实验二 卷积神经网络

袁雨 PB20151804

一、实验要求

使用 pytorch 或者 tensorflow 实现卷积神经网络，在 ImageNet 数据集上进行图片分类。研究 dropout, normalization, learning rate decay, residual connection, network depth等超参数对分类性能的影响。

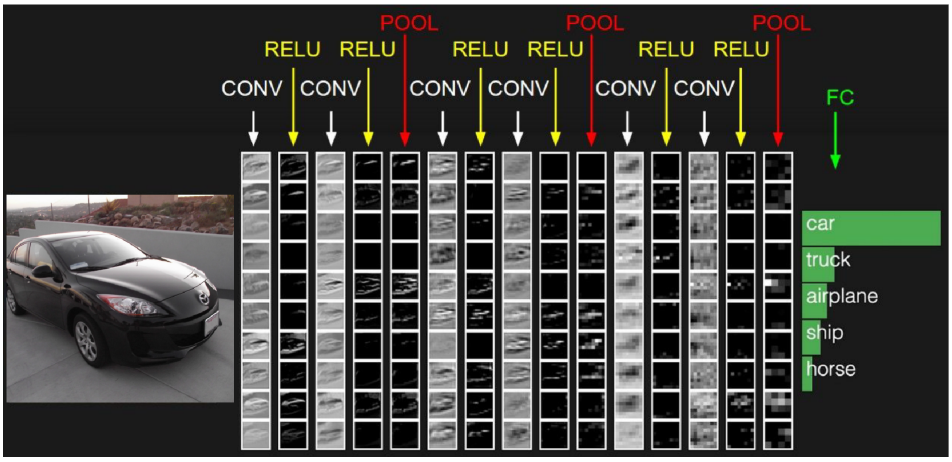
二、实验步骤

1. **网络框架**：要求选择 pytorch 或 tensorflow 其中之一，依据官方网站的指引安装包。（如果前面实验已经安装过，则这个可以跳过）
2. **数据集**：这次实验使用 Tiny-Imagenet-200 数据集，包含 200 个类，每个类有 500 张训练图像，50 张验证图像和 50 张测试图像。由于测试图像没有标签，因此使用数据集中的验证集当作测试集，并从训练集中手动划分新的训练集和测试集。
3. **模型搭建**：采用 pytorch 或 tensorflow 所封装的 module 编写模型，例如 torch.nn.Linear(),torch.nn.Relu() 等，无需手动完成底层 forward、backward 过程。
4. **模型训练**：将生成的训练集输入搭建好的模型进行前向的 loss 计算和反向的梯度传播，从而训练模型，同时也建议使用网络框架封装的 optimizer 完成参数更新过程。训练过程中记录模型在训练集和验证集上的损失，并绘图可视化。
5. **调参分析**：将训练好的模型在验证集上进行测试，以 **Top 1 Accuracy(ACC)** 作为网络性能指标。然后，对 dropout, normalization, learning rate decay, residual connection, network depth 进行调整，再重新训练、测试，并分析对模型性能的影响。
6. **测试性能**：选择你认为最合适的（例如，在验证集上表现最好的）一组超参数，重新训练模型，并在测试集上测试（注意，这理应是你的实验中**唯一**一次在测试集上的测试），并记录测试的结果（ACC）。

三、实验原理

1. 整体架构

卷积神经网络是用卷积层代替全连接层。整体架构为：输入层--卷积层--池化层--全连接层。



2. 动机

- 参数共享

  一个特征检测器对图片所有区域都适用。

- 稀疏连接

  根据常理，一个特征往往是分布于一个区域的，所以由卷积产生的结果中的一个小格子，只和原有图片中滤波器算子大小的格子相关联，而不是全连接，这样就将无用的连接抹除了。这种稀疏连接还带来了平移不变属性，即使移动一些像素，只要整体特征没有发生明显变化，则判断不会改变。

3. 残差网络

深层网络难以训练的原因之一就是梯度爆炸和梯度消失。残差网络（Residual Network，ResNet）可以有效解决这种问题。残差网络是通过给非线性的卷积层增加直连边的方式来提高信息的传播效率。

