Project1

陈乐偲 19307130195

摘要

主要完成了以下任务，

- 基于numpy，仿照pytorch实现了简单的深度网络框架示例clstorch
- 基于clstorch完成了要求中的所有功能，并进行对比试验探究
- 探究了其他的训练神经网络的重要内容，如参数初始化方式、学习率衰减等，并将其运用于模型架构和模型训练中
- 在单隐层网络上，100轮次内可以达到92.8%的准确率，经过优化在双隐层、多隐层、卷积网络其他更复杂的结构上以及2000轮次以内的训练中可以达到95%以上的准确率
- 在理论上经过适当的扩充使用clstorch在此基础上也可以搭建Lenet5、AlexNet、Vgg等深度学习网络

代码说明

仿照pytorch，基于numpy，自己实现了clstorch，并在函数及调用上尽可能像pytorch学习。定义了神经网络中部分常用的函数和模块，并且自己实现了前向和反向传播等过程。

尽可能复现pytorch的效果，在准确率上尽可能接近pytorch。但由于没有使用pytorch中的初始化技术等，并不能达到和pytorch相当的准确率。以及基于numpy的数值计算并不具有pytorch底层代码的数值精度，numpy利用cpu的运算也远不能达到和pytorch相当的计算性能。

当然，在实现细节和封装上，clstorch都不能和pytorch相比。相较于pytorch,clstorch存在以下主要弊端

- 没有实现计算图构建和自动求导机制，需要自己在程序中指定前向和反向传播
- 没有单独实现优化器类，将优化相关的部分直接定义在模型的可学习参数的结构中
- 仅实现简单的功能，功能上远不如pytorch强大
- 在细节优化上，如权重初始化方式等，仅采用了简单的方法

文件说明：

1.dataLoder.py 实现了dataloader类

2.transform.py 实现数据预处理函数

3.clstorch.py 定义一些神经网络模块和损失函数以及其前向和反向传播方法

仿照nn.Module定义自己的Module虚类，以下的class均继承自该类

```
class Module():
    def __init__(self):
        #在前传和反传的过程中记录x，y的值
        self.x = 0
        self.y = 0

    def __call__(self,x):
        #便于隐式调用forward方法
        return self.forward(x)

    def forward(self,x):
        pass

    def backward(self,g):
        pass
```

定义的激活函数

$Sigmoid(x) = \sigma(x) = \frac{1}{1+e^{-x}}$

$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1-\sigma(x))$

$Relu(x) = max(x,0)$

$\frac{\partial Relu(x)}{\partial x} = \#\ 1 \text{ if Relu(x)} > 0 \text{ else } 0$

```
class Sigmoid(Module):
    def forward(self,x):
        self.x = x
        self.y = 1/(1+np.exp(-x))
        return self.y

    def backward(self,g):
        return g * self.y * (1-self.y)

class Normal2one(Module):
    #将向量x归一化
    def forward(self,x):
        self.x = x
        s = x.sum(axis=1).reshape(-1,1)
        self.y = x / s
        return self.y

    def backward(self,g):
        s = x.sum(axis=1).reshape(-1,1)
        return g * (s-self.x) / np.square(s)
```

```
class Relu(Module):
    def forward(self,x):
        self.x = x
        z = np.zeros(x.shape)
        self.y = np.stack([x,z],axis=0).max(axis=0)
        return self.y

    def backward(self,g):
        g[self.y<0] = 0
        return g
```

定义线性函数

$y = xW + b$

$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x} = \frac{\partial L}{\partial y}W^T$

$\frac{\partial L}{\partial W} = \frac{\partial y}{\partial W}\frac{\partial L}{\partial y} = x^T\frac{\partial L}{\partial y}$

$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial y}$

```
class Linear(Module):
    def __init__(self,in_num,out_num,lr=0.01, momentum=0,penalty=0.1):
        #W，b为可学习参数，初始化时将学习率，动量，范数惩罚项均传入，便于参数学习时更新
        super().__init__()

        #使用xavier初始化方法，使得输出的方差与输入的方差保持一致
        self.W = np.random.randn(in_num,out_num) / np.sqrt(in_num + out_num)
        self.b = np.random.randn(1,out_num)

        #记录上一步的参数，动量法更新时使用
        self.W_last = np.array(self.W)
        self.b_last = np.array(self.b)

        self.lr = lr
        self.momentum = momentum
        self.penalty = penalty

    def forward(self,x):
        self.x = x
        self.y = np.matmul(x,self.W) + self.b
        return self.y

    def backward(self,g):
        dx = np.matmul(g, self.W.T)
        dW = np.matmul(self.x.T, g)
        db = g

        #二范数惩罚等价于权重衰减
        dW += self.penalty * self.W / np.linalg.norm(self.W,"fro")
        db += self.penalty * self.b / np.linalg.norm(self.b,"fro")

        self.W = self.W  - self.lr * dW + self.momentum * (self.W - self.W_last)
        self.b = self.b  - self.lr * db + self.momentum * (self.b - self.b_last)
        self.W_last = np.array(self.W)
        self.b_last = np.array(self.b)
```

```
        return dx
```

参数初始化的方法定义非常关键，采用标准正态的初始化方法，模型难以训练，而采用xavier方法初始化，损失在很少的轮次内迅速下降且准确率提高明显。

由于在损失函数中加入$\lambda\|W\|_2$这样一项，由$\frac{\partial Loss}{\partial W}=2\lambda W$,也即相当于在权重更新时加入$penalty=2\lambda$ 为比例的权重衰减，所以可将范数惩罚定义至参数更新的过程。

定义损失函数，继承自虚类LossModule

$L=\|x-y\|_2$

$\frac{\partial L}{\partial x}=\frac{x}{\|x-y\|_2}$

```
class LossModule:
    def __init__(self):
        super().__init__()
        self.l = 0
        self.x = 0
        self.y = 0
    def forloss(self,x,y):
        pass
    def backloss(self):
        pass

class MSELoss(LossModule):
    def forloss(self,x,y):
        self.x = x
        self.y = y
        self.l = np.linalg.norm(x-y,"fro")
        return self.l
    def backloss(self):
        return (self.x - self.y) / self.l
```

定义transform方法

```
def oneHot(y):
    #将标签转换为独热向量
    a = np.zeros((y.shape[0],10))
    for i in range(len(y)):
        a[i,y[i,0]] = 1
    return a

def normalize(x):
    #将图片的灰度值归一至0-255之间，并且标准化
    x = x / 255
    x = x - np.mean(x, axis=1).reshape(-1,1)
    x / np.std(x, axis=1).reshape(-1,1)
    return x
```

