11 | Optimization Algorithms

If you read the book in sequence up to this point you already used a number of advanced optimization algorithms to train deep learning models. They were the tools that allowed us to continue updating model parameters and to minimize the value of the loss function, as evaluated on the training set. Indeed, anyone content with treating optimization as a black box device to minimize objective functions in a simple setting might well content oneself with the knowledge that there exists an array of incantations of such a procedure (with names such as "Adam", "NAG", or "SGD").

To do well, however, some deeper knowledge is required. Optimization algorithms are important for deep learning. On one hand, training a complex deep learning model can take hours, days, or even weeks. The performance of the optimization algorithm directly affects the model's training efficiency. On the other hand, understanding the principles of different optimization algorithms and the role of their parameters will enable us to tune the hyperparameters in a targeted manner to improve the performance of deep learning models.

In this chapter, we explore common deep learning optimization algorithms in depth. Almost all optimization problems arising in deep learning are *nonconvex*. Nonetheless, the design and analysis of algorithms in the context of convex problems has proven to be very instructive. It is for that reason that this section includes a primer on convex optimization and the proof for a very simple stochastic gradient descent algorithm on a convex objective function.

11.1 Optimization and Deep Learning

In this section, we will discuss the relationship between optimization and deep learning as well as the challenges of using optimization in deep learning. For a deep learning problem, we will usually define a loss function first. Once we have the loss function, we can use an optimization algorithm in attempt to minimize the loss. In optimization, a loss function is often referred to as the objective function of the optimization problem. By tradition and convention most optimization algorithms are concerned with *minimization*. If we ever need to maximize an objective there is a simple solution: just flip the sign on the objective.

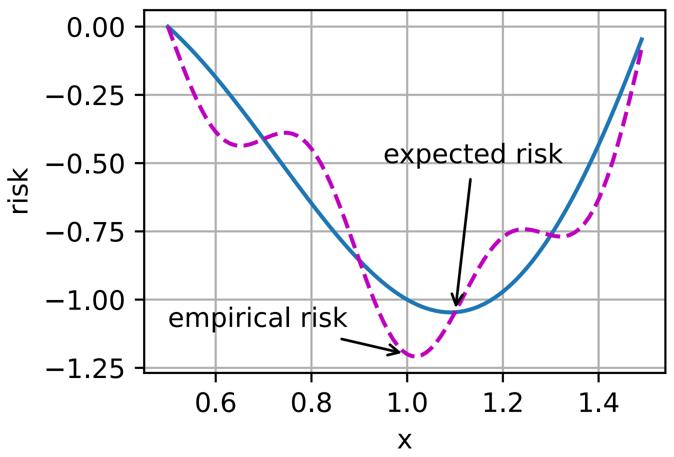
11.1.1 Optimization and Estimation

Although optimization provides a way to minimize the loss function for deep learning, in essence, the goals of optimization and deep learning are fundamentally different. The former is primarily concerned with minimizing an objective whereas the latter is concerned with finding a suitable model, given a finite amount of data. In Section 4.4, we discussed the difference between these two goals in detail. For instance, training error and generalization error generally differ: since the objective function of the optimization algorithm is usually a loss function based on the training dataset, the goal of optimization is to reduce the training error. However, the goal of statistical inference (and thus of deep learning) is to reduce the generalization error. To accomplish the latter we need to pay attention to overfitting in addition to using the optimization algorithm to reduce the training error. We begin by importing a few libraries with a function to annotate in a figure.

```
%matplotlib inline
import d2l
from mpl_toolkits import mplot3d
from mxnet import np, npx
npx.set_np()

# Saved in the d2l package for later use
def annotate(text, xy, xytext):
    d2l.plt.gca().annotate(text, xy=xy, xytext=xytext,
                           arrowprops=dict(arrowstyle='->'))
```

The graph below illustrates the issue in some more detail. Since we have only a finite amount of data the minimum of the training error may be at a different location than the minimum of the expected error (or of the test error).

```
def f(x): return x * np.cos(np.pi * x)
def g(x): return f(x) + 0.2 * np.cos(5 * np.pi * x)

d2l.set_figsize((4.5, 2.5))
x = np.arange(0.5, 1.5, 0.01)
d2l.plot(x, [f(x), g(x)], 'x', 'risk')
annotate('empirical risk', (1.0, -1.2), (0.5, -1.1))
annotate('expected risk', (1.1, -1.05), (0.95, -0.5))
```



11.1.2 Optimization Challenges in Deep Learning

In this chapter, we are going to focus specifically on the performance of the optimization algorithm in minimizing the objective function, rather than a model's generalization error. In Section 3.1 we distinguished between analytical solutions and numerical solutions in optimization problems. In deep learning, most objective functions are complicated and do not have analytical solutions. Instead, we must use numerical optimization algorithms. The optimization algorithms below all fall into this category.

There are many challenges in deep learning optimization. Some of the most vexing ones are local minima, saddle points and vanishing gradients. Let's have a look at a few of them.

Local Minima

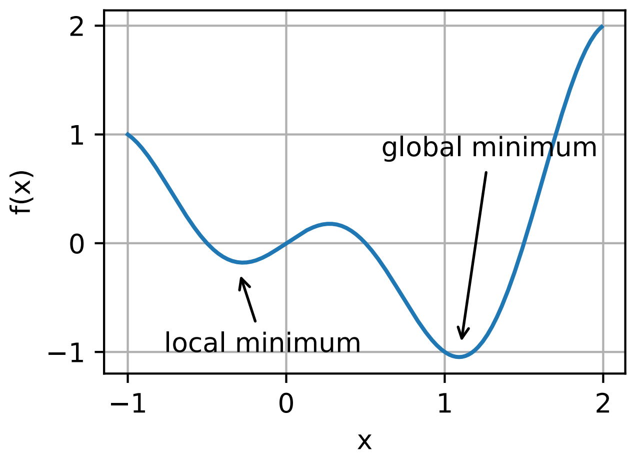
For the objective function $f(x)$, if the value of $f(x)$ at x is smaller than the values of $f(x)$ at any other points in the vicinity of x, then $f(x)$ could be a local minimum. If the value of $f(x)$ at x is the minimum of the objective function over the entire domain, then $f(x)$ is the global minimum.

For example, given the function

$$f(x) = x \cdot \text{cos}(\pi x) \text{ for } -1.0 \leq x \leq 2.0, \tag{11.1.1}$$

we can approximate the local minimum and global minimum of this function.

```
x = np.arange(-1.0, 2.0, 0.01)
d2l.plot(x, [f(x), ], 'x', 'f(x)')
annotate('local minimum', (-0.3, -0.25), (-0.77, -1.0))
annotate('global minimum', (1.1, -0.95), (0.6, 0.8))
```

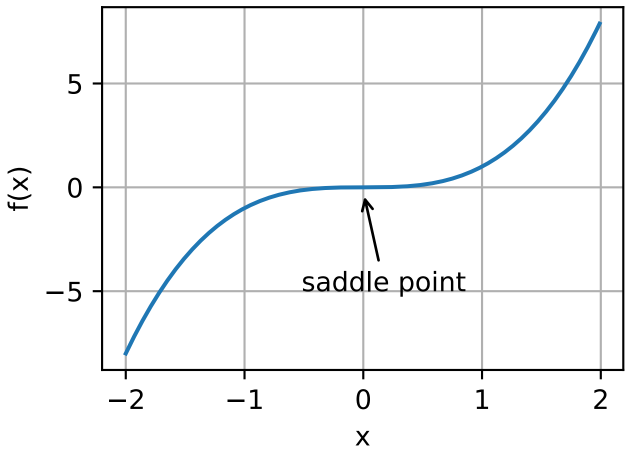


The objective function of deep learning models usually has many local optima. When the numerical solution of an optimization problem is near the local optimum, the numerical solution obtained by the final iteration may only minimize the objective function locally, rather than globally, as the gradient of the objective function's solutions approaches or becomes zero. Only some degree of noise might knock the parameter out of the local minimum. In fact, this is one of the beneficial properties of stochastic gradient descent where the natural variation of gradients over minibatches is able to dislodge the parameters from local minima.

Saddle Points

Besides local minima, saddle points are another reason for gradients to vanish. A saddle point[148] is any location where all gradients of a function vanish but which is neither a global nor a local minimum. Consider the function $f(x) = x^3$. Its first and second derivative vanish for $x = 0$. Optimization might stall at the point, even though it is not a minimum.

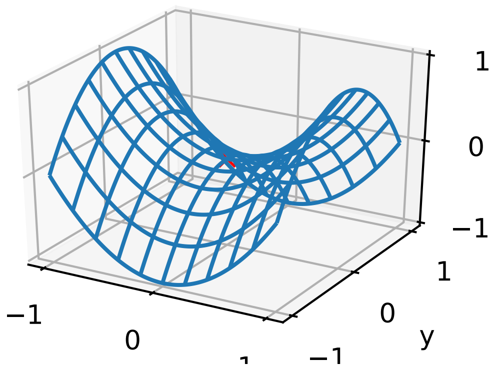
```
x = np.arange(-2.0, 2.0, 0.01)
d2l.plot(x, [x**3], 'x', 'f(x)')
annotate('saddle point', (0, -0.2), (-0.52, -5.0))
```



Saddle points in higher dimensions are even more insidious, as the example below shows. Consider the function $f(x, y) = x^2 - y^2$. It has its saddle point at $(0, 0)$. This is a maximum with respect to y and a minimum with respect to x. Moreover, it *looks* like a saddle, which is where this mathematical property got its name.

```
x, y = np.meshgrid(np.linspace(-1, 1, 101), np.linspace(-1, 1, 101),
                   indexing='ij')

z = x**2 - y**2

ax = d2l.plt.figure().add_subplot(111, projection='3d')
ax.plot_wireframe(x, y, z, **{'rstride': 10, 'cstride': 10})
ax.plot([0], [0], [0], 'rx')
ticks = [-1, 0, 1]
d2l.plt.xticks(ticks)
d2l.plt.yticks(ticks)
ax.set_zticks(ticks)
d2l.plt.xlabel('x')
d2l.plt.ylabel('y');
```

[148] https://en.wikipedia.org/wiki/Saddle_point



We assume that the input of a function is a k-dimensional vector and its output is a scalar, so its Hessian matrix will have k eigenvalues (refer to Section 17.1). The solution of the function could be a local minimum, a local maximum, or a saddle point at a position where the function gradient is zero:

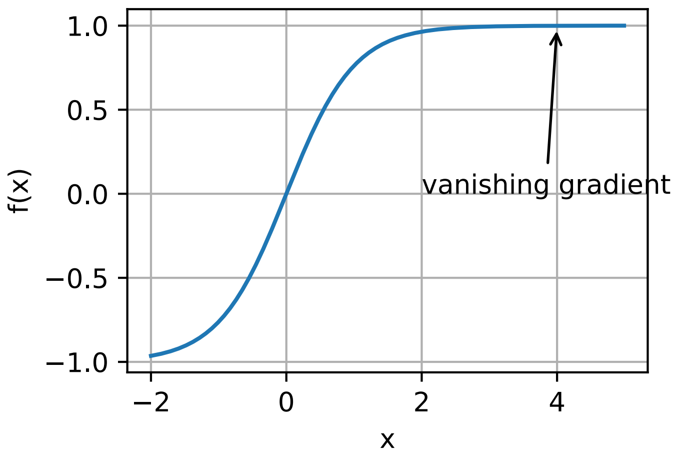
- When the eigenvalues of the function's Hessian matrix at the zero-gradient position are all positive, we have a local minimum for the function.
- When the eigenvalues of the function's Hessian matrix at the zero-gradient position are all negative, we have a local maximum for the function.
- When the eigenvalues of the function's Hessian matrix at the zero-gradient position are negative and positive, we have a saddle point for the function.

For high-dimensional problems the likelihood that at least some of the eigenvalues are negative is quite high. This makes saddle points more likely than local minima. We will discuss some exceptions to this situation in the next section when introducing convexity. In short, convex functions are those where the eigenvalues of the Hessian are never negative. Sadly, though, most deep learning problems do not fall into this category. Nonetheless it is a great tool to study optimization algorithms.

Vanishing Gradients

Probably the most insidious problem to encounter are vanishing gradients. For instance, assume that we want to minimize the function $f(x) = \tanh(x)$ and we happen to get started at $x = 4$. As we can see, the gradient of f is close to nil. More specifically $f'(x) = 1 - \tanh^2(x)$ and thus $f'(4) = 0.0013$. Consequently optimization will get stuck for a long time before we make progress. This turns out to be one of the reasons that training deep learning models was quite tricky prior to the introduction of the ReLU activation function.

```
x = np.arange(-2.0, 5.0, 0.01)
d2l.plot(x, [np.tanh(x)], 'x', 'f(x)')
annotate('vanishing gradient', (4, 1), (2, 0.0))
```



As we saw, optimization for deep learning is full of challenges. Fortunately there exists a robust range of algorithms that perform well and that are easy to use even for beginners. Furthermore, it is not really necessary to find *the* best solution. Local optima or even approximate solutions thereof are still very useful.

Summary

- Minimizing the training error does *not* guarantee that we find the best set of parameters to minimize the expected error.
- The optimization problems may have many local minima.
- The problem may have even more saddle points, as generally the problems are not convex.
- Vanishing gradients can cause optimization to stall. Often a reparametrization of the problem helps. Good initialization of the parameters can be beneficial, too.

Exercises

1. Consider a simple multilayer perceptron with a single hidden layer of, say, d dimensions in the hidden layer and a single output. Show that for any local minimum there are at least $d!$ equivalent solutions that behave identically.
2. Assume that we have a symmetric random matrix $\mathbf{M}$ where the entries $M_{ij} = M_{ji}$ are each drawn from some probability distribution p_{ij}. Furthermore assume that $p_{ij}(x) = p_{ij}(-x)$, i.e., that the distribution is symmetric (see e.g., (Wigner, 1958) for details).
    - Prove that the distribution over eigenvalues is also symmetric. That is, for any eigenvector $\mathbf{v}$ the probability that the associated eigenvalue λ satisfies $P(\lambda > 0) = P(\lambda < 0)$.
    - Why does the above *not* imply $P(\lambda > 0) = 0.5$?
3. What other challenges involved in deep learning optimization can you think of?
4. Assume that you want to balance a (real) ball on a (real) saddle.
    - Why is this hard?
    - Can you exploit this effect also for optimization algorithms?

11.2 Convexity

Convexity plays a vital role in the design of optimization algorithms. This is largely due to the fact that it is much easier to analyze and test algorithms in this context. In other words, if the algorithm performs poorly even in the convex setting we should not hope to see great results otherwise. Furthermore, even though the optimization problems in deep learning are generally nonconvex, they often exhibit some properties of convex ones near local minima. This can lead to exciting new optimization variants such as (Izmailov et al., 2018).

11.2.1 Basics

Let's begin with the basics.

Sets

Sets are the basis of convexity. Simply put, a set X in a vector space is convex if for any $a, b \in X$ the line segment connecting a and b is also in X. In mathematical terms this means that for all $\lambda \in [0, 1]$ we have

$$\lambda \cdot a + (1 - \lambda) \cdot b \in X \text{ whenever } a, b \in X. \tag{11.2.1}$$

This sounds a bit abstract. Consider the picture Fig. 11.2.1. The first set is not convex since there are line segments that are not contained in it. The other two sets suffer no such problem.

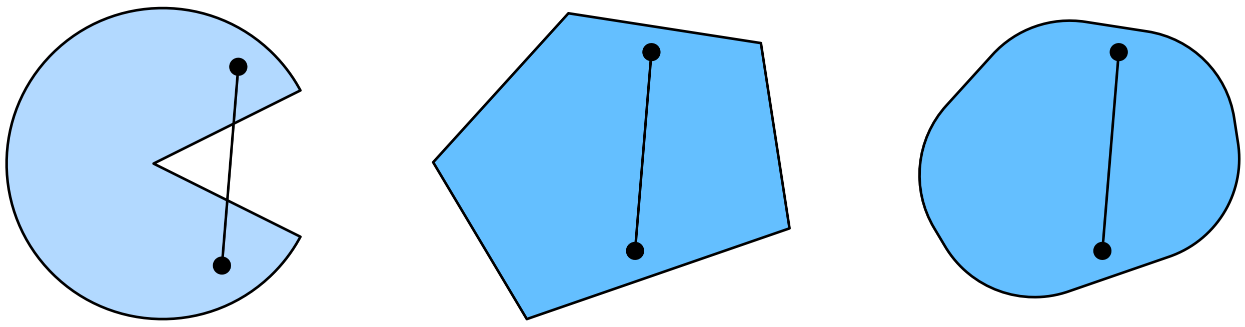


Fig. 11.2.1: Three shapes, the left one is nonconvex, the others are convex

Definitions on their own are not particularly useful unless you can do something with them. In this case we can look at unions and intersections as shown in Fig. 11.2.2. Assume that X and Y are convex sets. Then $X \cap Y$ is also convex. To see this, consider any $a, b \in X \cap Y$. Since X and Y are convex, the line segments connecting a and b are contained in both X and Y. Given that, they also need to be contained in $X \cap Y$, thus proving our first theorem.

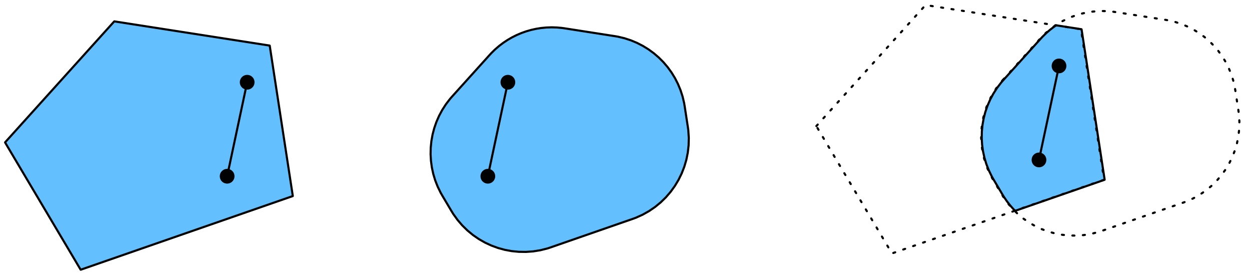


Fig. 11.2.2: The intersection between two convex sets is convex

We can strengthen this result with little effort: given convex sets X_i, their intersection $\cap_i X_i$ is convex. To see that the converse is not true, consider two disjoint sets $X \cap Y = \emptyset$. Now pick $a \in X$ and $b \in Y$. The line segment in Fig. 11.2.3 connecting a and b needs to contain some part that is neither in X nor Y, since we assumed that $X \cap Y = \emptyset$. Hence the line segment is not in $X \cup Y$ either, thus proving that in general unions of convex sets need not be convex.

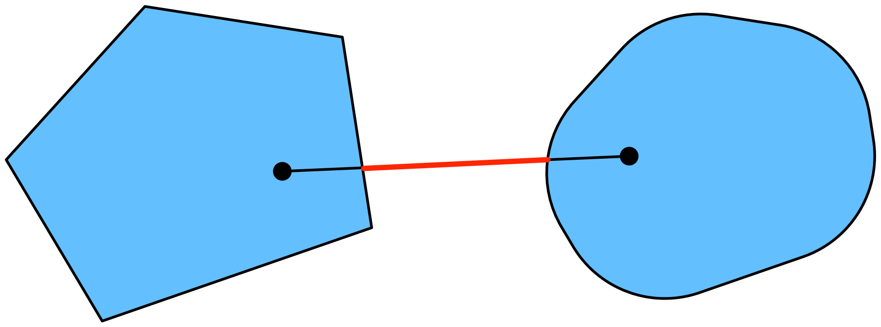


Fig. 11.2.3: The union of two convex sets need not be convex

Typically the problems in deep learning are defined on convex domains. For instance $\mathbb{R}^d$ is a convex set (after all, the line between any two points in $\mathbb{R}^d$ remains in $\mathbb{R}^d$). In some cases we work with variables of bounded length, such as balls of radius r as defined by $\{\mathbf{x} | \mathbf{x} \in \mathbb{R}^d \text{ and } \|\mathbf{x}\|_2 \leq r\}$.

Functions

Now that we have convex sets we can introduce convex functions f. Given a convex set X a function defined on it $f : X \to \mathbb{R}$ is convex if for all $x, x' \in X$ and for all $\lambda \in [0, 1]$ we have

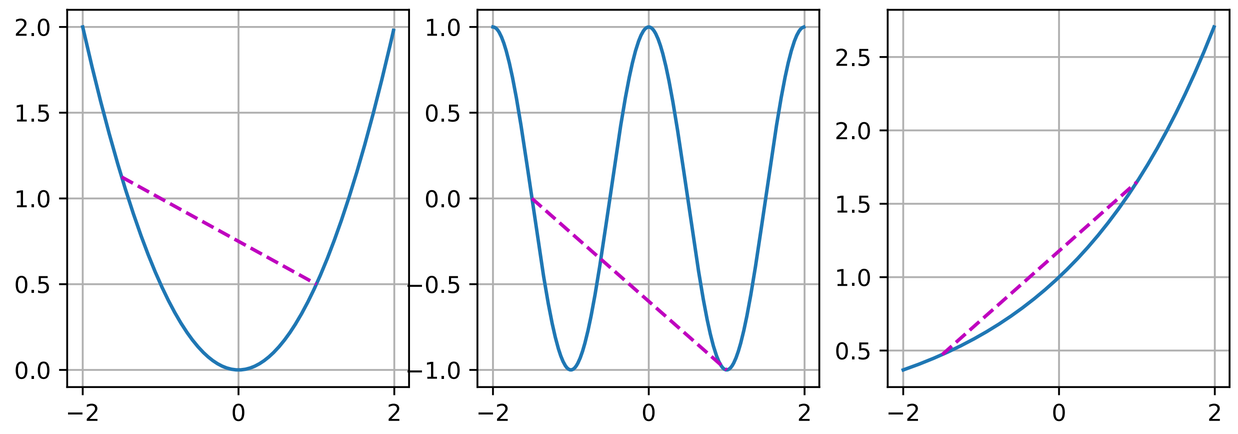
$$\lambda f(x) + (1 - \lambda) f(x') \geq f(\lambda x + (1 - \lambda) x'). \tag{11.2.2}$$

To illustrate this let's plot a few functions and check which ones satisfy the requirement. We need to import a few libraries.

```
%matplotlib inline
import d2l
from mpl_toolkits import mplot3d
from mxnet import np, npx
npx.set_np()
```

Let's define a few functions, both convex and nonconvex.

```
def f(x):
    return 0.5 * x**2  # Convex

def g(x):
    return np.cos(np.pi * x)  # Nonconvex

def h(x):
    return np.exp(0.5 * x)  # Convex

x, segment = np.arange(-2, 2, 0.01), np.array([-1.5, 1])
d2l.use_svg_display()
_, axes = d2l.plt.subplots(1, 3, figsize=(9, 3))

for ax, func in zip(axes, [f, g, h]):
    d2l.plot([x, segment], [func(x), func(segment)], axes=ax)
```



As expected, the cosine function is nonconvex, whereas the parabola and the exponential function are. Note that the requirement that $\mathcal{X}$ is necessary for the condition to make sense. Otherwise the outcome of $f(\lambda x + (1-\lambda)x')$ might not be well defined. Convex functions have a number of desirable properties.

Jensen's Inequality

One of the most useful tools is Jensen's inequality. It amounts to a generalization of the definition of convexity:

$$\sum_i \alpha_i f(x_i) \geq f\left(\sum_i \alpha_i x_i\right) \text{ and } E_x[f(x)] \geq f\left(E_x[x]\right). \tag{11.2.3}$$

In other words, the expectation of a convex function is larger than the convex function of an expectation. To prove the first inequality we repeatedly apply the definition of convexity to one term in the sum at a time. The expectation can be proven by taking the limit over finite segments.

One of the common applications of Jensen's inequality is with regard to the log-likelihood of partially observed random variables. That is, we use

$$E_{y \sim P(y)}[-\log P(x \mid y)] \geq -\log P(x). \tag{11.2.4}$$

This follows since $\int P(y)P(x \mid y)dy = P(x)$. This is used in variational methods. Here y is typically the unobserved random variable, $P(y)$ is the best guess of how it might be distributed and $P(x)$ is the distribution with y integrated out. For instance, in clustering y might be the cluster labels and $P(x \mid y)$ is the generative model when applying cluster labels.

11.2.2 Properties

Convex functions have a few useful properties. We describe them as follows.

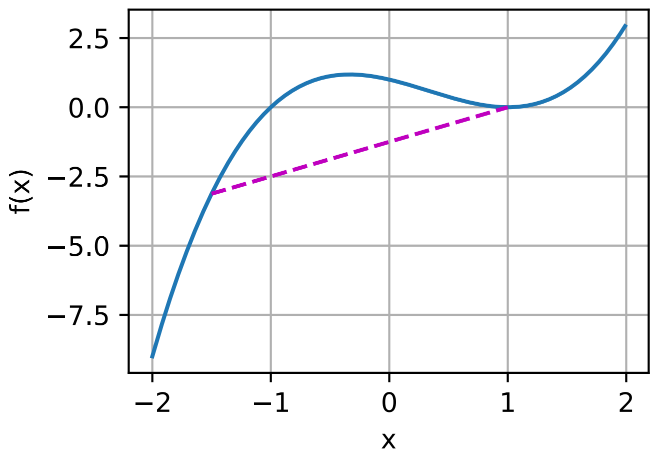
No Local Minima

In particular, convex functions do not have local minima. Let's assume the contrary and prove it wrong. If $x \in X$ is a local minimum there exists some neighborhood of x for which $f(x)$ is the smallest value. Since x is only a local minimum there has to be another $x' \in X$ for which $f(x') < f(x)$. However, by convexity the function values on the entire *line* $\lambda x + (1-\lambda)x'$ have to be less than $f(x')$ since for $\lambda \in [0, 1)$

$$f(x) > \lambda f(x) + (1-\lambda)f(x') \geq f(\lambda x + (1-\lambda)x'). \tag{11.2.5}$$

This contradicts the assumption that $f(x)$ is a local minimum. For instance, the function $f(x) = (x+1)(x-1)^2$ has a local minimum for $x = 1$. However, it is not a global minimum.

```
def f(x):
    return (x-1)**2 * (x+1)

d2l.set_figsize((3.5, 2.5))
d2l.plot([x, segment], [f(x), f(segment)], 'x', 'f(x)')
```



The fact that convex functions have no local minima is very convenient. It means that if we minimize functions we cannot "get stuck". Note, though, that this does not mean that there cannot be more than one global minimum or that there might even exist one. For instance, the function $f(x) = \max(|x|-1, 0)$ attains its minimum value over the interval $[-1, 1]$. Conversely, the function $f(x) = \exp(x)$ does not attain a minimum value on $\mathbb{R}$. For $x \to -\infty$ it asymptotes to 0, however there is no x for which $f(x) = 0$.