四、实验过程

1.数据预处理

2.数据增强

```
# 数据增强
class Data_Augmentation(Dataset):

    def __init__(self, data):
        self.data = data
        self.metd =6+1 #水平翻转 垂直翻转 图片裁剪 图片加噪声 图像剪切 对比度
        self.len = data.__len__() * self.metd
        self.tran1 = transforms.Compose([
            # transforms.ToPILImage(),
            transforms.RandomHorizontalFlip(p=0.6),
        ])
        self.tran2 = transforms.Compose([
            transforms.RandomVerticalFlip(p=0.6),
        ])
        self.tran3 = transforms.Compose([
            transforms.RandomCrop((40,40),pad_if_needed=True),
            transforms.Resize((64,64)),
        ])
        self.tran4 = transforms.Compose([
            transforms.GaussianBlur(kernel_size=(3,3), sigma=(0.1,2.0)),
```

```
        ])
        self.tran5 = transforms.Compose([
            transforms.RandomRotation(degrees=(-60,60)),
        ])
        self.tran6 = transforms.Compose([
            transforms.ColorJitter(brightness=0.5, hue= 0.3),
        ])
    def __getitem__(self, index):
        remain = index % self.metd
        label, image = self.data.__getitem__(index//self.metd)
        if remain == 0:
            pass
        elif remain == 1:
            image = self.tran1(image)
        elif remain== 2:
            image = self.tran2(image)
        elif remain == 3:
            image = self.tran3(image)
        elif remain == 4:
            image = self.tran4(image)
        elif remain == 5:
            image = self.tran5(image)
        elif remain== 6:
            image = self.tran6(image)
        return label, image

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

3.定义网络结构

- 单层卷积网络

```
class nor_cov(nn.Module):
    """单层卷积网络
    """
    def __init__(self, in_channel, out_channel, dropout=False, normalize=False):
        super(nor_cov, self).__init__()

        self.cov = nn.Conv2d(in_channel, out_channel, kernel_size=3, stride = 1,
padding = 1)

        self.normalize = normalize
        if normalize:
            self.nor = nn.BatchNorm2d(out_channel)

        self.relu = nn.ReLU()
        self.dropout = dropout
        if dropout:
            self.drop = nn.Dropout(p=0.2)

    def forward(self, x):
        x = self.cov(x)

        if self.normalize:
            x = self.nor(x)
        x = self.relu(x)
```

```
        if self.dropout:
            x = self.drop(x)

        return x
```

- 残差网络

```
class residual_block(nn.Module):
    """残差网络
    """
    def __init__(self, channel, dropout=False):
        super(residual_block, self).__init__()

        self.cov1 = nn.Conv2d(channel, channel, kernel_size = 3, stride = 1,
padding = 1)
        self.cov2 = nn.Conv2d(channel, channel, 3, 1, 1)

        self.nor1 = nn.BatchNorm2d(channel)
        self.nor2 = nn.BatchNorm2d(channel)
        self.relu = nn.ReLU()
        self.dropout = dropout
        if dropout:
            self.drop = nn.Dropout(p=0.2)

    def forward(self, x):
        x_ = self.cov1(x)
        x_ = self.nor1(x_)
        x_ = self.relu(x_)

        x_ = self.cov2(x_)
        x_ = self.nor2(x_)
        x_ = self.relu(x_)

        if self.dropout:
            x_ = self.drop(x_)

        x = x + x_
        return x
```

- 双层卷积网络

```
class dou_cov(nn.Module):
    """双层卷积网络
    """
    def __init__(self, channel, dropout=False, normalize=False):
        super(dou_cov, self).__init__()
        self.cov1 = nor_cov(in_channel=channel, out_channel=channel,
dropout=dropout, normalize=normalize)
        self.cov2 = nor_cov(in_channel=channel, out_channel=channel,
dropout=dropout, normalize=normalize)

    def forward(self, x):
        x = self.cov1(x)
        x = self.cov2(x)
        return x
```

- 自定义卷积网络

