学习笔记-8: Task 8

22 - 图像识别案例2

Kaggle上的狗品种识别 (ImageNet Dogs)

本节将解决Kaggle竞赛中的犬种识别挑战，在这项比赛中，我们尝试确定120种不同的狗。该比赛中使用的数据集实际上是著名的ImageNet数据集的子集。

整理数据集

从比赛网址上下载的数据集，其目录结构为：

```
| Dog Breed Identification
    | train
    |   | 000bec180eb18c7604dcecc8fe0dba07.jpg
    |   | 00a338a92e4e7bf543340dc849230e75.jpg
    |   | ...
    | test
    |   | 00a3edd22dc7859c487a64777fc8d093.jpg
    |   | 00a6892e5c7f92c1f465e213fd904582.jpg
    |   | ...
    | labels.csv
    | sample_submission.csv
```

train和test目录下分别是训练集和测试集的图像，训练集包含10,222张图像，测试集包含10,357张图像，图像格式都是JPEG，每张图像的文件名是一个唯一的id。labels.csv包含训练集图像的标签，文件包含10,222行，每行包含两列，第一列是图像id，第二列是狗的类别。狗的类别一共有120种。

我们希望对数据进行整理，方便后续的读取，主要目标是：

- 从训练集中划分出验证数据集，用于调整超参数。划分之后，数据集应该包含4个部分：划分后的训练集、划分后的验证集、完整训练集、完整测试集
- 对于4个部分，建立4个文件夹：train, valid, train_valid, test。在上述文件夹中，对每个类别都建立一个文件夹，在其中存放属于该类别的图像。前三个部分的标签已知，所以各有120个子文件夹，而测试集的标签未知，所以仅建立一个名为unknown的子文件夹，存放所有测试数据。

希望整理后的数据集目录结构为：

```
| train_valid_test
    | train
    |   | affenpinscher
    |   |   | 00ca18751837cd6a22813f8e221f7819.jpg
    |   |   | ...
    |   | afghan_hound
    |   |   | 0a4f1e17d720cdff35814651402b7cf4.jpg
    |   |   | ...
    |   | ...
```

```
      | valid
      |   | affenpinscher
      |   |   | 56af8255b46eb1fa5722f37729525405.jpg
      |   |   | ...
      |   | afghan_hound
      |   |   | 0df400016a7e7ab4abff824bf2743f02.jpg
      |   |   | ...
      |   | ...
      | train_valid
      |   | affenpinscher
      |   |   | 00ca18751837cd6a22813f8e221f7819.jpg
      |   |   | ...
      |   | afghan_hound
      |   |   | 0a4f1e17d720cdff35814651402b7cf4.jpg
      |   |   | ...
      |   | ...
      | test
      |   | unknown
      |   |   | 00a3edd22dc7859c487a64777fc8d093.jpg
      |   |   | ...
```

图像增强

- 对训练集
    1. 随机对图像裁剪出面积为原图像面积`0.08~1`倍、且高和宽之比在`3/4~4/3`的图像，再放缩为高和宽均为`224`像素的新图像
    2. 以`0.5`的概率随机水平翻转
    3. 随机更改亮度、对比度和饱和度
    4. 对各个通道做标准化
- 在测试集上的图像增强只做确定性的操作

读取数据

`new_data_dir`目录下有`train`, `valid`, `train_valid`, `test`四个目录。这四个目录中，每个子目录表示一种类别，目录中是属于该类别的所有图像。

定义模型

这个比赛的数据属于`ImageNet`数据集的子集，这里使用微调的方法，选用在`ImageNet`完整数据集上预训练的模型来抽取图像特征，以作为自定义小规模输出网络的输入。
此处使用已训练的`ResNet-34`模型，直接复用预训练模型在输出层的输入，即抽取的特征，然后重新定义输出层，本次仅对重定义的输出层的参数进行训练，而对于用于抽取特征的部分，保留预训练模型的参数。

