期中project

CNN

实验原理

CNN的引入

全连接神经网络可以用于拟合非线性的数学模型，且有很好的效果，但是对于数字图像，则存在几个缺点：

1. 将每个像素考虑为变量忽视了像素之间的相邻特征，会影响局部特征的获取
2. 数字图像的像素一般比较大，不进行预处理会导致参数非常多，而且容易过拟合。
3. 数字图像的RGB通道是有明显关联的，应该综合考虑以充分挖掘局部的语义信息。

因此有人提出了卷积神经网络，该网络有三个特点：

1. 局部连接，通过滤波器能够挖掘局部的特征，比如边缘检测
2. 权值共享，不同位置的像素能够利用同一个卷积核的信息，减少参数个数
3. 空间上的下采样，能够对数字图像进行简化处理，加快训练速度，简化网络结构。

CNN的网络结构

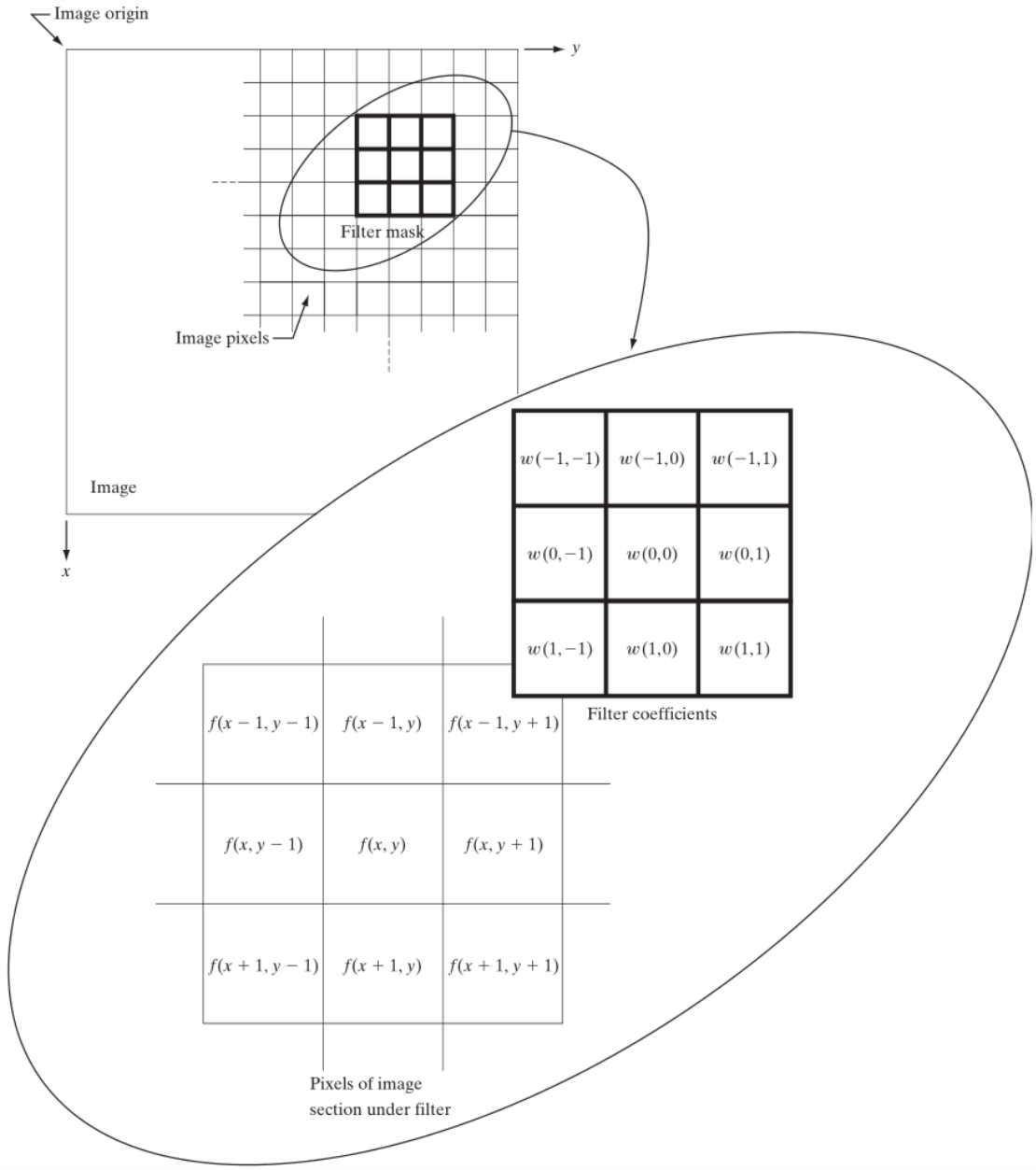
卷积层

卷积是数字图像中的一种处理方法，卷积公式如下：

$f(x,y),h(x,y)$分别是图像和滤波器在空间域上的公式。

另一种计算方式是使用$h(x,y)$对应的卷积核进行滤波，

图示如下：



而实际上CNN使用的卷积与上面所说的两种卷积方法都是不一样的，它的卷积等价于数字图像中的互相关(Correlation)，数字图像中的**卷积**和**互相关**的差别在于卷积需要将图像（或者滤波器）旋转180°再做对应加权求和，而互相关是直接进行加权求和，由于CNN的滤波器是学习时不断更新的，是否旋转180°没有任何影响，并且旋转时需要多余的操作，因此使用的是(2)式。

CNN的卷积方法还可以指定步长(stride)和填充值(padding)，前者可以使得卷积得到的图像更小，后者可以在卷积前对图像边缘进行填充，根据卷积核的大小进行设置，可以使得卷积后图像大小不变，如：3×3的卷积核和1的步长，以及1的填充值可以使得图像大小不变。

由于数字图像可以是有多个通道的，因此卷积核也需要有多个通道。此外，卷积时可以使用多个滤波器进行滤波，这样就可以使得输入的通道与输出的通道不一样。

此外，一个卷积层的每个滤波器都有自己的偏置。

以$6\times 6\times 3$的图像（3代表RGB三个通道）为例，如果使用5个$3\times 3\times 3$的卷积核（最后一个3与RGB三个通道对应），则最终会得到$4\times 4\times 5$的输出，输出前5个通道的图像都会加上各自的一个偏置。即一个5×1的向量。

池化层

池化层用于减少参数数量，一般有最大池化和平均池化两种。基本相当于下采样。

全连接层

将图像展开成一个维度的向量。

输出层

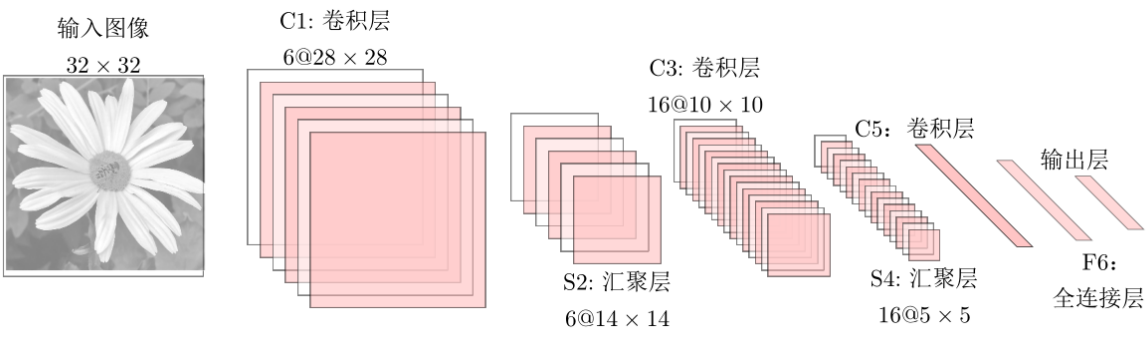
经过softmax层，输出结果最大的作为分类的预测结果。

下面是一种CNN网络结构，

```
class CNN(nn.Module):
    def __init__(self,in_dim,n_class):
        super(lenet5, self).__init__()
        self.l1 = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=6, kernel_size=3, stride=1
                      ),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Sigmoid()
        )
        self.l2 = nn.Sequential(
            nn.Conv2d(in_channels=6, out_channels=16, kernel_size=3,
stride=1
                      ),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Sigmoid()
        )
        self.l3 = nn.Sequential(
            nn.Conv2d(in_channels=16, out_channels=120, kernel_size=3)
        )
        self.linear = nn.Sequential(
            nn.Linear(1920, 10)
        )

    def forward(self, x):
        out1 = self.l1(x)
        out2 = self.l2(out1)
        out3 = self.l3(out2)
        out3 = out3.view(out3.size(0), -1)
        output = self.linear(out3)
        return output
```

