contained a mistake, but the proposed algorithm still worked well for the vast majority of problems. A version that fixes this mistake is called `AdamW` and is the default we use in this book.

Another extension of Adam is NAdam, where the *N* stands for Nesterov. As you might guess, this version adapts Adam to use Nesterov momentum rather than standard momentum. A third flavor is AdaMax, which replaces some of the multiplication operations in Adam with `max` operations to improve the algorithm's numerical stability. All of these flavors have pros and cons that are beyond the scope of this book, but any Adam variant will serve you well.

Now that we have learned about Adam, let's try it. The following code again resets the weights for the neural network we have been training over and over, but using `AdamW` this time:

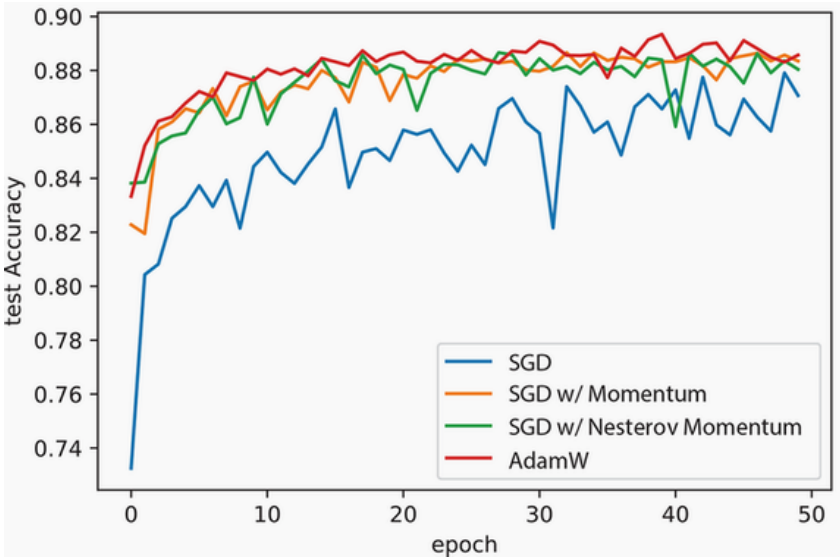
```
fc_model.apply(weight_reset)

optimizer = torch.optim.AdamW(fc_model.parameters()) ❶

fc_results_adam = train_network(fc_model, loss_func, train_loader,
➥ test_loader=test_loader, epochs=epochs, optimizer=optimizer,
➥ score_funcs={'Accuracy': accuracy_score}, device=device)
```

❶ We don't set the learning rate for Adam because you should always use the default, and Adam can be more sensitive to large changes in the learning rate.

Now we can plot `AdamW` along with the three versions of SGD we have already looked at. The result shows that AdamW performs as well as SGD with either flavor of momentum, but its behavior is a bit more stable when the dips down are not as frequent or dramatic:

```
sns.lineplot(x='epoch', y='test Accuracy', data=fc_results, label='SGD')
sns.lineplot(x='epoch', y='test Accuracy', data=fc_results_momentum,
➥ label='SGD w/ Momentum')
sns.lineplot(x='epoch', y='test Accuracy', data=fc_results_nestrov,
➥ label='SGD w/ Nestrov Momentum')
sns.lineplot(x='epoch', y='test Accuracy', data=fc_results_adam,
➥ label='AdamW')
```

```
[27]: <AxesSubplot:xlabel='epoch', ylabel='test Accuracy'>
```



We can also combine these new optimizers with the learning rate schedules we have learned about. Here, we train Adam and SGD with Nesterov momentum combined with the cosine annealing schedule:

```
fc_model.apply(weight_reset) optimizer =
torch.optim.AdamW(fc_model.parameters())                                    ❶

scheduler = torch.optim.lr_scheduler.
➥ CosineAnnealingLR(optimizer, epochs//3)
fc_results_adam_coslr = train_network(fc_model, loss_func, train_loader,
➥ test_loader=test_loader, epochs=epochs, optimizer=optimizer,
➥ lr_schedule=scheduler, score_funcs={'Accuracy': accuracy_score},
➥ device=device)

fc_model.apply(weight_reset) optimizer =
torch.optim.SGD(fc_model.parameters(), lr=eta_0, momentum=0.9, nesterov=True) ❷

scheduler = torch.optim.lr_scheduler.
➥ CosineAnnealingLR(optimizer, epochs//3)
fc_results_nesterov_coslr = train_network(fc_model, loss_func,
➥ train_loader, test_loader=test_loader, epochs=epochs,
➥ optimizer=optimizer, lr_schedule=scheduler,
➥ score_funcs={'Accuracy': accuracy_score}, device=device)
```

