Exercise2： LeNet5

朱霞洋 2113301

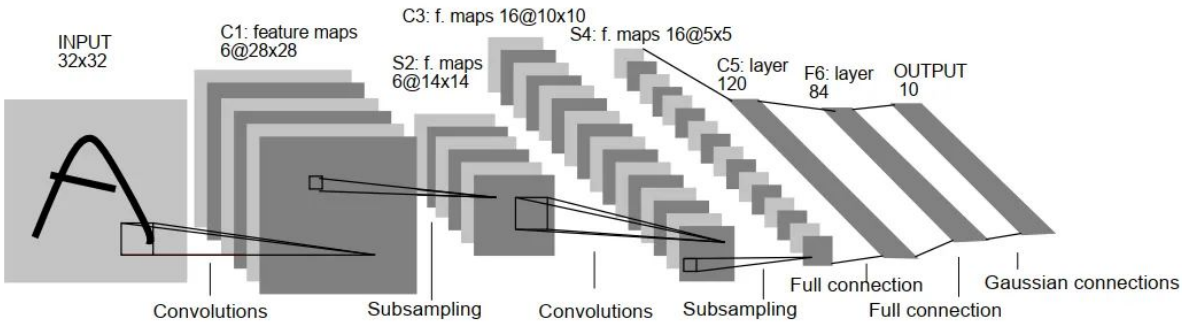
实验环境与目的

本次实验的目的是用 Python自主实现一个 LeNet5网络 来完成对 MNIST 数据集中 0-9 共 10 个手写数字的分类，实验的环境如下：

- python 3.10.11
- 仅引用了numpy计算库

LeNet5网络结构

LeNet-5 是由 Yann LeCun 在 1998 年提出的卷积神经网络结构，是深度学习领域的经典之一，主要应用于手写数字的识别，LeNet5 由 7 层组成，包括 2 个卷积层、2 个池化层、2 个全连接层和 1 个输出层，下面是 LeNet-5 网络的详细介绍：



1. 输入：

- 输入尺寸： 32 * 32大小的灰度图像，通道数为 1
- 每次训练按照一定batchsize（实验中设置为256）取样本进行训练，因此实际的输入尺寸是 batchsize * 1 * 32 * 32
- **注意**：由于MINIST数据集的图像大小为**28*28**，为了贴合原本的LeNet5网络，在输入层对其**进行了padding=2的填充**，保证了INPUT的形状是32 * 32

2. 第一层 - 卷积层 C1：

- 卷积核大小： 5 * 5
- 卷积核数量：6 (即输出通道数为6)
- 步长：1
- 激活函数：ReLu (此处与原本的LeNet5不同，原本的激活函数是Sigmoid函数，此处做了更改，具体原因见下文)
- 输出尺寸：batchsize * 6 * 28 * 28

3. 第二层 - 池化层 S2：

- 池化类型：2x2 最大池化 (MaxPooling)
- 步长：2
- 输出尺寸：batchsize * 6 * 14 * 14

4. 第三层 - 卷积层 C3：

- 卷积核大小： 5 * 5
- 卷积核数量：16
- 步长：1
- 激活函数：ReLu
- 输出尺寸：batchsize * 16 * 10 * 10

5. 第四层 - 池化层 S4：

- 池化类型：2x2 最大池化
- 步长：2
- 输出尺寸：batchsize * 16 * 5 * 5

6. 第五层 - 全连接层 C5：

- 输入神经元数量： 5 * 5 * 16 = 400
- 输出神经元数量：120
- 激活函数：ReLu

7. 第六层 - 全连接层 F6：

- 输入神经元数量：120
- 输出神经元数量：84
- 激活函数：ReLu

8. 输出层 - 全连接层 Output：

- 输入神经元数量：84
- 输出神经元数量：10（对应于手写数字的 0-9）
- 激活函数：Softmax

Softmax输出结果

在LeNet5中，softmax函数被用于输出层，将卷积层的输出映射到类别概率上，这样做不仅能够起到归一化的作用，在梯度计算时也能带来方便，具体实现见下一部分

ReLu激活函数

原本的LeNet5中使用Sigmoid函数作为激活函数，而在我的实现中将其更改为了当前使用更为广泛的ReLu函数，这样做是出于几点考虑：

- **解决梯度消失问题：** 在深度神经网络中，Sigmoid 函数在输入非常大或非常小的情况下，梯度接近于零，会导致梯度消失问题。ReLU 函数在正数范围内输出恒为正值，则不会导致梯度消失；
- **计算高效：** ReLU 的计算相对简单，只需要比较输入是否大于零，而 Sigmoid 需要进行指数运算，计算量相对较大；
- **稀疏激活性：** ReLU 对于负数的输入直接输出零，因此具有稀疏激活性，能够更加有效地学习和表示数据中的稀疏特征；
- **减轻梯度爆炸问题：** 在深度网络中，通过 ReLU 的正向传播不会使输入值变得非常大，从而减轻了梯度爆炸的问题

