安徽大学《深度学习与神经网络》
实验报告 6

学号：WA2214014　　专业：人工智能　　姓名：杨跃浙

实验日期：06.22　　教师签字：　　成绩：

[实验名称] 神经网络综合应用实验

[实验目的]

1．熟悉和掌握利用神经网络开展视觉分类

2．熟悉和掌握利用神经网络开展风格迁移

3．熟悉和掌握利用神经网络开展语义分割

[实验要求]

1．采用 Python 语言基于 PyTorch 深度学习框架进行编程

2．代码可读性强：变量、函数、类等命名可读性强，包含必要的注释

3．提交实验报告要求：

- 命名方式："学号-姓名-Lab-N"（N 为实验课序号，即：1-6）；
- 截止时间：下次实验课当晚 23:59；
- 提交方式：智慧安大-网络教育平台-作业；
- 按时提交（**过时不补**）；

[实验内容]

1．视觉分类：

- 学习、运行、调试参考教材 13.14 小节 Kaggle 狗的品种识别
- 完成练习题 2

2．风格迁移：

- 学习、运行和调试参考教材 13.12 小节内容
- 完成练习 1 和 2

3．语义分割：

- 学习、运行和调试参考教材 13.9 小节内容
- 完成练习题 1

4．参考资料：

- **参考教材**：https://zh-v2.d2l.ai/d2l-zh-pytorch.pdf
- **PyTorch 官方文档**：https://pytorch.org/docs/2.0/；
- **PyTorch 官方论坛**：https://discuss.pytorch.org/

[实验代码和结果]

1.视觉分类：

实验代码：

```
import os
import shutil
import pandas as pd
import torch
import torchvision
from torch import nn
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
from torchvision import transforms
from PIL import Image
import torchvision.models as models
from tqdm import tqdm
from d2l import torch as d2l
import torch.optim as optim
from torch.optim.lr_scheduler import StepLR

# 数据路径
data_dir =
'./NNDL-Class/Project6/Data/dog-breed-identification'
train_dir = os.path.join(data_dir, 'train')
test_dir = os.path.join(data_dir, 'test')
labels_file = os.path.join(data_dir, 'labels.csv')

# 结果保存路径
result_dir = './NNDL-Class/Project6/Result'
os.makedirs(result_dir, exist_ok=True)

# 1. 数据整理：根据标签文件将图像移动到相应的子文件夹中
labels_df = pd.read_csv(labels_file)

# 创建目标文件夹，如果它不存在
for breed in labels_df['breed'].unique():
```

```
os.makedirs(os.path.join(train_dir, breed), exist_ok=True)

# 将图片移动到相应的文件夹
for _, row in labels_df.iterrows():
image_name = row['id'] + '.jpg'
breed = row['breed']
src_path = os.path.join(train_dir, image_name)
dst_path = os.path.join(train_dir, breed, image_name)
# 如果图片存在，则移动
if os.path.exists(src_path):
shutil.move(src_path, dst_path)


# 2. 数据加载和预处理
transform_train = transforms.Compose([
transforms.RandomResizedCrop(224, scale=(0.08, 1.0),
ratio=(3.0/4.0, 4.0/3.0)),
transforms.RandomHorizontalFlip(),
transforms.RandomRotation(15),
transforms.ColorJitter(brightness=0.3, contrast=0.3,
saturation=0.3),
transforms.ToTensor(),
transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224,
0.225])
])

transform_test = transforms.Compose([
transforms.Resize(256),
transforms.CenterCrop(224),
transforms.ToTensor(),
transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224,
0.225])
])

# 加载训练集和验证集
train_ds = torchvision.datasets.ImageFolder(train_dir,
transform=transform_train)
train_size = int(0.9 * len(train_ds)) # 90% 用于训练
valid_size = len(train_ds) - train_size # 10% 用于验证
```

```
train_data, valid_data =
torch.utils.data.random_split(train_ds, [train_size,
valid_size])

# 减小批量大小以适配 GPU 内存
train_iter = DataLoader(train_data, batch_size=64,
shuffle=True, num_workers=4)
valid_iter = DataLoader(valid_data, batch_size=64,
shuffle=False, num_workers=4)

# 加载测试集图像（无需标签）
class TestDataset(torch.utils.data.Dataset):
def __init__(self, image_dir, transform=None):
self.image_dir = image_dir
self.transform = transform
self.image_files = [f for f in os.listdir(image_dir) if
f.endswith('.jpg')]

def __len__(self):
return len(self.image_files)

def __getitem__(self, idx):
img_name = os.path.join(self.image_dir,
self.image_files[idx])
image = Image.open(img_name)
if self.transform:
image = self.transform(image)
return image, self.image_files[idx] # 返回文件名用于预测结果


# 创建 test_images 文件夹（如果不存在）
test_images_dir = os.path.join(test_dir, 'test_images')
os.makedirs(test_images_dir, exist_ok=True)

# 移动 test 文件夹中的所有文件到 test_images
for filename in os.listdir(test_dir):
file_path = os.path.join(test_dir, filename)
if os.path.isfile(file_path) and filename.endswith('.jpg'):
# 确保是 jpg 文件
```

```
shutil.move(file_path, os.path.join(test_images_dir,
filename))

test_ds = TestDataset(test_images_dir,
transform=transform_test)
test_iter = DataLoader(test_ds, batch_size=64,
shuffle=False, drop_last=False, num_workers=4)

def get_net(devices):
# 加载预训练的 ResNet34 模型
finetune_net =
models.resnet34(weights=models.ResNet34_Weights.IMAGENET1
K_V1)
# 替换 ResNet34 的输出层，调整为 120 个类别
# 添加更多层以提高模型容量
finetune_net.fc = nn.Sequential(
nn.Dropout(0.5),
nn.Linear(finetune_net.fc.in_features, 1024),
nn.ReLU(),
nn.BatchNorm1d(1024),
nn.Dropout(0.3),
nn.Linear(1024, 512),
nn.ReLU(),
nn.BatchNorm1d(512),
nn.Linear(512, 120) # 120 类狗的品种
)
finetune_net = finetune_net.to(devices[0])

# 解冻最后两个残差块以进行微调
for name, param in finetune_net.named_parameters():
if 'layer3' in name or 'layer4' in name or 'fc' in name:
param.requires_grad = True
else:
param.requires_grad = False

return finetune_net

# 评估模型损失和准确率
def evaluate_loss(data_iter, net, devices):
```

```
net.eval()
l_sum, acc_sum, n = 0.0, 0.0, 0
with torch.no_grad():
for features, labels in tqdm(data_iter, desc='Evaluating
Validation'):
features, labels = features.to(devices[0]),
labels.to(devices[0])
output = net(features)
l = nn.CrossEntropyLoss()(output, labels)
acc = (output.argmax(dim=1) == labels).sum().item()
l_sum += l.item()
acc_sum += acc
n += labels.shape[0]
return l_sum / len(data_iter), acc_sum / n

# 训练函数
def train(net, train_iter, valid_iter, num_epochs, lr, wd,
devices, lr_period, lr_decay):
# 只优化需要梯度的参数
params_to_optimize = [p for p in net.parameters() if
p.requires_grad]
trainer = optim.SGD(params_to_optimize, lr=lr, momentum=0.9,
weight_decay=wd)
scheduler = StepLR(trainer, step_size=lr_period,
gamma=lr_decay)
loss_fn = nn.CrossEntropyLoss()
epoch = []
train_loss = []
train_acc = []
valid_loss = []
valid_acc = []
best_valid_acc = 0.0
for epoch_idx in range(num_epochs):
net.train()
metric = d2l.Accumulator(3) # 用于保存损失和准确率
for features, labels in tqdm(train_iter, desc=f'Epoch
{epoch_idx+1}/{num_epochs} Training'):
```

```
features, labels = features.to(devices[0]),
labels.to(devices[0])
trainer.zero_grad()
output = net(features)
loss = loss_fn(output, labels)
loss.backward()
trainer.step()
# 更新指标
metric.add(loss.item(), (output.argmax(dim=1) ==
labels).sum().item(), labels.shape[0])
# 更新学习率
scheduler.step()
current_lr = trainer.param_groups[0]['lr']
# 计算本 epoch 指标
epoch.append(epoch_idx + 1)
train_loss.append(metric[0] / metric[2])
train_acc.append(metric[1] / metric[2])
# 验证集评估
valid_loss_val, valid_acc_val = evaluate_loss(valid_iter,
net, devices)
valid_loss.append(valid_loss_val)
valid_acc.append(valid_acc_val)
print(f'Epoch {epoch_idx+1}/{num_epochs}: '
f'Train Loss: {train_loss[-1]:.4f}, Train Acc:
{train_acc[-1]:.4f}, '
f'Valid Loss: {valid_loss[-1]:.4f}, Valid Acc:
{valid_acc[-1]:.4f}, '
f'LR: {current_lr:.6f}')
# 保存最佳模型
if valid_acc_val > best_valid_acc:
best_valid_acc = valid_acc_val
save_model(net, 'best_model.pth')
print(f'Best model saved with validation accuracy:
{best_valid_acc:.4f}')
# 保存最终训练曲线图像
save_training_curve(epoch, train_loss, train_acc,
valid_loss, valid_acc, 'final_training_curve.png')
return net
```

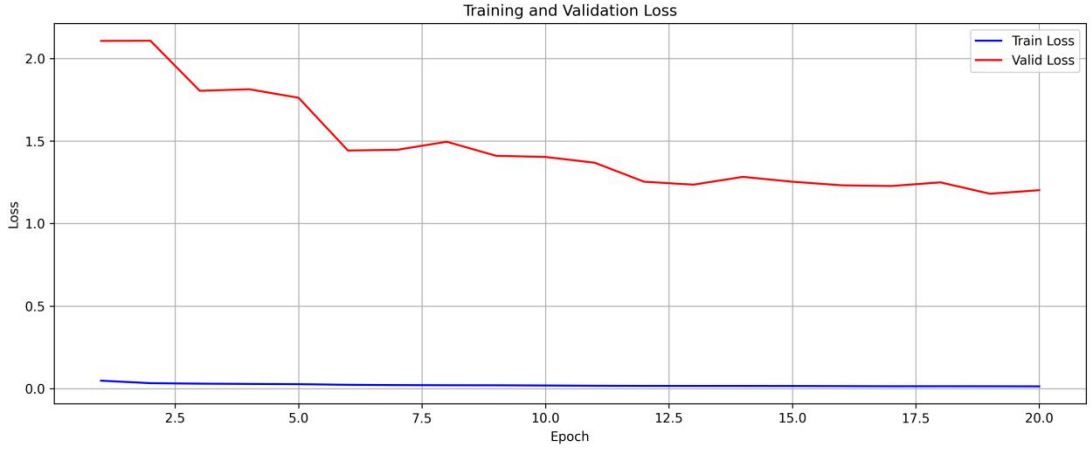
```
# 保存训练曲线的函数
def save_training_curve(epoch, train_loss, train_acc,
valid_loss, valid_acc, filename):
plt.figure(figsize=(12, 10))
# 损失曲线
plt.subplot(2, 1, 1)
plt.plot(epoch, train_loss, 'b-', label="Train Loss")
plt.plot(epoch, valid_loss, 'r-', label="Valid Loss")
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.title('Training and Validation Loss')
plt.grid(True)
# 准确率曲线
plt.subplot(2, 1, 2)
plt.plot(epoch, train_acc, 'b-', label="Train Accuracy")
plt.plot(epoch, valid_acc, 'r-', label="Valid Accuracy")
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Training and Validation Accuracy')
plt.grid(True)
plt.tight_layout()
plt.savefig(os.path.join(result_dir, filename), dpi=300)
plt.close()

# 保存模型
def save_model(net, filename):
torch.save(net.state_dict(), os.path.join(result_dir,
filename))
print(f'Model saved to {filename}')

# 保存预测结果
def save_predictions(net, test_iter, filename):
net.eval()
preds = []
ids = []
with torch.no_grad():
```

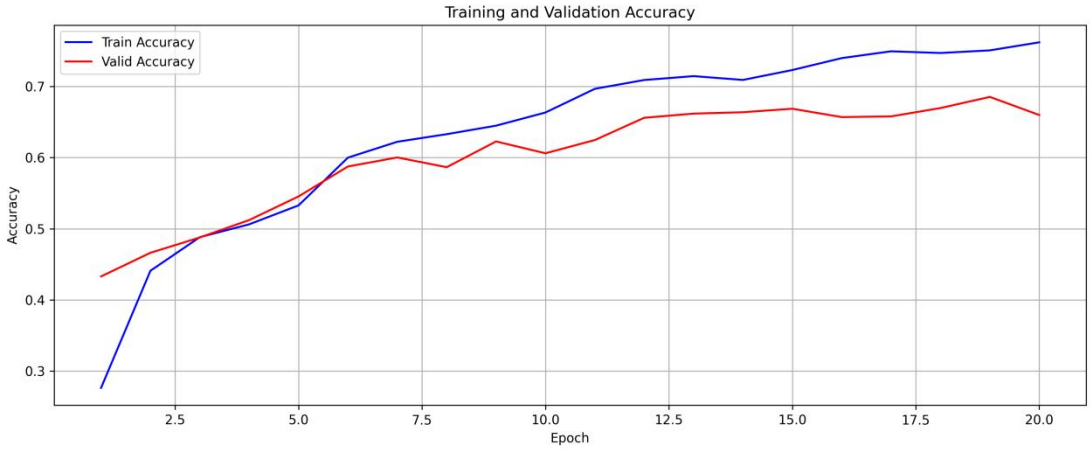
```
for images, filenames in tqdm(test_iter, desc='Predicting
Test Set'):
images = images.to(devices[0])
outputs = net(images)
probs = torch.nn.functional.softmax(outputs, dim=1)
preds.extend(probs.cpu().numpy())
ids.extend([fname.replace('.jpg', '') for fname in
filenames])
# 创建提交文件
classes = sorted(os.listdir(train_dir))
submission_df = pd.DataFrame(preds, columns=classes)
submission_df.insert(0, 'id', ids)
submission_df.to_csv(os.path.join(result_dir, filename),
index=False)
print(f'Predictions saved to {filename}')

# 获取设备
devices = d2l.try_all_gpus()
print(f"Using devices: {devices}")
net = get_net(devices)

# 训练模型并保存训练曲线
# 增加学习率，添加学习率调度
net = train(net, train_iter, valid_iter, num_epochs=20,
lr=0.01, wd=1e-4,
devices=devices, lr_period=5, lr_decay=0.5)

# 保存最终模型
save_model(net, 'final_model.pth')

# 预测并保存测试集结果
save_predictions(net, test_iter, 'submission.csv')
```