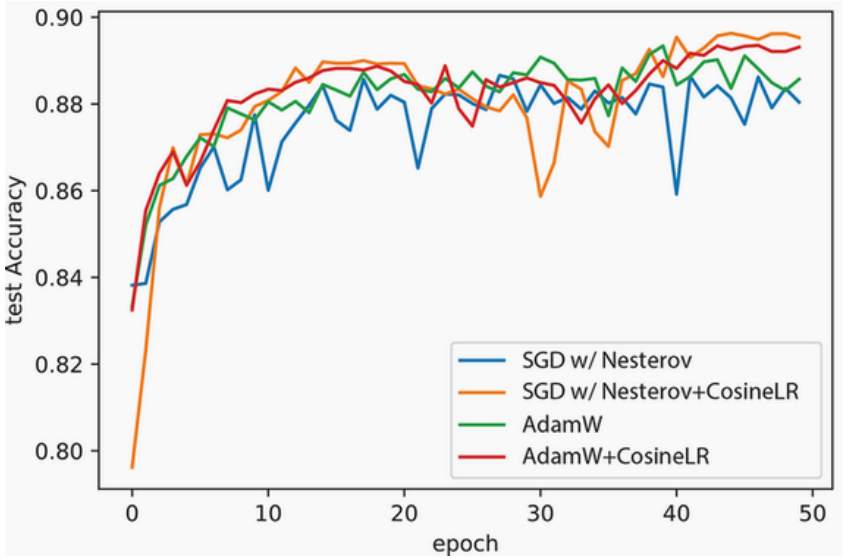
❶ Adam with cosine annealing

❶ SGD+Nesterov with cosine annealing

Now we can compare the results with and without the cosine schedule in the following code and plot. Again we see a similar trend, that adding the learning rate schedule gives a bump to accuracy and that the version with `AdamW` is *slightly* better behaved:

```
sns.lineplot(x='epoch', y='test Accuracy', data=fc_results_nesterov,
➥ label='SGD w/ Nesterov')
sns.lineplot(x='epoch', y='test Accuracy', data=fc_results_nesterov_coslr,
➥ label='SGD w/ Nesterov+CosineLR')
sns.lineplot(x='epoch', y='test Accuracy', data=fc_results_adam,
➥ label='AdamW')
sns.lineplot(x='epoch', y='test Accuracy', data=fc_results_adam_coslr,
➥ label='AdamW+CosineLR')
```

```
[29]: <AxesSubplot:xlabel='epoch', ylabel='test Accuracy'>
```



5.3.3 Gradient clipping: Avoiding exploding gradients

We have one last trick to talk about, which is compatible with both optimizers like Adam and SGD as well as learning rate schedules. It is called *gradient clipping*, and it helps us solve the exploding gradient problem. Unlike all the mathematical intuition and logic that went into everything we have discussed so far, gradient clipping is refreshingly simple: if any (absolute) value in a gradient is larger than some threshold z, just set it to a maximum value of z. So if we use $z = 5$ and our gradient is $\boldsymbol{g} = [1,-2,1000,3,-7]$, the clipped version becomes $\text{clip}_5(\boldsymbol{g}) = [1,-2,5,3,-5]$. The idea is that any value larger than our threshold z clearly indicates the *direction*; but it is set to an unreasonable *distance*, so we forcibly clip it to something reasonable.

The following code shows how to add gradient clipping to any neural network. We grab the parameters Θ using the `model.parameters()` function and use `register_hook` to register a callback that is executed every time the gradients are used. In this case, we simply take the tensor `grad` that represents the gradients and use the `clamp` function that returns a new version of `grad` where nothing is smaller than `-5` or larger than `5`. That's all it takes:

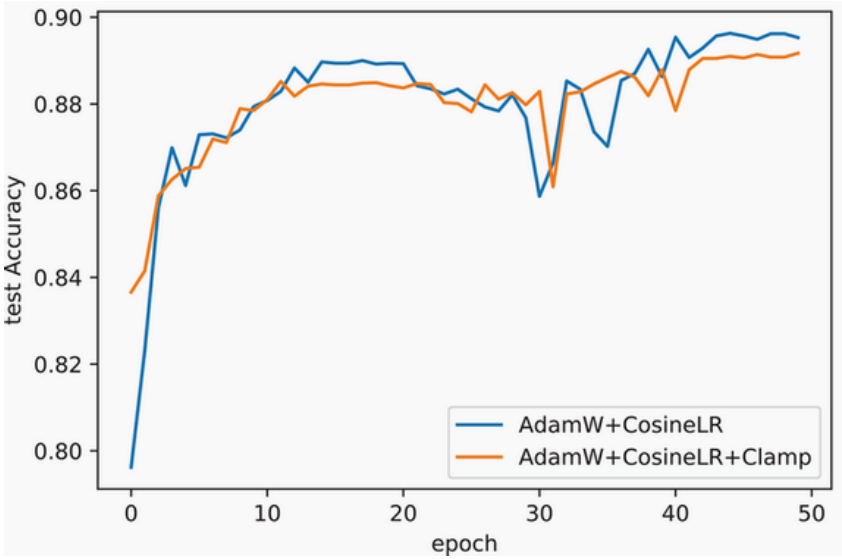
```
fc_model.apply(weight_reset)
```

```
for p in fc_model.parameters():
p.register_hook(lambda grad: torch.clamp(grad, -5, 5))

optimizer = torch.optim.AdamW(fc_model.parameters())
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, epochs//3)
fc_results_nesterov_coslr_clamp = train_network(fc_model, loss_func,
➥ train_loader, test_loader=test_loader, epochs=epochs,
➥ optimizer=optimizer, lr_schedule=scheduler,
➥ score_funcs={'Accuracy': accuracy_score}, device=device)
```

