人工智能基础 第二次大作业
实验报告
自 72 2017011607 来昆

一、 任务描述

本次大作业要求通过深度学习解决图像分类问题，以在线比赛的形式进行，完成 10 类图片的分类问题，图像示例及类别如下：

本次作业的数据包含 30000 张图片组成的训练集（train.npy 和 train.csv）以及 5000 张图片组成的测试集（test.npy），均可以从 kaggle 网站 Data 栏目下下载。npy 文件可通过 numpy.load()函数读取，每个 npy 文件包含一个 N*784 的矩阵，N 为图片数量。矩阵每行对应一张 28*28 的图片。

在预测环节，需要利用训练好的模型对测试集中的 5000 张图片进行分类，预测结果应生成 submit.csv 文件，同样包含 image_id 和 label 两列，共 5000 行，每行对应一张图片的结果。

在 kaggle 网站上提交该文件后会看到自己的分数（指标为 catergorization accuracy）及排名。

编程语言要求：

编语言原则上应使用 Python，深度学习框架可从 pytorch/tensorflow(keras/caffe 中任选其一。

大作业需提交以下三份材料：

1. 源代码，通过网络学堂提交；
2. 实验报告，通过网络学堂提交；
3. 预测结果（csv 文件），通过 kaggle 网站提交，在线实时排名。

二、 设计与实现

完成图像分类任务，整体分为以下几个部分：

数据读取和预处理；网络搭建；训练网络保存模型；预测结果并输出保存预测结果；参数选取和调节；

下面分别介绍几部分的设计及实现。

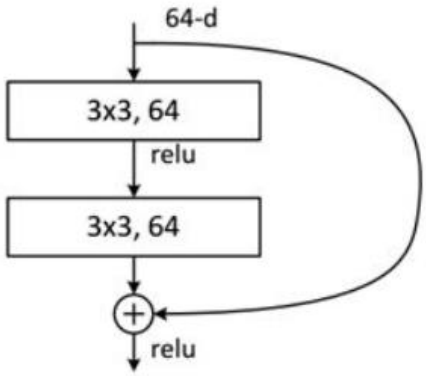
1. 数据读取和预处理
   1) 使用神经网络对图片进行处理，需要进行 pooling,convolution 等操作，因此，输入网络的需要是矩阵，而 npy 文件中的每张图片矩阵为 784*1，需要用 numpy 库函数 reshape 成 28*28 的矩阵。
   2) 由于我选用了 pytorch 作为框架，需要将数据类型转换成 pytorch 网络的标准数据类型 tensor。
   3) 对于标签文件 csv，使用 pandas 库进行读取，并把 label 保存为 np 矩阵形式。
2. 网络选取和搭建

   基本的卷积神经网络结构分为输入层，卷积层，全连接层，还要选取激活函数、优化器等内容。尝试过自己搭建的卷积神经网络，最终提交的 Accuracy 最高的 csv 是参照 ResNet18 残差神经网络搭建的模型训练后预测的结果。

   下面介绍残差神经网络的内容和上述几部分的选取，结构及作用。

   1) ResNet 简单介绍

ResNet 主要解决的是网络过深的时候的退化问题，与一般卷积神经网络的最大不同就是引入了残差块的使用，对残差进行学习，加入 shortcut connection，保证即使在残差为 0 的情况下至少做恒等映射而不导致网络退化。残差块的基本结构如图所示。



输入和输出维度相同的时候，直接将输入加到输出上，维度不一致时，进行卷积来使维度相同。

使用 ResNet 的原因是，本次任务的图像过小，既难以进行大 kernel 的卷积操作，又难以使用太多层数的卷积，因为本身的特征太少，卷积后 channel 增加，很容易出现过采样和梯度消失等问题，导致网络的退化，也容易产生过拟合的现象。通过残差块来减少这样的情况出现。

2) 输入层

输入层即为 28*28 的图片矩阵。

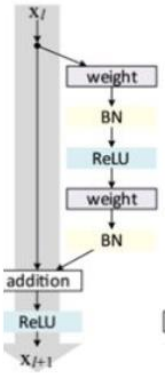
3) 卷积层

只使用了一个卷积层，使用 3*3 的 kernel，步长 stride 为 1，padding 为 1，无偏置，输出 channel 数为 64.