实验证明，是否进行归一化对模型学习的影响很大，将灰度值归一至0-255之间后，模型变得很好学习。

定义自己的dataloader，仿照pytorch中的Dataloader类。

```
class Dataloader:
    def __init__(self, x, y, batch_size, shuffle=True, transform=None):
        if transform is not None:
            self.x, self.y = transform(x,y)
        else:
            self.x, self.y = x,y

        self.batch_size = batch_size
        self.size = len(x) // batch_size

        #将数据随机打乱
        if shuffle is True:
            perm = np.array(range(self.size))
            x = x[perm]
            y = y[perm]

    #定义__getitem__和__len__方法,便于迭代
    def __getitem__(self,index):
        B = self.batch_size
        if B*(index+1) >= len(self.x):
            return self.x[B*index:], self.y[B*index:]
        else:
            return self.x[B*index:B*(index+1)], self.y[B*index:B*(index+1)]

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

定义了上述函数后，便可以开始在手写数据集上训练自己的神经网络了。

1.网络结构

改变网络结构

先定义LinearSequence类, 传入一个列表表示隐层的参数，中间使用Relu激活。

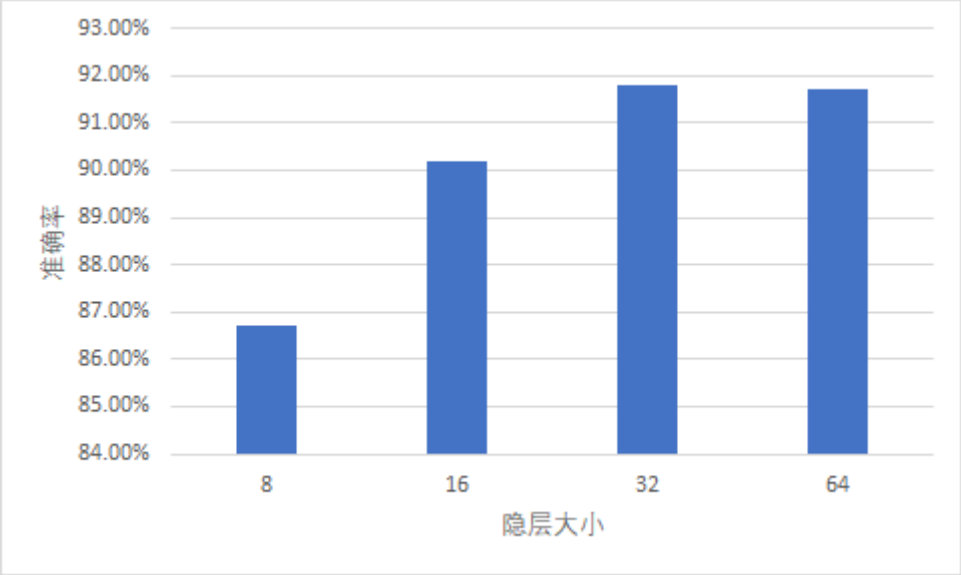
```
class LinearSequnce(Module):
    def __init__(self,lst,lr=0.001,momentum=0,penalty=0):
        super().__init__()
        self.lst = lst
        self.layers = []

        in_num = lst[0]
        for i in range(1,len(lst)):
            out_num = lst[i]
            li = Linear(in_num,out_num,lr,momentum,penalty)
            ri = Relu()
            self.layers.append(li)
            self.layers.append(ri)
            in_num = out_num

    def forward(self,x):
        for layer in self.layers:
            x = layer.forward(x)
        return x

    def backward(self,g):
        for layer in self.layers[::-1]:
            g = layer.backward(g)
        return g
```

定义网络并训练，使用单层全连接层，MSE损失作为损失函数

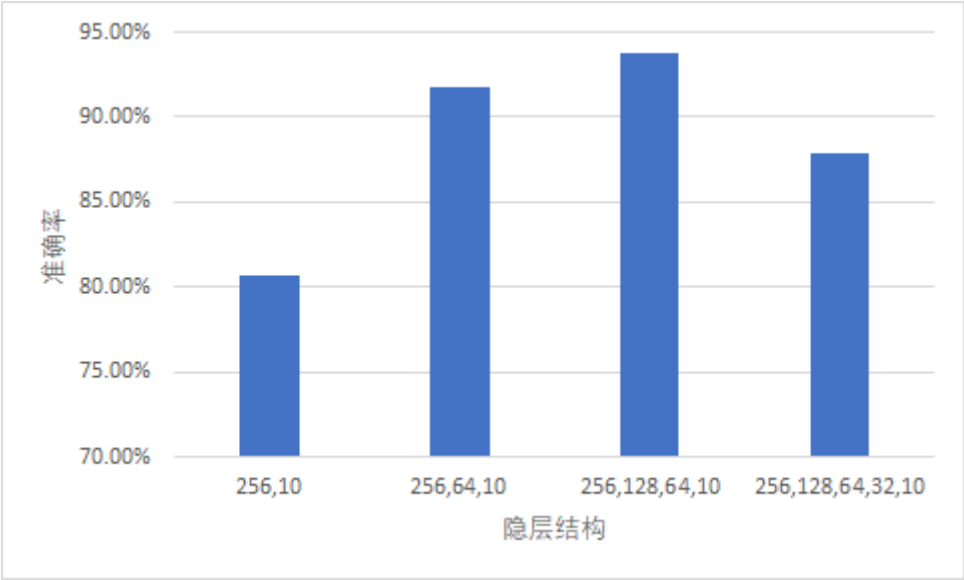
```
data = scipy.io.loadmat("./digits.mat")
Xtest,ytest = np.array(data["Xtest"]), np.array(data["ytest"]-1)
X,y = np.array(data["X"]), np.array(data["y"]-1)
Xvalid, yvalid = np.array(data["Xvalid"]), np.array(data["yvalid"]-1)
dataLoader = Dataloader(X,y,batch_size=32,transform=myTransform)
testLoader = Dataloader(Xtest,ytest,batch_size=32,transform=myTransform)

class Net():
    def __init__(self):
        self.fc = Linear(256,10,lr=0.01,momentum=0)
        self.criterion = MSELoss()

    def forward(self,x):
        x = self.fc(x)
        return x

    def backward(self,g):
        g = self.fc.backward(g)

    def train(self, dataLoader):
        for e in range(200):
            loss = 0
            for data in dataLoader:
                x,y = data
                if len(x) == 0:
                   break
                #随机梯度下降，在一批量数据中随机采样，计算梯度并反向传播
```

```
                sample = np.random.randint(len(x))
                x,y = x[sample].reshape(1,-1), y[sample].reshape(1,-1)

                pred = self.forward(x)
                loss += self.criterion.forloss(pred,y)

                g = self.criterion.backloss()
                _ = self.backward(g)

            loss /= len(dataLoader)
            if e%100 == 0:
                print("loss", loss)

    def test(self,dataLoader):
        correct = 0
        total = 0
        for data in dataLoader:
            x,y = data
            if len(x) == 0:
                break
            prob = self.forward(x)
            pred = np.argmax(prob,axis=1)
            label = np.argmax(y,axis=1)
            correct += np.equal(pred,label).sum()
            total += len(pred)
        accucacy = correct / total
        print("accucacy", accucacy)

