华东理工大学

模式识别大作业

题目	Dogs vs. Cats
院系	信息科学与工程学院
专业	控制科学与工程
组员	涂远来
指导老师	赵海涛

Dogs vs. Cats

一、Dogs vs. Cats 比赛简介

猫与狗大赛(Dogs vs. Cats)来自 Kaggel 上的一个比赛，建立一个算法来区分数据集中的猫与狗，数据集是 Asirra(Animal Species Image Recognition for Restricting Access)数据集的一个子集，当时在此数据集上表现的很好的算法是《Machine Learning Attacks Against the Asirra CAPTCHA》[1]文章中所用的方法，准确率在 80%左右。在本次比赛中，我们需要在此基础上做到更精确的对此数据集进行分类。

二、整体解决方案

很明显这个分类问题是个二分类问题，并且数据集是以图片的形式给出，自然而然的想到神经网络中的卷积神经网络在图片识别上的优势，并且随着迁移学习的发展，很多大型的网络并且适用的网络模型不在需要重新训练，可以通过迁移学习的方法对已有的网络进行微调后再使用。

本次实验既是通过迁移模型的方法，修改模型结构后对已有的数据集进行再训练，并且反复修改超参数来反复调节模型，最终达到一个较好的分类效果。

2.1 数据的准备

数据从 kaggel 此处下载（https://www.kaggle.com/c/dogs-vs-cats/data），数据包括训练集 train.zip,和测试集 test.zip,训练集中包括 25000 张图片，内容如下：

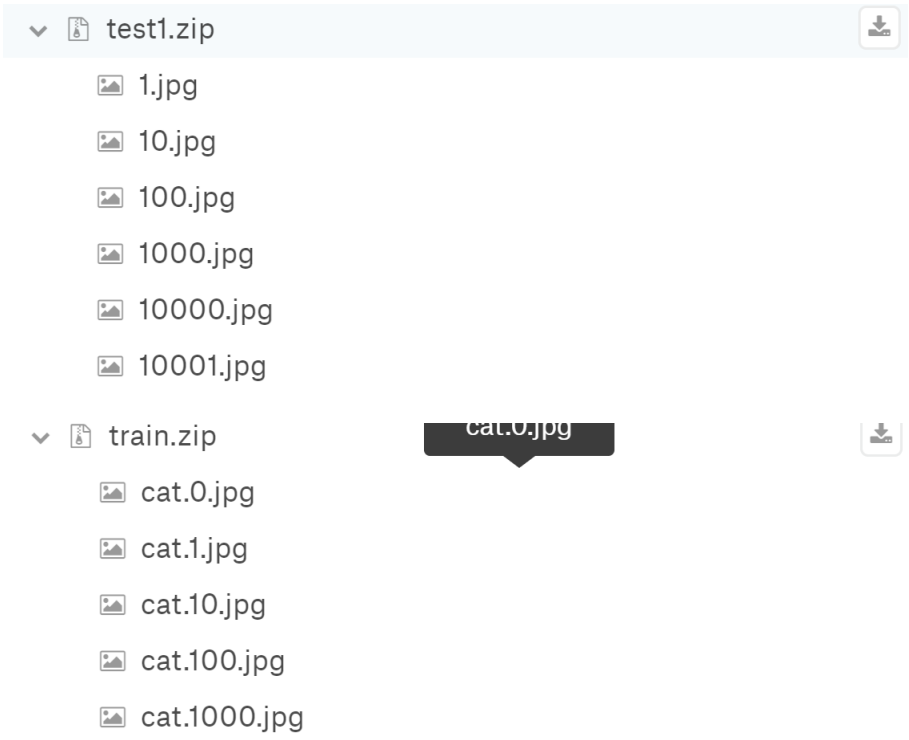

图 1 数据集信息

其中 train 中包含了 25000 张图片，12500 张是带有的猫的图片，另外 12500 张是狗的图片。

此次训练中，将训练的数据集分为两个子数据集："train"和"val"，训练集与验证集，为了快速的检验模型的效果，我们将训练集设置为每个种类 5000 张图片，并且预留每个种类 1000 张图片作为验证集。

使用 jupyter nootbook 进行编程，首先准备好所需要的数据：

将数据存入 data/PetImages/根目入下：

```
In [1]: import os
        import shutil
        import re

        base_dir = "data/PetImages/"

        files = os.listdir(base_dir)
```