Convex Functions and Sets

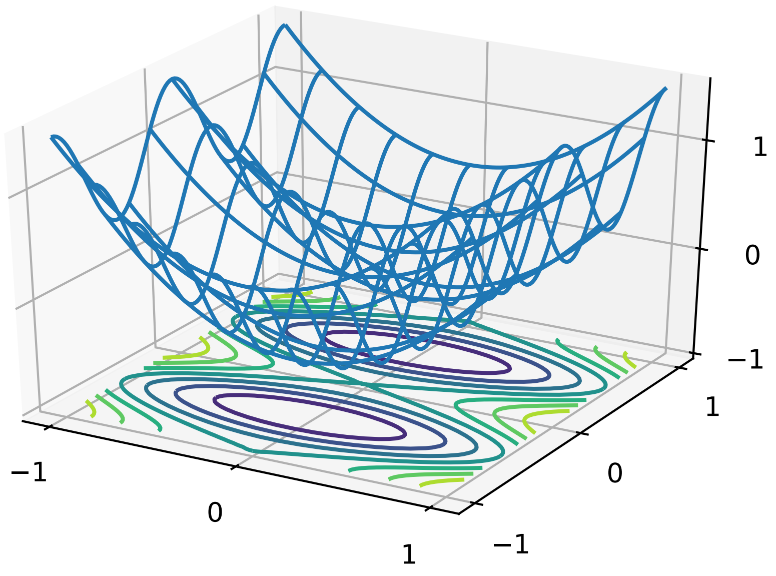
Convex functions define convex sets as *below-sets*. They are defined as

$$S_b := \{x | x \in X \text{ and } f(x) \le b\}. \tag{11.2.6}$$

Such sets are convex. Let's prove this quickly. Remember that for any $x, x' \in S_b$ we need to show that $\lambda x + (1-\lambda)x' \in S_b$ as long as $\lambda \in [0, 1]$. But this follows directly from the definition of convexity since $f(\lambda x + (1-\lambda)x') \le \lambda f(x) + (1-\lambda) f(x') \le b$.

Have a look at the function $f(x, y) = 0.5x^2 + \cos(2\pi y)$ below. It is clearly nonconvex. The level sets are correspondingly nonconvex. In fact, they are typically composed of disjoint sets.

```
x, y = np.meshgrid(np.linspace(-1, 1, 101), np.linspace(-1, 1, 101),
                   indexing='ij')

z = x**2 + 0.5 * np.cos(2 * np.pi * y)

# Plot the 3D surface
d2l.set_figsize((6, 4))
ax = d2l.plt.figure().add_subplot(111, projection='3d')
ax.plot_wireframe(x, y, z, **{'rstride': 10, 'cstride': 10})
ax.contour(x, y, z, offset=-1)
ax.set_zlim(-1, 1.5)

# Adjust labels
for func in [d2l.plt.xticks, d2l.plt.yticks, ax.set_zticks]:
    func([-1, 0, 1])
```



Derivatives and Convexity

Whenever the second derivative of a function exists it is very easy to check for convexity. All we need to do is check whether $\partial_x^2 f(x) \succeq 0$, i.e., whether all of its eigenvalues are nonnegative. For instance, the function $f(\mathbf{x}) = \frac{1}{2}\|\mathbf{x}\|_2^2$ is convex since $\partial_{\mathbf{x}}^2 f = \mathbf{1}$, i.e., its derivative is the identity matrix.

The first thing to realize is that we only need to prove this property for one-dimensional functions. After all, in general we can always define some function $g(z) = f(\mathbf{x} + z \cdot \mathbf{v})$. This function has the first and second derivatives $g' = (\partial_{\mathbf{x}} f)^\top \mathbf{v}$ and $g'' = \mathbf{v}^\top (\partial_{\mathbf{x}}^2 f)\mathbf{v}$ respectively. In particular, $g'' \geq 0$ for all $\mathbf{v}$ whenever the Hessian of f is positive semidefinite, i.e., whenever all of its eigenvalues are greater equal than zero. Hence back to the scalar case.

To see that $f''(x) \geq 0$ for convex functions we use the fact that

$$\frac{1}{2} f(x+\epsilon) + \frac{1}{2} f(x-\epsilon) \geq f\left(\frac{x+\epsilon}{2} + \frac{x-\epsilon}{2}\right) = f(x). \tag{11.2.7}$$

Since the second derivative is given by the limit over finite differences it follows that

$$f''(x) = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) + f(x-\epsilon) - 2f(x)}{\epsilon^2} \geq 0. \tag{11.2.8}$$

To see that the converse is true we use the fact that $f'' \geq 0$ implies that f' is a monotonically increasing function. Let $a < x < b$ be three points in $\mathbb{R}$. We use the mean value theorem to express

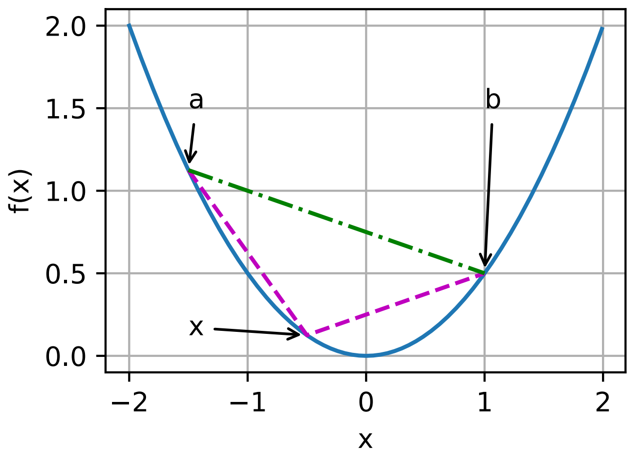
$$\begin{aligned} f(x) - f(a) = (x-a)f'(\alpha) \text{ for some } \alpha \in [a, x] \text{ and} \\ f(b) - f(x) = (b-x)f'(\beta) \text{ for some } \beta \in [x, b]. \end{aligned} \tag{11.2.9}$$

By monotonicity $f'(\beta) \geq f'(\alpha)$, hence

$$\begin{aligned} f(b) - f(a) &= f(b) - f(x) + f(x) - f(a) \\ &= (b-x)f'(\beta) + (x-a)f'(\alpha) \\ &\geq (b-a)f'(\alpha). \end{aligned} \tag{11.2.10}$$

By geometry it follows that $f(x)$ is below the line connecting $f(a)$ and $f(b)$, thus proving convexity. We omit a more formal derivation in favor of a graph below.

```python
def f(x):
    return 0.5 * x**2

x = np.arange(-2, 2, 0.01)
axb, ab = np.array([-1.5, -0.5, 1]), np.array([-1.5, 1])

d2l.set_figsize((3.5, 2.5))
d2l.plot([x, axb, ab], [f(x) for x in [x, axb, ab]], 'x', 'f(x)')
d2l.annotate('a', (-1.5, f(-1.5)), (-1.5, 1.5))
d2l.annotate('b', (1, f(1)), (1, 1.5))
d2l.annotate('x', (-0.5, f(-0.5)), (-1.5, f(-0.5)))
```



11.2.3 Constraints

One of the nice properties of convex optimization is that it allows us to handle constraints efficiently. That is, it allows us to solve problems of the form:

$$\begin{aligned} &\underset{\mathbf{x}}{\text{minimize}}\ f(\mathbf{x}) \\ &\text{subject to } c_i(\mathbf{x}) \leq 0 \text{ for all } i \in \{1, \ldots, N\}. \end{aligned} \tag{11.2.11}$$

Here f is the objective and the functions c_i are constraint functions. To see what this does consider the case where $c_1(\mathbf{x}) = \|\mathbf{x}\|_2 - 1$. In this case the parameters $\mathbf{x}$ are constrained to the unit ball. If a second constraint is $c_2(\mathbf{x}) = \mathbf{v}^\top \mathbf{x} + b$, then this corresponds to all $\mathbf{x}$ lying on a halfspace. Satisfying both constraints simultaneously amounts to selecting a slice of a ball as the constraint set.

Lagrange Function

In general, solving a constrained optimization problem is difficult. One way of addressing it stems from physics with a rather simple intuition. Imagine a ball inside a box. The ball will roll to the place that is lowest and the forces of gravity will be balanced out with the forces that the sides of the box can impose on the ball. In short, the gradient of the objective function (i.e., gravity) will be offset by the gradient of the constraint function (need to remain inside the box by virtue of the walls "pushing back"). Note that any constraint that is not active (i.e., the ball does not touch the wall) will not be able to exert any force on the ball.

Skipping over the derivation of the Lagrange function L (see e.g., the book by Boyd and Vandenberghe for details (Boyd & Vandenberghe, 2004)) the above reasoning can be expressed via the following saddlepoint optimization problem:

$$L(\mathbf{x}, \alpha) = f(\mathbf{x}) + \sum_i \alpha_i c_i(\mathbf{x}) \text{ where } \alpha_i \geq 0. \tag{11.2.12}$$

Here the variables α_i are the so-called *Lagrange Multipliers* that ensure that a constraint is properly enforced. They are chosen just large enough to ensure that $c_i(\mathbf{x}) \leq 0$ for all i. For instance, for any $\mathbf{x}$ for which $c_i(\mathbf{x}) < 0$ naturally, we'd end up picking $\alpha_i = 0$. Moreover, this is a *saddlepoint* optimization problem where one wants to *maximize* L with respect to α and simultaneously *minimize* it with respect to $\mathbf{x}$. There is a rich body of literature explaining how to arrive at the function $L(\mathbf{x}, \alpha)$. For our purposes it is sufficient to know that the saddlepoint of L is where the original constrained optimization problem is solved optimally.

Penalties

One way of satisfying constrained optimization problems at least approximately is to adapt the Lagrange function L. Rather than satisfying $c_i(\mathbf{x}) \leq 0$ we simply add $\alpha_i c_i(\mathbf{x})$ to the objective function $f(x)$. This ensures that the constraints will not be violated too badly.

In fact, we have been using this trick all along. Consider weight decay in Section 4.5. In it we add $\frac{\lambda}{2}\|\mathbf{w}\|^2$ to the objective function to ensure that **w** does not grow too large. Using the constrained optimization point of view we can see that this will ensure that $\|\mathbf{w}\|^2 - r^2 \leq 0$ for some radius r. Adjusting the value of λ allows us to vary the size of **w**.

In general, adding penalties is a good way of ensuring approximate constraint satisfaction. In practice this turns out to be much more robust than exact satisfaction. Furthermore, for nonconvex problems many of the properties that make the exact approach so appealing in the convex case (e.g., optimality) no longer hold.

Projections

An alternative strategy for satisfying constraints are projections. Again, we encountered them before, e.g., when dealing with gradient clipping in Section 8.5. There we ensured that a gradient has length bounded by c via

$$\mathbf{g} \leftarrow \mathbf{g} \cdot \mathrm{min}(1, c/\|\mathbf{g}\|). \tag{11.2.13}$$

This turns out to be a *projection* of g onto the ball of radius c. More generally, a projection on a (convex) set X is defined as

$$\mathrm{Proj}_X(\mathbf{x}) = \mathop{\mathrm{argmin}}_{\mathbf{x}' \in X} \|\mathbf{x} - \mathbf{x}'\|_2. \tag{11.2.14}$$

It is thus the closest point in X to **x**. This sounds a bit abstract. Fig. 11.2.4 explains it somewhat more clearly. In it we have two convex sets, a circle and a diamond. Points inside the set (yellow) remain unchanged. Points outside the set (black) are mapped to the closest point inside the set (red). While for ℓ_2 balls this leaves the direction unchanged, this need not be the case in general, as can be seen in the case of the diamond.

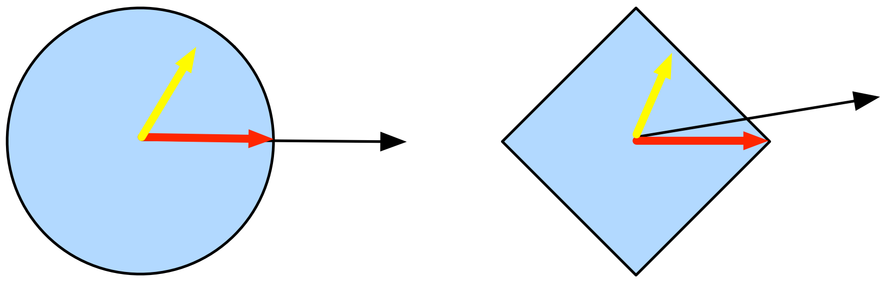


Fig. 11.2.4: Convex Projections

One of the uses for convex projections is to compute sparse weight vectors. In this case we project **w** onto an ℓ_1 ball (the latter is a generalized version of the diamond in the picture above).

Summary

In the context of deep learning the main purpose of convex functions is to motivate optimization algorithms and help us understand them in detail. In the following we will see how gradient descent and stochastic gradient descent can be derived accordingly.

- Intersections of convex sets are convex. Unions are not.
- The expectation of a convex function is larger than the convex function of an expectation (Jensen's inequality).
- A twice-differentiable function is convex if and only if its second derivative has only non-negative eigenvalues throughout.
- Convex constraints can be added via the Lagrange function. In practice simply add them with a penalty to the objective function.
- Projections map to points in the (convex) set closest to the original point.

Exercises

1. Assume that we want to verify convexity of a set by drawing all lines between points within the set and checking whether the lines are contained.
    - Prove that it is sufficient to check only the points on the boundary.
    - Prove that it is sufficient to check only the vertices of the set.
2. Denote by $B_p[r] := \{\mathbf{x} | \mathbf{x} \in \mathbb{R}^d \text{ and } \|\mathbf{x}\|_p \leq r\}$ the ball of radius r using the p-norm. Prove that $B_p[r]$ is convex for all $p \geq 1$.
3. Given convex functions f and g show that $\max(f, g)$ is convex, too. Prove that $\min(f, g)$ is not convex.
4. Prove that the normalization of the softmax function is convex. More specifically prove the convexity of $f(x) = \log \sum_i \exp(x_i)$.
5. Prove that linear subspaces are convex sets, i.e., $X = \{\mathbf{x} | \mathbf{W}\mathbf{x} = \mathbf{b}\}$.
6. Prove that in the case of linear subspaces with $\mathbf{b} = 0$ the projection Proj_X can be written as $\mathbf{M}\mathbf{x}$ for some matrix $\mathbf{M}$.
7. Show that for convex twice differentiable functions f we can write $f(x+\epsilon) = f(x) + \epsilon f'(x) + \frac{1}{2}\epsilon^2 f''(x+\xi)$ for some $\xi \in [0, \epsilon]$.
8. Given a vector $\mathbf{w} \in \mathbb{R}^d$ with $\|\mathbf{w}\|_1 > 1$ compute the projection on the ℓ_1 unit ball.
    - As intermediate step write out the penalized objective $\|\mathbf{w} - \mathbf{w}'\|_2^2 + \lambda \|\mathbf{w}'\|_1$ and compute the solution for a given $\lambda > 0$.
    - Can you find the 'right' value of λ without a lot of trial and error?
9. Given a convex set X and two vectors $\mathbf{x}$ and $\mathbf{y}$ prove that projections never increase distances, i.e., $\|\mathbf{x} - \mathbf{y}\| \geq \|\mathrm{Proj}_X(\mathbf{x}) - \mathrm{Proj}_X(\mathbf{y})\|$.

11.3 Gradient Descent

In this section we are going to introduce the basic concepts underlying gradient descent. This is brief by necessity. See e.g., (Boyd & Vandenberghe, 2004) for an in-depth introduction to convex optimization. Although the latter is rarely used directly in deep learning, an understanding of gradient descent is key to understanding stochastic gradient descent algorithms. For instance, the optimization problem might diverge due to an overly large learning rate. This phenomenon can already be seen in gradient descent. Likewise, preconditioning is a common technique in gradient descent and carries over to more advanced algorithms. Let's start with a simple special case.

11.3.1 Gradient Descent in One Dimension

Gradient descent in one dimension is an excellent example to explain why the gradient descent algorithm may reduce the value of the objective function. Consider some continuously differentiable real-valued function $f : \mathbb{R} \rightarrow \mathbb{R}$. Using a Taylor expansion (Section 17.3) we obtain that

$$f(x + \epsilon) = f(x) + \epsilon f'(x) + \mathcal{O}(\epsilon^2). \tag{11.3.1}$$

That is, in first approximation $f(x + \epsilon)$ is given by the function value $f(x)$ and the first derivative $f'(x)$ at x. It is not unreasonable to assume that for small ϵ moving in the direction of the negative gradient will decrease f. To keep things simple we pick a fixed step size $\eta > 0$ and choose $\epsilon = -\eta f'(x)$. Plugging this into the Taylor expansion above we get

$$f(x - \eta f'(x)) = f(x) - \eta f'^2(x) + \mathcal{O}(\eta^2 f'^2(x)). \tag{11.3.2}$$

If the derivative $f'(x) \neq 0$ does not vanish we make progress since $\eta f'^2(x) > 0$. Moreover, we can always choose η small enough for the higher order terms to become irrelevant. Hence we arrive at

$$f(x - \eta f'(x)) \lessapprox f(x). \tag{11.3.3}$$

This means that, if we use

$$x \leftarrow x - \eta f'(x) \tag{11.3.4}$$

to iterate x, the value of function $f(x)$ might decline. Therefore, in gradient descent we first choose an initial value x and a constant $\eta > 0$ and then use them to continuously iterate x until the stop condition is reached, for example, when the magnitude of the gradient $|f'(x)|$ is small enough or the number of iterations has reached a certain value.

For simplicity we choose the objective function $f(x) = x^2$ to illustrate how to implement gradient descent. Although we know that $x = 0$ is the solution to minimize $f(x)$, we still use this simple function to observe how x changes. As always, we begin by importing all required modules.

```
%matplotlib inline
import d2l
from mxnet import np, npx
npx.set_np()

def f(x):
    return x**2  # Objective function

def gradf(x):
    return 2 * x  # Its derivative
```

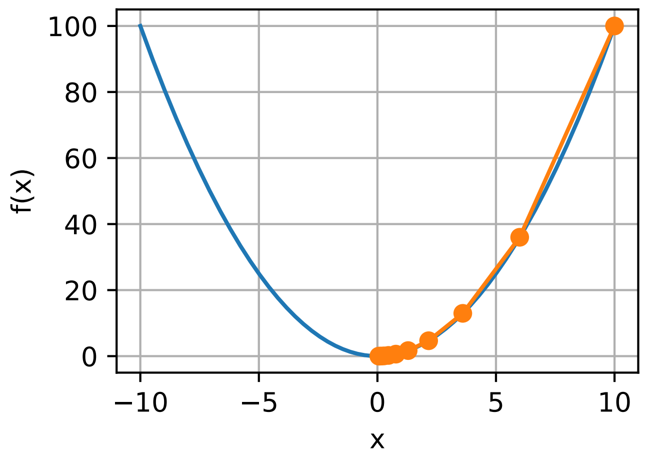
Next, we use $x = 10$ as the initial value and assume $\eta = 0.2$. Using gradient descent to iterate x for 10 times we can see that, eventually, the value of x approaches the optimal solution.

```
def gd(eta):
    x = 10
    results = [x]
    for i in range(10):
        x -= eta * gradf(x)
        results.append(x)
    print('epoch 10, x:', x)
    return results

res = gd(0.2)
```

```
epoch 10, x: 0.06046617599999997
```

The progress of optimizing over x can be plotted as follows.

```
def show_trace(res):
    n = max(abs(min(res)), abs(max(res)))
    f_line = np.arange(-n, n, 0.01)
    d2l.set_figsize((3.5, 2.5))
    d2l.plot([f_line, res], [[f(x) for x in f_line], [f(x) for x in res]],
             'x', 'f(x)', fmts=['-', '-o'])

show_trace(res)
```

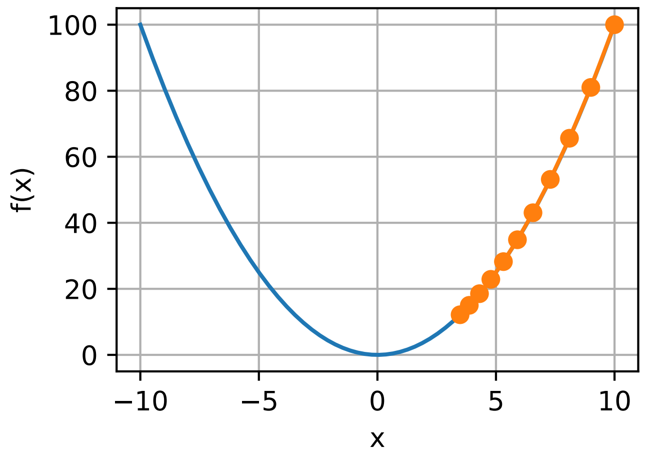


Learning Rate

The learning rate η can be set by the algorithm designer. If we use a learning rate that is too small, it will cause x to update very slowly, requiring more iterations to get a better solution. To show what happens in such a case, consider the progress in the same optimization problem for $\eta = 0.05$. As we can see, even after 10 steps we are still very far from the optimal solution.

```
show_trace(gd(0.05))
```

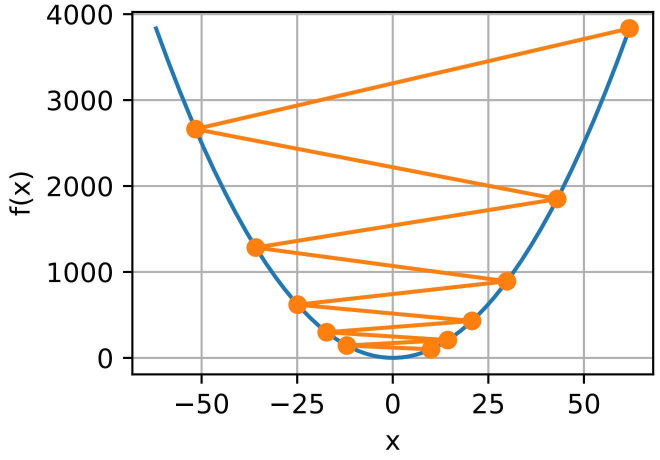
```
epoch 10, x: 3.4867844009999995
```



Conversely, if we use an excessively high learning rate, $|\eta f'(x)|$ might be too large for the first-order Taylor expansion formula. That is, the term $\mathcal{O}(\eta^2 f'^2(x))$ in (11.3.1) might become significant. In this case, we cannot guarantee that the iteration of x will be able to lower the value of $f(x)$. For example, when we set the learning rate to $\eta = 1.1$, x overshoots the optimal solution $x = 0$ and gradually diverges.

```
show_trace(gd(1.1))
```

```
epoch 10, x: 61.917364224000096
```

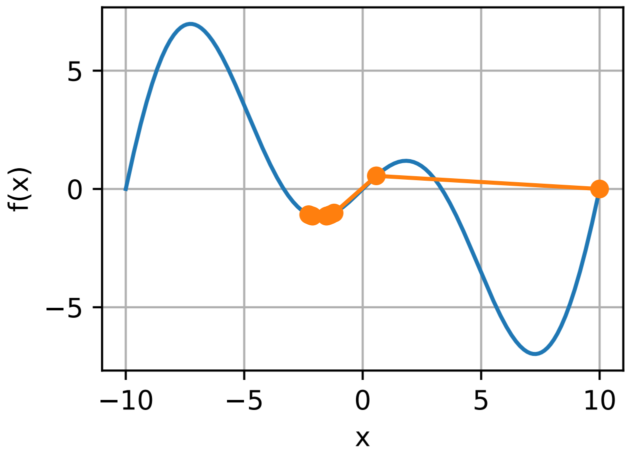


Local Minima

To illustrate what happens for nonconvex functions consider the case of $f(x) = x \cdot \cos cx$. This function has infinitely many local minima. Depending on our choice of learning rate and depending on how well conditioned the problem is, we may end up with one of many solutions. The example below illustrates how an (unrealistically) high learning rate will lead to a poor local minimum.

```
c = 0.15 * np.pi

def f(x):
    return x * np.cos(c * x)

def gradf(x):
    return np.cos(c * x) - c * x * np.sin(c * x)

show_trace(gd(2))
```

```
epoch 10, x: -1.528165927635083
```



11.3.2 Multivariate Gradient Descent

Now that have a better intuition of the univariate case, let's consider the situation where $\mathbf{x} \in \mathbb{R}^d$. That is, the objective function $f : \mathbb{R}^d \to \mathbb{R}$ maps vectors into scalars. Correspondingly its gradient is multivariate, too. It is a vector consisting of d partial derivatives:

$$\nabla f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_d}\right]^\top. \tag{11.3.5}$$