定义训练函数

```
def evaluate_loss_acc(data_iter, net, device):
    # 计算data_iter上的平均损失与准确率
    loss = nn.CrossEntropyLoss()
    is_training = net.training  # Bool net是否处于train模式
    net.eval()
```

```
    l_sum, acc_sum, n = 0, 0, 0
    with torch.no_grad():
        for X, y in data_iter:
            X, y = X.to(device), y.to(device)
            y_hat = net(X)
            l = loss(y_hat, y)
            l_sum += l.item() * y.shape[0]
            acc_sum += (y_hat.argmax(dim=1) == y).sum().item()
            n += y.shape[0]
    net.train(is_training)  # 恢复net的train/eval状态
    return l_sum / n, acc_sum / n

def train(net, train_iter, valid_iter, num_epochs, lr, wd, device, lr_period,
          lr_decay):
    loss = nn.CrossEntropyLoss()
    optimizer = optim.SGD(net.fc.parameters(), lr=lr, momentum=0.9,
weight_decay=wd)
    net = net.to(device)
    for epoch in range(num_epochs):
        train_l_sum, n, start = 0.0, 0, time.time()
        if epoch > 0 and epoch % lr_period == 0:  # 每lr_period个epoch，学习率衰减
一次
            lr = lr * lr_decay
            for param_group in optimizer.param_groups:
                param_group['lr'] = lr
        for X, y in train_iter:
            X, y = X.to(device), y.to(device)
            optimizer.zero_grad()
            y_hat = net(X)
            l = loss(y_hat, y)
            l.backward()
            optimizer.step()
            train_l_sum += l.item() * y.shape[0]
            n += y.shape[0]
        time_s = "time %.2f sec" % (time.time() - start)
        if valid_iter is not None:
            valid_loss, valid_acc = evaluate_loss_acc(valid_iter, net, device)
            epoch_s = ("epoch %d, train loss %f, valid loss %f, valid acc %f, "
                       % (epoch + 1, train_l_sum / n, valid_loss, valid_acc))
        else:
            epoch_s = ("epoch %d, train loss %f, "
                       % (epoch + 1, train_l_sum / n))
        print(epoch_s + time_s + ', lr ' + str(lr))
```

调参

在train上训练，在valid上验证并调整超参数

在完整数据集上训练模型

使用上面的超参数设置，在完整数据集train_valid上训练模型

对测试集分类并提交结果

用训练好的模型对测试数据进行预测。比赛要求对测试集中的每张图片，都要预测其属于各个类别的概率。

23 - 生成对抗网络

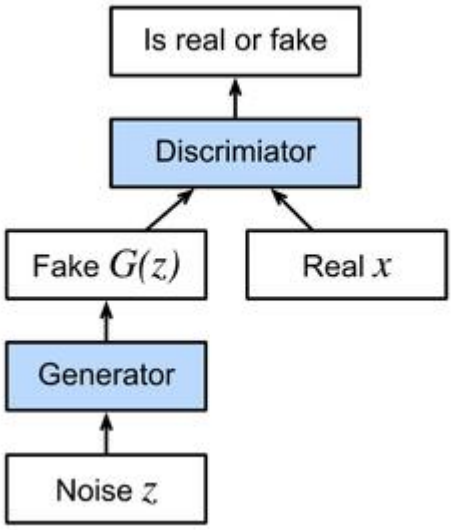
Generative Adversarial Networks

Throughout most of this book, we have talked about how to make predictions. In some form or another, we used deep neural networks learned mappings **from data points to labels**. This kind of learning is called `discriminative learning`, as in, we'd like to be able to discriminate between photos cats and photos of dogs. `Classifiers` and `regressors` are both examples of `discriminative learning`. And neural networks trained by backpropagation have upended everything we thought we knew about discriminative learning on large complicated datasets. Classification accuracies on high-res images has gone from useless to human-level (with some caveats) in just 5-6 years. We will spare you another spiel about all the other discriminative tasks where deep neural networks do astoundingly well.