ReLU 激活函数的这些优点使得它在深度学习中得到了广泛的应用，在本次实验中，利用ReLu作为激活函数也取得了较好的结果

交叉熵损失函数

交叉熵损失函数（Cross-Entropy Loss）是在分类问题中常用的损失函数之一，尤其是在深度学习中应用广泛，其不仅能反应训练结果和实际值的差异，同时在进行梯度计算时也有着方便计算的特性。本次实验也选取交叉熵损失函数来衡量训练结果和实际值之间的差异，对其具体的求导分析在上一次实验中已经做过讨论，此处不再赘述，具体可见代码细节部分。

代码细节

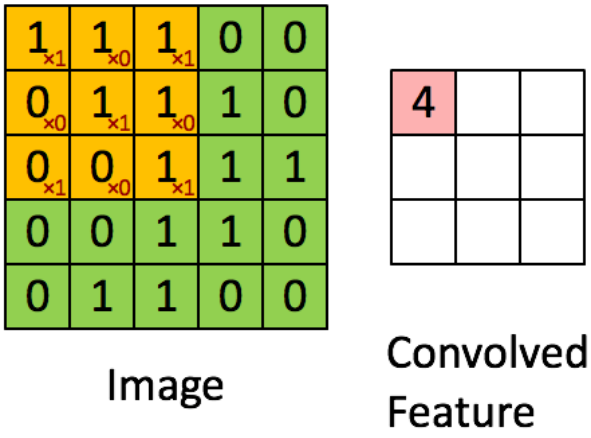
在本次实验代码的layers.py中，完成了各层的代码实现，包括前向计算，和梯度的反向传播，而在LeNet.py中完成了模型的组网和整体流程的完善，在AdamGrad.py中完成了优化器的实现，能够正确迭代网络参数。该部分会涉及各个参数的梯度的计算公式，符号说明为：

符号	说明
$Weight$	卷积核的梯度，或全连接层的权重参数
$Bias$	卷积层和全连接层的偏置参数
$input$	卷积层或全连接层的输入
$output$	卷积层或全连接层的输出
$grad$	反向传播时，下一层传递给该层的梯度（即output的梯度）
$grad\_next$	反向传播给上一层的梯度（即input的梯度）
W_grad	卷积核或全连接层的权重的梯度
B_grad	偏置参数的梯度

具体的细节如下：

卷积层

卷积层在前向传播时的计算过程和逻辑较为简单，代码实现起来也较为容易：




```
# in layers.py class Conv:
def conv2d(self,input,kernel,padding = 0,Bias = True): #卷积运算,使用kernel对input进行卷积
        N,C,H,W = input.shape #batchsize,channels,Height,Width
        if padding!= 0:      #进行padding操作，第一个卷积层C1会对28*28的图像进行padding=2的填充，符合原论文32*32的输入
            input= np.pad(input, ((0,0),(0,0),(padding, padding), (padding, padding)),
                                        'constant',constant_values = (0,0))
        num_kernels,_,filter_size,_ = kernel.shape
        # 计算输出特征图的宽度和高度
```

```
        output_W = (W + 2*padding - filter_size) // self.stride + 1
        output_W = int(output_W)
        output_H = (H + 2*padding - filter_size) // self.stride + 1
        output_H = int(output_H)
        # 初始化输出矩阵
        output_matrix = np.zeros((N, num_kernels, output_H, output_W))
        #进行卷积运算
        for h in range(output_H):
            for w in range(output_W):
                #计算卷积的区域(h_start,h_end),(w_start,w_end):
                h_start = h * self.stride
                h_end = h_start + filter_size
                w_start = w * self.stride
                w_end = w_start + filter_size
                #选取input的这一块区域，并进行reshape
                input_region = input[:, :, h_start:h_end,
w_start:w_end].reshape((N, 1, C, filter_size, filter_size))
                output_matrix[:, :, h, w] += np.sum(input_region * kernel, axis=
(2, 3, 4)) #通过numpy矩阵运算进行卷积
                if Bias is True: #是否加上偏置参数Bias
                    output_matrix[:, :, h, w] += self.Bias
        return output_matrix
```