Each partial derivative element $\partial f(\mathbf{x})/\partial x_i$ in the gradient indicates the rate of change of f at $\mathbf{x}$ with respect to the input x_i. As before in the univariate case we can use the corresponding Taylor approximation for multivariate functions to get some idea of what we should do. In particular, we have that

$$f(\mathbf{x} + \epsilon) = f(\mathbf{x}) + \epsilon^\top \nabla f(\mathbf{x}) + \mathcal{O}(\|\epsilon\|^2). \tag{11.3.6}$$

In other words, up to second order terms in **epsilon** the direction of steepest descent is given by the negative gradient $-\nabla f(\mathbf{x})$. Choosing a suitable learning rate $\eta > 0$ yields the prototypical gradient descent algorithm:

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f(\mathbf{x}).$$

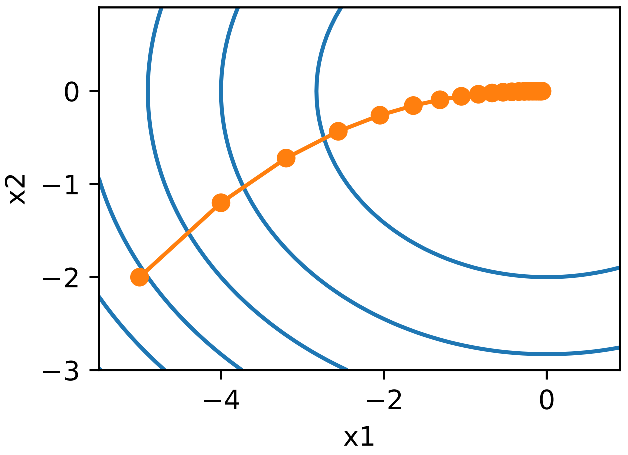
To see how the algorithm behaves in practice let's construct an objective function $f(\mathbf{x}) = x_1^2 + 2x_2^2$ with a two-dimensional vector $\mathbf{x} = [x_1, x_2]^\top$ as input and a scalar as output. The gradient is given by $\nabla f(\mathbf{x}) = [2x_1, 4x_2]^\top$. We will observe the trajectory of $\mathbf{x}$ by gradient descent from the initial position $[-5, -2]$. We need two more helper functions. The first uses an update function and applies it 20 times to the initial value. The second helper visualizes the trajectory of $\mathbf{x}$.

```
# Saved in the d2l package for later use
def train_2d(trainer, steps=20):
    """Optimize a 2-dim objective function with a customized trainer."""
    # s1 and s2 are internal state variables and will
    # be used later in the chapter
    x1, x2, s1, s2 = -5, -2, 0, 0
    results = [(x1, x2)]
    for i in range(steps):
        x1, x2, s1, s2 = trainer(x1, x2, s1, s2)
        results.append((x1, x2))
    print('epoch %d, x1 %f, x2 %f' % (i + 1, x1, x2))
    return results

# Saved in the d2l package for later use
def show_trace_2d(f, results):
    """Show the trace of 2D variables during optimization."""
    d2l.set_figsize((3.5, 2.5))
    d2l.plt.plot(*zip(*results), '-o', color='#ff7f0e')
    x1, x2 = np.meshgrid(np.arange(-5.5, 1.0, 0.1), np.arange(-3.0, 1.0, 0.1))
    d2l.plt.contour(x1, x2, f(x1, x2), colors='#1f77b4')
    d2l.plt.xlabel('x1')
    d2l.plt.ylabel('x2')
```

Next, we observe the trajectory of the optimization variable $\mathbf{x}$ for learning rate $\eta = 0.1$. We can see that after 20 steps the value of $\mathbf{x}$ approaches its minimum at $[0, 0]$. Progress is fairly well-behaved albeit rather slow.

```
def f(x1, x2):
    return x1 ** 2 + 2 * x2 ** 2  # Objective

def gradf(x1, x2):
    return (2 * x1, 4 * x2)  # Gradient

def gd(x1, x2, s1, s2):
    (g1, g2) = gradf(x1, x2)  # Compute gradient
    return (x1 - eta * g1, x2 - eta * g2, 0, 0)  # Update variables

eta = 0.1
show_trace_2d(f, train_2d(gd))
```

```
epoch 20, x1 -0.057646, x2 -0.000073
```



11.3.3 Adaptive Methods

As we could see in Section 11.3.1, getting the learning rate η "just right" is tricky. If we pick it too small, we make no progress. If we pick it too large, the solution oscillates and in the worst case it might even diverge. What if we could determine η automatically or get rid of having to select a step size at all? Second order methods that look not only at the value and gradient of the objective but also at its *curvature* can help in this case. While these methods cannot be applied to deep learning directly due to the computational cost, they provide useful intuition into how to design advanced optimization algorithms that mimic many of the desirable properties of the algorithms outlined below.

Newton's Method

Reviewing the Taylor expansion of f there is no need to stop after the first term. In fact, we can write it as

$$f(\mathbf{x} + \epsilon) = f(\mathbf{x}) + \epsilon^\top \nabla f(\mathbf{x}) + \frac{1}{2}\epsilon^\top \nabla\nabla^\top f(\mathbf{x})\epsilon + \mathcal{O}(\|\epsilon\|^3). \tag{11.3.7}$$

To avoid cumbersome notation we define $H_f := \nabla\nabla^\top f(\mathbf{x})$ to be the *Hessian* of f. This is a $d \times d$ matrix. For small d and simple problems H_f is easy to compute. For deep networks, on the other hand, H_f may be prohibitively large, due to the cost of storing $\mathcal{O}(d^2)$ entries. Furthermore it may be too expensive to compute via backprop as we would need to apply backprop to the backpropagation call graph. For now let's ignore such considerations and look at what algorithm we'd get.

After all, the minimum of f satisfies $\nabla f(\mathbf{x}) = 0$. Taking derivatives of (11.3.7) with regard to ϵ and ignoring higher order terms we arrive at

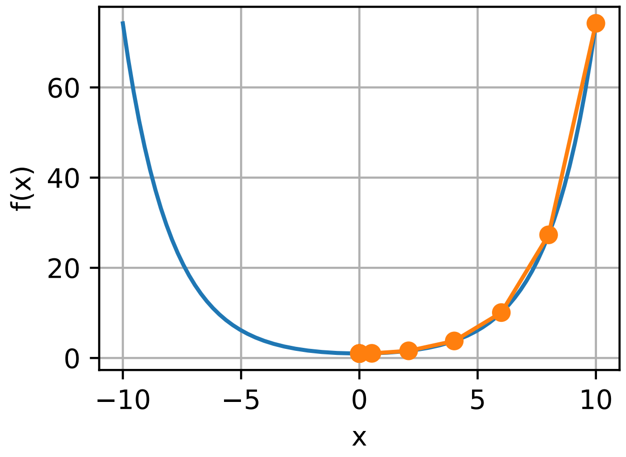
$$\nabla f(\mathbf{x}) + H_f\epsilon = 0 \text{ and hence } \epsilon = -H_f^{-1}\nabla f(\mathbf{x}). \tag{11.3.8}$$

That is, we need to invert the Hessian H_f as part of the optimization problem.

For $f(x) = \frac{1}{2}x^2$ we have $\nabla f(x) = x$ and $H_f = 1$. Hence for any x we obtain $\epsilon = -x$. In other words, a single step is sufficient to converge perfectly without the need for any adjustment! Alas, we got a bit lucky here since the Taylor expansion was exact. Let's see what happens in other problems.

```
c = 0.5

def f(x):
    return np.cosh(c * x)  # Objective

def gradf(x):
    return c * np.sinh(c * x)  # Derivative

def hessf(x):
    return c**2 * np.cosh(c * x)  # Hessian

# Hide learning rate for now
def newton(eta=1):
    x = 10
    results = [x]
    for i in range(10):
        x -= eta * gradf(x) / hessf(x)
        results.append(x)
    print('epoch 10, x:', x)
    return results

show_trace(newton())
```

```
epoch 10, x: 0.0
```



Now let's see what happens when we have a *nonconvex* function, such as $f(x) = x\cos(cx)$. After all, note that in Newton's method we end up dividing by the Hessian. This means that if the second derivative is *negative* we would walk into the direction of *increasing* f. That is a fatal flaw of the algorithm. Let's see what happens in practice.

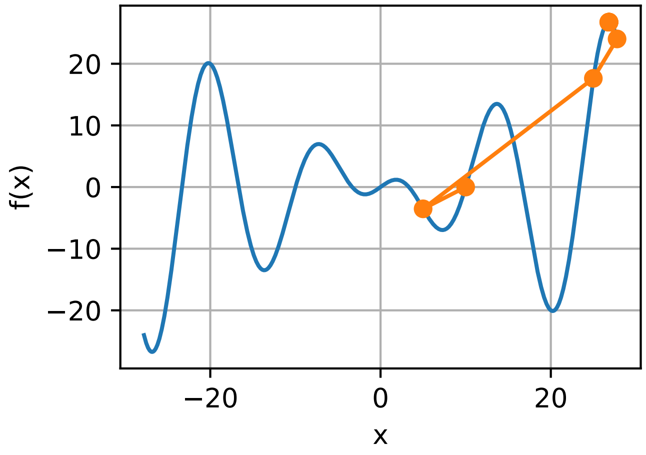
```
c = 0.15 * np.pi

def f(x):
    return x * np.cos(c * x)

def gradf(x):
    return np.cos(c * x) - c * x * np.sin(c * x)
```

(continues on next page)

(continued from previous page)

```python
def hessf(x):
    return - 2 * c * np.sin(c * x) - x * c**2 * np.cos(c * x)

show_trace(newton())
```

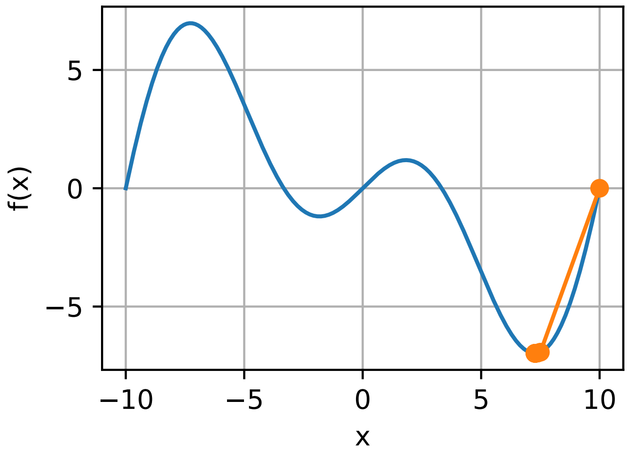
```
epoch 10, x: 26.83413291324767
```



This went spectacularly wrong. How can we fix it? One way would be to "fix" the Hessian by taking its absolute value instead. Another strategy is to bring back the learning rate. This seems to defeat the purpose, but not quite. Having second order information allows us to be cautious whenever the curvature is large and to take longer steps whenever the objective is flat. Let's see how this works with a slightly smaller learning rate, say $\eta = 0.5$. As we can see, we have quite an efficient algorithm.

```python
show_trace(newton(0.5))
```

```
epoch 10, x: 7.269860168684531
```



Convergence Analysis

We only analyze the convergence rate for convex and three times differentiable f, where at its minimum x^* the second derivative is nonzero, i.e., where $f''(x^*) > 0$. The multivariate proof is a straightforward extension of the argument below and omitted since it does not help us much in terms of intuition.

Denote by x_k the value of x at the k-th iteration and let $e_k := x_k - x^*$ be the distance from optimality. By Taylor series expansion we have that the condition $f'(x^*) = 0$ can be written as

$$0 = f'(x_k - e_k) = f'(x_k) - e_k f''(x_k) + \frac{1}{2} e_k^2 f'''(\xi_k). \tag{11.3.9}$$

This holds for some $\xi_k \in [x_k - e_k, x_k]$. Recall that we have the update $x_{k+1} = x_k - f'(x_k)/f''(x_k)$. Dividing the above expansion by $f''(x_k)$ yields

$$e_k - f'(x_k)/f''(x_k) = \frac{1}{2} e_k^2 f'''(\xi_k)/f''(x_k). \tag{11.3.10}$$

Plugging in the update equations leads to the following bound $e_{k+1} \leq e_k^2 f'''(\xi_k)/f'(x_k)$. Consequently, whenever we are in a region of bounded $f'''(\xi_k)/f''(x_k) \leq c$, we have a quadratically decreasing error $e_{k+1} \leq c e_k^2$.

As an aside, optimization researchers call this *linear* convergence, whereas a condition such as $e_{k+1} \leq \alpha e_k$ would be called a *constant* rate of convergence. Note that this analysis comes with a number of caveats: We do not really have much of a guarantee when we will reach the region of rapid convergence. Instead, we only know that once we reach it, convergence will be very quick. Second, this requires that f is well-behaved up to higher order derivatives. It comes down to ensuring that f does not have any "surprising" properties in terms of how it might change its values.

Preconditioning

Quite unsurprisingly computing and storing the full Hessian is very expensive. It is thus desirable to find alternatives. One way to improve matters is by avoiding to compute the Hessian in its entirety but only compute the *diagonal* entries. While this is not quite as good as the full Newton method, it is still much better than not using it. Moreover, estimates for the main diagonal elements are what drives some of the innovation in stochastic gradient descent optimization algorithms. This leads to update algorithms of the form

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \mathrm{diag}(H_f)^{-1} \nabla \mathbf{x}. \tag{11.3.11}$$

To see why this might be a good idea consider a situation where one variable denotes height in millimeters and the other one denotes height in kilometers. Assuming that for both the natural scale is in meters we have a terrible mismatch in parameterizations. Using preconditioning removes this. Effectively preconditioning with gradient descent amounts to selecting a different learning rate for each coordinate.

Gradient Descent with Line Search

One of the key problems in gradient descent was that we might overshoot the goal or make insufficient progress. A simple fix for the problem is to use line search in conjunction with gradient descent. That is, we use the direction given by $\nabla f(\mathbf{x})$ and then perform binary search as to which step length η minimizes $f(\mathbf{x} - \eta \nabla f(\mathbf{x}))$.

This algorithm converges rapidly (for an analysis and proof see e.g., (Boyd & Vandenberghe, 2004)). However, for the purpose of deep learning this is not quite so feasible, since each step of the line search would require us to evaluate the objective function on the entire dataset. This is way too costly to accomplish.

Summary

- Learning rates matter. Too large and we diverge, too small and we do not make progress.
- Gradient descent can get stuck in local minima.
- In high dimensions adjusting learning the learning rate is complicated.
- Preconditioning can help with scale adjustment.
- Newton's method is a lot faster *once* it has started working properly in convex problems.
- Beware of using Newton's method without any adjustments for nonconvex problems.

Exercises

1. Experiment with different learning rates and objective functions for gradient descent.
2. Implement line search to minimize a convex function in the interval $[a, b]$.
    - Do you need derivatives for binary search, i.e., to decide whether to pick $[a, (a + b)/2]$ or $[(a + b)/2, b]$.
    - How rapid is the rate of convergence for the algorithm?
    - Implement the algorithm and apply it to minimizing $\log(\exp(x) + \exp(-2 * x - 3))$.
3. Design an objective function defined on $\mathbb{R}^2$ where gradient descent is exceedingly slow. Hint - scale different coordinates differently.
4. Implement the lightweight version of Newton's method using preconditioning:
    - Use diagonal Hessian as preconditioner.
    - Use the absolute values of that rather than the actual (possibly signed) values.
    - Apply this to the problem above.
5. Apply the algorithm above to a number of objective functions (convex or not). What happens if you rotate coordinates by 45 degrees?

11.4 Stochastic Gradient Descent

In this section, we are going to introduce the basic principles of stochastic gradient descent.

```
%matplotlib inline
import d2l
import math
from mxnet import np, npx
npx.set_np()
```

11.4.1 Stochastic Gradient Updates

In deep learning, the objective function is usually the average of the loss functions for each example in the training dataset. We assume that $f_i(\mathbf{x})$ is the loss function of the training data instance with n examples, an index of i, and parameter vector of $\mathbf{x}$, then we have the objective function

$$f(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\mathbf{x}). \tag{11.4.1}$$

The gradient of the objective function at $\mathbf{x}$ is computed as

$$\nabla f(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\mathbf{x}). \tag{11.4.2}$$

If gradient descent is used, the computing cost for each independent variable iteration is $\mathcal{O}(n)$, which grows linearly with n. Therefore, when the model training data instance is large, the cost of gradient descent for each iteration will be very high.

Stochastic gradient descent (SGD) reduces computational cost at each iteration. At each iteration of stochastic gradient descent, we uniformly sample an index $i \in \{1, \ldots, n\}$ for data instances at random, and compute the gradient $\nabla f_i(\mathbf{x})$ to update $\mathbf{x}$:

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f_i(\mathbf{x}). \tag{11.4.3}$$

Here, η is the learning rate. We can see that the computing cost for each iteration drops from $\mathcal{O}(n)$ of the gradient descent to the constant $\mathcal{O}(1)$. We should mention that the stochastic gradient $\nabla f_i(\mathbf{x})$ is the unbiased estimate of gradient $\nabla f(\mathbf{x})$.

$$\mathbb{E}_i \nabla f_i(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\mathbf{x}) = \nabla f(\mathbf{x}). \tag{11.4.4}$$

This means that, on average, the stochastic gradient is a good estimate of the gradient.

Now, we will compare it to gradient descent by adding random noise with a mean of 0 to the gradient to simulate a SGD.

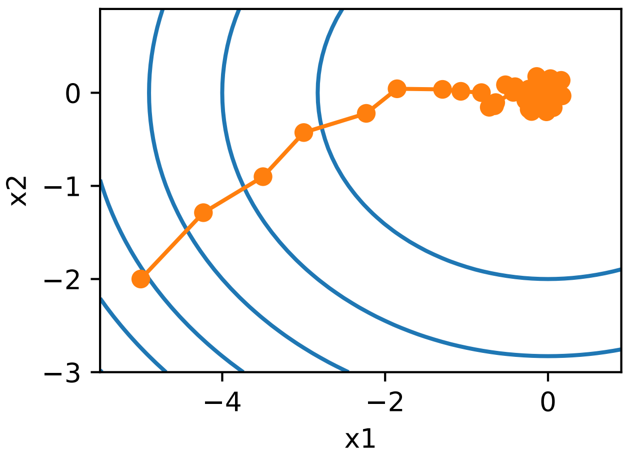
```
def f(x1, x2):
    return x1 ** 2 + 2 * x2 ** 2  # Objective

def gradf(x1, x2):
    return (2 * x1, 4 * x2)  # Gradient
```

(continues on next page)

(continued from previous page)

```
def sgd(x1, x2, s1, s2):  # Simulate noisy gradient
    global lr  # Learning rate scheduler
    (g1, g2) = gradf(x1, x2)  # Compute gradient
    (g1, g2) = (g1 + np.random.normal(0.1), g2 + np.random.normal(0.1))
    eta_t = eta * lr()  # Learning rate at time t
    return (x1 - eta_t * g1, x2 - eta_t * g2, 0, 0)  # Update variables

eta = 0.1
lr = (lambda: 1)  # Constant learning rate
d2l.show_trace_2d(f, d2l.train_2d(sgd, steps=50))
```

```
epoch 50, x1 -0.522513, x2 0.085780
```



As we can see, the trajectory of the variables in the SGD is much more noisy than the one we observed in gradient descent in the previous section. This is due to the stochastic nature of the gradient. That is, even when we arrive near the minimum, we are still subject to the uncertainty injected by the instantaneous gradient via $\eta \nabla f_i(\mathbf{x})$. Even after 50 steps the quality is still not so good. Even worse, it will not improve after additional steps (we encourage the reader to experiment with a larger number of steps to confirm this on his own). This leaves us with the only alternative—change the learning rate η. However, if we pick this too small, we will not make any meaningful progress initially. On the other hand, if we pick it too large, we will not get a good solution, as seen above. The only way to resolve these conflicting goals is to reduce the learning rate *dynamically* as optimization progresses.

This is also the reason for adding a learning rate function `lr` into the `sgd` step function. In the example above any functionality for learning rate scheduling lies dormant as we set the associated `lr` function to be constant, i.e., `lr = (lambda: 1)`.

11.4.2 Dynamic Learning Rate

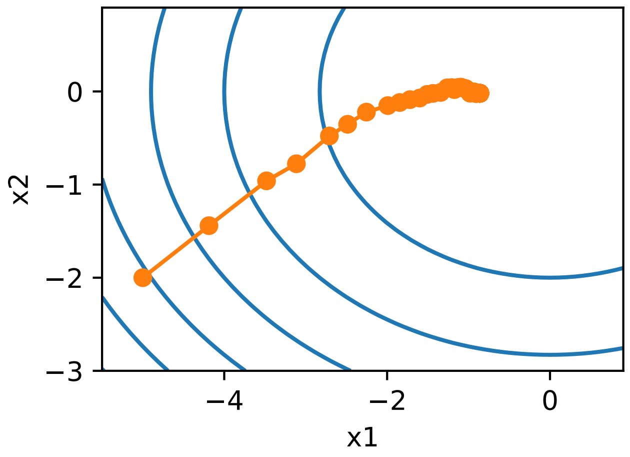
Replacing η with a time-dependent learning rate $\eta(t)$ adds to the complexity of controlling convergence of an optimization algorithm. In particular, need to figure out how rapidly η should decay. If it is too quick, we will stop optimizing prematurely. If we decrease it too slowly, we waste too much time on optimization. There are a few basic strategies that are used in adjusting η over time (we will discuss more advanced strategies in a later chapter):

$$\begin{aligned}
\eta(t) &= \eta_i \text{ if } t_i \leq t \leq t_{i+1} && \text{piecewise constant} \\
\eta(t) &= \eta_0 \cdot e^{-\lambda t} && \text{exponential} \\
\eta(t) &= \eta_0 \cdot (\beta t + 1)^{-\alpha} && \text{polynomial}
\end{aligned} \tag{11.4.5}$$

In the first scenario we decrease the learning rate, e.g., whenever progress in optimization has stalled. This is a common strategy for training deep networks. Alternatively we could decrease it much more aggressively by an exponential decay. Unfortunately this leads to premature stopping before the algorithm has converged. A popular choice is polynomial decay with $\alpha = 0.5$. In the case of convex optimization there are a number of proofs which show that this rate is well behaved. Let's see what this looks like in practice.

```
def exponential():
    global ctr
    ctr += 1
    return math.exp(-0.1 * ctr)

ctr = 1
lr = exponential  # Set up learning rate
d2l.show_trace_2d(f, d2l.train_2d(sgd, steps=1000))
```

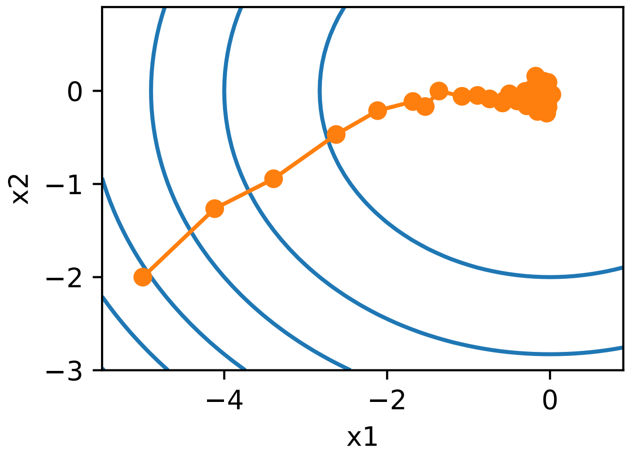
```
epoch 1000, x1 -0.862200, x2 -0.019736
```



As expected, the variance in the parameters is significantly reduced. However, this comes at the expense of failing to converge to the optimal solution $\mathbf{x} = (0, 0)$. Even after 1000 steps are we are still very far away from the optimal solution. Indeed, the algorithm fails to converge at all. On the other hand, if we use a polynomial decay where the learning rate decays with the inverse square root of the number of steps convergence is good.

```
def polynomial():
    global ctr
    ctr += 1
    return (1 + 0.1 * ctr)**(-0.5)

ctr = 1
lr = polynomial  # Set up learning rate
d2l.show_trace_2d(f, d2l.train_2d(sgd, steps=50))
```

```
epoch 50, x1 -0.024847, x2 0.090820
```



There exist many more choices for how to set the learning rate. For instance, we could start with a small rate, then rapidly ramp up and then decrease it again, albeit more slowly. We could even alternate between smaller and larger learning rates. There exists a large variety of such schedules. For now let's focus on learning rate schedules for which a comprehensive theoretical analysis is possible, i.e., on learning rates in a convex setting. For general nonconvex problems it is very difficult to obtain meaningful convergence guarantees, since in general minimizing nonlinear nonconvex problems is NP hard. For a survey see e.g., the excellent lecture notes[152] of Tibshirani 2015.

11.4.3 Convergence Analysis for Convex Objectives

The following is optional and primarily serves to convey more intuition about the problem. We limit ourselves to one of the simplest proofs, as described by (Nesterov & Vial, 2000). Significantly more advanced proof techniques exist, e.g., whenever the objective function is particularly well behaved. (Hazan et al., 2008) show that for strongly convex functions, i.e., for functions that can be bounded from below by $\mathbf{x}^\top \mathbf{Q}\mathbf{x}$, it is possible to minimize them in a small number of steps while decreasing the learning rate like $\eta(t) = \eta_0/(\beta t + 1)$. Unfortunately this case never really occurs in deep learning and we are left with a much more slowly decreasing rate in practice.

