Cat and Dog Classification with VGG16 and Resnet50

Experiment Description

使用`Kaggle`猫狗分类的原始数据集，实现模型最终的准确率达到75%及以上。

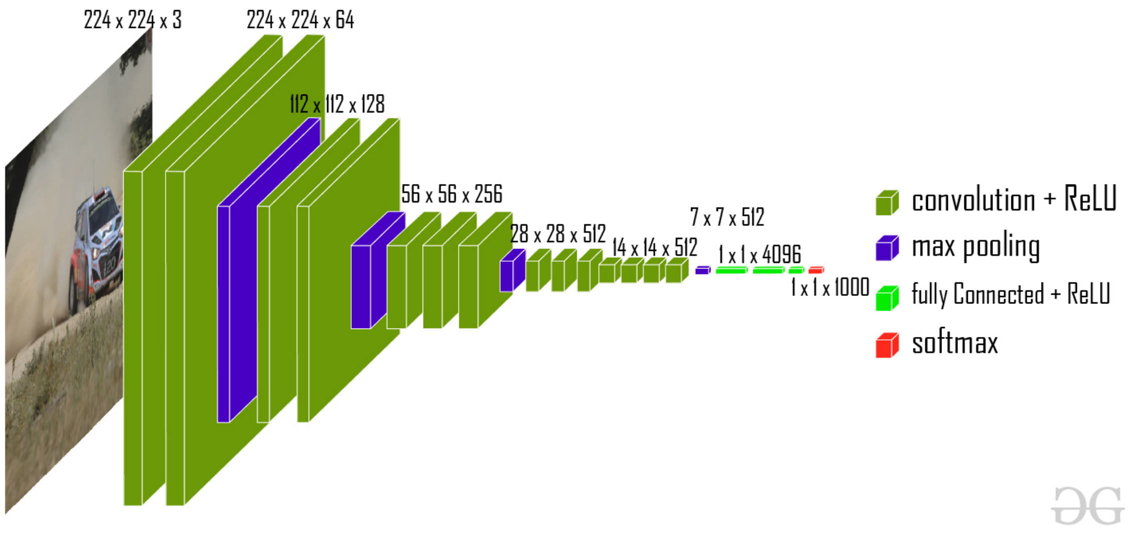
- 为了进一步掌握使用深度学习框架进行图像分类任务的具体流程如：读取数据、构造网络、训练和测试模型
- 掌握经典卷积神经网络`VGG16`、`ResNet50`的基本结构

Data Download

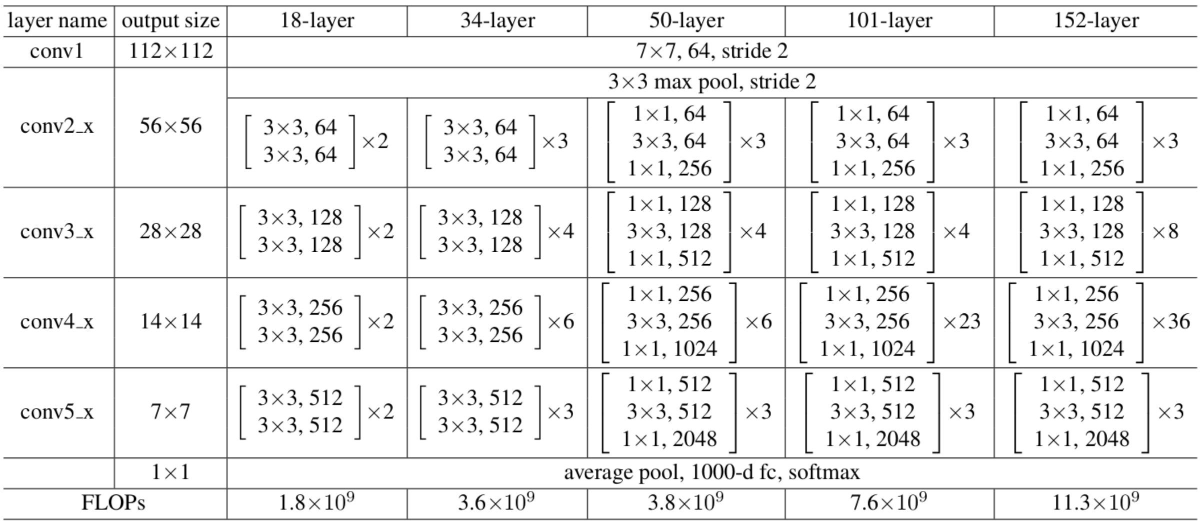
https://www.microsoft.com/en-us/download/details.aspx?id=54765

Experiment Principle

VGG16 stacks several small convolutional kernel and enlarges the receptive field, and achieves the same effect of using a large convolutional kernel, which can recognize bigger features. The structure of the network is very well organized, using repetitive blocks. VGG has a large amount of parameters (can reach 100 million), and takes a long training time.