```
class CNN_net(nn.Module):
    """自定义卷积网络
    """

    def __init__(self):
        super(CNN_net, self).__init__()

        self.cov1 = nor_cov(in_channel=3, out_channel=64, dropout=True,
normalize=True)
        self.res1 = residual_block(64, dropout=False)
        self.res2 = residual_block(64, dropout=False)
        self.pool1 = nn.MaxPool2d(2, 2)

        self.cov2 = nor_cov(in_channel=64, out_channel=128, dropout=True,
normalize=True)
        self.res3 = residual_block(128, dropout=False)
        self.res4 = residual_block(128, dropout=False)
        self.pool2 = nn.MaxPool2d(2, 2)

        self.cov3 = nor_cov(in_channel=128, out_channel=256, dropout=True,
normalize=True)
        self.res5 = residual_block(256, dropout=False)
        self.res6 = residual_block(256, dropout=False)
        self.pool3 = nn.MaxPool2d(2, 2)

        self.cov4 = nor_cov(in_channel=256, out_channel=512, dropout=True,
normalize=True)
        self.res7 = residual_block(512, dropout=False)
        self.res8 = residual_block(512, dropout=False)
        self.pool4 = nn.MaxPool2d(2, 2)

        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))

        self.fc1 = nn.Linear(512 * 1 * 1, 200)

    def forward(self, x):
        x = self.cov1(x)
        x = self.res1(x)
        x = self.res2(x)
        x = self.pool1(x)

        x = self.cov2(x)
        x = self.res3(x)
        x = self.res4(x)
        x = self.pool2(x)

        x = self.cov3(x)
        x = self.res5(x)
        x = self.res6(x)
        x = self.pool3(x)

        x = self.cov4(x)
        x = self.res7(x)
        x = self.res8(x)
        x = self.pool4(x)
```

```
        x = self.avgpool(x)

        x = x.reshape(x.shape[0], -1)

        x = self.fc1(x)