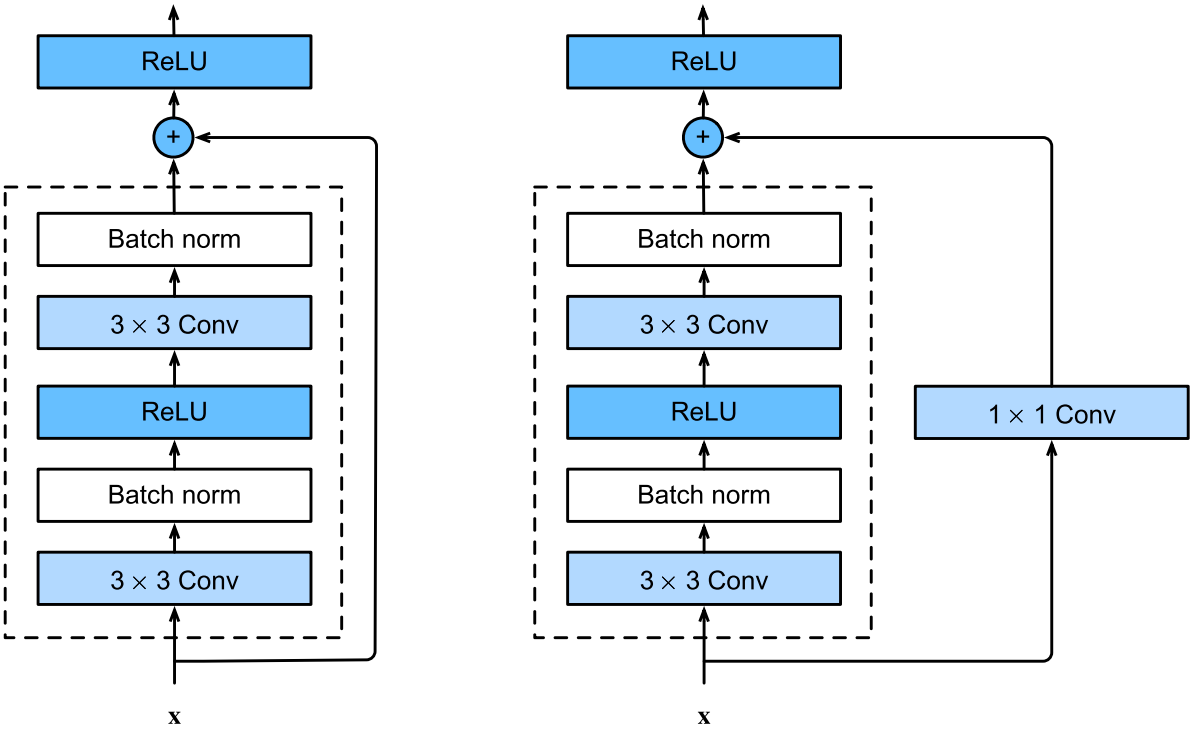
LeNet-5

LeNet-5使用了上面的几种隐层。



ResNet

ResNet方法提出，若拟合的函数是$F(x)$，潜在的映射是$H(x)$，让拟合的函数学习$H(x)-x$比直接学习$H(x)$更简单。尤其是对于较深的网络，使用relu可以保证冗余层不会导致模型的性能更糟，而如果冗余层提取到了特征，则模型的性能会提升。



上图是普通的残差块，注意为了使得$F(x)+x$能够符合矩阵的加法条件，x需要通过**shortcut**来变换，shortcut中的变换与F有关，如果F没有改变图像的大小，则**shortcut为恒等变换**；如果缩小为原来的一半，则shortcut需要进行一次**卷积**来缩小图像。

残差块继承nn.Moudle类，

```
def conv3x3(in_channels, out_channels, stride=1):
    return nn.Conv2d(in_channels, out_channels, kernel_size=3,
stride=stride,
                     padding=1, bias=False)
class ResidualBlock(nn.Module):
    def __init__(self, inchannel, outchannel, stride=1):
        super(ResidualBlock, self).__init__()
        self.left = nn.Sequential(
            conv3x3(inchannel, outchannel,stride),
            nn.BatchNorm2d(outchannel),
            nn.ReLU(inplace=True),
            conv3x3(outchannel, outchannel),
            nn.BatchNorm2d(num_features=outchannel)
        )
        if stride != 1 or inchannel != outchannel:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels=inchannel, out_channels=outchannel,
                          kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(num_features=outchannel)
            )
        else:
            self.shortcut = nn.Sequential()
```

```
    def forward(self, x):
        out = self.left(x)
        out += self.shortcut(x)
        out = F.relu(out)
        return out

```

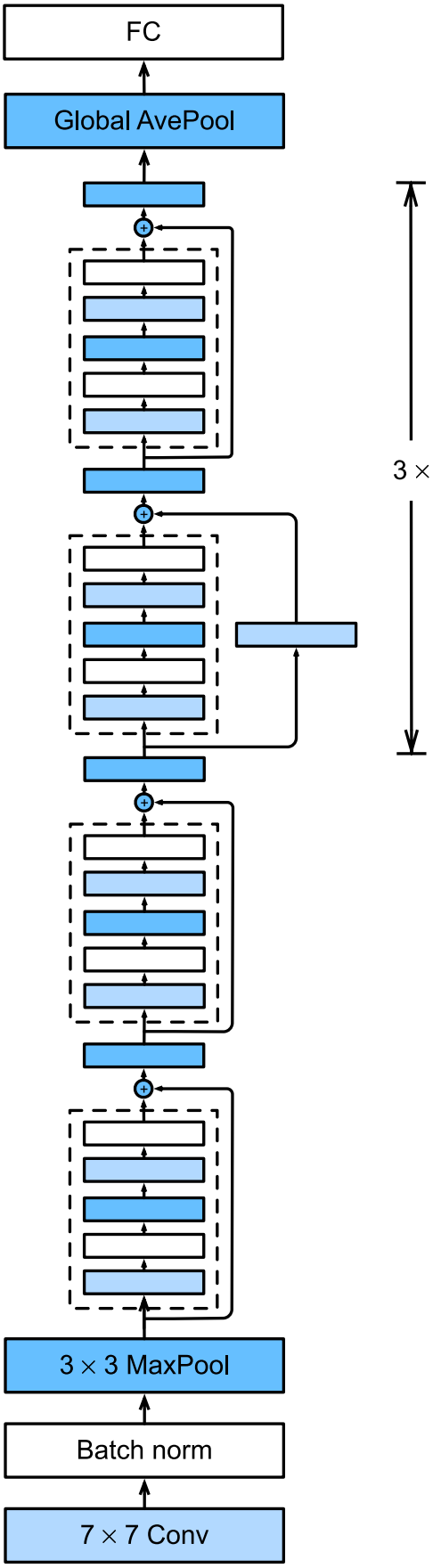
其中涉及到BatchNorm2d，称为**BN层**，该方法对一个batch的feature map的每个channel使用均值和方差（训练中通过学习得到）进行归一化。在ResNet中，每次卷积后都会通过一次BN层。