Consider the case where

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \partial_{\mathbf{w}} l(\mathbf{x}_t, \mathbf{w}). \tag{11.4.6}$$

[152] https://www.stat.cmu.edu/~ryantibs/convexopt-F15/lectures/26-nonconvex.pdf

In particular, assume that $\mathbf{x}_t$ is drawn from some distribution $P(\mathbf{x})$ and that $l(\mathbf{x}, \mathbf{w})$ is a convex function in $\mathbf{w}$ for all $\mathbf{x}$. Last denote by

$$R(\mathbf{w}) = E_{\mathbf{x}\sim P}[l(\mathbf{x}, \mathbf{w})] \tag{11.4.7}$$

the expected risk and by R^* its minimum with regard to $\mathbf{w}$. Last let $\mathbf{w}^*$ be the minimizer (we assume that it exists within the domain which $\mathbf{w}$ is defined). In this case we can track the distance between the current parameter $\mathbf{w}_t$ and the risk minimizer $\mathbf{w}^*$ and see whether it improves over time:

$$\begin{aligned}\|\mathbf{w}_{t+1} - \mathbf{w}^*\|^2 &= \|\mathbf{w}_t - \eta_t \partial_{\mathbf{w}} l(\mathbf{x}_t, \mathbf{w}) - \mathbf{w}^*\|^2 \\ &= \|\mathbf{w}_t - \mathbf{w}^*\|^2 + \eta_t^2 \|\partial_{\mathbf{w}} l(\mathbf{x}_t, \mathbf{w})\|^2 - 2\eta_t \left\langle \mathbf{w}_t - \mathbf{w}^*, \partial_{\mathbf{w}} l(\mathbf{x}_t, \mathbf{w})\right\rangle.\end{aligned} \tag{11.4.8}$$

The gradient $\partial_{\mathbf{w}} l(\mathbf{x}_t, \mathbf{w})$ can be bounded from above by some Lipschitz constant L, hence we have that

$$\eta_t^2 \|\partial_{\mathbf{w}} l(\mathbf{x}_t, \mathbf{w})\|^2 \leq \eta_t^2 L^2. \tag{11.4.9}$$

We are mostly interested in how the distance between $\mathbf{w}_t$ and $\mathbf{w}^*$ changes *in expectation*. In fact, for any specific sequence of steps the distance might well increase, depending on whichever $\mathbf{x}_t$ we encounter. Hence we need to bound the inner product. By convexity we have that

$$l(\mathbf{x}_t, \mathbf{w}^*) \geq l(\mathbf{x}_t, \mathbf{w}_t) + \left\langle \mathbf{w}^* - \mathbf{w}_t, \partial_{\mathbf{w}} l(\mathbf{x}_t, \mathbf{w}_t)\right\rangle. \tag{11.4.10}$$

Using both inequalities and plugging it into the above we obtain a bound on the distance between parameters at time $t+1$ as follows:

$$\|\mathbf{w}_t - \mathbf{w}^*\|^2 - \|\mathbf{w}_{t+1} - \mathbf{w}^*\|^2 \geq 2\eta_t (l(\mathbf{x}_t, \mathbf{w}_t) - l(\mathbf{x}_t, \mathbf{w}^*)) - \eta_t^2 L^2. \tag{11.4.11}$$

This means that we make progress as long as the expected difference between current loss and the optimal loss outweighs $\eta_t L^2$. Since the former is bound to converge to 0 it follows that the learning rate η_t also needs to vanish.

Next we take expectations over this expression. This yields

$$E_{\mathbf{w}_t}\left[\|\mathbf{w}_t - \mathbf{w}^*\|^2\right] - E_{\mathbf{w}_{t+1}|\mathbf{w}_t}\left[\|\mathbf{w}_{t+1} - \mathbf{w}^*\|^2\right] \geq 2\eta_t[E[R[\mathbf{w}_t]] - R^*] - \eta_t^2 L^2. \tag{11.4.12}$$

The last step involves summing over the inequalities for $t \in \{t, \ldots, T\}$. Since the sum telescopes and by dropping the lower term we obtain

$$\|\mathbf{w}_0 - \mathbf{w}^*\|^2 \geq 2\sum_{t=1}^{T} \eta_t [E[R[\mathbf{w}_t]] - R^*] - L^2 \sum_{t=1}^{T} \eta_t^2. \tag{11.4.13}$$

Note that we exploited that $\mathbf{w}_0$ is given and thus the expectation can be dropped. Last define

$$\bar{\mathbf{w}} := \frac{\sum_{t=1}^{T} \eta_t \mathbf{w}_t}{\sum_{t=1}^{T} \eta_t}. \tag{11.4.14}$$

Then by convexity it follows that

$$\sum_t \eta_t E[R[\mathbf{w}_t]] \geq \sum \eta_t \cdot [E[\bar{\mathbf{w}}]]. \tag{11.4.15}$$

Plugging this into the above inequality yields the bound

$$[E[\bar{\mathbf{w}}]] - R^* \leq \frac{r^2 + L^2 \sum_{t=1}^{T} \eta_t^2}{2\sum_{t=1}^{T} \eta_t}. \tag{11.4.16}$$

Here $r^2 := \|\mathbf{w}_0 - \mathbf{w}^*\|^2$ is a bound on the distance between the initial choice of parameters and the final outcome. In short, the speed of convergence depends on how rapidly the loss function changes via the Lipschitz constant L and how far away from optimality the initial value is r. Note that the bound is in terms of $\bar{\mathbf{w}}$ rather than $\mathbf{w}_T$. This is the case since $\bar{\mathbf{w}}$ is a smoothed version of the optimization path. Now let's analyze some choices for η_t.

- **Known Time Horizon**. Whenever r, L and T are known we can pick $\eta = r/L\sqrt{T}$. This yields as upper bound $rL(1 + 1/T)/2\sqrt{T} < rL/\sqrt{T}$. That is, we converge with rate $\mathcal{O}(1/\sqrt{T})$ to the optimal solution.
- **Unknown Time Horizon**. Whenever we want to have a good solution for *any* time T we can pick $\eta = \mathcal{O}(1/\sqrt{T})$. This costs us an extra logarithmic factor and it leads to an upper bound of the form $\mathcal{O}(\log T/\sqrt{T})$.

Note that for strongly convex losses $l(\mathbf{x}, \mathbf{w}') \geq l(\mathbf{x}, \mathbf{w}) + \langle \mathbf{w}' - \mathbf{w}, \partial_{\mathbf{w}} l(\mathbf{x}, \mathbf{w}) \rangle + \frac{\lambda}{2}\|\mathbf{w} - \mathbf{w}'\|^2$ we can design even more rapidly converging optimization schedules. In fact, an exponential decay in η leads to a bound of the form $\mathcal{O}(\log T/T)$.

11.4.4 Stochastic Gradients and Finite Samples

So far we have played a bit fast and loose when it comes to talking about stochastic gradient descent. We posited that we draw instances x_i, typically with labels y_i from some distribution $p(x, y)$ and that we use this to update the weights w in some manner. In particular, for a finite sample size we simply argued that the discrete distribution $p(x, y) = \frac{1}{n}\sum_{i=1}^{n} \delta_{x_i}(x)\delta_{y_i}(y)$ allows us to perform SGD over it.

However, this is not really what we did. In the toy examples in the current section we simply added noise to an otherwise non-stochastic gradient, i.e., we pretended to have pairs (x_i, y_i). It turns out that this is justified here (see the exercises for a detailed discussion). More troubling is that in all previous discussions we clearly did not do this. Instead we iterated over all instances exactly once. To see why this is preferable consider the converse, namely that we are sampling n observations from the discrete distribution with replacement. The probability of choosing an element i at random is N^{-1}. Thus to choose it at least once is

$$P(\text{choose } i) = 1 - P(\text{omit } i) = 1 - (1 - N^{-1})^N \approx 1 - e^{-1} \approx 0.63. \quad (11.4.17)$$

A similar reasoning shows that the probability of picking a sample exactly once is given by $\binom{N}{1} N^{-1}(1 - N^{-1})^{N-1} = \frac{N-1}{N}(1 - N^{-1})^N \approx e^{-1} \approx 0.37$. This leads to an increased variance and decreased data efficiency relative to sampling without replacement. Hence, in practice we perform the latter (and this is the default choice throughout this book). Last note that repeated passes through the dataset traverse it in a *different* random order.

Summary

- For convex problems we can prove that for a wide choice of learning rates Stochastic Gradient Descent will converge to the optimal solution.
- For deep learning this is generally not the case. However, the analysis of convex problems gives us useful insight into how to approach optimization, namely to reduce the learning rate progressively, albeit not too quickly.
- Problems occur when the learning rate is too small or too large. In practice a suitable learning rate is often found only after multiple experiments.

- When there are more examples in the training dataset, it costs more to compute each iteration for gradient descent, so SGD is preferred in these cases.
- Optimality guarantees for SGD are in general not available in nonconvex cases since the number of local minima that require checking might well be exponential.

Exercises

1. Experiment with different learning rate schedules for SGD and with different numbers of iterations. In particular, plot the distance from the optimal solution $(0, 0)$ as a function of the number of iterations.
2. Prove that for the function $f(x_1, x_2) = x_1^2 + 2x_2^2$ adding normal noise to the gradient is equivalent to minimizing a loss function $l(\mathbf{x}, \mathbf{w}) = (x_1 - w_1)^2 + 2(x_2 - w_2)^2$ where x is drawn from a normal distribution.
    - Derive mean and variance of the distribution for $\mathbf{x}$.
    - Show that this property holds in general for objective functions $f(\mathbf{x}) = \frac{1}{2}(\mathbf{x} - \mu)^\top Q(\mathbf{x} - \mu)$ for $Q \succeq 0$.
3. Compare convergence of SGD when you sample from $\{(x_1, y_1), \ldots, (x_m, y_m)\}$ with replacement and when you sample without replacement.
4. How would you change the SGD solver if some gradient (or rather some coordinate associated with it) was consistently larger than all other gradients?
5. Assume that $f(x) = x^2(1 + \sin x)$. How many local minima does f have? Can you change f in such a way that to minimize it one needs to evaluate all local minima?

11.5 Minibatch Stochastic Gradient Descent

So far we encountered two extremes in the approach to gradient based learning: Section 11.3 uses the full dataset to compute gradients and to update parameters, one pass at a time. Conversely Section 11.4 processes one observation at a time to make progress. Each of them has its own drawbacks. Gradient Descent is not particularly *data efficient* whenever data is very similar. Stochastic Gradient Descent is not particularly *computationally efficient* since CPUs and GPUs cannot exploit the full power of vectorization. This suggests that there might be a happy medium, and in fact, that's what we've been using so far in the examples we discussed.

11.5.1 Vectorization and Caches

At the heart of the decision to use minibatches is computational efficiency. This is most easily understood when considering parallelization to multiple GPUs and multiple servers. In this case we need to send at least one image to each GPU. With 8 GPUs per server and 16 servers we already arrive at a minibatch size of 128.

Things are a bit more subtle when it comes to single GPUs or even CPUs. These devices have multiple types of memory, often multiple type of compute units and different bandwidth constraints between them. For instance, a CPU has a small number of registers and then L1, L2 and in some cases even L3 cache (which is shared between the different processor cores). These caches are of increasing size and latency (and at the same time they're of decreasing bandwidth). Suffice it to say, the processor is capable of performing many more operations than what the main memory interface is able to provide.

- A 2GHz CPU with 16 cores and AVX-512 vectorization can process up to $2 \cdot 10^9 \cdot 16 \cdot 32 = 10^{12}$ bytes per second. The capability of GPUs easily exceeds this number by a factor of 100. On the other hand, a midrange server processor might not have much more than 100 GB/s bandwidth, i.e., less than one tenth of what would be required to keep the processor fed. To make matters worse, not all memory access is created equal: first, memory interfaces are typically 64 bit wide or wider (e.g., on GPUs up to 384 bit), hence reading a single byte incurs the cost of a much wider access.
- There is significant overhead for the first access whereas sequential access is relatively cheap (this is often called a burst read). There are many more things to keep in mind, such as caching when we have multiple sockets, chiplets and other structures. A detailed discussion of this is beyond the scope of this section. See e.g., this Wikipedia article[154] for a more in-depth discussion.

The way to alleviate these constraints is to use a hierarchy of CPU caches which are actually fast enough to supply the processor with data. This is *the* driving force behind batching in deep learning. To keep matters simple, consider matrix-matrix multiplication, say $\mathbf{A} = \mathbf{BC}$. We have a number of options for calculating $\mathbf{A}$. For instance we could try the following:

1. We could compute $\mathbf{A}_{ij} = \mathbf{B}_{i,:}\mathbf{C}_{:,j}^\top$, i.e., we could compute it element-wise by means of dot products.
2. We could compute $\mathbf{A}_{:,j} = \mathbf{B}\mathbf{C}_{:,j}^\top$, i.e., we could compute it one column at a time. Likewise we could compute $\mathbf{A}$ one row $\mathbf{A}_{i,:}$ at a time.
3. We could simply compute $\mathbf{A} = \mathbf{BC}$.
4. We could break $\mathbf{B}$ and $\mathbf{C}$ into smaller block matrices and compute $\mathbf{A}$ one block at a time.

If we follow the first option, we will need to copy one row and one column vector into the CPU each time we want to compute an element $\mathbf{A}_{ij}$. Even worse, due to the fact that matrix elements are aligned sequentially we are thus required to access many disjoint locations for one of the two vectors as we read them from memory. The second option is much more favorable. In it, we are able to keep the column vector $\mathbf{C}_{:,j}$ in the CPU cache while we keep on traversing through B. This halves the memory bandwidth requirement with correspondingly faster access. Of course, option 3 is most desirable. Unfortunately, most matrices might not entirely fit into cache (this is what we're discussing after all). However, option 4 offers a practically useful alternative: we can move blocks of the matrix into cache and multiply them locally. Optimized libraries take care of this for us. Let's have a look at how efficient these operations are in practice.

[154] https://en.wikipedia.org/wiki/Cache_hierarchy

Beyond computational efficiency, the overhead introduced by Python and by the deep learning framework itself is considerable. Recall that each time we execute a command the Python interpreter sends a command to the MXNet engine which needs to insert it into the compute graph and deal with it during scheduling. Such overhead can be quite detrimental. In short, it is highly advisable to use vectorization (and matrices) whenever possible.

```
%matplotlib inline
import d2l
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
npx.set_np()

timer = d2l.Timer()
A = np.zeros((1024, 1024))
B = np.random.normal(0, 1, (1024, 1024))
C = np.random.normal(0, 1, (1024, 1024))
```

Element-wise assignment simply iterates over all rows and columns of **B** and **C** respectively to assign the value to **A**.

```
# Compute A = B C one element at a time
timer.start()
for i in range(1024):
    for j in range(1024):
        A[i, j] = np.dot(B[i, :], C[:, j])
A.wait_to_read()
timer.stop()
```

```
868.5792026519775
```

A faster strategy is to perform column-wise assignment.

```
# Compute A = B C one column at a time
timer.start()
for j in range(1024):
    A[:, j] = np.dot(B, C[:, j])
A.wait_to_read()
timer.stop()
```

```
0.7444274425506592
```

Last, the most effective manner is to perform the entire operation in one block. Let's see what the respective speed of the operations is.

```
# Compute A = B C in one go
timer.start()
A = np.dot(B, C)
A.wait_to_read()
timer.stop()

# Multiply and add count as separate operations (fused in practice)
gigaflops = [2/i for i in timer.times]
```

(continues on next page)

(continued from previous page)

```
print("Performance in Gigaflops: element {:.3f}, \
      column {:.3f}, full {:.3f}".format(*gigaflops))
```

```
Performance in Gigaflops: element 0.002,       column 2.687, full 600.301
```

11.5.2 Minibatches

In the past we took it for granted that we would read *minibatches* of data rather than single observations to update parameters. We now give a brief justification for it. Processing single observations requires us to perform many single matrix-vector (or even vector-vector) multiplications, which is quite expensive and which incurs a significant overhead on behalf of the underlying deep learning framework. This applies both to evaluating a network when applied to data (often referred to as inference) and when computing gradients to update parameters. That is, this applies whenever we perform $\mathbf{w} \leftarrow \mathbf{w} - \eta_t \mathbf{g}_t$ where

$$\mathbf{g}_t = \partial_{\mathbf{w}} f(\mathbf{x}_t, \mathbf{w}) \tag{11.5.1}$$

We can increase the *computational* efficiency of this operation by applying it to a minibatch of observations at a time. That is, we replace the gradient $\mathbf{g}_t$ over a single observation by one over a small batch

$$\mathbf{g}_t = \partial_{\mathbf{w}} \frac{1}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} f(\mathbf{x}_i, \mathbf{w}) \tag{11.5.2}$$

Let's see what this does to the statistical properties of $\mathbf{g}_t$: since both $\mathbf{x}_t$ and also all elements of the minibatch $\mathcal{B}_t$ are drawn uniformly at random from the training set, the expectation of the gradient remains unchanged. The variance, on the other hand, is reduced significantly. Since the minibatch gradient is composed of $b := |\mathcal{B}_t|$ independent gradients which are being averaged, its standard deviation is reduced by a factor of $b^{-\frac{1}{2}}$. This, by itself, is a good thing, since it means that the updates are more reliably aligned with the full gradient.

Naively this would indicate that choosing a large minibatch $\mathcal{B}_t$ would be universally desirable. Alas, after some point, the additional reduction in standard deviation is minimal when compared to the linear increase in computational cost. In practice we pick a minibatch that is large enough to offer good computational efficiency while still fitting into the memory of a GPU. To illustrate the savings let's have a look at some code. In it we perform the same matrix-matrix multiplication, but this time broken up into "minibatches" of 64 columns at a time.

```
timer.start()
for j in range(0, 1024, 64):
    A[:, j:j+64] = np.dot(B, C[:, j:j+64])
timer.stop()
print("Performance in Gigaflops: block {:.3f}".format(2/timer.times[3]))
```

```
Performance in Gigaflops: block 142.625
```

As we can see, the computation on the minibatch is essentially as efficient as on the full matrix. A word of caution is in order. In Section 7.5 we used a type of regularization that was heavily dependent on the amount of variance in a minibatch. As we increase the latter, the variance decreases and with it the benefit of the noise-injection due to batch normalization. See e.g., (Ioffe, 2017) for details on how to rescale and compute the appropriate terms.

11.5.3 Reading the Dataset

Let's have a look at how minibatches are efficiently generated from data. In the following we use a dataset developed by NASA to test the wing noise from different aircraft[155] to compare these optimization algorithms. For convenience we only use the first $1,500$ examples. The data is whitened for preprocessing, i.e., we remove the mean and rescale the variance to 1 per coordinate.

```
# Saved in the d2l package for later use
d2l.DATA_HUB['airfoil'] = (d2l.DATA_URL+'airfoil_self_noise.dat',
                           '76e5be1548fd8222e5074cf0faae75edff8cf93f')
def get_data_ch11(batch_size=10, n=1500):
    data = np.genfromtxt(d2l.download('airfoil'),
                         dtype=np.float32, delimiter='\t')
    data = (data - data.mean(axis=0)) / data.std(axis=0)
    data_iter = d2l.load_array(
        (data[:n, :-1], data[:n, -1]), batch_size, is_train=True)
    return data_iter, data.shape[1]-1
```

11.5.4 Implementation from Scratch

Recall the minibatch SGD implementation from Section 3.2. In the following we provide a slightly more general implementation. For convenience it has the same call signature as the other optimization algorithms introduced later in this chapter. Specifically, we add the status input `states` and place the hyperparameter in dictionary `hyperparams`. In addition, we will average the loss of each minibatch example in the training function, so the gradient in the optimization algorithm does not need to be divided by the batch size.

```
def sgd(params, states, hyperparams):
    for p in params:
        p[:] -= hyperparams['lr'] * p.grad
```

Next, we implement a generic training function to facilitate the use of the other optimization algorithms introduced later in this chapter. It initializes a linear regression model and can be used to train the model with minibatch SGD and other algorithms introduced subsequently.

```
# Saved in the d2l package for later use
def train_ch11(trainer_fn, states, hyperparams, data_iter,
               feature_dim, num_epochs=2):
    # Initialization
    w = np.random.normal(scale=0.01, size=(feature_dim, 1))
    b = np.zeros(1)
    w.attach_grad()
    b.attach_grad()
    net, loss = lambda X: d2l.linreg(X, w, b), d2l.squared_loss
    # Train
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[0, num_epochs], ylim=[0.22, 0.35])
    n, timer = 0, d2l.Timer()
    for _ in range(num_epochs):
        for X, y in data_iter:
```

(continues on next page)

[155] https://archive.ics.uci.edu/ml/datasets/Airfoil+Self-Noise

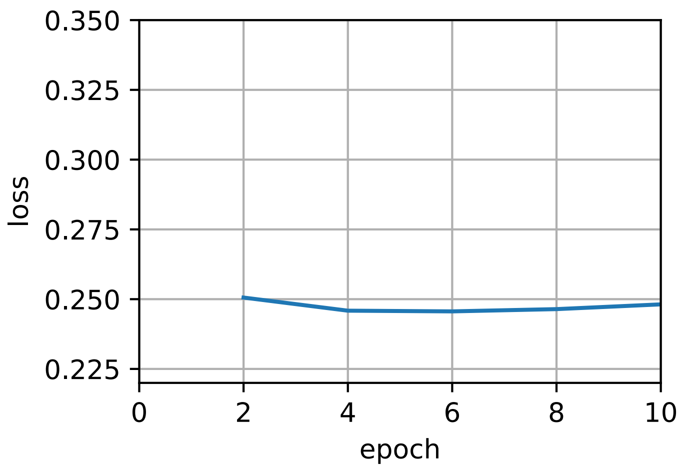
(continued from previous page)

```
            with autograd.record():
                l = loss(net(X), y).mean()
            l.backward()
            trainer_fn([w, b], states, hyperparams)
            n += X.shape[0]
            if n % 200 == 0:
                timer.stop()
                animator.add(n/X.shape[0]/len(data_iter),
                             (d2l.evaluate_loss(net, data_iter, loss),))
                timer.start()
    print('loss: %.3f, %.3f sec/epoch' % (animator.Y[0][-1], timer.avg()))
    return timer.cumsum(), animator.Y[0]
```

Let's see how optimization proceeds for batch gradient descent. This can be achieved by setting the minibatch size to 1500 (i.e., to the total number of examples). As a result the model parameters are updated only once per epoch. There is little progress. In fact, after 6 steps progress stalls.

```
def train_sgd(lr, batch_size, num_epochs=2):
    data_iter, feature_dim = get_data_ch11(batch_size)
    return train_ch11(
        sgd, None, {'lr': lr}, data_iter, feature_dim, num_epochs)

gd_res = train_sgd(1, 1500, 10)
```

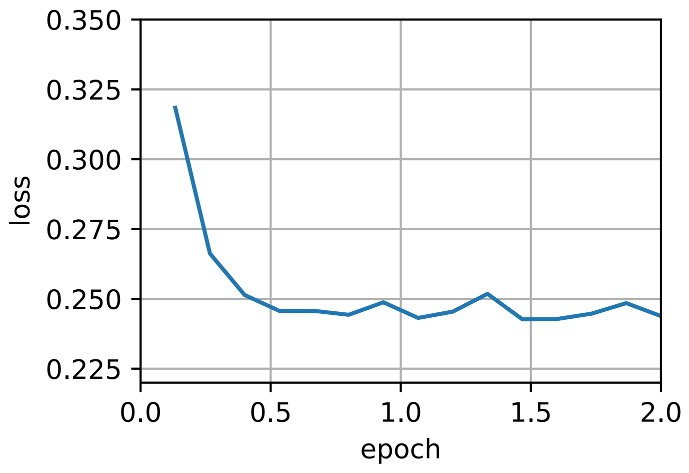
```
loss: 0.248, 0.070 sec/epoch
```



When the batch size equals 1, we use SGD for optimization. For simplicity of implementation we picked a constant (albeit small) learning rate. In SGD, the model parameters are updated whenever an example is processed. In our case this amounts to 1500 updates per epoch. As we can see, the decline in the value of the objective function slows down after one epoch. Although both the procedures processed 1500 examples within one epoch, SGD consumes more time than gradient descent in our experiment. This is because SGD updated the parameters more frequently and since it is less efficient to process single observations one at a time.

```
sgd_res = train_sgd(0.005, 1)
```

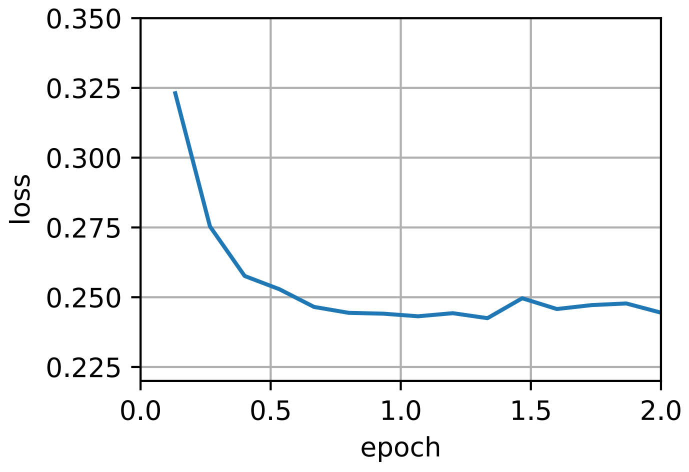
```
loss: 0.244, 0.379 sec/epoch
```



Last, when the batch size equals 100, we use minibatch SGD for optimization. The time required per epoch is longer than the time needed for SGD and the time for batch gradient descent.

```
mini1_res = train_sgd(.4, 100)
```

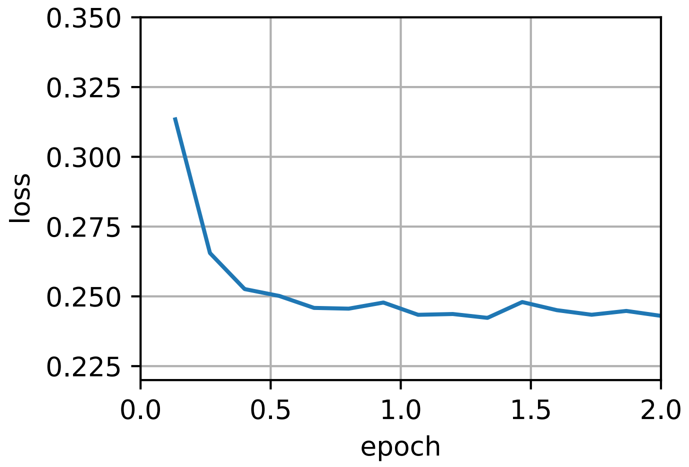
```
loss: 0.244, 0.008 sec/epoch
```



Reducing the batch size to 10, the time for each epoch increases because the workload for each batch is less efficient to execute.

