人工神经网络

Lab 4：卷积神经网络(CNN)

数据科学与计算机学院　17大数据与人工智能
17341015　陈鸿峥

目录

一、 均方差损失

均方差MSELoss的实现在01.toy.py中，核心代码如下。

```python
def mse_loss(input, target):
    n = input.shape[0]
    one_hot = torch.zeros(input.shape)
    one_hot[torch.arange(n),target] = 1
    return torch.mean((input - one_hot) ** 2)
```

注意输入的input是每个类别的概率值，而target仅仅是一个目标类别。故需要先将target用独热码编码为与input维度一致，这里用到NumPy的fancy indexing进行独热码的创建。

运行结果如图1所示，可见成功学习到异或规律。

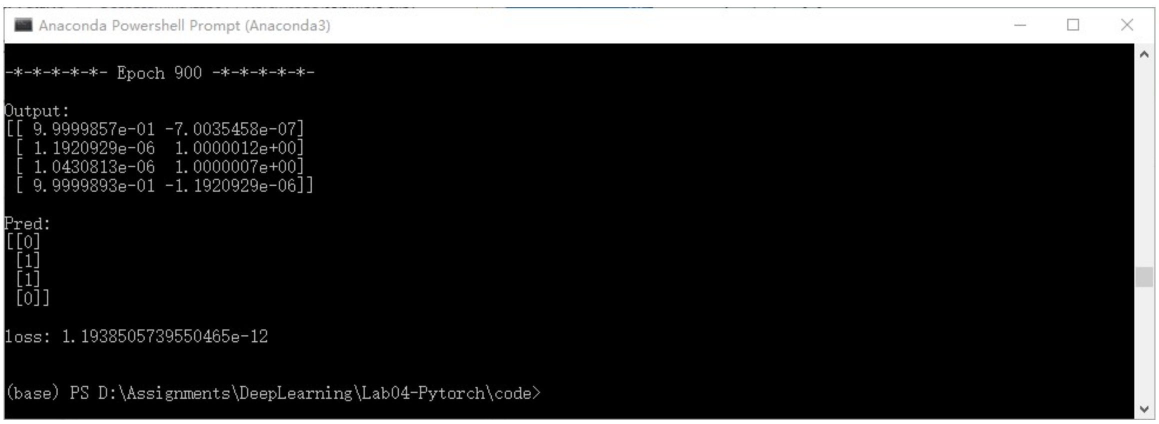


图 1: 题1结果

二、 学习数数

CNN的模型在02.learn-to-count.py中，核心代码如下。

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # in_chan, out_chan, kernel_size
        self.conv1 = nn.Conv2d(1, 8, 3, stride=1)
        self.relu = nn.ReLU()
        self.fc = nn.Linear(26 * 26 * 8, 10)