根据式子 $K=\frac{N-m+2P}{S}+1$，输入与输出的数据大小 K 相同。

4) 残差层

使用的残差层的基本模型如图所示：



主通道为两层与卷积层参数相同的卷积，卷积后进行 BatchNormalize 并 ReLU 激活，shortcut connection 与前面的描述相同，如果输入输出通道数相同，就直接与主通道结果相加，不同则进行卷积并 BatchNormalize。求和后进行 ReLU 激活，送入下一层。

5) Pooling 层

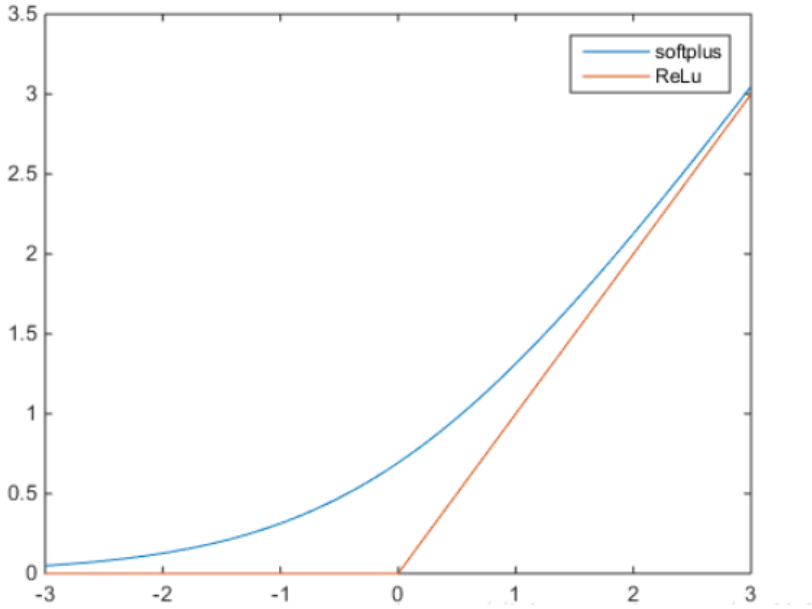
在所有层结束后，全连接层之前，加入 pooling 层进行降维，根据 ResNet 模型的方法，使用边长为 4，步长为 4 的 Avgpooling。

6) 全连接层

在全连接层中，先用 Dropout 舍弃 30%的输出参数，然后对剩下的神经元进行全连接操作。

7) 激活函数

根据助教大作业讲解的推荐和上课学习，使用 ReLU 函数作为激活函数。



8) 优化器 Optimizer

一开始尝试使用上课学到的梯度下降法 SGD(Stochastic Gradient Descent)，并加入 Momentum 参数防止陷入局部最优点，并加入了 L2 正则化防止过拟合。提交到 kaggle 的正确率达到 86.2%。但是根据训练时输出的 Loss 和训练 Accuracy，训练速度有些慢，需要增加 Epoch 的数目，但 Epoch 增加，过拟合的几率也随之变高。于是经过搜索和查找相关资料，使用了 Adam 优化器，训练速度大大提高。

9) Loss 计算

根据上课学习的知识，使用了交叉熵 CrossEntropy 作为损失函数。

3. 训练网络保存模型

使用 sklearn 库中的 KFold 将数据集切分为 5 份，依次训练 5 个模型。

训练之前，用 KFold 将训练集切分成 5 份，循环 5 次，每次抽其中 4 份做训练。训练过程中每个 Epoch 结束后，计算训练集上的正确率和损失，并输出。

训练开始，先将 np 数据转换成 Pytorch 能处理的 Tensor 类型，建立 DataLoader，初始化 Loss 和 Accuracy。

从训练的 loader 中循环读取，每次读取 1 个 Batch 的数据，进行训练。

将输入读入模型中，得到输出，初始化优化器梯度(zero_grad)，用 LossFunction 计算损失并将 Loss 反向传播，然后使用优化器进行梯度下降，更新所有参数(.step())，完成一个 Batch 的训练。

