

第三部分：预训练模型

计算机科学技术学院 周训哲 20307110315

2023 年 4 月 16 日

一、预训练模型的基本结构

在本次实验中使用的预训练的模型为 ResNet18，其基本结构就是卷积神经网络，但是使用了残差网络的模式对基本的 CNN 进行了改进。

残差网络

残差网络相比于传统的网络，多了一个直接跳跃连接的部分，直接将输入传给输出部分。每次训练的时候并非传统的将输入通过函数拟合到目标值，而是拟合到目标值与输入的差值。

$$\mathbf{y} = \mathcal{F}(\mathbf{x}, \mathbf{W}) + \mathbf{x} \quad (1)$$

这样通过堆积残差网络的层数，就可以实现想目标值逐步逼近，同时差值越来越小，训练成本也越来越小，训练速度也可以加快。这样还可以通过直接跳跃的功能使得每一次的输出都是有意义的，每一层都可以学习新的残差，使得网络很深也能很好的训练，从而避免传统网络出现的梯度爆炸或梯度消失等问题。

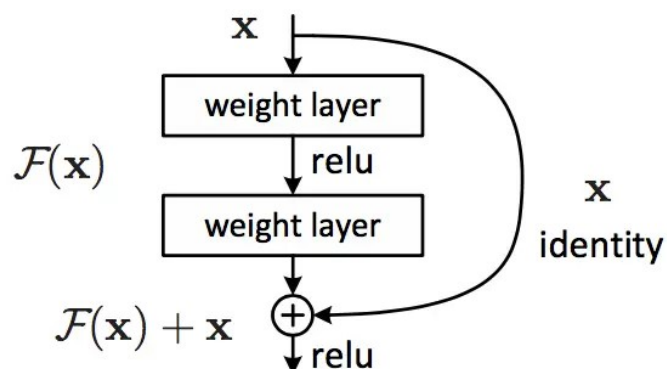


图 1: 残差网络的结构

ResNet

ResNet 是一种利用残差网络结构的网络，其特征提取的方法采用的是卷积神经网络和线性层结合的方式。具体结构为首先第一层为一个卷积层，将输入的数据进行一次特征提取，然后经过一个最大池化层对特征进行压缩，输出经过一个激活函数后作为下一层的输入。

然后就是利用残差网络的思想构造残差块。每一个残差块都包含了两个卷积层和一个跳跃连接，跳跃连接直接将输入传到输出结果，然后与卷积神经网络拟合到残差的结果进行相加得到输出，经过一个激活函数后将输出作为下一层的输入。对于每个残差块，可能需要在跳跃连接过程中通道数需要变化，则可以使用 1×1 卷积改变步长来调整通道数。

