

数据增强

常用的数据增强方法

常用的数据增强方法如下：

- 1.对图片进行一定比例缩放
- 2.对图片进行随机位置的截取
- 3.对图片进行随机的水平和竖直翻转
- 4.对图片进行随机角度的旋转
- 5.对图片进行亮度、对比度和颜色的随机变化

这些方法 pytorch 都已经为我们内置在了 torchvision 里面，我们在安装 pytorch 的时候也安装了 torchvision，下面我们来依次展示一下这些数据增强方法

```
from PIL import Image
from torchvision import transforms as tfs
```

```
# 读入一张图片
im = Image.open('./cat.png')
im
```

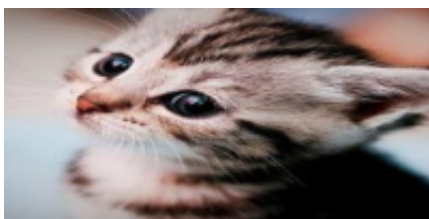


随机比例放缩

随机比例缩放主要使用的是 `torchvision.transforms.Resize()` 这个函数，第一个参数可以是一个整数，那么图片会保存现在的宽和高的比例，并将更短的边缩放到这个整数的大小，第一个参数也可以是一个 tuple，那么图片会直接把宽和高缩放到这个大小；第二个参数表示放缩图片使用的方法，比如最邻近法，或者双线性差值等，一般双线性差值能够保留图片更多的信息，所以 pytorch 默认使用的是双线性差值，你可以手动去改这个参数，更多的信息可以看看[文档](#)

```
# 比例缩放
print('before scale, shape: {}'.format(im.size))
new_im = tfs.Resize((100, 200))(im)
print('after scale, shape: {}'.format(new_im.size))
new_im
```

```
before scale, shape: (224, 224)
after scale, shape: (200, 100)
```



随机位置截取

随机位置截取能够提取出图片中局部的信息，使得网络接受的输入具有多尺度的特征，所以能够有较好的效果。在 torchvision 中主要有下面两种方式，一个是 `torchvision.transforms.RandomCrop()`，传入的参数就是截取出的图片的长和宽，对图片在随机位置进行截取；第二个是

`torchvision.transforms.CenterCrop()`，同样传入截取出的图片的大小作为参数，会在图片的中心进行截取

```
# 随机裁剪出 100 x 100 的区域
random_im1 = tfs.RandomCrop(100)(im)
random_im1
```



随机裁剪出 150 x 100 的区域

```
random_im2 = tfs.RandomCrop((150, 100))(im)  
random_im2
```



中心裁剪出 100 x 100 的区域

```
center_im = tfs.CenterCrop(100)(im)  
center_im
```



随机的水平和竖直方向翻转

对于上面这一张猫的图片，如果我们将它翻转一下，它仍然是一张猫，但是图片就有了更多的多样性，所以随机翻转也是一种非常有效的手段。在 `torchvision` 中，随机翻转使用的是

`torchvision.transforms.RandomHorizontalFlip()` 和

`torchvision.transforms.RandomVerticalFlip()`

```
# 随机水平翻转
```

```
h_flip = tfs.RandomHorizontalFlip()(im)
```

```
h_flip
```



```
# 随机竖直翻转
```

```
v_flip = tfs.RandomVerticalFlip()(im)
```

```
v_flip
```



随机角度旋转

一些角度的旋转仍然是非常有用的数据增强方式，在 torchvision 中，使用

`torchvision.transforms.RandomRotation()` 来实现，其中第一个参数就是随机旋转的角度，比如填入 10，那么每次图片就会在 -10 ~ 10 度之间随机旋转

