学习笔记-6: Task 6

16 - 批量归一化和残差网络

批量归一化 (BatchNormalization)

对输入的标准化 (浅层模型)

处理后的任意一个特征在数据集中所有样本上的均值为0、标准差为1。
标准化处理输入数据使各个特征的分布相近。

批量归一化 (深度模型)

利用小批量上的均值和标准差，不断调整神经网络中间输出，从而使整个神经网络在各层的中间输出的数值更稳定。

1.对全连接层做批量归一化

位置：全连接层中的仿射变换和激活函数之间。

全连接：

$$\boldsymbol{x} = \boldsymbol{W}\boldsymbol{u} + \boldsymbol{b}$$

$$output = \phi(\boldsymbol{x})$$

批量归一化：

$$output = \phi(\text{BN}(\boldsymbol{x}))$$

$$\boldsymbol{y}^{(i)} = \text{BN}(\boldsymbol{x}^{(i)})$$

$$\boldsymbol{\mu}_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i = 1}^{m} \boldsymbol{x}^{(i)},$$

$$\boldsymbol{\sigma}_\mathcal{B}^2 \leftarrow \frac{1}{m} \sum_{i=1}^{m}(\boldsymbol{x}^{(i)} - \boldsymbol{\mu}_\mathcal{B})^2,$$

$$\hat{\boldsymbol{x}}^{(i)} \leftarrow \frac{\boldsymbol{x}^{(i)} - \boldsymbol{\mu}_\mathcal{B}}{\sqrt{\boldsymbol{\sigma}_\mathcal{B}^2 + \epsilon}},$$

这里$\epsilon > 0$是个很小的常数，保证分母大于0

$${\boldsymbol{y}}^{(i)} \leftarrow \boldsymbol{\gamma} \odot \hat{\boldsymbol{x}}^{(i)} + \boldsymbol{\beta}.$$

引入可学习参数：拉伸参数γ和偏移参数β。若$\boldsymbol{\gamma} = \sqrt{\boldsymbol{\sigma}_\mathcal{B}^2 + \epsilon}$和$\boldsymbol{\beta} = \boldsymbol{\mu}_\mathcal{B}$，批量归一化无效。

2.对卷积层做批量归一化

位置：卷积计算之后、应用激活函数之前。
如果卷积计算输出多个通道，我们需要对这些通道的输出分别做批量归一化，且每个通道都拥有独立的拉伸和偏移参数。
计算：对单通道，`batchsize=m`，卷积计算输出=`pxq`，对该通道中`m×p×q`个元素同时做批量归一化,使用相同的均值和方差。

3.预测时的批量归一化

训练：以`batch`为单位，对每个`batch`计算均值和方差。
预测：用移动平均估算整个训练数据集的样本均值和方差。

从零实现

```
def batch_norm(is_training, X, gamma, beta, moving_mean, moving_var, eps,
momentum):
    # 判断当前模式是训练模式还是预测模式
    if not is_training:
        # 如果是在预测模式下，直接使用传入的移动平均所得的均值和方差
        X_hat = (X - moving_mean) / torch.sqrt(moving_var + eps)
    else:
        assert len(X.shape) in (2, 4)
        if len(X.shape) == 2:
            # 使用全连接层的情况，计算特征维上的均值和方差
            mean = X.mean(dim=0)
            var = ((X - mean) ** 2).mean(dim=0)
        else:
            # 使用二维卷积层的情况，计算通道维上（axis=1）的均值和方差。这里我们需要保持
            # X的形状以便后面可以做广播运算
            mean = X.mean(dim=0, keepdim=True).mean(dim=2,
keepdim=True).mean(dim=3, keepdim=True)
            var = ((X - mean) ** 2).mean(dim=0, keepdim=True).mean(dim=2,
keepdim=True).mean(dim=3, keepdim=True)
        # 训练模式下用当前的均值和方差做标准化
        X_hat = (X - mean) / torch.sqrt(var + eps)
        # 更新移动平均的均值和方差
        moving_mean = momentum * moving_mean + (1.0 - momentum) * mean
        moving_var = momentum * moving_var + (1.0 - momentum) * var
    # 拉伸和偏移，这里的乘法是按元素乘，且使用到了广播机制
    Y = gamma * X_hat + beta
    return Y, moving_mean, moving_var
```

