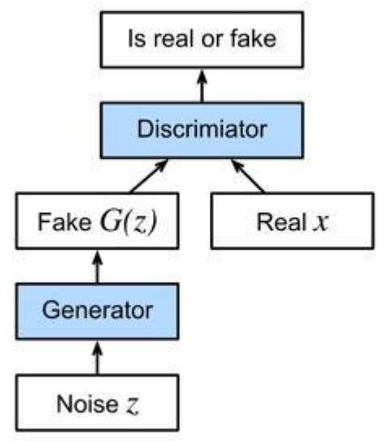
生成对抗网络（Generative Adversarial Networks）

在本书的绝大部分中，我们都在说如何（用模型）进行预测。例如，我们使用深度神经网络的形式把数据点映射到要预测的标签上。这种学习形式称为判别式学习（discriminative learning）。例如，我们用判别式学习来分辨猫的图片和狗的图片。分类模型和回归模型都是判别式学习的例子。在大型复杂的数据集上，通过反向传播算法，我们基本可以使用神经网络完成期望的判别式学习。短短5、6年间，在高分辨率图像上的分类精度，已经从不可用到达了人类的水平。在这些判别式学习领域，神经网络已经做得很好，这里我们就不再给出更多的案例了。

但是对于机器学习来说，需要解决的任务不只是判别式学习。举个例子，给定一个大数据集，这个数据集不含有标签。在这种情况下，我们可能会需要去学习一个能够捕捉数据集特征的模型。在这个模型的基础上，我们可以采样人工合成的数据，这些数据点都服从训练数据的分布。比如，给定一个大量人脸照片的数据集，我们希望生成一张新的人脸，就好像是从这个数据集里出来的人脸照片。这种形式的学习称为生成式模型。

就在不久之前，我们还没有方法来人工合成拟真的照片。深度神经网络在判别式学习上的成功为我们提供了新的可能。在过去三年里，深度网络发展的一个重要趋势，就是应用于原本我们不认为是监督学习的任务上。循环神经网络语言模型就是这样一个例子，通过判别式方法训练的神经网络（用预测下一个单词的方式进行训练），可以用作一个生成模型。

2014年，一篇论文突破性地介绍了生成对抗网络（Generative adversarial networks，GANs）`Goodfellow.Pouget-Abadie.Mirza.ea.2014`。生成对抗网络采用了一种巧妙的方法，把判别式模型的能力用于生成式模型。GAN的核心思想是，如果我们不能分辨真实数据和生成器生成的假数据，那么该数据生成器就是有效的。在统计学上，这称为双样本测试（two-sample test），用来测试数据集 $\boldsymbol{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ 和 $\boldsymbol{X}' = \{\boldsymbol{x}'_1, \ldots, \boldsymbol{x}'_n\}$ 是否来自同一个数据分布。大多数统计学论文与生成对抗网络的主要区别是，后者用构造式方法来实现双样本测试这一概念。换句话说，生成对抗网络不仅仅是训练一个模型，然后说“hey，这两个数据集看起来不像是从同一个分布来的”。生成对抗网络把双样本测试 two-sample test 的结果提供给生成模型作为训练信号。这样，我们就可以提升数据生成的质量，直到生成了逼近真实数据的样例。最终，生成器要骗过分类器，即使这个分类器是深度神经网络中的佼佼者。



生成对抗网络的结构如图所示。可以看到，在这个结构中有两个部分：首先，我们需要一个模型（例如，一个深度网络，但也可以是任何模型，比如一个游戏渲染引擎），这个模型拥有生成逼真数据的潜力。如果我们要处理的目标是图片，那么它就是一个用来生成图片的模型。如果我们要处理的是语音，那么就需要一个生成语音序列的模型。这个模型称为生成器网络（generator network）。第二个部分是判别器网络（discriminator network）。这个网络试图分辨伪造的数据和真实的数据。两个网络相互竞争。生成器网络尝试欺骗判别式网络。同时，判别器网络会学习应对新的伪造数据。这个信息，又被用来提升生成器网络的效果，如此往复。