此处对于卷积运算做了优化，避免了使用多重循环去逐个计算卷积结果的值，而是利用了numpy的向量计算的特性，先将要卷积的区域reshape为（batchsize，1，input_channels，filtersize，filtersize）的形状，然后与kernel (size = (ouput_channels，input_channels，filtersize，filtersize))进行numpy的矩阵运算，并在2，3，4轴上求和，最终可以计算出该batch中所有样本每一个通道上的卷积结果，从而避免了复杂的多层循环。

而卷积层在梯度反向传播时，主要有两个工作：计算卷积核的梯度，和计算要反向传输给上一层的梯度（即该卷积层的输入的梯度）

$$W\_grad = input \otimes grad$$
$$grad\_next = input \otimes Weight_{rot180^\circ}$$
$$B\_grad = \sum grad\_next$$

其中卷积层的输入的梯度，实际上等同于**下一层反向传播给该卷积层的梯度，经过零填充（padding = filtersize-1）后，与卷积核旋转180度后的卷积**，可以**复用上文的conv2d函数**进行计算，具体的代码如下：

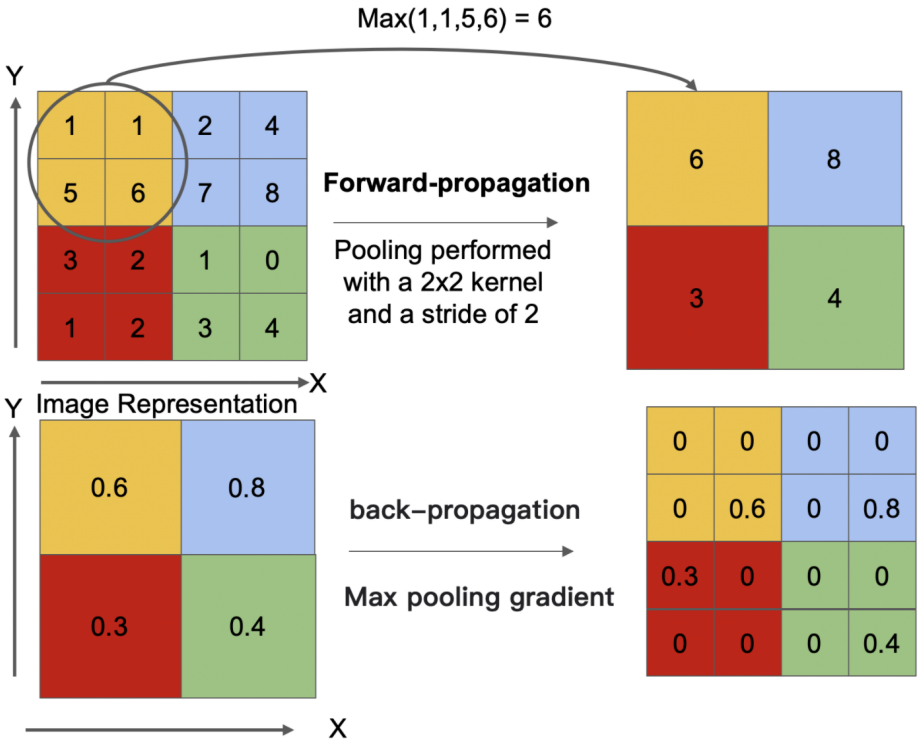
```
def backprop(self,grad):
        N, C, H, W = self.input.shape
        _, _, output_H, output_W = grad.shape #grad.shape和输出的shape相同，grad是卷
积层输出的梯度
        #得到卷积核的180°翻转
        reverse_kernel = self.weight.transpose((1,0,2,3))
        reverse_kernel  = np.flip(reverse_kernel,axis=(2,3))
        #卷积层输入的梯度实际上是输出的梯度经过padding后，与180°翻转的卷积核进行卷积的结果
        grad_next =  self.conv2d(grad,reverse_kernel,self.filter_size-
1,Bias=False)
        #计算卷积核的梯度
        self.W_grad = np.zeros_like(self.weight)
        for h in range(output_H):
            for w in range(output_W):
```

```
                tmp_back_grad = grad[:, :, h, w].T.reshape((self.out_channels, 1,
1, 1, N))
                tmp_x = self.input[:, :, h * self.stride:h * self.stride +
self.filter_size, w * self.stride:w * self.stride +
self.filter_size].transpose((1, 2, 3, 0))
                self.W_grad += np.sum(tmp_back_grad * tmp_x, axis=4)

        self.B_grad = np.sum(grad, axis=(0, 2, 3))  #计算偏置参数的梯度

        return grad_next
```

