# 实验二 猫狗分类

## 一、实验目的

1.进一步理解和掌握卷积神经网络中卷积层、卷积步长、卷积核、池化层、池化核等概念。

2.进一步掌握使用深度学习框架进行图像分类任务的具体流程：如读取数据、构造网络、训练和测试模型等等。

## 二、实验要求

1.基于Python语言和任意一种深度学习框架（实验指导书中使用Pytorch框架进行介绍），从零开始一步步完成数据读取、网络构建、模型训练和模型测试等过程，最终实现一个可以进行猫狗图像分类的分类器。

2.考虑到同学们机器性能的差异，该实验不强制要求使用Kaggle猫狗竞赛的原始数据集，大家可以根据自己的实际情况将原始数据集中训练集里的猫狗图像人为重新划分训练集和测试集。原则上要求人为划分的数据集中，训练集图像总数不少于2000张，测试集图像总数不少于大于500，最终模型的准确率要求不低于75%。鼓励在机器性能满足条件的情况下，使用大的数据集提高猫狗分类的准确率。本人采用训练数据集中的1000cat+1000dog训练，训练数据集中不同的500cat+500dog测试。训练集分类准确率可达到97%左右，测试集可达到91%以上。

3.按规定时间在课程网站提交实验报告、代码以及PPT。

## 三、实验原理

在构建图像分类网络时，主要使用到卷积层、池化层以及全连接层。在pytorch中，他们的定义如下：

1.Class torch.nn.Conv2d(in\_channels, out\_channels, kernel\_size, stride=1, padding=0, dilation=1, groups=1, bias=True)

参数：

in\_channels(int) // 输入信号的通道

out\_channels(int) // 卷积产生的通道

kerner\_size(int or tuple) // 卷积核的尺寸

stride(int or tuple, optional) // 卷积步长

padding(int or tuple, optional) // 输入的每一条边补充0的层数

dilation(int or tuple, optional) // 卷积核元素之间的间距

groups(int, optional) // 从输入通道到输出通道的阻塞连接数

bias(bool, optional) // 如果bias=True，添加偏置

2.Class torch.nn.MaxPool2d(kernel\_size, stride=None, padding=0, dilation=1, return\_indices=False, ceil\_mode=False)

参数：

kernel\_size(int or tuple) // max pooling窗口大小

stride(int or tuple, optional) // max pooling窗口移动的步长，默认值是kernel\_size

padding(int or tuple, optional) // 输入的每一条边补充0的层数

dilation(int or tuple, optional) //一个控制窗口中元素步幅的参数

return\_indices // 如果等于True，会返回输出最大值的序号，对于上采样操作会有帮助

ceil\_mode // 如果等于True，计算输出信号大小的时候，会使用向上取整，代替默认的向下取整的操作

3.Class torch.nn.Linear(in\_features, out\_features, bias=True)

参数：

in\_features // 每个输入样本的大小

out\_features // 每个输出样本的大小

bias // 若设置为False，这层不会学习偏置。默认值：True

## 四、实验所用数据集及工具

1.数据集

本实验使用实验数据基于kaggle Dogs vs. Cats 竞赛提供的官方数据集，数据集可在百度网盘中进行下载：

链接：https://pan.baidu.com/s/13hw4LK8ihR6-6-8mpjLKDA 密码：dmp4。

数据集的目录划分如下：

data

|— train

|— cat

|— dog

|— test

|— cat

|— dog

将数据集划分为训练集（train）和测试集（test），均包含dog和cat两个目录，且每个目录下包含与目录名类别相同的RGB图。数据集共25000张照片，其中训练集猫狗照片各10000张，验证集猫狗照片各2500张。（注：如实验要求所述，可根据资源情况自行调整训练集和测试集大小）本人采用训练数据集中的1000cat+1000dog训练，训练数据集中不同的500cat+500dog测试。

原始数据集如图1、2、3、4所示。

|  |  |
| --- | --- |
|  | C:\Users\63039\AppData\Local\Microsoft\Windows\INetCache\Content.Word\dog.9496.jpg |
| 图1 | 图2 |

|  |  |
| --- | --- |
|  | C:\Users\63039\AppData\Local\Microsoft\Windows\INetCache\Content.Word\cat.7013.jpg |
| 图3 | 图4 |