But there is **more** to machine learning than just solving discriminative tasks. For example, given a large dataset, without any labels, we might want to learn a model that concisely captures the characteristics of this data. Given such a model, we could sample synthetic data points that resemble the distribution of the training data. For example, given a large corpus of photographs of faces, we might want to be able to **generate** a new photorealistic image that looks like it might plausibly have come from the same dataset. This kind of learning is called `generative modeling`.

Until recently, we had no method that could synthesize novel photorealistic images. But the success of deep neural networks for discriminative learning opened up new possibilities. One big trend over the last three years has been the application of discriminative deep nets to overcome challenges in problems that we *do not* generally think of as supervised learning problems. The recurrent neural network language models are one example of using a discriminative network (trained to predict the next character) that once trained can act as a generative model.

In 2014, a breakthrough paper introduced Generative adversarial networks (GANs) `Goodfellow.Pouget-Abadie.Mirza.ea.2014`, a clever new way to leverage the power of discriminative models to get good generative models. At their heart, GANs rely on the idea that a data generator is good if we cannot tell fake data apart from real data. In statistics, this is called a two-sample test - a test to answer the question whether datasets $X = x_1, \ldots, x_n$ and $X' = x'_1, \ldots, x'_n$ were drawn from the same distribution. The main difference between most statistics papers and GANs is that the latter use this idea in a constructive way. In other words, rather than just training a model to say "hey, these two datasets do not look like they came from the same distribution", they use the two-sample test to provide training signals to a generative model. This allows us to improve the data generator until it generates something that resembles the real data. At the very least, it needs to fool the classifier. Even if our classifier is a state of the art deep neural network.



The GAN architecture is illustrated. As you can see, there are two pieces in GAN architecture - first off, we need a device (say, a deep network but it really could be anything, such as a game rendering engine) that might potentially be able to generate data that looks just like the real thing. If we are dealing with images, this needs to generate images. If we are dealing with speech, it needs to generate audio sequences, and so on. We call this the generator network. The second component is the discriminator network. It attempts to distinguish fake and real data from each other. Both networks are in competition with each other. The generator network attempts to fool the discriminator network. At that point, the discriminator network adapts to the new fake data. This information, in turn is used to improve the generator network, and so on.

The discriminator is a binary classifier to distinguish if the input x is real (from real data) or fake (from the generator). Typically, the discriminator outputs a scalar prediction $o \in \mathbb{R}$ for input $\mathbf{x}$, such as using a dense layer with hidden size 1, and then applies sigmoid function to obtain the predicted probability $D(\mathbf{x}) = 1/(1+e^{-o})$. Assume the label y for the true data is 1 and 0 for the fake data. We train the discriminator to minimize the cross-entropy loss, *i.e.*,

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

For the generator, it first draws some parameter $\mathbf{z} \in \mathbb{R}^d$ from a source of randomness, *e.g.*, a normal distribution $\mathbf{z} \sim \mathcal{N}(0,1)$. We often call $\mathbf{z}$ as the latent variable. It then applies a function to generate $\mathbf{x}' = G(\mathbf{z})$. The goal of the generator is to fool the discriminator to classify $\mathbf{x}' = G(\mathbf{z})$ as true data, *i.e.*, we want $D(G(\mathbf{z})) \approx 1$. In other words, for a given discriminator D, we update the parameters of the generator G to maximize the cross-entropy loss when $y = 0$, *i.e.*,

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

If the discriminator does a perfect job, then $D(\mathbf{x}') \approx 0$ so the above loss near 0, which results the gradients are too small to make a good progress for the generator. So commonly we minimize the following loss:

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

which is just feed $\mathbf{x}' = G(\mathbf{z})$ into the discriminator but giving label $y = 1$.

To sum up, D and G are playing a "minimax" game with the comprehensive objective function:

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

Many of the GANs applications are in the context of images. As a demonstration purpose, we are going to content ourselves with fitting a much simpler distribution first. We will illustrate what happens if we use GANs to build the world's most inefficient estimator of parameters for a Gaussian. Let's get started.