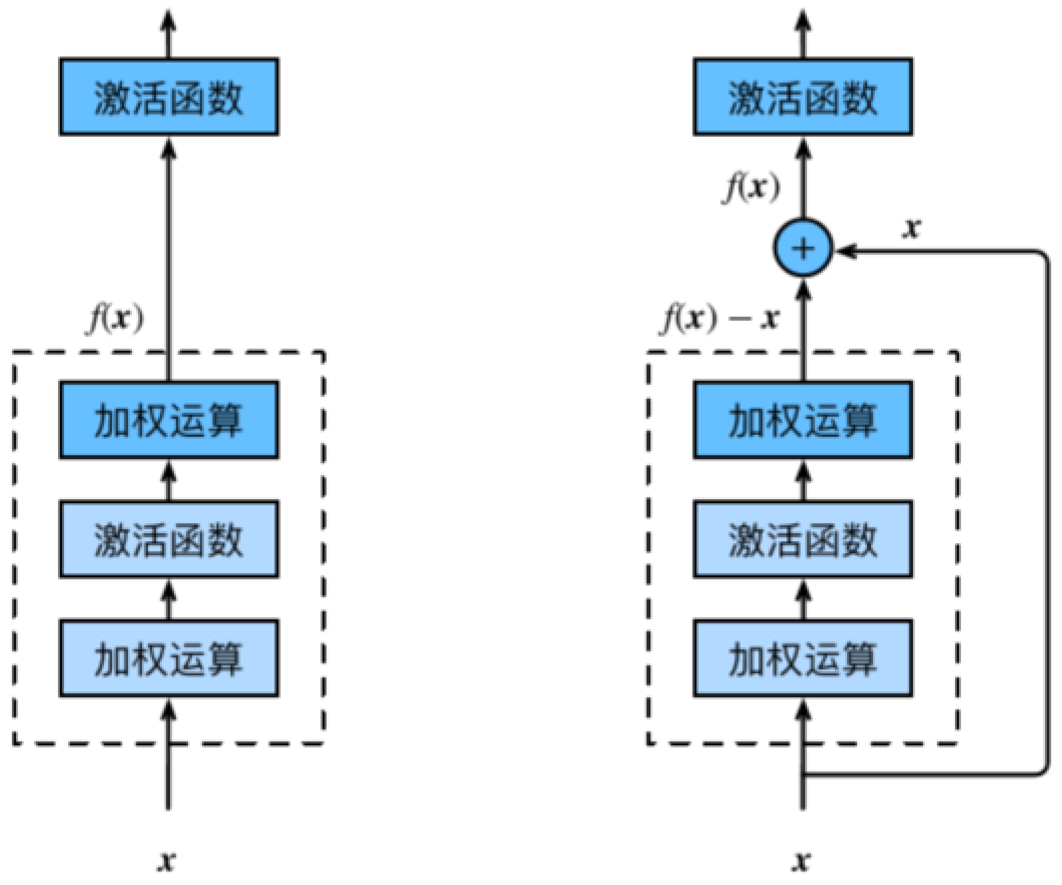
```
class BatchNorm(nn.Module):
    def __init__(self, num_features, num_dims):
        super(BatchNorm, self).__init__()
        if num_dims == 2:
            shape = (1, num_features) #全连接层输出神经元
        else:
            shape = (1, num_features, 1, 1)  #通道数
        # 参与求梯度和迭代的拉伸和偏移参数，分别初始化成0和1
        self.gamma = nn.Parameter(torch.ones(shape))
        self.beta = nn.Parameter(torch.zeros(shape))
        # 不参与求梯度和迭代的变量，全在内存上初始化成0
        self.moving_mean = torch.zeros(shape)
        self.moving_var = torch.zeros(shape)

    def forward(self, X):
        # 如果X不在内存上，将moving_mean和moving_var复制到X所在显存上
        if self.moving_mean.device != X.device:
            self.moving_mean = self.moving_mean.to(X.device)
            self.moving_var = self.moving_var.to(X.device)
        # 保存更新过的moving_mean和moving_var, Module实例的traning属性默认为true, 调
用.eval()后设成false
        Y, self.moving_mean, self.moving_var = batch_norm(self.training,
            X, self.gamma, self.beta, self.moving_mean,
            self.moving_var, eps=1e-5, momentum=0.9)
        return Y
```

残差网络 (ResNet)

深度学习的问题：深度CNN网络达到一定深度后再一味地增加层数并不能带来进一步地分类性能提高，反而会招致网络收敛变得更慢，准确率也变得更差。

残差块 (Residual Block)

恒等映射：
左边：`f(x) = x`
右边：`f(x)-x = 0` (易于捕捉恒等映射的细微波动)



在残差块中，输入可通过跨层的数据线路更快地向前传播。

```
class Residual(nn.Module):
    # 可以设定输出通道数、是否使用额外的1x1卷积层来修改通道数以及卷积层的步幅
    def __init__(self, in_channels, out_channels, use_1x1conv=False, stride=1):
        super(Residual, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3,
padding=1, stride=stride)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,
padding=1)
        if use_1x1conv:
            self.conv3 = nn.Conv2d(in_channels, out_channels, kernel_size=1,
stride=stride)
        else:
            self.conv3 = None
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.bn2 = nn.BatchNorm2d(out_channels)

    def forward(self, X):
        Y = F.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        return F.relu(Y + X)
```

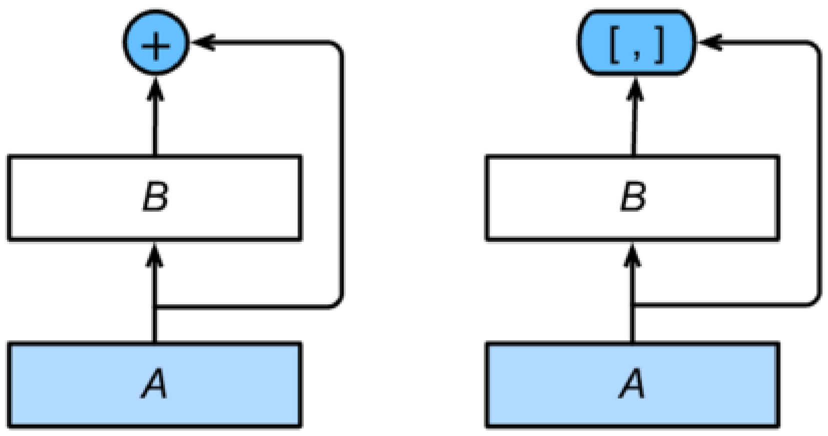
```
# 输入输出通道数相等，不需要额外的1×1卷积层
blk = Residual(3, 3)
X = torch.rand((4, 3, 6, 6))
blk(X).shape
>>> torch.Size([4, 3, 6, 6])
# 输入输出通道不等，需要额外的1×1卷积层
blk = Residual(3, 6, use_1x1conv=True, stride=2)
blk(X).shape
>>> torch.Size([4, 6, 3, 3])
```

ResNet模型

卷积(`64, 7x7, 3`)
批量一体化
最大池化(`3x3, 2`)
残差块x4(通过步幅为2的残差块在每个模块之间减小高和宽)
全局平均池化
全连接

```
net = nn.Sequential(
        nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3),
        nn.BatchNorm2d(64),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=3, stride=2, padding=1))

def resnet_block(in_channels, out_channels, num_residuals, first_block=False):
    if first_block:
        assert in_channels == out_channels # 第一个模块的通道数同输入通道数一致
    blk = []
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.append(Residual(in_channels, out_channels, use_1x1conv=True,
stride=2))
        else:
            blk.append(Residual(out_channels, out_channels))
    return nn.Sequential(*blk)

net.add_module("resnet_block1", resnet_block(64, 64, 2, first_block=True))
net.add_module("resnet_block2", resnet_block(64, 128, 2))
net.add_module("resnet_block3", resnet_block(128, 256, 2))
net.add_module("resnet_block4", resnet_block(256, 512, 2))

net.add_module("global_avg_pool", d2l.GlobalAvgPool2d())
# GlobalAvgPool2d的输出: (Batch, 512, 1, 1)
net.add_module("fc", nn.Sequential(d2l.FlattenLayer(), nn.Linear(512, 10)))
```

稠密连接网络 (DenseNet)