model = Net()
model.train(dataLoader)
model.test(testLoader)
```

控制其他条件相同的情况下，探究隐层结构对结果的影响。



隐层大小对结果的影响



隐层数目对结果的影响

由实验结果可见，增加隐层参数量和增加网络深度都可以使得模型准确率提升。

神经网络越复杂，网络的学习能力越强，但当模型过于复杂时，由于参数量的变多，虽然理论上模型的学习能力更强了，但由于调节难度也随之增加，可能导致结果反而不好，同时过于复杂的模型也容易产生过拟合的现象。

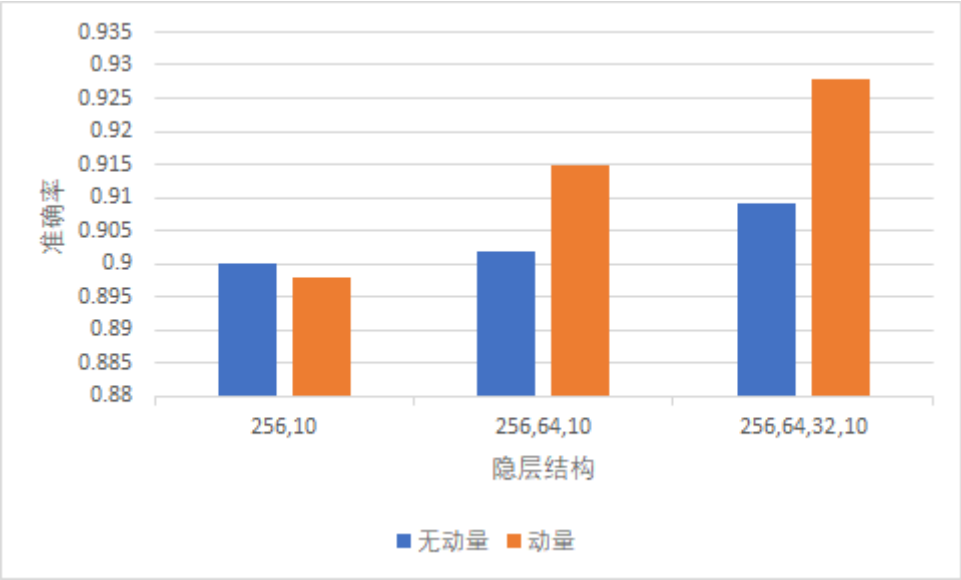
由实验结果也可见，单纯增加网络深度，或者单纯增加隐层参数量，对准确率的提升效果不如在增加深度的同时藏家参数量。

2.动量

带动量的随机梯度下降算法

在参数更新过程中实现

```
self.W = self.W  - self.lr * dW + self.momentum * (self.W - self.W_last)
self.b = self.b  - self.lr * db + self.momentum * (self.b - self.b_last)
self.W_last = np.array(self.W)
self.b_last = np.array(self.b)
```



动量对结果的影响

从实验中可以看出，当隐层结构简单的时候，动量对结果影响不大。但当结构复杂的时候，使用了动量法后准确率的确提升了。而且绘制出loss曲线可以发现，使用动量法之后loss的波动性，也即方差有所减少。

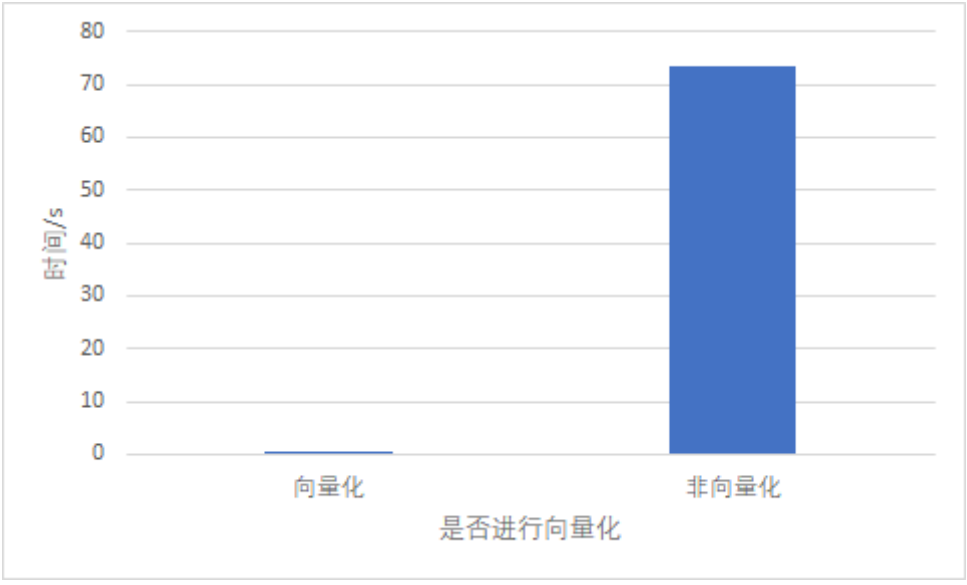
给出的解释是，当隐层结构更复杂的时候，优化函数更加复杂，存在更多的局部极小值等复杂情况，而采用了动量法可以减轻局部极小值对优化的影响。另一个可能的解释是，当隐层结构更加复杂的时候，梯度下降算法需要更长的时间收敛，而动量法吸收了前一步的下降方向，可以起到加速收敛的效果。

3.向量化

向量化函数定义

之前的所有运算都使用矩阵运算，具体可以详见代码部分。

用numpy编写简单的矩阵运算函数，对比是否向量化的运行时间。可见，向量化对性能的提高非常明显。使用向量化取代for循环操作可以更好地利用计算机地并行特性，提高运算效率。



向量化对执行相同矩阵运算的影响

4.正则化

正则化

范数惩罚等价于权重衰减，推导过程可详见上一份报告。

增加范数惩罚，在参数更新过程中实现

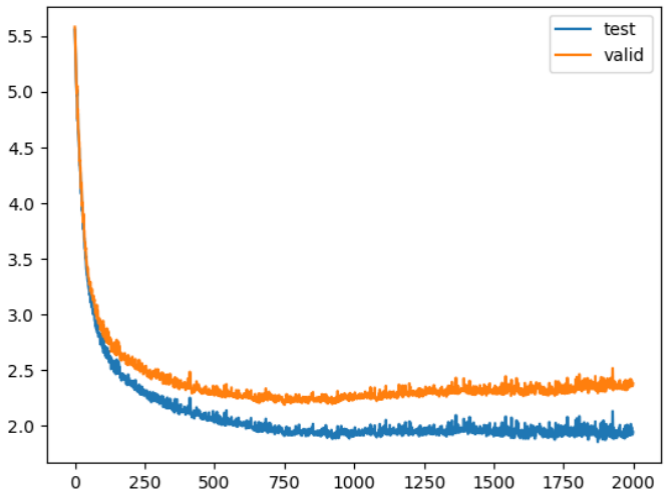
```
dW += self.penalty * self.W / np.linalg.norm(self.W,"fro")
db += self.penalty * self.b / np.linalg.norm(self.b,"fro")
```

使用提前终止策略，当$\frac{err-min_err}{min_er} > treshhold$时终止训练