- 判别式学习(discriminative learning): 构建从数据集`dataset (x)`到标签`label (y)`的条件概率`P(y|x)`
- 生成式学习(generative modeling): 构建数据集`dataset (x)`本身的概率分布`P(x)`，以便进一步对数据进行采样和压缩
- `GAN`的中心思想: 希望训练出能够骗过一个很好的分类器`discriminator`的生成器`generator`，通过让分类器和生成器相互对抗，最终让生成器生成更真实的数据

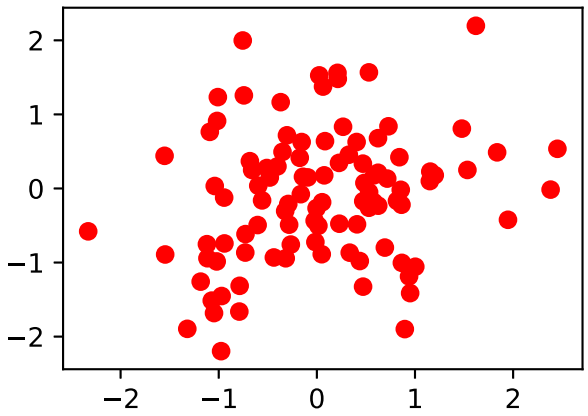
Generate some "real" data

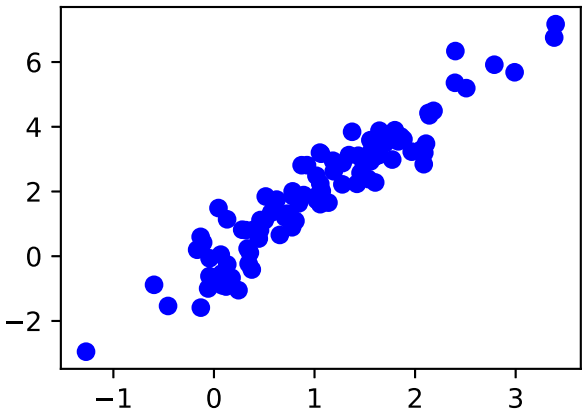
Since this is going to be the world's lamest example, we simply generate data drawn from a Gaussian.

```
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
```

Let's see what we got. This should be a Gaussian shifted in some rather arbitrary way with mean b and covariance matrix $A^T A$.

```
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]]
```

Generator

Our generator network will be the simplest network possible - a single layer linear model. This is since we will be driving that linear network with a Gaussian data generator. Hence, it literally only needs to learn the parameters to fake things perfectly.

```
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_()
```

Discriminator

For the discriminator we will be a bit more discriminating: we will use an MLP with 3 layers to make things a bit more interesting.

```
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_()
```

Training

First we define a function to update the discriminator.

```
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())
```

The generator is updated similarly. Here we reuse the cross-entropy loss but change the label of the fake data from 0 to 1 (to avoid vanishing gradient).

