深度学习中的优化算法研究

内容概要

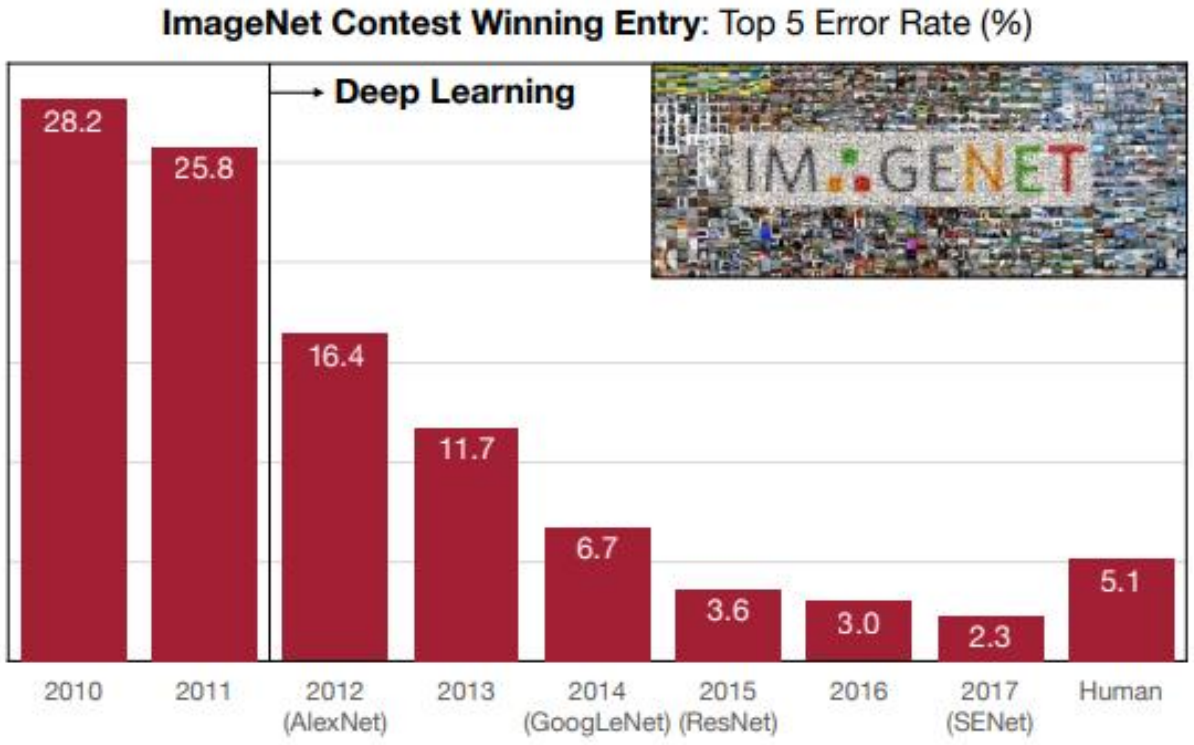
- 深度学习的应用
- 深度学习中的优化挑战
- 深度学习的各种优化算法
- 各类优化算法的特点

深度学习无处不在

深度学习在各个领域都有应用

深度学习在图像分类的应用

深度神经网络在ImageNet上实现了超越人类的分类准确度



ImageNet: https://image-net.org/

深度学习在图像分类的应用

深度学习在日常生活的应用

图像分类

图像标签

手机上的图像识别(Image Recognition)

深度学习在手机上的应用极大的简化了用户操作

iPone上的人像识别

Recognizing People in Photos Through Private On-Device Machine Learning [Apple, 2021]

深度学习在图像生成上的应用

扩散模型(Diffusion models)通过自然语言描述生成图片

荒凉星球上的猛犸象

一个宇航员骑着马在火星

湖边两只小狗在玩耍

豆包:https://www.doubao.com/

移动设备端上的图像生成(Image Generation)

Source:https://www.apple.com/apple-intelligence/

深度学习无处不在

深度学习在各个领域都有应用

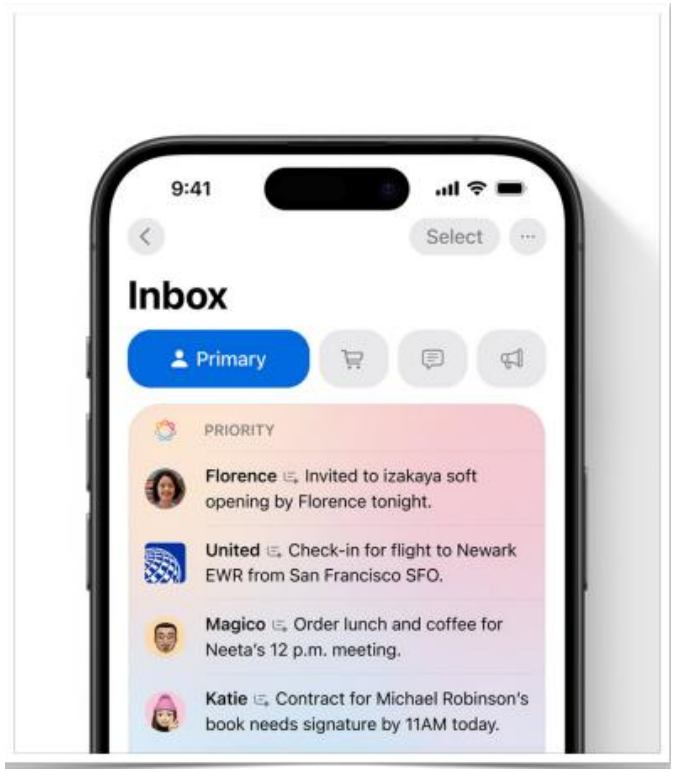
ChatGPT 和大语言模型(LLM)