ResNet mitigates the problem of slow convergence of the mode. It adds side paths in the module design, and accelerates model convergence.



layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times23$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Experiment Process I: VGG16

Preprocess Data

- File format preprocessing: While checking the file format, it is found that although the file extensions are all .jpg, the file actually includes formats cannot be processes like .bmp and .gif. Only .jpg,.jpeg and .png are acceptable in the network, so a seperate script is implemented to delete the files in other files from the dataset(a few dozens in total), making the model to run properly.

```
# test.py
import os
from pathlib import Path
from PIL import Image

# dataset path
data_dir = Path('data')

# allowed format
valid_formats = {'JPEG', 'PNG', 'JPG'}

# iterate and check format
for img_path in data_dir.glob("**/*"):
        with Image.open(img_path) as img:
            actual_format = img.format
            if actual_format not in valid_formats:
                print(f"Deleting {img_path}, format is 
{actual_format}")
                os.remove(img_path)
```

- Data preprocessing: Use `transforms.Resize((224, 224))` to resize the pictures. Use `transforms.ToTensor` to change pictures into processable tensor, with the value between 0 and 1.

  ```
  import torch
  import torch.nn as nn
  import torch.optim as optim
  from torch.utils.data import DataLoader
  from torchvision import datasets, transforms
  from torch import device
  from PIL import Image

  # preprocess: size, tensor, normalization
  transform = transforms.Compose([
      transforms.Resize((224, 224)),
      transforms.ToTensor(),
      transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225])
  ```

Dataset Procession

Load dataset as train_data from `data` folder, use transform defined above to preprocess. Define iterator train_loader, randomly load 32 data per batch.

```

train_data = datasets.ImageFolder('data', transform=transform)
train_loader = DataLoader(train_data, batch_size=32, shuffle=True)
```

Use mps acceleration.

```
device = torch.device('mps' if torch.backends.mps.is_available() else 'cpu')
```

Define VGG16

Use 3*3 convolutional kernels. The first two layer groups are convolution-convolution-pooling, the last three layer groups are convolution-convolution-convolution-pooling, and three full connection layers, 16 layers in total.

```
# VGG16.py
import torch.nn as nn

# VGG16
```

```python
class VGG16(nn.Module):
    def __init__(self, num_classes=2):
        super(VGG16, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, padding=1),

            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(256, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )

        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.5),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.5),
            nn.Linear(4096, num_classes),  # output 2 classes
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)  # flatten
        x = self.classifier(x)
        return x
```

Define ResNet50

```
import torch
import torch.nn as nn
from torch.autograd import Variable

# check MPS
device = torch.device("mps" if torch.backends.mps.is_available() else "cpu")

class convolutional_block(nn.Module):  # convolutional_block
    def __init__(self, cn_input, cn_middle, cn_output, s=2):
        super(convolutional_block, self).__init__()
        self.step1 = nn.Sequential(nn.Conv2d(cn_input, cn_middle, (1, 1), (s,
s), padding=0, bias=False),
                                   nn.BatchNorm2d(cn_middle, affine=False),
nn.ReLU(inplace=True),
                                   nn.Conv2d(cn_middle, cn_middle, (3, 3),
(1, 1), padding=(1, 1), bias=False),
                                   nn.BatchNorm2d(cn_middle, affine=False),
nn.ReLU(inplace=True),
                                   nn.Conv2d(cn_middle, cn_output, (1, 1),
(1, 1), padding=0, bias=False),
                                   nn.BatchNorm2d(cn_output, affine=False))
        self.step2 = nn.Sequential(nn.Conv2d(cn_input, cn_output, (1, 1), (s,
s), padding=0, bias=False),
                                   nn.BatchNorm2d(cn_output, affine=False))
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        x_tmp = x
        x = self.step1(x)
        x_tmp = self.step2(x_tmp)
        x = x + x_tmp
        x = self.relu(x)
        return x


class identity_block(nn.Module):  # identity_block
    def __init__(self, cn, cn_middle):
        super(identity_block, self).__init__()
        self.step = nn.Sequential(nn.Conv2d(cn, cn_middle, (1, 1), (1, 1),
padding=0, bias=False),
                                  nn.BatchNorm2d(cn_middle, affine=False),
nn.ReLU(inplace=True),
                                  nn.Conv2d(cn_middle, cn_middle, (3, 3), (1,
1), padding=1, bias=False),
                                  nn.BatchNorm2d(cn_middle, affine=False),
nn.ReLU(inplace=True),
                                  nn.Conv2d(cn_middle, cn, (1, 1), (1, 1),
padding=0, bias=False),
                                  nn.BatchNorm2d(cn, affine=False))
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        x_tmp = x
        x = self.step(x)
        x = x + x_tmp
        x = self.relu(x)
        return x


class Resnet(nn.Module):  # main model
    def __init__(self, c_block, i_block):
        super(Resnet, self).__init__()
        self.conv = nn.Sequential(nn.Conv2d(3, 64, (7, 7), (2, 2), padding=
(3, 3), bias=False),
                                  nn.BatchNorm2d(64, affine=False),
nn.ReLU(inplace=True), nn.MaxPool2d((3, 3), 2, 1))
        self.layer1 = c_block(64, 64, 256, 1)
        self.layer2 = i_block(256, 64)
        self.layer3 = c_block(256, 128, 512)
        self.layer4 = i_block(512, 128)
        self.layer5 = c_block(512, 256, 1024)
        self.layer6 = i_block(1024, 256)
        self.layer7 = c_block(1024, 512, 2048)
        self.layer8 = i_block(2048, 512)
        self.out = nn.Linear(2048, 2, bias=False)
        self.avgpool = nn.AvgPool2d(7, 7)
    def forward(self, input):
        x = self.conv(input)
        x = self.layer1(x)
        for i in range(2):
            x = self.layer2(x)
        x = self.layer3(x)
        for i in range(3):
            x = self.layer4(x)
        x = self.layer5(x)
        for i in range(5):
            x = self.layer6(x)
        x = self.layer7(x)
        for i in range(2):
            x = self.layer8(x)

        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        output = self.out(x)
        return output

# implement and move to mps
net = Resnet(convolutional_block, identity_block).to('mps')
```