```
err = 1 - self.eval(validLoader)
if err < min_error:
    min_error = err
elif (err-min_error) / min_error > threshhold:
    print("stop at epoch ",e)
    break
```

首先探究过拟合现象

当训练轮次为500轮时，测试集准确率有92.8%。但当轮次为2000轮的时候，准确率仅有90.9%，猜想是发生了过拟合现象。

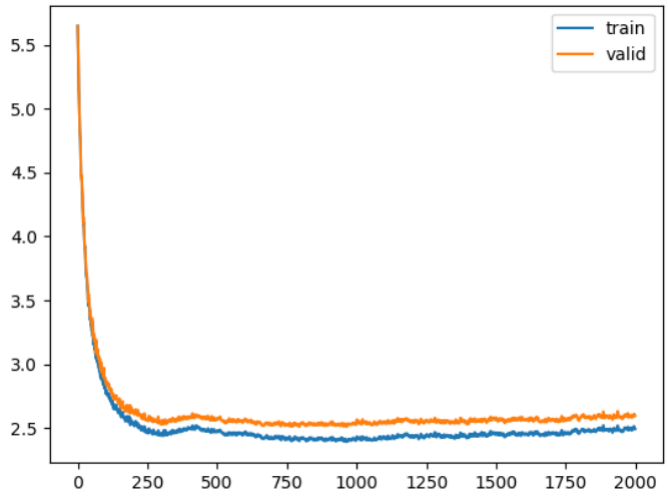


训练集和验证集上的损失曲线

绘制出训练集和验证集上的损失曲线后发现，的确在800轮次左右，验证集上的损失有上升的趋势。

使用提前终止策略，在783轮次时模型终止了训练，的确达到了想要的减少过拟合的效果。

使用范数惩罚策略，画出损失曲线可以发现，训练集损失和验证集损失之间的gap的确变小了，范数惩罚的确起到了很好的防止过拟合的作用，而且附带的效果是损失的波动也同时减小了。



5.softmax

定义softmax函数

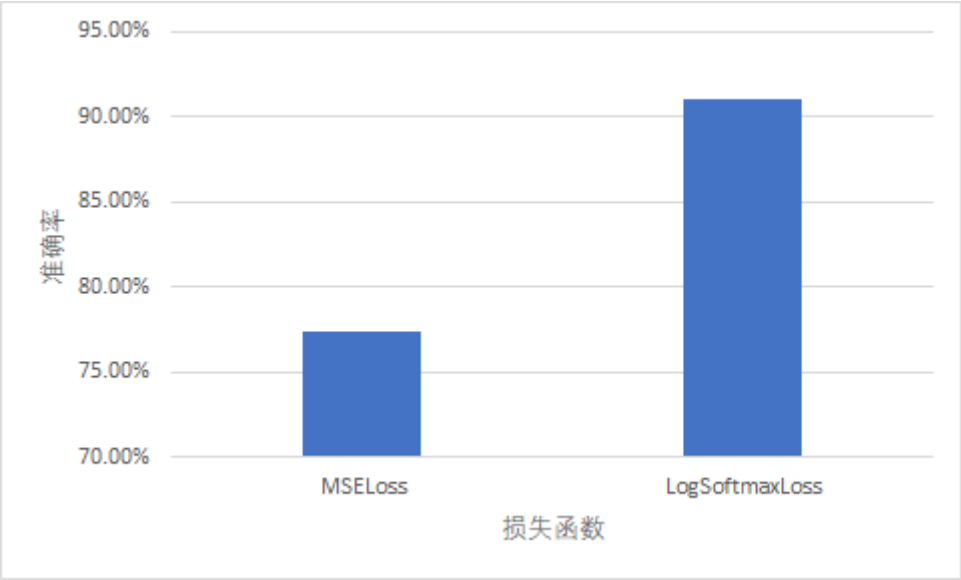
将softmax经过负对数似然后的损失定义为LogSoftmaxLoss

$s = Softxmax(x) = [e^{x_1}, e^{x_2}, \dots, e^{x_n}] / \sum_{i=1}^{n} e^{x_i}$

$L_j = -\lg(e^{x_j} / \sum_{i=1}^{n} e^{x_i}) = -x_j + \log(\sum_{i=1}^{n} e^{x_i})$

$\frac{\partial L_j}{\partial x_i} = \frac{\partial L_j}{\partial s_i}\frac{\partial s_i}{\partial x_i} = \begin{cases} s_j - 1 & i = j \\ s_i & i \neq j \end{cases}$

```
class LogSoftmaxLoss(Module):
    def __init__(self):
        super().__init__()
        self.s = 0

    def forloss(self,x,y):
        self.x, self.y = x,y
        s = np.exp(x)
        s = s / np.sum(s)
        l = np.multiply(-np.log(s),y).sum()
        self.s, self.l = s,l
        return self.l

    def backloss(self):
        return self.s - self.y
```



两种损失函数的对比

使用最简单的单隐层，训练100轮次，使用Softmax之后的准确率明显提升。

给出的解释是，使用softmax后不仅模型输出的值可以被解释为概率，由于其将其归一至0-1之间，使得线性函数不需要学习到归一化这一性质，使得模型更加易于训练。

6.bias

$Wx+b=[x,1]\cdot[W,b]$

```
#定义没有偏置的线性层
class LinaerWithoutBias(Module):
    def __init__(self,in_num,out_num,lr=0.01, momentum=0,penalty=0.1):
        super().__init__()
        self.W = np.random.randn(in_num,out_num) / np.sqrt(in_num)
        self.W_last = np.array(self.W)

        self.lr = lr
        self.momentum = momentum
        self.penalty = penalty
```

```
#网络结构也做相应的修改
class Net():
    def __init__(self):
        self.fc = LinaerWithoutBias(257,10,lr=0.01,momentum=0)
        self.criterion = MSELoss()

    def forward(self,x):
        #在x后添加一列全1向量
        t = np.ones((x.shape[0],1))
        x = np.concatenate([x,t],axis=1)
        x = self.fc(x)
        return x
```

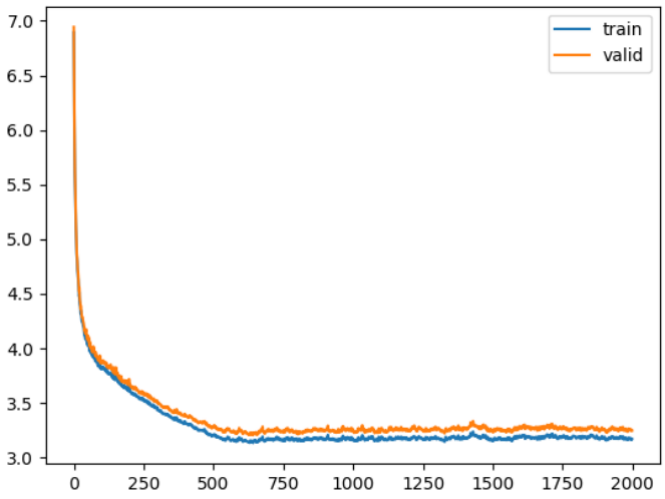
更改后模型与原来表现基本一致。

7.dropout

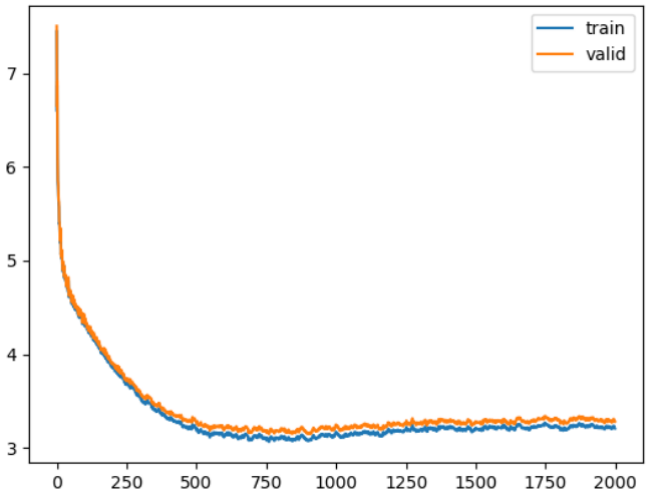
定义dropout方法