LLMs是一个高效的查询工具

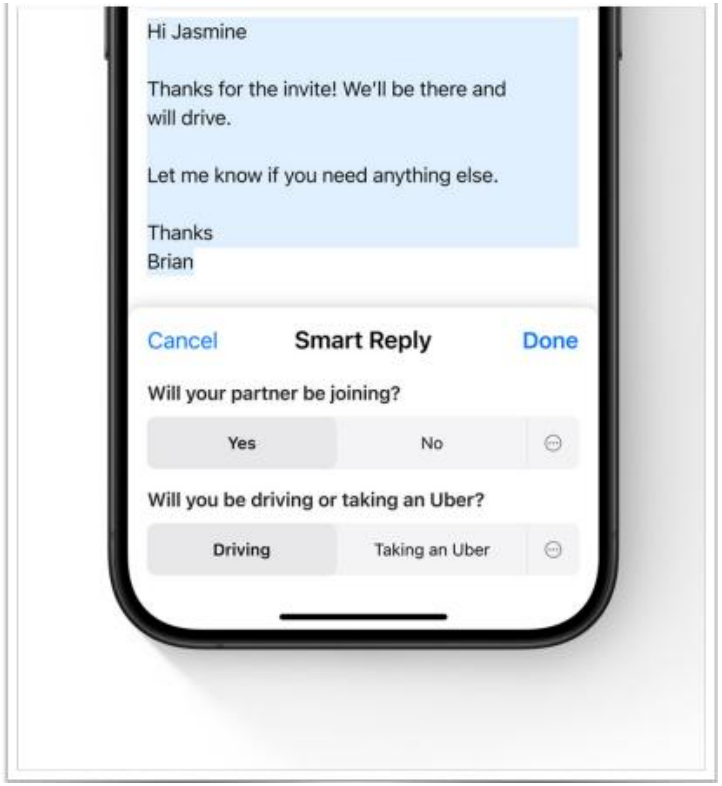
ChatGPT: https://chat.openai.com/

LLM在手机上的应用

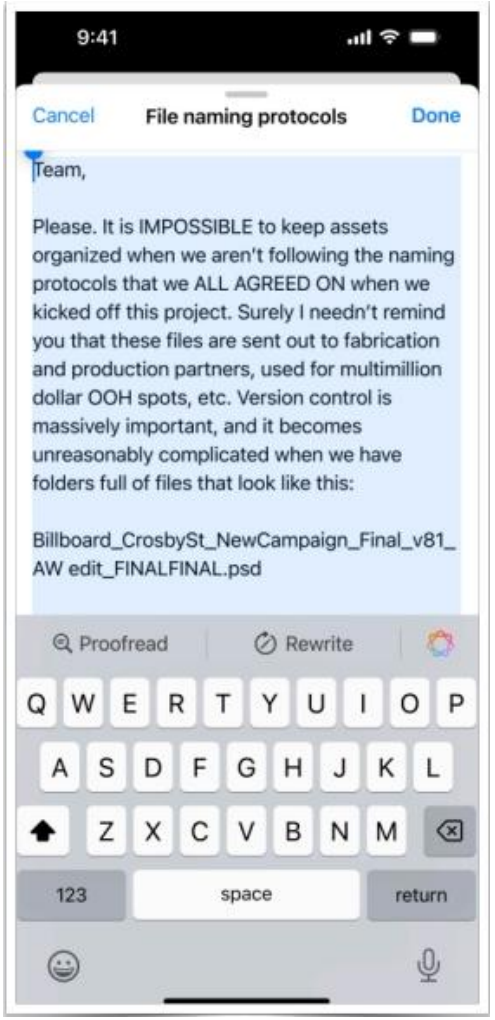
在移动设备端LLM极大的丰富了写作工具



总结摘要



智能问答



书写工具

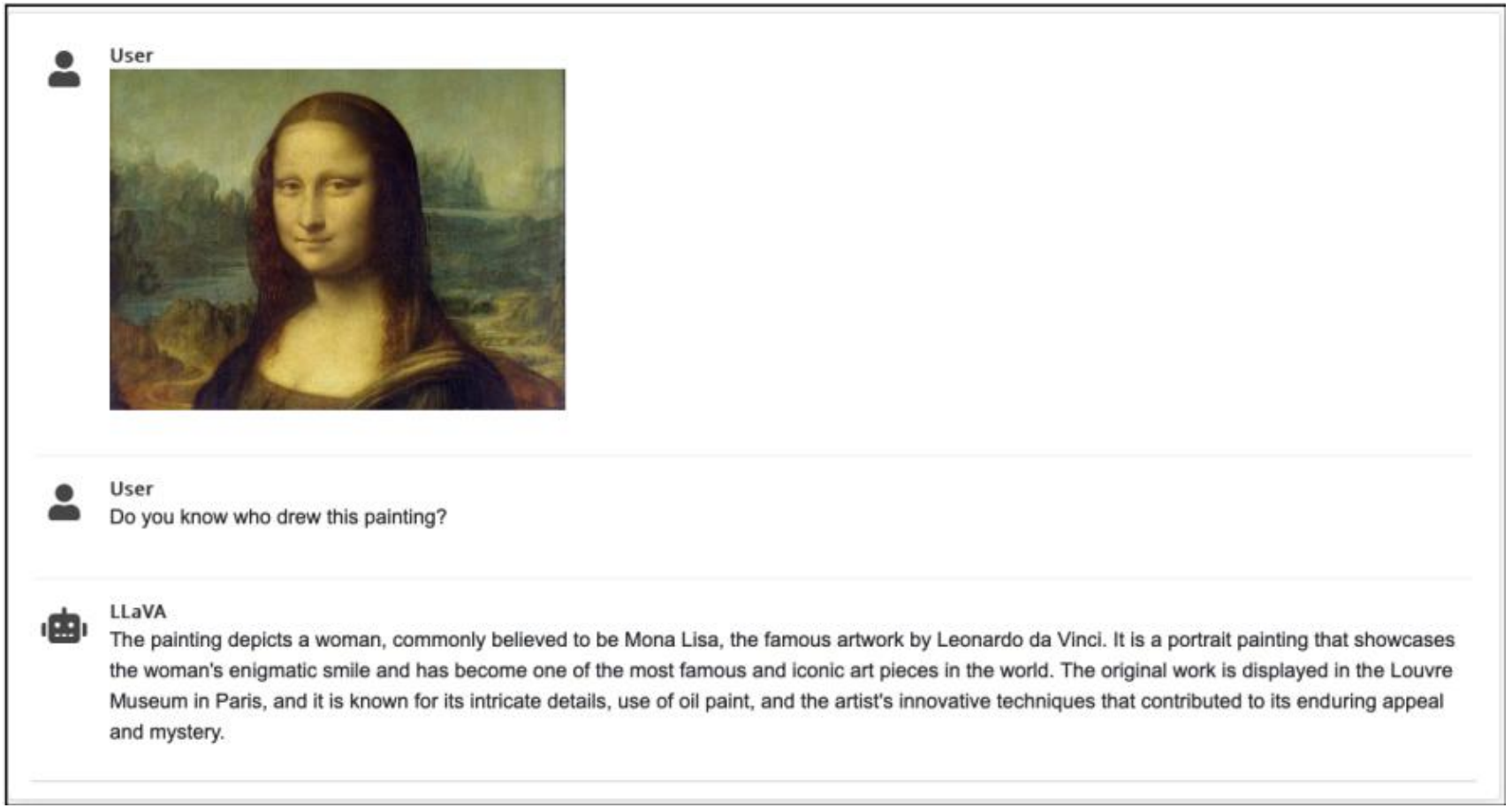
Source:https://www.apple.com/apple-intelligence/

深度学习无处不在

深度学习在各个领域都有应用

视觉语言模型(Vision-Language Models)

LLaVA实现了一般的视觉和语言的理解



LLaVA: https://llava-vl.github.io/

深度学习在其他领域的应用

AlphaGo使用深度神经网络&树搜索成为围棋领域的“大师”

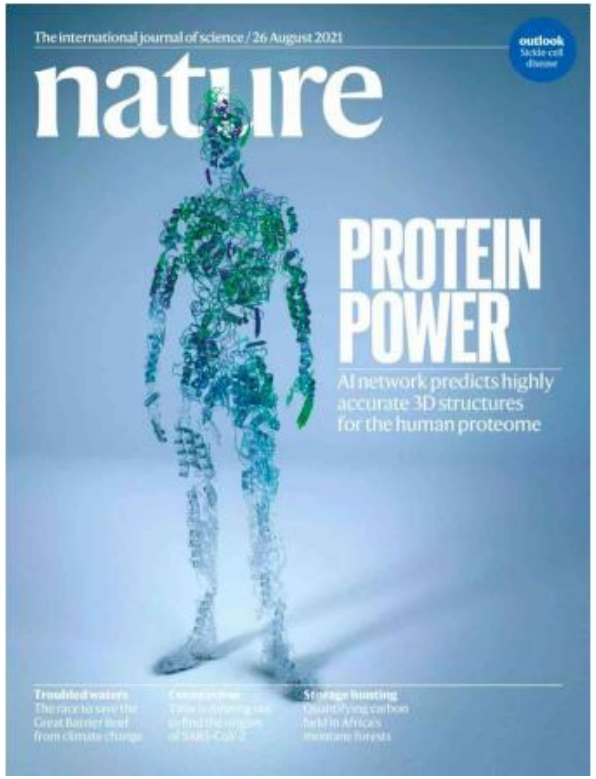
AlphaGo(Nature 2016)

AlphaGo vs 李世石(4-1)