Plotting the results, you see that they are *usually* the same. In this case, they are a little worse, but they could have been better, instead. It depends on the problem:

```
sns.lineplot(x='epoch', y='test Accuracy', data=fc_results_nesterov_coslr,
➥ label='AdamW+CosineLR')
sns.lineplot(x='epoch', y='test Accuracy',
➥ data=fc_results_nesterov_coslr_clamp, label='AdamW+CosineLR+Clamp')
```

```
[31]: <AxesSubplot:xlabel='epoch', ylabel='test Accuracy'>
```



Exploding gradients are usually a *catastrophic* problem. If your model has them, it's probably learning a degenerate solution or not converging

at all. Since this dataset and network don't have that problem, gradient clipping isn't beneficial.

In most applications, I don't use gradient clipping unless my models are not learning well to begin with. If they learn well, I probably don't have exploding gradients, so I focus on other changes. If the model isn't good, I test gradient clipping to see if that fixes the problem. If I'm working with recurrent neural networks, I *always* use gradient clipping because the recurrent connections tend to cause exploding gradients, making them a common issue in that scenario. But if you want to always include clipping, doing so is an equally valid strategy. Using a clip value of $z = 5$ or $z = 10$ is common and works well for most problems.

5.4 Hyperparameter optimization with Optuna

Now that we have improved how to do training, we will reuse many of these techniques throughout the book because they are useful for *any and every* neural network you will ever train. The improvements so far have focused on optimizing when we have gradients. But hyperparameters are things we would like to optimize for which we do not have any gradients, such as the initial learning rate η to use and the value of the momentum term μ. We would also like to optimize the architecture of our networks: should we use two layers or three? How about the number of neurons in each hidden layer?

The first hyperparameter tuning method most people learn in machine learning is called *grid search*. While valuable, grid search works well only for optimizing one or two variables at a time due to its exponential cost as more variables are added. When training a neural network, we usually have at least three parameters we want to optimize (number of layers, number of neurons in each layer, and learning rate η). We instead use a newer approach—Optuna—to tuning hyperparameters, which works much better. Unlike grid search, it requires fewer decisions, finds better parameter values, can handle more hyperparameters, and can be adapted to your computational budget (i.e., how long you are willing to wait).

With all that said, hyperparameter tuning is still very expensive, and these examples can't run in a few minutes because they require training tens of models. In the real world, there could even be hundreds of mod-

els. This makes Optuna impractical for use in future chapters because we do not have the time, but using it is still a critical skill for you to know.

5.4.1 Optuna

To perform a smarter type of hyperparameter optimization, we use a library called Optuna. Optuna can be used with any framework, as long as you can describe the goal as a single numeric value. Luckily for us, we have the accuracy or error as our goal, so we can use Optuna. Optuna does a better job of hyperparameter optimization by using a Bayesian technique to model the hyperparameter problem as its own machine learning task. We could spend a whole chapter (or more) on the technical details of how Optuna works: in short, it trains its own machine learning algorithm to predict a model's accuracy (the label) based on its hyperparameters (the features). Optuna then sequentially tries new models based on what it predicts to be good hyperparameters, trains the model to find out how well it did, adds that information to improve the model, and then selects a new guess. But for now, let's focus on how to use Optuna as a tool.

To start, let's see a little about how Optuna works. Similar to PyTorch, it has a "define by run" concept. For Optuna, we define a function that we want to minimize (or maximize), which takes as input a `trial` object. This `trial` object is used to get `guesses` for each parameter we want to tune and returns a score at the end. The returned values are floats and integers, just as we would use ourselves, making it easy to use. Figure 5.16 shows how this works.