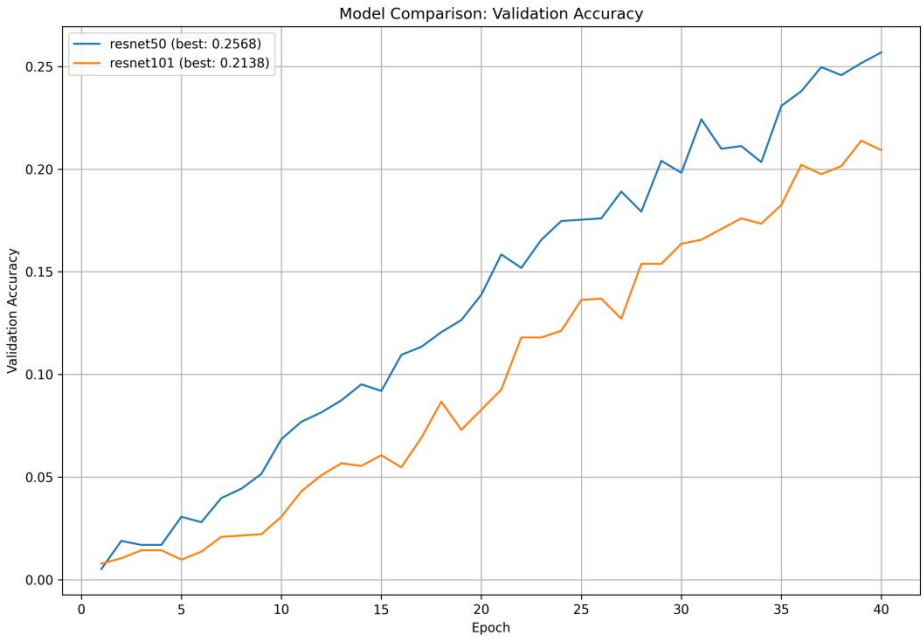
实验结果：

```
(yyzttt) (base) yyz@4028Dog:~$ /usr/local/anaconda3/envs/yyzttt/bin/python /home/yyz/NNDL-Class/Project6/Cod
e/classification.py
Using devices: [device(type='cuda', index=0), device(type='cuda', index=1), device(type='cuda', index=2), de
vice(type='cuda', index=3), device(type='cuda', index=4), device(type='cuda', index=5), device(type='cuda',
index=6), device(type='cuda', index=7)]
Epoch 1/20 Training: 100%|| 144/144 [00:14<00:00,  9.77it/s]
Evaluating Validation: 100%|| 16/16 [00:01<00:00,  8.64it/s]
Epoch 1/20: Train Loss: 0.0480, Train Acc: 0.2764, Valid Loss: 2.1064, Valid Acc: 0.4330, LR: 0.010000
Model saved to best_model.pth
Best model saved with validation accuracy: 0.4330
Epoch 2/20 Training: 100%|| 144/144 [00:14<00:00, 10.15it/s]
Evaluating Validation: 100%|| 16/16 [00:01<00:00,  8.79it/s]
Epoch 2/20: Train Loss: 0.0327, Train Acc: 0.4410, Valid Loss: 2.1075, Valid Acc: 0.4663, LR: 0.010000
Model saved to best_model.pth
Best model saved with validation accuracy: 0.4663
Epoch 3/20 Training: 100%|| 144/144 [00:14<00:00, 10.18it/s]
Evaluating Validation: 100%|| 16/16 [00:01<00:00,  8.63it/s]
Epoch 3/20: Train Loss: 0.0298, Train Acc: 0.4882, Valid Loss: 1.8040, Valid Acc: 0.4878, LR: 0.010000
Model saved to best_model.pth
Best model saved with validation accuracy: 0.4878
Epoch 4/20 Training: 100%|| 144/144 [00:13<00:00, 10.37it/s]
Evaluating Validation: 100%|| 16/16 [00:01<00:00,  8.45it/s]
Epoch 4/20: Train Loss: 0.0282, Train Acc: 0.5063, Valid Loss: 1.8136, Valid Acc: 0.5122, LR: 0.010000
Model saved to best_model.pth
Best model saved with validation accuracy: 0.5122
Epoch 5/20 Training: 100%|| 144/144 [00:13<00:00, 10.30it/s]
Evaluating Validation: 100%|| 16/16 [00:01<00:00,  8.67it/s]
Epoch 5/20: Train Loss: 0.0267, Train Acc: 0.5329, Valid Loss: 1.7620, Valid Acc: 0.5455, LR: 0.005000
Model saved to best_model.pth
Best model saved with validation accuracy: 0.5455
Epoch 6/20 Training: 100%|| 144/144 [00:14<00:00, 10.18it/s]
Evaluating Validation: 100%|| 16/16 [00:01<00:00,  8.39it/s]
Epoch 6/20: Train Loss: 0.0229, Train Acc: 0.6000, Valid Loss: 1.4423, Valid Acc: 0.5875, LR: 0.005000
Model saved to best_model.pth
Best model saved with validation accuracy: 0.5875
Epoch 7/20 Training:  42%|                                  | 61/144 [00:06<00:08,  9.82it/s]
```

```
Epoch 18/20: Train Loss: 0.0142, Train Acc: 0.7469, Valid Loss: 1.2493, Valid Acc: 0.6696, LR: 0.001250
Model saved to best_model.pth
Best model saved with validation accuracy: 0.6696
Epoch 19/20 Training: 100%|| 144/144 [00:14<00:00, 10.02it/s]
Evaluating Validation: 100%|| 16/16 [00:01<00:00,  8.61it/s]
Epoch 19/20: Train Loss: 0.0141, Train Acc: 0.7506, Valid Loss: 1.1806, Valid Acc: 0.6852, LR: 0.001250
Model saved to best_model.pth
Best model saved with validation accuracy: 0.6852
Epoch 20/20 Training: 100%|| 144/144 [00:14<00:00, 10.05it/s]
Evaluating Validation: 100%|| 16/16 [00:01<00:00,  8.72it/s]
Epoch 20/20: Train Loss: 0.0135, Train Acc: 0.7619, Valid Loss: 1.2021, Valid Acc: 0.6598, LR: 0.000625
Model saved to final_model.pth
Predicting Test Set: 100%|| 162/162 [00:12<00:00, 12.69it/s]
Predictions saved to submission.csv
```







练习题 2：如果使用更深的预训练模型，会得到更好

的结果吗？如何调整超参数？能进一步改善结果吗？

实验代码：

```
import os
import shutil
import pandas as pd
import torch
import torchvision
from torch import nn
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
from torchvision import transforms
from PIL import Image
import torchvision.models as models
from tqdm import tqdm
from d2l import torch as d2l
import torch.optim as optim
from torch.optim.lr_scheduler import CosineAnnealingLR,
ReduceLROnPlateau
import numpy as np
import time
from sklearn.metrics import confusion_matrix,
classification_report
import seaborn as sns

# 数据路径
data_dir =
'./NNDL-Class/Project6/Data/dog-breed-identification'
train_dir = os.path.join(data_dir, 'train')
test_dir = os.path.join(data_dir, 'test')
labels_file = os.path.join(data_dir, 'labels.csv')

# 结果保存路径
result_dir = './NNDL-Class/Project6/Result/class'
os.makedirs(result_dir, exist_ok=True)

# 1. 数据整理：根据标签文件将图像移动到相应的子文件夹中
labels_df = pd.read_csv(labels_file)
```

```
# 创建目标文件夹，如果它不存在
for breed in labels_df['breed'].unique():
os.makedirs(os.path.join(train_dir, breed), exist_ok=True)

# 将图片移动到相应的文件夹
for _, row in labels_df.iterrows():
image_name = row['id'] + '.jpg'
breed = row['breed']
src_path = os.path.join(train_dir, image_name)
dst_path = os.path.join(train_dir, breed, image_name)
# 如果图片存在，则移动
if os.path.exists(src_path):
shutil.move(src_path, dst_path)


# 2. 数据加载和预处理 - 使用更丰富的数据增强
transform_train = transforms.Compose([
transforms.RandomResizedCrop(224, scale=(0.08, 1.0)),
transforms.RandomHorizontalFlip(),
transforms.RandomVerticalFlip(p=0.2),
transforms.RandomRotation(30),
transforms.ColorJitter(brightness=0.4, contrast=0.4,
saturation=0.4, hue=0.1),
transforms.RandomAffine(degrees=0, translate=(0.1, 0.1)),
transforms.RandomPerspective(distortion_scale=0.2, p=0.3),
transforms.RandomApply([
transforms.GaussianBlur(kernel_size=(5, 5), sigma=(0.1,
0.5))
], p=0.3),
transforms.ToTensor(),
transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224,
0.225])
])

transform_test = transforms.Compose([
transforms.Resize(256),
transforms.CenterCrop(224),
transforms.ToTensor(),
```

```
transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224,
0.225])
])

# 加载训练集和验证集
train_ds = torchvision.datasets.ImageFolder(train_dir,
transform=transform_train)
train_size = int(0.85 * len(train_ds)) # 85% 用于训练
valid_size = len(train_ds) - train_size # 15% 用于验证

train_data, valid_data =
torch.utils.data.random_split(train_ds, [train_size,
valid_size])

# 使用较小的批量大小以适应更深的模型
batch_size = 32
train_iter = DataLoader(train_data, batch_size=batch_size,
shuffle=True, num_workers=4, pin_memory=True)
valid_iter = DataLoader(valid_data, batch_size=batch_size,
shuffle=False, num_workers=4, pin_memory=True)

# 加载测试集图像（无需标签）
class TestDataset(torch.utils.data.Dataset):
def __init__(self, image_dir, transform=None):
self.image_dir = image_dir
self.transform = transform
self.image_files = [f for f in os.listdir(image_dir) if
f.endswith('.jpg')]

def __len__(self):
return len(self.image_files)

def __getitem__(self, idx):
img_name = os.path.join(self.image_dir,
self.image_files[idx])
image = Image.open(img_name).convert('RGB') # 确保 RGB 格式
if self.transform:
image = self.transform(image)
return image, self.image_files[idx] # 返回文件名用于预测结果
```

```
# 创建 test_images 文件夹（如果不存在）
test_images_dir = os.path.join(test_dir, 'test_images')
os.makedirs(test_images_dir, exist_ok=True)

# 移动 test 文件夹中的所有文件到 test_images
if not os.listdir(test_images_dir):
print("Organizing test set...")
for filename in os.listdir(test_dir):
file_path = os.path.join(test_dir, filename)
if os.path.isfile(file_path) and filename.endswith('.jpg'):
shutil.move(file_path, os.path.join(test_images_dir,
filename))
else:
print("Test set already organized.")

test_ds = TestDataset(test_images_dir,
transform=transform_test)
test_iter = DataLoader(test_ds, batch_size=batch_size,
shuffle=False, drop_last=False, num_workers=4)

def get_net(model_name='resnet50', freeze_backbone=True,
devices=None):
# 根据模型名称加载预训练模型
if model_name == 'resnet50':
model =
models.resnet50(weights=models.ResNet50_Weights.IMAGENET1
K_V1)
in_features = model.fc.in_features
elif model_name == 'resnet101':
model =
models.resnet101(weights=models.ResNet101_Weights.IMAGENE
T1K_V1)
in_features = model.fc.in_features
# 替换输出层以适应 120 个类别
if 'resnet' in model_name:
model.fc = nn.Sequential(
nn.Dropout(0.5),
```

```
nn.Linear(in_features, 1024),
nn.ReLU(),
nn.BatchNorm1d(1024),
nn.Dropout(0.3),
nn.Linear(1024, 512),
nn.ReLU(),
nn.BatchNorm1d(512),
nn.Linear(512, 120)
)
model = model.to(devices[0])
# 冻结特征提取层
if freeze_backbone:
for name, param in model.named_parameters():
if 'fc' not in name and 'classifier' not in name:
param.requires_grad = False
else:
param.requires_grad = True
else:
# 解冻最后两个阶段进行微调
for name, param in model.named_parameters():
if 'layer3' in name or 'layer4' in name or 'features.7' in
name or 'features.8' in name:
param.requires_grad = True
else:
param.requires_grad = False
# 打印可训练参数数量
total_params = sum(p.numel() for p in model.parameters())
trainable_params = sum(p.numel() for p in model.parameters()
if p.requires_grad)
print(f"Model: {model_name}, Total parameters:
{total_params:,}, Trainable parameters:
{trainable_params:,}")
return model

def train_model(model_name='resnet50', num_epochs=30,
lr=0.001, wd=1e-4,
freeze_backbone=True, scheduler_type='plateau',
devices=None):
```

```
# 创建模型特定的结果目录
model_result_dir = os.path.join(result_dir, model_name)
os.makedirs(model_result_dir, exist_ok=True)
# 获取当前时间戳用于保存唯一结果
timestamp = time.strftime("%Y%m%d_%H%M%S")
print(f"Training {model_name} for {num_epochs} epochs with
lr={lr}, wd={wd}")
# 加载模型
net = get_net(model_name, freeze_backbone, devices)
# 优化器 - 使用 AdamW 优化器
params_to_optimize = [p for p in net.parameters() if
p.requires_grad]
optimizer = optim.AdamW(params_to_optimize, lr=lr,
weight_decay=wd)
# 学习率调度器
if scheduler_type == 'plateau':
scheduler = ReduceLROnPlateau(optimizer, mode='max',
factor=0.5, patience=3)
elif scheduler_type == 'cosine':
scheduler = CosineAnnealingLR(optimizer, T_max=num_epochs,
eta_min=1e-6)
else:
scheduler = None
loss_fn = nn.CrossEntropyLoss(label_smoothing=0.1) # 添加标
签平滑
# 训练跟踪
history = {
'epoch': [],
'train_loss': [],
'train_acc': [],
'valid_loss': [],
'valid_acc': [],
'lr': []
}
best_valid_acc = 0.0
early_stop_counter = 0
early_stop_patience = 7
start_time = time.time()
```

```
for epoch_idx in range(num_epochs):
epoch = epoch_idx + 1
net.train()
running_loss = 0.0
running_corrects = 0
total_samples = 0
# 训练阶段
for inputs, labels in tqdm(train_iter, desc=f'Epoch
{epoch}/{num_epochs} Training'):
inputs, labels = inputs.to(devices[0]),
labels.to(devices[0])
optimizer.zero_grad()
outputs = net(inputs)
loss = loss_fn(outputs, labels)
loss.backward()
optimizer.step()
# 统计
_, preds = torch.max(outputs, 1)
running_loss += loss.item() * inputs.size(0)
running_corrects += torch.sum(preds == labels.data)
total_samples += inputs.size(0)
epoch_loss = running_loss / total_samples
epoch_acc = running_corrects.double() / total_samples
# 验证阶段
valid_loss, valid_acc, valid_preds, valid_labels =
evaluate_loss(valid_iter, net, devices)
# 更新学习率
current_lr = optimizer.param_groups[0]['lr']
if scheduler:
if scheduler_type == 'plateau':
scheduler.step(valid_acc)
else:
scheduler.step()
# 记录历史
history['epoch'].append(epoch)
history['train_loss'].append(epoch_loss)
history['train_acc'].append(epoch_acc.item())
history['valid_loss'].append(valid_loss)
```

```
history['valid_acc'].append(valid_acc)
history['lr'].append(current_lr)
# 打印统计信息
print(f'Epoch {epoch}/{num_epochs}: '
f'Train Loss: {epoch_loss:.4f} Acc: {epoch_acc:.4f} | '
f'Valid Loss: {valid_loss:.4f} Acc: {valid_acc:.4f} | '
f'LR: {current_lr:.6f}')
# 保存最佳模型
if valid_acc > best_valid_acc:
best_valid_acc = valid_acc
early_stop_counter = 0
save_model(net, f'{model_name}_best_model.pth',
model_result_dir)
print(f'Best model saved with validation accuracy:
{best_valid_acc:.4f}')
# 保存验证集的预测结果用于后续分析
save_validation_results(valid_preds, valid_labels,
valid_data,
f'{model_name}_best_val_preds.csv', model_result_dir)
else:
early_stop_counter += 1
print(f'No improvement for
{early_stop_counter}/{early_stop_patience} epochs')
# 早停检查
if early_stop_counter >= early_stop_patience:
print(f'Early stopping at epoch {epoch} after
{early_stop_patience} epochs without improvement')
break
# 训练完成后保存最终结果
training_time = time.time() - start_time
print(f'Training completed in {training_time//60:.0f}m
{training_time%60:.0f}s')
# 保存最终模型
save_model(net, f'{model_name}_final_model.pth',
model_result_dir)
# 保存训练曲线
```

```
save_training_curve(history,
f'{model_name}_final_training_curve.png',
model_result_dir)
# 保存混淆矩阵
_, _, valid_preds, valid_labels = evaluate_loss(valid_iter,
net, devices)
save_confusion_matrix(valid_preds, valid_labels,
valid_data,
f'{model_name}_final_confusion_matrix.png',
model_result_dir)
# 保存分类报告
save_classification_report(valid_preds, valid_labels,
valid_data,
f'{model_name}_classification_report.txt',
model_result_dir)
# 保存训练历史
save_training_history(history,
f'{model_name}_training_history.csv', model_result_dir)
# 在测试集上进行预测
save_predictions(net, test_iter,
f'{model_name}_submission.csv', model_result_dir, devices)
return net, history

def evaluate_loss(data_iter, net, devices):
net.eval()
running_loss = 0.0
running_corrects = 0
total_samples = 0
all_preds = []
all_labels = []
with torch.no_grad():
for inputs, labels in tqdm(data_iter, desc='Evaluating'):
inputs, labels = inputs.to(devices[0]),
labels.to(devices[0])
outputs = net(inputs)
loss = nn.CrossEntropyLoss()(outputs, labels)
_, preds = torch.max(outputs, 1)
running_loss += loss.item() * inputs.size(0)
```

```
running_corrects += torch.sum(preds == labels.data)
total_samples += inputs.size(0)
all_preds.extend(preds.cpu().numpy())
all_labels.extend(labels.cpu().numpy())
epoch_loss = running_loss / total_samples
epoch_acc = running_corrects.double() / total_samples
return epoch_loss, epoch_acc.item(), np.array(all_preds),
np.array(all_labels)

def save_training_curve(history, filename, save_dir):
plt.figure(figsize=(15, 10))
# 损失曲线
plt.subplot(1, 3, 1)
plt.plot(history['epoch'], history['train_loss'], 'b-',
label="Train Loss")
plt.plot(history['epoch'], history['valid_loss'], 'r-',
label="Valid Loss")
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.title('Training and Validation Loss')
plt.grid(True)
# 准确率曲线
plt.subplot(1, 3, 2)
plt.plot(history['epoch'], history['train_acc'], 'b-',
label="Train Accuracy")
plt.plot(history['epoch'], history['valid_acc'], 'r-',
label="Valid Accuracy")
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Training and Validation Accuracy')
plt.grid(True)
# 最佳准确率标记
best_acc = max(history['valid_acc'])
best_epoch =
history['epoch'][np.argmax(history['valid_acc'])]
plt.subplot(1, 3, 3)
```

```
plt.text(0.1, 0.5, f'Best Validation Accuracy:
{best_acc:.4f}\nat Epoch: {best_epoch}',
fontsize=12, bbox=dict(facecolor='white', alpha=0.5))
plt.axis('off')
plt.tight_layout()
plt.savefig(os.path.join(save_dir, filename), dpi=300)
plt.close()

def save_confusion_matrix(preds, labels, dataset, filename,
save_dir):
# 获取类别名称
class_names = list(dataset.dataset.class_to_idx.keys())
# 计算混淆矩阵
cm = confusion_matrix(labels, preds)
plt.figure(figsize=(20, 20))
sns.heatmap(cm, annot=False, fmt='d', cmap='Blues',
xticklabels=class_names, yticklabels=class_names)
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.xticks(rotation=90, fontsize=6)
plt.yticks(fontsize=6)
plt.tight_layout()
plt.savefig(os.path.join(save_dir, filename), dpi=300)
plt.close()

def save_classification_report(preds, labels, dataset,
filename, save_dir):
class_names = list(dataset.dataset.class_to_idx.keys())
report = classification_report(labels, preds,
target_names=class_names)
with open(os.path.join(save_dir, filename), 'w') as f:
f.write(report)

def save_training_history(history, filename, save_dir):
df = pd.DataFrame(history)
df.to_csv(os.path.join(save_dir, filename), index=False)
```

```
def save_validation_results(preds, labels, dataset,
filename, save_dir):
class_names = list(dataset.dataset.class_to_idx.keys())
true_classes = [class_names[i] for i in labels]
pred_classes = [class_names[i] for i in preds]
# 获取原始图像路径
image_paths = [dataset.dataset.samples[i][0] for i in
dataset.indices]
df = pd.DataFrame({
'image_path': image_paths,
'true_label': true_classes,
'pred_label': pred_classes,
'correct': [t == p for t, p in zip(true_classes,
pred_classes)]
})
df.to_csv(os.path.join(save_dir, filename), index=False)

def save_model(net, filename, save_dir):
torch.save(net.state_dict(), os.path.join(save_dir,
filename))
print(f'Model saved to {os.path.join(save_dir, filename)}')

def save_predictions(net, test_iter, filename, save_dir,
devices):
net.eval()
all_probs = []
all_ids = []
with torch.no_grad():
for images, filenames in tqdm(test_iter, desc='Predicting
Test Set'):
images = images.to(devices[0])
outputs = net(images)
probs = torch.nn.functional.softmax(outputs, dim=1)
all_probs.extend(probs.cpu().numpy())
all_ids.extend([fname.replace('.jpg', '') for fname in
filenames])
# 创建提交文件
class_names = sorted(os.listdir(train_dir))
```

```
submission_df = pd.DataFrame(all_probs,
columns=class_names)
submission_df.insert(0, 'id', all_ids)
submission_df.to_csv(os.path.join(save_dir, filename),
index=False)
print(f'Predictions saved to {os.path.join(save_dir,
filename)}')

# 获取设备
devices = d2l.try_all_gpus()
print(f"Using devices: {devices}")

# 训练不同模型并保存结果
models_to_train = [
{'model_name': 'resnet50', 'num_epochs': 40, 'lr': 0.001,
'freeze_backbone': False, 'scheduler_type': 'plateau'},
{'model_name': 'resnet101', 'num_epochs': 40, 'lr': 0.0005,
'freeze_backbone': False, 'scheduler_type': 'plateau'},
]

all_results = {}

for config in models_to_train:
print(f"\n{'='*50}")
print(f"Training {config['model_name']} model")
print(f"{'='*50}")
model, history = train_model(
model_name=config['model_name'],
num_epochs=config['num_epochs'],
lr=config['lr'],
wd=1e-4,
freeze_backbone=config['freeze_backbone'],
scheduler_type=config['scheduler_type'],
devices=devices
)
all_results[config['model_name']] = {
'history': history,
'best_val_acc': max(history['valid_acc'])
}
```

```
# 释放内存
del model
torch.cuda.empty_cache()

# 比较不同模型的结果
print("\nModel Comparison Results:")
for model_name, result in all_results.items():
print(f"{model_name}: Best Validation Accuracy =
{result['best_val_acc']:.4f}")

# 绘制所有模型的验证准确率比较
plt.figure(figsize=(12, 8))
for model_name, result in all_results.items():
plt.plot(result['history']['epoch'],
result['history']['valid_acc'],
label=f"{model_name} (best:
{result['best_val_acc']:.4f})")

plt.xlabel('Epoch')
plt.ylabel('Validation Accuracy')
plt.title('Model Comparison: Validation Accuracy')
plt.legend()
plt.grid(True)
plt.savefig(os.path.join(result_dir,
'model_comparison.png'), dpi=300)
plt.close()
```