ResNet（左）与DenseNet（右）在跨层连接上的主要区别：使用相加和使用连结

主要构建模块

稠密块dense block： 定义了输入和输出是如何连结的
过渡层transition layer：用来控制通道数，使之不过大

稠密块

```
def conv_block(in_channels, out_channels):
    blk = nn.Sequential(nn.BatchNorm2d(in_channels),
                        nn.ReLU(),
                        nn.Conv2d(in_channels, out_channels, kernel_size=3,
padding=1))
    return blk

class DenseBlock(nn.Module):
    def __init__(self, num_convs, in_channels, out_channels):
        super(DenseBlock, self).__init__()
        net = []
        for i in range(num_convs):
            in_c = in_channels + i * out_channels
            net.append(conv_block(in_c, out_channels))
        self.net = nn.ModuleList(net)
        self.out_channels = in_channels + num_convs * out_channels # 计算输出通道数

    def forward(self, X):
        for blk in self.net:
            Y = blk(X)
            X = torch.cat((X, Y), dim=1)  # 在通道维上将输入和输出连结
        return X
```

过渡层

1×1卷积层：来减小通道数
步幅为2的平均池化层：减半高和宽

```
def transition_block(in_channels, out_channels):
    blk = nn.Sequential(
            nn.BatchNorm2d(in_channels),
            nn.ReLU(),
            nn.Conv2d(in_channels, out_channels, kernel_size=1),
            nn.AvgPool2d(kernel_size=2, stride=2))
    return blk
```

DenseNet模型

```
net = nn.Sequential(
        nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3),
        nn.BatchNorm2d(64),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=3, stride=2, padding=1))

num_channels, growth_rate = 64, 32  # num_channels为当前的通道数
num_convs_in_dense_blocks = [4, 4, 4, 4]

for i, num_convs in enumerate(num_convs_in_dense_blocks):
    DB = DenseBlock(num_convs, num_channels, growth_rate)
    net.add_module("DenseBlosk_%d" % i, DB)
    # 上一个稠密块的输出通道数
    num_channels = DB.out_channels
    # 在稠密块之间加入通道数减半的过渡层
    if i != len(num_convs_in_dense_blocks) - 1:
        net.add_module("transition_block_%d" % i, transition_block(num_channels,
num_channels // 2))
        num_channels = num_channels // 2

net.add_module("BN", nn.BatchNorm2d(num_channels))
net.add_module("relu", nn.ReLU())
net.add_module("global_avg_pool", d2l.GlobalAvgPool2d())
# GlobalAvgPool2d的输出: (Batch_size, num_channels, 1, 1)
net.add_module("fc", nn.Sequential(d2l.FlattenLayer(), nn.Linear(num_channels,
10)))
```

17 - 凸优化

优化与估计

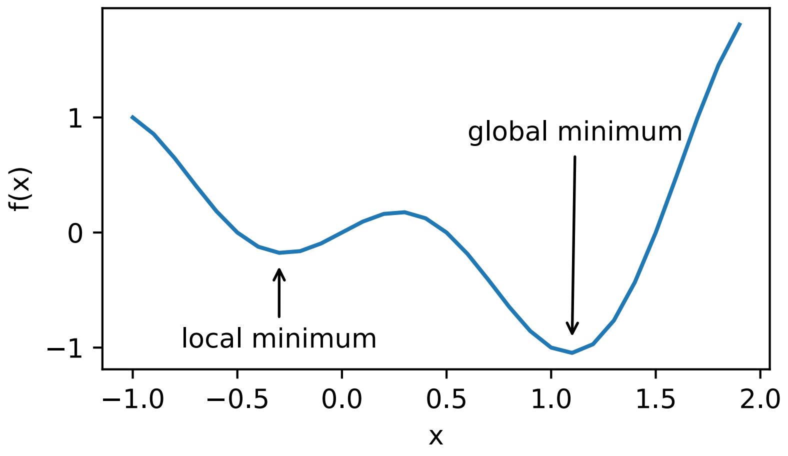
尽管优化方法可以最小化深度学习中的损失函数值，但本质上优化方法达到的目标与深度学习的目标并不相同。

- 优化方法目标：最小化**训练集**损失函数值
- 深度学习目标：最小化**测试集**损失函数值 (泛化性)