判别器是个二分类器，用来分辨输入 $\boldsymbol{x}$ 是真的（在真实数据中）还是伪造的（由生成器生成）。具体来说，对于输入 $\mathbf{x}$，判别器输出一个标量预测 $o \in \mathbb{R}$，例如可以使用一个含有一层隐藏层的全连接网络，然后用 sigmoid 函数获得预测概率 $D(\mathbf{x}) = 1/(1 + e^{-o})$。假定把真实数据的标签 y 设为 1，然后把伪造数据的标签设为 0。我们可以最小化交叉熵损失函数，以此来训练判别器，即，

$$\min_D \{-y \log D(\mathbf{x}) - (1-y)\log(1 - D(\mathbf{x}))\},$$

对生成器来说，首先要随机初始化一些参数 $\mathbf{z} \in \mathbb{R}^d$，例如从一个正态分布中生成，$\mathbf{z} \sim \mathcal{N}(0, 1)$。通常我们把 $\mathbf{z}$ 称为隐变量（latent variable）。
然后用函数来生成 $\mathbf{x}' = G(\mathbf{z})$。生成器的目标是欺骗判别器，让判别器把生成的数据 $\mathbf{x}' = G(\mathbf{z})$ 分类为真实数据，也就是说，$D(G(\mathbf{z})) \approx 1$。

给定一个判别器 D，我们更新生成器 G 的参数，以此最大化交叉熵损失函数，当 $y = 0$ 时，

$$\max_G \{-(1-y)\log(1 - D(G(\mathbf{z})))\} = \max_G \{-\log(1 - D(G(\mathbf{z})))\}.$$

如果判别器工作得很好（没有被骗），那么 $D(\mathbf{x}') \approx 0$，所以上面的损失函数接近于0，这意味着梯度非常小，以至于生成器几乎不会更新。因此，通常我们最小化下面的损失函数：

$$\min_G \{-y \log(D(G(\mathbf{z})))\} = \min_G \{-\log(D(G(\mathbf{z})))\},$$

也就是把 $\mathbf{x}' = G(\mathbf{z})$ 送入判别器，但把标签设为 $y = 1$。

总结一下，D 和 G 在玩一个称为“minimax”的博弈，可以由下面这个目标函数表示：

$$min_D max_G \{-E_{x \sim \text{Data}} log D(\mathbf{x}) - E_{z \sim \text{Noise}} log(1 - D(G(\mathbf{z})))\}.$$

生成对抗网络的许多应用是在图片图像领域。为了便于展示，我们首先会拟合一个非常简单的数据分布。我们会展示，如果用生成对抗网络来构建一个世界上最低效的高斯分布的模拟器，会发生什么。让我们开始吧。

In [1]:

```
%matplotlib inline
import matplotlib.pyplot as plt
from torch.utils.data import DataLoader
from torch import nn
import numpy as np
from torch.autograd import Variable
import torch
```

生成一些“真实”数据

这可能是世界上最糟糕的例子了，我们简单地从高斯分布生成一些数据。

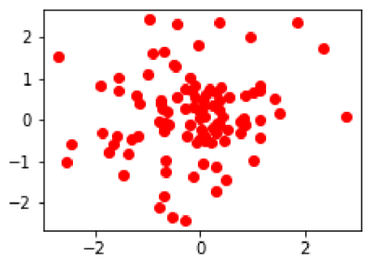
In [2]:

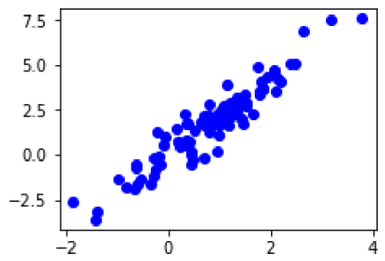
```
X=np.random.normal(size=(1000,2))
A=np.array([[1,2],[-0.1,0.5]])
b=np.array([1,2])
data=X.dot(A)+b
```