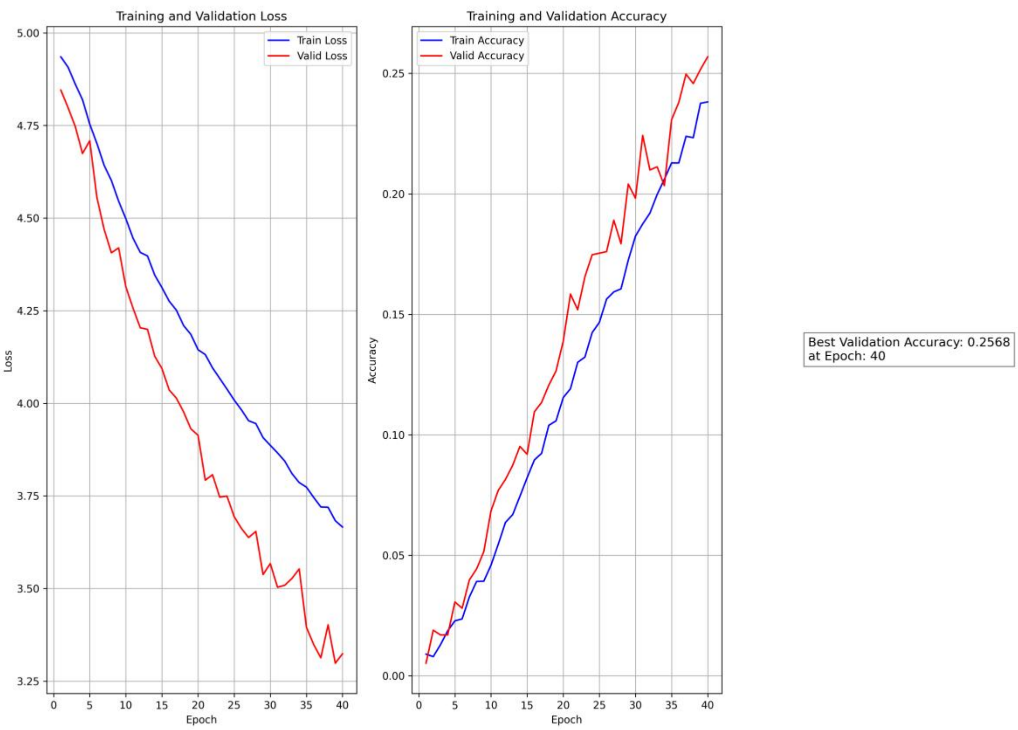
实验结果:

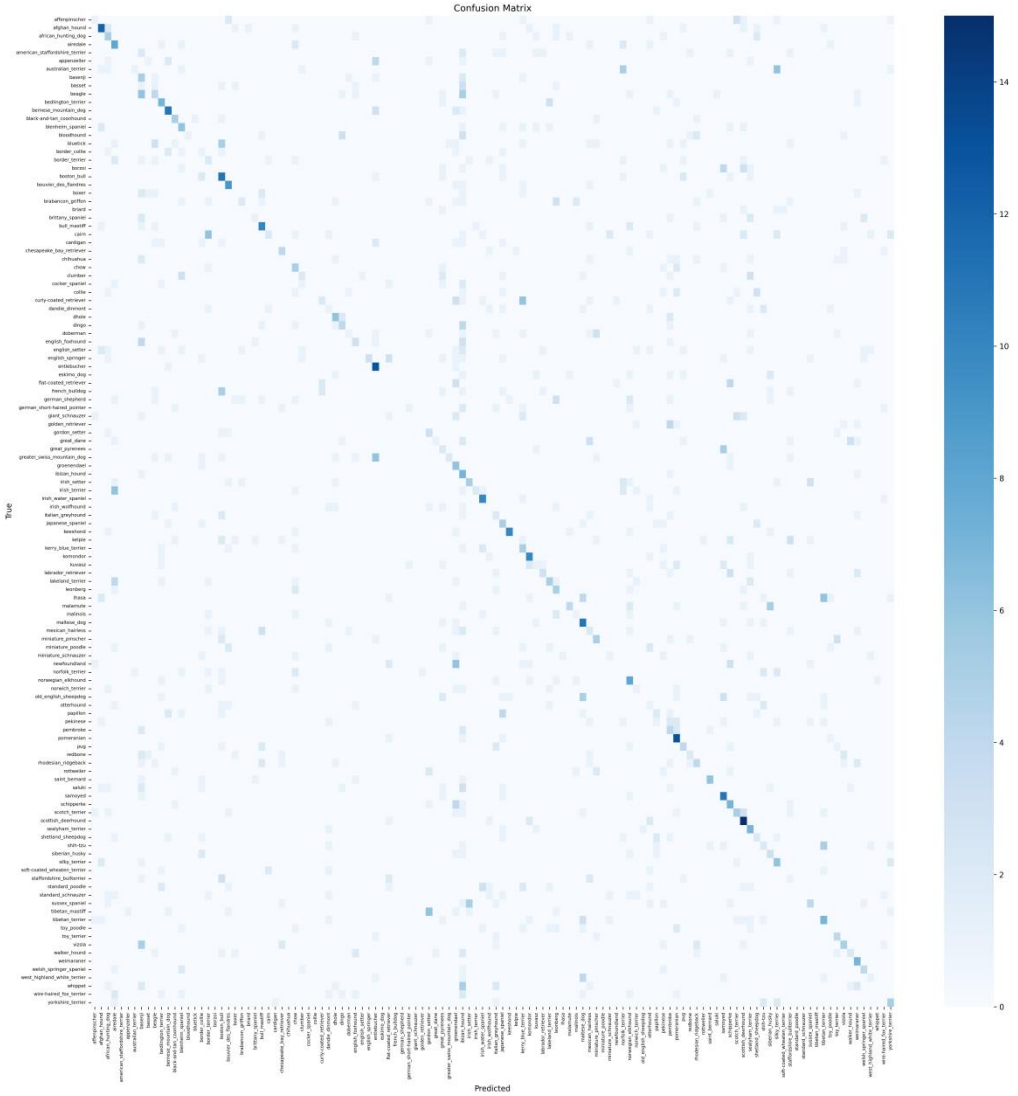
```
(yyzttt) (base) yyz@4028Dog:~$ /usr/local/anaconda3/envs/yyzttt/bin/python /home/yyz/NNDL-Class/Project6/Cod
e/class_deep.py
Test set already organized.
Using devices: [device(type='cuda', index=0), device(type='cuda', index=1), device(type='cuda', index=2), de
vice(type='cuda', index=3), device(type='cuda', index=4), device(type='cuda', index=5), device(type='cuda',
index=6), device(type='cuda', index=7)]

==================================================
Training resnet50 model
==================================================
Training resnet50 for 40 epochs with lr=0.001, wd=0.0001
Model: resnet50, Total parameters: 26,195,640, Trainable parameters: 22,063,104
Epoch 1/40 Training: 100%|| 272/272 [00:25<00:00, 10.64it/s]
Evaluating: 100%|| 48/48 [00:04<00:00, 10.02it/s]
Epoch 1/40: Train Loss: 4.9353 Acc: 0.0090 | Valid Loss: 4.8456 Acc: 0.0052 | LR: 0.001000
Model saved to ./NNDL-Class/Project6/Result/class/resnet50/resnet50_best_model.pth
Best model saved with validation accuracy: 0.0052
Epoch 2/40 Training: 100%|| 272/272 [00:25<00:00, 10.87it/s]
Evaluating: 100%|| 48/48 [00:04<00:00, 10.31it/s]
Epoch 2/40: Train Loss: 4.9075 Acc: 0.0079 | Valid Loss: 4.7981 Acc: 0.0189 | LR: 0.001000
Model saved to ./NNDL-Class/Project6/Result/class/resnet50/resnet50_best_model.pth
Best model saved with validation accuracy: 0.0189
Epoch 3/40 Training: 100%|| 272/272 [00:24<00:00, 10.94it/s]
Evaluating: 100%|| 48/48 [00:04<00:00, 10.65it/s]
Epoch 3/40: Train Loss: 4.8616 Acc: 0.0129 | Valid Loss: 4.7472 Acc: 0.0169 | LR: 0.001000
No improvement for 1/7 epochs
Epoch 4/40 Training: 100%|| 272/272 [00:24<00:00, 11.03it/s]
Evaluating: 100%|| 48/48 [00:04<00:00, 11.00it/s]
Epoch 4/40: Train Loss: 4.8195 Acc: 0.0186 | Valid Loss: 4.6744 Acc: 0.0169 | LR: 0.001000
No improvement for 2/7 epochs
Epoch 5/40 Training: 100%|| 272/272 [00:25<00:00, 10.85it/s]
Evaluating: 100%|| 48/48 [00:04<00:00, 10.64it/s]
```

```
ro_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
Predicting Test Set: 100%|| 324/324 [00:12<00:00, 25.40it/s]
Predictions saved to ./NNDL-Class/Project6/Result/class/resnet50/resnet50_submission.csv

==================================================
Training resnet101 model
==================================================
Training resnet101 for 40 epochs with lr=0.0005, wd=0.0001
Model: resnet101, Total parameters: 45,187,768, Trainable parameters: 41,055,232
Epoch 1/40 Training: 100%|| 272/272 [00:26<00:00, 10.31it/s]
Evaluating: 100%|| 48/48 [00:04<00:00,  9.91it/s]
Epoch 1/40: Train Loss: 4.9374 Acc: 0.0083 | Valid Loss: 10.5910 Acc: 0.0078 | LR: 0.000500
Model saved to ./NNDL-Class/Project6/Result/class/resnet101/resnet101_best_model.pth
Best model saved with validation accuracy: 0.0078
Epoch 2/40 Training: 100%|| 272/272 [00:25<00:00, 10.50it/s]
Evaluating: 100%|| 48/48 [00:04<00:00,  9.89it/s]
Epoch 2/40: Train Loss: 4.9404 Acc: 0.0083 | Valid Loss: 4.8634 Acc: 0.0104 | LR: 0.000500
Model saved to ./NNDL-Class/Project6/Result/class/resnet101/resnet101_best_model.pth
Best model saved with validation accuracy: 0.0104
Epoch 3/40 Training: 100%|| 272/272 [00:26<00:00, 10.39it/s]
Evaluating: 100%|| 48/48 [00:04<00:00, 10.20it/s]
Epoch 3/40: Train Loss: 4.9331 Acc: 0.0092 | Valid Loss: 4.8073 Acc: 0.0143 | LR: 0.000500
Model saved to ./NNDL-Class/Project6/Result/class/resnet101/resnet101_best_model.pth
Best model saved with validation accuracy: 0.0143
Epoch 4/40 Training: 100%|| 272/272 [00:25<00:00, 10.47it/s]
Evaluating: 100%|| 48/48 [00:04<00:00,  9.96it/s]
Epoch 4/40: Train Loss: 4.9185 Acc: 0.0105 | Valid Loss: 4.8019 Acc: 0.0143 | LR: 0.000500
No improvement for 1/7 epochs
Epoch 5/40 Training: 100%|| 272/272 [00:26<00:00, 10.44it/s]
Evaluating: 100%|| 48/48 [00:05<00:00,  9.31it/s]
Epoch 5/40: Train Loss: 4.9050 Acc: 0.0105 | Valid Loss: 5.9193 Acc: 0.0098 | LR: 0.000500
No improvement for 2/7 epochs
Epoch 6/40 Training: 100%|| 272/272 [00:25<00:00, 10.49it/s]
Evaluating: 100%|| 48/48 [00:05<00:00,  9.51it/s]
Epoch 6/40: Train Loss: 4.8689 Acc: 0.0117 | Valid Loss: 4.7675 Acc: 0.0137 | LR: 0.000500
No improvement for 3/7 epochs
Epoch 7/40 Training: 100%|| 272/272 [00:26<00:00, 10.44it/s]
Evaluating: 100%|| 48/48 [00:04<00:00,  9.73it/s]
Epoch 7/40: Train Loss: 4.8463 Acc: 0.0148 | Valid Loss: 4.7174 Acc: 0.0209 | LR: 0.000500
```