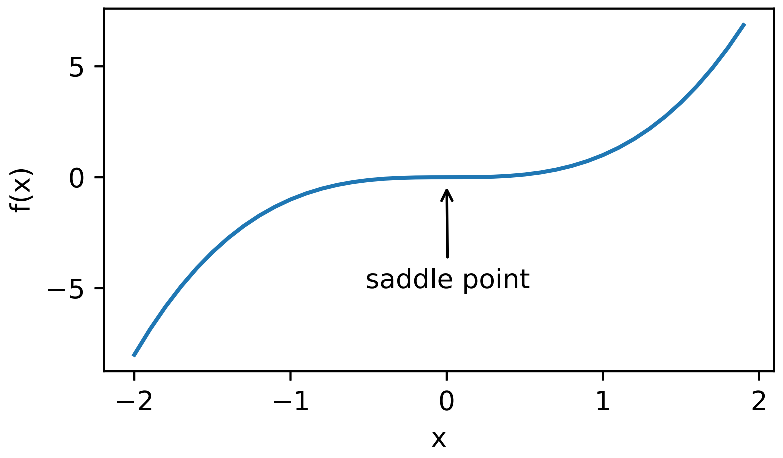
优化在深度学习中的挑战

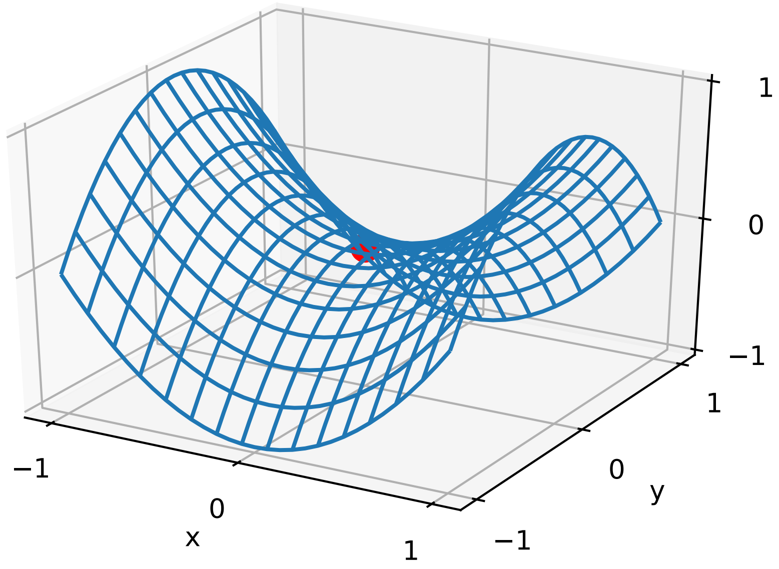
1.局部最小值

$$f(x) = x\cos\pi x$$

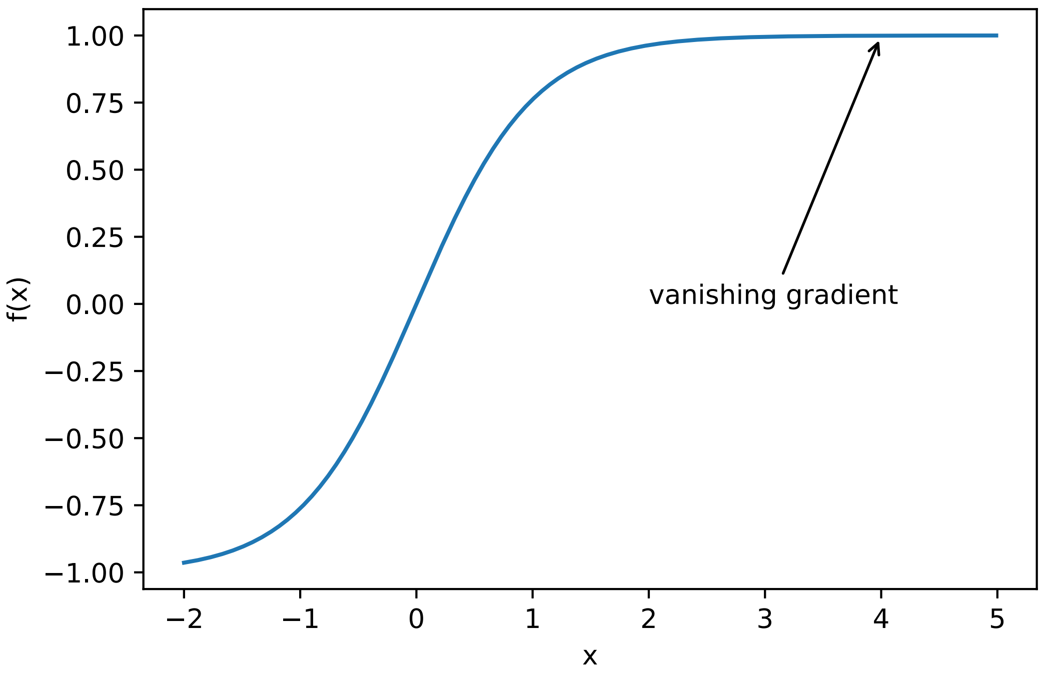


2.鞍点





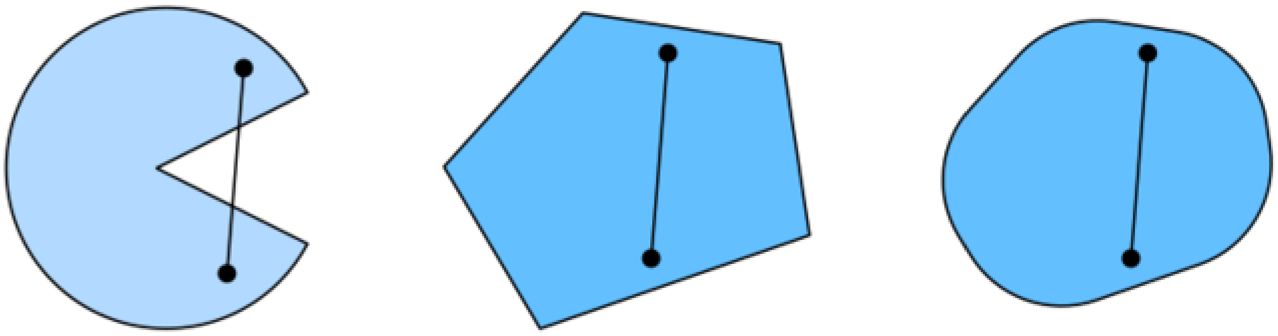
3.梯度消失



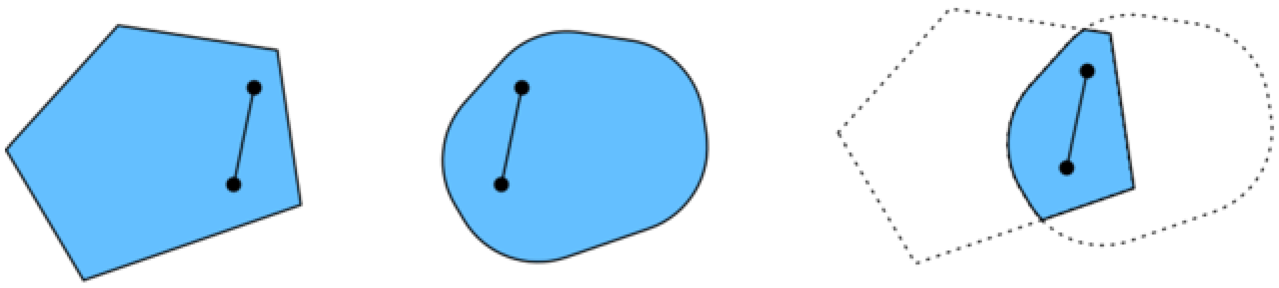
凸性 (Convexity)