2.Pytorch深度学习框架

PyTorch是一个[开源](https://baike.baidu.com/item/%E5%BC%80%E6%BA%90/246339)的[Python](https://baike.baidu.com/item/Python)机器学习库，由[Facebook](https://baike.baidu.com/item/Facebook/7449587)人工智能研究院（FAIR）于2017年1月基于Torch推出。它是一个基于Python的可续计算包，提供两个高级功能：（1）具有强大的GPU加速的张量计算（如NumPy）；（2）包含自动求导系统的深度神经网络。同学们可以自行查找资料进行一步一步学习。

## 五、实验步骤与方法

对猫狗照片识别分类的卷积神经网络模型可以自行设计，本实验指导书给出的方法是自定义简单的卷积网络结构以实现猫狗图像分类。

实验可简单地划分为数据准备、模型定义、模型训练和测试三个步骤。

**1.数据准备**

首先要将自己的数据集划分为训练集和验证集，根据资源情况可以自行调整训练集和验证集大小。

首先定义图像的预处理操作，在这里可以使用更多图像增强方法以提升模型的性能，根据自己的实际情况进行设置。

然后使用Pytorch自带的ImageFolder进行数据读取，使用DataLoader进行数据导入。

# 1.Get data, Generate DataLoader of train&test together

# use\_gpu = torch.cuda.is\_available()

use\_gpu **=** **False**

path **=** 'data'

mean **=** **[**0.5**,** 0.5**,** 0.5**]**

std **=** **[**0.5**,** 0.5**,** 0.5**]**

loader\_batch\_size **=** 10

transform **=** transforms**.**Compose**([**transforms**.**CenterCrop**(**224**),**

transforms**.**ToTensor**(),**

transforms**.**Normalize**(**mean**,** std**)])**

data\_image **=** **{**x**:** datasets**.**ImageFolder**(**root**=**os**.**path**.**join**(**path**,** x**),**

transform**=**transform**)** **for** x **in** **[**'train'**,** 'test'**]}**

data\_loader **=** **{**x**:** DataLoader**(**

dataset**=**data\_image**[**x**],** batch\_size**=**loader\_batch\_size**,** shuffle**=True)** **for** x **in** **[**'train'**,** 'test'**]}**

classes **=** data\_image**[**'train'**].**classes

classes\_index **=** data\_image**[**'train'**].**class\_to\_idx

**print(**'classes:'**,** classes**)**

**print(**'classes\_index:'**,** classes\_index**)**

**print(**'train data set:'**,** **len(**data\_image**[**'train'**]))**

**print(**'test data set:'**,** **len(**data\_image**[**'test'**]))**

**2.模型定义**

模型构建时，需要创建一个继承自nn.Module的类，在这个类的\_init\_( )中定义网络结构，在forward中定义前向传播过程。

示例中定义了一个仅仅由两层卷积层和池化层构成的网络结构。在实验中大家可以自己定义更加复杂的网络结构以取得更好的效果。

本人采用VGG16网络，但是对最后的全连接层输出进行了更改，将1000改为了2，因为类别数为2。

初始化模型，定义优化器为Adam、损失函数为交叉熵损失。

# 2.VGG\_16

vgg\_cfg **=** **[**64**,** 64**,** 'M'**,** 128**,** 128**,** 'M'**,** 256**,** 256**,** 256**,** 'C'**,** 512**,** 512**,** 512**,** 'M'**,**

512**,** 512**,** 512**,** 'M'**]**

**def** vgg**(**cfg**,** i**,** batch\_norm**=False):**

layers **=** **[]**

in\_channels **=** i

**for** v **in** cfg**:**

**if** v **==** 'M'**:**

layers **+=** **[**nn**.**MaxPool2d**(**kernel\_size**=**2**,** stride**=**2**)]**

**elif** v **==** 'C'**:**

layers **+=** **[**nn**.**MaxPool2d**(**kernel\_size**=**2**,** stride**=**2**,** ceil\_mode**=True)]**

**else:**

conv2d **=** nn**.**Conv2d**(**in\_channels**,** v**,** kernel\_size**=**3**,** padding**=**1**)**

**if** batch\_norm**:**

layers **+=** **[**conv2d**,** nn**.**BatchNorm2d**(**v**),** nn**.**ReLU**(**inplace**=True)]**