    def forward(self, x):
        output = self.conv1(x)
        output = self.relu(output)
        output = output.view(output.shape[0],-1)
        output = self.fc(output)
        return output
```

运行结果如图2所示，可见在MNIST数据集上分类准确率达到了98.04%。

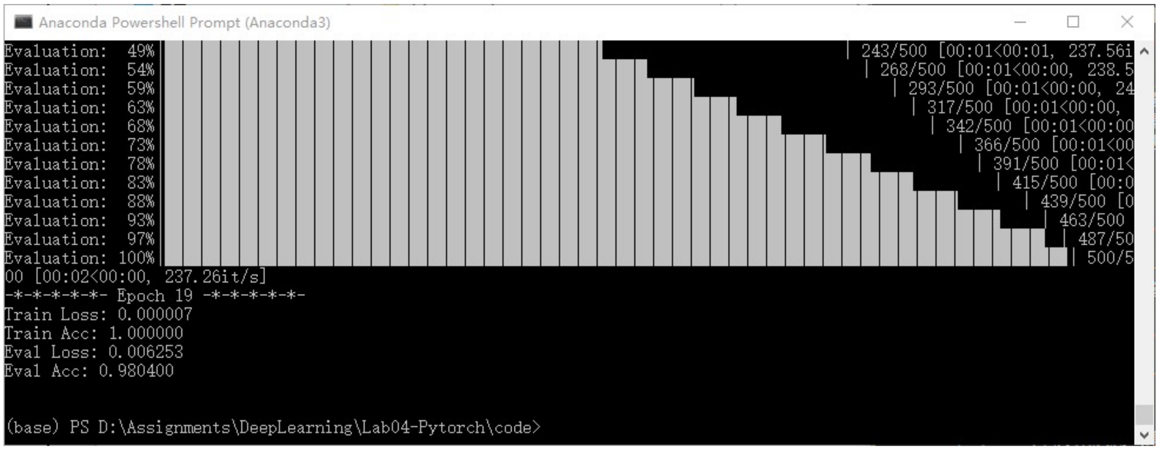


图 2: 题2结果

三、 简单可用的AI

在本实验中我实现了两个网络LeNet5 [1]和VGG16 [2]，网络定义参见`mynet.py`，完整训练代码见`03.simple-ai.py`。为了避免过拟合，我采取了以下措施：

- **Dropout**：在全连接层前面添加`nn.Dropout`，以一定概率（默认为0.5）隐藏神经元
- **批归一化**：在每个卷积层后面添加`nn.BatchNorm2D`，实现批归一化(batch normalization)
- **早停**：在`myutils.py`中实现了`EarlyStopping`类，通过判断验证集上的损失是否持续下降，来决定是否继续训练。

LeNet和VGG的代码如下，都用`nn.Sequential`进行封装，由于论文中对其网络架构已经描述得很清楚了，故在PyTorch上只需将各层合并起来即可。这里提前将论文中提及的不同层数的网络结构用`vgg_config`进行描述，在网络初始化过程中才将对应的卷积层插入。

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class LeNet(nn.Module):
    """
    TODO: Implementation of a simple Convolutional neural network.
    HINT: You can refer to the baby model in ‘01.toy.py‘, and
          the document of PyTorch from the official website.
    """
    """YOUR CODE HERE"""
    def __init__(self):
        super(LeNet, self).__init__()
        self.features = nn.Sequential( # 3*32*32
```

```
            nn.Conv2d(3, 6, 5), # 6*28*28
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2, 2), # 6*14*14
            nn.Conv2d(6, 16, 5), # 16*10*10
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2, 2) # 16*5*5
        )
        self.classifier = nn.Sequential(
            nn.Linear(16 * 5 * 5, 120),
            nn.ReLU(inplace=True),
            nn.Linear(120, 84),
            nn.ReLU(inplace=True),
            nn.Linear(84, 10)
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(-1, 16 * 5 * 5)
        x = self.classifier(x)
        return x
    """END OF YOUR CODE"""

vgg_config = {
    'VGG11': [64, 'M', 128, 'M', 256, 256, 'M', 512, 512, 'M', 512, 512, 'M'],
    'VGG13': [64, 64, 'M', 128, 128, 'M', 256, 256, 'M', 512, 512, 'M', 512, 512, 'M'],
    'VGG16': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 512, 512, 512, 'M', 512, 512, 512, 'M'],
    'VGG19': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 256, 'M', 512, 512, 512, 512, 'M', 512, 512, 512, 512, 'M'],
}

class VGG(nn.Module):
    """
    Ref: Karen Simonyan, Andrew Zisserman
         Very Deep Convolutional Networks for Large-Scale Image Recognition
         ICLR, 2015
    """
    def __init__(self, name):
        super(VGG, self).__init__()
        self.features = self._make_layers(vgg_config[name])
        self.classifier = nn.Sequential( # three fcns
            nn.Dropout(), # avoid overfitting
            nn.Linear(512, 512),
            nn.ReLU(inplace=True),
```

```
            nn.Dropout(),
            nn.Linear(512, 512),
            nn.ReLU(inplace=True),
            nn.Linear(512, 10)
        )

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

    def _make_layers(self, cfg):
        layers = []
        in_channels = 3
        for out_channels in cfg:
            if out_channels == "M": # max pooling
                layers += [nn.MaxPool2d(2)]
            else:
                # preserve image resolution
                layers += [nn.Conv2d(in_channels, out_channels, kernel_size=3,
                    ↪ padding=1),
                           nn.BatchNorm2d(out_channels), # avoid overfitting
                           nn.ReLU(inplace=True)]
                in_channels = out_channels
        return nn.Sequential(*layers)
```

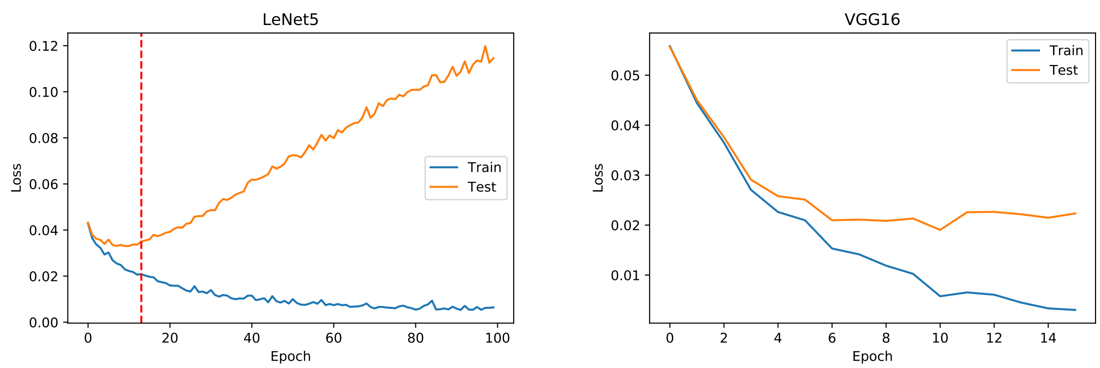
早停的代码实施如下，其中patience即可忍耐的损失不下降的轮数。

```
class EarlyStopping():

    def __init__(self, patience=5):
        self.patience = patience
        self.cnt = 0
        self.loss = []
        self.best_loss = None

    def __call__(self, eval_loss): # one number, not an array
        if self.best_loss is None:
            self.best_loss = eval_loss
        elif eval_loss < self.best_loss:
            self.cnt = 0
            self.best_loss = eval_loss
        else:
            self.cnt += 1
            if self.cnt >= self.patience: # early stopping
```

```
                return True
        self.loss.append(eval_loss)
        return False
```