Define Hyperparameters

This is a binary classification problem, num_classes=2, move model to mps to train. Use cross-entropy for classification problem. Use Adam optimizer, with a learning rate of 0.0001, train for 10 epochs.

```
# implement model
# model = VGG16(num_classes=2)
# model = models.vgg16(pretrained=True)
model = models.resnet50(pretrained=True)
# model = Resnet(convolutional_block, identity_block)

model = model.to(device)

# loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.0001)
```

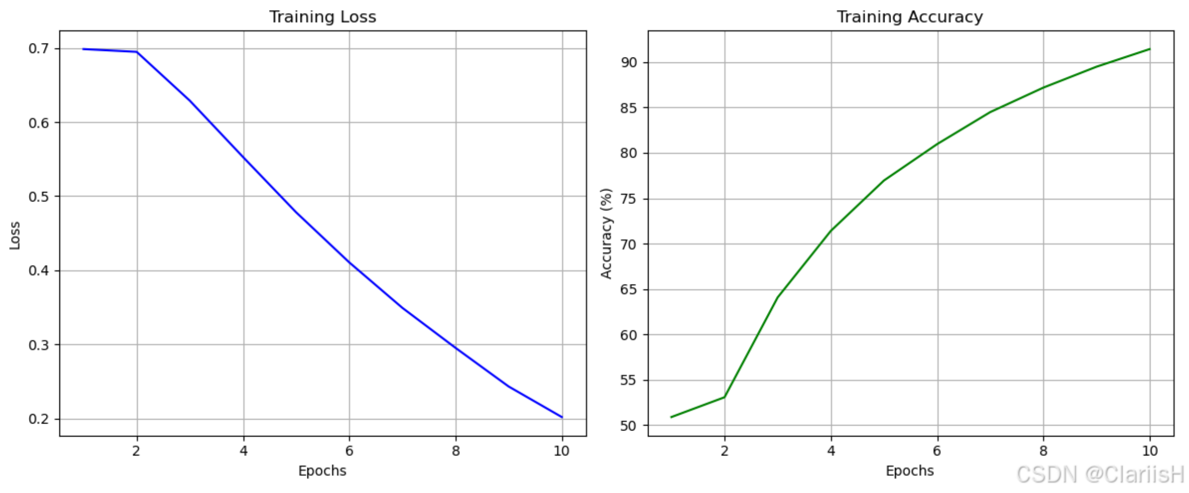
Training

Train the model for 10 epochs, save the model, and plot the result.