经过多个残差块之后，得到一个拟合的结果，经过一个全局平均池化，将特征图最终压缩成一个特征向量，这样做可以极大减少了模型中的参数数量。

在最后则是通过一个线性层将上一步得到的特征向量拟合成最后的输出。

对于 ResNet18 来说，网络具体结构可以简化成如下形式：

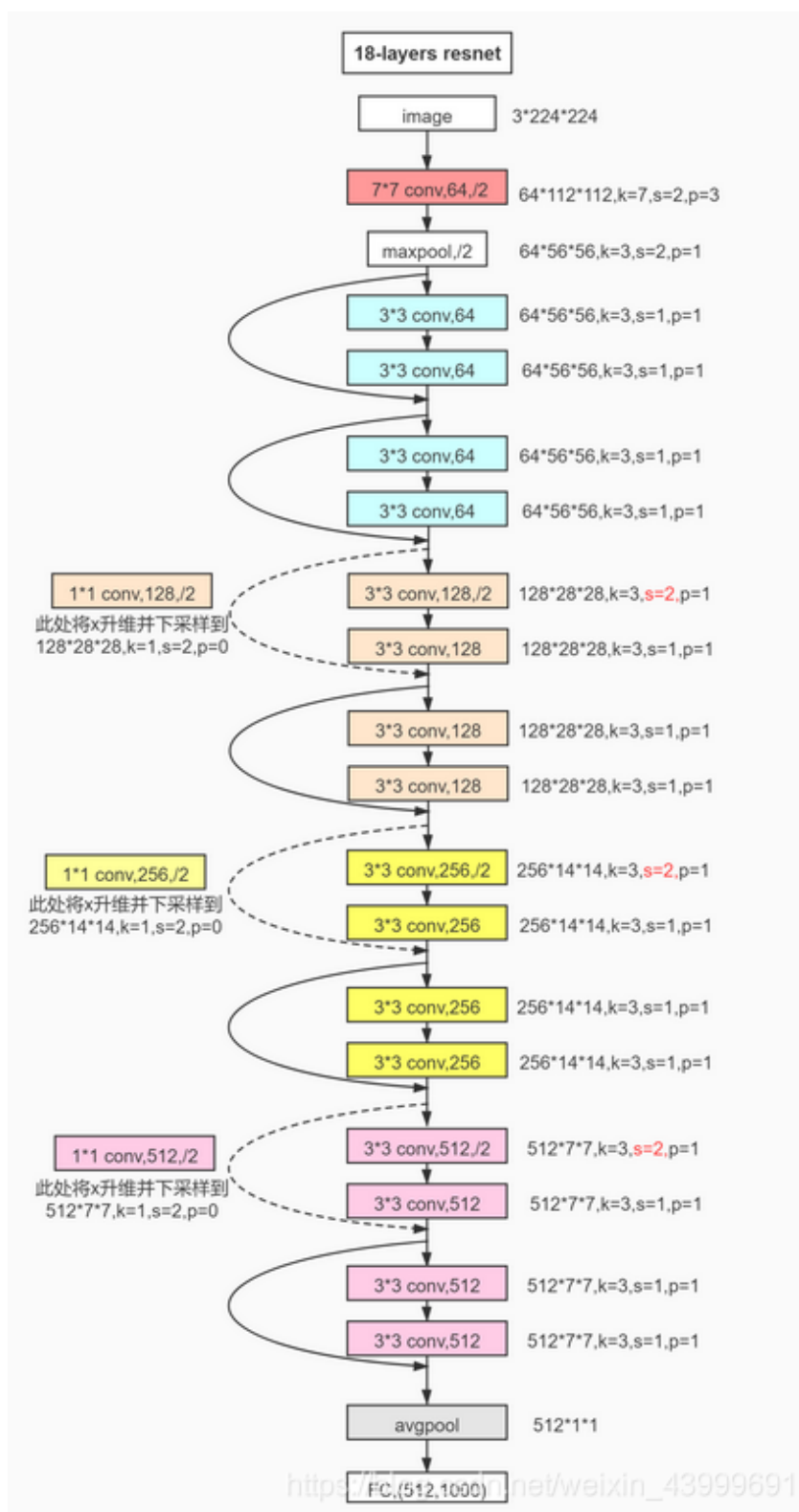


图 2: ResNet18 的结构

预训练模型

本次实验中采用的是 ResNet18 网络，在实验过程中，由于原本的网络输入为 3 个通道数，所以调用模型中的参数获取网络结构并将其修改为 1 个通道数的输入。

同时原本输出为 1000 维，用同样的方法使得输出只有 MNIST 数据集要求的 10 个维度。

其余步长、padding 等参数均保持原始参数，预训练参数使用的是 IMAGENET1K_V1 参数，即使用 ImageNet 做训练集得到的 V1 版本的参数。

二、实验代码及结果


代码结构

代码的基本架构就是首先调用模型，并存储到一个自定义的类中

```
1 class ResNet18(nn.Module):
2     def __init__(self):
3         super(ResNet18, self).__init__()
4         self.resnet18 = models.resnet18(\
5             weights=models.ResNet18_Weights.IMAGENET1K_V1)
6         self.resnet18.conv1 = nn.Conv2d(1, 64, kernel_size=7,
7             stride=2, padding=3, bias=False)
8         self.resnet18.fc = nn.Linear(self.resnet18.fc.in_features, 10)
9
10    def forward(self, x):
11        x = self.resnet18(x)
12        return x
```