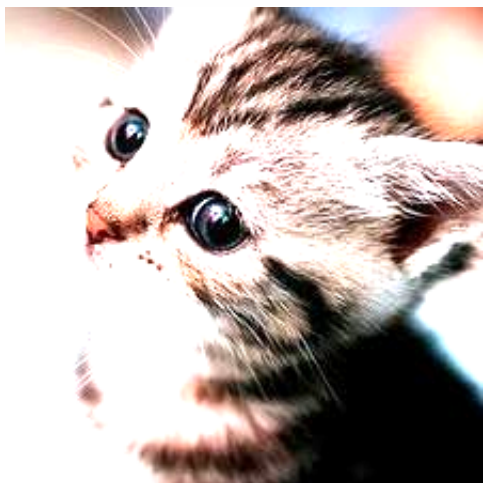
```
rot_im = tfs.RandomRotation(45)(im)
rot_im
```



亮度、对比度和颜色的变化

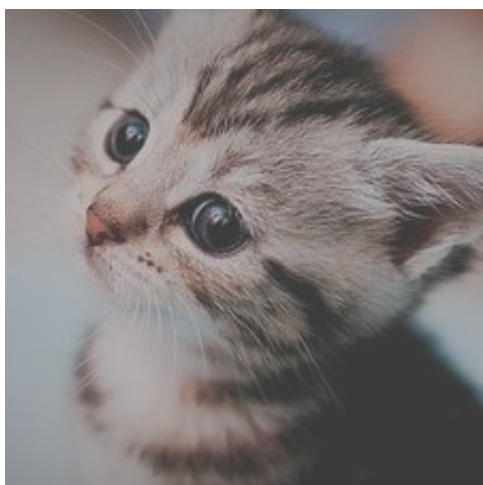
除了形状变化外，颜色变化又是另外一种增强方式，其中可以设置亮度变化，对比度变化和颜色变化等，在 torchvision 中主要使用 `torchvision.transforms.ColorJitter()` 来实现的，第一个参数就是亮度的比例，第二个是对比度，第三个是饱和度，第四个是颜色

```
# 亮度
bright_im = tfs.ColorJitter(brightness=1)(im) # 随机从 0 ~ 2 之间亮度变化，1 表示原图
bright_im
```



对比度

```
contrast_im = tfs.ColorJitter(contrast=1)(im) # 随机从 0 ~ 2 之间对比度变化, 1 表示原图  
contrast_im
```



颜色

```
color_im = tfs.ColorJitter(hue=0.5)(im) # 随机从 -0.5 ~ 0.5 之间对颜色变化  
color_im
```