        return x
```

4.定义模型

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

class Lab2Model(object):

    def __init__(self, batch_size=64, num_workers=10, seed=0):
        self.seed = seed
        self.setup_seed()
         # 这里 ToTensor 会把 numpy 类型转换为 tensor 类型，并对数据归一化到 [0, 1]
        train_dataset = Data(type_="train",
transform=transforms.Compose([transforms.ToTensor()]))
        # 从训练数据中手动划分训练集和验证集
        train_dataset, self.val_dataset = random_split(train_dataset,

[int(len(train_dataset) * 0.8), len(train_dataset) - int(len(train_dataset) *
0.8)],

 generator=torch.Generator().manual_seed(0))
        self.train_dataset = Data_Augmentation(train_dataset)

        self.batch_size = batch_size
        self.num_workers = num_workers
        self.train_dataloader = DataLoader(dataset=self.train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True,
                                           num_workers=num_workers,
                                           drop_last=True)
        self.val_dataloader = DataLoader(dataset=self.val_dataset,
                                         batch_size=batch_size,
                                         shuffle=False,
                                         num_workers=num_workers,
                                         drop_last=True)

        self.net = None
        self.lr = None
        self.optimizer = None
        self.schedule = None
        self.fig_name = None
        self.loss_list = {"train": [], "val": []}

    def train(self, lr=0.01, epochs=10, wait=8, lrd=False, fig_name="lab2"):
        self.lr = lr
        self.fig_name = fig_name
        self.net = CNN_net().to(device)
        self.optimizer = optim.Adam(self.net.parameters(), lr=lr)
        if lrd:
            self.schedule = ReduceLROnPlateau(self.optimizer, 'min', factor =
0.5, patience=3, verbose=True)
```

```
        total_params = sum([param.nelement() for param in self.net.parameters()
if param.requires_grad])

        print(">>> Total params: {}".format(total_params))

        print(">>> Start training")
        min_val_loss = np.inf
        min_val_loss_acc = 0.0
        delay = 0
        for epoch in range(epochs):

            # train train data
            for data in tqdm(self.train_dataloader):
                labels, inputs = data
                inputs = inputs.to(device)
                labels = labels.to(device)
                self.optimizer.zero_grad()
                outputs = self.net(inputs)
                loss = nn.CrossEntropyLoss()(outputs, target=labels)
                loss.backward()
                self.optimizer.step()

            # calc train loss and train acc
            train_loss = 0.0
            train_acc = 0.0
            val_loss = 0.0
            val_acc = 0.0
            with torch.no_grad():
                for data in self.train_dataloader:
                    labels, inputs = data
                    inputs = inputs.to(device)
                    labels = labels.to(device)
                    outputs = self.net(inputs)
                    loss = nn.CrossEntropyLoss()(outputs, target=labels)

                    train_loss += loss.item()
                    train_acc += self.acc(labels=labels.cpu().numpy(),
outputs=outputs.detach().cpu().numpy())
                train_loss = train_loss / len(self.train_dataloader)
                train_acc = train_acc / len(self.train_dataloader)
                self.loss_list['train'].append(train_loss)

                for data in self.val_dataloader:
                    labels, inputs = data
                    inputs = inputs.to(device)
                    labels = labels.to(device)
                    outputs = self.net(inputs)
                    loss = nn.CrossEntropyLoss()(outputs, labels)

                    val_loss += loss.item()
                    val_acc += self.acc(labels=labels.cpu().numpy(),
outputs=outputs.detach().cpu().numpy())
                val_loss = val_loss / len(self.val_dataloader)
                val_acc = val_acc / len(self.val_dataloader)
                self.loss_list['val'].append(val_loss)
                print(f"Epoch {epoch}: train loss {train_loss:10.6f}, "
                      f"val loss {val_loss:10.6f}")
```

```
            # if necessary, reduce the learning rate by val loss
            if lrd:
                self.schedule.step(val_loss)

            if val_loss < min_val_loss:
                min_val_loss = val_loss
                relative_train_loss = train_loss
                min_val_loss_acc = val_acc
                relative_train_loss_acc = train_acc
                print(f"Update min_val_loss to {min_val_loss:10.6f}, relative
train_loss is {relative_train_loss:10.6f}")
                delay = 0
            else:
                delay = delay + 1

            if delay > wait:
                break
        print(">>> Finished training")
        self.plot_loss()
        print(">>> Finished plot loss")
        return relative_train_loss_acc,min_val_loss_acc

    def test(self):
        test_data = Data(type_="val",
transform=transforms.Compose([transforms.ToTensor()]))
        test_data_loader = DataLoader(dataset=test_data,
                                      batch_size=self.batch_size,
                                      shuffle=False,
                                      num_workers=self.num_workers,
                                      drop_last=False)

        test_acc = 0.0
        for data in test_data_loader:
            labels, inputs = data
            inputs = inputs.to(device)
            outputs = self.net(inputs)
            test_acc += self.acc(labels.numpy(), outputs.detach().cpu().numpy())

        test_acc = test_acc / len(test_data_loader)
        return test_acc

    def acc(self, labels, outputs, type_="top1"):
        acc = 0
        if type_ == "top1":
            pre_labels = np.argmax(outputs, axis=1)
            labels = labels.reshape(len(labels))
            acc = np.sum(pre_labels == labels) / len(pre_labels)

        return acc

    def setup_seed(self):
        seed = self.seed
        torch.manual_seed(seed)
        np.random.seed(seed)
        torch.cuda.manual_seed(seed)
        random.seed(seed)

    def plot_loss(self):
```

```
        plt.figure()
        train_loss = self.loss_list['train']
        val_loss = self.loss_list['val']
        plt.plot(train_loss, c="red", label="train_loss")
        plt.plot(val_loss, c="blue", label="val_loss")
        plt.legend()
        plt.xlabel("Epoch")
        plt.ylabel("CrossEntropyLoss")
        plt.title("CrossEntropyLoss of Train and Validation in each Epoch")
        plt.savefig(f"./fig/{self.fig_name}_loss.png")
```