ResNet论文中提及的种类如下，我使用的是ResNet18。

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

表格中有一点细节没有体现出来，就是conv2_x的部分，因为进行了stride = 2的max pool，图像的长宽已经缩小为原来的一半，因此conv2_x的两个residual_block的stride=1，剩下的残差块则是第一个residual_block的stride为2，第二个为1，这样就能每通过一组residual_block，图像长宽缩小为原来的一半。

完整的ResNet结构如下：



实现堆叠出残差块组的make_layer函数，就可以写出ResNet了，我只实现了18和34版本的，训练使用ResNet18。

```
class ResNet(nn.Module):
    def __init__(self, ResidualBlock, layers, num_classes=10):
        super(ResNet, self).__init__()
        self.inchannel = 64
        self.conv1 = nn.Sequential(
```

```
            nn.Conv2d(in_channels=3, out_channels=64, kernel_size=7,
stride=2, padding=3, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        )
        self.layer1 = self.make_layer(ResidualBlock, 64,  layers[0],
stride=1)
        self.layer2 = self.make_layer(ResidualBlock, 128, layers[1],
stride=2)
        self.layer3 = self.make_layer(ResidualBlock, 256, layers[2],
stride=2)
        self.layer4 = self.make_layer(ResidualBlock, 512, layers[3],
stride=2)
        self.fc = nn.Linear(512, num_classes)

    def make_layer(self, block, channels, num_blocks, stride):
        strides = [stride] + [1] * (num_blocks - 1)
        layers = []
        for stride in strides:
            layers.append(block(self.inchannel, channels, stride))
            self.inchannel = channels
        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.conv1(x)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = F.avg_pool2d(out, 7)
        out = out.view(out.size(0), -1)
        out = self.fc(out)
        return out

def ResNet18():
    return ResNet(ResidualBlock,[2,2,2,2])
def ResNet34():
    return ResNet(ResidualBlock,[3,4,6,3])
```

注意因为cifar-10训练集的图片大小是32*32的，在最后的平均池化层会变为1*1，这样实际上损失了非常多的信息，因此在开始时可以不使用7*7的卷积核和最大池化，而是使用更小的卷积核比如3*3，而且不进行池化，最后的平均池化层也可以使用更小的，比如4*4。

结果分析

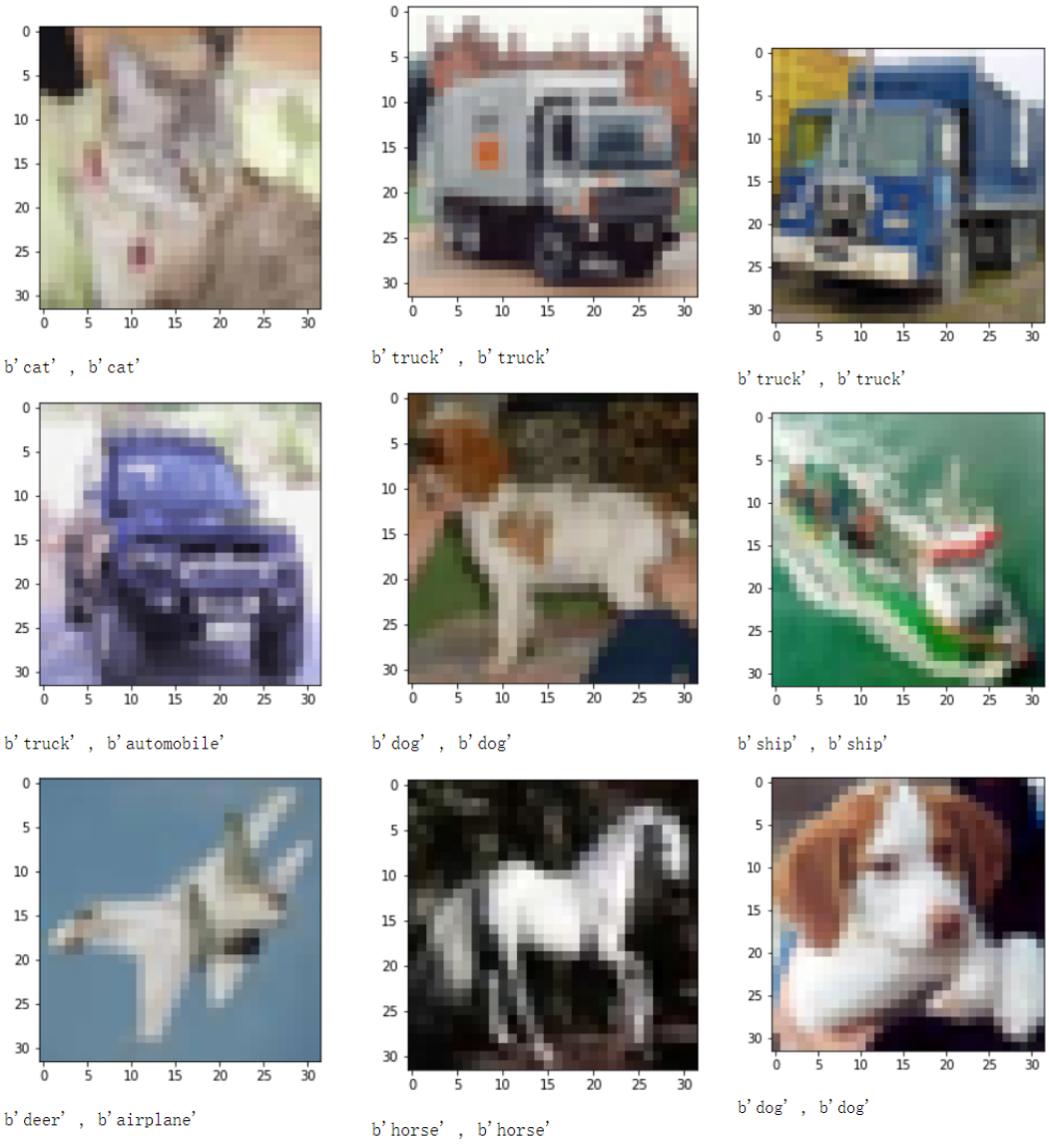
CNN

经过一段时间的训练，准确率可以达到66%。

```
>>> from lenet import *
lenet5(
  (m1): Sequential(
    (0): Conv2d(3, 6, kernel_size=(3, 3), stride=(1, 1))
    (1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (2): Sigmoid()
```