对 Loss 求和并除 Batch 的个数得到平均 Loss,以及对输出的 10 个类的值中取最大值作为预测的结果计算正确率,每个 Epoch 输出一次。
对于一个模型，跑完全部 Epoch 后，储存模型参数。
训练部分的代码:

```
#当在整个训练集上进行5个模型的训练时
j = 0
for np_X_train, np_Y_train, np_X_test, np_Y_test in kfold(np_train_images,np_train_labels,5):
    model = ResNet18()
    model.cuda()
    #optimizer = torch.optim.SGD(model.parameters(),lr=LR,momentum=0.9,weight_decay=5e-4)
    optimizer = torch.optim.Adam(model.parameters(),lr=LR)
    X_train = torch.from_numpy(np_X_train)
    Y_train = torch.from_numpy(np_Y_train)
    X_test = torch.from_numpy(np_X_test)
    Y_test = torch.from_numpy(np_Y_test)
    dataset = TensorDataset(X_train,Y_train)
    test_dataset = TensorDataset(X_test,Y_test)

    train_loader = DataLoader(
        dataset,batch_size=Batch_size,shuffle=True)
    test_loader = DataLoader(
        test_dataset,batch_size = 1,shuffle = False)
    loss_val = []
    acc_val = []
    predictions = []
    print("######")
    print(j)
    for epoch in range(EPOCH):
        train_loss = 0
        train_acc = 0
        model.train()
        for i,(x,y) in enumerate(train_loader):
            batch_x = Variable(x.cuda())
            batch_y = Variable(y.cuda())

            output = model(batch_x)

            loss = loss_func(output, batch_y)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()


            train_loss += float(loss)
            _,pred = output.max(1)
            predictions.append(pred)
            corrects = (pred == batch_y).sum()
            acc = int(corrects)/batch_x.shape[0]
            train_acc += acc
        loss_val.append(train_loss/len(train_loader))
        acc_val.append(train_loss/len(train_loader))
```

```
        print("epoch"+' '+str(epoch))
        print("Acc"+' '+str(train_acc/len(train_loader)))
        print("Loss"+' '+str(train_loss/len(train_loader)))
    torch.save(model.state_dict(),'multipara_Resnet_19_2_'+str(j)+'.pkl')
    j += 1
```

某次训练的部分输出:

```
#####
0
epoch 0
Acc 0.703873005319149
Loss 0.7851690613525979
epoch 1
Acc 0.8047706117021277
Loss 0.5196055925272881
epoch 2
Acc 0.8413397606382979
Loss 0.4302480506136062
epoch 3
Acc 0.8619930186170213
Loss 0.3739982973387901
epoch 4
Acc 0.875748005319149
Loss 0.33808824554719824
epoch 5
Acc 0.8901678856382979
Loss 0.29611675599788095
```

4. 预测结果并输出保存

   预测结果分为 2 种类型，第一种：对已有 label 的数据集进行切分，使用一部分训练，剩下的部分进行预测，来观察训练的结果和网络参数的合理性；第二种：对 test.npy 中的图片进行预测，输出预测结果到 csv 中。

   第一种:

   1） 实例化网络，选择为评估模式，将网络参数送到 cuda 中。
   2） 转换 test 变量为 Tensor 类型，读取网络参数并对数据集进行预测，重复五次。
   3） 将 5 个模型预测结果相加, 取结果中 10 个数值的最大值作为预测的分类结果输出。
   4） 将预测正确的数据个数求和，计算正确率。

   第二种:

   基本与第一种相同，只是依次将预测结果添加到 np 数组中，最后全部预测完成后用 pandas.DataFrame 标明 image_id 和 label。并输出到指定名称的 csv 中。

5. 参数的选取和调节

   主要需要调节的参数为网络内部和训练参数。网络内部参数包括网络深度，卷积层中的 kernel,padding,stride,Pooling 层 Pooling 窗口的大小，Dropout 的比例等。训练参数包括 Batch_size,Learning_rate,优化器中的 weight_decay,momentum 等。

   本人的参数选取基本参照文章中和经典网络的参数选取, 训练得到 Baseline 结果后，逐步根据输出的 Loss，Accuracy，训练时间，显存占用等进行调节优化。