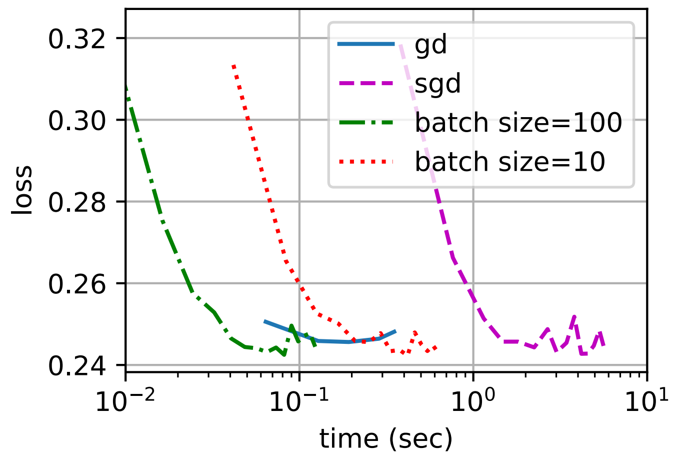
```
mini2_res = train_sgd(.05, 10)
```

```
loss: 0.243, 0.042 sec/epoch
```



Finally, we compare the time versus loss for the preview four experiments. As can be seen, despite SGD converges faster than GD in terms of number of examples processed, it uses more time to reach the same loss than GD because that computing gradient example by example is not efficient. Minibatch SGD is able to trade-off the convergence speed and computation efficiency. A minibatch size 10 is more efficient than SGD; a minibatch size 100 even outperforms GD in terms of runtime.

```
d2l.set_figsize([6, 3])
d2l.plot(*list(map(list, zip(gd_res, sgd_res, mini1_res, mini2_res))),
         'time (sec)', 'loss', xlim=[1e-2, 10],
         legend=['gd', 'sgd', 'batch size=100', 'batch size=10'])
d2l.plt.gca().set_xscale('log')
```



11.5.5 Concise Implementation

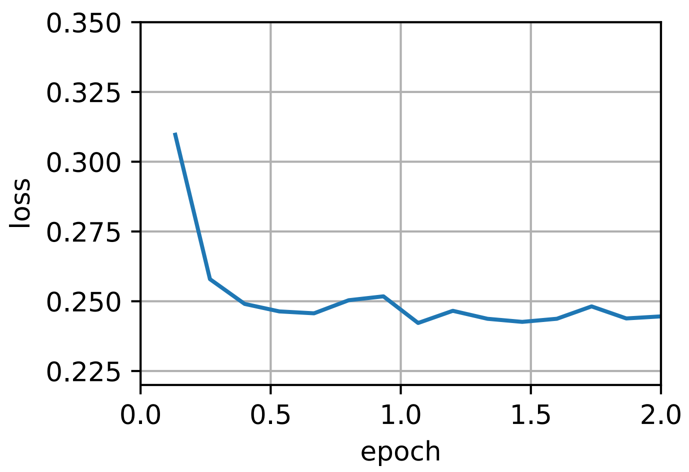
In Gluon, we can use the `Trainer` class to call optimization algorithms. This is used to implement a generic training function. We will use this throughout the current chapter.

```
# Saved in the d2l package for later use
def train_gluon_ch11(tr_name, hyperparams, data_iter, num_epochs=2):
    # Initialization
    net = nn.Sequential()
    net.add(nn.Dense(1))
    net.initialize(init.Normal(sigma=0.01))
    trainer = gluon.Trainer(net.collect_params(), tr_name, hyperparams)
    loss = gluon.loss.L2Loss()
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[0, num_epochs], ylim=[0.22, 0.35])
    n, timer = 0, d2l.Timer()
    for _ in range(num_epochs):
        for X, y in data_iter:
            with autograd.record():
                l = loss(net(X), y)
            l.backward()
            trainer.step(X.shape[0])
            n += X.shape[0]
            if n % 200 == 0:
                timer.stop()
                animator.add(n/X.shape[0]/len(data_iter),
                             (d2l.evaluate_loss(net, data_iter, loss),))
                timer.start()
    print('loss: %.3f, %.3f sec/epoch' % (animator.Y[0][-1], timer.avg()))
```

Using Gluon to repeat the last experiment shows identical behavior.

```
data_iter, _ = get_data_ch11(10)
train_gluon_ch11('sgd', {'learning_rate': 0.05}, data_iter)
```

```
loss: 0.245, 0.038 sec/epoch
```



Summary

- Vectorization makes code more efficient due to reduced overhead arising from the deep learning framework and due to better memory locality and caching on CPUs and GPUs.
- There is a trade-off between statistical efficiency arising from SGD and computational efficiency arising from processing large batches of data at a time.
- Minibatch stochastic gradient descent offers the best of both worlds: computational and statistical efficiency.
- In minibatch SGD we process batches of data obtained by a random permutation of the training data (i.e., each observation is processed only once per epoch, albeit in random order).
- It is advisable to decay the learning rates during training.
- In general, minibatch SGD is faster than SGD and gradient descent for convergence to a smaller risk, when measured in terms of clock time.

Exercises

1. Modify the batch size and learning rate and observe the rate of decline for the value of the objective function and the time consumed in each epoch.
2. Read the MXNet documentation and use the `Trainer` class `set_learning_rate` function to reduce the learning rate of the minibatch SGD to 1/10 of its previous value after each epoch.
3. Compare minibatch SGD with a variant that actually *samples with replacement* from the training set. What happens?
4. An evil genie replicates your dataset without telling you (i.e., each observation occurs twice and your dataset grows to twice its original size, but nobody told you). How does the behavior of SGD, minibatch SGD and that of gradient descent change?

11.6 Momentum

In Section 11.4 we reviewed what happens when performing stochastic gradient descent, i.e., when performing optimization where only a noisy variant of the gradient is available. In particular, we noticed that for noisy gradients we need to be extra cautious when it comes to choosing the learning rate in the face of noise. If we decrease it too rapidly, convergence stalls. If we are too lenient, we fail to converge to a good enough solution since noise keeps on driving us away from optimality.

11.6.1 Basics

In this section, we will explore more effective optimization algorithms, especially for certain types of optimization problems that are common in practice.

Leaky Averages

The previous section saw us discussing minibatch SGD as a means for accelerating computation. It also had the nice side-effect that averaging gradients reduced the amount of variance.

$$\mathbf{g}_t = \partial_{\mathbf{w}} \frac{1}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} f(\mathbf{x}_i, \mathbf{w}_{t-1}) = \frac{1}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} \mathbf{g}_{i,t-1}. \tag{11.6.1}$$

Here we used $\mathbf{g}_{ii} = \partial_{\mathbf{w}} f(\mathbf{x}_i, \mathbf{w}_t)$ to keep the notation simple. It would be nice if we could benefit from the effect of variance reduction even beyond averaging gradients on a mini-batch. One option to accomplish this task is to replace the gradient computation by a "leaky average":

$$\mathbf{v}_t = \beta \mathbf{v}_{t-1} + \mathbf{g}_{t,t-1} \tag{11.6.2}$$

for some $\beta \in (0, 1)$. This effectively replaces the instantaneous gradient by one that's been averaged over multiple *past* gradients. **v** is called *momentum*. It accumulates past gradients similar to how a heavy ball rolling down the objective function landscape integrates over past forces. To see what is happening in more detail let's expand $\mathbf{v}_t$ recursively into

$$\mathbf{v}_t = \beta^2 \mathbf{v}_{t-2} + \beta \mathbf{g}_{t-1,t-2} + \mathbf{g}_{t,t-1} = \ldots = \sum_{\tau=0}^{t-1} \beta^{\tau} \mathbf{g}_{t-\tau,t-\tau-1}. \tag{11.6.3}$$

Large β amounts to a long-range average, whereas small β amounts to only a slight correction relative to a gradient method. The new gradient replacement no longer points into the direction of steepest descent on a particular instance any longer but rather in the direction of a weighted average of past gradients. This allows us to realize most of the benefits of averaging over a batch without the cost of actually computing the gradients on it. We will revisit this averaging procedure in more detail later.

The above reasoning formed the basis for what is now known as *accelerated* gradient methods, such as gradients with momentum. They enjoy the additional benefit of being much more effective in cases where the optimization problem is ill-conditioned (i.e., where there are some directions where progress is much slower than in others, resembling a narrow canyon). Furthermore, they allow us to average over subsequent gradients to obtain more stable directions of descent. Indeed, the aspect of acceleration even for noise-free convex problems is one of the key reasons why momentum works and why it works so well.

As one would expect, due to its efficacy momentum is a well-studied subject in optimization for deep learning and beyond. See e.g., the beautiful expository article[157] by (Goh, 2017) for an in-depth analysis and interactive animation. It was proposed by (Polyak, 1964). (Nesterov, 2018) has a detailed theoretical discussion in the context of convex optimization. Momentum in deep learning has been known to be beneficial for a long time. See e.g., the discussion by (Sutskever et al., 2013) for details.

[157] https://distill.pub/2017/momentum/

An Ill-conditioned Problem

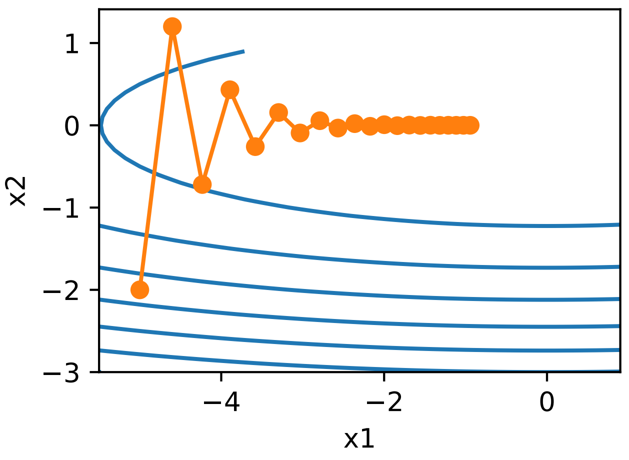
To get a better understanding of the geometric properties of the momentum method we revisit gradient descent, albeit with a significantly less pleasant objective function. Recall that in Section 11.3 we used $f(\mathbf{x}) = x_1^2 + 2x_2^2$, i.e., a moderately distorted ellipsoid objective. We distort this function further by stretching it out in the x_1 direction via

$$f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2. \tag{11.6.4}$$

As before f has its minimum at $(0, 0)$. This function is *very* flat in the direction of x_1. Let's see what happens when we perform gradient descent as before on this new function. We pick a learning rate of 0.4.

```
%matplotlib inline
import d2l
from mxnet import np, npx
npx.set_np()

eta = 0.4
def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2
def gd_2d(x1, x2, s1, s2):
    return (x1 - eta * 0.2 * x1, x2 - eta * 4 * x2, 0, 0)

d2l.show_trace_2d(f_2d, d2l.train_2d(gd_2d))
```

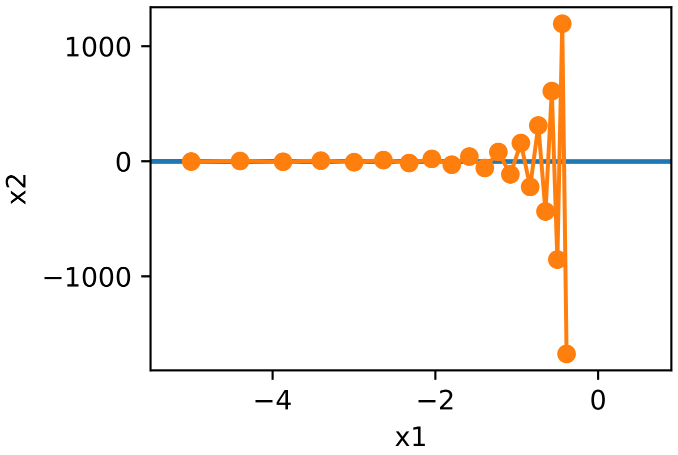
```
epoch 20, x1 -0.943467, x2 -0.000073
```



By construction, the gradient in the x_2 direction is *much* higher and changes much more rapidly than in the horizontal x_1 direction. Thus we are stuck between two undesirable choices: if we pick a small learning rate we ensure that the solution does not diverge in the x_2 direction but we're saddled with slow convergence in the x_1 direction. Conversely, with a large learning rate we progress rapidly in the x_1 direction but diverge in x_2. The example below illustrates what happens even after a slight increase in learning rate from 0.4 to 0.6. Convergence in the x_1 direction improves but the overall solution quality is much worse.

```
eta = 0.6
d2l.show_trace_2d(f_2d, d2l.train_2d(gd_2d))
```

```
epoch 20, x1 -0.387814, x2 -1673.365109
```



The Momentum Method

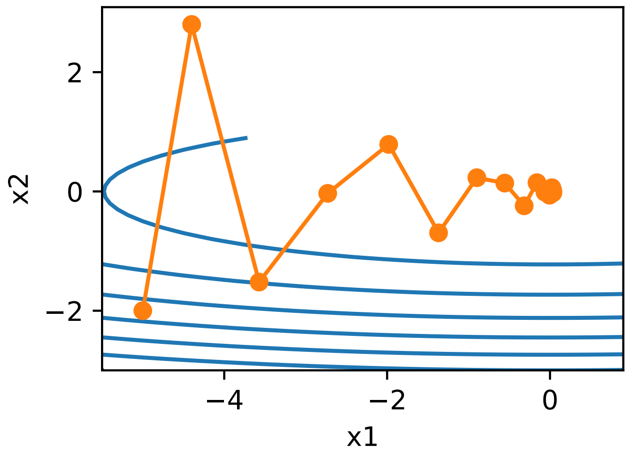
The momentum method allows us to solve the gradient descent problem described above. Looking at the optimization trace above we might intuit that averaging gradients over the past would work well. After all, in the x_1 direction this will aggregate well-aligned gradients, thus increasing the distance we cover with every step. Conversely, in the x_2 direction where gradients oscillate, an aggregate gradient will reduce step size due to oscillations that cancel each other out. Using $\mathbf{v}_t$ instead of the gradient $\mathbf{g}_t$ yields the following update equations:

$$\begin{aligned}\mathbf{v}_t &\leftarrow \beta \mathbf{v}_{t-1} + \mathbf{g}_{t,t-1}, \\ \mathbf{x}_t &\leftarrow \mathbf{x}_{t-1} - \eta_t \mathbf{v}_t.\end{aligned} \tag{11.6.5}$$

Note that for $\beta = 0$ we recover regular gradient descent. Before delving deeper into the mathematical properties let's have a quick look at how the algorithm behaves in practice.

```
def momentum_2d(x1, x2, v1, v2):
    v1 = beta * v1 + 0.2 * x1
    v2 = beta * v2 + 4 * x2
    return x1 - eta * v1, x2 - eta * v2, v1, v2

eta, beta = 0.6, 0.5
d2l.show_trace_2d(f_2d, d2l.train_2d(momentum_2d))
```

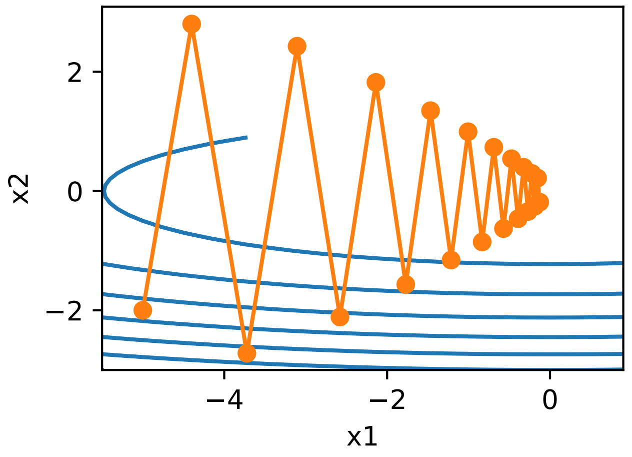
```
epoch 20, x1 0.007188, x2 0.002553
```



As we can see, even with the same learning rate that we used before, momentum still converges well. Let's see what happens when we decrease the momentum parameter. Halving it to $\beta = 0.25$ leads to a trajectory that barely converges at all. Nonetheless, it's a lot better than without momentum (when the solution diverges).

```
eta, beta = 0.6, 0.25
d2l.show_trace_2d(f_2d, d2l.train_2d(momentum_2d))
```

```
epoch 20, x1 -0.126340, x2 -0.186632
```

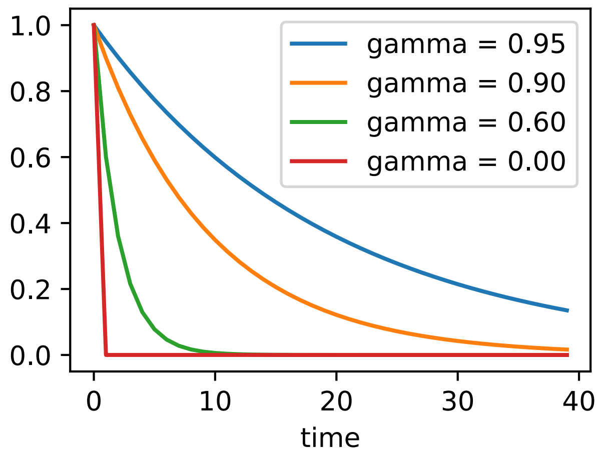


Note that we can combine momentum with SGD and in particular, minibatch-SGD. The only change is that in that case we replace the gradients $\mathbf{g}_{t,t-1}$ with $\mathbf{g}_t$. Last, for convenience we initialize $\mathbf{v}_0 = 0$ at time $t = 0$. Let's look at what leaky averaging actually does to the updates.

Effective Sample Weight

Recall that $\mathbf{v}_t = \sum_{\tau=0}^{t-1} \beta^\tau \mathbf{g}_{t-\tau, t-\tau-1}$. In the limit the terms add up to $\sum_{\tau=0}^\infty \beta^\tau = \frac{1}{1-\beta}$. In other words, rather than taking a step of size η in GD or SGD we take a step of size $\frac{\eta}{1-\beta}$ while at the same time, dealing with a potentially much better behaved descent direction. These are two benefits in one. To illustrate how weighting behaves for different choices of β consider the diagram below.

```
gammas = [0.95, 0.9, 0.6, 0]
d2l.set_figsize((3.5, 2.5))
for gamma in gammas:
    x = np.arange(40).asnumpy()
    d2l.plt.plot(x, gamma ** x, label='gamma = %.2f' % gamma)
d2l.plt.xlabel('time')
d2l.plt.legend();
```



11.6.2 Practical Experiments

Let's see how momentum works in practice, i.e., when used within the context of a proper optimizer. For this we need a somewhat more scalable implementation.

Implementation from Scratch

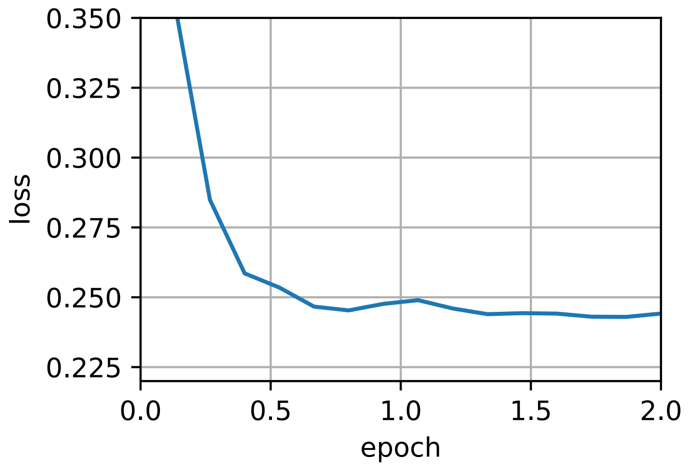
Compared with (minibatch) SGD the momentum method needs to maintain a set of auxiliary variables, i.e., velocity. It has the same shape as the gradients (and variables of the optimization problem). In the implementation below we call these variables `states`.

```
def init_momentum_states(feature_dim):
    v_w = np.zeros((feature_dim, 1))
    v_b = np.zeros(1)
    return (v_w, v_b)

def sgd_momentum(params, states, hyperparams):
    for p, v in zip(params, states):
        v[:] = hyperparams['momentum'] * v + p.grad
        p[:] -= hyperparams['lr'] * v
```

Let's see how this works in practice.

```
def train_momentum(lr, momentum, num_epochs=2):
    d2l.train_ch11(sgd_momentum, init_momentum_states(feature_dim),
                   {'lr': lr, 'momentum': momentum}, data_iter,
                   feature_dim, num_epochs)

data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
train_momentum(0.02, 0.5)
```

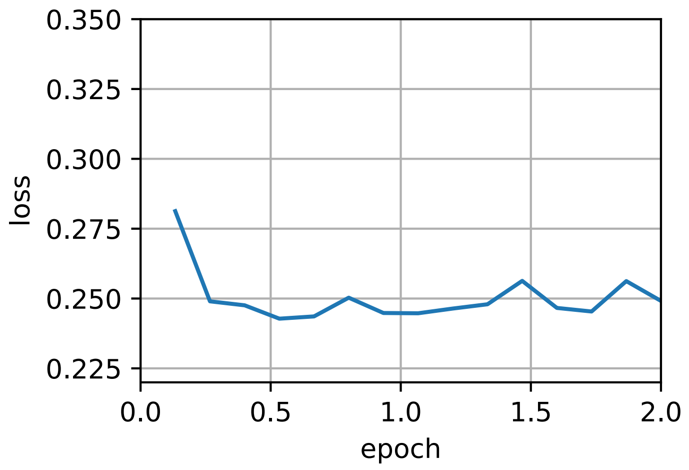
```
loss: 0.244, 0.053 sec/epoch
```



When we increase the momentum hyperparameter `momentum` to 0.9, it amounts to a significantly larger effective sample size of $\frac{1}{1-0.9} = 10$. We reduce the learning rate slightly to 0.01 to keep matters under control.

```
train_momentum(0.01, 0.9)
```

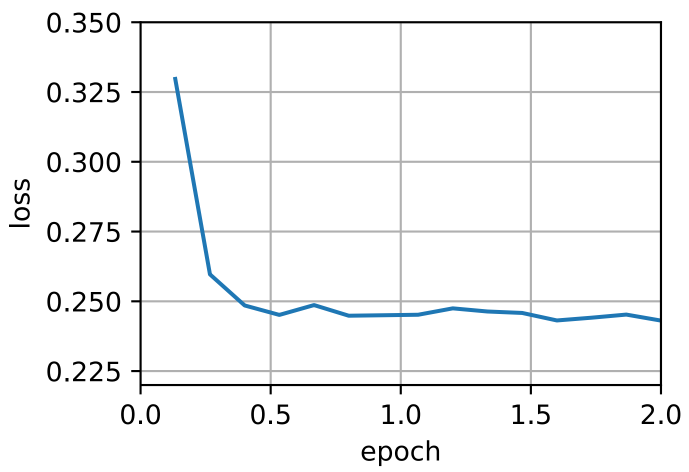
```
loss: 0.249, 0.054 sec/epoch
```



Reducing the learning rate further addresses any issue of non-smooth optimization problems. Setting it to 0.005 yields good convergence properties.

```
train_momentum(0.005, 0.9)
```

```
loss: 0.243, 0.053 sec/epoch
```

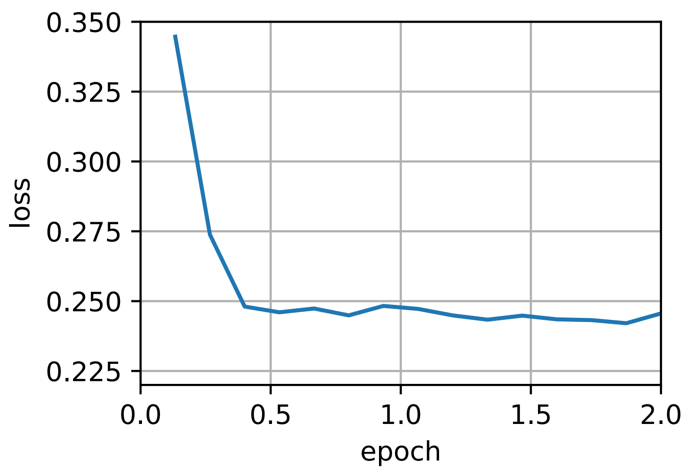


Concise Implementation

There's very little to do in Gluon since the standard sgd solver already had momentum built in. Setting matching parameters yields a very similar trajectory.

```
d2l.train_gluon_ch11('sgd', {'learning_rate': 0.005, 'momentum': 0.9},
                     data_iter)
```

```
loss: 0.246, 0.035 sec/epoch
```



11.6.3 Theoretical Analysis

So far the 2D example of $f(x) = 0.1x_1^2 + 2x_2^2$ seemed rather contrived. We will now see that this is actually quite representative of the types of problem one might encounter, at least in the case of minimizing convex quadratic objective functions.