设置好文件的存放形式：

In [2]:
```
def train_maker(name):
  train_dir = f"{base_dir}/train/{name}"
  for f in files:
        search_object = re.search(name, f)
        if search_object:
          shutil.move(f'{base_dir}/{name}', train_dir)

train_maker("Cat")
train_maker("Dog")



cat_train = base_dir + "train/Cat/"
cat_val = base_dir + "val/Cat/"
dog_train = base_dir + "train/Dog/"
dog_val = base_dir + "val/Dog/"

cat_files = os.listdir(cat_train)
dog_files = os.listdir(dog_train)


for f in cat_files:
    validationCatsSearch0bj = re.search("5\d\d\d", f)
    if validationCatsSearch0bj:
        shutil.move(f'{cat_train}/{f}', cat_val)

for f in dog_files:
    validationCatsSearch0bj = re.search("5\d\d\d", f)
    if validationCatsSearch0bj:
        shutil.move(f'{dog_train}/{f}', dog_val)
```

具体数据的存放结构为：

- data
  - PetImages
    - train
      - Cat
      - Dog
    - val
      - Cat
      - Dog

2.2 数据的加载

在准备完数据之后，我们先将必要的数据哭导入环境中，这里我们使用的是 pytorch 框架，我们需要很多 torch 包例如 nn 神经网络包，optimizers 优化器，以及 DataLoards 数据加载器等。同时使用 matplotlib 包去可视化数据。Numpy 用来处理数据的计算：

In [3]:
```
from __future__ import print_function, division

import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
import torchvision
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt
import numpy as np
import time
import os
import copy
```

接下来我们需要加载我们的训练数据为神经网络的训练做准备，利用 pytorch 的 trainsforms 功能，并且对数据进行处理，必须使训练输入的数据大小保持一致，使用到 trainsforms.Resize 。并且将数据进行变换进而达到扩大数据集的效果，变换包括随机的旋转以及剪裁。将图像转换成张量的形式有利于数据的处理以及运算，这里只只将训练集进行变换操作，并不需要对验证集进行扩充。

In [4]:
```
mean_nums = [0.485, 0.456, 0.406]
std_nums = [0.229, 0.224, 0.225]

chosen_transforms = {'train': transforms.Compose([
        transforms.RandomResizedCrop(size=256),
        transforms.RandomRotation(degrees=15),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize(mean_nums, std_nums)
]), 'val': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean_nums, std_nums)
]),
}
```

此时开始进入数据的导入并且设置好导入路径：

In [5]:
```
data_dir = 'data/PetImages/'

chosen_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x),
  chosen_transforms[x])
                   for x in ['train', 'val']}
```

将数据的路径准备好之后，用 DataLoaders 创建一个用于训练迭代的对象，并且设置好小批量的大小，这里的 batch_size 设置为 4，并且将数据打乱 shuffle=Ture。

In [6]:
```
dataloaders = {x: torch.utils.data.DataLoader(chosen_datasets[x], batch_size=4,
  shuffle=True, num_workers=4)
               for x in ['train', 'val']}
```

由于数据对象的格式为图片，并且转换成张量，用 GPU 来运算将会是计算速度大大加快，这里选用的 GPU 型号为 GTX970M，之后的计算时间是基于此设备的。这里将数据打包送给 GPU：

```
In [7]: dataset_sizes = {x: len(chosen_datasets[x]) for x in ['train', 'val']}
class_names = chosen_datasets['train'].classes

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

现在用一个函数来可视化一些图像。取一个输入，创建一个 Numpy 数组，然后转置。然后将使用平均值和标准差对输入进行标准化。最后，我们将值变化为 0，1 之间，然后显示图像:

```
In [8]: def imshow(inp, title=None):
    inp = inp.numpy().transpose((1, 2, 0))
    mean = np.array([mean_nums])
    std = np.array([std_nums])
    inp = std * inp + mean
    inp = np.clip(inp, 0, 1)
    plt.imshow(inp)
    if title is not None:
        plt.title(title)
    plt.pause(0.001)
```

一个小批量里的数据如下效果，已经进行过变换操作：

```
In [9]: inputs, classes = next(iter(dataloaders['train']))

out = torchvision.utils.make_grid(inputs)

imshow(out, title=[class_names[x] for x in classes])
```

2.3 建立一个预训练模型

建立一个迁移学习的预训练模型，也就是已经训练好的模型，这里选用的是 resnet34，按照其原本的权重，重置它的全连接层，使得符合本竞赛中二分类的要求：