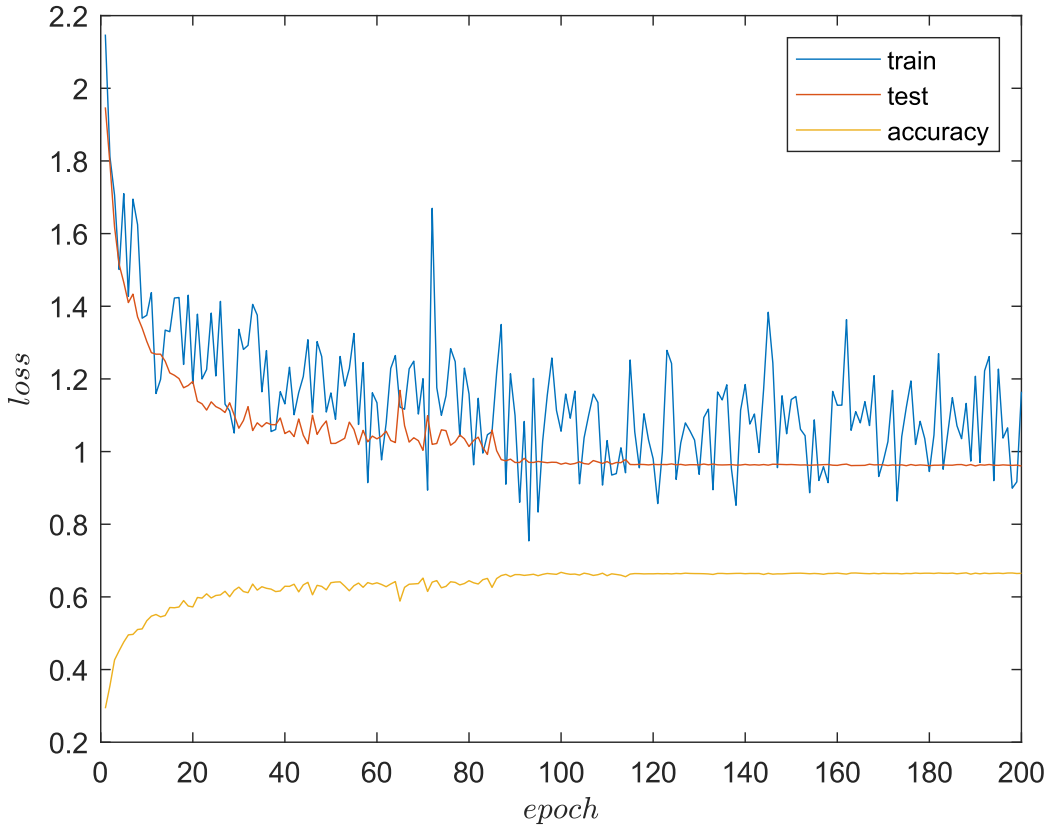
```
  )
  (m2): Sequential(
    (0): Conv2d(6, 16, kernel_size=(3, 3), stride=(1, 1))
    (1): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (2): Sigmoid()
  )
  (m3): Sequential(
    (0): Conv2d(16, 120, kernel_size=(3, 3), stride=(1, 1))
  )
  (linear): Sequential(
    (0): Linear(in_features=1920, out_features=10, bias=True)
  )
)
>>> test(model, 1, criterion, test_loader)

Test set: Average loss: 0.9699, Accuracy: 6634/10000 (66%)
(tensor(0.6634), tensor(0.9699, device='cuda:0'))
```