**else:**

layers **+=** **[**conv2d**,** nn**.**ReLU**(**inplace**=True)]**

in\_channels **=** v

**return** nn**.**Sequential**(\***layers**)**

**class** vgg16**(**nn**.**Module**):**

**def** \_\_init\_\_**(**self**):**

**super(**vgg16**,** self**).**\_\_init\_\_**()**

self**.**vgg **=** vgg**(**vgg\_cfg**,** 3**)**

self**.**avgpool **=** nn**.**AdaptiveAvgPool2d**((**7**,** 7**))**

self**.**classifier **=** nn**.**Sequential**(**

nn**.**Linear**(**512 **\*** 7 **\*** 7**,** 4096**),**

nn**.**ReLU**(True),**

nn**.**Dropout**(),**

nn**.**Linear**(**4096**,** 4096**),**

nn**.**ReLU**(True),**

nn**.**Dropout**(),**

nn**.**Linear**(**4096**,** 1000**),** # 1000 can be changed as 2

**)**

**def** forward**(**self**,** x**):**

x **=** self**.**vgg**(**x**)**

x **=** self**.**avgpool**(**x**)**

x **=** torch**.**flatten**(**x**,** 1**)**

x **=** self**.**classifier**(**x**)**

**return** x

# 3.Use vgg16 model and change outputs dim as 2, Define the criterion & optimizer

model **=** models**.**vgg16**(**pretrained**=True)**

**for** parma **in** model**.**parameters**():**

parma**.**requires\_grad **=** **False**

# print(model)

model**.**classifier **=** nn**.**Sequential**(**torch**.**nn**.**Linear**(**512**\***7**\***7**,** 4096**),**

torch**.**nn**.**ReLU**(),**

torch**.**nn**.**Dropout**(**p**=**0.5**),**

torch**.**nn**.**Linear**(**4096**,** 4096**),**

torch**.**nn**.**ReLU**(),**

torch**.**nn**.**Dropout**(**p**=**0.5**),**

torch**.**nn**.**Linear**(**4096**,** 2**))**

**for** index**,** parma **in** **enumerate(**model**.**classifier**.**parameters**()):**

**if** index **==** 6**:**

parma**.**requires\_grad **=** **True**

**if** use\_gpu**:**

model **=** model**.**cuda**()**

criterion **=** torch**.**nn**.**CrossEntropyLoss**()**

optimizer **=** torch**.**optim**.**Adam**(**model**.**classifier**.**parameters**())**

# print(model)

**3.模型训练和测试**

训练和测试写在一块，但是会根据训练还是测试进而误差反传和不传。最后都会输出损失和准确率。

# 4.Train & Test

n\_epochs **=** 5

**for** epoch **in** **range(**n\_epochs**):**

start\_time **=** time**.**time**()**

**print(**'-'**\***50**,** '\nEpoch{}/{}\n'**.format(**epoch**+**1**,** n\_epochs**),** '-'**\***50**)**

**for** param **in** **[**'train'**,** 'test'**]:**

**if** param **==** 'train'**:**

model**.**train **=** **True**

**else:**

model**.**train **=** **False**

running\_loss **=** 0.0

running\_correct **=** 0

batch **=** 0

**for** data **in** data\_loader**[**param**]:**

batch **+=** 1

X**,** y **=** data

**if** use\_gpu**:**

X**,** y **=** Variable**(**X**.**cuda**()),** Variable**(**y**.**cuda**())**

**else:**

X**,** y **=** Variable**(**X**),** Variable**(**y**)**

optimizer**.**zero\_grad**()**

y\_pred **=** model**(**X**)**

\_**,** pred **=** torch**.max(**y\_pred**.**data**,** 1**)**

loss **=** criterion**(**y\_pred**,** y**)**

**if** param **==** 'train'**:**

loss**.**backward**()**

optimizer**.**step**()**

running\_loss **+=** loss**.**data

running\_correct **+=** torch**.sum(**pred **==** y**.**data**)**

**if** batch **%** 10 **==** 0 **and** param **==** 'train'**:**

**print(**'Batch {},\t Image {},\t Train Loss:{:.4f},\t Train ACC:{:.4f}%'**.format(**

batch**,** loader\_batch\_size**\***batch**,** running\_loss**/(**loader\_batch\_size**\***batch**),** 100**\***running\_correct**/(**loader\_batch\_size**\***batch**)))**

epoch\_loss **=** running\_loss**/len(**data\_image**[**param**])**

epoch\_correct **=** 100**\***running\_correct**/len(**data\_image**[**param**])**