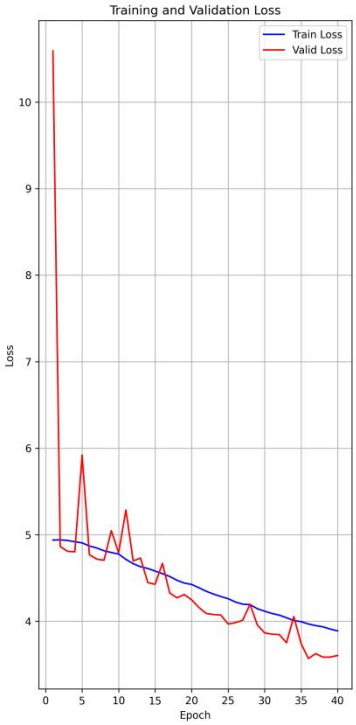
```
Epoch 38/40: Train Loss: 3.9345 Acc: 0.1703 | Valid Loss: 3.5831 Acc: 0.2014 | LR: 0.000500
No improvement for 2/7 epochs
Epoch 39/40 Training: 100%|| 272/272 [00:26<00:00, 10.36it/s]
Evaluating: 100%|| 48/48 [00:04<00:00, 10.46it/s]
Epoch 39/40: Train Loss: 3.9086 Acc: 0.1826 | Valid Loss: 3.5852 Acc: 0.2138 | LR: 0.000500
Model saved to ./NNDL-Class/Project6/Result/class/resnet101/resnet101_best_model.pth
Best model saved with validation accuracy: 0.2138
Epoch 40/40 Training: 100%|| 272/272 [00:26<00:00, 10.35it/s]
Evaluating: 100%|| 48/48 [00:04<00:00, 10.09it/s]
Epoch 40/40: Train Loss: 3.8878 Acc: 0.1829 | Valid Loss: 3.6034 Acc: 0.2093 | LR: 0.000500
No improvement for 1/7 epochs
Training completed in 21m 39s
Model saved to ./NNDL-Class/Project6/Result/class/resnet101/resnet101_final_model.pth
Evaluating: 100%|| 48/48 [00:04<00:00, 10.17it/s]
/usr/local/anaconda3/envs/yyzttt/lib/python3.9/site-packages/sklearn/metrics/_classification.py:1565: Undefi
nedMetricWarning: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `ze
ro_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/anaconda3/envs/yyzttt/lib/python3.9/site-packages/sklearn/metrics/_classification.py:1565: Undefi
nedMetricWarning: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `ze
ro_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/anaconda3/envs/yyzttt/lib/python3.9/site-packages/sklearn/metrics/_classification.py:1565: Undefi
nedMetricWarning: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `ze
ro_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
Predicting Test Set: 100%|| 324/324 [00:15<00:00, 21.59it/s]
Predictions saved to ./NNDL-Class/Project6/Result/class/resnet101/resnet101_submission.csv

Model Comparison Results:
resnet50: Best Validation Accuracy = 0.2568
resnet101: Best Validation Accuracy = 0.2138
```

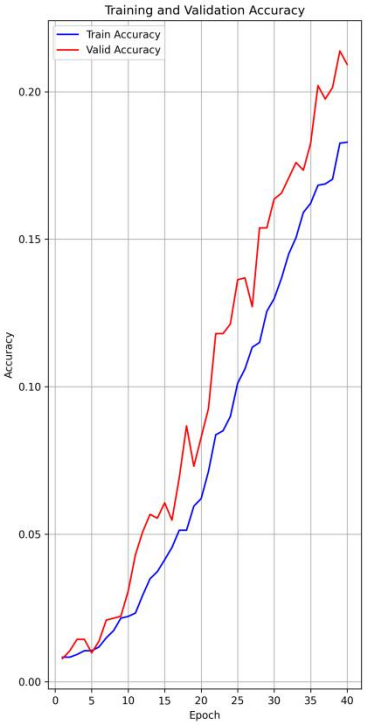
ResNet50:



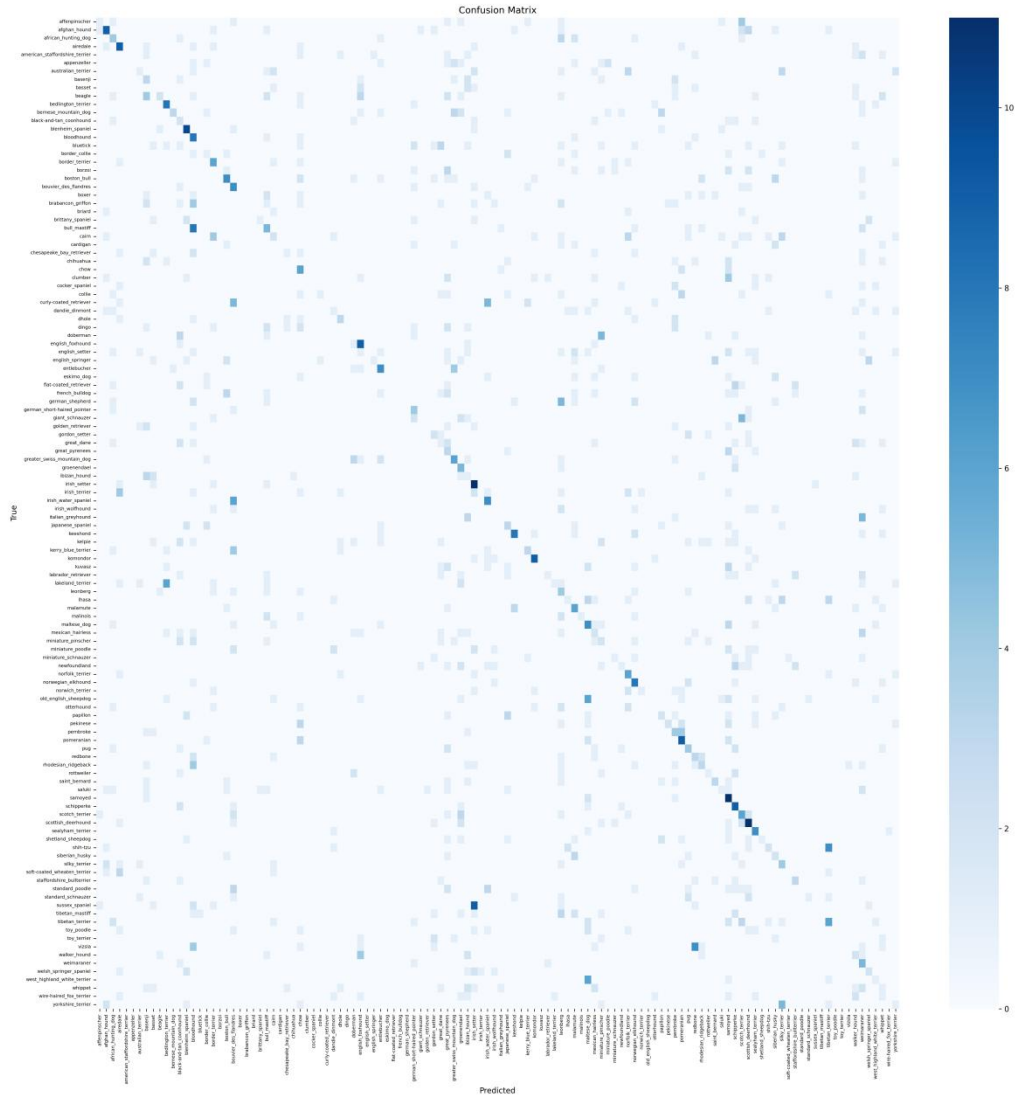


ResNet101:





Best Validation Accuracy: 0.2138
at Epoch: 39



在实验中，我尝试使用更深的预训练模型（如 ResNet101）来进行狗的品种识别，然而结果显示其最佳验证准确率（0.2138）反而低于较浅的 ResNet50（0.2568）。这表明更深的模型未必能带来更好的结果，可能是因为更深的网络需要更多的训练数据和计算资源，而当前数据集规模有限，导致模型出现过拟合或训练不充分的情况。

在超参数调整方面，对于更深的模型，我减小了批次大小至 32 以适配 GPU 内存，但学习率设置为 0.0005（低于 ResNet50 的 0.001），或许可以尝试使用学习率预热策略或

动态调整学习率的调度器（如余弦退火）来优化收敛过程。

同时，权重衰减参数可适当增大以抑制过拟合，比如从 1e-4 调整为 1e-3。此外，数据增强策略可进一步丰富，如增加旋转角度范围、添加更多几何变换等，以提升模型对不同姿态和视角的鲁棒性。还可以尝试解冻更多的网络层进行微调，而非仅解冻最后两个残差块，比如逐步解冻中间层以提取更丰富的特征。若想进一步改善结果，可考虑引入更大的数据集、使用混合精度训练加速收敛，或结合注意力机制增强模型对关键特征的捕捉能力，同时通过早停策略避免训练过程中的性能退化，这些调整或许能让更深的模型发挥出应有的优势。

2. 风格迁移：

实验代码：

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
from torch.utils.data import DataLoader
from torchvision import transforms
from PIL import Image
import matplotlib.pyplot as plt
from tqdm import tqdm
# 图像的预处理函数
def preprocess(img, image_shape):
transform = transforms.Compose([
transforms.Resize(image_shape),
transforms.ToTensor(),
```

```
transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225])
])
return transform(img).unsqueeze(0)

# 后处理函数
def postprocess(img):
img = img.squeeze(0) # 去掉 batch_size 维度
img = torch.clamp(img.permute(1, 2, 0) * torch.tensor([0.229,
0.224, 0.225]).to('cuda') + torch.tensor([0.485, 0.456,
0.406]).to('cuda'), 0, 1)
return transforms.ToPILImage()(img.permute(2, 0, 1)) # 返回
图片

# 加载预训练的 VGG-19 模型
pretrained_net =
torchvision.models.vgg19(pretrained=True).features

# 将 VGG 模型移动到正确的设备
device = torch.device('cuda' if torch.cuda.is_available()
else 'cpu')
pretrained_net = pretrained_net.to(device)

# 使用 VGG 网络的某些层来提取内容和风格特征
content_layers = [21] # 选择 VGG 的某层作为内容层
style_layers = [0, 5, 10, 19, 28] # 选择 VGG 的某些层作为风格层

# 提取特征的函数
def extract_features(X, content_layers, style_layers):
contents = []
styles = []
for i in range(len(pretrained_net)):
X = pretrained_net[i](X)
if i in style_layers:
styles.append(X)
if i in content_layers:
contents.append(X)
return contents, styles
```

```
# 计算内容损失
def content_loss(Y_hat, Y):
return torch.square(Y_hat - Y.detach()).mean()

# 计算风格损失
def gram(X):
num_channels, n = X.shape[1], X.numel() // X.shape[1]
X = X.reshape((num_channels, n))
return torch.matmul(X, X.T) / (num_channels * n)

def style_loss(Y_hat, gram_Y):
return torch.square(gram(Y_hat) - gram_Y.detach()).mean()

# 计算全变分损失
def tv_loss(Y_hat):
return 0.5 * (torch.abs(Y_hat[:, :, 1:, :] -
Y_hat[:, :, :-1, :]).mean() +
torch.abs(Y_hat[:, :, :, 1:] - Y_hat[:, :, :, :-1]).mean())

# 风格迁移的损失函数
def compute_loss(X, contents_Y_hat, styles_Y_hat,
contents_Y, styles_Y_gram):
content_loss_val = [content_loss(Y_hat, Y) for Y_hat, Y in
zip(contents_Y_hat, contents_Y)]
style_loss_val = [style_loss(Y_hat, gram_Y) for Y_hat,
gram_Y in zip(styles_Y_hat, styles_Y_gram)]
tv_loss_val = tv_loss(X)
return sum(content_loss_val + style_loss_val +
[tv_loss_val])

# 读取内容图像和风格图像
content_img = Image.open('./Data/Content.jpg') # 内容图像
style_img = Image.open('./Data/Style.jpg') # 风格图像

# 设置图像形状和设备
image_shape = (300, 450) # 图像大小
device = torch.device('cuda' if torch.cuda.is_available()
else 'cpu')
```

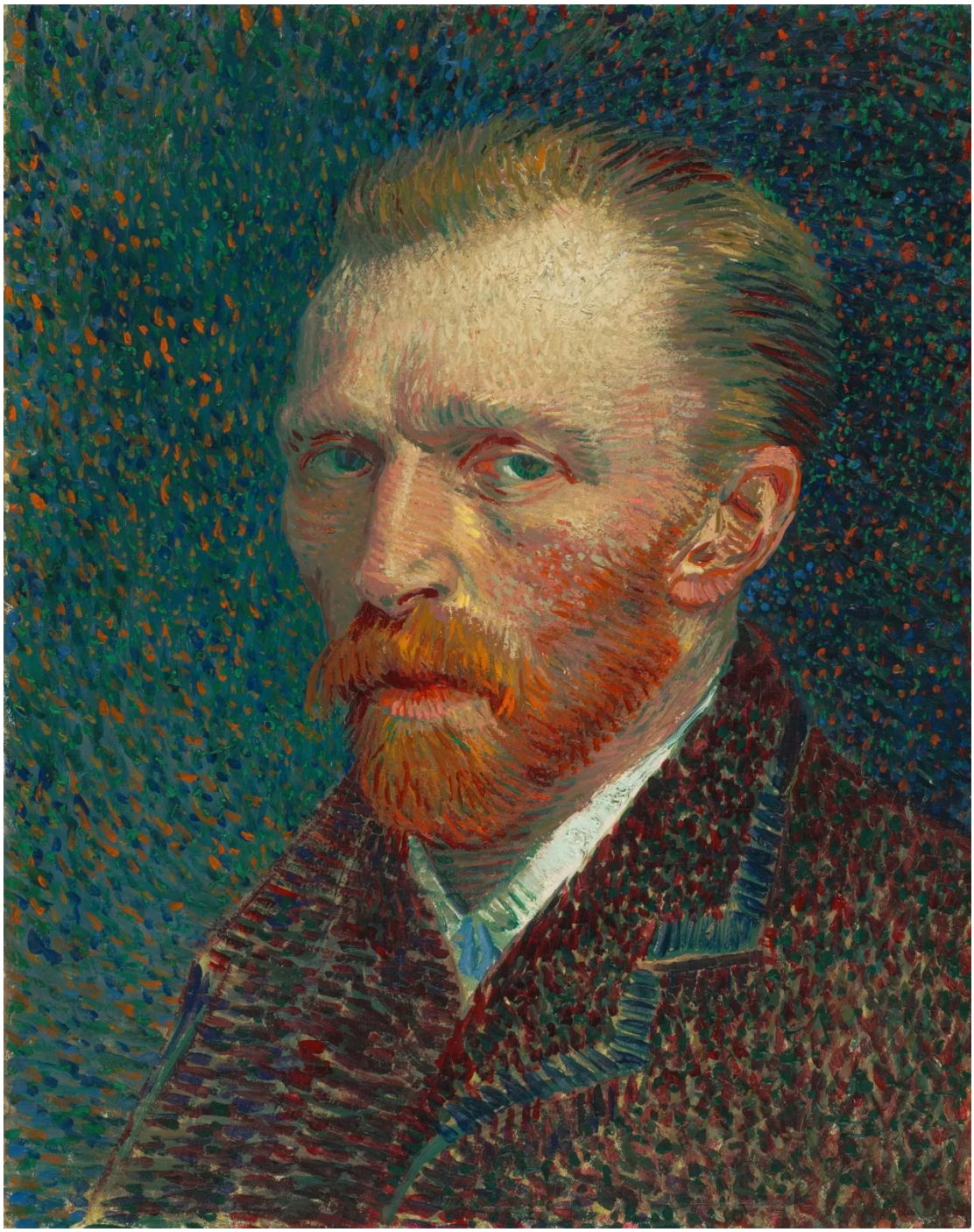
```
# 预处理并提取内容和风格特征
content_X = preprocess(content_img, image_shape).to(device)
style_X = preprocess(style_img, image_shape).to(device)