Quadratic Convex Functions

Consider the function

$$h(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{x}^\top \mathbf{c} + b. \tag{11.6.6}$$

This is a general quadratic function. For positive semidefinite matrices $\mathbf{Q} \succ 0$, i.e., for matrices with positive eigenvalues this has a minimizer at $\mathbf{x}^* = -\mathbf{Q}^{-1}\mathbf{c}$ with minimum value $b - \frac{1}{2}\mathbf{c}^\top \mathbf{Q}^{-1}\mathbf{c}$. Hence we can rewrite h as

$$h(\mathbf{x}) = \frac{1}{2}(\mathbf{x} - \mathbf{Q}^{-1}\mathbf{c})^\top \mathbf{Q}(\mathbf{x} - \mathbf{Q}^{-1}\mathbf{c}) + b - \frac{1}{2}\mathbf{c}^\top \mathbf{Q}^{-1}\mathbf{c}. \tag{11.6.7}$$

The gradient is given by $\partial_{\mathbf{x}} f(\mathbf{x}) = \mathbf{Q}(\mathbf{x} - \mathbf{Q}^{-1}\mathbf{c})$. That is, it is given by the distance between $\mathbf{x}$ and the minimizer, multiplied by $\mathbf{Q}$. Consequently also the momentum is a linear combination of terms $\mathbf{Q}(\mathbf{x}_t - \mathbf{Q}^{-1}\mathbf{c})$.

Since $\mathbf{Q}$ is positive definite it can be decomposed into its eigensystem via $\mathbf{Q} = \mathbf{O}^\top \boldsymbol{\Lambda}\mathbf{O}$ for an orthogonal (rotation) matrix $\mathbf{O}$ and a diagonal matrix $\boldsymbol{\Lambda}$ of positive eigenvalues. This allows us to perform a change of variables from $\mathbf{x}$ to $\mathbf{z} := \mathbf{O}(\mathbf{x} - \mathbf{Q}^{-1}\mathbf{c})$ to obtain a much simplified expression:

$$h(\mathbf{z}) = \frac{1}{2}\mathbf{z}^\top \boldsymbol{\Lambda}\mathbf{z} + b'. \tag{11.6.8}$$

Here $c' = b - \frac{1}{2}\mathbf{c}^\top \mathbf{Q}^{-1}\mathbf{c}$. Since $\mathbf{O}$ is only an orthogonal matrix this doesn't perturb the gradients in a meaningful way. Expressed in terms of $\mathbf{z}$ gradient descent becomes

$$\mathbf{z}_t = \mathbf{z}_{t-1} - \boldsymbol{\Lambda}\mathbf{z}_{t-1} = (\mathbf{I} - \boldsymbol{\Lambda})\mathbf{z}_{t-1}. \tag{11.6.9}$$

The important fact in this expression is that gradient descent *does not mix* between different eigenspaces. That is, when expressed in terms of the eigensystem of $\mathbf{Q}$ the optimization problem proceeds in a coordinate-wise manner. This also holds for momentum.

$$\begin{aligned}
\mathbf{v}_t & = \beta \mathbf{v}_{t-1} + \boldsymbol{\Lambda}\mathbf{z}_{t-1} \\
\mathbf{z}_t & = \mathbf{z}_{t-1} - \eta \left(\beta \mathbf{v}_{t-1} + \boldsymbol{\Lambda}\mathbf{z}_{t-1}\right) \\
& = (\mathbf{I} - \eta \boldsymbol{\Lambda})\mathbf{z}_{t-1} - \eta \beta \mathbf{v}_{t-1}.
\end{aligned} \tag{11.6.10}$$

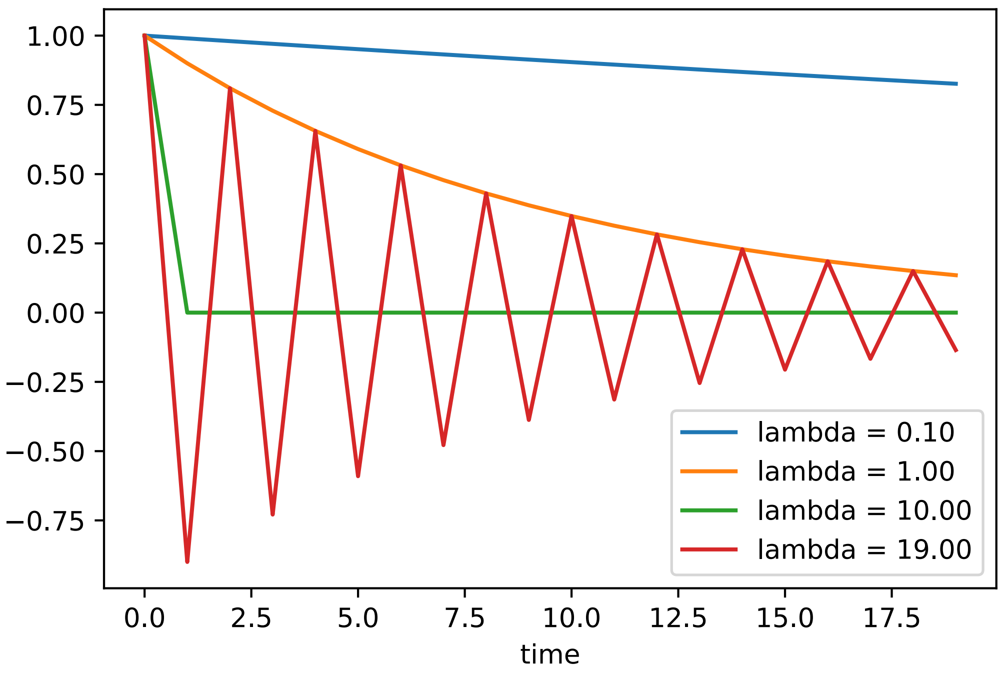
In doing this we just proved the following theorem: Gradient Descent with and without momentum for a convex quadratic function decomposes into coordinate-wise optimization in the direction of the eigenvectors of the quadratic matrix.

Given the above result let's see what happens when we minimize the function $f(x) = \frac{\lambda}{2}x^2$. For gradient descent we have

$$x_{t+1} = x_t - \eta\lambda x_t = (1 - \eta\lambda)x_t. \tag{11.6.11}$$

Whenever $|1 - \eta\lambda| < 1$ this optimization converges at an exponential rate since after t steps we have $x_t = (1 - \eta\lambda)^t x_0$. This shows how the rate of convergence improves initially as we increase the learning rate η until $\eta\lambda = 1$. Beyond that things diverge and for $\eta\lambda > 2$ the optimization problem diverges.

```
lambdas = [0.1, 1, 10, 19]
eta = 0.1
d2l.set_figsize((6, 4))
for lam in lambdas:
    t = np.arange(20).asnumpy()
    d2l.plt.plot(t, (1 - eta * lam) ** t, label='lambda = %.2f' % lam)
d2l.plt.xlabel('time')
d2l.plt.legend();
```



To analyze convergence in the case of momentum we begin by rewriting the update equations in terms of two scalars: one for x and one for the momentum v. This yields:

$$\begin{bmatrix} v_{t+1} \\ x_{t+1} \end{bmatrix} = \begin{bmatrix} \beta & \lambda \\ -\eta\beta & (1 - \eta\lambda) \end{bmatrix} \begin{bmatrix} v_t \\ x_t \end{bmatrix} = \mathbf{R}(\beta, \eta, \lambda) \begin{bmatrix} v_t \\ x_t \end{bmatrix}. \tag{11.6.12}$$

We used $\mathbf{R}$ to denote the 2×2 governing convergence behavior. After t steps the initial choice $[v_0, x_0]$ becomes $\mathbf{R}(\beta, \eta, \lambda)^t [v_0, x_0]$. Hence, it is up to the eigenvalues of $\mathbf{R}$ to detmine the speed of convergence. See the Distill post[158] of (Goh, 2017) for a great animation and (Flammarion & Bach,

[158] https://distill.pub/2017/momentum/

2015) for a detailed analysis. One can show that $0 < \eta\lambda < 2 + 2\beta$ momentum converges. This is a larger range of feasible parameters when compared to $0 < \eta\lambda < 2$ for gradient descent. It also suggests that in general large values of β are desirable. Further details require a fair amount of technical detail and we suggest that the interested reader consult the original publications.

Summary

- Momentum replaces gradients with a leaky average over past gradients. This accelerates convergence significantly.
- It is desirable for both noise-free gradient descent and (noisy) stochastic gradient descent.
- Momentum prevents stalling of the optimization process that is much more likely to occur for stochastic gradient descent.
- The effective number of gradients is given by $\frac{1}{1-\beta}$ due to exponentiated downweighting of past data.
- In the case of convex quadratic problems this can be analyzed explicitly in detail.
- Implementation is quite straightforward but it requires us to store an additional state vector (momentum $\mathbf{v}$).

Exercises

1. Use other combinations of momentum hyperparameters and learning rates and observe and analyze the different experimental results.
2. Try out GD and momentum for a quadratic problem where you have multiple eigenvalues, i.e., $f(x) = \frac{1}{2}\sum_i \lambda_i x_i^2$, e.g., $\lambda_i = 2^{-i}$. Plot how the values of x decrease for the initialization $x_i = 1$.
3. Derive minimum value and minimizer for $h(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{x}^\top \mathbf{c} + b$.
4. What changes when we perform SGD with momentum? What happens when we use minibatch SGD with momentum? Experiment with the parameters?

11.7 Adagrad

Let us begin by considering learning problems with features that occur infrequently.

11.7.1 Sparse Features and Learning Rates

Imagine that we're training a language model. To get good accuracy we typically want to decrease the learning rate as we keep on training, usually at a rate of $\mathcal{O}(t^{-\frac{1}{2}})$ or slower. Now consider a model training on sparse features, i.e., features that occur only infrequently. This is common for natural language, e.g., it is a lot less likely that we'll see the word *preconditioning* than *learning*. However, it is also common in other areas such as computational advertising and personalized collaborative filtering. After all, there are many things that are of interest only for a small number of people.

Parameters associated with infrequent features only receive meaningful updates whenever these features occur. Given a decreasing learning rate we might end up in a situation where the parameters for common features converge rather quickly to their optimal values, whereas for infrequent features we are still short of observing them sufficiently frequently before their optimal values can be determined. In other words, the learning rate either decreases too slowly for frequent features or too slowly for infrequent ones.

A possible hack to redress this issue would be to count the number of times we see a particular feature and to use this as a clock for adjusting learning rates. That is, rather than choosing a learning rate of the form $\eta = \frac{\eta_0}{\sqrt{t+c}}$ we could use $\eta_i = \frac{\eta_0}{\sqrt{s(i,t)+c}}$. Here $s(i,t)$ counts the number of nonzeros for feature i that we have observed up to time t. This is actually quite easy to implement at no meaningful overhead. However, it fails whenever we don't quite have sparsity but rather just data where the gradients are often very small and only rarely large. After all, it is unclear where one would draw the line between something that qualifies as an observed feature or not.

Adagrad by (Duchi et al., 2011) addresses this by replacing the rather crude counter $s(i,t)$ by an aggregate of the squares of previously observed gradients. In particular, it uses $s(i,t+1) = s(i,t) + (\partial_i f(\mathbf{x}))^2$ as a means to adjust the learning rate. This has two benefits: first, we no longer need to decide just when a gradient is large enough. Second, it scales automatically with the magnitude of the gradients. Coordinates that routinely correspond to large gradients are scaled down significantly, whereas others with small gradients receive a much more gentle treatment. In practice this leads to a very effective optimization procedure for computational advertising and related problems. But this hides some of the additional benefits inherent in Adagrad that are best understood in the context of preconditioning.

11.7.2 Preconditioning

Convex optimization problems are good for analyzing the characteristics of algorithms. After all, for most nonconvex problems it is difficult to derive meaningful theoretical guarantees, but *intuition* and *insight* often carry over. Let's look at the problem of minimizing $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{c}^\top \mathbf{x} + b$.

As we saw in Section 11.6, it is possible to rewrite this problem in terms of its eigendecomposition $\mathbf{Q} = \mathbf{U}^\top \boldsymbol{\Lambda} \mathbf{U}$ to arrive at a much simplified problem where each coordinate can be solved individually:

$$f(\mathbf{x}) = \bar{f}(\bar{\mathbf{x}}) = \frac{1}{2}\bar{\mathbf{x}}^\top \boldsymbol{\Lambda} \bar{\mathbf{x}} + \bar{\mathbf{c}}^\top \bar{\mathbf{x}} + b. \tag{11.7.1}$$

Here we used $\mathbf{x} = \mathbf{U}\mathbf{x}$ and consequently $\mathbf{c} = \mathbf{U}\mathbf{c}$. The modified problem has as its minimizer $\bar{\mathbf{x}} = -\boldsymbol{\Lambda}^{-1}\bar{\mathbf{c}}$ and minimum value $-\frac{1}{2}\bar{\mathbf{c}}^\top \boldsymbol{\Lambda}^{-1}\bar{\mathbf{c}} + b$. This is much easier to compute since $\boldsymbol{\Lambda}$ is a diagonal matrix containing the eigenvalues of $\mathbf{Q}$.

If we perturb $\mathbf{c}$ slightly we would hope to find only slight changes in the minimizer of f. Unfortunately this is not the case. While slight changes in $\mathbf{c}$ lead to equally slight changes in $\bar{\mathbf{c}}$, this is not the case for the minimizer of f (and of $\bar{f}$ respectively). Whenever the eigenvalues $\boldsymbol{\Lambda}_i$ are large we will see only small changes in $\bar{x}_i$ and in the minimum of $\bar{f}$. Conversely, for small $\boldsymbol{\Lambda}_i$ changes in $\bar{x}_i$ can be dramatic. The ratio between the largest and the smallest eigenvalue is called the condition number of an optimization problem.

$$\kappa = \frac{\boldsymbol{\Lambda}_1}{\boldsymbol{\Lambda}_d}. \tag{11.7.2}$$

If the condition number κ is large, it is difficult to solve the optimization problem accurately. We need to ensure that we are careful in getting a large dynamic range of values right. Our analysis leads to an obvious, albeit somewhat naive question: couldn't we simply "fix" the problem by distorting the space such that all eigenvalues are 1. In theory this is quite easy: we only need the eigenvalues and eigenvectors of $\mathbf{Q}$ to rescale the problem from $\mathbf{x}$ to one in $\mathbf{z} := \boldsymbol{\Lambda}^{\frac{1}{2}}\mathbf{U}\mathbf{x}$. In the new coordinate system $\mathbf{x}^\top\mathbf{Q}\mathbf{x}$ could be simplified to $\|\mathbf{z}\|^2$. Alas, this is a rather impractical suggestion. Computing eigenvalues and eigenvectors is in general *much more* expensive than solving the actual problem.

While computing eigenvalues exactly might be expensive, guessing them and computing them even somewhat approximately may already be a lot better than not doing anything at all. In particular, we could use the diagonal entries of $\mathbf{Q}$ and rescale it accordingly. This is *much* cheaper than computing eigenvalues.

$$\tilde{\mathbf{Q}} = \mathrm{diag}^{-\frac{1}{2}}(\mathbf{Q})\mathbf{Q}\mathrm{diag}^{-\frac{1}{2}}(\mathbf{Q}). \tag{11.7.3}$$

In this case we have $\tilde{\mathbf{Q}}_{ij} = \mathbf{Q}_{ij}/\sqrt{\mathbf{Q}_{ii}\mathbf{Q}_{jj}}$ and specifically $\tilde{\mathbf{Q}}_{ii} = 1$ for all i. In most cases this simplifies the condition number considerably. For instance, the the cases we discussed previously, this would entirely eliminate the problem at hand since the problem is axis aligned.

Unfortunately we face yet another problem: in deep learning we typically don't even have access to the second derivative of the objective function: for $\mathbf{x} \in \mathbb{R}^d$ the second derivative even on a minibatch may require $\mathcal{O}(d^2)$ space and work to compute, thus making it practically infeasible. The ingenious idea of Adagrad is to use a proxy for that elusive diagonal of the Hessian that is both relatively cheap to compute and effective—the magnitude of the gradient itself.

In order to see why this works, let's look at $\bar{f}(\bar{\mathbf{x}})$. We have that

$$\partial_{\bar{\mathbf{x}}} \bar{f}(\bar{\mathbf{x}}) = \boldsymbol{\Lambda}\bar{\mathbf{x}} + \bar{\mathbf{c}} = \boldsymbol{\Lambda}\left(\bar{\mathbf{x}} - \bar{\mathbf{x}}_0\right), \tag{11.7.4}$$

where $\bar{\mathbf{x}}_0$ is the minimizer of $\bar{f}$. Hence the magnitude of the gradient depends both on $\boldsymbol{\Lambda}$ and the distance from optimality. If $\bar{\mathbf{x}} - \bar{\mathbf{x}}_0$ didn't change, this would be all that's needed. After all, in this case the magnitude of the gradient $\partial_{\bar{\mathbf{x}}} \bar{f}(\bar{\mathbf{x}})$ suffices. Since AdaGrad is a stochastic gradient descent algorithm, we will see gradients with nonzero variance even at optimality. As a result we can safely use the variance of the gradients as a cheap proxy for the scale of the Hessian. A thorough analysis is beyond the scope of this section (it would be several pages). We refer the reader to (Duchi et al., 2011) for details.

11.7.3 The Algorithm

Let's formalize the discussion from above. We use the variable $\mathbf{s}_t$ to accumulate past gradient variance as follows.

$$\begin{aligned}
\mathbf{g}_t & = \partial_{\mathbf{w}} l(y_t, f(\mathbf{x}_t, \mathbf{w})), \\
\mathbf{s}_t & = \mathbf{s}_{t-1} + \mathbf{g}_t^2, \\
\mathbf{w}_t & = \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \cdot \mathbf{g}_t.
\end{aligned} \tag{11.7.5}$$

Here the operation are applied coordinate wise. That is, $\mathbf{v}^2$ has entries v_i^2. Likewise $\frac{1}{\sqrt{v}}$ has entries $\frac{1}{\sqrt{v_i}}$ and $\mathbf{u} \cdot \mathbf{v}$ has entries $u_i v_i$. As before η is the learning rate and ϵ is an additive constant that ensures that we do not divide by 0. Last, we initialize $\mathbf{s}_0 = \mathbf{0}$.

Just like in the case of momentum we need to keep track of an auxiliary variable, in this case to allow for an individual learning rate per coordinate. This doesn't increase the cost of Adagrad significantly relative to SGD, simply since the main cost is typically to compute $l(y_t, f(\mathbf{x}_t, \mathbf{w}))$ and its derivative.

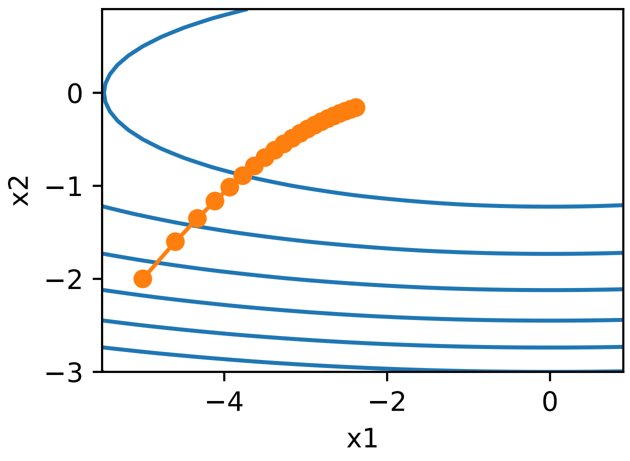
Note that accumulating squared gradients in $\mathbf{s}_t$ means that $\mathbf{s}_t$ grows essentially at linear rate (somewhat slower than linearly in practice, since the gradients initially diminish). This leads to an $\mathcal{O}(t^{-\frac{1}{2}})$ learning rate, albeit adjusted on a per coordinate basis. For convex problems this is perfectly adequate. In deep learning, though, we might want to decrease the learning rate rather more slowly. This led to a number of Adagrad variants that we will discuss in the subsequent chapters. For now let's see how it behaves in a quadratic convex problem. We use the same problem as before:

$$f(\mathbf{x}) = 0.1 x_1^2 + 2 x_2^2. \tag{11.7.6}$$

We are going to implement Adagrad using the same learning rate previously, i.e., $\eta = 0.4$. As we can see, the iterative trajectory of the independent variable is smoother. However, due to the cumulative effect of s_t, the learning rate continuously decays, so the independent variable does not move as much during later stages of iteration.

```python
%matplotlib inline
import d2l
import math
from mxnet import np, npx
npx.set_np()

def adagrad_2d(x1, x2, s1, s2):
    eps = 1e-6
    g1, g2 = 0.2 * x1, 4 * x2
    s1 += g1 ** 2
    s2 += g2 ** 2
    x1 -= eta / math.sqrt(s1 + eps) * g1
    x2 -= eta / math.sqrt(s2 + eps) * g2
    return x1, x2, s1, s2

def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2

eta = 0.4
d2l.show_trace_2d(f_2d, d2l.train_2d(adagrad_2d))
```

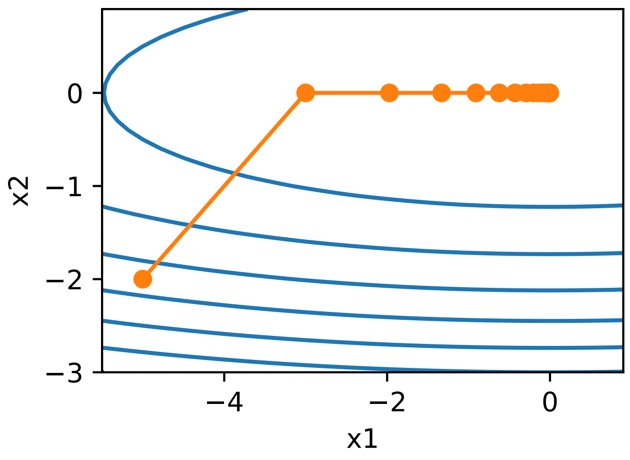
```
epoch 20, x1 -2.382563, x2 -0.158591
```



As we increase the learning rate to 2 we see much better behavior. This already indicates that the decrease in learning rate might be rather aggressive, even in the noise-free case and we need to ensure that parameters converge appropriately.

```
eta = 2
d2l.show_trace_2d(f_2d, d2l.train_2d(adagrad_2d))
```

```
epoch 20, x1 -0.002295, x2 -0.000000
```



11.7.4 Implementation from Scratch

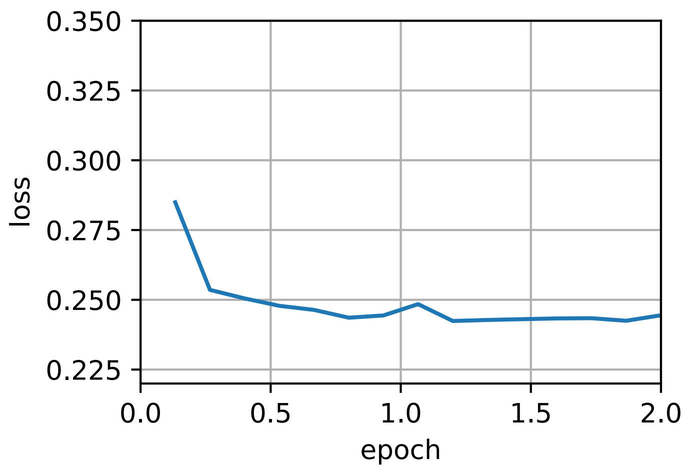
Just like the momentum method, Adagrad needs to maintain a state variable of the same shape as the parameters.

```
def init_adagrad_states(feature_dim):
    s_w = np.zeros((feature_dim, 1))
    s_b = np.zeros(1)
    return (s_w, s_b)

def adagrad(params, states, hyperparams):
    eps = 1e-6
    for p, s in zip(params, states):
        s[:] += np.square(p.grad)
        p[:] -= hyperparams['lr'] * p.grad / np.sqrt(s + eps)
```

Compared to the experiment in Section 11.5 we use a larger learning rate to train the model.

```
data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(adagrad, init_adagrad_states(feature_dim),
               {'lr': 0.1}, data_iter, feature_dim);
```

```
loss: 0.244, 0.052 sec/epoch
```

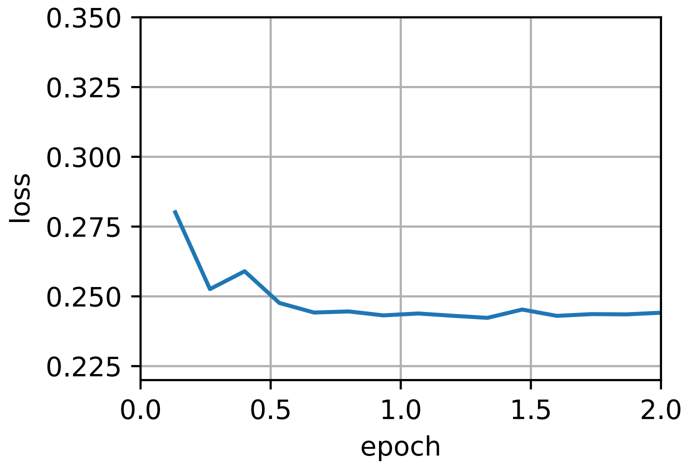


11.7.5 Concise Implementation

Using the `Trainer` instance of the algorithm `adagrad`, we can invoke the Adagrad algorithm in Gluon.

```
d2l.train_gluon_ch11('adagrad', {'learning_rate': 0.1}, data_iter)
```

```
loss: 0.244, 0.069 sec/epoch
```



Summary

- Adagrad decreases the learning rate dynamically on a per-coordinate basis.
- It uses the magnitude of the gradient as a means of adjusting how quickly progress is achieved - coordinates with large gradients are compensated with a smaller learning rate.
- Computing the exact second derivative is typically infeasible in deep learning problems due to memory and computational constraints. The gradient can be a useful proxy.
- If the optimization problem has a rather uneven uneven structure Adagrad can help mitigate the distortion.
- Adagrad is particularly effective for sparse features where the learning rate needs to decrease more slowly for infrequently occurring terms.
- On deep learning problems Adagrad can sometimes be too aggressive in reducing learning rates. We will discuss strategies for mitigating this in the context of Section 11.10.