集合

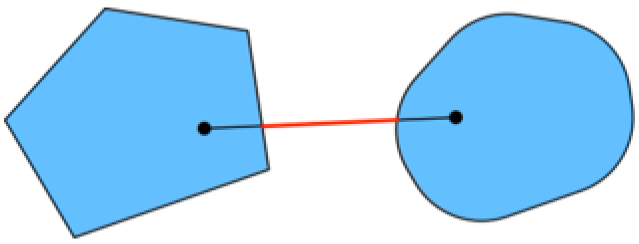
1. 一个集合内任意两点的连线都在该集合内部，这样的集合叫做凸集



2. 凸集的交一定是凸集

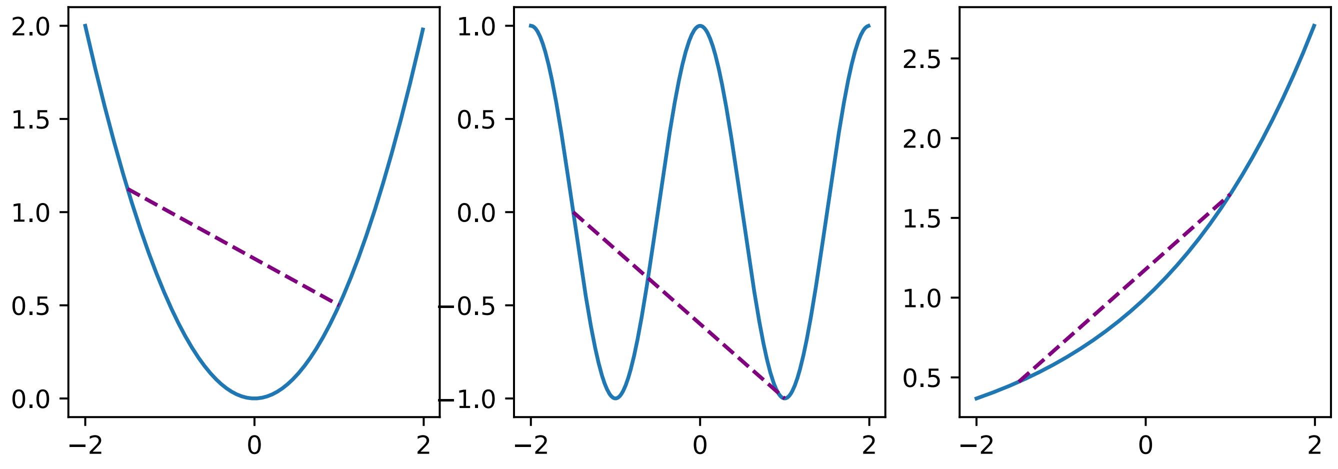


3. 凸集的并不一定是凸集



函数

$$\lambda f(x)+(1-\lambda) f\left(x^{\prime}\right) \geq f\left(\lambda x+(1-\lambda) x^{\prime}\right)$$



上图中间的函数不是凸函数，其余两个都是凸函数

Jensen 不等式

$$\sum_{i} \alpha_{i} f\left(x_{i}\right) \geq f\left(\sum_{i} \alpha_{i} x_{i}\right) \text { and } E_{x}[f(x)] \geq f\left(E_{x}[x]\right)$$

函数值的期望 $\geq$ 期望的函数值

性质

1.无局部极小值

证明：假设存在 $x \in X$ 是局部最小值，则存在全局最小值 $x' \in X$, 使得 $f(x)>f(x')$, 则对 $\lambda \in (0,1]$:

$$f(x)>\lambda f(x)+(1-\lambda) f(x') \geq f(\lambda x+(1-\lambda) x')$$

而这与 x 是局部极小值矛盾，因此凸函数没有局部极小值

2.与凸集的关系

对于凸函数 $f(x)$，定义集合 $S_b := x | x \in X \text{ and } f(x) \leq b$，则集合 S_b 为凸集

证明：对于点 $x, x' \in S_b$, 有 $f\left(\lambda x+(1-\lambda) x^{\prime}\right) \leq \lambda f(x)+(1-\lambda) f\left(x^{\prime}\right) \leq b$, 故 $\lambda x+(1-\lambda) x^{\prime} \in S_b$

3.凸函数与二阶导数

$f^{''}(x) \geq 0 \Longleftrightarrow f(x)$ 是凸函数

必要性 ($\Leftarrow$):

对于凸函数：

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

故:

$$f''(x) = \lim_{\varepsilon \to 0} \frac{\frac{f(x+\epsilon)-f(x)}{\epsilon} - \frac{f(x)-f(x-\epsilon)}{\epsilon}}{\epsilon}$$

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

充分性 (⇒):

令 $a < x < b$ 为 $f(x)$ 上的三个点，由拉格朗日中值定理:

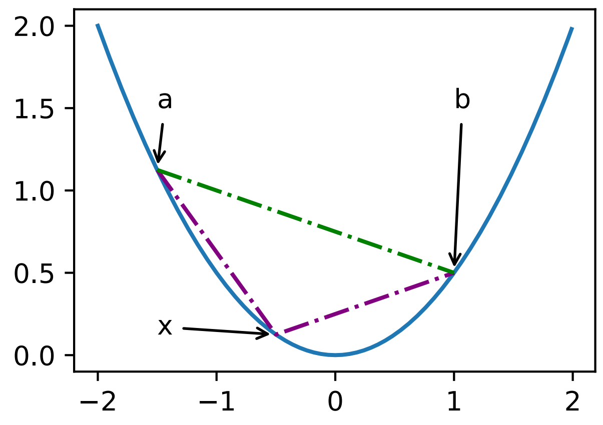
$$f(x) - f(a) = (x-a)f'(\alpha) \quad \text{for some } \alpha \in [a, x]$$

$$f(b) - f(x) = (b-x)f'(\beta) \quad \text{for some } \beta \in [x, b]$$

根据单调性，有 $f'(\beta) \geq f'(\alpha)$, 故:

$$f(b) - f(a) = f(b) - f(x) + f(x) - f(a)$$

$$= (b-x)f'(\beta) + (x-a)f'(\alpha) \geq (b-a)f'(\alpha)$$



限制条件

$$\underset{\mathbf{x}}{\text{minimize}}\, f(\mathbf{x}) \qquad \text{subject to} \quad c_i(\mathbf{x}) \leq 0 \text{ for all } i \in 1, \ldots, N$$

拉格朗日乘子法

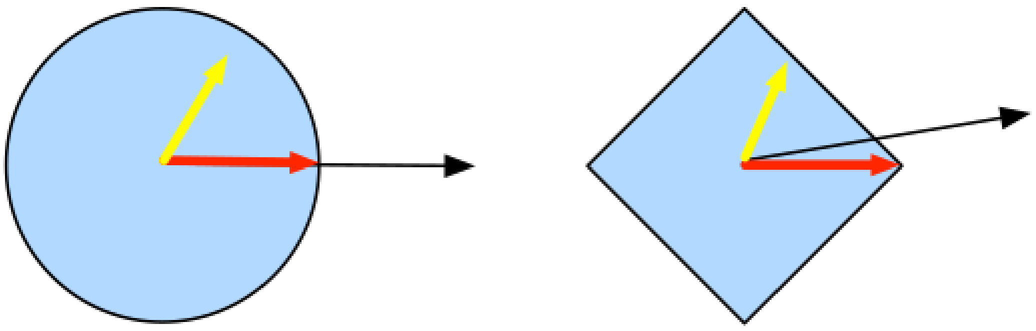
Boyd & Vandenberghe, 2004

惩罚项

欲使 $c_i(x) \leq 0$, 将项 $\alpha_i c_i(x)$ 加入目标函数，如多层感知机中的 $\frac{\lambda}{2}||w||^2$