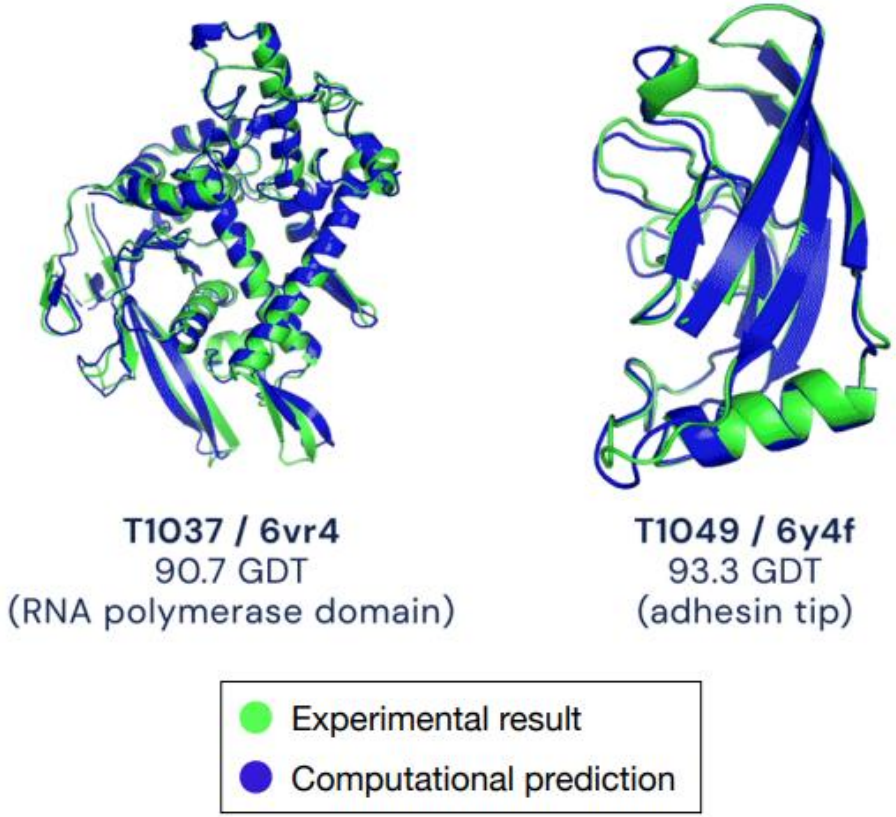
AlphaGo: https://www.deepmind.com/research/highlighted-research/alphago

深度学习在科学研究上的应用

AlphaFold揭示了蛋白质的结构



AlphaFold(Nature 2021)



AlphaFold 2: https://www.deepmind.com/blog/alphafold-a-solution-to-a-50-year-old-grand-challenge-in-biology

深度学习中的优化算法

优化和深度学习

在深度学习中，我们通常使用梯度下降算法来优化模型参数。梯度下降算法是一种迭代优化算法，它通过计算目标函数的梯度来更新模型参数，以最小化目标函数。在深度学习中，目标函数通常是模型的损失函数，用于衡量模型预测结果与真实标签之间的差异。对于深度学习问题，我们通常会先定义损失函数。一旦我们有了损失函数，我们就可以使用优化算法来尝试最小化损失。

深度学习中的优化挑战

局部最小值

对于任何目标函数$f(x)$，如果在x处对应的$f(x)$值小于在附近任意其他点的$f(x)$值，那么$f(x)$可能是局部最小值。如果$f(x)$在x处的值是整个域中目标函数的最小值，那么$f(x)$是全局最小值。 例如，给定函数

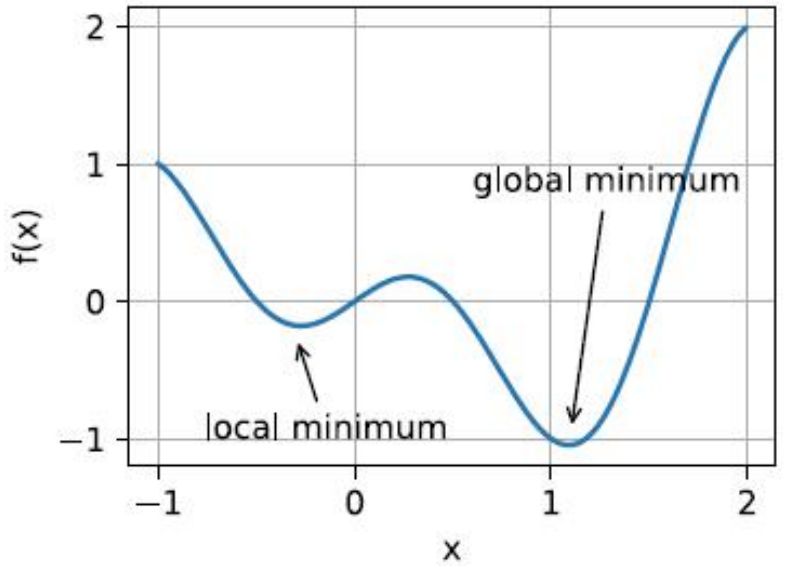
$$f(x) = x * cos(\pi x) \quad for\ -1.0 \leq x \leq 2.0$$

我们可以近似该函数的局部最小值和全局最小值。

深度学习中的优化挑战

局部最小值

```
def f(x):
    return x * torch.cos(np.pi * x)

def annotate(text, xy, xytext):  #@save
    d2l.plt.gca().annotate(text, xy=xy, xytext=xytext,
                           arrowprops=dict(arrowstyle='->'))


x = torch.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))
```

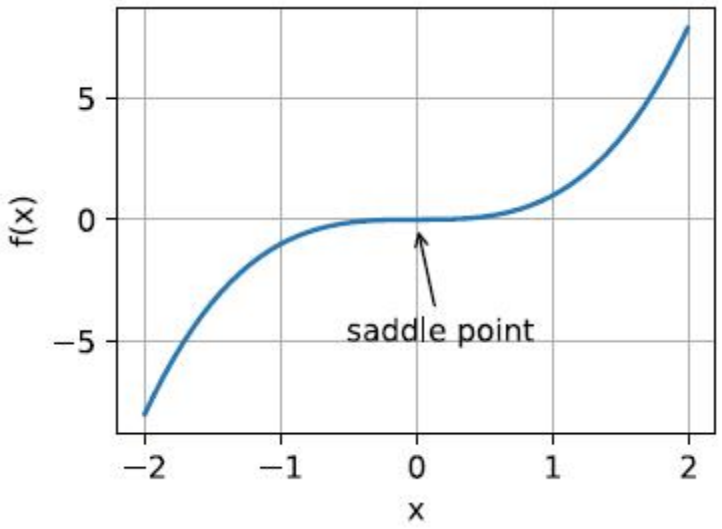


深度学习模型的目标函数通常有许多局部最优解。当优化问题的数值解接近局部最优值时，随着目标函数解的梯度接近或变为零，通过最终迭代获得的数值解可能仅使目标函数局部最优，而不是全局最优。只有一定程度的噪声可能会使参数跳出局部最小值。事实上，这是小批量随机梯度下降的有利特性之一。在这种情况下，小批量上梯度的自然变化能够将参数从局部极小值中跳出。

深度学习中的优化挑战

鞍点

除了局部最小值之外，鞍点是梯度消失的另一个原因。鞍点（saddle point）是指函数的所有梯度都消失但既不是全局最小值也不是局部最小值的任何位置。考虑函数$f(x) = x\text{^}3$。它的一阶和二阶导数在时消失。这时优化可能会停止，但它不是最小值。

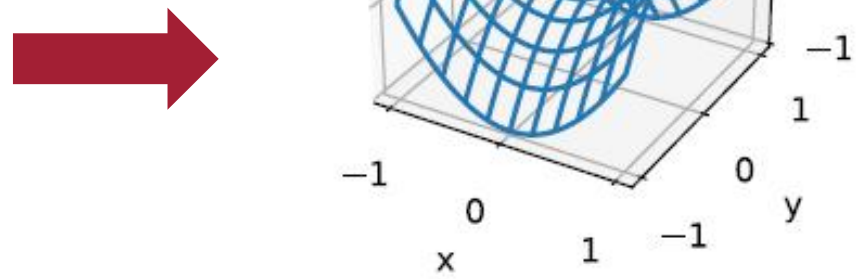


深度学习中的优化挑战

鞍点

较高维度的鞍点甚至更加隐蔽。考虑这个函数$f(x, y) = x^2 - y^2$。它的鞍点为$(0, 0)$。这是关于y的最大值，也是关于x的最小值。此外，它看起来像个马鞍，这就是鞍点的名字由来。

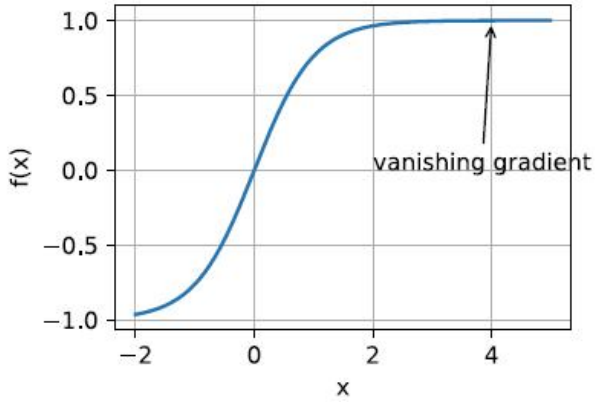
```
x, y = torch.meshgrid(
    torch.linspace(-1.0, 1.0, 101), torch.linspace(-1.0, 1.0, 101))
z = x**2 - y**2

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')
```