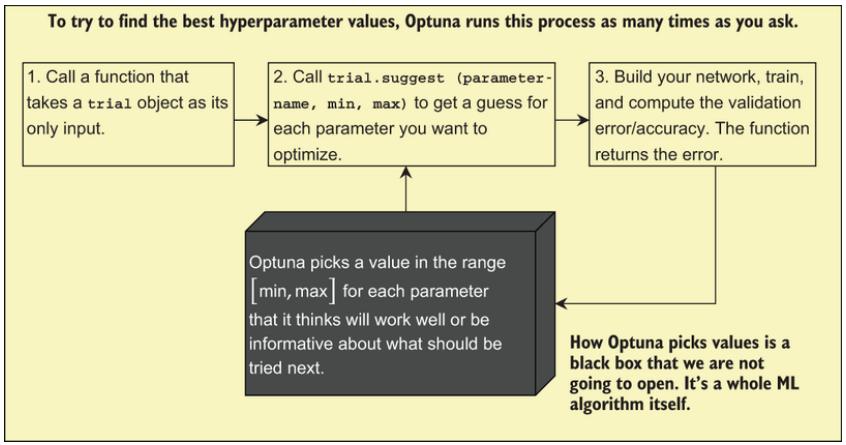


Figure 5.16 You supply Optuna with a function that does the three steps outlined in this diagram. Optuna uses its own algorithm to pick values for each hyperparameter, which you tell Optuna about using the `trial.suggest` functions; this function also informs Optuna about the minimum and maximum values you want to consider. You tell Optuna how many times to do this process, and each time it does, the black box gets better at picking new values to try.

Let's look at a toy function that we want to minimize: $f(x,y) = \text{abs}((x-3)\cdot(y+2))$. It's easy to tell that one minimum exists at $x = 3$ and $y = -2$. But can Optuna figure that out? First we need to import the Optuna library, which is a simple `pip` command:

```
!pip install optuna
```

Now we can import Optuna:

```
import optuna
```

Next we need to define the function to be minimized. `toyFunc` takes in the `trial` object. Optuna figures out how many hyperparameters exist by means of us using the `trial` object to obtain a guess for each parameter. This happens with the `suggest_uniform` function, which requires us to provide a range of possible values (something we must do for any hyperparameter optimization approach):

```
def toyFunc(trial):
    x = trial.suggest_uniform('x', -10.0, 10.0) ❶❷
    y = trial.suggest_uniform('y', -10.0, 10.0) ❶❸
```

```
    return abs((x-3)*(y+2))                    ❹
```

❶ These two calls ask Optuna for two parameters and define the minimum and maximum value for each one.

❷ $x \sim \mathcal{U}(-10,10)$

❸ $y \sim \mathcal{U}(-10,10)$

❹ Computes and returns the result. Optuna tries to minimize this value: |(x-3)(y+2)|.

That's all we needed to do. Now we can use the `create_study` function to build the task and call `optimize` with the number of trials we want to let Optuna have to minimize the function:

```
study = optuna.create_study(direction='minimize') ❶
study.optimize(toyFunc, n_trials=100)             ❷
```

❶ If you use direction='maximize', Optuna will try to maximize the value returned by toyFunc.

❷ Tells Optuna which function to minimize and that it gets 100 attempts to do so

When you run the code, you should see a long list of output something like this:

```
Finished trial#12 with value: 2.285 with parameters:
{'x': 0.089, 'y': -2.785}. Best is trial#9 with value: 0.535. Finished trial#13 with value: 3.069 w
{'x': -1.885, 'y': -2.628}. Best is trial#9 with value: 0.535. Finished trial#14 with value: 0.018 
{'x': -3.183, 'y': -1.997}. Best is trial#14 with value: 0.018.
```

When I ran this code, Optuna got a very precise answer: $5.04 \cdot 10^{-5}$ is very close to the true minimum of zero. The values it returned were also close to what we knew was the true answer. We can access these using `study.best_params`, which contains a `dict` object mapping the hyperparameters to the values that, in combination, gave the best result:

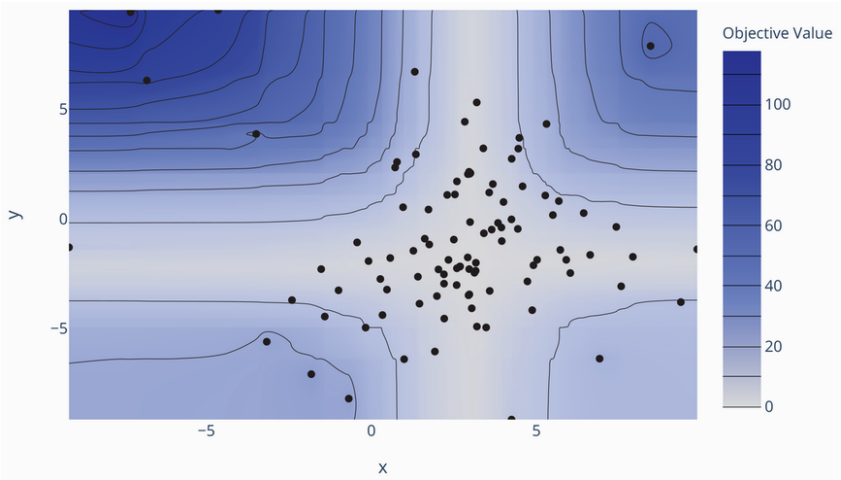
```
print(study.best_params) ❶
{'x': 2.984280340674378, 'y': -1.8826725682225192\}
```

❶ This dictionary holds the parameter values Optuna found to be best.

We can also use the `study` object to get information about the optimization process that happened. Optuna is powerful because it uses machine learning to explore the space of parameter values. By specifying the parameters with min and max values, we give Optuna the constraints—and it attempts to balance exploring the space to understand what it looks like and minimizing the score based on its current understanding of the space.