# 使用 VGG 模型提取内容和风格特征
content_Y, _ = extract_features(content_X, content_layers,
style_layers)
_, style_Y = extract_features(style_X, content_layers,
style_layers)

# 计算风格的 Gram 矩阵
style_Y_gram = [gram(y) for y in style_Y]

# 初始化合成图像
generated_image =
content_X.clone().requires_grad_(True).to(device)

# 定义优化器
optimizer = optim.Adam([generated_image], lr=0.3)

# 存储每个周期的损失
epoch_losses = []

# 训练模型
num_epochs = 500
for epoch in tqdm(range(num_epochs), desc="Training"):
optimizer.zero_grad()
contents_Y_hat, styles_Y_hat =
extract_features(generated_image, content_layers,
style_layers)
loss = compute_loss(generated_image, contents_Y_hat,
styles_Y_hat, content_Y, style_Y_gram)
loss.backward()
optimizer.step()

epoch_losses.append(loss.item())
if (epoch + 1) % 50 == 0:
print(f'Epoch [{epoch+1}/{num_epochs}], Loss:
{loss.item():.4f}')
```

```
# 最终输出风格迁移后的图像
final_img = postprocess(generated_image)
final_img.save('./Result/style/final_styled_image.jpg')

# 绘制并保存训练曲线
plt.figure()
plt.plot(range(1, num_epochs + 1), epoch_losses,
label='Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss Curve')
plt.legend()
plt.savefig('./Result/style/training_loss_curve.jpg')
```

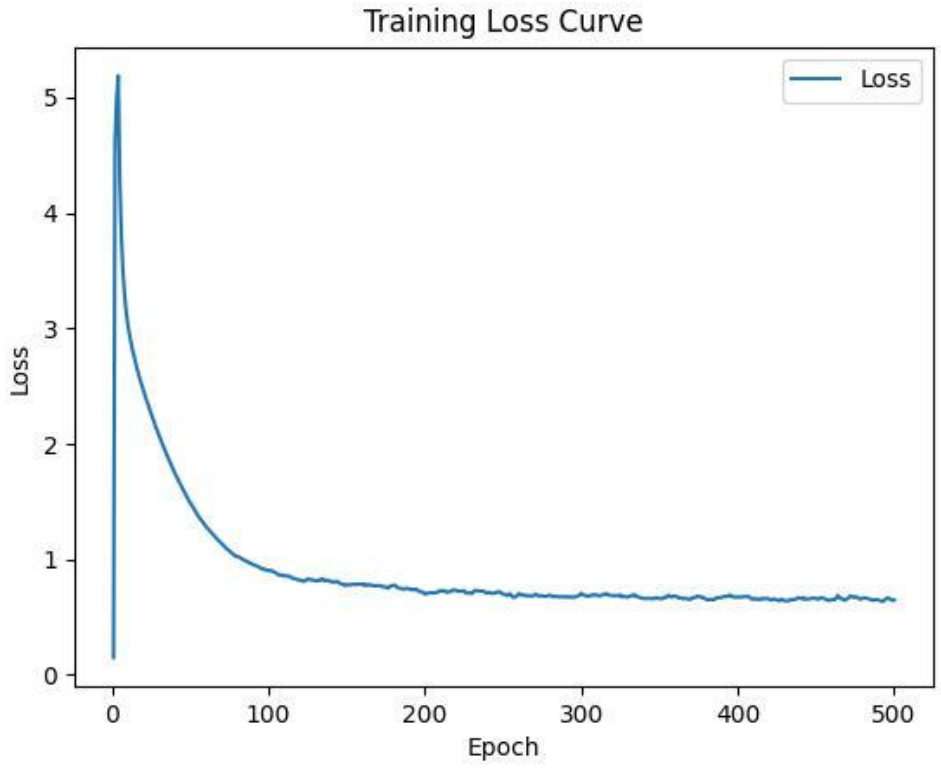
输入图像：

内容：

风格：



实验结果：


Training Loss Curve
Loss
Loss
Epoch
0
1
2
3
4
5
0
100
200
300
400
500

练习题 1：选择不同的内容和风格层，输出有什么变化？

实验代码：

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
from PIL import Image
import matplotlib.pyplot as plt
from torchvision import transforms
from tqdm import tqdm
# 预处理函数
def preprocess(img, image_shape):
transform = transforms.Compose([
transforms.Resize(image_shape),
transforms.ToTensor(),
transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225])
])
return transform(img).unsqueeze(0)

# 后处理函数
def postprocess(img):
img = img.squeeze(0) # 去掉 batch_size 维度
img = torch.clamp(img.permute(1, 2, 0) * torch.tensor([0.229,
0.224, 0.225]).to('cuda') + torch.tensor([0.485, 0.456,
0.406]).to('cuda'), 0, 1)
```

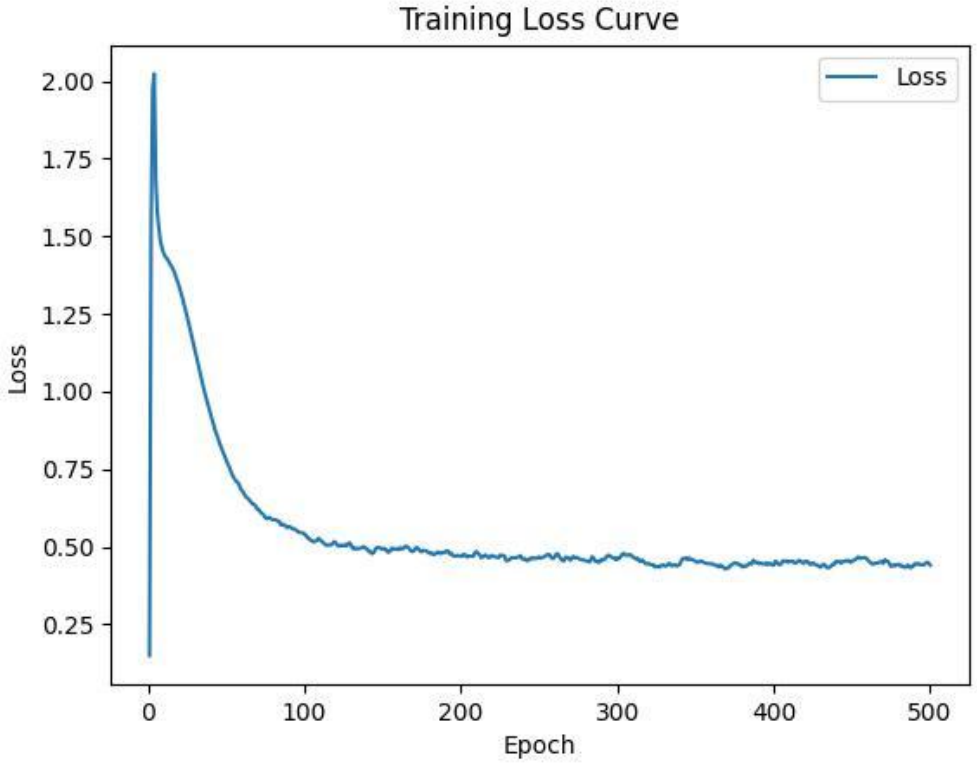
```
return transforms.ToPILImage()(img.permute(2, 0, 1)) # 返回图片

# 加载预训练的 VGG-19 模型
pretrained_net = torchvision.models.vgg19(pretrained=True).features
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
pretrained_net = pretrained_net.to(device)

# 内容层和风格层的选择
# 不同的层会影响结果
content_layers = [25] # 选择 VGG 的较高层作为内容层（更注重结构）
style_layers = [0, 5, 10, 19, 28] # 选择 VGG 的不同层作为风格层

# 特征提取
def extract_features(X, content_layers, style_layers):
contents = []
styles = []
for i in range(len(pretrained_net)):
X = pretrained_net[i](X)
if i in style_layers:
styles.append(X)
if i in content_layers:
contents.append(X)
return contents, styles

# 计算内容损失
def content_loss(Y_hat, Y):
return torch.square(Y_hat - Y.detach()).mean()

# 计算风格损失
def gram(X):
num_channels, n = X.shape[1], X.numel() // X.shape[1]
X = X.reshape((num_channels, n))
return torch.matmul(X, X.T) / (num_channels * n)

def style_loss(Y_hat, gram_Y):
return torch.square(gram(Y_hat) - gram_Y.detach()).mean()
```

```
# 计算全变分损失
def tv_loss(Y_hat):
return 0.5 * (torch.abs(Y_hat[:, :, 1:, :] -
Y_hat[:, :, :-1, :]).mean() +
torch.abs(Y_hat[:, :, :, 1:] - Y_hat[:, :, :, :-1]).mean())

# 风格迁移的损失函数
def compute_loss(X, contents_Y_hat, styles_Y_hat,
contents_Y, styles_Y_gram):
content_loss_val = [content_loss(Y_hat, Y) for Y_hat, Y in
zip(contents_Y_hat, contents_Y)]
style_loss_val = [style_loss(Y_hat, gram_Y) for Y_hat,
gram_Y in zip(styles_Y_hat, styles_Y_gram)]
tv_loss_val = tv_loss(X)
return sum(content_loss_val + style_loss_val +
[tv_loss_val])

# 读取内容图像和风格图像
content_img = Image.open('./Data/Content.jpg') # 内容图像
style_img = Image.open('./Data/Style.jpg') # 风格图像

# 设置图像形状和设备
image_shape = (300, 450) # 图像大小
device = torch.device('cuda' if torch.cuda.is_available()
else 'cpu')

# 预处理并提取内容和风格特征
content_X = preprocess(content_img, image_shape).to(device)
style_X = preprocess(style_img, image_shape).to(device)

# 使用 VGG 模型提取内容和风格特征
content_Y, _ = extract_features(content_X, content_layers,
style_layers)
_, style_Y = extract_features(style_X, content_layers,
style_layers)

# 计算风格的 Gram 矩阵
style_Y_gram = [gram(y) for y in style_Y]
```

```
# 初始化合成图像
generated_image =
content_X.clone().requires_grad_(True).to(device)

# 定义优化器
optimizer = optim.Adam([generated_image], lr=0.3)

# 存储每个周期的损失
epoch_losses = []

# 训练模型
num_epochs = 500
for epoch in tqdm(range(num_epochs), desc="Training"):
optimizer.zero_grad()
contents_Y_hat, styles_Y_hat =
extract_features(generated_image, content_layers,
style_layers)
loss = compute_loss(generated_image, contents_Y_hat,
styles_Y_hat, content_Y, style_Y_gram)
loss.backward()
optimizer.step()

epoch_losses.append(loss.item())
if (epoch + 1) % 50 == 0:
print(f'Epoch [{epoch+1}/{num_epochs}], Loss:
{loss.item():.4f}')

# 最终输出风格迁移后的图像
final_img = postprocess(generated_image)
final_img.save('./Result/style/final_styled_image_diff_la
yers.jpg')



# 绘制并保存训练曲线
plt.figure()
plt.plot(range(1, num_epochs + 1), epoch_losses,
label='Loss')
plt.xlabel('Epoch')
```

```
plt.ylabel('Loss')
plt.title('Training Loss Curve')
plt.legend()
plt.savefig('./Result/style/training_loss_curvee_diff_layers.jpg')
```

实验结果：




```
(yyzttt) (base) yyz@4028Dog:~/NNDL-Class/Project6$ python ./Code/style_layer.py
/usr/local/anaconda3/envs/yyzttt/lib/python3.9/site-packages/torchvision/models/_utils.py:208:
The parameter 'pretrained' is deprecated since 0.13 and may be removed in the future, please us
nstead.
  warnings.warn(
/usr/local/anaconda3/envs/yyzttt/lib/python3.9/site-packages/torchvision/models/_utils.py:223:
Arguments other than a weight enum or `None` for 'weights' are deprecated since 0.13 and may be
he future. The current behavior is equivalent to passing `weights=VGG19_Weights.IMAGENET1K_V1`.
 use `weights=VGG19_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
Training:  10%|                                                  | 49/500 [00:01<00:1
Epoch [50/500], Loss: 0.7781
Training:  19%|                                                  | 97/500 [00:03<00:1
Epoch [100/500], Loss: 0.5410
Training:  30%|                                                  | 148/500 [00:05<00:1
Epoch [150/500], Loss: 0.4925
Training:  40%|                                                  | 199/500 [00:07<00:1
Epoch [200/500], Loss: 0.4716
Training:  49%|                                                  | 245/500 [00:08<00:0
Epoch [250/500], Loss: 0.4680
Training:  59%|                                                  | 295/500 [00:09<00:0
Epoch [300/500], Loss: 0.4640
Training:  69%|                                                  | 345/500 [00:10<00:0
Epoch [350/500], Loss: 0.4526
Training:  79%|                                                  | 396/500 [00:12<00:0
Epoch [400/500], Loss: 0.4411
Training:  90%|                                                  | 449/500 [00:13<00:0
Epoch [450/500], Loss: 0.4525
Training:  99%|                                                  | 497/500 [00:14<00:0
Epoch [500/500], Loss: 0.4408
Training: 100%|                                                  | 500/500 [00:14<00:0
```

在风格迁移实验中，选择不同的内容和风格层会显著影响输出结果。原始实验中内容层选取 VGG19 的第 21 层，该层属于中等深度特征层，既能保留一定细节又具备基础语义

结构；而当内容层调整为第 25 层（更高层特征）时，由于高层特征更关注图像的整体语义结构（如物体轮廓、布局等），生成的图像会呈现出更抽象的内容结构 —— 细节纹理减少，但内容的主体轮廓和布局保留更完整。例如，若原始内容图像是风景，高层内容层会让生成图像的山川河流框架更突出，而树木枝叶等细节可能被风格化的笔触替代。

风格层的选择同样影响显著：原始实验中风格层包含第 0、5、10、19、28 层，覆盖了从低级纹理（如第 0 层的边缘、颜色）到高级风格特征（如第 28 层的全局风格模式）的多尺度信息。若减少风格层数量或调整层级，比如仅使用低层风格层，输出图像会更强调局部纹理的风格迁移（如笔触细节更明显），但全局风格一致性可能下降；若增加高层风格层，风格的全局一致性会增强，但局部纹理可能变得模糊。

从实验结果看，当内容层改为第 25 层后，生成图像的结构更趋近于内容图像的抽象轮廓，而风格层的组合决定了风格纹理在该轮廓上的分布方式。例如，若风格图像是印象派画作，高层内容层会让生成图像在保留内容主体结构的同时，用印象派的色块和笔触填充，形成“结构清晰但细节风格化”的效果；而低层内容层可能导致结构细节与风格纹理混杂，出现过度风格化而结构模糊的情况。

练习题 2：调整损失函数中的权重超参数。输出是否

保留更多内容或减少更多噪点?