深度学习中的优化挑战

梯度消失

可能遇到的最隐蔽问题是梯度消失。例如，假设我们想最小化函数$f(x) = tanh(x)$，然后我们恰好从$f(x)$开始。正如我们所看到的那样， f的梯度接近零。更具体地说，$f'(x) = 1 - tanhx^2$因此$f'(4) = 0.0013$。因此，在我们取得进展之前，优化将会停滞很长一段时间。事实证明，这是在引入ReLU激活函数之前训练深度学习模型相当棘手的原因之一。



深度学习中的优化挑战

小结

正如我们所看到的那样，深度学习的优化充满挑战。幸运的是，有一系列强大的算法表现良好，即使对于初学者也很容易使用。此外，没有必要找到最优解。有时，局部最优解或其近似解仍然非常有用

- 优化问题可能有许多局部最小值。
- 一个问题可能有很多的鞍点，因为问题通常不是凸的。
- 梯度消失可能会导致优化停滞，对参数进行良好的初始化也可能是有益的。

梯度下降优化算法

梯度下降

尽管梯度下降（gradient descent）很少直接用于深度学习， 但了解它是理解随机梯度下降算法的关键。 例如，由于学习率过大，优化问题可能会发散，这种现象早已在梯度下降中出现。

梯度下降优化算法

一维梯度下降

为什么梯度下降算法可以优化目标函数？ 一维中的梯度下降可以给我们很好的启发。 考虑一类连续可微实值函数$f: \mathbb{R} \to \mathbb{R}$， 利用泰勒展开，我们可以得到

$$f(x+\epsilon) = f(x) + \epsilon f'(x) + O(\epsilon^2).$$

即在一阶近似中， $f(x+\epsilon)$可通过x处的函数值$f(x)$和一阶导数$f'(x)$得出。我们可以假设在负梯度方向上移动的ϵ会减少f的值。为简单起见，选择固定步长η, 然后取$\epsilon = \eta f'(x)$。将其代入泰勒展开式我们可以得到

$$f(x - \eta f'(x)) = f(x) - \eta f'^2(x) + O(\eta^2 f'^2(x))$$

如果导数$f'(x) \neq 0$没有消失，我们就能继续展开，因为$\eta f'^2(x) > 0$,可以令η小到足以使高阶项变得不相关。因此，

$$f(x - \eta f'(x)) \lesssim f(x).$$

这意味着，如果我们使用

$$x \leftarrow x - \eta f'(x)$$

梯度下降优化算法

一维梯度下降

来迭代x，函数$f(x)$的值可能会下降。 因此，在深度学习中的梯度下降，我们首先选择初始值x和常数$\eta > 0$， 然后使用它们连续迭代，直到停止条件达成。

下面来展示如何实现梯度下降。为了简单起见，我们选用目标函数$f(x) = x^2$。我们知道$x = 0$时$f(x)$能取得最小值，但我们仍然可以使用这个简单的函数来观察x的变化。

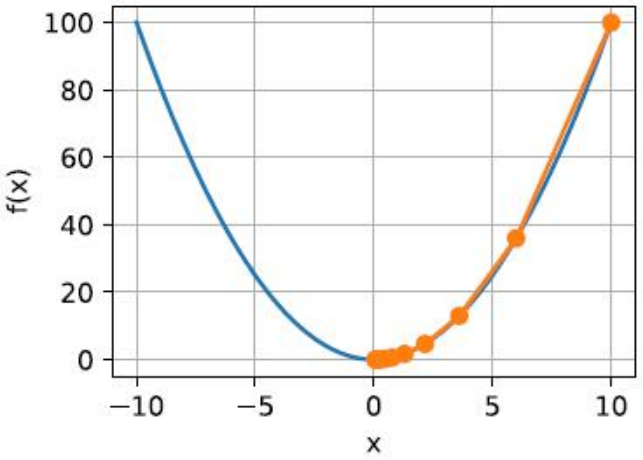
```python
def f(x):
    return x**2

def f_grad(x):
    return 2 * x
```

```python
def gd(eta, f_grad):
    x = 10          # init x position
    results = [x]
    for i in range(10):
        x -= eta * f_grad(x)
        results.append(float(x))
    print(f'epoch 10, x:{x:f}')
    return results

results = gd(0.2, f_grad)
```

```
epoch 10, x:0.060466
```

初始$x = 10$
$\eta = 0.2$

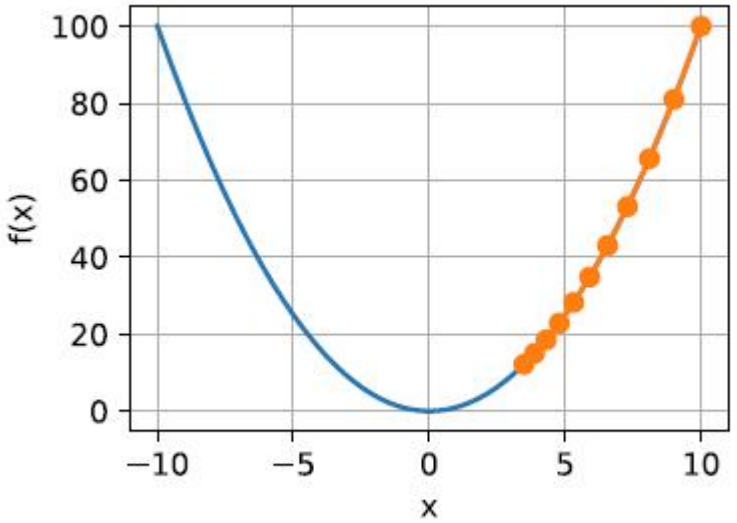
一共迭代10次



梯度下降优化算法

学习率

学习率（learning rate）决定目标函数能否收敛到局部最小值，以及何时收敛到最小值。值得注意的一点是，，如果使用的学习率太小，将导致x的更新非常缓慢，需要更多的迭代。任然考虑上面同一个的问题，如果设置$\eta = 0.05$，如下图所示，经过相同的10次迭代，值仍然离最优解很远。



$\eta = 0.05$，迭代10次

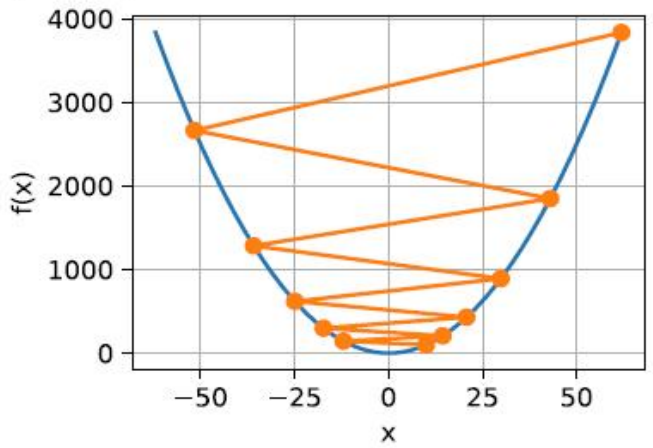
梯度下降优化算法

学习率

相反，如果我们使用过高的学习率，$|\eta f'(x)|$对于一阶泰勒展开式可能太大。也就是说$O(\eta^2 f'(x)^2)$可能变得显著了。在这种情况下，x的迭代不能保证降低$f(x)$的值。例如，我们以学习率$\eta = 1.1$时，x超出了最优解$x = 0$并逐渐发散

```
show_trace(gd(1.1, f_grad), f)
```

```
epoch 10, x:61.917364
```