```
In [10]: res_mod = models.resnet34(pretrained=True)

num_ftrs = res_mod.fc.in_features
res_mod.fc = nn.Linear(num_ftrs, 2)
```

最后模型的结构被修改为：

```
In [11]: for name, child in res_mod.named_children():
             print(name)

conv1
bn1
relu
maxpool
layer1
layer2
layer3
layer4
avgpool
fc
```

实际上我们只是将模型的结构修改成所需要的样子，但是并未对其中权重参数做修改。

现在将模型发送到训练设备 GPU 上，并且选择损失函数以及优化器，选择的损失函数是 cross – sentropyloss，选择的是 SGD 优化器，这里还有其它的选择，这只是常用的两种，并且结果表明训练效果很好。

并且选择一个学习速率的调节器，在优化器迭代中调整学习速率，有效的防止了因为学习速率过大而不收敛的情况：

```
In [12]: res_mod = res_mod.to(device)
         criterion = nn.CrossEntropyLoss()

         optimizer_ft = optim.SGD(res_mod.parameters(), lr=0.001, momentum=0.9)

         exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=7, gamma=0.1)
```

接下来只需要定义训练模型和可视化预测的函数，从训练开，使用选择好的模型以及选择的优化器等。设置好迭代周期数，每一个周期都有训练和验证，训练阶段需要完成的任务是：

1. 调整学习率
2. 初始化梯度，初始化梯度为零
3. 执行前向传播
4. 计算损失函数
5. 使用优化器进行反向传播和更新权重

并且在训练和验证的过程中进行跟踪，将最好的结果保存下来。

2.4 训练模型

定义一个训练函数：

In [13]:
```
def train_model(model, criterion, optimizer, scheduler, num_epochs=10):
    since = time.time()

    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0

    for epoch in range(num_epochs):
        print('Epoch {}/{}'.format(epoch, num_epochs - 1))
        print('-' * 10)

        for phase in ['train', 'val']:
            if phase == 'train':
                scheduler.step()
                model.train()
            else:
                model.eval()

            current_loss = 0.0
            current_corrects = 0

            print('Iterating through data...')
```

将每个标签以及输入给到 GPU，进行前向传播以及反向传播：

```
            for inputs, labels in dataloaders[phase]:
                inputs = inputs.to(device)
                labels = labels.to(device)

                optimizer.zero_grad()

                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)

                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                current_loss += loss.item() * inputs.size(0)
                current_corrects += torch.sum(preds == labels.data)

            epoch_loss = current_loss / dataset_sizes[phase]
            epoch_acc = current_corrects.double() / dataset_sizes[phase]

            print('{} Loss: {:.4f} Acc: {:.4f}'.format(
                phase, epoch_loss, epoch_acc))

            if phase == 'val' and epoch_acc > best_acc:
                best_acc = epoch_acc
                best_model_wts = copy.deepcopy(model.state_dict())

        print()
```

输出计算时间，以及保存最好的验证结果：

```
    time_since = time.time() - since
    print('Training complete in {:.0f}m {:.0f}s'.format(
        time_since // 60, time_since % 60))
    print('Best val Acc: {:4f}'.format(best_acc))

    model.load_state_dict(best_model_wts)
    return model
```

定义一个可视化函数，对结果进行可视化：

In [14]:
```
def visualize_model(model, num_images=6):
    was_training = model.training
    model.eval()
    images_handeled = 0
    fig = plt.figure()

    with torch.no_grad():
        for i, (inputs, labels) in enumerate(dataloaders['val']):
            inputs = inputs.to(device)
            labels = labels.to(device)

            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)

            for j in range(inputs.size()[0]):
                images_handeled += 1
                ax = plt.subplot(num_images//2, 2, images_handeled)
                ax.axis('off')
                ax.set_title('predicted: {}'.format(class_names[preds[j]]))
                imshow(inputs.cpu().data[j])

                if images_handeled == num_images:
                    model.train(mode=was_training)
                    return
        model.train(mode=was_training)
```

最后调用训练函数以及可视化函数，设置 3 个周期，结果如下：

In [15]:
```
base_model = train_model(res_mod, criterion, optimizer_ft, exp_lr_scheduler, num_epochs=3)
visualize_model(base_model)
plt.show()
```