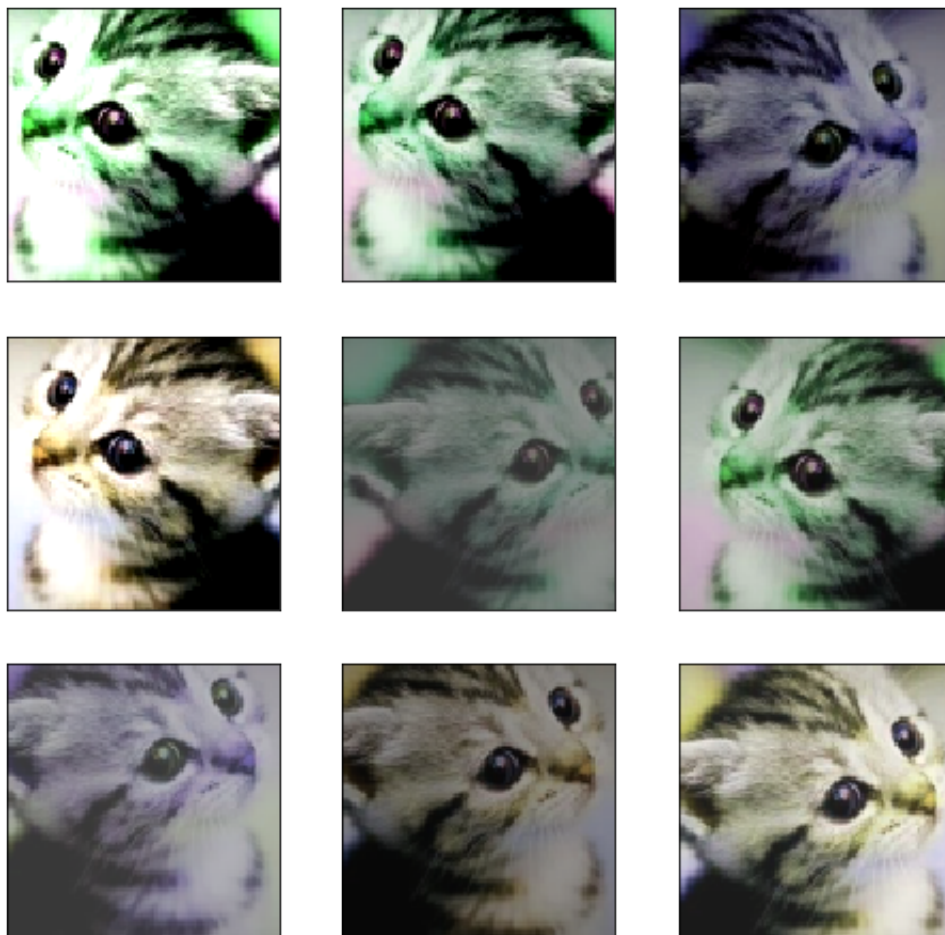


上面我们讲了这么多图片增强的方法，其实这些方法都不是孤立起来用的，可以联合起来用，比如先做随机翻转，然后随机截取，再做对比度增强等等，torchvision 里面有个非常方便的函数能够将这些变化合起来，就是 `torchvision.transforms.Compose()`，下面我们举个例子

```
im_aug = tfs.Compose([
    tfs.Resize(120),
    tfs.RandomHorizontalFlip(),
    tfs.RandomCrop(96),
    tfs.ColorJitter(brightness=0.5, contrast=0.5, hue=0.5)
])
```

```
import matplotlib.pyplot as plt
%matplotlib inline
```

```
nrows = 3
ncols = 3
figsize = (8, 8)
_, figs = plt.subplots(nrows, ncols, figsize=figsize)
for i in range(nrows):
    for j in range(ncols):
        figs[i][j].imshow(im_aug(im))
        figs[i][j].axes.get_xaxis().set_visible(False)
        figs[i][j].axes.get_yaxis().set_visible(False)
plt.show()
```

可以看到每次做完增强之后的图片都有一些变化，所以这就是我们前面讲的，增加了一些'新'数据

下面我们使用图像增强进行训练网络，看看具体的提升究竟在什么地方，使用前面讲的 ResNet 进行训练

```
import numpy as np
import torch
from torch import nn
import torch.nn.functional as F
from torch.autograd import Variable
from torchvision.datasets import CIFAR10
from utils import train, resnet
from torchvision import transforms as tfs
```

使用数据增强

```
def train_tf(x):
    im_aug = tfs.Compose([
        tfs.Resize(120),
        tfs.RandomHorizontalFlip(),
        tfs.RandomCrop(96),
        tfs.ColorJitter(brightness=0.5, contrast=0.5, hue=0.5),
```



```

        tfs.ToTensor(),
        tfs.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])
    ])
    x = im_aug(x)
    return x

def test_tf(x):
    im_aug = tfs.Compose([
        tfs.Resize(96),
        tfs.ToTensor(),
        tfs.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])
    ])
    x = im_aug(x)
    return x

train_set = CIFAR10('./data', train=True, transform=train_tf)
train_data = torch.utils.data.DataLoader(train_set, batch_size=64, shuffle=True)
test_set = CIFAR10('./data', train=False, transform=test_tf)
test_data = torch.utils.data.DataLoader(test_set, batch_size=128, shuffle=False)

net = resnet(3, 10)
optimizer = torch.optim.SGD(net.parameters(), lr=0.01)
criterion = nn.CrossEntropyLoss()

```

```

train(net, train_data, test_data, 10, optimizer, criterion)