$\eta = 1.1$，迭代10次

梯度下降优化算法

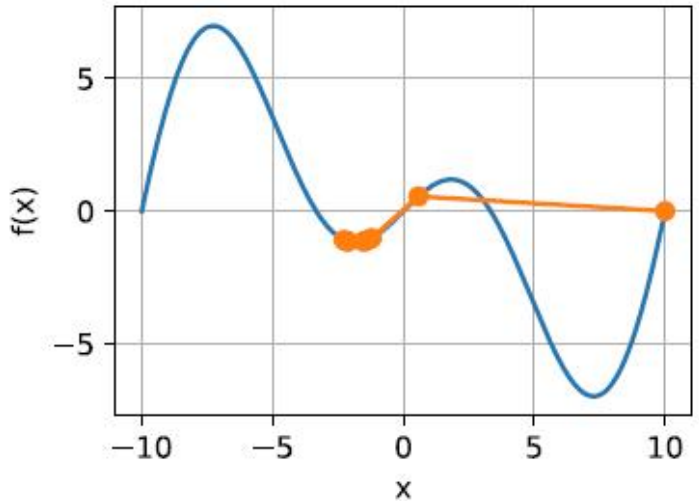
局部最小值

对于非凸函数的梯度下降，考虑函数$f(x) = x \cdot \cos(cx)$，其中c为某常数。 这个函数有无穷多个局部最小值。根据选择的学习率，我们最终可能只会得到许多解的一个。 下面的例子说明了高学习率会导致较差的局部最小值。

```
c = torch.tensor(0.15 * np.pi)

def f(x):  # 目标函数
    return x * torch.cos(c * x)

def f_grad(x):  # 目标函数的梯度
    return torch.cos(c * x) - c * x * torch.sin(c * x)

show_trace(gd(2, f_grad), f)
```

```
epoch 10, x:-1.528166
```

$\eta = 2$，迭代10次



梯度下降优化算法

多元梯度下降

现在我们对单变量的情况有了更好的理解，考虑$\mathbf{x}= [x_1, x_2, \ldots, x_n]$的情况。 即目标函数$f: \mathbb{R}^d \to \mathbb{R}$将向量映射成标量。 相应地，它的梯度也是多元的，它是一个由d个偏导数组成的向量：

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

和一维变量的情况一样，我们可以对多变量函数使用相应的泰勒近似来思考。 具体来说，

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

从上面的等式，可以看出在ϵ的二阶项中， 最陡下降的方向由负梯度$-\nabla f(\mathbf{x})$决定，选择合适的学习率$\eta > 0$来进行典型的梯度下降算法：

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

梯度下降优化算法

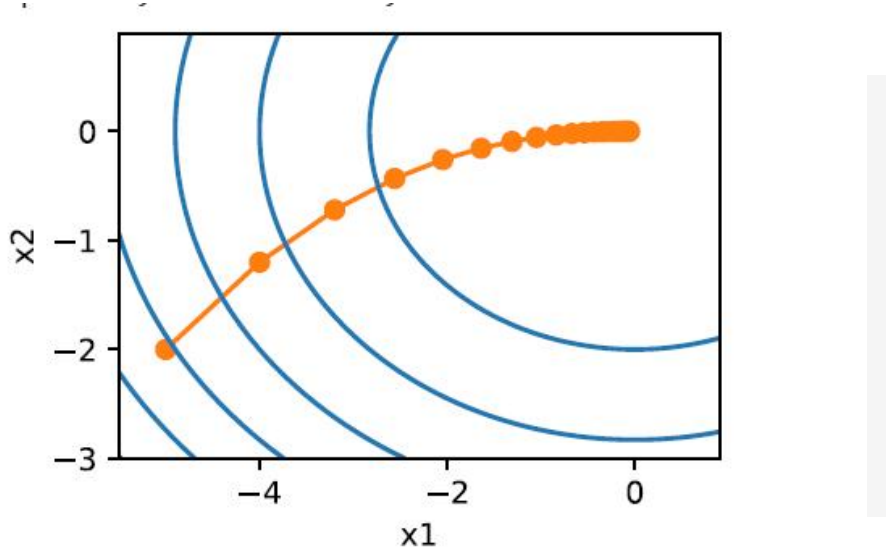
多元梯度下降

这个算法在实践中的表现如何呢？ 我们构造一个目标函数$f(\mathbf{x}) = x_1^2 + 2x_2^2$,并有二维向量$\mathbf{x} = [x_1, x_2]^T$作为输入， 标量作为输出。 梯度由$\nabla f(\mathbf{x}) = [2x_1, 4x_2]^T$给出。假设初始位置从$[-5, -2]$开始通过梯度观察$\mathbf{x}$的轨迹

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

def f_2d_grad(x1, x2):  # 目标函数的梯度
    return (2 * x1, 4 * x2)

def gd_2d(x1, x2, s1, s2, f_grad):
    g1, g2 = f_grad(x1, x2)
    return (x1 - eta * g1, x2 - eta * g2, 0, 0)

eta = 0.1
show_trace_2d(f_2d, train_2d(gd_2d, f_grad=f_2d_grad))
```

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

定义优化目标，目标梯度



定义迭代函数

梯度下降优化算法

小结

- 学习率的大小很重要：学习率太大会使模型发散，学习率太小会没有进展。
- 梯度下降会可能陷入局部极小值，而得不到全局最小值。。
- 高维模型中，调整学习率是很复杂的。

随机梯度下降

随机梯度更新

在深度学习中，目标函数通常是训练数据集中每个样本的损失函数的平均值。给定n个样本的训练数据集，我们假设$f_i(\mathbf{x})$是关于索引i的训练样本的损失函数，其中$\mathbf{x}$是参数向量。然后我们得到目标函数

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

$\mathbf{x}$的目标函数的梯度计算为：

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

如果使用梯度下降法，则每个自变量迭代的计算代价为$O(n)$，它随线性增长。因此，当训练数据集较大时，每次迭代的梯度下降计算代价将较高。

随机梯度下降

随机梯度更新

随机梯度下降（SGD）可降低每次迭代时的计算代价。在随机梯度下降的每次迭代中，对数据样本随机均匀采样一个索引i,其中$i \in \{i, \ldots, n\}$,并计算梯度$\nabla f_i(\mathbf{x})$用来更新$\mathbf{x}$:

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

每次迭代的计算代价从$O(n)$ 降至常数$O(1)$,另外随机梯度$\nabla f_i(\mathbf{x})$是对完整梯度$\nabla f(\mathbf{x})$的无偏估计，因为

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

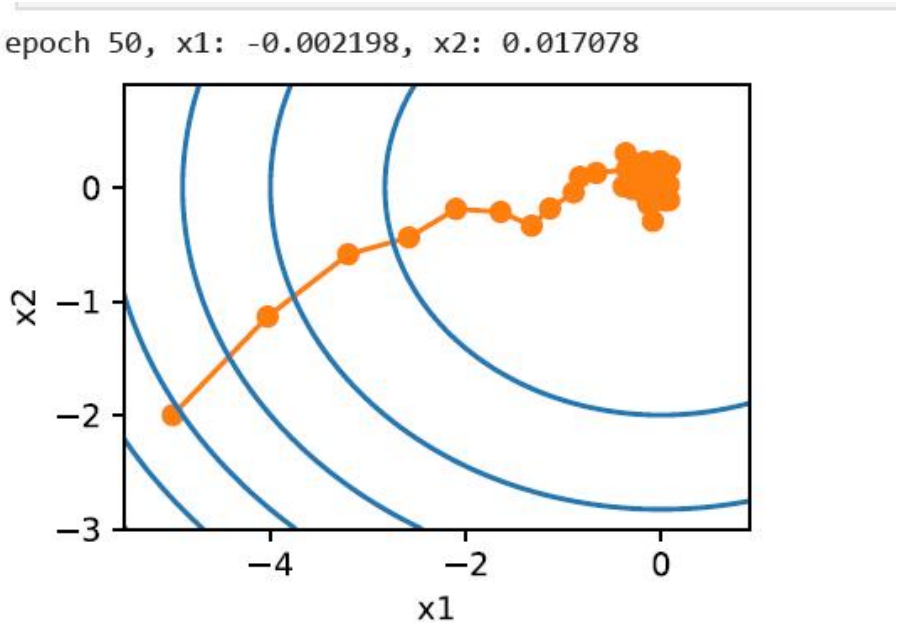
也就是说，平均而言，随机梯度是对梯度的良好估计。

随机梯度下降

随机梯度更新

现在，将随机梯度下降与梯度下降进行比较，方法是向梯度添加均值为0、方差为1的随机噪声，来模拟随机梯度下降。