下面是一些测试结果，可以看到，

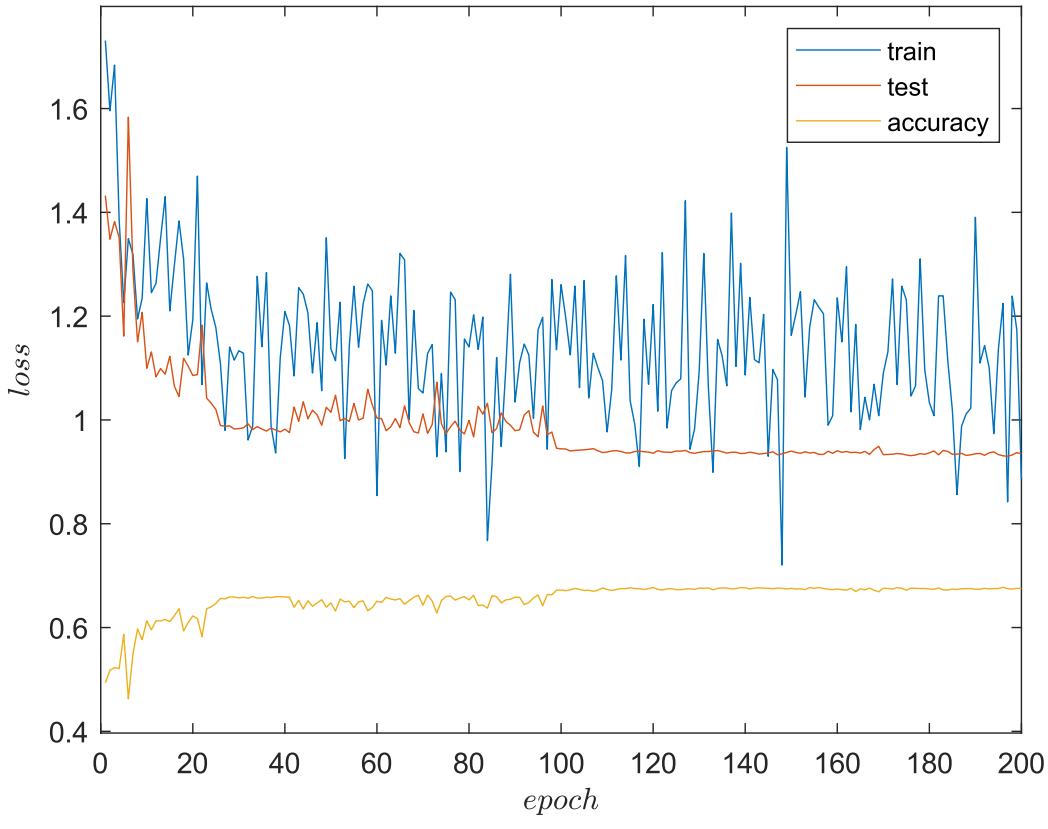


该模型预测错误的，肉眼也比较难识别，因为特征不够明显。

训练的过程中，loss和accuracy变化如下：

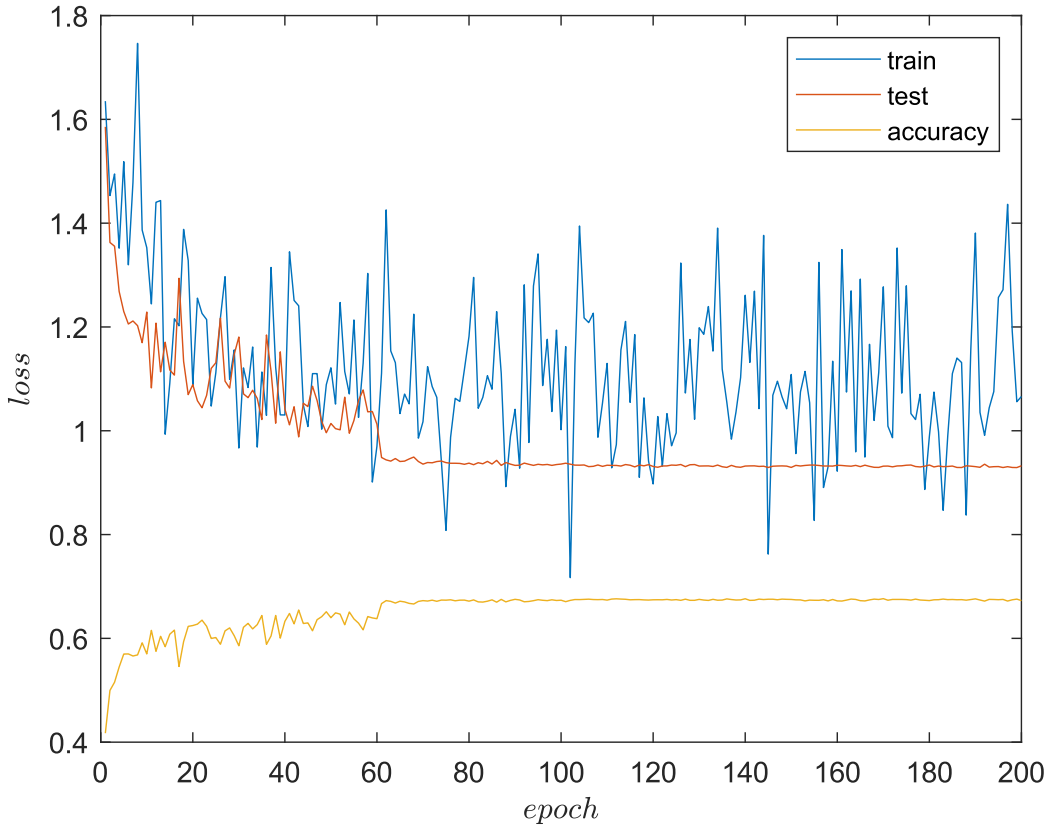


使用dropout后，变化如下：



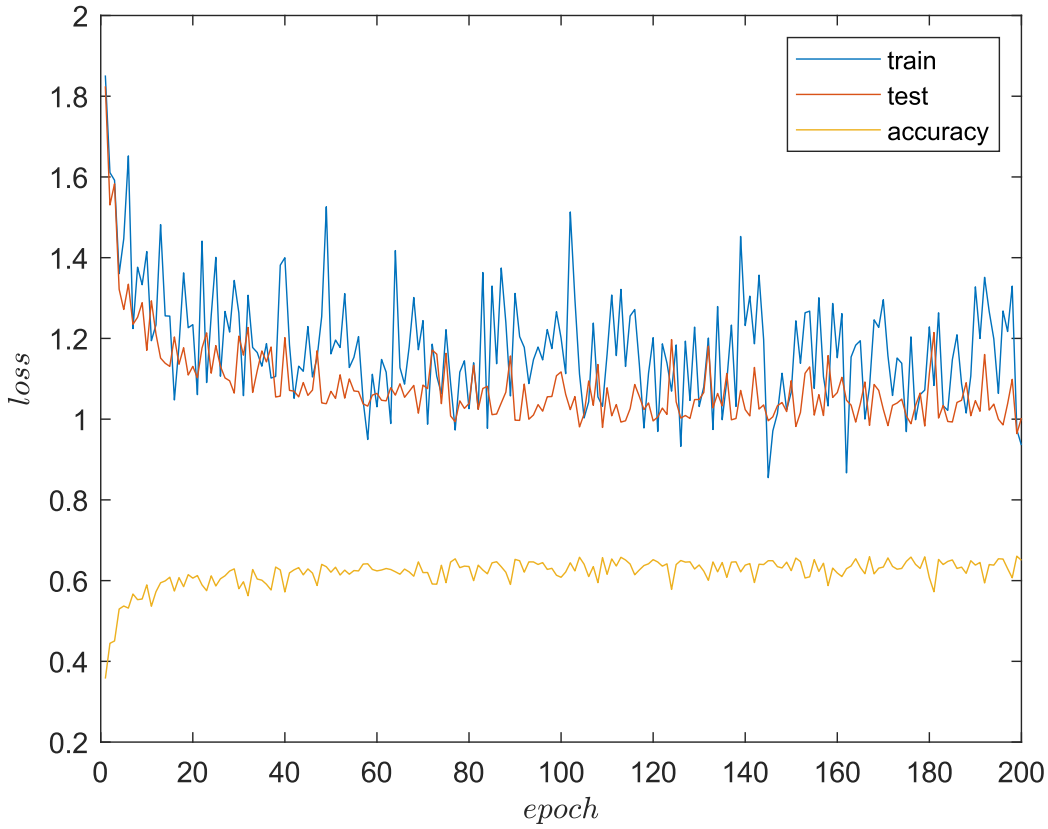
变化非常明显，准确率和loss的下降都非常快，但是最终准确率没有明显提升。

加入BN层后，变化如下：



可以看到，loss下降的速度更快，但是最终的准确率没有很大的提升，因为BN层对较深的网络会有较好的效果。

修改卷积核的大小也会影响训练，将第一个卷积核的size设置为11，padding=5，训练时变化如下：

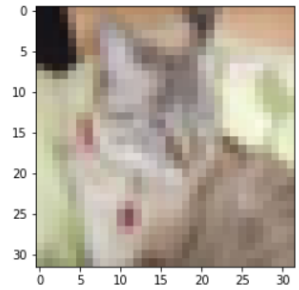


可以看到loss下降的速度也变快了，说明学习图像的大面积的特征有利于提高准确率，但是准确率依然是66%左右，应该是受限于网络的结构特点，比如深度不足。

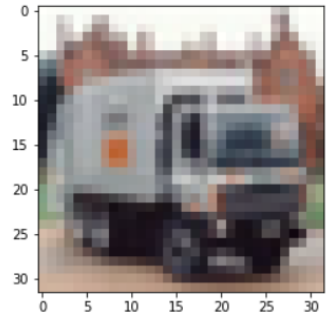
LeNet-5

经过200个epoch的训练，准确率可以达到76%。

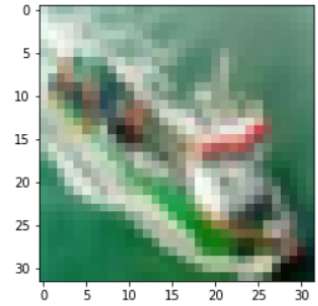
```
>>> from lenet5_train import *
lenet5(
  (l1): Sequential(
    (0): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1), padding=(1, 1))
    (1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (2): ReLU()
  )
  (l2): Sequential(
    (0): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1), padding=(1, 1))
    (1): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (2): ReLU()
  )
  (l3): Sequential(
    (0): Conv2d(16, 120, kernel_size=(3, 3), stride=(1, 1))
  )
  (linear): Sequential(
    (0): Linear(in_features=1920, out_features=120, bias=True)
    (1): ReLU()
    (2): Linear(in_features=120, out_features=84, bias=True)
    (3): ReLU()
    (4): Linear(in_features=84, out_features=10, bias=True)
  )
)
>>> test(model, 1, nn.CrossEntropyLoss(), test_loader)

Test set: Average loss: 0.6845, Accuracy: 7634/10000 (76%)
(tensor(0.7634), tensor(0.6845, device='cuda:0'))
```



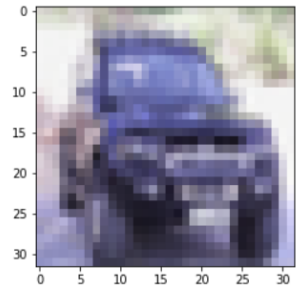
b'cat' , b'cat'



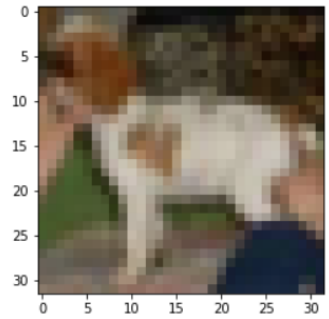
b'truck' , b'truck'



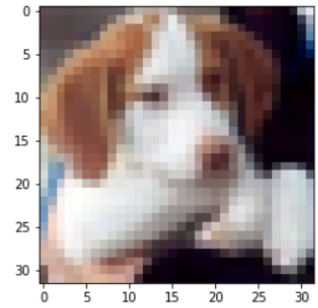
b'ship' , b'ship'



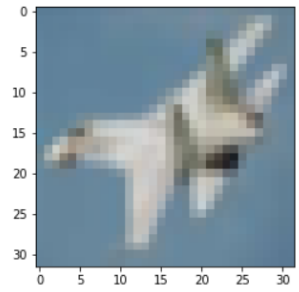
b'automobile' , b'automobile'