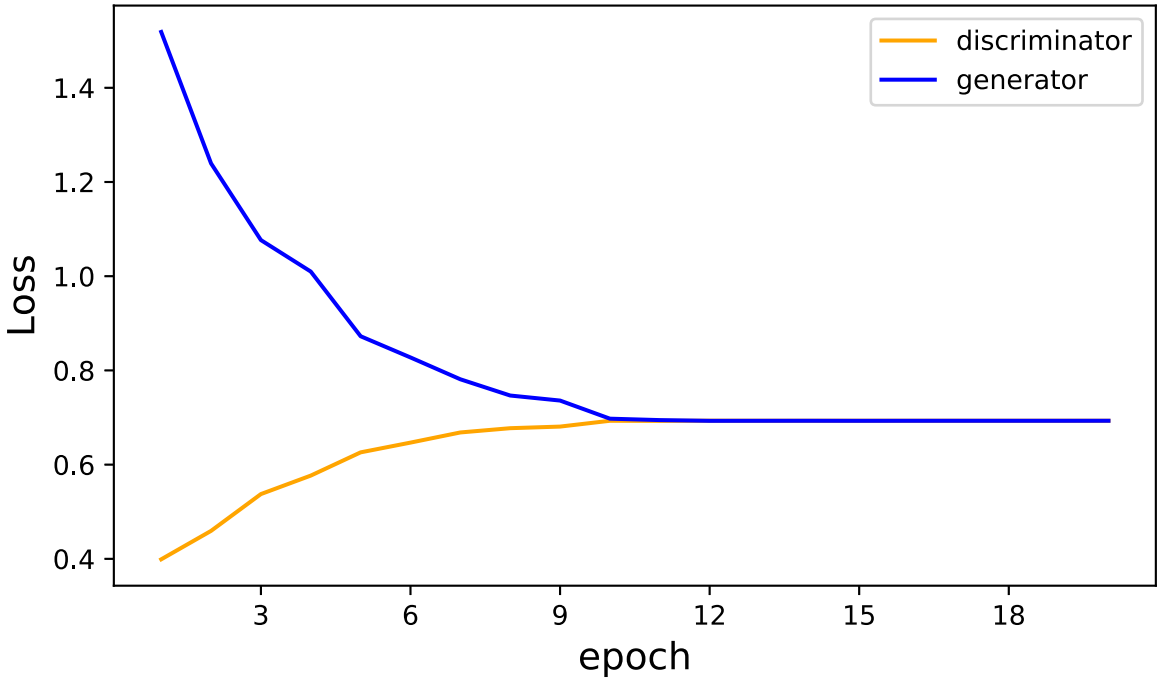
```
def update_G(Z,net_D,net_G,loss,trainer_G):
    batch_size=Z.shape[0]
    Tensor=torch.FloatTensor
    ones=Variable(Tensor(np.ones((batch_size,)))).view(batch_size,1)
    fake_X=net_G(Z)
    fake_Y=net_D(fake_X)
    loss_G=loss(fake_Y,ones)
    loss_G.backward()
    trainer_G.step()
    return float(loss_G.sum())
```

Both the discriminator and the generator performs a binary logistic regression with the cross-entropy loss. We use Adam to smooth the training process. In each iteration, we first update the discriminator and then the generator. We visualize both losses and generated examples.

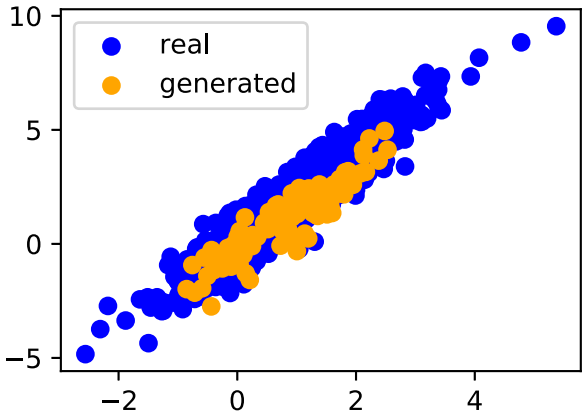
```
def train(net_D,net_G,data_iter,num_epochs,lr_D,lr_G,latent_dim,data):
    loss=nn.BCELoss()
    Tensor=torch.FloatTensor
    trainer_D=torch.optim.Adam(net_D.parameters(),lr=lr_D)
    trainer_G=torch.optim.Adam(net_G.parameters(),lr=lr_G)
    plt.figure(figsize=(7,4))
    d_loss_point=[]
    g_loss_point=[]
    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()
```

Now we specify the hyper-parameters to fit the Gaussian distribution.

```
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)
```




```
>>> 0.693149209022522 0.6932129859924316
```



Summary

- Generative adversarial networks (GANs) composes of two deep networks, the generator and the discriminator.
- The generator generates the image as much closer to the true image as possible to fool the discriminator, via maximizing the cross-entropy loss, *i.e.*, $\max \log(D(\mathbf{x}'))$.
- The discriminator tries to distinguish the generated images from the true images, via minimizing the cross-entropy loss, *i.e.*, $\min -y \log D(\mathbf{x}) - (1-y) \log(1-D(\mathbf{x}))$.