五、实验分析

- **dropout**

dropout(cov,res)	train acc	val acc
(False,False)	0.624563	0.391526
(0.1,0.1)	0.573538	0.388822
(0.2,False)	0.645913	0.406550
(0.2,0.2)	0.592875	0.401893
(0.2,0.5)	0.541362	0.398287
(0.5,0,5)	0.542675	0.400290
(0.9,0.9)	0.116288	0.110276

dropout可以避免某些特定神经元过分依赖某些输入特征而导致过拟合。

实验可知若dropout概率太小则过拟合风险大，若dropout概率太大则会导致欠拟合。选择（0.2，False）较合适。

- **normalization**

normalization	train acc	val acc
True	0.534312	0.379808
False	0.388887	0.283554

normalization使输入数据具有相同的尺度和范围，避免数据之间的差异太大而导致神经网络无法正确学习。

实验可知如果不进行normalization，网络的学习效果很差。

- **learning rate**

lr	factor	train acc	val acc
0.001	0.1	0.534312	0.379808
0.001	0.3	0.550475	0.387570
0.001	0.5	0.645913	0.406550
0.001	0.9	0.541150	0.386619
0.003	0.1	0.535625	0.383864
0.003	0.5	0.533400	0.382913
0.01	0.1	0.425187	0.305990

学习率一开始要保持大些保证收敛速度，在收敛到最优点附近时要小些以避免来回振荡。

当学习率较小时，收敛较慢，适合较大的factor。当学习率较大时，容易发生震荡，适合较小的factor。实验可知 lr = 0.001，factor=0.5 较合适。

- **residual connection**
  - net1：

    cov1(3,64)-res1(64)-res2(64)-maxpool1(2,2)--
    cov2(64,128)-res3(128)-res4(128)-maxpool2(2,2)--
    cov3(128,256)-res5(256)-res6(256)-maxpool3(2,2)--
    cov4(256,512)-res7(512)-res8(512)-maxpool4(2,2)--
    cov5(512,256)-res9(256)-res10(256)-maxpool5(2,2)--
    cov6(256,128)-res11(128)-res12(128)-maxpool6(2,2)--
    fc1(128)
  - net2：

    cov1(3,64)-dou_cov1(64)-dou_cov2(64)-maxpool1(2,2)--
    cov2(64,128)-dou_cov3(128)-dou_cov4(128)-maxpool2(2,2)--
    cov3(128,256)-dou_cov5(256)-dou_cov6(256)-maxpool3(2,2)--
    cov4(256,512)-dou_cov7(512)-dou_cov8(512)-maxpool4(2,2)--
    cov5(512,256)-dou_cov9(256)-dou_cov10(256)-maxpool5(2,2)--
    cov6(256,128)-dou_cov11(128)-dou_cov12(128)-maxpool6(2,2)--
    fc1(128)

net	train acc	val acc
net1	0.645913	0.406550
net2	0.356863	0.253606

深层网络难以训练的原因之一就是梯度爆炸和梯度消失。残差网络可以有效解决这种问题。

实验可知在相同卷积层数下，有残差连接的net1效果较好。

- **network depth**
  - net1：

    cov1(3,64)-res1(64)-res2(64)-maxpool1(2,2)--

    cov2(64,128)-res3(128)-res4(128)-maxpool2(2,2)--

    cov3(128,256)-res5(256)-res6(256)-maxpool3(2,2)--

    cov4(256,512)-res7(512)-res8(512)-maxpool4(2,2)--

    cov5(512,256)-res9(256)-res10(256)-maxpool5(2,2)--

    cov6(256,128)-res11(128)-res12(128)-maxpool6(2,2)--

    fc1(128)
  - net2

    cov1(3,64)-res1(64)-maxpool1(2,2)--

    cov2(64,128)-res2(128)-maxpool2(2,2)--

    cov3(128,256)-res3(256)-maxpool3(2,2)--

    cov4(256,512)-res4(512)-maxpool4(2,2)--

    cov5(512,256)-res5(256)-maxpool5(2,2)--

    cov6(256,128)-res6(128)-maxpool6(2,2)--

    fc1(128)
  - net2：

    cov1(3,64)-res1(64)-res2(64)-maxpool1(2,2)--

    cov2(64,128)-res3(128)-res4(128)-maxpool2(2,2)--

    cov3(128,256)-res5(256)-res6(256)-maxpool3(2,2)--

    cov4(256,512)-res7(512)-res8(512)-maxpool4(2,2)--

    cov5(512,256)-res9(256)-res10(256)-maxpool5(2,2)--

    fc1(256)--

    avgpool(1,1)
  - net3

    cov1(3,64)-res1(64)-res2(64)-maxpool1(2,2)--

    cov2(64,128)-res3(128)-res4(128)-maxpool2(2,2)--

    cov3(128,256)-res5(256)-res6(256)-maxpool3(2,2)--

    cov4(256,512)-res7(512)-res8(512)-maxpool4(2,2)--

    fc1(512)--

    avgpool(1,1)

net	train acc	val acc
net1	0.645913	0.406550
net2	0.568312	0.382762
net3	0.617537	0.432943
net4	0.692712	0.466897