```
class Dropout(Module):
    def __init__(self,p=0.5):
        super().__init__()
        self.mask = 0
        self.p = p

    #在训练和测试过程中有不同的表现
    def fordrop(self,x,traning):
        if traning is True:
            #训练时随机生成掩码，并除以概率p
            self.mask = (np.random.random(x.shape) < self.p) / self.p
            self.y = x * self.mask
        else:
            self.y = x
        return self.y

    def backward(self,g):
        return self.mask * g
```

使用dropout之后相当于进行了正则化，也相当于一种集成模型，学习效果得到增强。

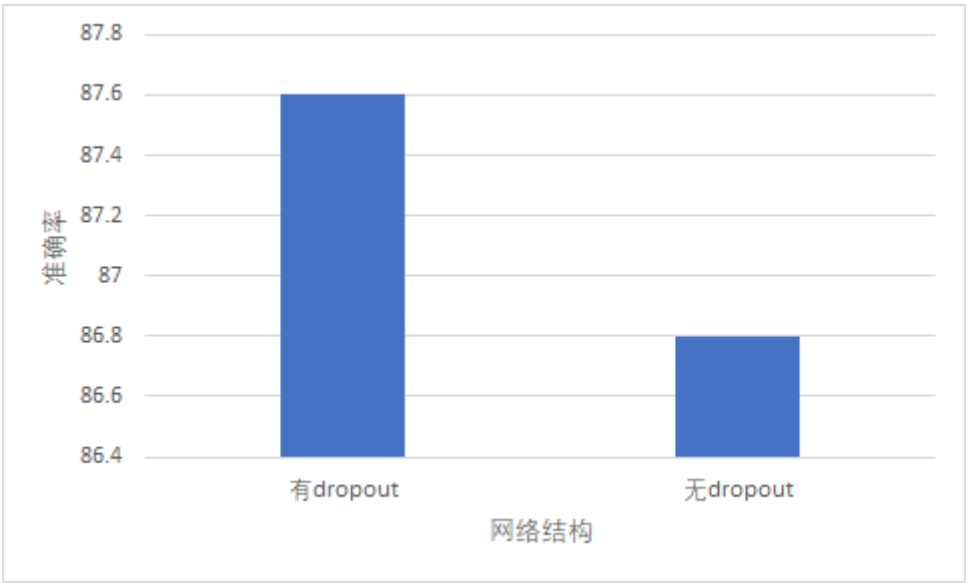


未使用dropout的损失曲线



使用了dropout的损失曲线

采用简单的双隐层结构进行实验，使用dropout后过拟合的现象的确得到了减轻。从测试集的准确率也可以看出。



dropout对结果的影响

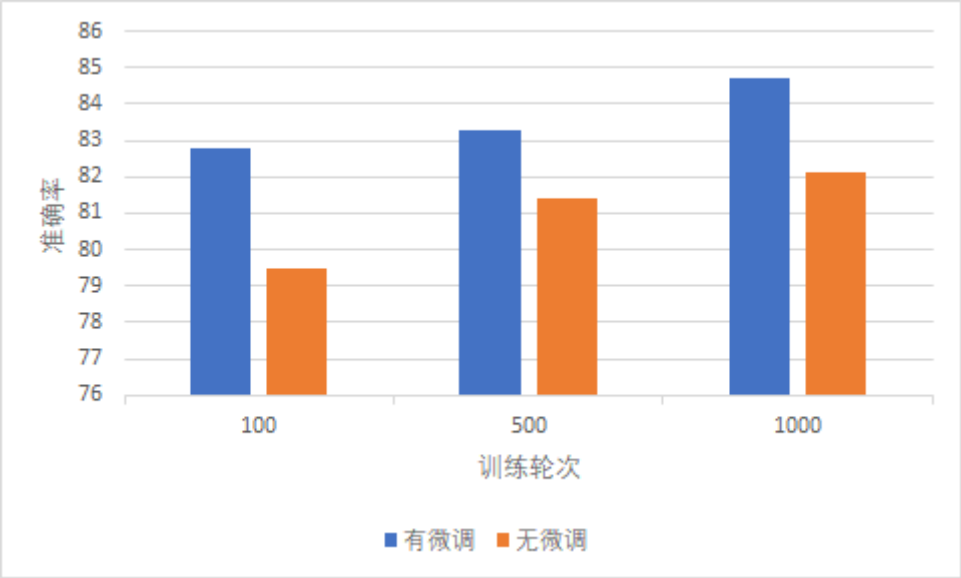
8.Finetune

对模型的最后一层微调

使用最小二乘法，使得最后一层的输出尽可能接近真实标签

```
def finetune(self,dataLoader):
```

```
        xlst =  []
        ylst =  []
        for i, data in enumerate(dataLoader):
            x,y = data
            if len(x) == 0:
                break
            if i==len(dataLoader)-1:
                break
            x = self.fc1(x)
            x = self.activate1(x)
            xlst.append(x)
            ylst.append(y)

        x = np.concatenate(xlst,axis=0)
        y = np.concatenate(ylst,axis=0)
        W = self.fc2.W
        b = self.fc2.b

        inv = np.linalg.inv(np.matmul(x.T,x))
        z = np.matmul(x.T,y-b)
        W = np.matmul(inv,z)
```



是否微调对结果的影响

从实验结果中可以看出，经过微调，模型的准确率得到了提升。

9.数据扩增

数据扩增

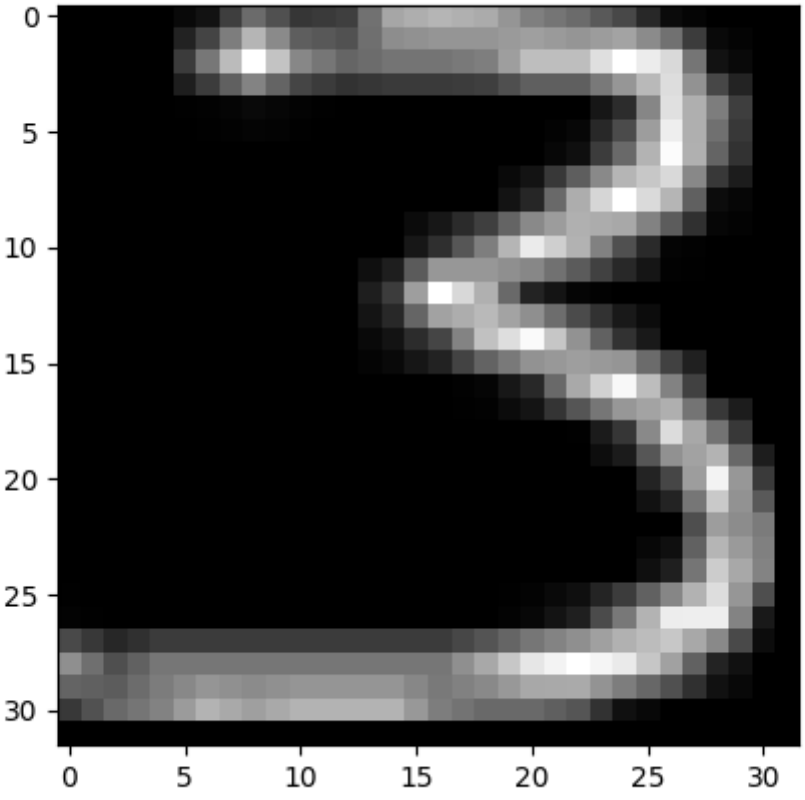
定义在transform.py中，首先实现双线性插值算法，在其基础上实现图像的平移、旋转、改变大小等操作。