24 - DCGAN

Deep Convolutional Generative Adversarial Networks

we introduced the basic ideas behind how GANs work. We showed that they can draw samples from some simple, easy-to-sample distribution, like a uniform or normal distribution, and transform them into samples that appear to match the distribution of some dataset. And while our example of matching a 2D Gaussian distribution got the point across, it is not especially exciting.

In this section, we will demonstrate how you can use GANs to generate photorealistic images. We will be basing our models on the deep convolutional GANs (DCGAN) introduced in `Radford.Metz.Chintala.2015`. We will borrow the convolutional architecture that have proven so successful for discriminative computer vision problems and show how via GANs, they can be leveraged to generate photorealistic images.

The Pokemon Dataset

The dataset we will use is a collection of Pokemon sprites obtained from pokemondb. First download, extract and load this dataset.
We resize each image into 64×64. The `ToTensor` transformation will project the pixel value into $[0, 1]$, while our generator will use the tanh function to obtain outputs in $[-1, 1]$. Therefore we normalize the data with 0.5 mean and 0.5 standard deviation to match the value range.

```
transform = transforms.Compose([
    transforms.Resize((64,64)),
    transforms.ToTensor(),
    transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5))
])
```

Let's visualize the first 20 images.

The Generator

The generator needs to map the noise variable $\mathbf{z} \in \mathbb{R}^d$, a length-$d$ vector, to a RGB image with width and height to be 64×64 . In `sec_fcn` we introduced the fully convolutional network that uses transposed convolution layer (refer to `sec_transposed_conv`) to enlarge input size. The basic block of the generator contains a transposed convolution layer followed by the batch normalization and ReLU activation.

```
class G_block(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size=4,strides=2,
padding=1):
        super(G_block,self).__init__()
        self.conv2d_trans=nn.ConvTranspose2d(in_channels, out_channels,
kernel_size=kernel_size,
                                             stride=strides, padding=padding,
bias=False)
        self.batch_norm=nn.BatchNorm2d(out_channels,0.8)
        self.activation=nn.ReLU()
    def forward(self,x):
        return self.activation(self.batch_norm(self.conv2d_trans(x)))
```

In default, the transposed convolution layer uses a $k_h = k_w = 4$ kernel, a $s_h = s_w = 2$ strides, and a $p_h = p_w = 1$ padding. With a input shape of $n_h^{'} \times n_w^{'} = 16 \times 16$, the generator block will double input's width and height.

$$n_h^{'} \times n_w^{'} = [(n_h k_h - (n_h - 1)(k_h - s_h) - 2p_h] \times [(n_w k_w - (n_w - 1)(k_w - s_w) - 2p_w]$$

$$= [(k_h + s_h(n_h - 1) - 2p_h] \times [(k_w + s_w(n_w - 1) - 2p_w]$$

$$= [(4 + 2 \times (16 - 1) - 2 \times 1] \times [(4 + 2 \times (16 - 1) - 2 \times 1] = 32 \times 32.$$

```
Tensor=torch.cuda.FloatTensor
x=Variable(Tensor(np.zeros((2,3,16,16))))
g_blk=G_block(3,20)
g_blk.cuda()
print(g_blk(x).shape)
>>> torch.Size([2, 20, 32, 32])
```

If changing the transposed convolution layer to a 4×4 kernel, 1×1 strides and zero padding. With a input size of 1×1, the output will have its width and height increased by 3 respectively.

```
x=Variable(Tensor(np.zeros((2,3,1,1))))
g_blk=G_block(3,20,strides=1,padding=0)
g_blk.cuda()
print(g_blk(x).shape)
>>> torch.Size([2, 20, 4, 4])
```