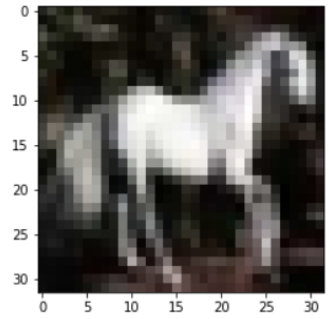
b'deer' , b'dog'



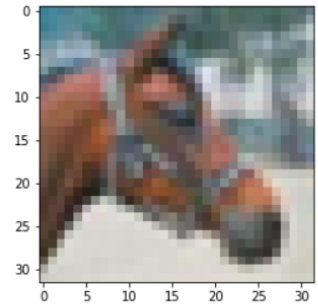
b'dog' , b'dog'



b'dog' , b'airplane'



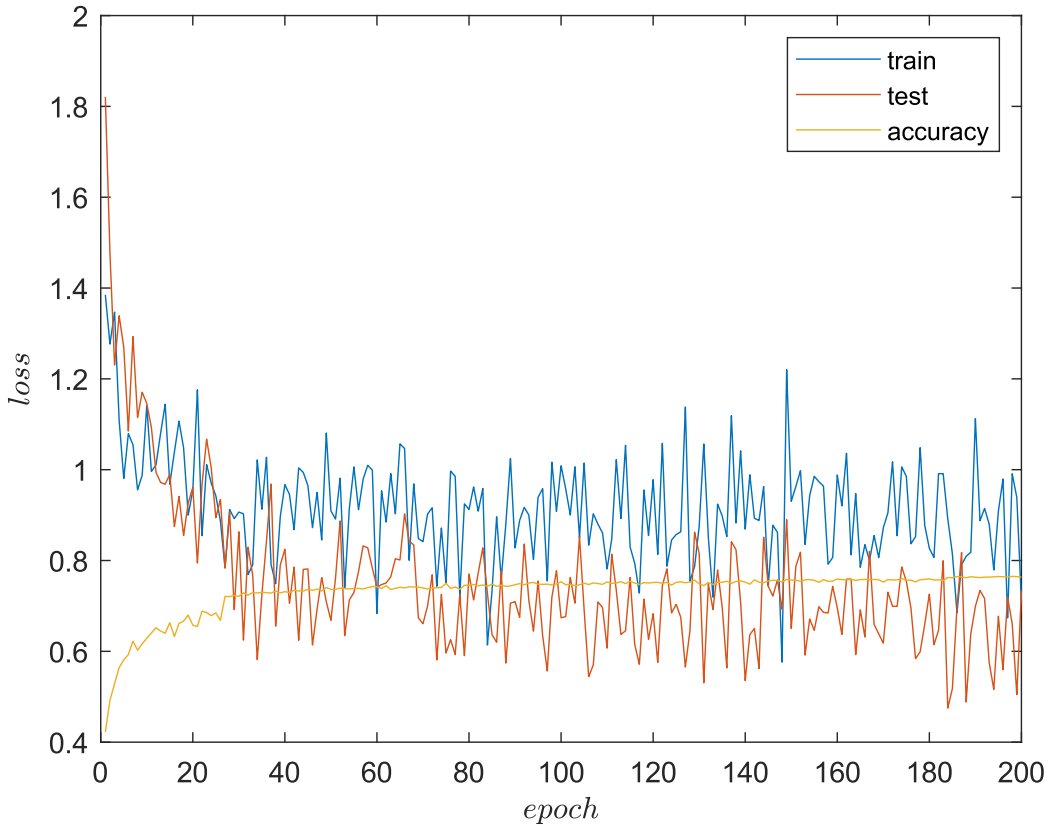
b'horse' , b'horse'



b'horse' , b'horse'

可以看到，LeNet-5准确率虽然提升了，但是也不是特别高。

训练过程中loss和accuracy的变化如下，可以看到，大约40个epoch的时候，就已经有较高的准确率，这是因为使用ReLu函数，能够提高收敛的速度。



ResNet

经过一段时间的训练，准确率可以达到92%。

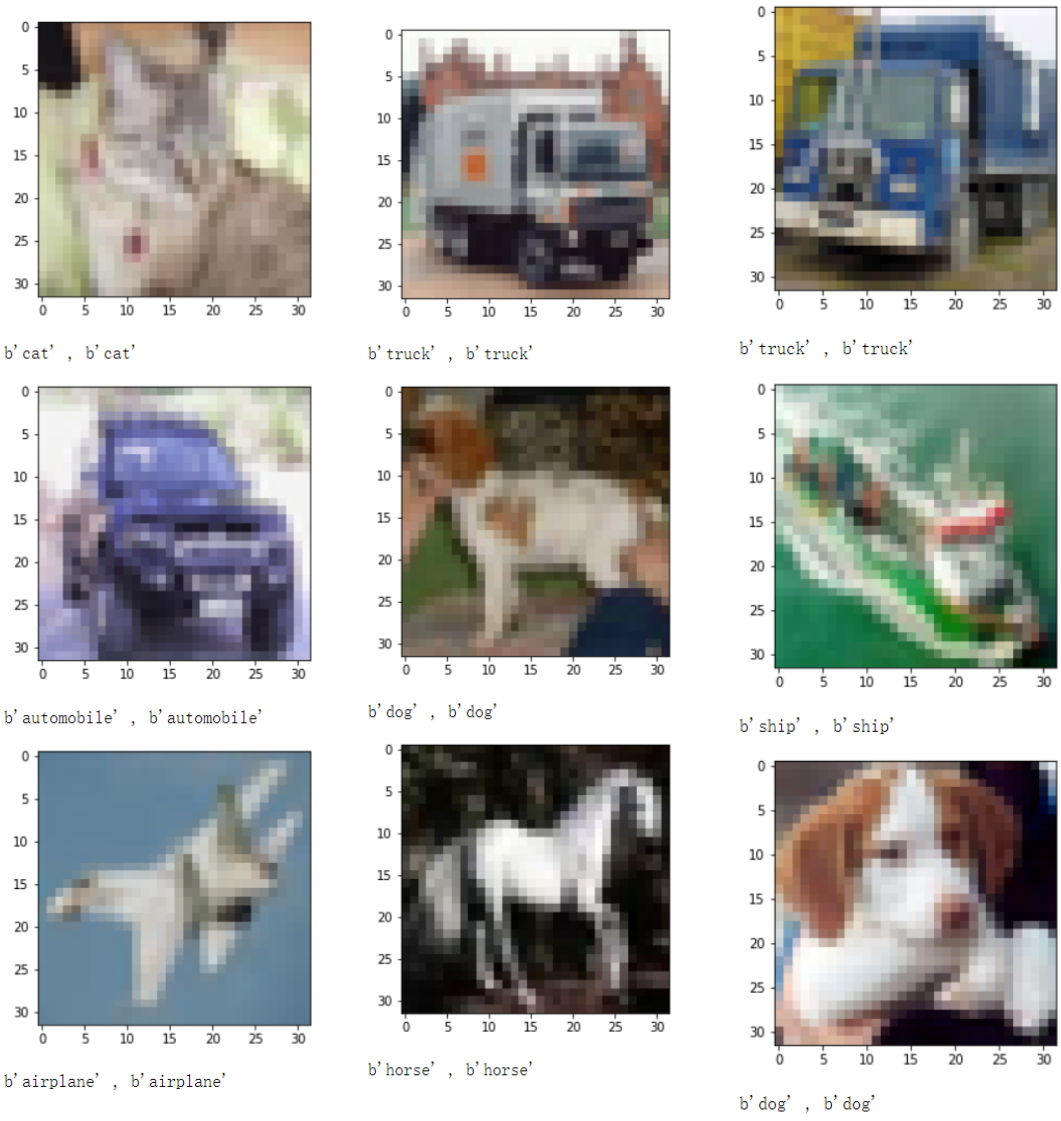
```
>>> from resnet import *
ResNet(
  (conv1): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
  )
  (layer1): Sequential(
    (0): ResidualBlock(
      (left): Sequential(
        (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (shortcut): Sequential()
    )
    (1): ResidualBlock(
      (left): Sequential(
        (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
```

```
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
        (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (shortcut): Sequential()
    )
  )
  (layer2): Sequential(
    (0): ResidualBlock(
      (left): Sequential(
        (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (shortcut): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): ResidualBlock(
      (left): Sequential(
        (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (shortcut): Sequential()
    )
  )
  (layer3): Sequential(
    (0): ResidualBlock(
      (left): Sequential(
        (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=
(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1), bias=False)
        (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
```

```
      (shortcut): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): ResidualBlock(
      (left): Sequential(
        (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (shortcut): Sequential()
    )
  (layer4): Sequential(
    (0): ResidualBlock(
      (left): Sequential(
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (shortcut): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): ResidualBlock(
      (left): Sequential(
        (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (shortcut): Sequential()
    )
  )
  (fc): Linear(in_features=512, out_features=10, bias=True)
)
>>> test(model, 1, criterion, test_loader)