```
Epoch 0/2
----------

c:\programdata\miniconda3\lib\site-packages\torch\optim\lr_scheduler.py:100: UserWarning: Det
imizer.step()`. In PyTorch 1.1.0 and later, you should call them in the opposite order: `opti
ilure to do this will result in PyTorch skipping the first value of the learning rate schedul
stable/optim.html#how-to-adjust-learning-rate
  "https://pytorch.org/docs/stable/optim.html#how-to-adjust-learning-rate", UserWarning)

Iterating through data...
train Loss: 0.3808 Acc: 0.8362
Iterating through data...
val Loss: 0.0624 Acc: 0.9770

Epoch 1/2
----------
Iterating through data...
train Loss: 0.2293 Acc: 0.8990
Iterating through data...
val Loss: 0.0662 Acc: 0.9685

Epoch 2/2
----------
Iterating through data...
train Loss: 0.1999 Acc: 0.9111
Iterating through data...
val Loss: 0.1249 Acc: 0.9595

Training complete in 22m 9s
Best val Acc: 0.977000
```

从训练结果可以看出，最好的验证集准确率在 97.7%，远远高于在前面提到的 80%，卷积神经网络对图像的处理的优势体现出来，并且 resnet 的网络结构也提供一定的正则化效果，可以看到在训练集中的准确率最好的情况是在 91%左右，而在验证集上的效果更好，这里没有过拟合现象，由于对原有数据集的扩充，所以在训练集上的的准确率会比验证集低。

而且可以看到训练时间是 22 分 9 秒，在 GPU 上的运算速度会远远优于 CPU。

然后我们利用可视化函数大致的看几张图片的分类效果：

2.5 固定特征提取器

训练结束后一般将此模型的特征提取器固定，只训练最后一层进行分类即可，这会大大缩短训练所需时间，将预训练的模型定义的部分权值冻结，不进行梯度的更新：

In [16]:
```
res_mod = models.resnet34(pretrained=True)
for param in res_mod.parameters():
    param.requires_grad = False

num_ftrs = res_mod.fc.in_features
res_mod.fc = nn.Linear(num_ftrs, 2)

res_mod = res_mod.to(device)
criterion = nn.CrossEntropyLoss()

optimizer_ft = optim.SGD(res_mod.fc.parameters(), lr=0.001, momentum=0.9)

exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=7, gamma=0.1)
```

打印出模型结构：

In [17]:
```
for name, child in res_mod.named_children():
    print(name)
```

```
conv1
bn1
relu
maxpool
layer1
layer2
layer3
layer4
avgpool
fc
```

清楚结构之后，选择所需要训练的层，将其权值解冻，进而可以开始更新训练：

In [18]:
```
for name, child in res_mod.named_children():
    if name in ['layer3', 'layer4']:
        print(name + 'has been unfrozen.')
        for param in child.parameters():
            param.requires_grad = True
    else:
        for param in child.parameters():
            param.requires_grad = False
```

```
layer3has been unfrozen.
layer4has been unfrozen.
```

优化器也需要进行同步的更新：

```
In [19]: optimizer_conv = torch.optim.SGD(filter(lambda x: x.requires_grad, res_mod.parameters()), lr=0.001, momentum=0.9)
```

然后调用训练函数以及可视化函数进行之前的训练操作：

```
In [20]: base_model = train_model(res_mod, criterion, optimizer_conv, exp_lr_scheduler, num_epochs=3)
         visualize_model(base_model)
         plt.show()
```

```
Epoch 0/2
----------
Iterating through data...
train Loss: 0.3162 Acc: 0.8645
Iterating through data...
val Loss: 0.0498 Acc: 0.9850

Epoch 1/2
----------
Iterating through data...
train Loss: 0.2229 Acc: 0.9089
Iterating through data...
val Loss: 0.0387 Acc: 0.9875

Epoch 2/2
----------
Iterating through data...
train Loss: 0.1725 Acc: 0.9301
Iterating through data...
val Loss: 0.0402 Acc: 0.9875

Training complete in 14m 26s
Best val Acc: 0.987500
```

可以看到经过训练后最优值进一步的提高，并且计算时间大大缩短，这就是迁移学习带来的效果。再看一下可视化结果：

三、实验结论

对于猫和狗的图片分类，我们将图片转化为张量的形式，并且对原始数据的裁剪与旋转进而达到一个扩充数据的目的，再使用迁移学习的方法将已有的 resnet 网络加载到本地进行结构的修改再训练，最后将训练好的模型的权值保存，固定特征提取器，只对最后两层进行训练，加快训练速度，最后可以得到一个相对较好的结果。