投影

$$\text{Proj}_X(\mathbf{x}) = \underset{\mathbf{x}' \in X}{\text{argmin}}\, ||\mathbf{x} - \mathbf{x}'||_2$$



18 - 梯度下降

Boyd & Vandenberghe, 2004

一维梯度下降

证明：沿梯度反方向移动自变量可以减小函数值

泰勒展开：

$$f(x+\epsilon)=f(x)+\epsilon f^{\prime}(x)+\mathcal{O}\left(\epsilon^{2}\right)$$

代入沿梯度方向的移动量 $\eta f^{\prime}(x)$：

$$f\left(x-\eta f^{\prime}(x)\right)=f(x)-\eta f^{\prime 2}(x)+\mathcal{O}\left(\eta^{2} f^{\prime 2}(x)\right)$$

当 η 取的足够小时，可以使 $\eta f^{\prime 2}(x)>\mathcal{O}\left(\eta^{2} f^{\prime 2}(x)\right)$

因此有

$$f\left(x-\eta f^{\prime}(x)\right) \lesssim f(x)$$

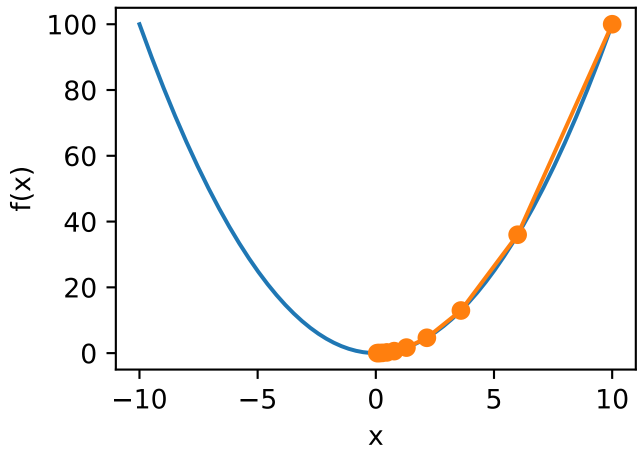
$$x \leftarrow x-\eta f^{\prime}(x)$$

一个例子

$$f(x)=x^{2}$$

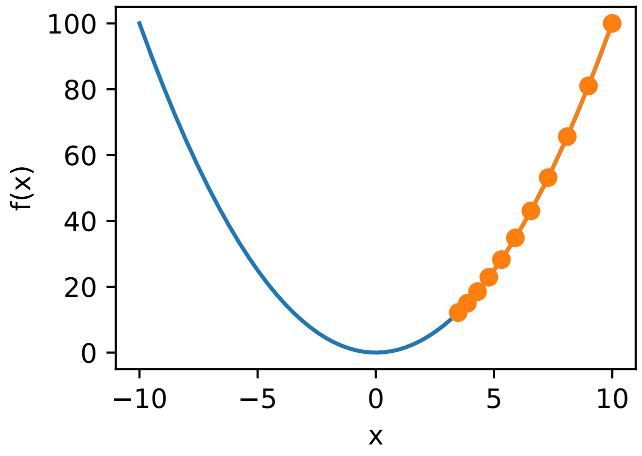
```
def f(x):
    return x**2  # Objective function

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

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
```

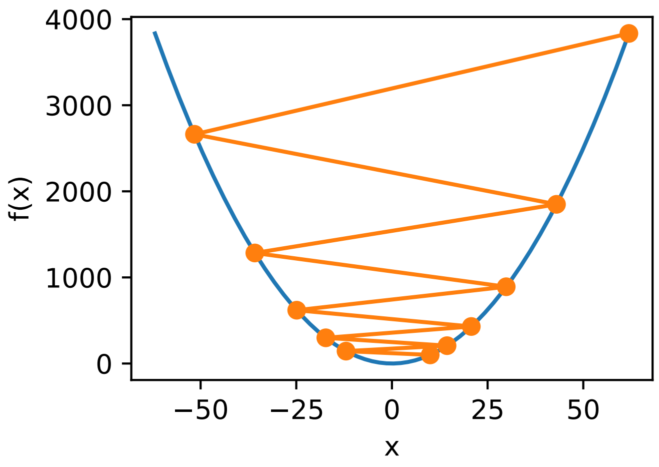


学习率

当学习率**=0.05**时



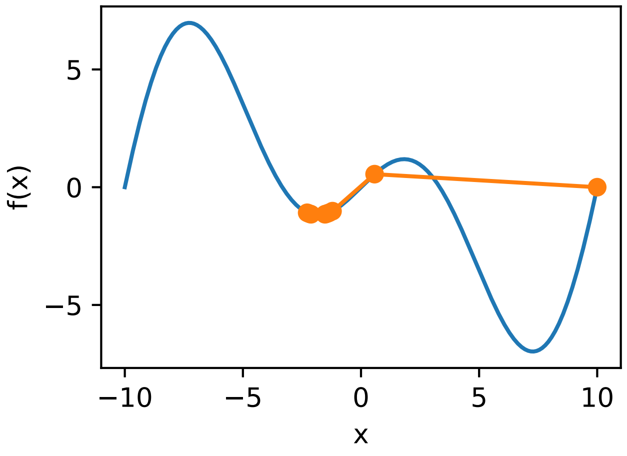
当学习率**=1.1**时



局部极小值

e.g.