```
def biInterpolate(x,i,j):
```

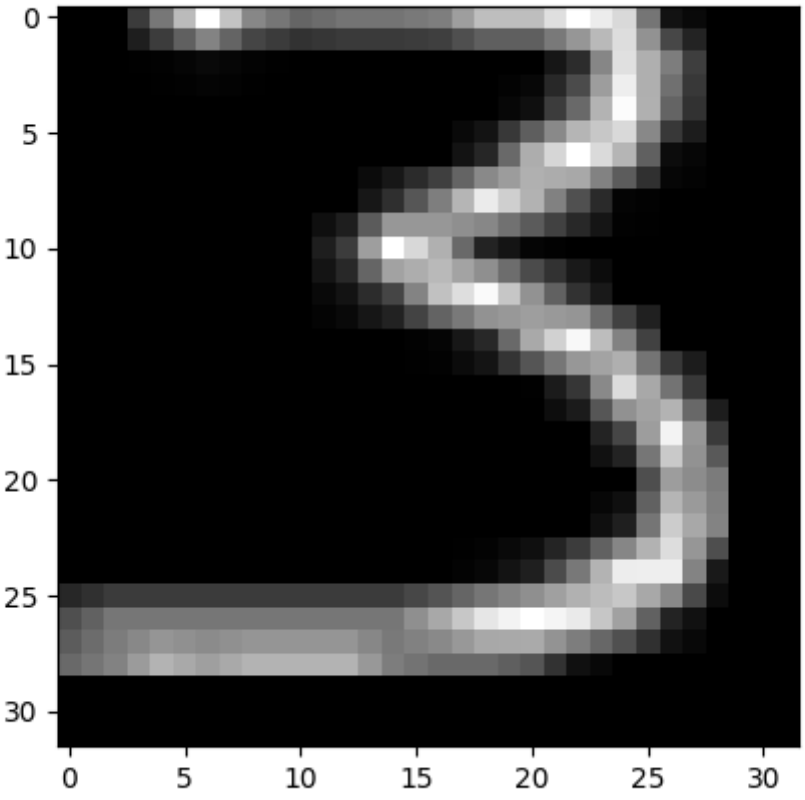
```
    rows,cols = x.shape
    up = int(np.floor(i))
    down = int(np.ceil(i))
    left = int(np.floor(j))
    right = int(np.ceil(j))
    if up<0 or left<0 or down>=rows or right>=cols:
        return 0
    u,v = i-up, j-left
    y = u*v*x[up,left] + u*(1-v)*x[up,right] + (1-u)*v*x[down,left] + (1-u)*(1-v)*x[down,right]
    return y

def shift(x,dx,dy):
    #平移
    y = np.empty(x.shape)
    n,m = x.shape
    for i in range(n):
        for j in range(m):
            y[i,j] = biInterpolate(x,i+dx,j+dy)
    return y

def resize(x,NH,NW):
    #将大小调整为NHxNW
    H,W = x.shape
    y = np.empty(x.shape)
    ch,cw = H/2, W/2
    for h in range(H):
        for w in range(W):
            preh = ch + H / NH * (h-ch)
            prew = cw + W / NW * (w-cw)
            y[h,w] = biInterpolate(x,preh,prew)
    return y