```
def sgd(x1, x2, s1, s2, f_grad):
    g1, g2 = f_grad(x1, x2)
    # 模拟有噪声的梯度
    g1 += torch.normal(0.0, 1, (1,)).item()
    g2 += torch.normal(0.0, 1, (1,)).item()
    eta_t = eta * lr()
    return (x1 - eta_t * g1, x2 - eta_t * g2, 0, 0)
```



深度学习中的优化算法

小批量随机梯度下降

到目前为止，基于梯度的学习方法中遇到了两个极端情况： 使用完整数据集来计算梯度并更新参数， 以及一次处理一个训练样本来取得进展。 二者各有利弊：每当数据非常相似时，梯度下降并不是非常“数据高效”。 而由于CPU和GPU无法充分利用向量化，随机梯度下降并不特别“计算高效”。 这暗示了两者之间可能有折中方案，这便涉及到小批量随机梯度下降（minibatch gradient descent）。

小批量随机梯度下降

小批量

使用随机梯度下降时，我们每次只处理一个训练样本。 也就是说，每当我们执行 $\mathbf{w} \leftarrow \mathbf{w} - \eta_t \boldsymbol{g}_t$,其中

$$\boldsymbol{g}_t = \partial_w f(\mathbf{x}_t, \mathbf{w}).$$

每次取出一个样本，没有充分利用CPU和GPU的并行计算能力。这对与于大型数据集来说尤其成为问题。

当我们使用小批量随机梯度下降时，我们每次处理一个由多个训练样本组成的“小批量”。也就是说，计算如下：

$$\boldsymbol{g}_t = \frac{1}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} \partial_w f(\mathbf{x}_\mathrm{i}, \mathbf{w}) .$$

由于$\mathbf{x}_t$和小批量$\mathcal{B}_t$的所有元素都是从训练集中随机抽出的，因此梯度的期望保持不变。另一方面，方差会显著降低，这是由于小批量梯度由正在被平均计算的$b \in |\mathcal{B}_t|$个独立梯度组成。

小批量随机梯度下降

小结

- 随机梯度下降的“统计效率”与大批量一次处理数据的“计算效率”之间存在权衡。小批量随机梯度下降提供了两全其美的答案：计算和统计效率。

- 在小批量随机梯度下降中，处理通过训练数据的随机排列获得的批量数据

- 一般来说，小批量随机梯度下降比随机梯度下降和梯度下降的速度快，收敛风险较小。

深度学习中的优化算法

动量法

随机梯度下降法在每次迭代中，沿着目标函数的负梯度方向更新参数。然而，随机梯度下降法容易陷入局部最优解，且收敛速度慢。它是一种加速梯度下降的优化算法。

动量法

泄漏平均值

小批量随机梯度下降作为加速计算的手段。 它也有很好的作用，即平均梯度减小了方差。如果我们能够从方差减少的影响中受益，甚至超过小批量上的梯度平均值，那很不错。 完成这项任务的一种选择是用泄漏平均值（leaky average）取代梯度计算：

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

其中$\beta \in (0,1)$。 这有效地将瞬时梯度替换为多个“过去”梯度的平均值。$\mathbf{v}$被称为(momentum),它累加了过去的梯度。为了更详细地解释，可以递归地将$\mathbf{v}_t$扩展

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

较大的β相当于长期平均值，而较小的β相对于梯度法只是略有修正。 新的梯度替换不再指向特定实例下降最陡的方向，而是指向过去梯度的加权平均值的方向。 这使我们能够实现对单批量计算平均值的大部分好处，而不产生实际计算其梯度的代价。

动量法

上述推理构成了“加速”梯度方法的基础，例如具有动量的梯度。 在优化问题条件不佳的情况（例如，有些方向的进展比其他方向慢得多，类似狭窄的峡谷），“加速”梯度还额外享受更有效的好处。此外，它们允许我们对随后的梯度计算平均值，以获得更稳定的下降方向。正如人们所期望的，由于其功效，动量是深度学习及其后优化中一个深入研究的主题。动量是由 (Polyak, 1964)提出的。(Nesterov, 2018)在凸优化的背景下进行了详细的理论讨论。

Lectures on convex optimization.[Nesterov,2018]

Some methods of speeding up the convergence of iteration methods.[Plolyak,1964]

动量法

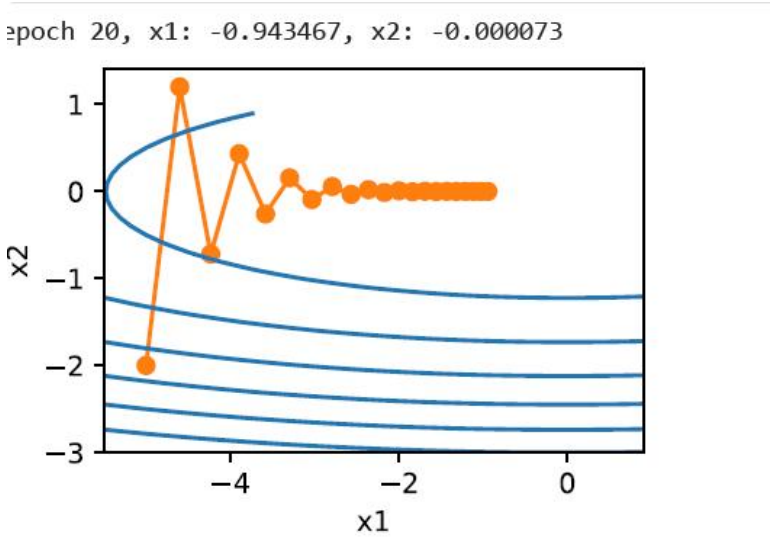
条件不佳的问题

为了更好地了解动量法的几何属性，回想我们在前面中使用了$f(\mathbf{x}) = x_1^2 + 2x_2^2$，即中度扭曲的椭球目标。通过向方向伸展它来进一步扭曲这个函数

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

与之前一样， 在$(0,0)$有最小值， 该函数在x_1的方向上非常平坦。

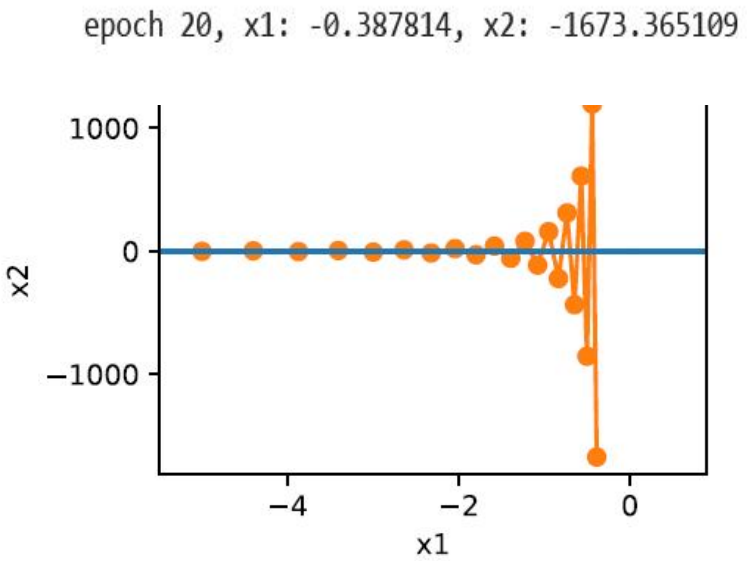
```
eta = 0.4
def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2
def gd_2d(x1, x2, s1, s2):
    return (x1 - eta * 0.2 * x1, x2 - eta * 4 * x2, 0, 0)
```



动量法

条件不佳的问题

从构造来看， x_2方向的梯度比水平x_1方向的梯度大得多，变化也快得多。 因此，我们陷入两难：如果选择较小的学习率，我们会确保不会在x_2方向发散，但要承受x_1在方向的缓慢收敛。相反，如果学习率较高，我们在x_1方向上进展很快，但在x_2方向将会发散。下面的画图说明了即使学习率从0.4略微提高到0.6，也会发生变化。



动量法

动量法优化

动量法（momentum）使我们能够解决上面描述的梯度下降问题。 观察上面的优化轨迹，我们可能会直觉到计算过去的平均梯度效果会很好。在 方向上，这将聚合非常对齐的梯度，从而增加我们在每一步中覆盖的距离。 相反，在梯度振荡的方向，由于相互抵消了对方的振荡，聚合梯度将减小步长大小。

使用$\mathbf{v}_t$而不是梯度$\boldsymbol{g}_t$可以生成以下更新等式：

$$\mathbf{v}_t \leftarrow \beta \mathbf{v}_{t-1} + \boldsymbol{g}_{t,t-1},$$
$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \eta \mathbf{v}_t.$$

值得注意的是，对于$\beta = 0$，我们恢复常规的梯度下降。

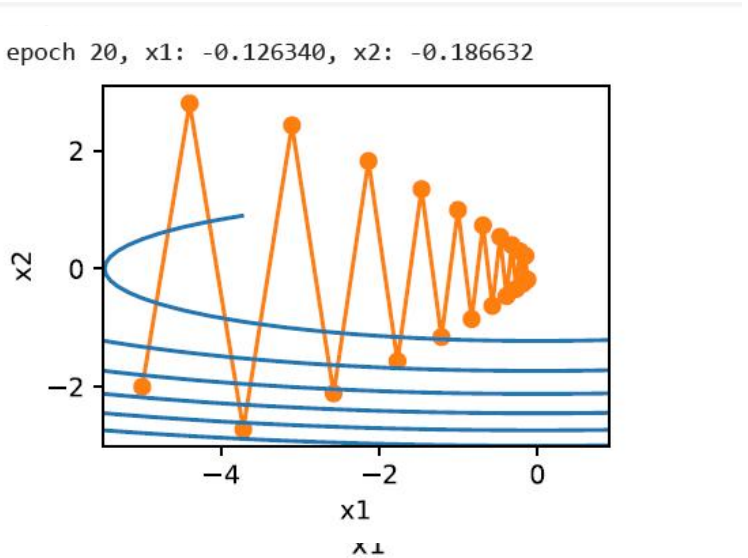
动量法

动量法优化

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