We can use a contour plot to see an example:

```
fig = optuna.visualization.plot_contour(study)
```



Notice how Optuna has spent most of its effort testing values near the minimum and very little in the larger, extreme areas of the space. Optuna quickly figures out that there is no way to find a better solution in these parts of the space and stops exploring those areas. This allows it to spend more time looking for the best solution and to better handle more than just two parameters.

5.4.2 Optuna with PyTorch

You now know all the basics of using Optuna; it's time to combine it with PyTorch to do some advanced parameter searching. We'll define a new function for Optuna to optimize our neural network. We do not want to go crazy, as optimizing without any gradients is still very difficult and Optuna is not a magic bullet. But we can use Optuna to help us make some decisions. For example, how many neurons should we have in each layer, and how many layers? Figure 5.17 shows how we can define a function to do this:

1. Create train/validation splits (we did that with the plateau schedule).
2. Ask Optuna to give us three critical hyperparameters.
3. Define our model using the parameters.
4. Compute and return the result from the validation split.

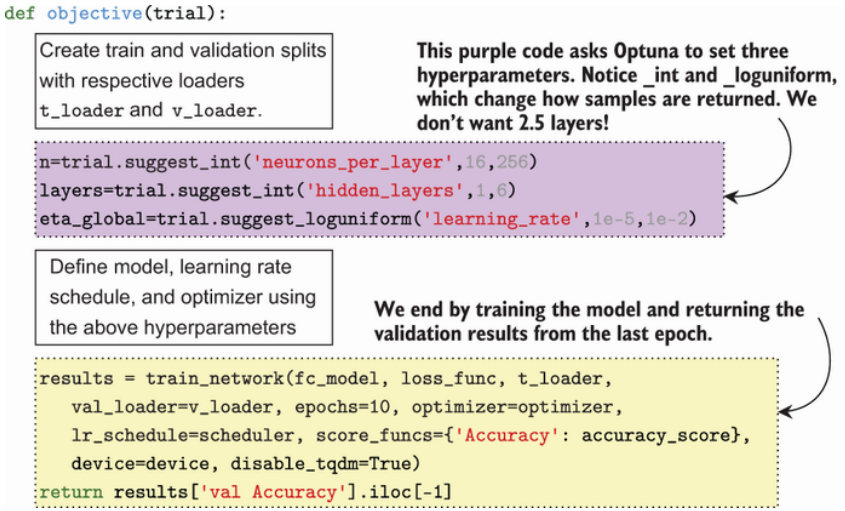


Figure 5.17 The four steps of defining an `objective` function that Optuna can use to optimize our neural network training. The two most important steps shown as code are getting the hyperparameters and computing the result!

An important thing to note here is that we have to create new training and validation splits using only the original training data. Why? Because we will reuse the validation set multiple times, and we do not want to overfit to the specifics of our validation data. So, we create a new validation set and save our original validation data until the very end. That way,

we only use the true validation data once to determine how well we did at optimizing our network architecture.

This function has very little *new* code; most of it is the same code we have used for several chapters to create data loaders, construct a network `Module`, and call our `train_network` function. Some significant PyTorch changes are that we set `disable_ tqdm=True` because Optuna does not play nicely with progress bars in the function it is trying to optimize.

Adding a dynamic number of layers

It may not be obvious at first, but we can very easily use the variable number of layers with our `nn.Sequential` object to adapt to what Optuna tells us to use. The code is as follows:

```
sequential_layers = [                ❶
    nn.Flatten(), nn.Linear(D, n), nn.Tanh(),
]

for _ in range(layers-1):            ❷
    sequential_layers.append( nn.Linear(n, n) )
    sequential_layers.append( nn.Tanh() ) sequential_layers.append(
nn.Linear(n, classes) )              ❸ fc_model =
nn.Sequential(*sequential_layers) ❹
```

❶ At least one hidden layer that takes in D inputs

❶ Adds a variable number of hidden layers, depending on what Optuna gave us for the “layers" parameter

❶ Output layer

❶ Turns the list of layers into a PyTorch sequential module

This breaks up the model specification into a few parts so that the number of hidden layers is a variable filled in with a `for` loop, `for _ in range(layers-1):`. It’s a little more verbose for small networks but makes the same code able to handle a variety of layers, and it’s less code if we want to add more layers.