让我们来看一下现在有什么。这是一个带偏移的正态分布，用均值 b 和协方差矩阵 $A^T A$ 描述。

In [3]:

```
plt.figure(figsize=(3.5,2.5))
plt.scatter(X[:100,0],X[:100,1],color='red')
plt.show()
plt.figure(figsize=(3.5,2.5))
plt.scatter(data[:100,0],data[:100,1],color='blue')
plt.show()
print("The covariance matrix is\n%s" % np.dot(A.T, A))
```






```
The covariance matrix is
[[1.01 1.95]
 [1.95 4.25]]
```

In [4]:

```
batch_size=8
data_iter=DataLoader(data,batch_size=batch_size)
```

生成器（Generator）

生成器网络是一个最简单的网络形式——就是一层线性模型。我们将使用高斯数据生成器来驱动这个线性网络。因此，它实际上只需要学习几个模型参数就可以完美地伪造数据了。

In [5]:

```
class net_G(nn.Module):
    def __init__(self):
        super(net_G,self).__init__()
        self.model=nn.Sequential(
            nn.Linear(2,2),
        )
        self._initialize_weights()
    def forward(self,x):
        x=self.model(x)
        return x
    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m,nn.Linear):
                m.weight.data.normal_(0,0.02)
                m.bias.data.zero_()
```

判别器（Discriminator）

对于判别器来说，需要稍微复杂一些：这里使用3层的感知机（MLP）来让整个展示更加有趣。

In [6]:

```
class net_D(nn.Module):
    def __init__(self):
        super(net_D,self).__init__()
        self.model=nn.Sequential(
            nn.Linear(2,5),
            nn.Tanh(),
            nn.Linear(5,3),
            nn.Tanh(),
            nn.Linear(3,1),
            nn.Sigmoid()
        )
        self._initialize_weights()
    def forward(self,x):
        x=self.model(x)
        return x
    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m,nn.Linear):
                m.weight.data.normal_(0,0.02)
                m.bias.data.zero_()
```

训练

首先我们定义一个函数来更新判别器。

In [7]:

```
# Saved in the d2l package for later use
def update_D(X,Z,net_D,net_G,loss,trainer_D):
    batch_size=X.shape[0]
    Tensor=torch.FloatTensor
    ones=Variable(Tensor(np.ones(batch_size))).view(batch_size,1)
    zeros = Variable(Tensor(np.zeros(batch_size))).view(batch_size,1)
    real_Y=net_D(X.float())
    fake_X=net_G(Z)
    fake_Y=net_D(fake_X)
    loss_D=(loss(real_Y,ones)+loss(fake_Y,zeros))/2
    loss_D.backward()
    trainer_D.step()
    return float(loss_D.sum())
```

生成器也可以通过类似的方法进行更新。这里我们再次使用交叉熵损失函数，但是要把标签伪造数据的标签从 0 设为 1.

In [8]:

```
# Saved in the d2l package for later use
def update_G(Z,net_D,net_G,loss,trainer_G):
    batch_size=Z.shape[0]
    Tensor=torch.FloatTensor
    ones=Variable(Tensor(np.ones((batch_size,)))).view(batch_size,1)
    fake_X=net_G(Z)
    fake_Y=net_D(fake_X)
    loss_G=loss(fake_Y,ones)
    loss_G.backward()
    trainer_G.step()
    return float(loss_G.sum())
```

判别器和生成器共同作用一个采用交叉熵损失函数的二分类逻辑回归。我们使用Adam来平滑训练过程。在每一个迭代轮次，首先更新判别器，然后更新生成器。损失和生成的样例可以用来展示。