```
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
```



正如所见，尽管学习率与我们以前使用的相同，动量法仍然很好地收敛了。 那么当降低动量参数时会发生什么。 将其减半至$\beta = 0.25$会导致一条几乎没有收敛的轨迹。 尽管如此，它比没有动量时解将会发散要好得多。

值得注意的是，相比于小批量随机梯度下降，动量方法需要维护一组辅助变量，即速度。 它与梯度以及优化问题的变量具有相同的形状。

动量法

小结

- 动量法用过去梯度的平均值来替换梯度，这大大加快了收敛速度。
- 动量法可以防止在随机梯度下降的优化过程停滞的问题。
- 动量法的实现非常简单，但它需要我们存储额外的状态向量
- 对于无噪声梯度下降和嘈杂随机梯度下降，动量法都是可取的。

AdaGrad算法

稀疏特征和学习率

为了获得良好的准确性，我们大多希望在训练的过程中降低学习率，速度通常为$O(t^{-\frac{1}{2}})$或更低。关于稀疏特征（即只在偶尔出现的特征）的模型训练，这对自然语言来说很常见。 例如，我们看到“预先条件”这个词比“学习”这个词的可能性要小得多。

只有在这些不常见的特征出现时，与其相关的参数才会得到有意义的更新。 鉴于学习率下降，我们可能最终会面临这样的情况：常见特征的参数相当迅速地收敛到最佳值，而对于不常见的特征，我们仍缺乏足够的观测以确定其最佳值。 换句话说，学习率要么对于常见特征而言降低太慢，要么对于不常见特征而言降低太快。

解决此问题的一个方法是记录我们看到特定特征的次数，然后将其用作调整学习率。 即我们可以使用大小为$\eta_i = \frac{\eta_0}{\sqrt{s(i,t)+c}}$的学习率，而不是$\eta_i = \frac{\eta_0}{\sqrt{t+c}}$。 在这里$s(i,t)$计下了我们截至$t$时观察到功能$i$的次数。 这其实很容易实施且不产生额外的计算损耗。

AdaGrad算法

AdaGrad算法 (Duchi et al., 2011)通过将粗略的计数器$s(i,t)$替换为先前观察所得梯度的平方之和来解决这个问题。 它使用$s(i,t+1)=s(i,t)+ \ (\partial_i f(x))^2$来调整学习率。这有两个好处：首先，我们不再需要决定梯度何时算足够大。 其次，它会随梯度的大小自动变化。通常对应于较大梯度的坐标会显著缩小，而其他梯度较小的坐标则会得到更平滑的处理。 在实际应用中，它促成了计算广告学及其相关问题中非常有效的优化程序。 但是，它遮盖了AdaGrad固有的一些额外优势，这些优势在预处理环境中很容易被理解。

Adaptive subgradient methods for online learning and stochastic optimization.[Duchi et al., 2011]

AdaGrad算法

算法实现

通过使用变量s_t来累加过去的梯度方差，如下所示：

$$\boldsymbol{g}_t = \partial_w \ell(y_t, f(\mathbf{x}_{t,} \mathbf{w})),$$

$$\mathbf{s}_t = \mathbf{s}_{t-1} + \boldsymbol{g}_t{}^2,$$

$$\mathbf{w_t} = \mathbf{w_{t-1}} - \frac{\eta}{\sqrt{s_t + \epsilon}} \cdot \boldsymbol{g}_t$$

与之前一样，η是学习率，ϵ是一个为维持数值稳定性而添加的常数，用来确保我们不会除以0,最后，我们初始化$\mathbf{s_0} = 0$。

就像在动量法中我们需要跟踪一个辅助变量一样，在AdaGrad算法中，我们允许每个坐标有单独的学习率。

AdaGrad算法

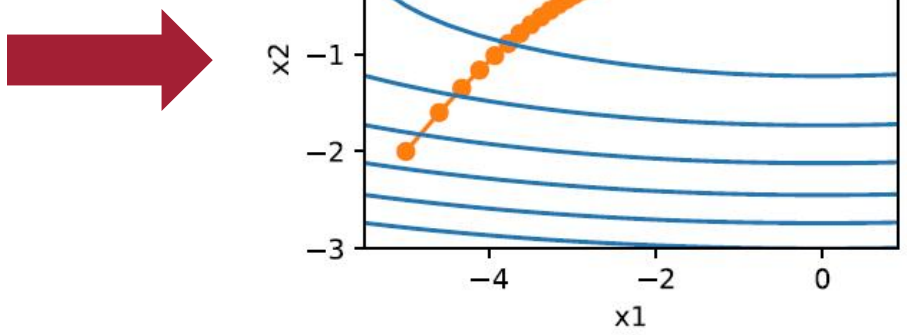
实际表现

眼下让我们先看看它在二次凸问题中的表现如何。 仍然以同一函数为例：

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

我们将使用与之前相同的学习率来实现AdaGrad算法，即$\eta = 0.4$。可以看到，自变量的迭代轨迹较平滑。 但由于$\mathbf{s}_t$的累加效果使学习率不断衰减，自变量在迭代后期的移动幅度较小。

```
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
```

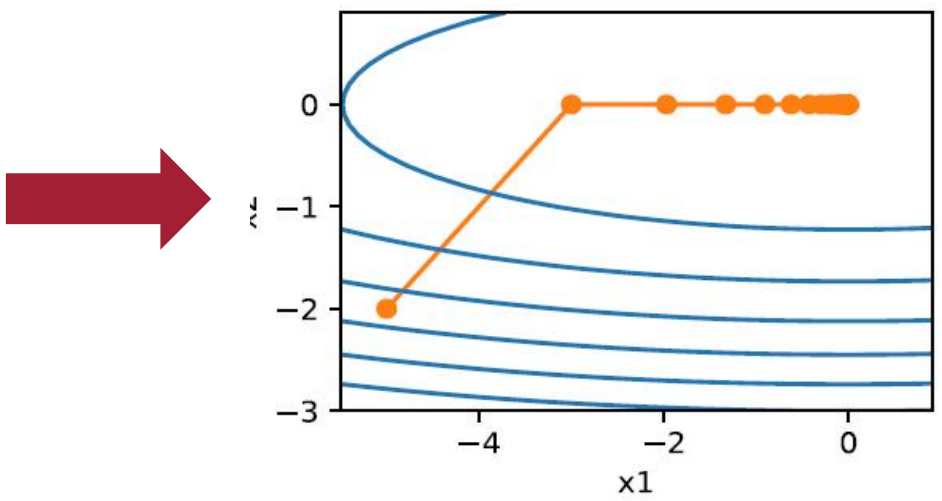


AdaGrad算法

实际表现

将学习率提高到2，可以看到更好的表现。