```

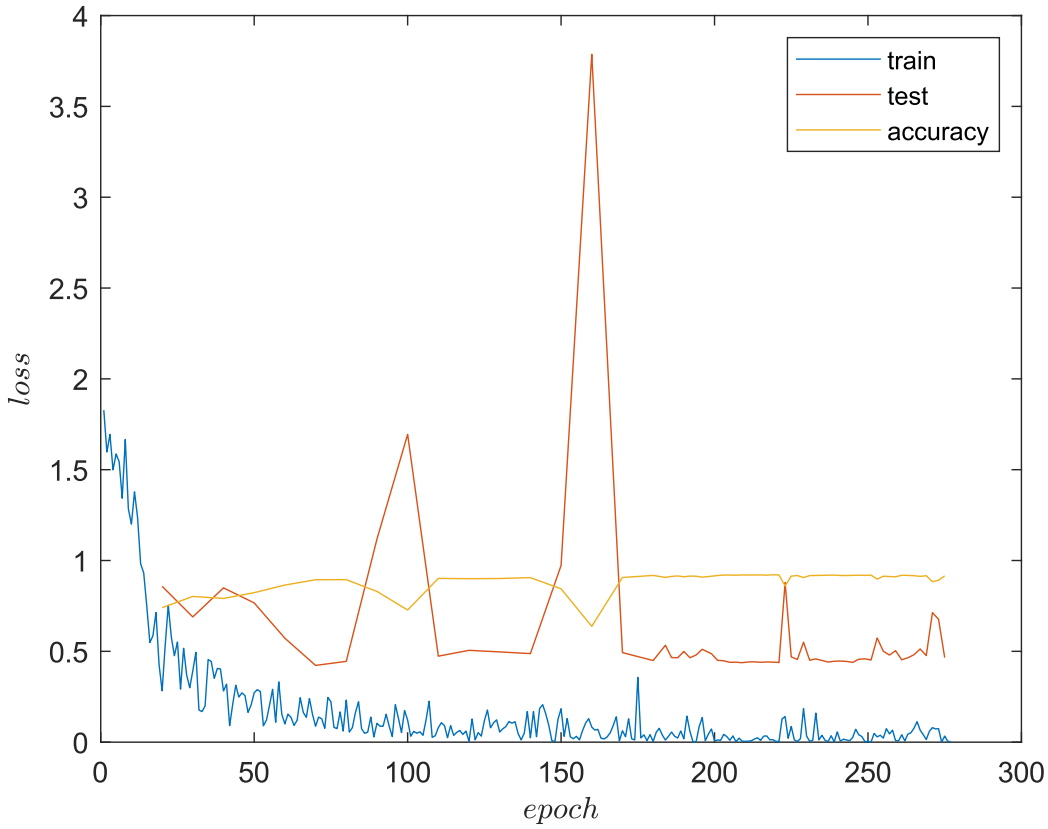
```
Test set: Average loss: 0.4529, Accuracy: 9203/10000 (92%)
(tensor(0.9203), tensor(0.4529, device='cuda:0'))
```

下面是一些例子，



可以看到，LeNet-5预测错误的，ResNet基本没有预测错，准确率是非常高的。因为ResNet的残差层能够很好的发挥深层神经网络的性能。

训练的过程中，loss和accuracy变化如下：



可以看到，准确率在开始的几个epoch就非常高。后面因为学习率比较高出现了反复跳跃。

创新

数据增强

只使用数据集中的原数据训练效果并不是很好，数据增强对数据进行预处理，可以增强其特征便于模型的学习。我使用了随机裁剪，翻转，和张量的归一化，使用的参数参考的是效果比较好的模型。

```
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])
```

使用数据增强前，100个epoch时，ResNet只有60%，LeNet只有30%。而经过数据增强，100个epoch内，ResNet达到90%,LeNet达到66%。因此上面的所有结果均使用了数据增强。

dropout

把一部分的隐层节点值置为0，可以明显地减少过拟合现象。因为这种方式可以减少不同隐层间的相互作用。

```
def forward(self, x):
    out1 = self.l1(x)
    out2 = self.l2(out1)
    out3 = self.l3(out2)
    out3 = F.dropout(out3, training=self.training)
    out3 = out3.view(out3.size(0), -1)
    output = self.linear(out3)
    return output
```

批规范化

统计机器学习中的一个经典假设是“源空间（source domain）和目标空间（target domain）的数据分布（distribution）是一致的”。神经网络的各层都是非线性层，输出的分布显然与各层对应的输入分布不同，而且差异会随着网络深度增大而增大，批规范化能够增大激活值的规模，所以可以**防止“梯度弥散”**。

计算公式如下：

最后一个公式称为仿射，使得output至少可以回到input的状态，使得引入BN不会使得模型更差。

PyTorch中使用`nn.BatchNorm2d`来增加BN层。

其中momentum参数通过下面的公式影响计算：

即根据历史的x进行指数平滑。下面是引入了BN层的lenet5的代码：

```
class lenet5(nn.Module):
    def __init__(self,in_dim,n_class):
        super(lenet8, self).__init__()
        self.l1 = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=6, kernel_size=3, stride=1
                      ),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Sigmoid()
        )
        self.l2 = nn.Sequential(
            nn.Conv2d(in_channels=6, out_channels=16, kernel_size=3,
stride=1
                      ),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.BatchNorm2d(16),
            nn.Sigmoid()
        )
        self.l3 = nn.Sequential(
            nn.Conv2d(in_channels=16, out_channels=120, kernel_size=3)
        )
        self.linear = nn.Sequential(
            nn.Linear(1920, 10)
        )
```

参考

- [1] Source code for torchvision.models.resnet
- [2] arXiv:1512.03385 **[cs.CV]**
- [3] Pytorch实战2：ResNet-18实现Cifar-10图像分类