**print(**'{} Loss:{:.4f}, Correct:{:.4f}%'**.format(**

param**,** epoch\_loss**,** epoch\_correct**))**

spend\_time **=** time**.**time**()** **-** start\_time

**print(**'Epoch{} training & testing time is: {:.0f}m {:.0f}s'**.format(**

epoch**+**1**,** spend\_time**//**60**,** spend\_time **%** 60**))**

torch**.**save**(**model**.**state\_dict**(),** './model\_params.pkl'**)**

## 六、实验结果

classes**:** **[**'cat'**,** 'dog'**]**

classes\_index**:** **{**'cat'**:** 0**,** 'dog'**:** 1**}**

train data **set:** 2000

test data **set:** 1000

**--------------------------------------------------**

Epoch1**/**5

**--------------------------------------------------**

Batch 10**,** Image 100**,** Train Loss**:**1.0441**,** Train ACC**:**58.0000**%**

Batch 20**,** Image 200**,** Train Loss**:**0.5841**,** Train ACC**:**72.5000**%**

**...**

Batch 190**,** Image 1900**,** Train Loss**:**0.1307**,** Train ACC**:**88.6316**%**

Batch 200**,** Image 2000**,** Train Loss**:**0.1258**,** Train ACC**:**88.8000**%**

train Loss**:**0.1258**,** Correct**:**88.8000**%**

test Loss**:**0.0348**,** Correct**:**93.2000**%**

Epoch1 training **&** testing time **is:** 17m 51s

**--------------------------------------------------**

Epoch2**/**5

**--------------------------------------------------**

Batch 10**,** Image 100**,** Train Loss**:**0.0201**,** Train ACC**:**97.0000**%**

Batch 20**,** Image 200**,** Train Loss**:**0.0191**,** Train ACC**:**96.0000**%**

**...**

Batch 190**,** Image 1900**,** Train Loss**:**0.0335**,** Train ACC**:**95.6316**%**

Batch 200**,** Image 2000**,** Train Loss**:**0.0339**,** Train ACC**:**95.5500**%**

train Loss**:**0.0339**,** Correct**:**95.5500**%**

test Loss**:**0.0378**,** Correct**:**95.6000**%**

Epoch2 training **&** testing time **is:** 16m 43s

**--------------------------------------------------**

Epoch3**/**5

**--------------------------------------------------**

Batch 10**,** Image 100**,** Train Loss**:**0.0149**,** Train ACC**:**97.0000**%**

Batch 20**,** Image 200**,** Train Loss**:**0.0130**,** Train ACC**:**98.0000**%**

**...**

Batch 190**,** Image 1900**,** Train Loss**:**0.0874**,** Train ACC**:**96.5263**%**

Batch 200**,** Image 2000**,** Train Loss**:**0.0927**,** Train ACC**:**96.3500**%**

train Loss**:**0.0927**,** Correct**:**96.3500**%**

test Loss**:**0.2201**,** Correct**:**92.1000**%**

Epoch3 training **&** testing time **is:** 16m 48s

**--------------------------------------------------**

Epoch4**/**5

**--------------------------------------------------**

Batch 10**,** Image 100**,** Train Loss**:**0.8451**,** Train ACC**:**86.0000**%**

Batch 20**,** Image 200**,** Train Loss**:**0.6393**,** Train ACC**:**90.5000**%**

**...**

Batch 190**,** Image 1900**,** Train Loss**:**0.1846**,** Train ACC**:**96.0000**%**

Batch 200**,** Image 2000**,** Train Loss**:**0.1829**,** Train ACC**:**96.0500**%**

train Loss**:**0.1829**,** Correct**:**96.0500**%**

test Loss**:**0.4702**,** Correct**:**91.6000**%**

Epoch4 training **&** testing time **is:** 18m 29s

**--------------------------------------------------**

Epoch5**/**5

**--------------------------------------------------**

Batch 10**,** Image 100**,** Train Loss**:**0.1273**,** Train ACC**:**97.0000**%**

Batch 20**,** Image 200**,** Train Loss**:**0.1143**,** Train ACC**:**97.0000**%**

**...**

Batch 190**,** Image 1900**,** Train Loss**:**0.1277**,** Train ACC**:**96.8947**%**

Batch 200**,** Image 2000**,** Train Loss**:**0.1296**,** Train ACC**:**96.9500**%**

train Loss**:**0.1296**,** Correct**:**96.9500**%**

test Loss**:**0.4617**,** Correct**:**91.3000**%**

Epoch5 training **&** testing time **is:** 19m 25s

从中可以看到，Epoch1/5时，训练准确率从开始的58.0%随数据量增多上升到88.8%。到Epoch2/5测试准确率最高，而后Epoch下降，或许是后续训练过拟合的缘故。但基本上测试集准确率能达到91%以上。本实验还有待更大数据量的训练。