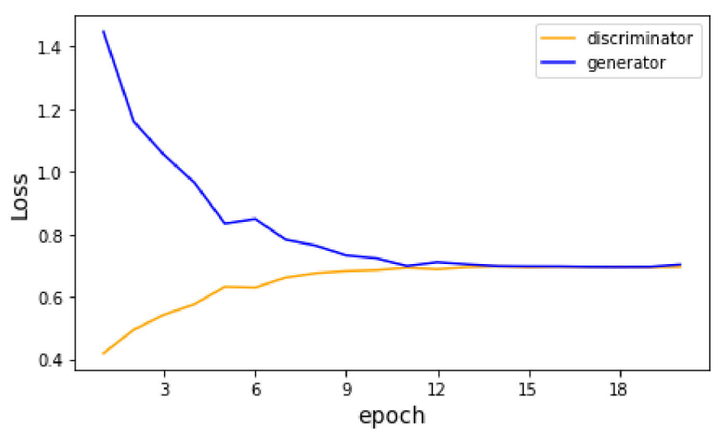
In [9]:

```
def train(net_D,net_G,data_iter,num_epochs,lr_D,lr_G,latent_dim,data):
    loss=nn.BCELoss()
    Tensor=torch.FloatTensor
    trainer_D=torch.optim.Adam(net_D.parameters(),lr=lr_D)
    trainer_G=torch.optim.Adam(net_G.parameters(),lr=lr_G)
    plt.figure(figsize=(7,4))
    d_loss_point=[]
    g_loss_point=[]
    d_loss=0
    g_loss=0
    for epoch in range(1,num_epochs+1):
        d_loss_sum=0
        g_loss_sum=0
        batch=0
        for X in data_iter:
            batch+=1
            X=Variable(X)
            batch_size=X.shape[0]
            Z=Variable(Tensor(np.random.normal(0,1,(batch_size,latent_dim))))
            trainer_D.zero_grad()
            d_loss = update_D(X, Z, net_D, net_G, loss, trainer_D)
            d_loss_sum+=d_loss
            trainer_G.zero_grad()
            g_loss = update_G(Z, net_D, net_G, loss, trainer_G)
            g_loss_sum+=g_loss
        d_loss_point.append(d_loss_sum/batch)
        g_loss_point.append(g_loss_sum/batch)
    plt.ylabel('Loss', fontdict={'size': 14})
    plt.xlabel('epoch', fontdict={'size': 14})
    plt.xticks(range(0,num_epochs+1,3))
    plt.plot(range(1,num_epochs+1),d_loss_point,color='orange',label='discriminator')
    plt.plot(range(1,num_epochs+1),g_loss_point,color='blue',label='generator')
    plt.legend()
    plt.show()
    print(d_loss,g_loss)
    Z =Variable(Tensor( np.random.normal(0, 1, size=(100, latent_dim))))
    fake_X=net_G(Z).detach().numpy()
    plt.figure(figsize=(3.5,2.5))
    plt.scatter(data[:,0],data[:,1],color='blue',label='real')
    plt.scatter(fake_X[:,0],fake_X[:,1],color='orange',label='generated')
    plt.legend()
    plt.show()
```

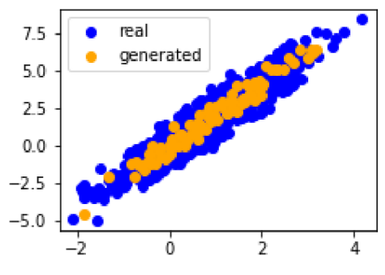
现在我们设置好超参数来拟合高斯分布。

In [10]:

```
if __name__ == '__main__':
    lr_D,lr_G,latent_dim,num_epochs=0.05,0.005,2,20
    generator=net_G()
    discriminator=net_D()
    train(discriminator,generator,data_iter,num_epochs,lr_D,lr_G,latent_dim,data)
```




```
0.7003533840179443 0.6753175258636475
```



总结

- 生成对抗网络（GANs）包含两个深度网络，生成器和判别器。
- 生成器生成尽量逼真的图片，以此来骗过判别器，这可以通过最大化交叉熵损失函数实现，即，$\max \log(D(\mathbf{x}'))$.
- 判别器尝试分辨生成的图片和真实的图片，这可以通过最小化交叉熵损失函数实现，即，$\min -y \log D(\mathbf{x}) - (1-y) \log(1-D(\mathbf{x}))$.

练习

- 是否存在一个平衡状态，在这个状态下，生成器赢了，也就说，对于有限的样例，判别器不能区分生成的数据和真实的数据？