RNN

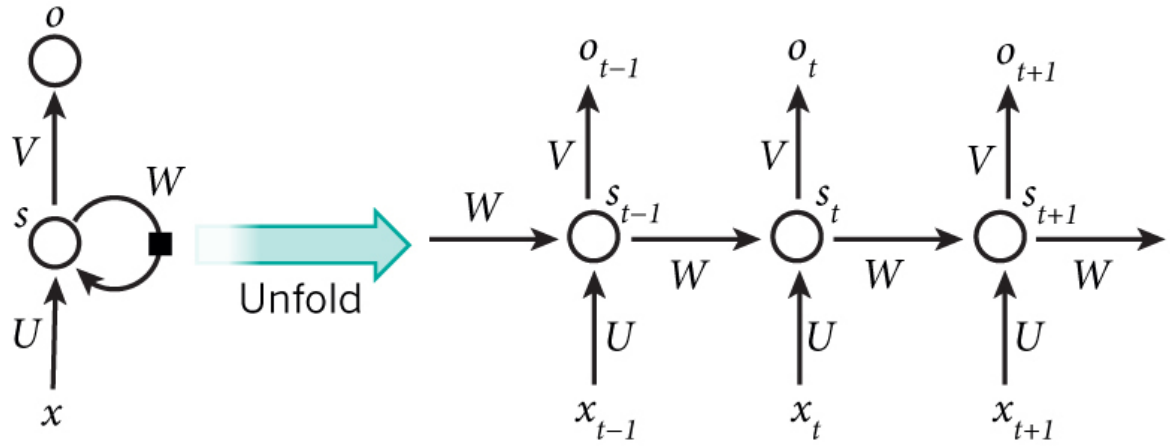
实验原理

RNN

RNN用于处理序列数据，与传统神经网络最大的区别在于它有记忆性，体现在隐藏层的值不仅取决于当前的输入，还取决于上一次隐藏层的值，公式表示为：

a_t为最终的分类结果。σ一般使用Tanh函数。

示意图如下：



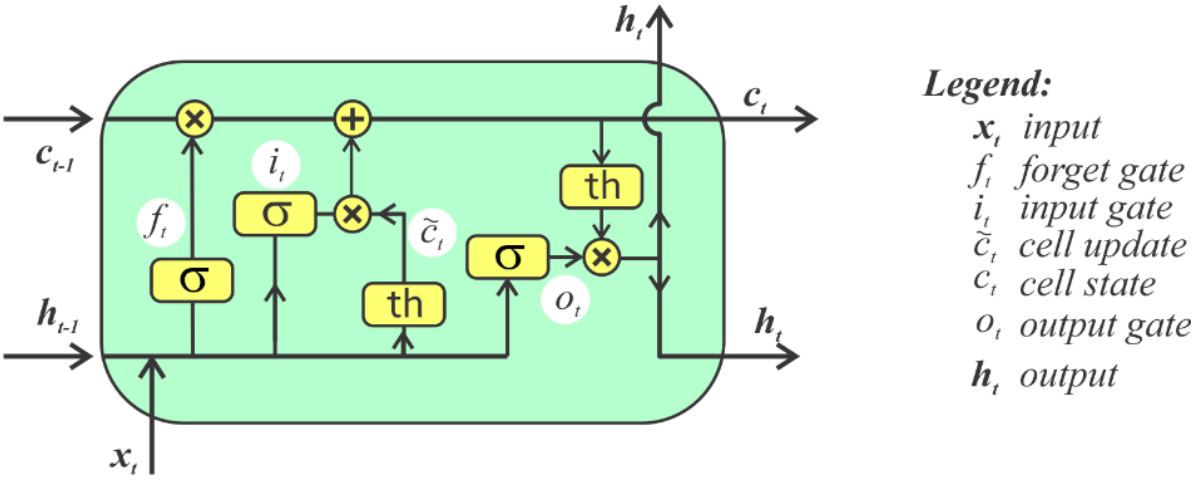
LSTM

由于RNN中有反向传播的连乘导致的梯度消失和梯度爆炸的问题，有人提出了LSTM，

LSTM内部有三个阶段，

1. 忘记阶段，选择性忘记不重要的，记住重要的。
2. 选择记忆阶段
3. 输出阶段

示意图如下：



其中有三个结构：

遗忘门

f_t是通过sigmoid函数得到的结果，为0时会丢弃c_{t-1}中对应的信息，为1时保留。

输入门

输入门也采用sigmoid函数，对经过处理的c_{t-1}进行再处理，公式为：

$\odot$代表Hadamard乘积

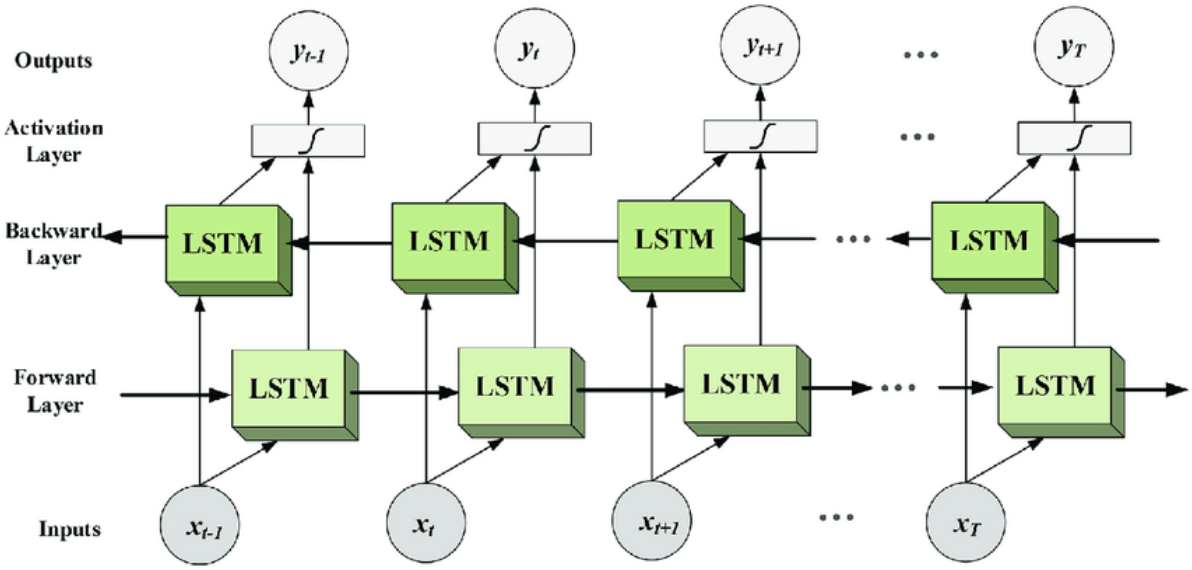
输出门

输出门的计算公式如下：

预测结果为

双向LSTM

BiLSTM是在在LSTM的基础上使用BiRNN的思想搭建的网络。正向计算时，隐藏层的s_t 与 $s_t - 1$ 有关；反向计算时，隐藏层的 s_t 与 $s_t + 1$ 有关。



条件随机场（Conditional Random Field）

CRF 是一个序列化标注算法，通过定义条件概率来描述模型。

首先介绍特征函数，对于自然语言，特征函数有四个自变量，

1. 句子s
2. i，代表第i个单词
3. l_i，表示要评分的标注序列给第i个单词标注的词性
4. l_{i-1}，表示要评分的标注序列给第i-1个单词标注的词性

我们可以定义特征函数的集合，用集合中的特征函数对同一个标注序列的评分综合得到标注序列最终的评分。每个特征函数f_j有一个权重λ_j。只要有一个句子s，标注序列l，就可以进行打分。分数是对每个位置的单词每个特征函数输出的特征值的和。

进行指数化和标准化，就可以得到概率值：

训练CRF模型，一般用最大似然方法作为损失函数。

结合BiLSTM，就可以得到一个BiLSTM+CRF。