The other changes are different `suggest` functions that Optuna provides via the `trial` object. There is `suggest_int` for integers, which makes sense for links like the number of neurons (76.8 neurons does not make sense) and the number of layers. We already saw `suggest_uniform`, which works for float values that are covered by a simple random range (like the momentum term μ, which should be between 0 and 1). The other important option is `suggest_loguniform`, which gives *exponentially spaced* random values. This is the function you should use for parameters that are altered by orders of magnitude, like the learning rate ($\eta = 0.001$, 0.01, and 0.1 differ by a factor of 10). The next code snippet shows how we can get three hyperparameter suggestions from Optuna by specifying the appropriate `suggest` function and providing minimum and maximum values that we are willing to consider:

```
n = trial.suggest_int('neurons_per_layer', 16, 256)
layers = trial.suggest_int('hidden_layers', 1, 6)
eta_global = trial.suggest_loguniform('learning_rate', 1e-5, 1e-2)
```

This ends by simply training our network and taking the validation accuracy from the last epoch. You must remember that this is a *validation split* and that we have not used the test set. We should only use the test set *after* the hyperparameters have been found, to determine the overall accuracy. The following code searches for the hyperparameters for this problem:

```
study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=10)                    ❶

print(study.best_params)

{'neurons\_per\_layer': 181, 'hidden\_layers': 3, 'learning\_rate':
0.005154640793181943\}
```

❶ Normally we would do more like 50 to 100 trials, but we are using fewer so this notebook runs in a reasonable amount of time.

You can see the parameters that Optuna selected. Now that we have trained our network, an exercise for you is to train a new model with this information to determine what final validation accuracy you get on the true validation set. Doing so completes the entire hyperparameter optimization process, which is outlined in figure 5.18.

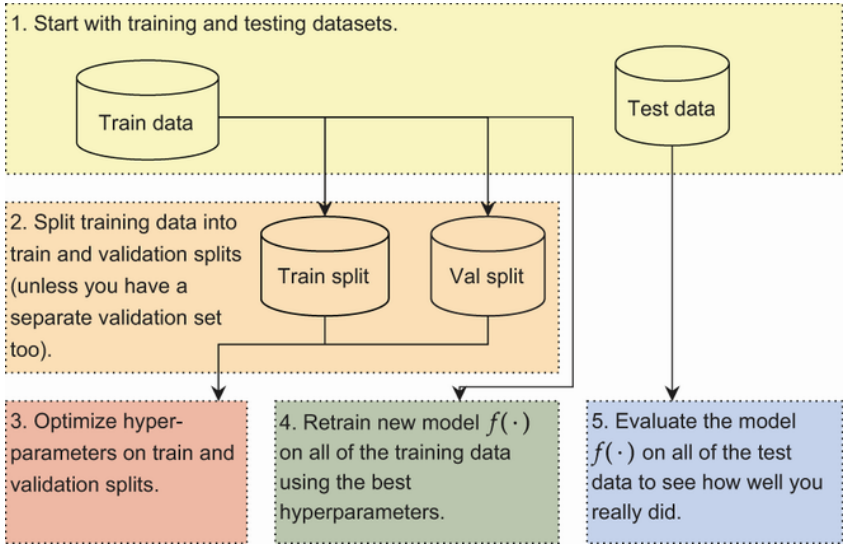


Figure 5.18 All the steps you need to follow to do hyperparameter optimization correctly. It may be tempting to merge or skip some of these steps, but doing so could give you misleading results about how well your model will actually do.

Visualizing Optuna results

Beyond just looking at the *final* answer, we can also look at the progress Optuna made over time and other views of the optimization process. Doing so can help us build some intuition about the range of "good" parameters. This information can be helpful when we set up new experiments so that we can hopefully start the optimization process closer to the true solution and thus reduce the number of optimization attempts required.

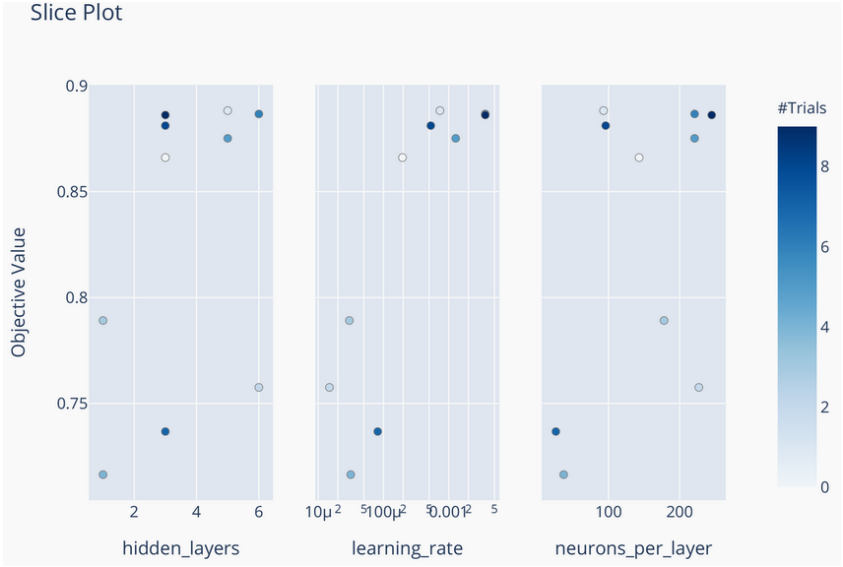
The following is one of the simplest options: plotting the validation accuracy (and individual trials) based on how many trials have been attempted. The red line on top shows the current best result, and each blue dot shows the result from one experiment. If big improvements in accuracy are still occurring (red line going up), we have a good reason to increase

the number of trials we let Optuna run. If accuracy has plateaued for a long time, we can reduce the number of trials in the future:

```
fig = optuna.visualization.plot_optimization_history(study)
fig.show()
```