The generator consists of four basic blocks that increase input's both width and height from 1 to 32. At the same time, it first projects the latent variable into 64×8 channels, and then halve the channels each time. At last, a transposed convolution layer is used to generate the output. It further doubles the width and height to match the desired 64×64 shape, and reduces the channel size to 3. The tanh activation function is applied to project output values into the $(-1, 1)$ range.

```
class net_G(nn.Module):
    def __init__(self,in_channels):
        super(net_G,self).__init__()

        n_G=64
        self.model=nn.Sequential(
            G_block(in_channels,n_G*8,strides=1,padding=0),
            G_block(n_G*8,n_G*4),
            G_block(n_G*4,n_G*2),
            G_block(n_G*2,n_G),
            nn.ConvTranspose2d(
                n_G,3,kernel_size=4,stride=2,padding=1,bias=False
            ),
            nn.Tanh()
        )
    def forward(self,x):
        x=self.model(x)
        return x


def weights_init_normal(m):
    classname = m.__class__.__name__
    if classname.find("Conv") != -1:
        torch.nn.init.normal_(m.weight.data, mean=0, std=0.02)
    elif classname.find("BatchNorm2d") != -1:
        torch.nn.init.normal_(m.weight.data, mean=1.0, std=0.02)
        torch.nn.init.constant_(m.bias.data, 0.0)
```

Generate a 100 dimensional latent variable to verify the generator's output shape.

```
x=Variable(Tensor(np.zeros((1,100,1,1))))
generator=net_G(100)
generator.cuda()
generator.apply(weights_init_normal)
print(generator(x).shape)
>>> torch.Size([1, 3, 64, 64])
```

The Discriminator

The discriminator is a normal convolutional network network except that it uses a `leaky ReLU` as its activation function. Given $\alpha \in [0,1]$, its definition is

$$\text{leaky ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{otherwise} \end{cases}.$$

As it can be seen, it is `normal ReLU` if $\alpha = 0$, and an identity function if $\alpha = 1$. For $\alpha \in (0, 1)$, `leaky ReLU` is a nonlinear function that give a non-zero output for a negative input. It aims to fix the `"dying ReLU"` problem that a neuron might always output a negative value and therefore cannot make any progress since the gradient of `ReLU` is 0.

The basic block of the discriminator is a `convolution layer` followed by a `batch normalization` layer and a `leaky ReLU` activation. The hyper-parameters of the convolution layer are similar to the transpose convolution layer in the generator block.

```
class D_block(nn.Module):
    def __init__(self,in_channels,out_channels,kernel_size=4,strides=2,
                 padding=1,alpha=0.2):
        super(D_block,self).__init__()

self.conv2d=nn.Conv2d(in_channels,out_channels,kernel_size,strides,padding,bias=Fa
lse)
        self.batch_norm=nn.BatchNorm2d(out_channels,0.8)
        self.activation=nn.LeakyReLU(alpha)
    def forward(self,X):
        return self.activation(self.batch_norm(self.conv2d(X)))
```

A basic block with default settings will halve the width and height of the inputs, as we demonstrated in `sec_padding`. For example, given a input shape $n_h = n_w = 16$, with a kernel shape $k_h = k_w = 4$, a stride shape $s_h = s_w = 2$, and a padding shape $p_h = p_w = 1$, the output shape will be:

$$n_h' \times n_w' = \lfloor (n_h - k_h + 2p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + 2p_w + s_w)/s_w \rfloor$$

$$= \lfloor (16 - 4 + 2 \times 1 + 2)/2 \rfloor \times \lfloor (16 - 4 + 2 \times 1 + 2)/2 \rfloor = 8 \times 8.$$

```
x = Variable(Tensor(np.zeros((2, 3, 16, 16))))
d_blk = D_block(3,20)
d_blk.cuda()
print(d_blk(x).shape)
>>> torch.Size([2, 20, 8, 8])
```

The discriminator is a mirror of the generator.

```
class net_D(nn.Module):
    def __init__(self,in_channels):
        super(net_D,self).__init__()
        n_D=64
        self.model=nn.Sequential(
            D_block(in_channels,n_D),
            D_block(n_D,n_D*2),
            D_block(n_D*2,n_D*4),
            D_block(n_D*4,n_D*8)
        )
```

```
        self.conv=nn.Conv2d(n_D*8,1,kernel_size=4,bias=False)
        self.activation=nn.Sigmoid()
        # self._initialize_weights()
    def forward(self,x):
        x=self.model(x)
        x=self.conv(x)
        x=self.activation(x)
        return x
```