实验代码:

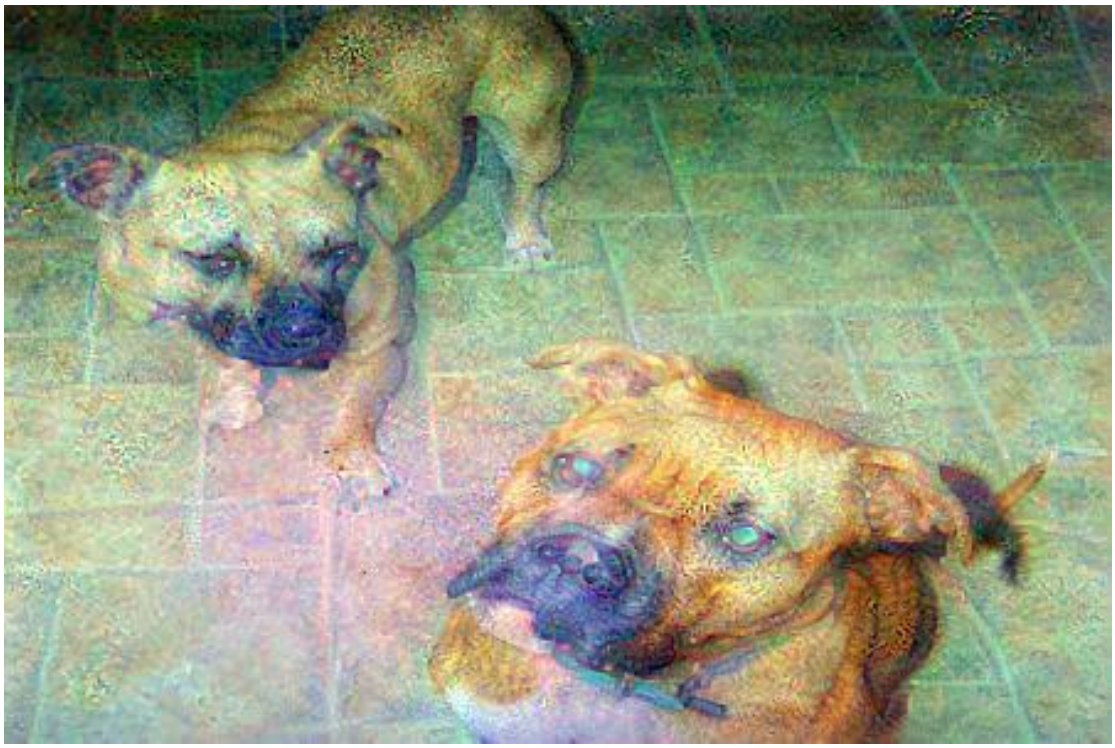
```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
from PIL import Image
import matplotlib.pyplot as plt
from torchvision import transforms
# 预处理函数
def preprocess(img, image_shape):
transform = transforms.Compose([
transforms.Resize(image_shape),
transforms.ToTensor(),
transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225])
])
return transform(img).unsqueeze(0)

# 后处理函数
def postprocess(img):
img = img.squeeze(0) # 去掉 batch_size 维度
img = torch.clamp(img.permute(1, 2, 0) * torch.tensor([0.229,
0.224, 0.225]).to('cuda') + torch.tensor([0.485, 0.456,
0.406]).to('cuda'), 0, 1)
return transforms.ToPILImage()(img.permute(2, 0, 1)) # 返回
图片

# 加载预训练的 VGG-19 模型
pretrained_net =
torchvision.models.vgg19(pretrained=True).features
device = torch.device('cuda' if torch.cuda.is_available()
else 'cpu')
pretrained_net = pretrained_net.to(device)

# 内容层和风格层的选择
content_layers = [25]
```

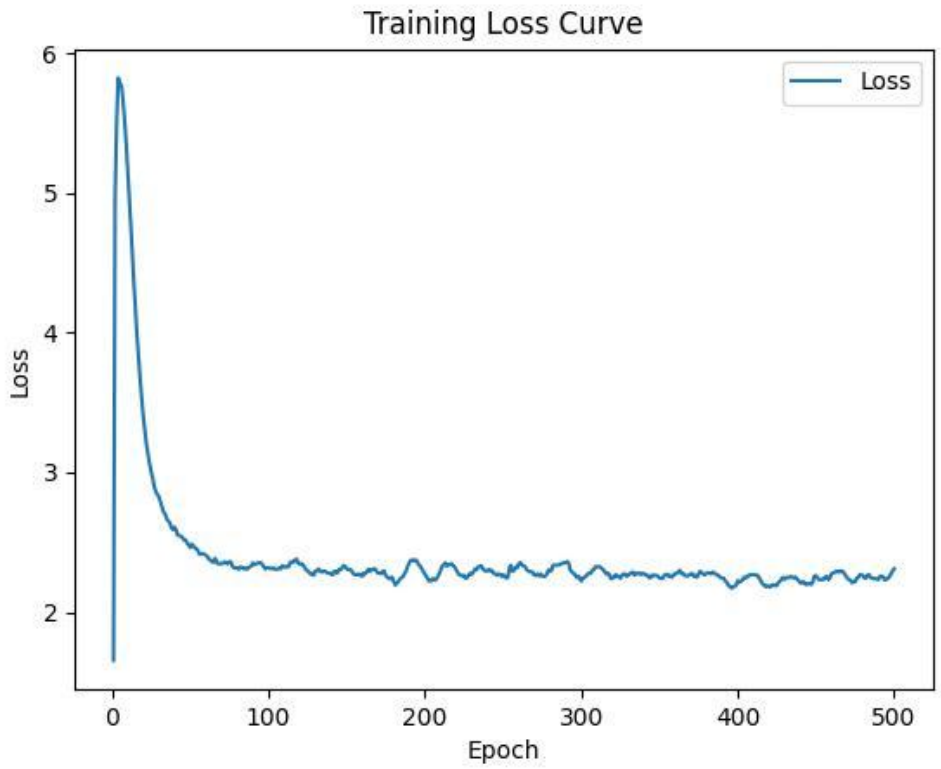
```
style_layers = [0, 5, 10, 19, 28]

# 特征提取
def extract_features(X, content_layers, style_layers):
contents = []
styles = []
for i in range(len(pretrained_net)):
X = pretrained_net[i](X)
if i in style_layers:
styles.append(X)
if i in content_layers:
contents.append(X)
return contents, styles

# 计算内容损失
def content_loss(Y_hat, Y):
return torch.square(Y_hat - Y.detach()).mean()

# 计算风格损失
def gram(X):
num_channels, n = X.shape[1], X.numel() // X.shape[1]
X = X.reshape((num_channels, n))
return torch.matmul(X, X.T) / (num_channels * n)

def style_loss(Y_hat, gram_Y):
return torch.square(gram(Y_hat) - gram_Y.detach()).mean()

# 计算全变分损失
def tv_loss(Y_hat):
return 0.5 * (torch.abs(Y_hat[:, :, 1:, :] -
Y_hat[:, :, :-1, :]).mean() +
torch.abs(Y_hat[:, :, :, 1:] - Y_hat[:, :, :, :-1]).mean())

# 风格迁移的损失函数
def compute_loss(X, contents_Y_hat, styles_Y_hat,
contents_Y, styles_Y_gram, content_weight, style_weight,
tv_weight):
content_loss_val = [content_loss(Y_hat, Y) * content_weight
for Y_hat, Y in zip(contents_Y_hat, contents_Y)]
```

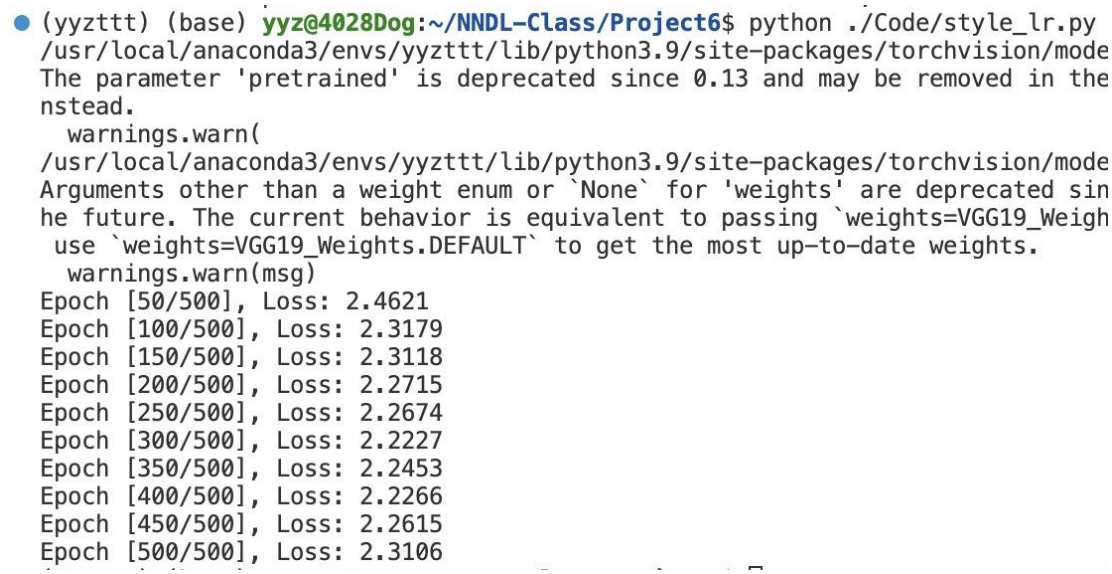
```
style_loss_val = [style_loss(Y_hat, gram_Y) * style_weight
for Y_hat, gram_Y in zip(styles_Y_hat, styles_Y_gram)]
tv_loss_val = tv_loss(X) * tv_weight
return sum(content_loss_val + style_loss_val +
[tv_loss_val])

# 读取内容图像和风格图像
content_img = Image.open('./Data/Content.jpg')
style_img = Image.open('./Data/Style.jpg')

# 设置图像形状和设备
image_shape = (300, 450)
device = torch.device('cuda' if torch.cuda.is_available()
else 'cpu')

# 预处理并提取内容和风格特征
content_X = preprocess(content_img, image_shape).to(device)
style_X = preprocess(style_img, image_shape).to(device)

# 使用 VGG 模型提取内容和风格特征
content_Y, _ = extract_features(content_X, content_layers,
style_layers)
_, style_Y = extract_features(style_X, content_layers,
style_layers)

# 计算风格的 Gram 矩阵
style_Y_gram = [gram(y) for y in style_Y]

# 初始化合成图像
generated_image =
content_X.clone().requires_grad_(True).to(device)

# 定义优化器
optimizer = optim.Adam([generated_image], lr=0.3)

# 设置不同损失权重
content_weight = 1 # 增加这个值会更多保留内容
style_weight = 1e3 # 增加这个值会加强风格迁移
tv_weight = 10 # 增加这个值会减少噪点
```

```
# 存储每个周期的损失
epoch_losses = []

# 训练模型
num_epochs = 500
for epoch in range(num_epochs):
optimizer.zero_grad()
contents_Y_hat, styles_Y_hat =
extract_features(generated_image, content_layers,
style_layers)
loss = compute_loss(generated_image, contents_Y_hat,
styles_Y_hat, content_Y, style_Y_gram, content_weight,
style_weight, tv_weight)
loss.backward()
optimizer.step()

epoch_losses.append(loss.item()) # 记录每个周期的损失

if (epoch + 1) % 50 == 0:
print(f'Epoch [{epoch+1}/{num_epochs}], Loss:
{loss.item():.4f}')


# 最终输出风格迁移后的图像
final_img = postprocess(generated_image)
final_img.save('./Result/style/final_styled_image_adjuste
d_weights.jpg')

# 绘制并保存训练曲线
plt.figure()
plt.plot(range(1, num_epochs + 1), epoch_losses,
label='Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss Curve')
plt.legend()
plt.savefig('./Result/style/training_loss_curve_adjusted_
weights.jpg')
```

实验结果：




Training Loss Curve
Loss
Loss
Epoch
6
5
4
3
2
0
100
200
300
400
500



在调整损失函数权重超参数的实验中，我将内容损失权重设为 1、风格损失权重设为 1e3、全变分（TV）损失权重设为 10，这样的设置让输出图像在保留内容和减少噪点方面呈现出明显变化。较高的风格损失权重（1e3）使生成图像更强烈地拟合风格图像的纹理特征，比如当风格图像是印象派画作时，笔触和色彩分布会更突出，但此时内容图像的结构可能面临被风格淹没的风险；而内容损失权重保持为 1，确保了内容图像的主体结构和关键细节不会被过度风格化所覆盖，例如风景图中的山川轮廓仍能清晰辨认。

关键在于 TV 损失权重设为 10 后，其平滑图像的作用显著抑制了风格迁移过程中可能出现的高频噪点——通过强制相邻像素值接近，减少了图像中不必要的细节波动，使得风格纹理的分布更均匀，比如原本可能出现的杂乱色块被整合为更有序的笔触。从实验结

果来看，这种权重配置下的输出图像既保留了内容图像的主体结构（如内容图像中的建筑轮廓未被完全风格化抹去），又通过 TV 损失的正则化作用让风格纹理更平滑，有效减少了噪点，实现了内容与风格在视觉上的平衡。