def rotate(x,angle):
    #旋转angle度
    H,W = x.shape
    y = np.empty(x.shape)
    ch,cw = H/2, W/2
    for h in range(H):
        for w in range(W):
            if w==cw:
                a = np.pi /2 if h<ch else np.pi * 3/2
            else:
                a = np.arctan((ch-h)/(w-cw))
                if w < cw:
                    a = a + np.pi
            r = np.sqrt((h-ch)**2 + (w-cw)**2)
            na = a + angle
            preh = ch - r * np.sin(na)
            prew = cw + r * np.cos(na)
            y[h,w] = biInterpolate(x,preh,prew)
    return y
```

对MNIST中的数字3进行数据扩增操作。

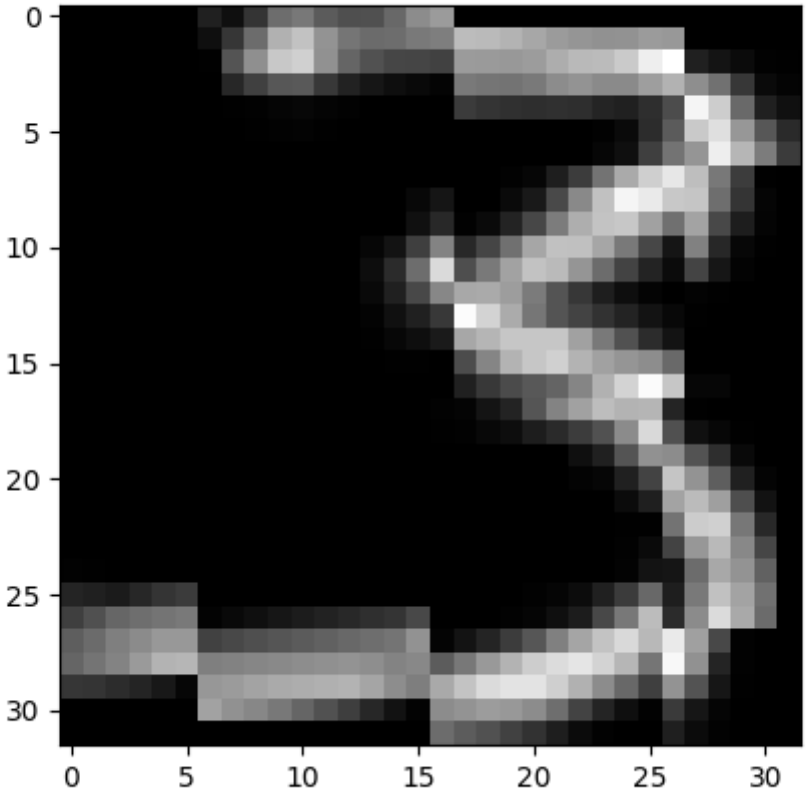
由于原始16x16大小的图片效果看起来不明显，先使用双线性插值将原图的分辨率放大2倍，再进行数据扩增。



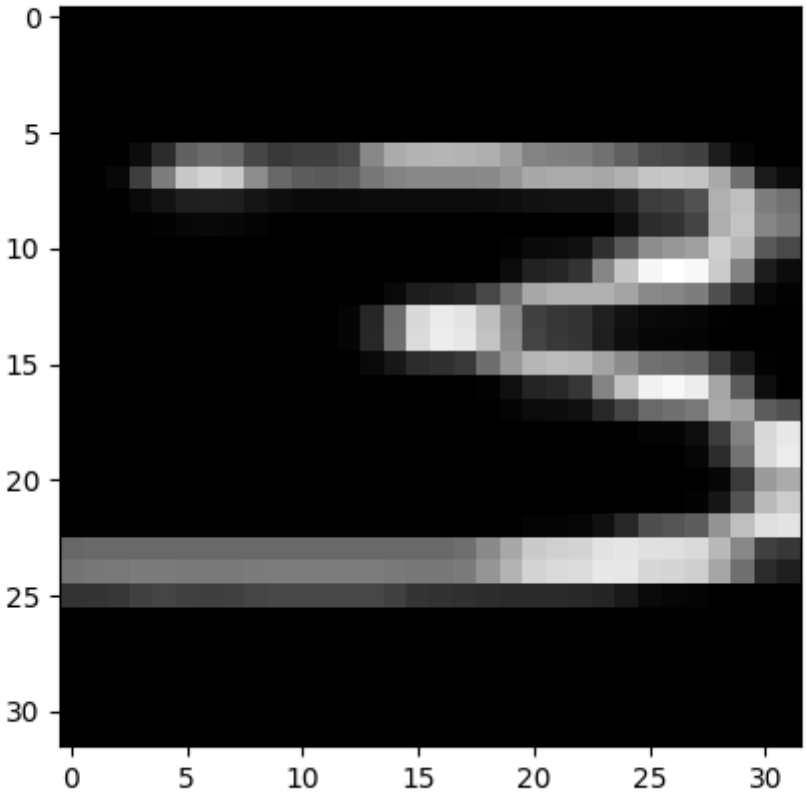
数据扩增前的图像



平移后的图片



旋转后的图片



拉伸后的图片

使用了数据扩增后，在相同条件下，模型在测试集上的准确率并没有得到很大的提升，但反而可以发现在验证集上的准确率有所下降。猜想模型在测试集上准确率没有提升的原因可能有，

- MNIST数据集数据量足够，数据扩增并不能起到很好的效果
- 本身手写数据集的数据就有足够的多样性
- 数据扩增对低分辨率图片效果不佳

而验证集上准确率下降来自于数据扩增后数据更加复杂了，相同的模型在训练相同的轮次下更难以学到数据的特征。

10.卷积

卷积

定义于clstorch.py文件中

首先实现零填充和逐点卷积操作

```
def padding(x,pad):
    y = np.pad(x,((0,0),(pad,pad),(pad,pad),(0,0)),"constant" )
    return y

def point_conv(a, W, b):
    z = np.sum(np.sum(a * W) + b)
    return z
```

定义Conv类

卷积的前向传播：

$y = conv(x, W, b, K, S, P)$

$p = padding(x, P)$

$s_{h,w,c} = window(p, h, w) = p_{hS:hS+K,wS:wS+K,c}$

$y_{h,w,c} = s_{h,w,c} \otimes W_c + b_c$

卷积的反向传播

$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial y} \otimes x$

$\frac{\partial L}{\partial x} = rot180(W) \otimes \frac{\partial L}{\partial y}$

反向传播,可以用上述公式计算，但在代码中为了方便采取模拟进行一遍前向传播的过程同时计算梯度

由于W,b在每次卷积中共享权重，最终计算梯度dW,db可以进行权重标准化，即除以卷积次数，使得参数更好学习

```
#首先定义两个辅助函数
def padding(x,pad):
    y = np.pad(x,((0,0),(pad,pad),(pad,pad),(0,0)))
    return y

def point_conv(a, W, b):
    z = np.sum(np.sum(a * W) + b)
    return z

#卷积类
#REF: [1]
class Conv(Module):
    def __init__(self,Cin,Cout,K,P,S,lr=0.01,momentum=0):
        super().__init__()
        self.Cin = Cin
        self.Cout = Cout
        self.P = P
        self.K = K
        self.S = S
        self.lr = lr
        self.momentum = momentum

        self.W = np.random.randn(K,K,Cin,Cout) / np.sqrt(K*K*Cin)
        self.b = np.random.randn(1,1,1,Cout)
        self.W_last = np.array(self.W)
        self.b_last = np.array(self.b)

    def forward(self,x):
        self.x = x
        B, H, W, Cin = x.shape
        K, K, Cin, Cout = self.W.shape
        P,K,S = self.P, self.K, self.S
        Hout =  (H - K + 2*P) // S + 1
        Wout =  (W - K + 2*P) // S + 1

        y = np.zeros((B,Hout,Wout,Cout))
        xpad = padding(x,P)

        for i in range(B):
            for h in range(Hout):
                for w in range(Wout):
                    for c in range(Cout):
                        up = h * S
                        down = up + K
                        left = w * S
                        right = left + K
                        xwindow = np.expand_dims(xpad[i, up:down, left:right, :],0)
                        y[i, h, w, c] = point_conv(xwindow, self.W[:, : , :, c],
self.b[0,0,0,c])
        self.y = y
        return self.y

    def backward(self,dy):
        B, H, W, Cin = self.x.shape
        P,K,S = self.P, self.K, self.S
        B,Hout,Wout,Cout = dy.shape
        xpad = padding(self.x, P)
```

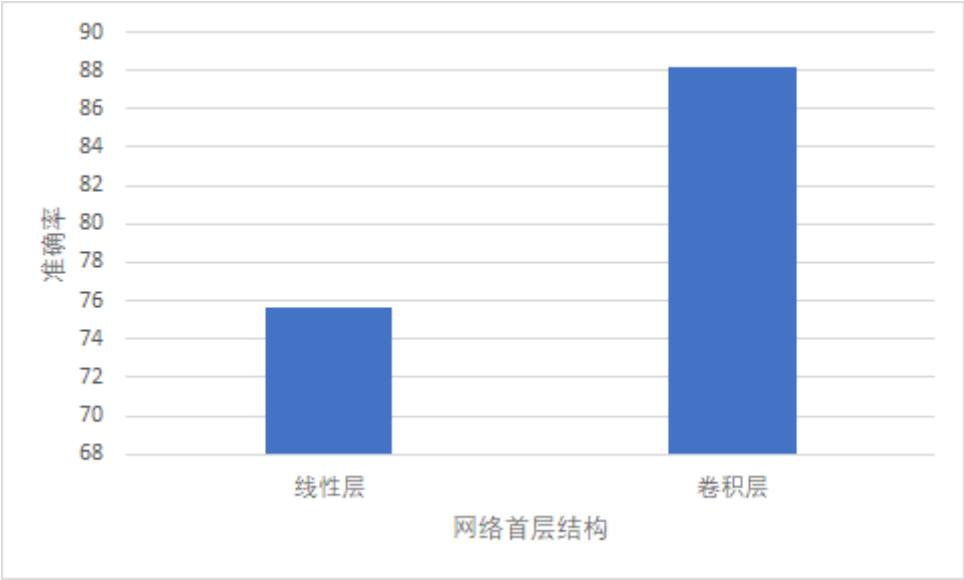
```
        dx = np.zeros(self.x.shape)
        dW = np.zeros(self.W.shape)
        db = np.zeros(self.b.shape)
        dxpad = np.zeros(xpad.shape)

        for i in range(B):
            for h in range(Hout):
                for w in range(Wout):
                    for c in range(Cout):
                        up = h * S
                        down = up + K
                        left = w * S
                        right = left + K
                        xwindow = xpad[i, up:down, left:right, :]