$$f(x) = x\cos cx$$



多维梯度下降

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

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

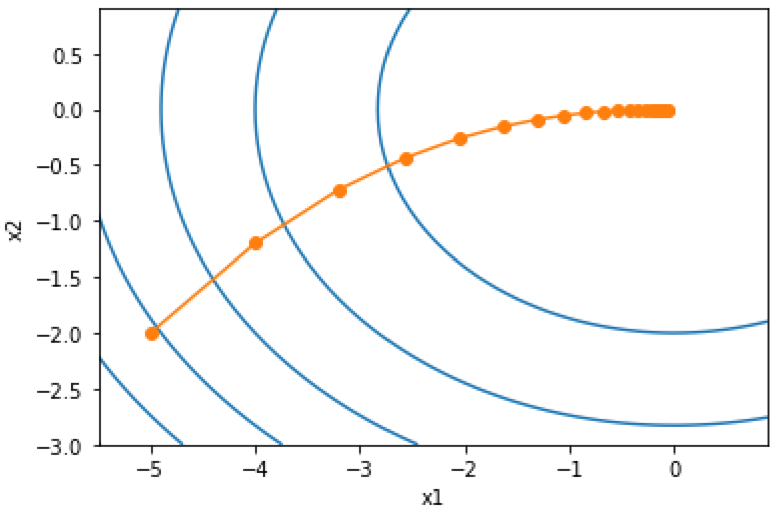
$$\mathbf{x} \leftarrow \mathbf{x} - \eta\nabla f(\mathbf{x})$$

一个二维梯度下降的例子

```
def train_2d(trainer, steps=20):
    x1, x2 = -5, -2
    results = [(x1, x2)]
    for i in range(steps):
        x1, x2 = trainer(x1, x2)
        results.append((x1, x2))
    print('epoch %d, x1 %f, x2 %f' % (i + 1, x1, x2))
    return results

# 展示梯度下降的过程
def show_trace_2d(f, results):
    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')
```

```
eta = 0.1   # 学习率

def f_2d(x1, x2):  # 目标函数
    return x1 ** 2 + 2 * x2 ** 2
```

```
def gd_2d(x1, x2):
    return (x1 - eta * 2 * x1, x2 - eta * 4 * x2)

show_trace_2d(f_2d, train_2d(gd_2d))
>>> epoch 20, x1 -0.057646, x2 -0.000073
```



自适应方法

牛顿法

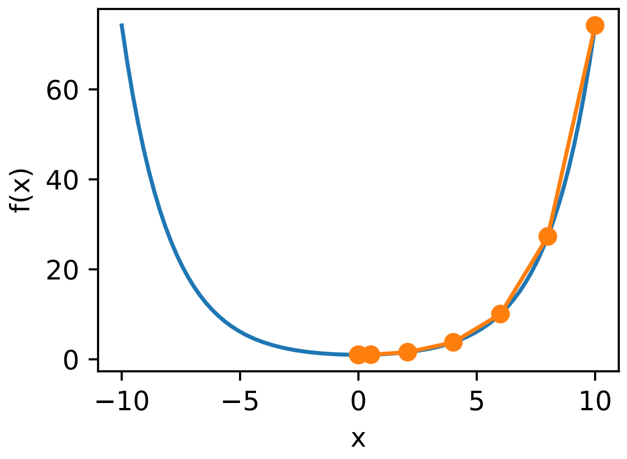
在 $x+\epsilon$ 处泰勒展开：

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

最小值点处满足: $\nabla f(\mathbf{x})=0$, 即希望 $\nabla f(\mathbf{x}+\epsilon)=0$, 对上式关于 ϵ 求导，忽略高阶无穷小，有：

$$\nabla f(\mathbf{x})+\boldsymbol{H}_f\boldsymbol{\epsilon}=0 \quad \text{and hence } \epsilon=-\boldsymbol{H}_f^{-1}\nabla f(\mathbf{x})$$

```
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
```



收敛性分析

只考虑在函数为凸函数, 且最小值点上 $f''(x_{min})>0$ 时的收敛速度：

令x_k为第k次迭代后x的值，$e_k=x_k-x_{min}$ 表示x_k到最小值点x_{min}的距离，由$f'(x_{min})=0$

$$0=f'\left(x_k-e_k\right)=f'\left(x_k\right)-e_k f''\left(x_k\right)+\frac{1}{2}e_k^2 f'''\left(\xi_k\right)$$

$$\text{for some } \xi_k\in\left[x_k-e_k,x_k\right]$$

两边除以 $f''(x_k)$, 有：

$$e_k-f'\left(x_k\right)/f''\left(x_k\right)=\frac{1}{2}e_k^2 f'''\left(\xi_k\right)/f''\left(x_k\right)$$

代入更新方程 $x_{k+1}=x_k-f'\left(x_k\right)/f''\left(x_k\right)$, 得到：

$$x_k-x_{min}-f'\left(x_k\right)/f''\left(x_k\right)=\frac{1}{2}e_k^2 f'''\left(\xi_k\right)/f''\left(x_k\right)$$

$$x_{k+1}-x_{min}=e_{k+1}=\frac{1}{2}e_k^2 f'''\left(\xi_k\right)/f''\left(x_k\right)$$

当 $\frac{1}{2}f'''\left(\xi_k\right)/f''\left(x_k\right)\leq c$ 时，有:

$$e_{k+1}\leq ce_k^2$$

预处理 (Heissan矩阵辅助梯度下降)

$$\mathbf{x}\leftarrow\mathbf{x}-\eta\operatorname{diag}\left(H_f\right)^{-1}\nabla\mathbf{x}$$

随机梯度下降

随机梯度下降参数更新

对于有 n 个样本的训练数据集，设 $f_i(x)$ 是第 i 个样本的损失函数, 则目标函数为:

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

其梯度为:

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

使用该梯度的一次更新的时间复杂度为 $\mathcal{O}(n)$

随机梯度下降更新公式 $\mathcal{O}(1)$:

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

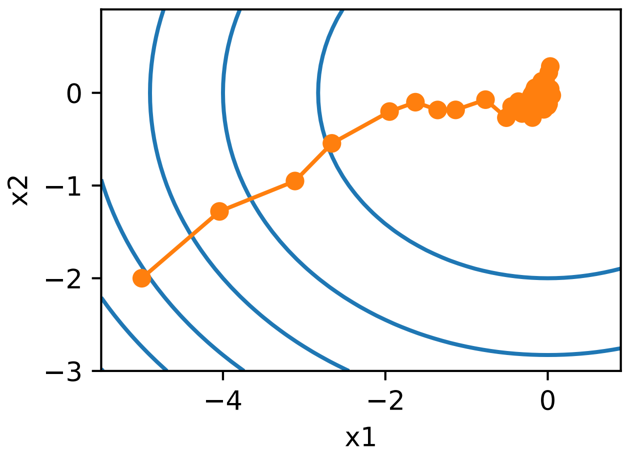
且有：

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

e.g.