We might also want to get an idea of how each hyperparameter performs with respect to the objective (accuracy). That can be done with a *slice plot*. The following example makes a scatter plot for each hyperparameter, with the objective on the y-axis; the color of the dot indicates which trial the result came from. This way, you can see if any one hyperparameter has a particularly strong relationship with your objective and how long it took Optuna to figure it out. In this specific example, in most cases, three to six hidden layers perform well, and learning rates above $\eta > 0.001$ are also consistently good. This kind of information can help us reduce the search range in future trials or even eliminate a hyperparameter if it seems to have little impact on the objective. Both of these will help Optuna converge to solutions with fewer attempts in future runs. Here's the code:

```
fig = optuna.visualization.plot_slice(study)
fig.show()
```



Optuna can also help you understand the interactions between hyperparameters. One option is the `plot_contour()` function, which creates a grid showing how every combination of two different hyperparameters impacts the results. The other option is the `plot_parallel_coordinate()` function, which shows all the results of every trial in one graph. Both of these are a little harder to read and require more trials to produce really interesting results, so I recommend trying them on your own when you have a chance to do 100 trials for a model.

5.4.3 Pruning trials with Optuna

Another feature that Optuna supports that is particularly useful when training neural networks is *pruning* trials early. The idea is that optimizing a neural network is iterative: we take multiple epochs through the dataset, and we (hopefully) improve with every epoch. This is valuable information we are not using. If we can determine early in the process that a model won't pan out, we can save a lot of time.

Say we start testing a set of parameters, and learning fails from the start, with terrible results on the first few epochs. Why continue training until the end? The model is *very unlikely* to start as degenerate and later become one of the best performers. If we report intermediate results to

Optuna, Optuna can prune out trials that look bad based on the trials that have already completed.

We can accomplish this by replacing the last two lines of code from the `def objective(trial):` function. Instead of calling `train_network` once with 10 epochs, we call it 10 times in a loop for 1 epoch each. After each epoch, we use the `report` function of the `trial` object to let Optuna know how we are *currently* doing and ask Optuna if we should stop. The revised code looks like this:

```
    for epoch in range(10):                                    ❶
        results = train_network(fc_model, loss_func, t_loader,
        ➥ val_loader=v_loader, epochs=1, optimizer=optimizer,
        ➥ lr_schedule=scheduler, score_funcs={'Accuracy': accuracy_score},
        ➥ device=device, disable_tqdm=True)                   ❷

        cur_accuracy = results['val Accuracy'].iloc[-1]

        trial.report(cur_accuracy, epoch)                      ❸

        if trial.should_prune():                               ❹
            raise optuna.exceptions.TrialPruned()              ❺

    return cur_accuracy                                        ❻
```

❶ Do a loop for each epoch ourselves.

❷ Do just one epoch of training, but reuse the same model and optimizer. This continues training the same model over and over.

❸ Lets Optuna know how well we are doing

❹ Asks Optuna if this looks hopeless

❺ If so, stop trying.

❻ We made it to the end: give the final answer.

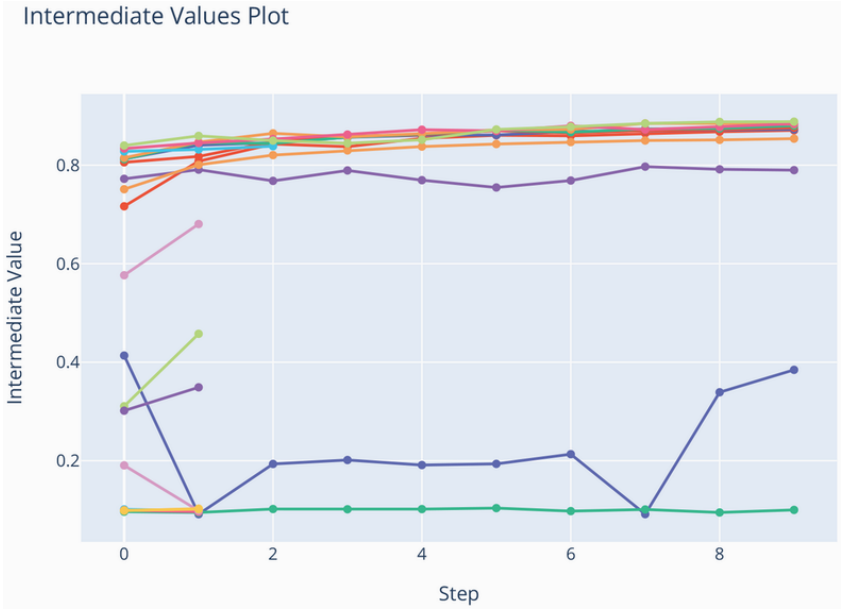
With this code change, I'm going to run a new trial with Optuna but intentionally set the number of neurons to go down to 1 (way too small) and the learning rate to go up to $\eta = 100$ (way too big). This will create some *re-*

ally bad models that are easy to prune out, just to show off this new pruning feature. All of these changes only needed to happen in the `objective` function: we call the same `create_study` and `optimize` functions and get pruning automatically. The following snippet shows this, but I set `n_trials=20` to give pruning more opportunities to occur since it depends on the best *current* models found by Optuna (it does not know what a bad run looks like until it sees a good run to compare against):

```
study2 = optuna.create_study(direction='maximize')
study2.optimize(objectivePrunable, n_trials=20)
```