                        dxpad[i, up:down, left:right, :] += self.W[:,:,:,c] * dy[i, h, w, c]
                        dW[:,:,:,c] += xwindow * dy[i,h,w,c]
                        db[0,0,0,c] += dy[i,h,w,c]

            dx[i,:,:,:] = dxpad[i, P:-P, P:-P, :]

        #权重标准化
        dW = (dW*S*S) / (H*W*Cin)
        db = (db*S*S) / (H*W*Cin)
        dx = dx / (K*K)

        self.W = self.W  - self.lr * dW + self.momentum * (self.W - self.W_last)
        self.b = self.b  - self.lr * db + self.momentum * (self.b - self.b_last)
        self.W_last = np.array(self.W)
        self.b_last = np.array(self.b)

        return dx
```

在2D卷积之间与线性函数之间需要做flatten操作，同样定义前传和反传操作

```
class Flatten(Module):
    def __init__(self):
        super().__init__()

    def forward(self,x):
        self.x  = x
        self.y = x.reshape(x.shape[0],-1)
        return self.y

    def backward(self,g):
        g = g.reshape(self.x.shape)
        return g
```

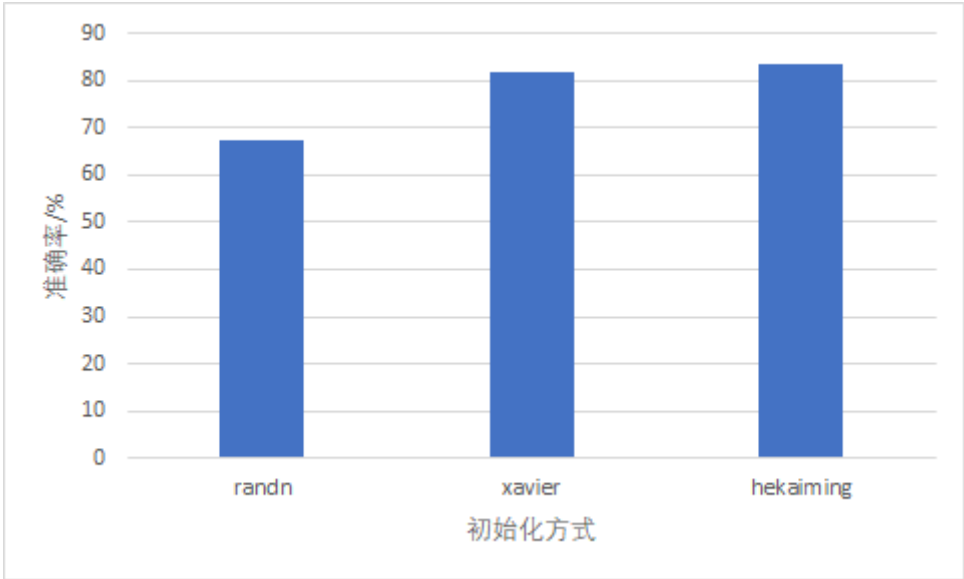
将简单模型的第一层改为卷积，训练较少的轮次，发现采用卷积层的效果确实更好。

卷积不仅具有很好的空间性质，而且使用了权重共享，可学习的参数量也大大降低了。



11. 其他实验

探究不同参数初始化方式，在同样的网络结构中改变参数初始化的方式，分别对比了随机正态初始化randn、xavier初始化、hekaiming初始化。发现hekaiming初始化和xavier初始化的确比randn初始化有很好地提升。而在实验中hekaiming初始化比xavier



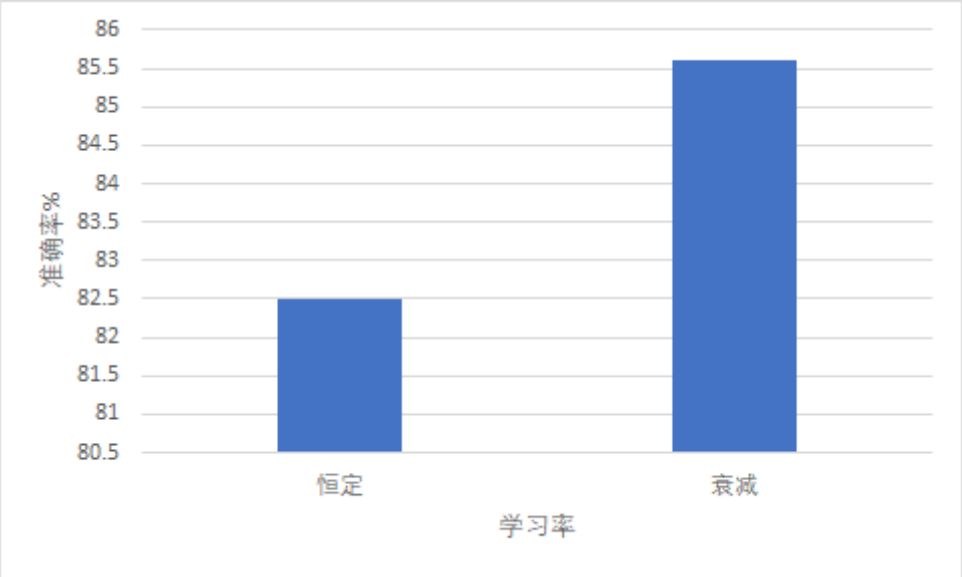
探究不同学习率曲线的影响，朴素的做法是使用恒定不变的学习率，但对于训练到后面的轮次，我们倾向于使用更小的学习率，因此采取学习率衰减的方式可能更为合适，简单的衰减方法可以如下在固定的轮次将学习率减半。

```
if e % fix_epoch == 0:
    self.fc1.lr /= 2
    self.fc2.lr /= 2
```

或者在多层模型中，使用，

```
for layer in self.fc.layers:
    if(isinstance(layer,Linear)):
        layer.lr /= 2
```

多次对比实验可以发现使用衰减的学习率的确具有更好的表现，当然也可以采用其他不同的学习率衰减方式，在此仅简单探究故不做其余实现。



Reference

[1] https://blog.csdn.net/yaoxunji/article/details/103224084

[2] https://pytorch.org/