It uses a convolution layer with output channel 1 as the last layer to obtain a single prediction value.

```
x = Variable(Tensor(np.zeros((1, 3, 64, 64))))
discriminator=net_D(3)
discriminator.cuda()
discriminator.apply(weights_init_normal)
print(discriminator(x).shape)
>>> torch.Size([1, 1, 1, 1])
```

Training

Compared to the basic GAN in `sec_basic_gan`, we use the same learning rate for both `generator` and `discriminator` since they are similar to each other. In addition, we change β_1 in `Adam` from 0.9 to 0.5. It decreases the smoothness of the momentum, the exponentially weighted moving average of past gradients, to take care of the rapid changing gradients because the `generator` and the `discriminator` fight with each other. Besides, the random generated noise `Z`, is a `4-D` tensor and we are using GPU to accelerate the computation.

```
def update_D(X,Z,net_D,net_G,loss,trainer_D):
    batch_size=X.shape[0]
    Tensor=torch.cuda.FloatTensor

ones=Variable(Tensor(np.ones(batch_size,)),requires_grad=False).view(batch_size,1)
    zeros =
Variable(Tensor(np.zeros(batch_size,)),requires_grad=False).view(batch_size,1)
    real_Y=net_D(X).view(batch_size,-1)
    fake_X=net_G(Z)
    fake_Y=net_D(fake_X).view(batch_size,-1)
    loss_D=(loss(real_Y,ones)+loss(fake_Y,zeros))/2
    loss_D.backward()
    trainer_D.step()
    return float(loss_D.sum())

def update_G(Z,net_D,net_G,loss,trainer_G):
    batch_size=Z.shape[0]
    Tensor=torch.cuda.FloatTensor

ones=Variable(Tensor(np.ones((batch_size,))),requires_grad=False).view(batch_size,
1)
```

```
    fake_X=net_G(Z)
    fake_Y=net_D(fake_X).view(batch_size,-1)
    loss_G=loss(fake_Y,ones)
    loss_G.backward()
    trainer_G.step()
    return float(loss_G.sum())


def train(net_D,net_G,data_iter,num_epochs,lr,latent_dim):
    loss=nn.BCELoss()
    Tensor=torch.cuda.FloatTensor
    trainer_D=torch.optim.Adam(net_D.parameters(),lr=lr,betas=(0.5,0.999))
    trainer_G=torch.optim.Adam(net_G.parameters(),lr=lr,betas=(0.5,0.999))
    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:
            X=X[:][0]
            batch+=1
            X=Variable(X.type(Tensor))
            batch_size=X.shape[0]
            Z=Variable(Tensor(np.random.normal(0,1,(batch_size,latent_dim,1,1))))

            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)
        print(
            "[Epoch %d/%d]  [D loss: %f] [G loss: %f]"
            % (epoch, num_epochs,  d_loss_sum/batch_size,  g_loss_sum/batch_size)
        )


    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=(21, latent_dim, 1,
1))),requires_grad=False)
    fake_x = generator(Z)
    fake_x=fake_x.cpu().detach().numpy()
    plt.figure(figsize=(14,6))
    for i in range(21):
        im=np.transpose(fake_x[i])
        plt.subplot(3,7,i+1)
        plt.imshow(im)
    plt.show()
```

Now let's train the model.

```
if __name__ == '__main__':
    lr,latent_dim,num_epochs=0.005,100,50
    train(discriminator,generator,data_iter,num_epochs,lr,latent_dim)
```

Summary

- DCGAN architecture has four convolutional layers for the `Discriminator` and four "fractionally-strided" convolutional layers for the `Generator`.
- The `Discriminator` is a 4-layer strided convolutions with batch normalization (except its input layer) and `leaky ReLU` activations.
- `Leaky ReLU` is a nonlinear function that give a non-zero output for a negative input. It aims to fix the `“dying ReLU”` problem and helps the gradients flow easier through the architecture.