Exercises

1. Prove that for an orthogonal matrix **U** and a vector **c** the following holds: $\|\mathbf{c} - \delta\|_2 = \|\mathbf{U}\mathbf{c} - \mathbf{U}\delta\|_2$. Why does this mean that the magnitude of perturbations does not change after an orthogonal change of variables?
2. Try out Adagrad for $f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$ and also for the objective function was rotated by 45 degrees, i.e., $f(\mathbf{x}) = 0.1(x_1 + x_2)^2 + 2(x_1 - x_2)^2$. Does it behave differently?
3. Prove Gerschgorin's circle theorem[160] which states that eigenvalues λ_i of a matrix **M** satisfy $|\lambda_i - \mathbf{M}_{jj}| \leq \sum_{k \neq j} |\mathbf{M}_{jk}|$ for at least one choice of j.
4. What does Gerschgorin's theorem tell us about the eigenvalues of the diagonally preconditioned matrix $\mathrm{diag}^{-\frac{1}{2}}(\mathbf{M})\mathbf{M}\mathrm{diag}^{-\frac{1}{2}}(\mathbf{M})$?
5. Try out Adagrad for a proper deep network, such as Section 6.6 when applied to Fashion MNIST.
6. How would you need to modify Adagrad to achieve a less aggressive decay in learning rate?

[160] https://en.wikipedia.org/wiki/Gershgorin_circle_theorem

11.8 RMSProp

One of the key issues in Section 11.7 is that the learning rate decreases at a predefined schedule of effectively $\mathcal{O}(t^{-\frac{1}{2}})$. While this is generally appropriate for convex problems, it might not be ideal for nonconvex ones, such as those encountered in deep learning. Yet, the coordinate-wise adaptivity of Adagrad is highly desirable as a preconditioner.

(Tieleman & Hinton, 2012) proposed the RMSProp algorithm as a simple fix to decouple rate scheduling from coordinate-adaptive learning rates. The issue is that Adagrad accumulates the squares of the gradient $\mathbf{g}_t$ into a state vector $\mathbf{s}_t = \mathbf{s}_{t-1} + \mathbf{g}_t^2$. As a result $\mathbf{s}_t$ keeps on growing without bound due to the lack of normalization, essentially linarly as the algorithm converges.

One way of fixing this problem would be to use $\mathbf{s}_t / t$. For reasonable distributions of $\mathbf{g}_t$ this will converge. Unfortunately it might take a very long time until the limit behavior starts to matter since the procedure remembers the full trajectory of values. An alternative is to use a leaky average in the same way we used in the momentum method, i.e., $\mathbf{s}_t \leftarrow \gamma \mathbf{s}_{t-1} + (1-\gamma)\mathbf{g}_t^2$ for some parameter $\gamma > 0$. Keeping all other parts unchanged yields RMSProp.

11.8.1 The Algorithm

Let's write out the equations in detail.

$$\begin{aligned} \mathbf{s}_t & \leftarrow \gamma \mathbf{s}_{t-1} + (1 - \gamma) \mathbf{g}_t^2, \\ \mathbf{x}_t & \leftarrow \mathbf{x}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t. \end{aligned} \tag{11.8.1}$$

The constant $\epsilon > 0$ is typically set to 10^{-6} to ensure that we don't suffer from division by zero or overly large step sizes. Given this expansion we are now free to control the learning rate η independently of the scaling that is applied on a per-coordinate basis. In terms of leaky averages we can apply the same reasoning as previously applied in the case of the momentum method. Expanding the definition of $\mathbf{s}_t$ yields

$$\begin{aligned} \mathbf{s}_t & = (1 - \gamma) \mathbf{g}_t^2 + \gamma \mathbf{s}_{t-1} \\ & = (1 - \gamma) \left(\mathbf{g}_t^2 + \gamma \mathbf{g}_{t-1}^2 + \gamma^2 \mathbf{g}_{t-2} + \ldots \right). \end{aligned} \tag{11.8.2}$$

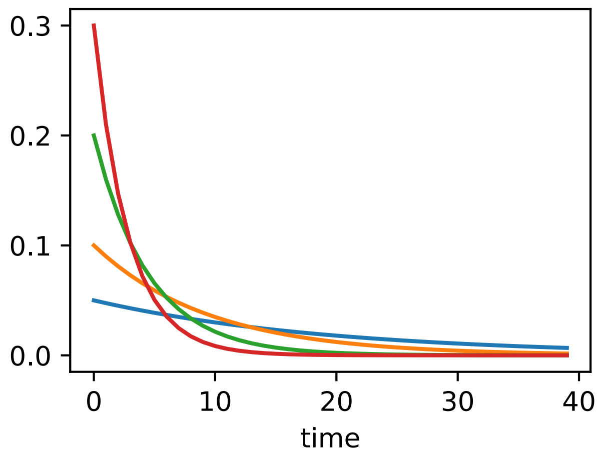
As before in Section 11.6 we use $1 + \gamma + \gamma^2 + \ldots = \frac{1}{1-\gamma}$. Hence the sum of weights is normalized to 1 with a half-life time of an observation of γ^{-1}. Let's visualize the weights for the past 40 timesteps for various choices of γ.

```
%matplotlib inline
import d2l
import math
from mxnet import np, npx
```

(continues on next page)

(continued from previous page)

```
npx.set_np()
d2l.set_figsize((3.5, 2.5))

gammas = [0.95, 0.9, 0.8, 0.7]
for gamma in gammas:
    x = np.arange(40).asnumpy()
    d2l.plt.plot(x, (1-gamma) * gamma ** x, label='gamma = %.2f' % gamma)
d2l.plt.xlabel('time');
```

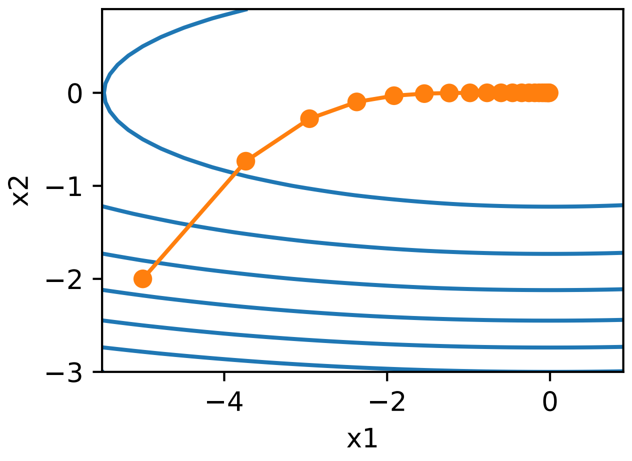


11.8.2 Implementation from Scratch

As before we use the quadratic function $f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$ to observe the trajectory of RMSProp. Recall that in Section 11.7, when we used Adagrad with a learning rate of 0.4, the variables moved only very slowly in the later stages of the algorithm since the learning rate decreased too quickly. Since η is controlled separately this does not happen with RMSProp.

```
def rmsprop_2d(x1, x2, s1, s2):
    g1, g2, eps = 0.2 * x1, 4 * x2, 1e-6
    s1 = gamma * s1 + (1 - gamma) * g1 ** 2
    s2 = gamma * s2 + (1 - gamma) * g2 ** 2
    x1 -= eta / math.sqrt(s1 + eps) * g1
    x2 -= eta / math.sqrt(s2 + eps) * g2
    return x1, x2, s1, s2

def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2

eta, gamma = 0.4, 0.9
d2l.show_trace_2d(f_2d, d2l.train_2d(rmsprop_2d))
```

```
epoch 20, x1 -0.010599, x2 0.000000
```



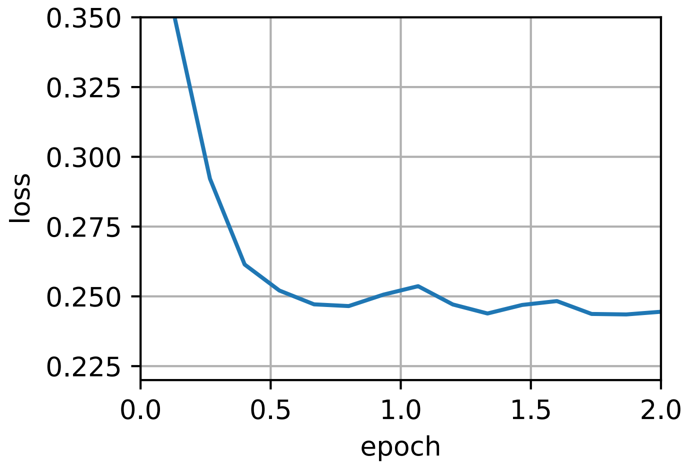
Next, we implement RMSProp to be used in a deep network. This is equally straightforward.

```
def init_rmsprop_states(feature_dim):
    s_w = np.zeros((feature_dim, 1))
    s_b = np.zeros(1)
    return (s_w, s_b)

def rmsprop(params, states, hyperparams):
    gamma, eps = hyperparams['gamma'], 1e-6
    for p, s in zip(params, states):
        s[:] = gamma * s + (1 - gamma) * np.square(p.grad)
        p[:] -= hyperparams['lr'] * p.grad / np.sqrt(s + eps)
```

We set the initial learning rate to 0.01 and the weighting term γ to 0.9. That is, **s** aggregates on average over the past $1/(1-\gamma) = 10$ observations of the square gradient.

```
data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(rmsprop, init_rmsprop_states(feature_dim),
               {'lr': 0.01, 'gamma': 0.9}, data_iter, feature_dim);
```

```
loss: 0.244, 0.067 sec/epoch
```

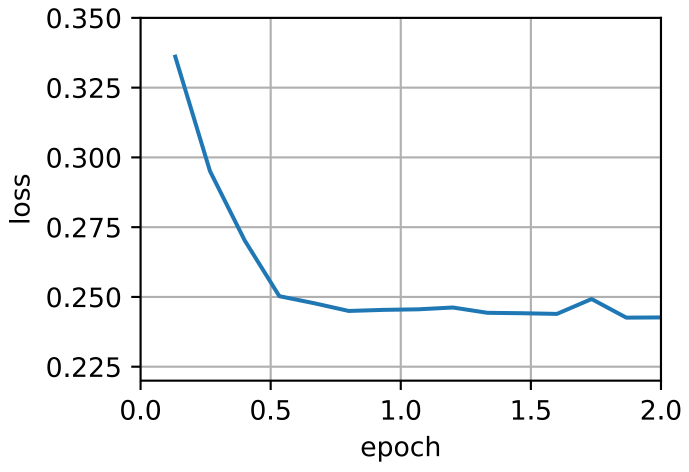


11.8.3 Concise Implementation

Since RMSProp is a rather popular algorithm it is also available in the `Trainer` instance. All we need to do is instantiate it using an algorithm named `rmsprop`, assigning γ to the parameter `gamma1`.

```
d2l.train_gluon_ch11('rmsprop', {'learning_rate': 0.01, 'gamma1': 0.9},
                     data_iter)
```

```
loss: 0.243, 0.036 sec/epoch
```



Summary

- RMSProp is very similar to Adagrad insofar as both use the square of the gradient to scale coefficients.
- RMSProp shares with momentum the leaky averaging. However, RMSProp uses the technique to adjust the coefficient-wise preconditioner.
- The learning rate needs to be scheduled by the experimenter in practice.

- The coefficient γ determines how long the history is when adjusting the per-coordinate scale.

Exercises

1. What happens experimentally if we set $\gamma = 1$? Why?
2. Rotate the optimization problem to minimize $f(\mathbf{x}) = 0.1(x_1 + x_2)^2 + 2(x_1 - x_2)^2$. What happens to the convergence?
3. Try out what happens to RMSProp on a real machine learning problem, such as training on FashionMNIST. Experiment with different choices for adjusting the learning rate.
4. Would you want to adjust γ as optimization progresses? How sensitive is RMSProp to this?

11.9 Adadelta

Adadelta is yet another variant of AdaGrad. The main difference lies in the fact that it decreases the amount by which the learning rate is adaptive to coordinates. Moreover, traditionally it referred to as not having a learning rate since it uses the amount of change itself as calibration for future change. The algorithm was proposed in (Zeiler, 2012). It is fairly straightforward, given the discussion of previous algorithms so far.

11.9.1 The Algorithm

In a nutshell Adadelta uses two state variables, $\mathbf{s}_t$ to store a leaky average of the second moment of the gradient and $\Delta\mathbf{x}_t$ to store a leaky average of the second moment of the change of parameters in the model itself. Note that we use the original notation and naming of the authors for compatibility with other publications and implementations (there's no other real reason why one should use different Greek variables to indicate a parameter serving the same purpose in momentum, Adagrad, RMSProp, and Adadelta). The parameter du jour is ρ. We obtain the following leaky updates:

$$\begin{aligned}
\mathbf{s}_t &= \rho \mathbf{s}_{t-1} + (1-\rho)\mathbf{g}_t^2, \\
\mathbf{g}_t' &= \sqrt{\frac{\Delta\mathbf{x}_{t-1} + \epsilon}{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t, \\
\mathbf{x}_t &= \mathbf{x}_{t-1} - \mathbf{g}_t', \\
\Delta\mathbf{x}_t &= \rho \Delta\mathbf{x}_{t-1} + (1-\rho)\mathbf{x}_t^2.
\end{aligned} \tag{11.9.1}$$

The difference to before is that we perform updates with the rescaled gradient $\mathbf{g}_t'$ which is computed by taking the ratio between the average squared rate of change and the average second moment of the gradient. The use of $\mathbf{g}_t'$ is purely for notational convenience. In practice we can implement this algorithm without the need to use additional temporary space for $\mathbf{g}_t'$. As before η is a parameter ensuring nontrivial numerical results, i.e., avoiding zero step size or infinite variance. Typically we set this to $\eta = 10^{-5}$.

11.9.2 Implementation

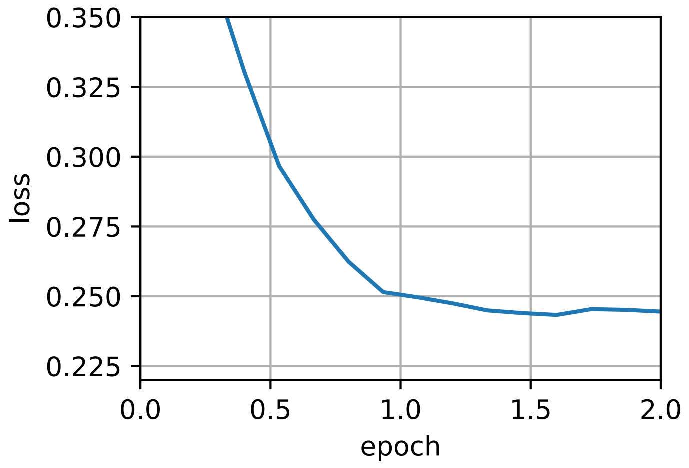
Adadelta needs to maintain two state variables for each variable, $\mathbf{s}_t$ and $\Delta\mathbf{x}_t$. This yields the following implementation.

```
%matplotlib inline
import d2l
from mxnet import np, npx
npx.set_np()

def init_adadelta_states(feature_dim):
    s_w, s_b = np.zeros((feature_dim, 1)), np.zeros(1)
    delta_w, delta_b = np.zeros((feature_dim, 1)), np.zeros(1)
    return ((s_w, delta_w), (s_b, delta_b))

def adadelta(params, states, hyperparams):
    rho, eps = hyperparams['rho'], 1e-5
    for p, (s, delta) in zip(params, states):
        # In-place updates via [:]
        s[:] = rho * s + (1 - rho) * np.square(p.grad)
        g = (np.sqrt(delta + eps) / np.sqrt(s + eps)) * p.grad
        p[:] -= g
        delta[:] = rho * delta + (1 - rho) * g * g
```

Choosing $\rho = 0.9$ amounts to a half-life time of 10 for each parameter update. This tends to work quite well. We get the following behavior.

```
data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(adadelta, init_adadelta_states(feature_dim),
               {'rho': 0.9}, data_iter, feature_dim);
```

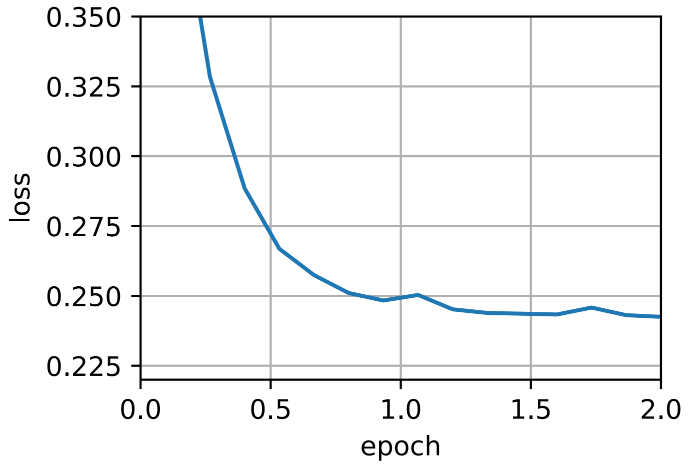
```
loss: 0.245, 0.089 sec/epoch
```



For a concise implementation we simply use the `adadelta` algorithm from the `Trainer` class. This yields the following one-liner for a much more compact invocation.

```
d2l.train_gluon_ch11('adadelta', {'rho': 0.9}, data_iter)
```

```
loss: 0.243, 0.089 sec/epoch
```



Summary

- Adadelta has no learning rate parameter. Instead, it uses the rate of change in the parameters itself to adapt the learning rate.
- Adadelta requires two state variables to store the second moments of gradient and the change in parameters.
- Adadelta uses leaky averages to keep a running estimate of the appropriate statistics.

Exercises

1. Adjust the value of ρ. What happens?
2. Show how to implement the algorithm without the use of $\mathbf{g}_t'$. Why might this be a good idea?
3. Is Adadelta really learning rate free? Could you find optimization problems that break Adadelta?
4. Compare Adadelta to Adagrad and RMS prop to discuss their convergence behavior.

11.10 Adam

In the discussions leading up to this section we encountered a number of techniques for efficient optimization. Let's recap them in detail here:

- We saw that Section 11.4 is more effective than Gradient Descent when solving optimization problems, e.g., due to its inherent resilience to redundant data.
- We saw that Section 11.5 affords significant additional efficiency arising from vectorization, using larger sets of observations in one minibatch. This is the key to efficient multi-machine, multi-GPU and overall parallel processing.
- Section 11.6 added a mechanism for aggregating a history of past gradients to accelerate convergence.
- Section 11.7 used per-coordinate scaling to allow for a computationally efficient preconditioner.
- Section 11.8 decoupled per-coordinate scaling from a learning rate adjustment.

Adam (Kingma & Ba, 2014) combines all these techniques into one efficient learning algorithm. As expected, this is an algorithm that has become rather popular as one of the more robust and effective optimization algorithms to use in deep learning. It is not without issues, though. In particular, (Reddi et al., 2019) show that there are situations where Adam can diverge due to poor variance control. In a follow-up work (Zaheer et al., 2018) proposed a hotfix to Adam, called Yogi which addresses these issues. More on this later. For now let's review the Adam algorithm.

11.10.1 The Algorithm

One of the key components of Adam is that it uses exponential weighted moving averages (also known as leaky averaging) to obtain an estimate of both the momentum and also the second moment of the gradient. That is, it uses the state variables

$$\begin{aligned}\mathbf{v}_t &\leftarrow \beta_1 \mathbf{v}_{t-1} + (1 - \beta_1) \mathbf{g}_t, \\ \mathbf{s}_t &\leftarrow \beta_2 \mathbf{s}_{t-1} + (1 - \beta_2) \mathbf{g}_t^2.\end{aligned} \tag{11.10.1}$$

Here β_1 and β_2 are nonnegative weighting parameters. Common choices for them are $\beta_1 = 0.9$ and $\beta_2 = 0.999$. That is, the variance estimate moves *much more slowly* than the momentum term. Note that if we initialize $\mathbf{v}_0 = \mathbf{s}_0 = 0$ we have a significant amount of bias initially towards smaller values. This can be addressed by using the fact that $\sum_{i=0}^t \beta^i = \frac{1 - \beta^t}{1 - \beta}$ to re-normalize terms. Correspondingly the normalized state variables are given by

$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1 - \beta_1^t} \text{ and } \hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1 - \beta_2^t}. \tag{11.10.2}$$

Armed with the proper estimates we can now write out the update equations. First, we rescale the gradient in a manner very much akin to that of RMSProp to obtain

$$\mathbf{g}_t' = \frac{\eta \hat{\mathbf{v}}_t}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon}. \tag{11.10.3}$$

Unlike RMSProp our update uses the momentum $\hat{\mathbf{v}}_t$ rather than the gradient itself. Moreover, there's a slight cosmetic difference as the rescaling happens using $\frac{1}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon}$ instead of $\frac{1}{\sqrt{\hat{\mathbf{s}}_t + \epsilon}}$. The former works arguably slightly better in practice, hence the deviation from RMSProp. Typically we pick $\epsilon = 10^{-6}$ for a good trade-off between numerical stability and fidelity.

Now we have all the pieces in place to compute updates. This is slightly anticlimactic and we have a simple update of the form

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \mathbf{g}_t'. \tag{11.10.4}$$

Reviewing the design of Adam its inspiration is clear. Momentum and scale are clearly visible in the state variables. Their rather peculiar definition forces us to debias terms (this could be fixed by a slightly different initialization and update condition). Second, the combination of both terms is pretty straightforward, given RMSProp. Last, the explicit learning rate η allows us to control the step length to address issues of convergence.

11.10.2 Implementation

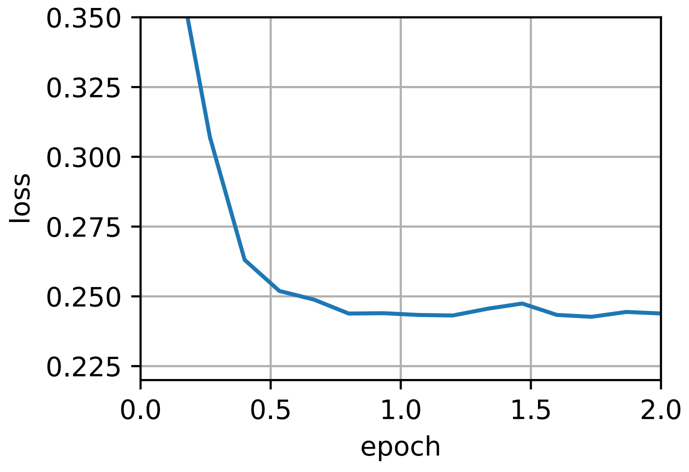
Implementing Adam from scratch isn't very daunting. For convenience we store the timestep counter t in the `hyperparams` dictionary. Beyond that all is straightforward.

```
%matplotlib inline
import d2l
from mxnet import np, npx
npx.set_np()

def init_adam_states(feature_dim):
    v_w, v_b = np.zeros((feature_dim, 1)), np.zeros(1)
    s_w, s_b = np.zeros((feature_dim, 1)), np.zeros(1)
    return ((v_w, s_w), (v_b, s_b))

def adam(params, states, hyperparams):
    beta1, beta2, eps = 0.9, 0.999, 1e-6
    for p, (v, s) in zip(params, states):
        v[:] = beta1 * v + (1 - beta1) * p.grad
        s[:] = beta2 * s + (1 - beta2) * np.square(p.grad)
        v_bias_corr = v / (1 - beta1 ** hyperparams['t'])
        s_bias_corr = s / (1 - beta2 ** hyperparams['t'])
        p[:] -= hyperparams['lr'] * v_bias_corr / (np.sqrt(s_bias_corr) + eps)
    hyperparams['t'] += 1
```

We are ready to use Adam to train the model. We use a learning rate of $\eta = 0.01$.

```
data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(adam, init_adam_states(feature_dim),
               {'lr': 0.01, 't': 1}, data_iter, feature_dim);
```

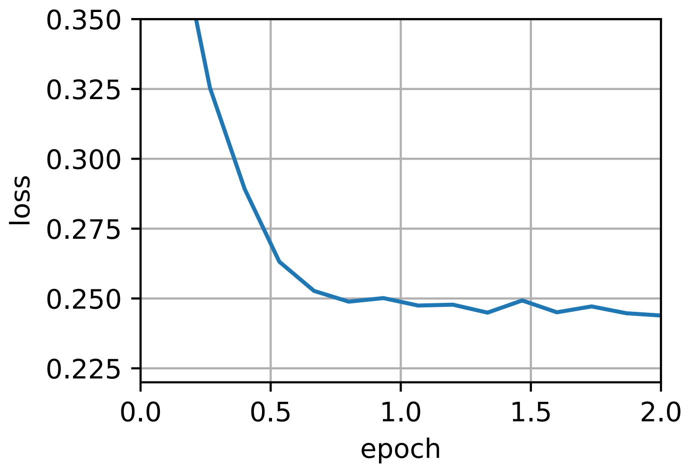
```
loss: 0.244, 0.091 sec/epoch
```



A more concise implementation is straightforward since `adam` is one of the algorithms provided as part of the Gluon `trainer` optimization library. Hence we only need to pass configuration parameters for an implementation in Gluon.

```
d2l.train_gluon_ch11('adam', {'learning_rate': 0.01}, data_iter)
```

```
loss: 0.244, 0.039 sec/epoch
```



11.10.3 Yogi

One of the problems of Adam is that it can fail to converge even in convex settings when the second moment estimate in $\mathbf{s}_t$ blows up. As a fix (Zaheer et al., 2018) proposed a refined update (and initialization) for $\mathbf{s}_t$. To understand what's going on, let's rewrite the Adam update as follows:

$$\mathbf{s}_t \leftarrow \mathbf{s}_{t-1} + (1 - \beta_2) \left(\mathbf{g}_t^2 - \mathbf{s}_{t-1}\right). \tag{11.10.5}$$

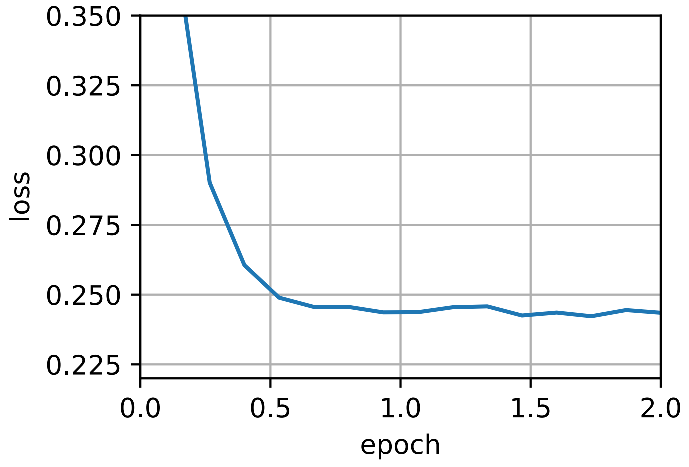
Whenever $\mathbf{g}_t^2$ has high variance or updates are sparse, $\mathbf{s}_t$ might forget past values too quickly. A possible fix for this is to replace $\mathbf{g}_t^2 - \mathbf{s}_{t-1}$ by $\mathbf{g}_t^2 \odot \mathop{\mathrm{sgn}}(\mathbf{g}_t^2 - \mathbf{s}_{t-1})$. Now the magnitude of the update no longer depends on the amount of deviation. This yields the Yogi updates

$$\mathbf{s}_t \leftarrow \mathbf{s}_{t-1} + (1 - \beta_2) \mathbf{g}_t^2 \odot \mathop{\mathrm{sgn}}(\mathbf{g}_t^2 - \mathbf{s}_{t-1}). \tag{11.10.6}$$

The authors furthermore advise to initialize the momentum on a larger initial batch rather than just initial pointwise estimate. We omit the details since they are not material to the discussion and since even without this convergence remains pretty good.

```
def yogi(params, states, hyperparams):
    beta1, beta2, eps = 0.9, 0.999, 1e-3
    for p, (v, s) in zip(params, states):
        v[:] = beta1 * v + (1 - beta1) * p.grad
        s[:] = s + (1 - beta2) * np.sign(
            np.square(p.grad) - s) * np.square(p.grad)
        v_bias_corr = v / (1 - beta1 ** hyperparams['t'])
        s_bias_corr = s / (1 - beta2 ** hyperparams['t'])
        p[:] -= hyperparams['lr'] * v_bias_corr / (np.sqrt(s_bias_corr) + eps)
    hyperparams['t'] += 1

data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(yogi, init_adam_states(feature_dim),
               {'lr': 0.01, 't': 1}, data_iter, feature_dim);
```

```
loss: 0.244, 0.092 sec/epoch
```



Summary

- Adam combines features of many optimization algorithms into a fairly robust update rule.
- Created on the basis of RMSProp, Adam also uses EWMA on the minibatch stochastic gradient
- Adam uses bias correction to adjust for a slow startup when estimating momentum and a second moment.
- For gradients with significant variance we may encounter issues with convergence. They can be amended by using larger minibatches or by switching to an improved estimate for $\mathbf{s}_t$. Yogi offers such an alternative.