$$f(x_1,x_2)=x_1^2+2x_2^2$$

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

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

def sgd(x1, x2):  # 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)  # Update variables

eta = 0.1
lr = (lambda: 1)  # Constant learning rate
show_trace_2d(f, train_2d(sgd, steps=50))
>>> epoch 50, x1 -0.027566, x2 0.137605
```



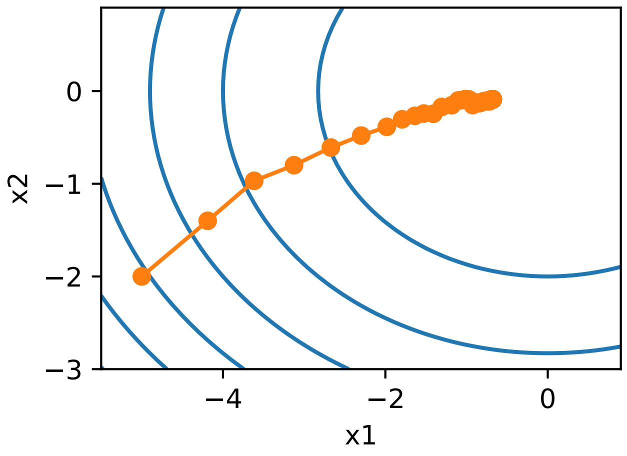
动态学习率

$$\eta(t) = \eta_i \quad \text{if } t_i \leq t \leq t_{i+1} \quad \text{piecewise constant}$$

$$\eta(t) = \eta_0 \cdot e^{-\lambda t} \quad \text{exponential}$$

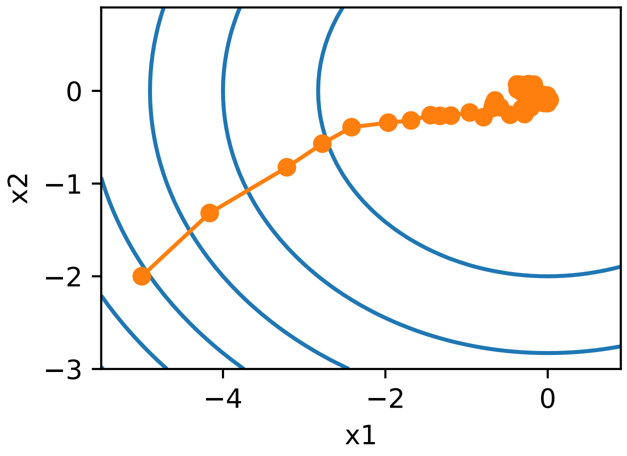
$$\eta(t) = \eta_0 \cdot (\beta t + 1)^{-\alpha} \quad \text{polynomial}$$

```
def exponential():
    global ctr  # 迭代次数t
    ctr += 1
    return math.exp(-0.1 * ctr)

ctr = 1
lr = exponential  # Set up learning rate
show_trace_2d(f, train_2d(sgd, steps=1000))
>>> epoch 1000, x1 -0.677947, x2 -0.089379
```



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

```
ctr = 1
lr = polynomial  # Set up learning rate
show_trace_2d(f, train_2d(sgd, steps=50))
>>> epoch 50, x1 -0.095244, x2 -0.041674
```



小批量随机梯度下降

读取数据

从零开始实现

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

def train_ch7(optimizer_fn, states, hyperparams, features, labels,
              batch_size=10, num_epochs=2):
    # 初始化模型
    net, loss = d2l.linreg, d2l.squared_loss

    w = torch.nn.Parameter(torch.tensor(np.random.normal(0, 0.01, size=
(features.shape[1], 1)), dtype=torch.float32),
                           requires_grad=True)
    b = torch.nn.Parameter(torch.zeros(1, dtype=torch.float32),
requires_grad=True)

    def eval_loss():
        return loss(net(features, w, b), labels).mean().item()

    ls = [eval_loss()]
    data_iter = torch.utils.data.DataLoader(
        torch.utils.data.TensorDataset(features, labels), batch_size,
shuffle=True)

    for _ in range(num_epochs):
        start = time.time()
        for batch_i, (X, y) in enumerate(data_iter):
            l = loss(net(X, w, b), y).mean()  # 使用平均损失
            # 梯度清零
            if w.grad is not None:
                w.grad.data.zero_()
                b.grad.data.zero_()
            l.backward()
            optimizer_fn([w, b], states, hyperparams)  # 迭代模型参数
            if (batch_i + 1) * batch_size % 100 == 0:
                ls.append(eval_loss())  # 每100个样本记录下当前训练误差
    # 打印结果和作图
    print('loss: %f, %f sec per epoch' % (ls[-1], time.time() - start))
    d2l.set_figsize()
    d2l.plt.plot(np.linspace(0, num_epochs, len(ls)), ls)
    d2l.plt.xlabel('epoch')
    d2l.plt.ylabel('loss')

def train_sgd(lr, batch_size, num_epochs=2):
    train_ch7(sgd, None, {'lr': lr}, features, labels, batch_size, num_epochs)
```

简洁实现

```
def train_pytorch_ch7(optimizer_fn, optimizer_hyperparams, features, labels,
                    batch_size=10, num_epochs=2):
    # 初始化模型
    net = nn.Sequential(
        nn.Linear(features.shape[-1], 1)
    )
    loss = nn.MSELoss()
    optimizer = optimizer_fn(net.parameters(), **optimizer_hyperparams)

    def eval_loss():
        return loss(net(features).view(-1), labels).item() / 2

    ls = [eval_loss()]
    data_iter = torch.utils.data.DataLoader(
        torch.utils.data.TensorDataset(features, labels), batch_size,
shuffle=True)

    for _ in range(num_epochs):
        start = time.time()
        for batch_i, (X, y) in enumerate(data_iter):
            # 除以2是为了和train_ch7保持一致, 因为squared_loss中除了2
            l = loss(net(X).view(-1), y) / 2

            optimizer.zero_grad()
            l.backward()
            optimizer.step()
            if (batch_i + 1) * batch_size % 100 == 0:
                ls.append(eval_loss())
    # 打印结果和作图
    print('loss: %f, %f sec per epoch' % (ls[-1], time.time() - start))
    d2l.set_figsize()
    d2l.plt.plot(np.linspace(0, num_epochs, len(ls)), ls)
    d2l.plt.xlabel('epoch')
    d2l.plt.ylabel('loss')
```