在训练模型的代码中，为了加速运算，我还尝试安装 cuda 相关组件进行 GPU 加速。效果惊人，训练一轮的时间平均只用 1 分钟。

```
1 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")  
2 print("Using device:", device)
```

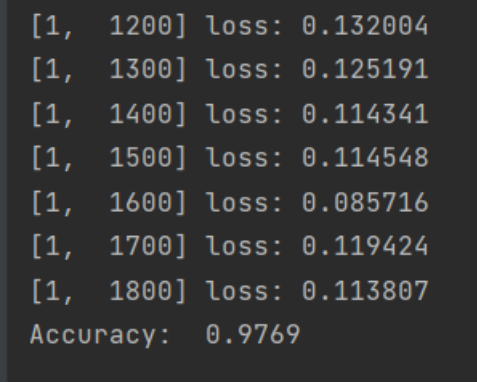
A dark-themed terminal window showing the output of the print statement: "Using device: cuda".

```
Using device: cuda
```

图 3: 使用 cuda 进行计算加速

后续就是使用 torchvision 库下载 MNIST 数据集，并对数据集进行一系列预处理，具体方法使用了标准化、批处理、shuffle 等操作。

训练时使用交叉熵作为损失函数，Adam 作为优化器进行训练。只训练一次就已经可以达到 97.69% 的准确率。

A dark-themed terminal window showing the output of a training loop. It lists loss values for iterations 1200 through 1800, followed by the final accuracy.

```
[1, 1200] loss: 0.132004  
[1, 1300] loss: 0.125191  
[1, 1400] loss: 0.114341  
[1, 1500] loss: 0.114548  
[1, 1600] loss: 0.085716  
[1, 1700] loss: 0.119424  
[1, 1800] loss: 0.113807  
Accuracy: 0.9769
```

图 4: 训练一次的准确率

具体代码实现如下

```
1 # load MNIST
2 transform = transforms.Compose([
3     transforms.ToTensor(),
4     transforms.Normalize((0.1307,), (0.3081,))
5 ])
6 train_set = datasets.MNIST(root='./data', train=True,
7     download=True, transform=transform)
8 test_set = datasets.MNIST(root='./data', train=False,
9     download=True, transform=transform)
10 # 训练集与测试集都是批处理, 增加运算速度
11 train_data = torch.utils.data.DataLoader(train_set,
12     batch_size=32, shuffle=True)
13 test_data = torch.utils.data.DataLoader(test_set,
14     batch_size=32, shuffle=False)
15
16 epochs = 1
17 model = ResNet18().to(device)
18 criterion = nn.CrossEntropyLoss()
19 optimizer = optim.Adam(model.parameters())
20
21 # train
22 loss_list = []
23 for epoch in range(epochs):
24     running_loss = 0.0
25     data_loss = []
26     for i, (inputs, labels) in enumerate(train_data, 0):
27         inputs, labels = inputs.to(device), labels.to(device)
28         optimizer.zero_grad()
29         outputs = model(inputs)
```

```

30         loss = criterion(outputs, labels)
31         loss.backward()
32         optimizer.step()
33
34         data_loss.append(loss.item())
35         running_loss += loss.item()
36         if i % 100 == 99:
37             print('[%d, %5d] loss: %f' % (epoch + 1, i + 1,
38                 running_loss / 100))
39             running_loss = 0.0
40         loss_list.append(data_loss)
41
42 loss_list = np.array(loss_list)
43 loss_list = loss_list.mean(axis=1)
44 plt.figure()
45 plt.scatter(range(epochs), loss_list, label='loss')
46 plt.legend()
47 plt.show()
48
49 # test
50 correct = 0
51 total = 0
52 with torch.no_grad():
53     for data in test_data:
54         images, labels = data
55         images, labels = images.to(device), labels.to(device)
56         outputs = model(images)
57         _, predicted = torch.max(outputs.data, 1)
58         total += labels.size(0)
59         correct += (predicted == labels).sum().item()
60

```

```
61 print('Accuracy: ', correct / total)
62 # torch.save(model, "model.pth")
```