Now you should see several `TrialState.PRUNED` logs from Optuna when you run this code. When I ran it, 10 of the 20 trials were pruned early. How many epochs into training were these models before they got pruned? We can have Optuna plot the results of all the trials with their intermediate values to help us understand that better. This is done with the `plot_intermediate_values()` function, as shown next:

```
fig = optuna.visualization.plot_intermediate_values(study2)
fig.show()
```



It looks like all 10 trials were pruned after just 1 or 2 epochs through the dataset. That's very early in the process: Optuna has reduced the number of effective trials almost by half. We also see some cases where models that did not perform well were allowed to run until completion, even though "better" models were pruned early. This happens because pruning is based on the best models Optuna has seen *so far*. Early on, Optuna lets bad models run to completion because it does not yet know they are bad models. Only after more trials and seeing much better models does it learn that the original ones could have been pruned. So pruning does not avoid *all* bad models, but it can avoid many of them.

Looking at the graph carefully, you should be able to see some cases where Optuna pruned models that were diverging (getting worse) and some that looked like they would improve but were not doing well enough that they would become competitive with what Optuna had already seen. These are also good cases for pruning and part of how Optuna saves us time.

NOTE While Optuna is one of my favorite tools, we won't use it again in this book. This is purely a computational concern, as I want to stick with examples that run in just a few minutes. Optuna needs to train multiple networks, which means *multiplying* training time. Just 10 trials is not a

lot, but an example that would take 6 minutes with no hyperparameters would take an hour or more with Optuna. However, when you are working on the job, you should *definitely* use Optuna to help you build your neural networks.

Exercises

Share and discuss your solutions on the Manning online platform at Inside Deep Learning Exercises (**https://liveproject.manning.com/project/945**). Once you submit your own answers, you will be able to see the solutions submitted by other readers, and see which ones the author judges to be the best.

1. Modify the `train_network` function to accept `lr_schedule=ReduceLROnPlateau` as a valid argument. If the `train_network` function gets this string argument, it should check whether validation and test sets have been provided and, if so, set up the `ReduceLROnPlateau` scheduler appropriately.
2. Rerun the experiments with `AdamW`, SGD with Nesterov momentum, and the cosine annealing schedule using batch sizes of $B = 1, 4, 32, 64, 128$. How does the change in batch size impact the effectiveness and accuracy of these three tools?
3. Write code that creates a neural network with $n = 256$ neurons and an argument to control how many hidden layers are in the network. Then train networks with 1, 6, 12, and 24 hidden layers using naive SGD and again using `AdamW` with cosine annealing. How do these new optimizers impact your ability to learn these deeper networks?
4. Retrain the three-layer bidirectional RNN from the last chapter with each of the new optimizers from this chapter. How do they impact the results?
5. Add gradient clipping to the experiments from exercise 4. Does it help or hurt the RNN?
6. Write your own function that uses Optuna to optimize the parameters of a fully connected neural network. Once it's done, create a new network with those hyperparameters, train it using all of the training data, and test it on the held-out test set. What results do you get on FashionMNIST, and how close is Optuna's guess at the accuracy compared to your test-set performance?

7. Redo exercise 6, but replace the hidden layers with convolutional ones and add a new argument that controls how many rounds of max pooling to perform. How does it perform on FashionMNIST compared with your results from exercise 6?

Summary

- The two main components of gradient descent are how we use gradients (optimizers) and how fast we follow them (learning rate schedules).
- By using information about the history of gradients, we can increase how quickly our models learn.
- Adding momentum to optimizers allows training even when gradients become very small.
- Gradient clipping mitigates exploding gradients, allowing training even when gradients become very large.
- By altering the learning rate, we can ease the optimization view of learning for further improvements.
- Instead of grid search, we can use powerful tools like Optuna to find the hyperparameters of a neural network such as the number of layers and number of neurons.
- By checking the results after each epoch, we can accelerate the hyperparameter tuning process by pruning bad models early.

---

[1] More on that in the chapter 6.↩

[2] If the test loss starts increasing, which indicates more severe overfitting, that is another good reason to slow down learning. That helps slow the overfitting. But ideally, the test loss has stabilized.↩

[3] I think using μ for momentum is confusing, but it's the most common notation, so I'm sticking with it so you can learn it.↩