```

```
Epoch 0. Train Loss: 1.846885, Train Acc: 0.323370, Valid Loss: 2.031208, Valid Acc: 0.348101, Time 00:00:27
Epoch 1. Train Loss: 1.421866, Train Acc: 0.493127, Valid Loss: 1.635981, Valid Acc: 0.448675, Time 00:00:28
Epoch 2. Train Loss: 1.213214, Train Acc: 0.571232, Valid Loss: 1.435900, Valid Acc: 0.509494, Time 00:00:27
Epoch 3. Train Loss: 1.068615, Train Acc: 0.624680, Valid Loss: 1.198399, Valid Acc: 0.589695, Time 00:00:27
Epoch 4. Train Loss: 0.966057, Train Acc: 0.665082, Valid Loss: 1.374483, Valid Acc: 0.556566, Time 00:00:27
Epoch 5. Train Loss: 0.881177, Train Acc: 0.693274, Valid Loss: 0.936225, Valid Acc: 0.675138, Time 00:00:28
Epoch 6. Train Loss: 0.817892, Train Acc: 0.716093, Valid Loss: 1.524335, Valid Acc: 0.553006, Time 00:00:27
Epoch 7. Train Loss: 0.775561, Train Acc: 0.729799, Valid Loss: 1.144188, Valid Acc: 0.630439, Time 00:00:28
Epoch 8. Train Loss: 0.728689, Train Acc: 0.746324, Valid Loss: 1.020422, Valid Acc: 0.666634, Time 00:00:28
Epoch 9. Train Loss: 0.691437, Train Acc: 0.759930, Valid Loss: 0.880125, Valid Acc: 0.714893, Time 00:00:28
```

不使用数据增强

```
def data_tf(x):
    im_aug = tfs.Compose([
        tfs.Resize(96),
        tfs.ToTensor(),
        tfs.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])
    ])
    x = im_aug(x)
    return x

train_set = CIFAR10('./data', train=True, transform=data_tf)
train_data = torch.utils.data.DataLoader(train_set, batch_size=64, shuffle=True)
test_set = CIFAR10('./data', train=False, transform=data_tf)
test_data = torch.utils.data.DataLoader(test_set, batch_size=128, shuffle=False)

net = resnet(3, 10)
optimizer = torch.optim.SGD(net.parameters(), lr=0.01)
criterion = nn.CrossEntropyLoss()
```

```
train(net, train_data, test_data, 10, optimizer, criterion)
```

```
Epoch 0. Train Loss: 1.450800, Train Acc: 0.466952, Valid Loss: 1.533483, Valid Acc: 0.471519, Time 00:00:25
Epoch 1. Train Loss: 1.007298, Train Acc: 0.641964, Valid Loss: 1.553861, Valid Acc: 0.483287, Time 00:00:27
Epoch 2. Train Loss: 0.790772, Train Acc: 0.721308, Valid Loss: 1.093117, Valid Acc: 0.623714, Time 00:00:27
Epoch 3. Train Loss: 0.626106, Train Acc: 0.781770, Valid Loss: 1.290194, Valid Acc: 0.579015, Time 00:00:27
Epoch 4. Train Loss: 0.499911, Train Acc: 0.830862, Valid Loss: 0.973577, Valid Acc: 0.680281, Time 00:00:27
Epoch 5. Train Loss: 0.389616, Train Acc: 0.865769, Valid Loss: 1.013823, Valid Acc: 0.685918, Time 00:00:27
Epoch 6. Train Loss: 0.286867, Train Acc: 0.903652, Valid Loss: 0.951596, Valid Acc: 0.708366, Time 00:00:27
Epoch 7. Train Loss: 0.210318, Train Acc: 0.929468, Valid Loss: 1.031382, Valid Acc: 0.706487, Time 00:00:27
Epoch 8. Train Loss: 0.153977, Train Acc: 0.949329, Valid Loss: 2.008988, Valid Acc: 0.548655, Time 00:00:27
Epoch 9. Train Loss: 0.122304, Train Acc: 0.959319, Valid Loss: 1.463229, Valid Acc: 0.673259, Time 00:00:27
```

从上面可以看出，对于训练集，不做数据增强跑 10 次，准确率已经到了 95%，而使用了数据增强，跑 10 次准确率只有 75%，说明数据增强之后变得更难了。

而对于测试集，使用数据增强进行训练的时候，准确率会比不使用更高，因为数据增强提高了模型应对于更多的不同数据集的泛化能力，所以有更好的效果。