实验结果

最终使用预训练 + 微调的训练方式，训练 20 轮，得到的 MNIST 数据集的预测准确率可以达到 99.22%。使用交叉熵函数作为误差统计的变化结果如下图所示。

```
[20, 1200] loss: 0.002253
[20, 1300] loss: 0.007238
[20, 1400] loss: 0.007101
[20, 1500] loss: 0.005957
[20, 1600] loss: 0.005999
[20, 1700] loss: 0.012749
[20, 1800] loss: 0.006322
Accuracy: 0.9922
```

图 5: 训练 20 轮的准确率

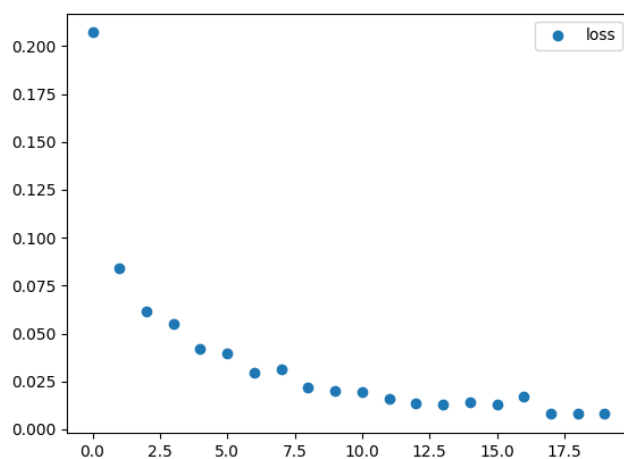


图 6: loss 变化曲线

三、对该预训练模型和“预训练 + 微调”的理解

预训练模型

本次实验使用的是 ResNet 模型，相比于传统的简单的卷积神经网络模型，其有更深的层数与更多参数的优势，可以在少量的训练轮数内训练出一个很好的结果；对比深层的卷积神经网络来说 ResNet 利用独有的残差块可以有效避免梯度爆炸/消失的问题。同时相比于庞大的 VGG 模型，ResNet 由于使用了残差块以及全局平均池化的结构，可以有效减少训练参数，对于小数据集可以有效避免过拟合现象的发生，同时也减少了训练的成本。

“预训练 + 微调”模式

“预训练 + 微调”的模式广泛应用于大模型或大数据集的情况，由于处理大模型需要很高的算力以及训练时间，所以可以使用一些算力高的机器和泛化性强的数据集预先进行大规模训练，得到一份训练好的参数之后进行模型的存储，这样就可以直接调用训练好的能力强的模型直接进行数据集的处理。

但不是所有的时候预训练模型都能很好的拟合当前数据集，这时候就需要使用微调手段对当前数据集进行二次训练。利用预训练模型得到的参数，在其基础上，根据模型的架构进行训练得到一个能够较好拟合当前数据集的新的模型。这种方式就是微调。可以根据需要对模型的参数和结构进行调整来满足不同的任务要求。

“预训练 + 微调”的模型可以有效解决训练成本的问题。直接调用已有的泛化能力以及预测能力好的参数和模型，根据本地需求进行相应的训练，得到一个能够很好符合不同需求的模型，极大的节省了得到高效率模型的计算成本，是一种快速且有效的训练模式。

四、实验总结

本次实验主要尝试了“预训练 + 微调”的训练模式，通过与之前自己构建网络的训练成果对比，可以明显感受到预训练对于准确率的提升非常

巨大。我通过调用 ResNet18 模型进行预训练和微调，不仅掌握了这样一个有效的模型训练方法，同时还理解了 ResNet 的网络结构，甚至还为电脑配置了 cuda 加速，可以说是收获颇丰。感谢助教和老师对 project1 的三个实验的精心设计，让我从简单的手写 bp 网络到使用 cnn 的过程看到了模型结构对预测结果的巨大影响。通过 cnn 实验让我学会了许多人工智能的数据处理以及训练的技巧。通过预训练实验让我体会到预训练过程对于准确率的巨大的提升，并且可以使得模型更加泛化。可以说三个实验层层递进，互相关联，为我之后的人工智能的学习打下坚实的基础。