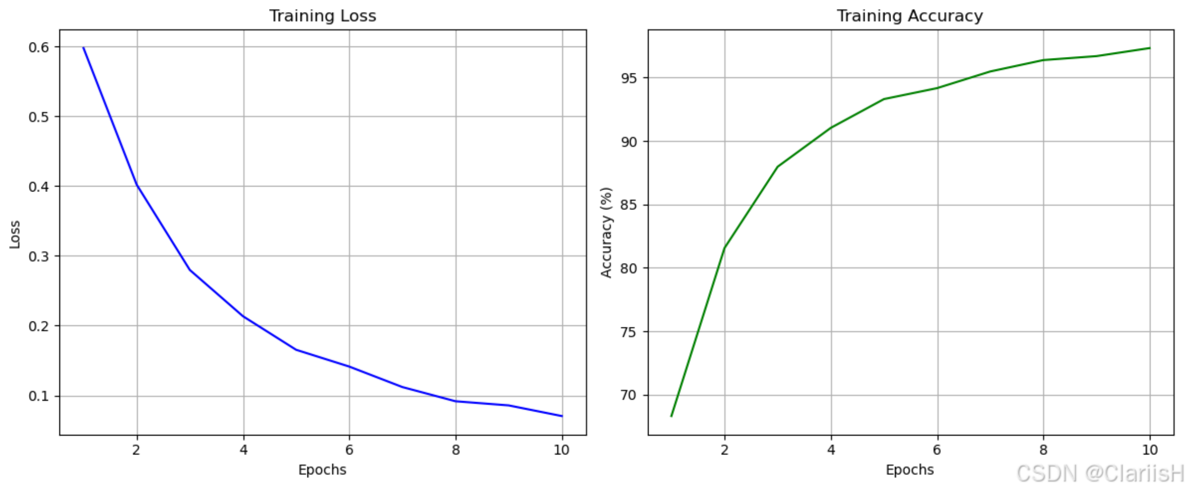
```
for epoch in range(num_epochs):
    model.train()  # train mode
    running_loss = 0.0
    correct = 0
    total = 0

    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)

        # empty gradients, prevent accumulatin
        optimizer.zero_grad()
        outputs = model(inputs) # forward
        loss = criterion(outputs, labels)
        loss.backward() # backward
        optimizer.step()

        running_loss += loss.item()

        # accuracy
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
    print(f"Epoch [{epoch+1}/{num_epochs}], Loss:
{running_loss/len(train_loader):.4f}, Accuracy: {100 * correct /
total:.2f}%")
```

Analysis

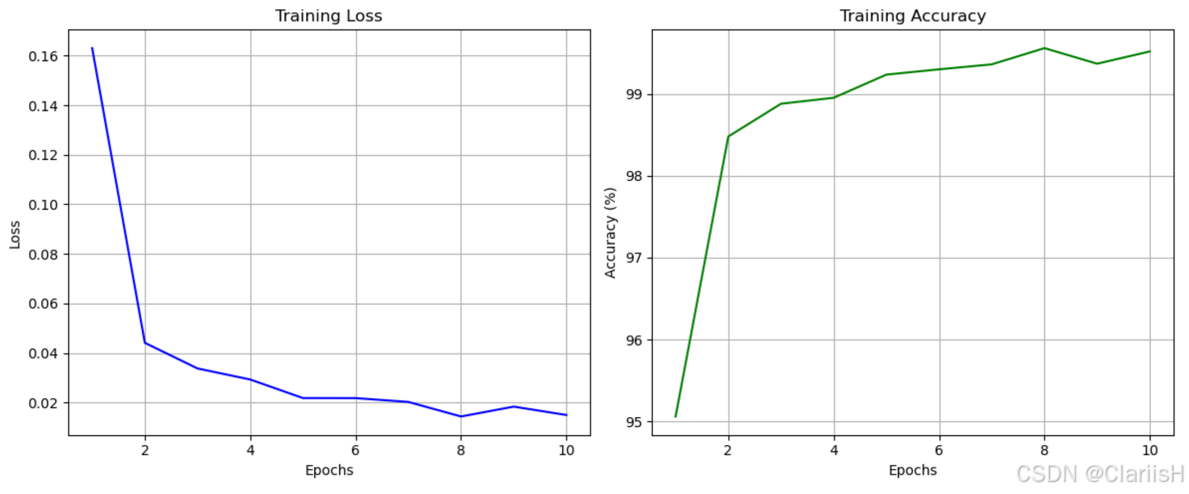
It can be seen from the output that the accuracy of VGG16 rised rapidly after two epochs of training. From the sixth epoch the accuracy increase slows down, and reach 93.78% in the tenth epoch. The performance of the model is good.



Resnet has a high accuracy from the beginning(70% approximately). Then it reached over 95% accuracy in the 10th epoch.



For pre-trained models, VGG16 has the accuracy of 95% from the beginning. After training for four epochs, the accuracy reached 99%.



The pre-trained Resnet50 model reached the accuracy of 97% from the beginning, and the accuracy raised to 99% after two epochs.

It can be seen from the experiment above that, ResNet50 has faster convergence speed and higher accuracy than VGG16, including the model built from scratch and the pre-trained model. Both of the pre-trained models have good performance.