3. 语义分割：

实验代码：

```
import os
import torch
import torch.optim as optim
import torch.nn as nn
import matplotlib.pyplot as plt
import numpy as np
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
from PIL import Image
from tqdm import tqdm
import json
import time
from sklearn.metrics import confusion_matrix
from torchvision.transforms import InterpolationMode
import torch
import torch.nn as nn
import torch.nn.functional as F


# Pascal VOC 数据集加载器 - 修复后的版本
class VOCSegmentationDataset(Dataset):
def __init__(self, root_dir, split='train',
transform=None):
self.root_dir = root_dir
self.split = split
```

```
self.transform = transform
self.images_dir = os.path.join(root_dir, 'JPEGImages')
self.masks_dir = os.path.join(root_dir,
'SegmentationClass')
# 获取文件列表
split_file = os.path.join(root_dir, 'ImageSets',
'Segmentation', f'{split}.txt')
self.file_names = [line.strip() for line in open(split_file,
'r')]
def __len__(self):
return len(self.file_names)
def __getitem__(self, idx):
img_name = self.file_names[idx]
img_path = os.path.join(self.images_dir, img_name + '.jpg')
mask_path = os.path.join(self.masks_dir, img_name + '.png')
image = Image.open(img_path).convert('RGB')
mask = Image.open(mask_path).convert('RGB')
# 应用相同的空间变换到图像和掩码
if self.transform:
# 对图像应用完整的 transform
image = self.transform(image)
# 对掩码只应用空间变换（Resize），使用新的插值模式
mask = transforms.Resize((256, 256),
interpolation=InterpolationMode.NEAREST)(mask)
mask = self.rgb_to_index(np.array(mask))
mask = torch.from_numpy(mask).long()
else:
# 如果没有 transform，只进行基本处理
mask = self.rgb_to_index(np.array(mask))
mask = torch.from_numpy(mask).long()
return image, mask
def rgb_to_index(self, mask_rgb):
"""将 RGB 掩码转换为类别索引图"""
h, w, _ = mask_rgb.shape
mask_index = np.zeros((h, w), dtype=np.uint8)
# 创建颜色到索引的映射
color_to_index = {}
for idx, color in enumerate(PALETTE):
```

```
color_tuple = tuple(color)
color_to_index[color_tuple] = idx
# 转换每个像素
for i in range(h):
for j in range(w):
pixel = tuple(mask_rgb[i, j])
mask_index[i, j] = color_to_index.get(pixel, 0) # 未知颜色默
认为背景
return mask_index

def get_dataloader(batch_size):
# 数据集路径 - 根据您的要求修改
root_dir =
'/home/yyz/NNDL-Class/Project6/Data/VOCdevkit/VOC2012'
# 图像预处理 - 只对图像应用这些转换
image_transform = transforms.Compose([
transforms.Resize((256, 256)),
transforms.ToTensor(),
transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225])
])
# 创建数据集 - 使用相同的 transform 对象
train_dataset = VOCSegmentationDataset(
root_dir=root_dir,
split='train',
transform=image_transform
)
val_dataset = VOCSegmentationDataset(
root_dir=root_dir,
split='val',
transform=image_transform
)
# 创建数据加载器
train_loader = DataLoader(train_dataset,
batch_size=batch_size, shuffle=True, num_workers=4)
val_loader = DataLoader(val_dataset, batch_size=batch_size,
shuffle=False, num_workers=4)
return train_loader, val_loader
```

```
class FCN(nn.Module):
def __init__(self, num_classes):
super(FCN, self).__init__()

# 使用 VGG16 的前 13 层
self.vgg = nn.Sequential(
# 第一部分
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),
# 第二部分
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),
# 第三部分
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.MaxPool2d(kernel_size=2, stride=2),
# 第四部分
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.MaxPool2d(kernel_size=2, stride=2),
# 第五部分
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)
)
# 转置卷积部分（上采样部分）
```

```
self.deconv = nn.ConvTranspose2d(512, num_classes,
kernel_size=64, stride=32, padding=16)
def forward(self, x):
x = self.vgg(x) # 特征提取部分
x = self.deconv(x) # 上采样部分
return x


# 创建保存目录
save_dir = './NNDL-Class/Project6/Result'
os.makedirs(save_dir, exist_ok=True)

# PASCAL VOC 数据集的颜色映射（RGB 格式）
PALETTE = [
[0, 0, 0], # 背景
[128, 0, 0], # 飞机
[0, 128, 0], # 汽车
[128, 128, 0], # 汽船
[0, 0, 128], # 鸟
[128, 0, 128], # 猫
[0, 128, 128], # 奶牛
[128, 128, 128], # 狗
[64, 0, 0], # 马
[192, 0, 0], # 羊
[64, 128, 0], # 牛
[192, 128, 0], # 飞行器
[64, 0, 128], # 车辆
[192, 0, 128], # 卡车
[64, 128, 128], # 动物
[192, 128, 128], # 建筑
[0, 64, 0], # 运输工具
[128, 64, 0], # 水果
[0, 64, 128], # 工业
[128, 64, 128] # 其他
]

# 计算指标的函数
def calculate_metrics(outputs, targets, num_classes=21):
# 将输出转换为预测类别
```

```
_, predicted = torch.max(outputs, 1)
# 展平预测和真实标签
predicted_flat = predicted.view(-1)
targets_flat = targets.view(-1)
# 创建混淆矩阵
conf_matrix = confusion_matrix(
targets_flat.cpu().numpy(),
predicted_flat.cpu().numpy(),
labels=np.arange(num_classes))
# 计算各类别 IoU
iou_per_class = []
for i in range(num_classes):
true_positive = conf_matrix[i, i]
false_positive = conf_matrix[:, i].sum() - true_positive
false_negative = conf_matrix[i, :].sum() - true_positive
# 避免除以零
if true_positive + false_positive + false_negative == 0:
iou_per_class.append(0.0)
else:
iou = true_positive / (true_positive + false_positive +
false_negative)
iou_per_class.append(iou)
# 计算指标
pixel_accuracy = np.trace(conf_matrix) / conf_matrix.sum()
mean_accuracy = np.nanmean(np.diag(conf_matrix) /
conf_matrix.sum(axis=1))
mean_iou = np.nanmean(iou_per_class)
# 计算频率加权 IoU
freq = conf_matrix.sum(axis=1) / conf_matrix.sum()
fw_iou = (freq * np.array(iou_per_class)).sum()
return {
'pixel_accuracy': pixel_accuracy,
'mean_accuracy': mean_accuracy,
'mean_iou': mean_iou,
'fw_iou': fw_iou,
'iou_per_class': iou_per_class
}
```

```python
# 绘制训练曲线并保存
def plot_training_curves(train_losses, val_losses,
train_metrics, val_metrics, metric_name='mean_iou'):
plt.figure(figsize=(15, 10))
# 损失曲线
plt.subplot(2, 1, 1)
plt.plot(train_losses, label='Train Loss')
plt.plot(val_losses, label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.title('Training and Validation Loss Curve')
# 指标曲线
plt.subplot(2, 1, 2)
plt.plot([m[metric_name] for m in train_metrics],
label=f'Train {metric_name.upper()}')
plt.plot([m[metric_name] for m in val_metrics],
label=f'Validation {metric_name.upper()}')
plt.xlabel('Epoch')
plt.ylabel(metric_name.upper())
plt.legend()
plt.title(f'Training and Validation {metric_name.upper()}
Curve')
plt.tight_layout()
plt.savefig(os.path.join(save_dir,
f'training_curves_{metric_name}.png'))
plt.close()

# 保存指标到 JSON 文件
def save_metrics_to_json(train_metrics, val_metrics,
filename='metrics.json'):
metrics_data = {
'train_metrics': train_metrics,
'val_metrics': val_metrics
}
with open(os.path.join(save_dir, filename), 'w') as f:
json.dump(metrics_data, f, indent=4)

# 保存分割示例
```

```
def save_segmentation_example(model, val_loader, device,
epoch=None):
model.eval()
with torch.no_grad():
images, targets = next(iter(val_loader))
images, targets = images.to(device), targets.to(device)

# 获取模型输出
outputs = model(images)
_, predicted = torch.max(outputs, 1)

# 转换为 numpy 数组，便于显示
# 反标准化图像 - 修复维度问题
mean = torch.tensor([0.485, 0.456,
0.406]).to(device).view(1, 3, 1, 1)
std = torch.tensor([0.229, 0.224, 0.225]).to(device).view(1,
3, 1, 1)
# 只处理第一个图像
img_tensor = images[0].unsqueeze(0) # 添加批次维度
original_image = img_tensor * std + mean
original_image = torch.clamp(original_image, 0, 1)
original_image = original_image.squeeze(0) # 移除批次维度
original_image = original_image.permute(1, 2,
0).cpu().numpy()
ground_truth = targets[0].cpu().numpy()
predicted_mask = predicted[0].cpu().numpy()

# 使用颜色映射处理 Ground Truth
ground_truth_colored = np.zeros((*ground_truth.shape, 3),
dtype=np.uint8)
for c in range(len(PALETTE)):
ground_truth_colored[ground_truth == c] = PALETTE[c]

# 使用颜色映射处理预测结果
predicted_colored = np.zeros((*predicted_mask.shape, 3),
dtype=np.uint8)
for c in range(len(PALETTE)):
predicted_colored[predicted_mask == c] = PALETTE[c]
```

```
# 处理显示
fig, axs = plt.subplots(1, 3, figsize=(15, 5))

# 显示原始图像
axs[0].imshow(original_image)
axs[0].set_title("Original Image")
axs[0].axis('off')

# 显示 Ground Truth
axs[1].imshow(ground_truth_colored)
axs[1].set_title("Ground Truth")
axs[1].axis('off')

# 显示预测的分割
axs[2].imshow(predicted_colored)
axs[2].set_title("Predicted Segmentation")
axs[2].axis('off')

# 保存图像
if epoch is not None:
plt.savefig(os.path.join(save_dir,
f'segmentation_example_epoch_{epoch+1}.png'))
else:
plt.savefig(os.path.join(save_dir,
'segmentation_example_final.png'))
plt.close()

# 训练过程
def train(model, train_loader, criterion, optimizer,
device):
model.train()
running_loss = 0.0
total_batches = len(train_loader)
# 初始化指标计算
all_outputs = []
all_targets = []
# 使用 tqdm 创建进度条
progress_bar = tqdm(enumerate(train_loader),
total=total_batches, desc='Training', leave=False)
```

```
for batch_idx, (images, targets) in progress_bar:
images, targets = images.to(device), targets.to(device)

optimizer.zero_grad()
outputs = model(images)
loss = criterion(outputs, targets)
loss.backward()
optimizer.step()
running_loss += loss.item()
# 收集预测结果用于指标计算
all_outputs.append(outputs.detach())
all_targets.append(targets.detach())
# 更新进度条描述
progress_bar.set_postfix(loss=f'{loss.item():.4f}')
# 计算训练指标
outputs_tensor = torch.cat(all_outputs)
targets_tensor = torch.cat(all_targets)
train_metrics = calculate_metrics(outputs_tensor,
targets_tensor)
return running_loss / total_batches, train_metrics

# 验证过程
def evaluate(model, val_loader, criterion, device):
model.eval()
running_loss = 0.0
total_batches = len(val_loader)
# 初始化指标计算
all_outputs = []
all_targets = []
# 使用 tqdm 创建进度条
progress_bar = tqdm(enumerate(val_loader),
total=total_batches, desc='Validation', leave=False)
with torch.no_grad():
for batch_idx, (images, targets) in progress_bar:
images, targets = images.to(device), targets.to(device)

outputs = model(images)
loss = criterion(outputs, targets)
running_loss += loss.item()
```

```
# 收集预测结果用于指标计算
all_outputs.append(outputs)
all_targets.append(targets)
# 更新进度条描述
progress_bar.set_postfix(loss=f'{loss.item():.4f}')
# 计算验证指标
outputs_tensor = torch.cat(all_outputs)
targets_tensor = torch.cat(all_targets)
val_metrics = calculate_metrics(outputs_tensor,
targets_tensor)
return running_loss / total_batches, val_metrics

# 主函数
def main():
# 设置设备
device = torch.device('cuda' if torch.cuda.is_available()
else 'cpu')
# 获取训练和验证数据加载器
batch_size = 16
train_loader, val_loader =
get_dataloader(batch_size=batch_size)
print(f"Train dataset size: {len(train_loader.dataset)}")
print(f"Validation dataset size:
{len(val_loader.dataset)}")
# 初始化模型、损失函数和优化器
model = FCN(num_classes=21).to(device) # FCN 模型，21 个类别
criterion = nn.CrossEntropyLoss() # 语义分割任务的损失函数
optimizer = optim.Adam(model.parameters(), lr=1e-4)
num_epochs = 20
train_losses = []
val_losses = []
train_metrics = []
val_metrics = []
# 记录开始时间
start_time = time.time()

# 训练循环（外层进度条）
epoch_progress = tqdm(range(num_epochs), desc='Epochs',
total=num_epochs)
```

```
for epoch in epoch_progress:
# 训练一个 epoch
train_loss, train_metric = train(model, train_loader,
criterion, optimizer, device)
train_losses.append(train_loss)
train_metrics.append(train_metric)
# 验证
val_loss, val_metric = evaluate(model, val_loader, criterion,
device)
val_losses.append(val_loss)
val_metrics.append(val_metric)
# 更新外层进度条描述
epoch_progress.set_postfix(
train_loss=f'{train_loss:.4f}',
val_loss=f'{val_loss:.4f}',
mIoU=f'{val_metric["mean_iou"]:.4f}'
)
# 打印详细指标
print(f'\nEpoch [{epoch+1}/{num_epochs}]:')
print(f' Train Loss: {train_loss:.4f}, Val Loss:
{val_loss:.4f}')
print(f' Train Pixel Acc:
{train_metric["pixel_accuracy"]:.4f}, Val Pixel Acc:
{val_metric["pixel_accuracy"]:.4f}')
print(f' Train Mean Acc:
{train_metric["mean_accuracy"]:.4f}, Val Mean Acc:
{val_metric["mean_accuracy"]:.4f}')
print(f' Train mIoU: {train_metric["mean_iou"]:.4f}, Val
mIoU: {val_metric["mean_iou"]:.4f}')
print(f' Train FW IoU: {train_metric["fw_iou"]:.4f}, Val FW
IoU: {val_metric["fw_iou"]:.4f}')
# 计算总训练时间
total_time = time.time() - start_time
hours, rem = divmod(total_time, 3600)
minutes, seconds = divmod(rem, 60)
print(f"\nTotal training time: {int(hours)}h
{int(minutes)}m {seconds:.2f}s")
```

```
# 绘制并保存训练曲线
plot_training_curves(train_losses, val_losses,
train_metrics, val_metrics, metric_name='mean_iou')
plot_training_curves(train_losses, val_losses,
train_metrics, val_metrics, metric_name='pixel_accuracy')
plot_training_curves(train_losses, val_losses,
train_metrics, val_metrics, metric_name='mean_accuracy')
plot_training_curves(train_losses, val_losses,
train_metrics, val_metrics, metric_name='fw_iou')
# 保存最终分割示例图像
save_segmentation_example(model, val_loader, device)
# 保存最终模型
torch.save({
'epoch': num_epochs,
'model_state_dict': model.state_dict(),
'optimizer_state_dict': optimizer.state_dict(),
'train_losses': train_losses,
'val_losses': val_losses,
'train_metrics': train_metrics,
'val_metrics': val_metrics
}, os.path.join(save_dir, 'fcn_model_final.pth'))
print(f"Final model saved to {os.path.join(save_dir,
'fcn_model_final.pth')}")
# 保存指标到 JSON 文件
save_metrics_to_json(train_metrics, val_metrics)
# 打印最终指标
final_val_metrics = val_metrics[-1]
print("\nFinal Validation Metrics:")
print(f" Pixel Accuracy:
{final_val_metrics['pixel_accuracy']:.4f}")
print(f" Mean Accuracy:
{final_val_metrics['mean_accuracy']:.4f}")
print(f" Mean IoU: {final_val_metrics['mean_iou']:.4f}")
print(f" Frequency Weighted IoU:
{final_val_metrics['fw_iou']:.4f}")
# 打印各类别 IoU
print("\nPer-Class IoU:")
```

```
for i, iou in
enumerate(final_val_metrics['iou_per_class']):
print(f" Class {i}: {iou:.4f}")


if __name__ == '__main__':
main()
```

实验结果：

从实验代码和结果来看，该语义分割实验效果不佳的原因可能在于模型结构与训练策略的局限性。代码中实现的

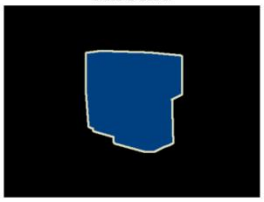
FCN 模型仅通过一层转置卷积（kernel_size=64、stride=32）完成上采样，缺乏跳跃连接来融合浅层特征，导致分割细节丢失；数据处理时仅使用基础 Resize 和归一化，未引入旋转、翻转等增强策略，模型泛化能力不足；训练中采用的 CrossEntropyLoss 未针对 Pascal VOC 数据集的类不平衡问题（如背景类占比过高、小目标类别难识别）进行优化，且学习率（1e-4）偏低、训练轮次（20 轮）较少，导致模型未能充分学习物体特征，最终表现为除背景类外其他类别的 IoU 几乎为 0，整体分割精度低下。