   例如:

   a. 原 ResNet18 中层数很多，我把层数降低到 1 个卷积层加 4 层残差层，共 9 层卷积，对本次任务也已经足够了。
   b. 原模型中没有 Dropout，我在最后一层 ReLU 之前加入 Dropout，再进行 ReLU 激活，最终送入全连接层。
   c. Dropout 先设定为 50%，经过几次尝试，最后确定在 30%

d. 为防止显存不够用，训练 Batch_size 先选定为 64，训练过程发现时间太久，改成 128。预测时 Batch_size 选为 100.
e. LearningRate 尝试过多次，学习率过低 Epoch 数就要增加，容易造成过拟合，学习率过高难以找到最优点，网络可能在很少几个 Epoch 后就无法优化。最终选定为 0.001
f. 使用 SGD 优化器时，根据一些文章讲解，设定 momentum=0.9,weigh_decay=5e-4，后改用 Adam 优化器，根据介绍，没有设定 momentum 和正则化参数。

6. GPU 加速

由于网络参数多，又使用 KFold 进行多模型训练交叉验证，使用 CPU 训练时间代价太高，于是使用 GPU 进行训练，在网络实例化后加入.cuda，将输入输出、训练参数等都送入 GPU 中进行训练。并在 cmd 中使用 Nvidia-smi 指令监测显存使用情况。相关代码如下：

```
model.cuda()
```

```
batch_x = Variable(x.cuda())
batch_y = Variable(y.cuda())
```

三、 实验结果

1. 正确率

最终的正确率为 PUBLIC:88.933%,排名 95
PRIVATE：89.200% 排名 98
队伍名称：a710483

2. 输出的网络参数文件

附在压缩文件包中的 pkl 文件夹中

四、 实验总结

一）过程中遇到的问题

1. 经常出现的 CUDA out of memory

训练过程和预测过程中都出现了 out of memory 的报错，上网查找相关资料后，发现使用 GPU 进行训练模型，每次要预留模型参数的空间，预留训练数据训练所需的空间，如果送进网络的数据加倍，需要留出的空间也加倍，因此训练过程中减小 BatchSize，减少了每次训练需要的空间，也就不会出先训练时显存不够的情况了。在预测时出现 out of memory 也是差不多的情况，只要减少预测的 BatchSize，也可以解决这一问题。但 BatchSize 减小，训练所需的时间也随之增加，训练的效果也会有一定的影响，因此 BatchSize 的调节也需要尝试和总结。

2. 过拟合

开始自己搭建的网络经常出现过拟合的情况，在提交的结果正确率比在训练集上低 10%以上是很经常的事情，加入了 Dropout 和 BatchNorm 也没有大的改观，最后通过以下方式减少了过拟合的程度：a)更改网络结构，使得增加层数也能得到较好的学习结果，因此学习速率加快，需要的 Epoch 数减少;b)使用 KFold 分割数据集的方法，每个模型都只使用了部分的训练集，最终的预测使用多个模型共同预测输出结果，降低了过拟合程度。

二）可能的改进

受时间限制，没能继续调参并优化网络。对进一步优化网络和训练数据，有以下的思考。

1. 对图像进行预处理:
   在很多文章和讲解中，都提到了预处理图片的方式来增加数据和防止过拟合，通过 Padding 后裁剪图片，旋转图片等方式来进行预处理。

2. 优化器中增加正则化参数防止过拟合
   最后使用的 Adam 优化器的优化方式和算法并不是很明白，只是参照了网上文章的讲解，也没敢乱添加参数，后续的改进中可以进一步尝试各种参数，观察效果。

3. 使用其他的数据集分割方式
   在调用 KFold 和 train_test_split 时，我发现即使加入 shuffle 和 random，最后分割出的结果仍然不是随机和充分 shuffle 过的，可能时 sklearn 函数本身的特点吧。如果自己写随机分割的函数，可以将数据集进行更多类型的分割，训练更多的模型，采用 10 folds 等，进行共同预测，可能会提高预测集合上的准确率。

三）总结和反思

经过本次大作业的练习，掌握了 CNN 的基本搭建方式，也通过不断地调参，知道了如何根据具体的任务和情况对网络进行优化，也把上课学到的各个单元的知识进行了更好的串联，在为了 Accuracy 努力的过程中不断搜索查找资料，也加强了上课讲过的关于各个结构的知识的理解和掌握。Kaggle 比赛的形式也十分有趣，不断 push 我们继续优化的过程比起我们自己自己做作业搭一个基础的网络，起到了更好的锻炼效果。

感谢助教多日来的耐心答疑和比赛维护!辛苦啦!