Exercises

1. Adjust the learning rate and observe and analyze the experimental results.
2. Can you rewrite momentum and second moment updates such that it doesn't require bias correction?
3. Why do you need to reduce the learning rate η as we converge?
4. Try to construct a case for which Adam diverges and Yogi converges?

11.11 Learning Rate Scheduling

So far we primarily focused on optimization *algorithms* for how to update the weight vectors rather than on the *rate* at which they're being updated. Nonetheless, adjusting the learning rate is often just as important as the actual algorithm. There are a number of aspects to consider:

- Most obviously the *magnitude* of the learning rate matters. If it's too large, optimization diverges, if it's too small, it takes too long to train or we end up with a suboptimal result. We saw previously that the condition number of the problem matters (see e.g., Section 11.6 for details). Intuitively it's the ratio of the amount of change in the least sensitive direction vs. the most sensitive one.
- Secondly, the rate of decay is just as important. If the learning rate remains large we may simply end up bouncing around the minimum and thus not reach optimality. Section 11.5 discussed this in some detail and we analyzed performance guarantees in Section 11.4. In short, we want the rate to decay, but probably more slowly than $\mathcal{O}(t^{-\frac{1}{2}})$ which would be a good choice for convex problems.
- Another aspect that is equally important is *initialization*. This pertains both to how the parameters are set initially (review Section 4.8 for details) and also how they evolve initially. This goes under the moniker of *warmup*, i.e., how rapidly we start moving towards the solution initially. Large steps in the beginning might not be beneficial, in particular since the initial set of parameters is random. The initial update directions might be quite meaningless, too.
- Lastly, there are a number of optimization variants that perform cyclical learning rate adjustment. This is beyond the scope of the current chapter. We recommend the reader to review details in (Izmailov et al., 2018), e.g., how to obtain better solutions by averaging over an entire *path* of parameters.

Given the fact that there's a lot of detail needed to manage learning rates, most deep learning frameworks have tools to deal with this automatically. In the current chapter we will review the effects that different schedules have on accuracy and also show how this can be managed efficiently via a *learning rate scheduler*.

11.11.1 Toy Problem

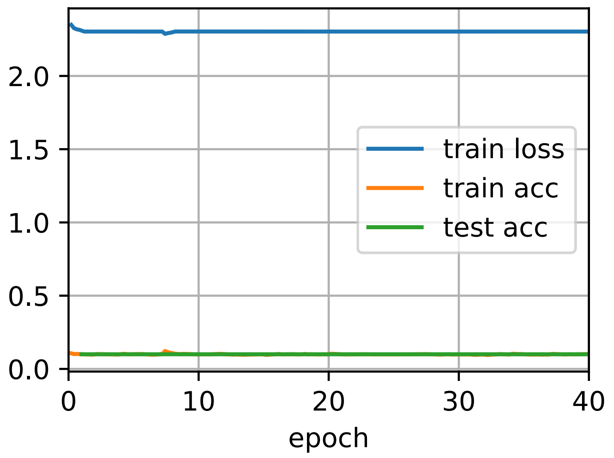
We begin with a toy problem that is cheap enough to compute easily, yet sufficiently nontrivial to illustrate some of the key aspects. For that we pick a slightly modernized version of LeNet (`relu` instead of `sigmoid` activation, MaxPooling rather than AveragePooling), as applied to Fashion MNIST. Moreover, we hybridize the network for performance. Since most of the code is standard we just introduce the basics without further detailed discussion. See Chapter 6 for a refresher as needed.

```python
%matplotlib inline
import d2l
from mxnet import autograd, gluon, init, lr_scheduler, np, npx
from mxnet.gluon import nn
npx.set_np()

net = nn.HybridSequential()
net.add(nn.Conv2D(channels=6, kernel_size=5, padding=2, activation='relu'),
        nn.MaxPool2D(pool_size=2, strides=2),
        nn.Conv2D(channels=16, kernel_size=5, activation='relu'),
        nn.MaxPool2D(pool_size=2, strides=2),
        nn.Dense(120, activation='relu'),
        nn.Dense(84, activation='relu'),
        nn.Dense(10))
net.hybridize()
loss = gluon.loss.SoftmaxCrossEntropyLoss()
ctx = d2l.try_gpu()

batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size=batch_size)

# The code is almost identical to "d2l.train_ch5" that defined in the lenet
# section of chapter convolutional neural networks
def train(net, train_iter, test_iter, num_epochs, loss, trainer, ctx):
    net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
    animator = d2l.Animator(xlabel='epoch', xlim=[0, num_epochs],
                            legend=['train loss', 'train acc', 'test acc'])
    for epoch in range(num_epochs):
        metric = d2l.Accumulator(3)  # train_loss, train_acc, num_examples
        for i, (X, y) in enumerate(train_iter):
            X, y = X.as_in_context(ctx), y.as_in_context(ctx)
            with autograd.record():
                y_hat = net(X)
                l = loss(y_hat, y)
            l.backward()
            trainer.step(X.shape[0])
            metric.add(l.sum(), d2l.accuracy(y_hat, y), X.shape[0])
            train_loss, train_acc = metric[0]/metric[2], metric[1]/metric[2]
            if (i+1) % 50 == 0:
                animator.add(epoch + i/len(train_iter),
                             (train_loss, train_acc, None))
        test_acc = d2l.evaluate_accuracy_gpu(net, test_iter)
        animator.add(epoch+1, (None, None, test_acc))
    print('train loss %.3f, train acc %.3f, test acc %.3f' % (
        train_loss, train_acc, test_acc))
```

Let's have a look at what happens if we invoke this algorithm with default settings, such as a learning rate of 0.5 and train for 40 iterations. Note how the training accuracy keeps on increasing while progress in terms of test accuracy stalls beyond a point. The gap between both curves indicates overfitting.

```
lr, num_epochs = 0.5, 40
net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
train(net, train_iter, test_iter, num_epochs, loss, trainer, ctx)
```

```
train loss 2.303, train acc 0.101, test acc 0.100
```



11.11.2 Schedulers

One way of adjusting the learning rate is to set it explicitly at each step. This is conveniently achieved by the `set_learning_rate` method. We could adjust it downward after every epoch (or even after every minibatch), e.g., in a dynamic manner in response to how optimization is progressing.

```
trainer.set_learning_rate(0.1)
print('Learning rate is now %.2f' % trainer.learning_rate)
```

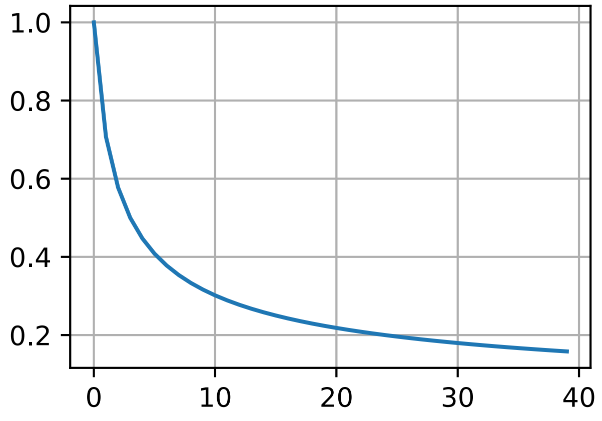
```
Learning rate is now 0.10
```

More generally we want to define a scheduler. When invoked with the number of updates it returns the appropriate value of the learning rate. Let's define a simple one that sets the learning rate to $\eta = \eta_0(t+1)^{-\frac{1}{2}}$.

```
class SquareRootScheduler(object):
    def __init__(self, lr=0.1):
        self.lr = lr

    def __call__(self, num_update):
        return self.lr * pow(num_update + 1.0, -0.5)
```

Let's plot its behavior over a range of values.

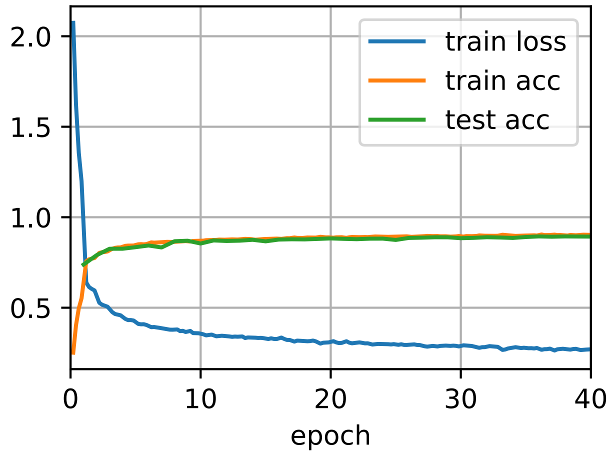
```
scheduler = SquareRootScheduler(lr=1.0)
d2l.plot(np.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```



Now let's see how this plays out for training on FashionMNIST. We simply provide the scheduler as an additional argument to the training algorithm.

```
trainer = gluon.Trainer(net.collect_params(), 'sgd',
                        {'lr_scheduler': scheduler})
train(net, train_iter, test_iter, num_epochs, loss, trainer, ctx)
```

```
train loss 0.271, train acc 0.902, test acc 0.893
```



This worked quite a bit better than previously. Two things stand out: the curve was rather more smooth than previously. Secondly, there was less overfitting. Unfortunately it is not a well-resolved question as to why certain strategies lead to less overfitting in *theory*. There is some argument that a smaller stepsize will lead to parameters that are closer to zero and thus simpler. However, this doesn't explain the phenomenon entirely since we don't really stop early but simply reduce the learning rate gently.

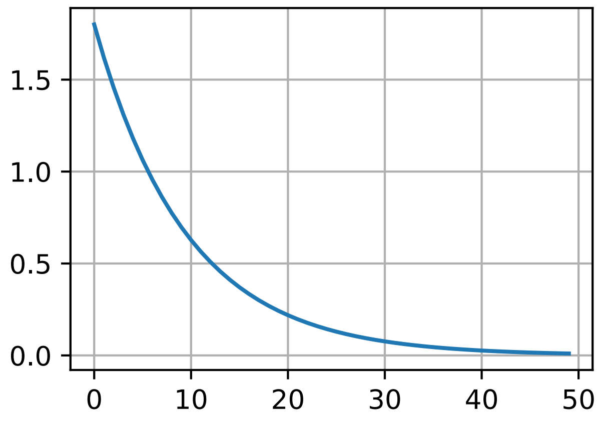
11.11.3 Policies

While we cannot possibly cover the entire variety of learning rate schedulers, we attempt to give a brief overview of popular policies below. Common choices are polynomial decay and piecewise constant schedules. Beyond that, cosine learning rate schedules have been found to work well empirically on some problems. Lastly, on some problems it is beneficial to warm up the optimizer prior to using large learning rates.

Factor Scheduler

One alternative to a polynomial decay would be a multiplicative one, that is $\eta_{t+1} \leftarrow \eta_t \cdot \alpha$ for $\alpha \in (0, 1)$. To prevent the learning rate from decaying beyond a reasonable lower bound the update equation is often modified to $\eta_{t+1} \leftarrow \max(\eta_{\min}, \eta_t \cdot \alpha)$.

```python
class FactorScheduler(object):
    def __init__(self, factor=1, stop_factor_lr=1e-7, base_lr=0.1):
        self.factor = factor
        self.stop_factor_lr = stop_factor_lr
        self.base_lr = base_lr

    def __call__(self, num_update):
        self.base_lr = max(self.stop_factor_lr, self.base_lr * self.factor)
        return self.base_lr

scheduler = FactorScheduler(factor=0.9, stop_factor_lr=1e-2, base_lr=2.0)
d2l.plot(np.arange(50), [scheduler(t) for t in range(50)])
```

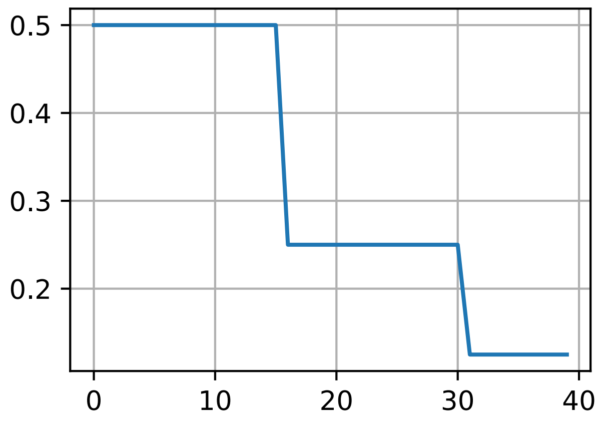


This can also be accomplished by a built-in scheduler in MXNet via the `lr_scheduler.FactorScheduler` object. It takes a few more parameters, such as warmup period, warmup mode (linear or constant), the maximum number of desired updates, etc.; Going forward we will use the built-in schedulers as appropriate and only explain their functionality here. As illustrated, it is fairly straightforward to build your own scheduler if needed.

Multi Factor Scheduler

A common strategy for training deep networks is to keep the learning rate piecewise constant and to decrease it by a given amount every so often. That is, given a set of times when to decrease the rate, such as $s = \{5, 10, 20\}$ decrease $\eta_{t+1} \leftarrow \eta_t \cdot \alpha$ whenever $t \in s$. Assuming that the values are halved at each step we can implement this as follows.

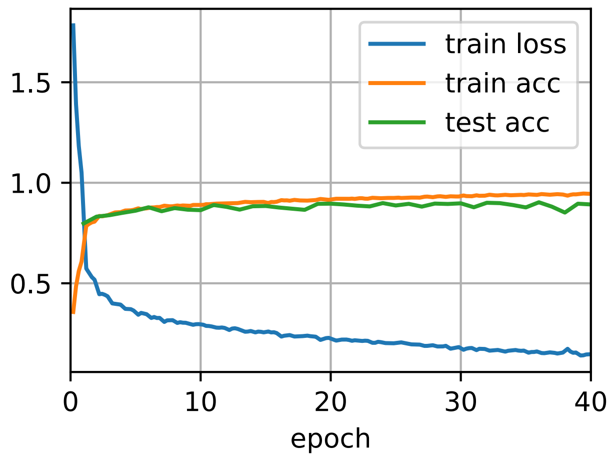
```
scheduler = lr_scheduler.MultiFactorScheduler(step=[15, 30], factor=0.5,
                                              base_lr=0.5)
d2l.plot(np.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```



The intuition behind this piecewise constant learning rate schedule is that one lets optimization proceed until a stationary point has been reached in terms of the distribution of weight vectors. Then (and only then) do we decrease the rate such as to obtain a higher quality proxy to a good local minimum. The example below shows how this can produce ever slightly better solutions.

```
trainer = gluon.Trainer(net.collect_params(), 'sgd',
                        {'lr_scheduler': scheduler})
train(net, train_iter, test_iter, num_epochs, loss, trainer, ctx)
```

```
train loss 0.149, train acc 0.945, test acc 0.892
```



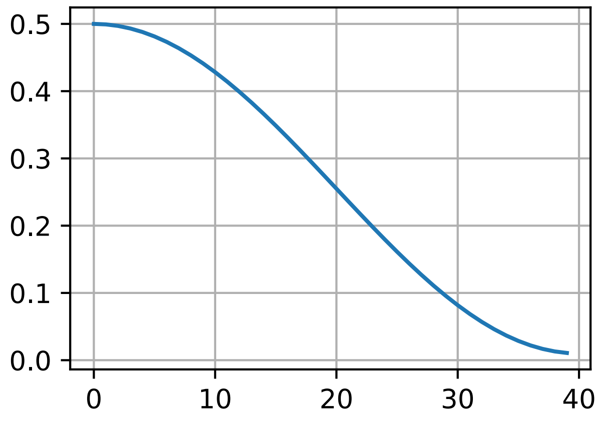
Cosine Scheduler

A rather perplexing heuristic was proposed by (Loshchilov & Hutter, 2016). It relies on the observation that we might not want to decrease the learning rate too drastically in the beginning and moreover, that we might want to "refine" the solution in the end using a very small learning rate. This results in a cosine-like schedule with the following functional form for learning rates in the range $t \in [0, T]$.

$$\eta_t = \eta_T + \frac{\eta_0 - \eta_T}{2} \left(1 + \cos(\pi t/T)\right) \tag{11.11.1}$$

Here η_0 is the initial learning rate, η_T is the target rate at time T. Furthermore, for $t > T$ we simply pin the value to η_T without increasing it again. In the following example, we set the max update step $T = 40$.

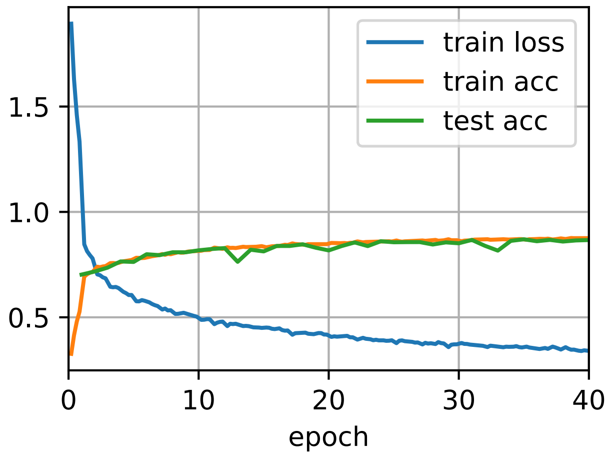
```
scheduler = lr_scheduler.CosineScheduler(max_update=40, base_lr=0.5,
                                         final_lr=0.01)
d2l.plot(np.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```



In the context of computer vision this schedule *can* lead to improved results. Note, though, that such improvements are not guaranteed (as can be seen below).

```
trainer = gluon.Trainer(net.collect_params(), 'sgd',
                        {'lr_scheduler': scheduler})
train(net, train_iter, test_iter, num_epochs, loss, trainer, ctx)
```

```
train loss 0.342, train acc 0.876, test acc 0.867
```

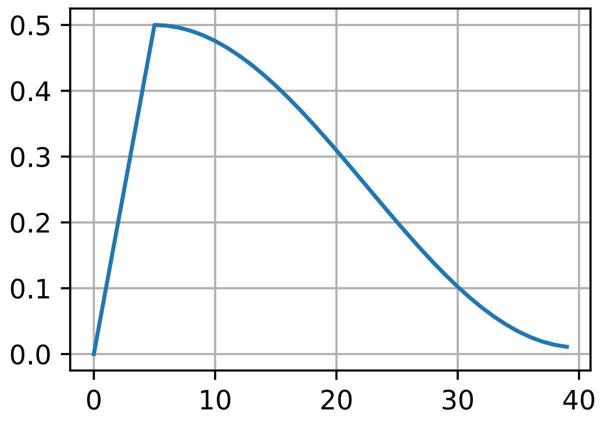


Warmup

In some cases initializing the parameters is not sufficient to guarantee a good solution. This particularly a problem for some advanced network designs that may lead to unstable optimization problems. We could address this by choosing a sufficiently small learning rate to prevent divergence in the beginning. Unfortunately this means that progress is slow. Conversely, a large learning rate initially leads to divergence.

A rather simple fix for this dilemma is to use a warmup period during which the learning rate *increases* to its initial maximum and to cool down the rate until the end of the optimization process. For simplicity one typically uses a linear increase for this purpose. This leads to a schedule of the form indicated below.

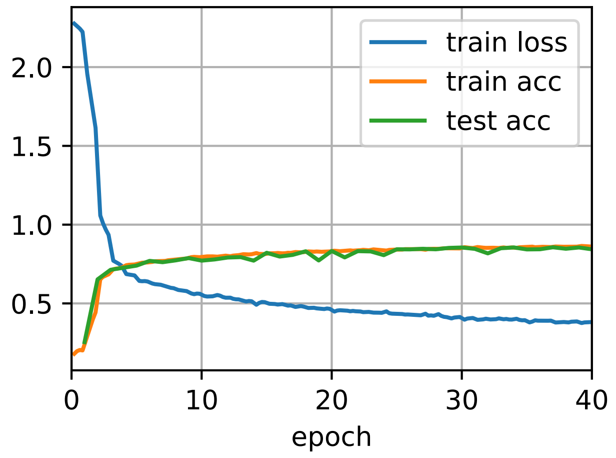
```
scheduler = lr_scheduler.CosineScheduler(40, warmup_steps=5, base_lr=0.5,
                                         final_lr=0.01)
d2l.plot(np.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```



Note that the network converges better initially (in particular observe the performance during the first 5 epochs).

```
trainer = gluon.Trainer(net.collect_params(), 'sgd',
                        {'lr_scheduler': scheduler})
train(net, train_iter, test_iter, num_epochs, loss, trainer, ctx)
```

```
train loss 0.380, train acc 0.862, test acc 0.843
```



Warmup can be applied to any scheduler (not just cosine). For a more detailed discussion of learning rate schedules and many more experiments see also (Gotmare et al., 2018). In particular they find that a warmup phase limits the amount of divergence of parameters in very deep networks. This makes intuitively sense since we would expect significant divergence due to random initialization in those parts of the network that take the most time to make progress in the beginning.

Summary

- Decreasing the learning rate during training can lead to improved accuracy and (most perplexingly) reduced overfitting of the model.
- A piecewise decrease of the learning rate whenever progress has plateaued is effective in practice. Essentially this ensures that we converge efficiently to a suitable solution and only then reduce the inherent variance of the parameters by reducing the learning rate.
- Cosine schedulers are popular for some computer vision problems. See e.g., GluonCV[165] for details of such a scheduler.
- A warmup period before optimization can prevent divergence.
- Optimization serves multiple purposes in deep learning. Besides minimizing the training objective, different choices of optimization algorithms and learning rate scheduling can lead to rather different amounts of generalization and overfitting on the test set (for the same amount of training error).

[165] http://gluon-cv.mxnet.io

Exercises

1. Experiment with the optimization behavior for a given fixed learning rate. What is the best model you can obtain this way?
2. How does convergence change if you change the exponent of the decrease in the learning rate? Use `PolyScheduler` for your convenience in the experiments.
3. Apply the cosine scheduler to large computer vision problems, e.g., training ImageNet. How does it affect performance relative to other schedulers?
4. How long should warmup last?
5. Can you connect optimization and sampling? Start by using results from (Welling & Teh, 2011) on Stochastic Gradient Langevin Dynamics.