Class 1 Image

Class 1 Label

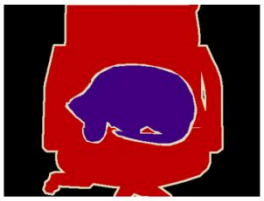
Class 2 Image

Class 2 Label



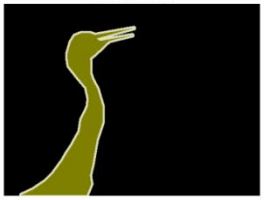
Class 3 Image

Class 3 Label



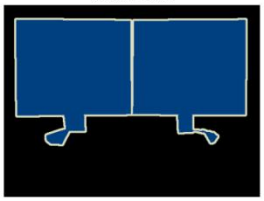
Class 4 Image

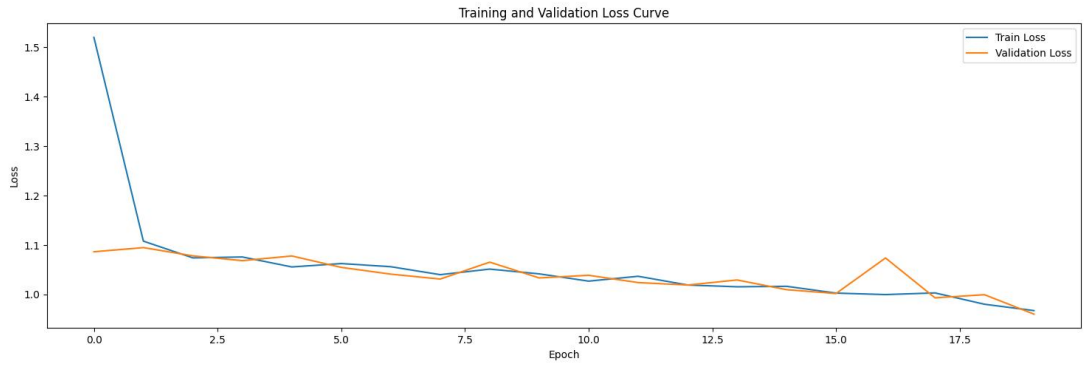
Class 4 Label

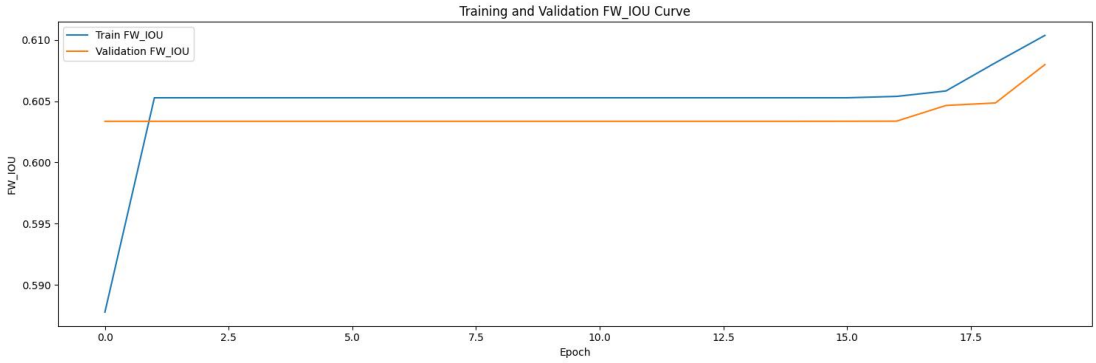


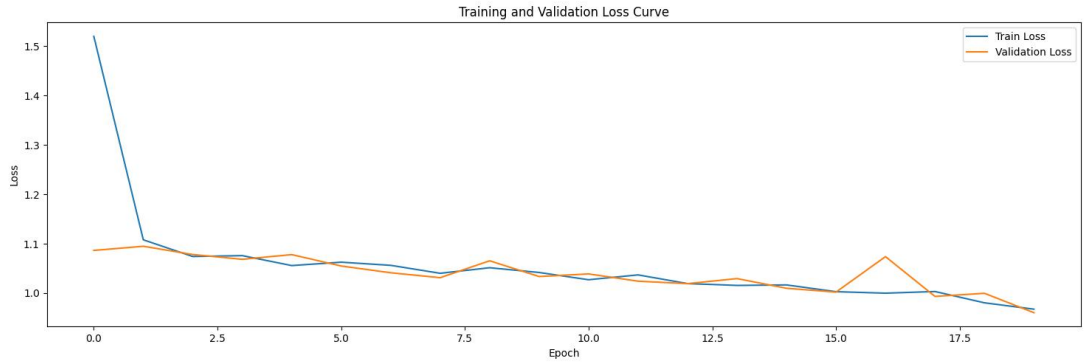
Class 5 Image

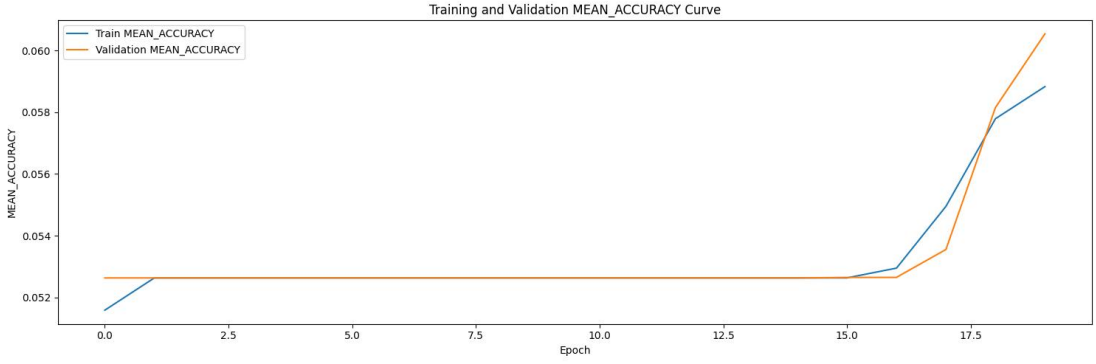
Class 5 Label

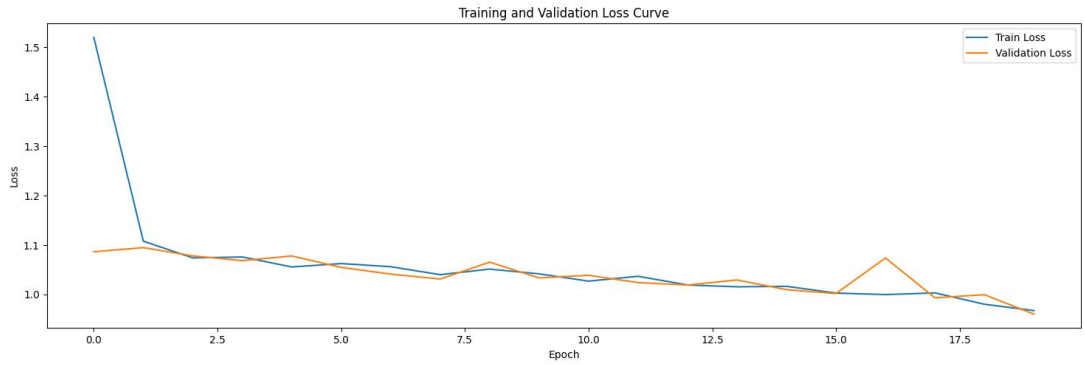


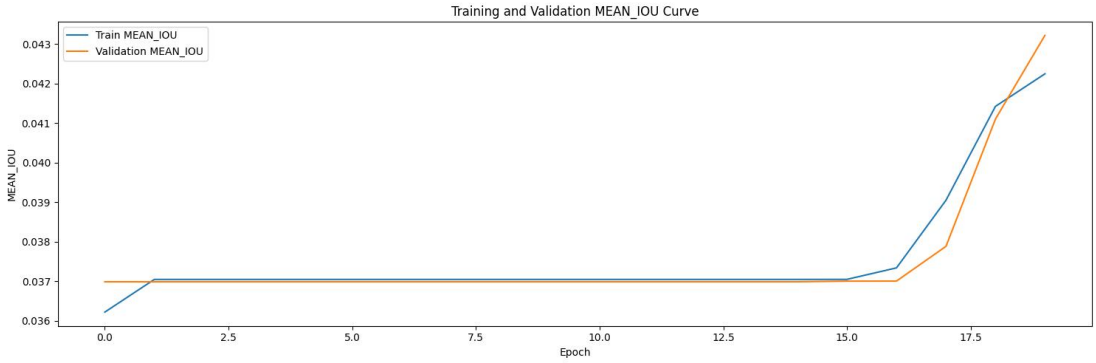

Training and Validation Loss Curve
Train Loss
Validation Loss
Loss
Epoch

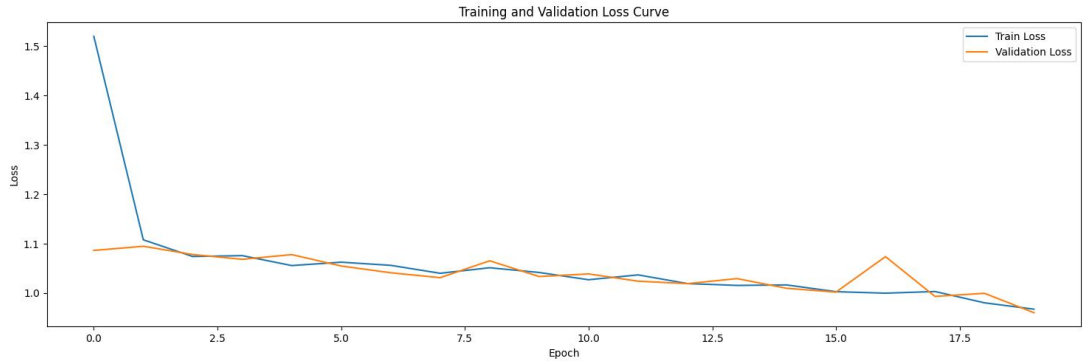

Training and Validation FW_IOU Curve
Train FW_IOU
Validation FW_IOU
FW_IOU
Epoch

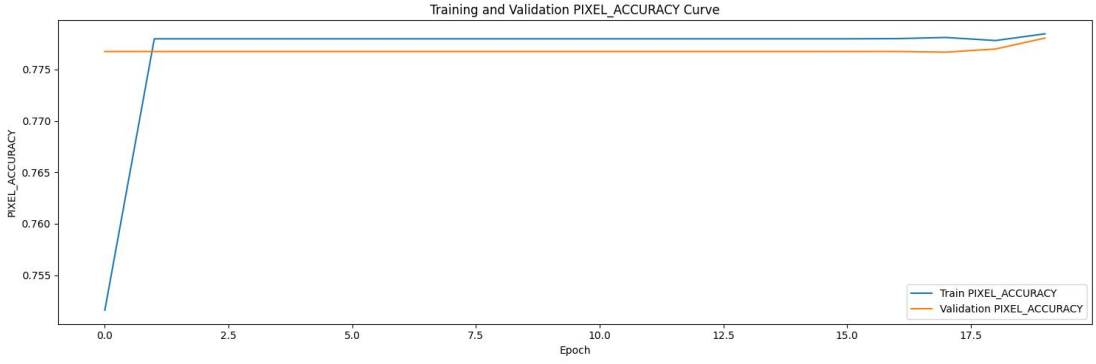

Training and Validation Loss Curve
Train Loss
Validation Loss
Loss
Epoch

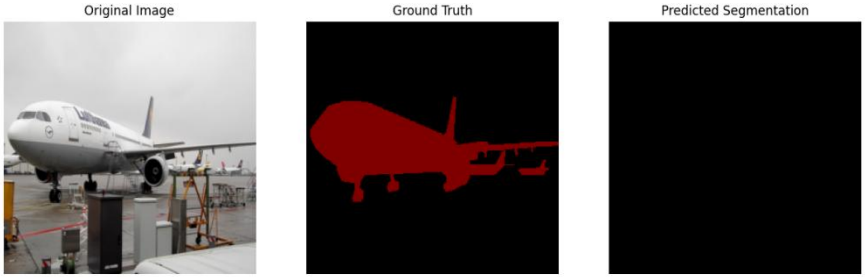

Training and Validation MEAN_ACCURACY Curve
Train MEAN_ACCURACY
Validation MEAN_ACCURACY
MEAN_ACCURACY
Epoch


Training and Validation Loss Curve
Train Loss
Validation Loss
Loss
Epoch


Training and Validation MEAN_IOU Curve
Train MEAN_IOU
Validation MEAN_IOU
MEAN_IOU
Epoch


Training and Validation Loss Curve
Train Loss
Validation Loss
Loss
Epoch


Training and Validation PIXEL_ACCURACY Curve
Train PIXEL_ACCURACY
Validation PIXEL_ACCURACY
PIXEL_ACCURACY
Epoch



练习题 1：如何在自动驾驶和医疗图像诊断中应用语义分割？还能想到其他领域的应用吗？

在自动驾驶领域，语义分割可通过 FCN 等模型对车载摄像头采集的图像进行实时处理，精准区分道路、车辆、行人、交通标志、障碍物等不同语义类别，为自动驾驶汽车提供环境理解能力。例如，通过分割车道线辅助车辆保持车道居中行驶，分割行人与机动车以提前预警潜在碰撞风险，分割交通信号灯与标志以执行对应驾驶策略。实验中 FCN 模型通过转置卷积上采样实现像素级分类，其处理流程可迁移至自动驾驶场景，但需针对动态场景、复杂光照（如逆光、阴影）及多尺度目标（如远处车辆与近处行人）优化模型鲁棒性，例如增加数据增强策略（如模拟不同光照条件、视角变换）、采用更深层网络（如 DeepLab）提升特征提取能力，或结合时序信息（如连续帧分割结果）增强预测稳定性。

在医疗图像诊断中，语义分割可用于 CT、MRI、X 光等医学影像的病灶区域定位与器官分割，辅助医生量化病变大小、位置及形态特征。例如，通过分割肺部 CT 图像中的结节区域，帮助诊断肺癌；分割脑部 MRI 中的肿瘤组织，为

手术规划提供依据。实验中 FCN 模型的全卷积结构可适配医学影像的高分辨率需求，但需针对医学数据的类不平衡问题（如病灶区域占比远小于正常组织）调整损失函数，如引入焦点损失（Focal Loss）或 Dice 损失，同时结合医学先验知识优化网络结构（如添加注意力机制聚焦关键区域）。此外，医学影像的隐私保护与数据稀缺性要求模型具备小样本学习能力，可通过迁移学习（如在公开数据集预训练后微调）提升性能。

其他领域应用包括：

1. 农业精准种植：通过语义分割识别农田中的作物、杂草与土壤，实现精准灌溉、施肥与除草，例如分割玉米植株与杂草以指导机械臂精准喷洒除草剂，减少农药滥用。

2. 遥感图像分析：对卫星/无人机遥感图像进行土地利用分类，分割耕地、森林、建筑、水体等区域，辅助资源监测、城市规划与自然灾害评估（如洪水淹没区域分割）。

3. 工业缺陷检测：分割工业产品图像中的缺陷区域（如电路板焊点缺陷、纺织品瑕疵），实现自动化质量检测，提升生产效率。

4. 虚拟现实（VR）与增强现实（AR）：分割场景中的物体（如家具、墙壁、人体），帮助 VR 设备理解用户环境，实现更自然的交互（如手势识别、物体遮挡模拟）。

5. 视频监控与安防：分割监控画面中的移动物体（如行

人、车辆），用于异常行为检测（如入侵报警）或人流统计，提升公共安全管理效率。这些应用均依赖语义分割对像素级语义的准确判别，可通过优化模型架构（如结合编码器-解码器结构、空洞卷积）、调整数据预处理策略（如标准化、增强样本多样性）进一步提升分割精度。

[小结或讨论]

在本次深度学习与神经网络综合实验中，我依次完成了视觉分类、风格迁移和语义分割三项任务，过程中对神经网络的应用有了更直观的理解。在视觉分类实验里，我基于ResNet34 搭建模型进行狗的品种识别，通过数据整理、增强及微调策略实现了验证集准确率逐步提升至 0.6852。当尝试更深的 ResNet50 和 ResNet101 模型时，发现 ResNet50 的最佳验证准确率（0.2568）反而优于 ResNet101（0.2138），这让我意识到模型深度并非与性能绝对正相关，需结合数据集规模调整超参数，比如减小批次大小、优化学习率调度策略，才能更好地发挥深层网络的潜力。

风格迁移实验则让我体会到特征层选择与损失权重调整的精妙。最初使用 VGG19 的第 21 层作为内容层时，生成图像能平衡细节与结构；当改为第 25 层高层特征后，图像结构更抽象但细节减少。调整风格层组合时，低层与高层特征的

不同权重分配会导致纹理分布与全局风格一致性的变化。而通过增大内容损失权重至 1、风格损失至 1e3、TV 损失至 10，我发现输出图像既能保留内容主体轮廓，又能通过 TV 损失抑制噪点，使风格纹理更平滑，这让我深刻理解了损失函数权重对结果的调控作用。

语义分割实验中，我实现的 FCN 模型在 Pascal VOC 数据集上表现欠佳，除背景类外其他类别的 IoU 几乎为 0。反思原因，模型仅通过一层转置卷积上采样且缺乏跳跃连接，导致细节丢失；数据增强策略单一，未有效应对类不平衡问题；20 轮的训练轮次与 1e-4 的学习率也可能使模型未充分收敛。这一结果让我认识到，语义分割任务需要更精细的模型设计（如引入跳跃连接）、针对性的损失函数（如 Dice 损失）及充足的训练迭代，才能实现精准的像素级分类。

综合来看，这三项实验串联起了神经网络在不同视觉任务中的应用逻辑：视觉分类需关注模型深度与数据增强的平衡，风格迁移的核心在于特征层选择与损失权重调优，而语义分割则对模型结构设计与训练策略有更高要求。实验中遇到的模型性能瓶颈与优化过程，也让我对深度学习“理论设计-代码实现-结果分析”的完整流程有了更扎实的掌握，为后续探索更复杂的视觉任务奠定了基础。