在一般情况下，神经网络的深度越深，其拟合能力就越强，但也会相应地增加模型的复杂度和计算成本。此外，若深度过深，可能会导致过拟合。

实验可知可选择net4对应的网络结构与深度。

- **data augmentation**

data augmentation	train acc	val acc
无	0.692712	0.466897
水平翻转 垂直翻转 图片裁剪	0.641359	0.504056
水平翻转 垂直翻转 图片裁剪 图片加噪声 图像剪切 对比度	0.676895	0.526643

数据增强可以扩充数据集容量、提高模型鲁棒性。

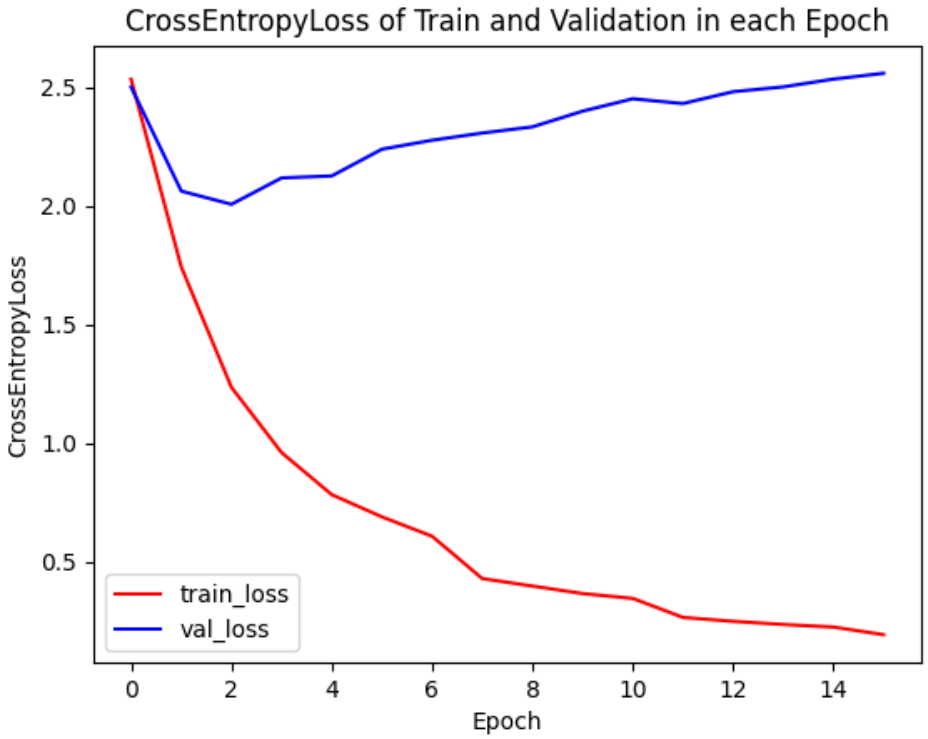
实验可知对训练集进行多种数据转换方式会提升学习效果。

六、实验结果

- 最合适的一组超参数
  - dropout：(0.2,False)
  - lr：0.001
  - factor：0.5
  - residual connection：True
  - normalize：True
  - data augmentation：True

- 模型在训练集和验证集上的损失及可视化图
  - 最低loss

    min_val_loss : 2.006712, relative train_loss : 1.235781
  - 可视化



可见train_loss和val_loss在迭代初期逐渐下降，后期train_loss下降，但val_loss逐渐上升，产生了过拟合。

实验中也尝试过调大dropout等，但过拟合现象并未得到缓解，可能是数据集不够大等缘故。

- 测试结果
  - 最优ACC

    train_acc : 0.676895, val_acc : 0.526643, test_acc : 0.568372