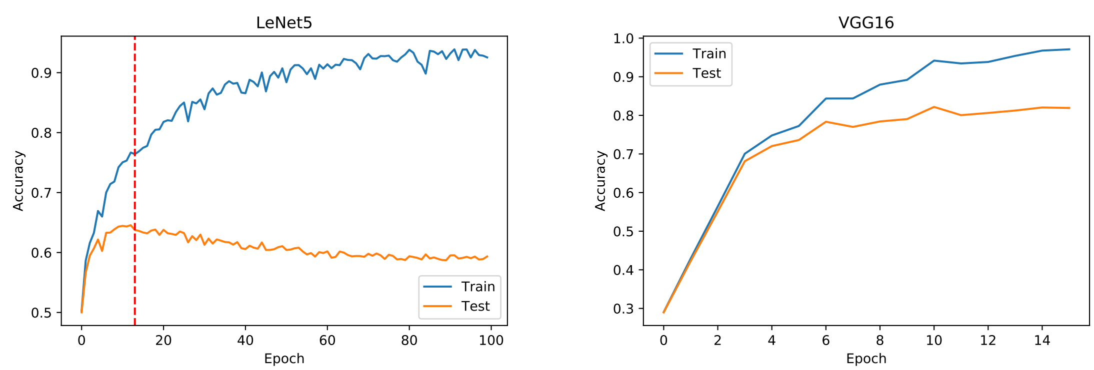
训练和验证的部分复用了`02.learn-to-count.py`的代码。同时增添了对训练、测试损失及精度的存储（以`.npz`格式），方便后续的可视化工作。另外由于网络训练实在太慢，故在本实验中我只选择了LeNet5和VGG16进行训练，同时使用了GPU[1]进行加速，批次大小32，学习率为10^{-3}，采用Adam优化器。

最终实验结果如下，图3为损失函数变化，图4为精度变化。可以看到采取了早停策略，LeNet5才不会继续过拟合，其在测试集的最高准确率为64.56%。对于VGG16，同样采取了早停策略[2]，可以看到对于最后几个epoch，损失函数和精度的变化都已经变缓了，因此早停可以有效避免继续训练的过拟合。



(a) LeNet5损失（红线为早停点）　(b) VGG16损失

图 3: 训练集及测试集Loss变化



(a) LeNet5准确率（红线为早停点）　(b) VGG16准确率

图 4: 训练集及测试集准确率变化

[1]Nvidia GTX 1050, CUDA 10.1

[2]当然最好是将patience设大一些，同时达到patience后降低学习率，不过为了方便本实验并没有做学习率衰减的测试。

图5展示了两个网络的准确率比较，它们都超过了60%的基准值，同时VGG16明显好于LeNet5的性能，最高达到了82.18%的准确率。不过由于时间和硬件的限制，本实验并没有继续做更多的调优，理论上VGG的准确率可以达到更高。

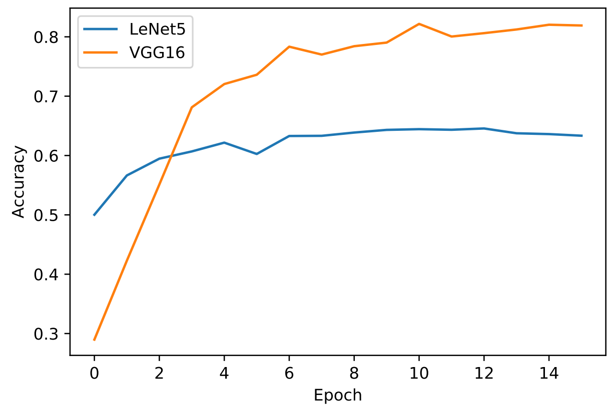


图 5: LeNet5与VGG16在CIFAR-10测试集上准确率比较

参考文献

[1] Yann Lecun, Léon Bottou Bottou, Yoshua Bengio, and Patrick Haffner, "Gradient-based learning applied to document recognition," in Proceedings of the IEEE, vol. 86, no. 11, pp. 2278-2324, Nov. 1998.

[2] Karen Simonyan, and Andrew Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in Proceedings of the International Conference of Learning and Representation (ICLR), 2015.