```
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 = 2
```



AdaGrad算法

小结

- AdaGrad算法会在单个坐标层面动态降低学习率。
- AdaGrad算法利用梯度的大小作为调整进度速率的手段：用较小的学习率来补偿带有较大梯度的坐标。
- 如果优化问题的结构相当不均匀，AdaGrad算法可以帮助缓解扭曲。
- 在深度学习问题上，AdaGrad算法有时在降低学习率方面可能过于剧烈。

RMSProp算法

引言

学习率按预定时间表$O(t^{-\frac{1}{2}})$显著降低。 虽然这通常适用于凸问题，但对于深度学习中遇到的非凸问题，可能并不理想。但是，作为一个预处理，Adagrad算法按坐标顺序的适应性是非常可取的。

(Tieleman and Hinton, 2012)建议以RMSProp算法作为将速率调度与坐标自适应学习率分离的简单修复方法。 问题在于，Adagrad算法将梯度$\boldsymbol{g}_t$的平方累加成状态矢量$\mathbf{s}_t = \mathbf{s}_{t-1} + \boldsymbol{g}_t{}^2$。 因此，由于缺乏规范化，没有约束力，$\mathbf{s}_t$持续增长，几乎上是在算法收敛时呈线性递增。

如果按动量法中的方式使用泄漏平均值，即$\mathbf{s}_t = \gamma \mathbf{s}_{t-1} + (1-\gamma)\boldsymbol{g}_t{}^2$，其中参数$\gamma > 0$。 保持所有其它部分不变就产生了RMSProp算法。

Lecture 6.5-rmsprop: divide the gradient by a running average of its recent magnitude.[Tieleman & Hinton, 2012 NeurIPS]

RMSProp算法

算法原理

和Adagrad算法一样，写出更新方程。

$$\mathbf{s}_t \leftarrow \gamma \mathbf{s}_{t-1} + (1-\gamma)\boldsymbol{g}_t{}^2,$$

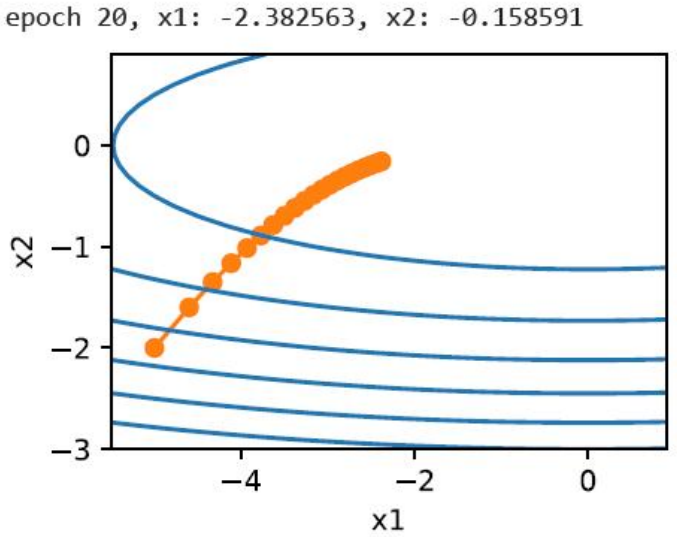
$$\mathbf{x}_t = \mathbf{x}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \cdot \boldsymbol{g}_t$$

常数 ϵ 通常设置为10^{-6}，以确保我们不会因除以零或步长过大而受到影响。

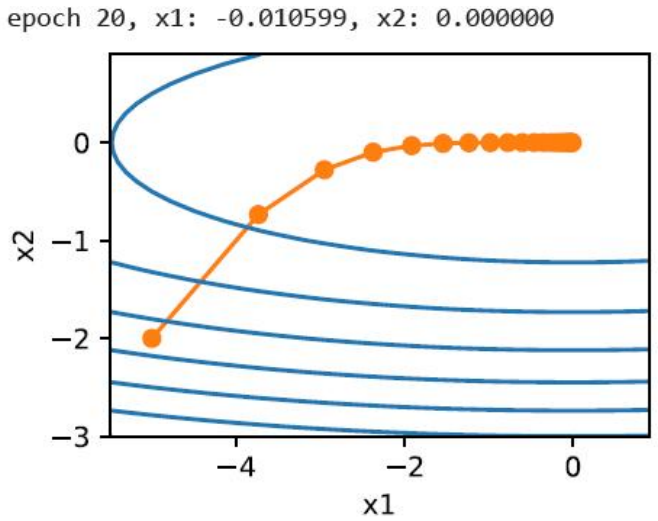
RMSProp算法

实际表现

和之前一样，使用二次函数$f(\mathbf{x}) = x_1^2 + 2x_2^2$.来观察RMSProp算法的轨迹。 当我们使用学习率为0.4的Adagrad算法时，变量在算法的后期阶段移动非常缓慢，因为学习率衰减太快。 RMSProp算法中不会发生这种情况，




```
1e-6
g1 ** 2
g2 ** 2
* g1
* g2


2
```



RMSProp算法

小结

- RMSProp算法与Adagrad算法非常相似，因为两者都使用梯度的平方来缩放系数。
- RMSProp算法与动量法都使用泄漏平均值。
- 系数γ决定了在调整每坐标比例时历史记录的时长。

Adam算法

引言

前面已经介绍了许多有效优化的技术。 首先让我们详细回顾一下这些技术：

- 随机梯度下降在解决优化问题时比梯度下降更有效
- 动量法中我们添加了一种机制，用于汇总过去梯度的历史以加速收敛。
- AdaGrad算法通过对每个坐标缩放来实现高效计算的预处理器。

Adam算法 (Kingma and Ba, 2014)将所有这些技术汇总到一个高效的学习算法中。 不出预料，作为深度学习中使用的更强大和有效的优化算法之一，它非常受欢迎。下面我们了解一下Adam算法。

Adam: a method for stochastic optimization.[Kingma & Ba, 2014]

Adam算法

算法原理

Adam算法的关键组成部分之一是：它使用指数加权移动平均值来估算梯度的动量和二次矩，即它使用状态变量

$$\mathbf{v}_t \leftarrow \beta_1 \mathbf{v}_{t-1} + (1-\beta_1)\boldsymbol{g}_t,$$
$$\mathbf{s}_t \leftarrow \beta_2 \mathbf{s}_{t-1} + (1-\beta_2)\boldsymbol{g}_t{}^2.$$

这里β_1和β_2是非负加权参数。 常将它们设置为$\beta_1 = 0.9$和$\beta_2 = 0.99(0.98)$。也就是说，方差估计的移动远远慢于动量估计的移动。

首先，我们以非常类似于RMSProp算法的方式重新缩放梯度以获得

$$\boldsymbol{g}_t{}' = \frac{\eta \mathbf{v}_t}{\sqrt{\mathbf{s}_t} + \epsilon}$$

与RMSProp不同，更新使用动量$\mathbf{v}_t$而不是梯度本身，最后，

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \boldsymbol{g}_t{}'$$

回顾Adam算法，它的设计灵感很清楚：动量和规模在状态变量中清晰可见，明确的学习率η使我们能够控制步长来解决收敛问题。

Adam算法

实现

```
def init_adam_states(feature_dim):
    v_w, v_b = torch.zeros((feature_dim, 1)), torch.zeros(1)
    s_w, s_b = torch.zeros((feature_dim, 1)), torch.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):
        with torch.no_grad():
            v[:] = beta1 * v + (1 - beta1) * p.grad
            s[:] = beta2 * s + (1 - beta2) * torch.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 / (torch.sqrt(s_bias_corr)
                                                       + eps)
        p.grad.data.zero_()
    hyperparams['t'] += 1
```

Adam算法

小结

- Adam算法将许多优化算法的功能结合到了相当强大的更新规则中。
- Adam算法在RMSProp算法基础上创建的。

谢谢聆听

2024.10.17