池化层

本次实现的LeNet5网络中，有两个2*2的最大池化层，在前向传播时其作用是取得每个2 * 2的块中最大的元素，对输入进行下采样

因此其forwad是一个简单的按块取最大操作，但是在反向传播backprop时，最大池化层要将每一个梯度映射回每一块的最大值的位置（也就是说反向传播时，每个2*2的块中，只有最大项才会有梯度）：



在我的实现中，通过一个mask掩码，在前向传播时记录下每个2*2块的最大值的位置，然后再反向传播时，根据掩码和梯度进行梯度的计算，具体代码和注释如下：

```
class MaxPool:
    def __init__(self,pool_size=None):
        if pool_size is None:
            pool_size = 2
        self.pool_size = pool_size
        self.output = None
        self.input = None
        self.mask = None

    def forward(self, X):
        N,C,W,H = X.shape
```

```
        self.input = X.copy()
        output_h = H // self.pool_size
        output_w = W // self.pool_size
        self.output = np.zeros((N,C,output_h,output_w))
        self.mask = np.zeros_like(X)  # 初始化最大值位置的掩码,用于记录每个2*2块中最大项的位置
        for i in range(output_h):
            for j in range(output_w):
                pool_window = X[:, :, i * self.pool_size:(i + 1) * self.pool_size, j * self.pool_size:(j + 1) * self.pool_size]
                self.mask[:, :, i * self.pool_size:(i + 1) * self.pool_size, j * self.pool_size:(j + 1) * self.pool_size] \
                    = (pool_window == np.max(pool_window, axis=(2, 3), keepdims=True)) # 记录最大值位置

                #取得最大值
                self.output[:,:,i,j] = np.max(X[:,:, i*self.pool_size:(i+1)*self.pool_size, j*self.pool_size:(j+1)*self.pool_size], axis=(2,3))
        return self.output

    def backprop(self, back_grad):
        N, C, H,W = back_grad.shape
        grad_next = np.zeros_like(self.input)

        for i in range(H):
            for j in range(W):
                #根据mask，将每一个梯度映射到每个2*2块的最大值位置
                grad_next[:, :, i*self.pool_size:(i+1)*self.pool_size, j*self.pool_size:(j+1)*self.pool_size]= \
                self.mask[:,:, i*self.pool_size:(i+1)*self.pool_size, j*self.pool_size:(j+1)*self.pool_size]* back_grad[:, :, i, j][:, :, None, None]

        return grad_next
```

在forwad过程中，池化层就将每个2*2块的最大值记录在mask中，在反向传播时即可利用该mask进行梯度的计算。

全连接层

全连接层实际上是矩阵之间的相乘，其前向传播和反向传播时的计算方法都较为简单：

$$output = input \times Weight + Bias$$

$$grad\_next = grad \times Weight.T$$
$$W\_grad = input.T \times grad$$
$$B\_grad = \sum grad\_next$$

具体的代码实现如下：

```
class Linear:
    def __init__(self,input_size,output_size) :
        self.Weight = np.random.normal(scale=1, size=(input_size, output_size))
        self.W_grad = None
        self.Bias = np.zeros(output_size)
```

```
        self.B_grad = None
        self.input = None

    def forward(self,X):
        self.input = X.copy()
        #利用numpy的矩阵乘法，计算输出
        return np.dot(X,self.Weight) + self.Bias

    def backprop(self,back_grad):
        self.W_grad = np.dot(self.input.T, back_grad)
        self.B_grad = np.sum(back_grad, axis=0)

        # 计算输入的梯度，用于传递给上一层
        grad_next = np.dot(back_grad, self.Weight.T)

        return grad_next
```

ReLu

ReLu的前向传播的计算方法为：

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

即只取输入的大于0的部分，而在反向传播时，输入中也仅仅只有大于0的项会有梯度，小于0的项的梯度均为0：

```
class Relu:
    def __init__(self):
        self.x = None

    def forward(self, x):
        self.x = x
        return np.maximum(0, x)

    def backprop(self,grad):
        #大于0的地方才会有梯度
        return np.where(self.x > 0, 1, 0) * grad
```

softmax

在最后的Output层计算出结果之后，还会对结果进行一个softmax操作，并且计算预测结果和真实值的差别（交叉熵损失函数），以及此时的梯度。由于softmax和交叉熵函数的特性，可以推导得知此时的梯度正是$(y_pred - y)/N$，因此代码实现如下：

```python
def softmax(y_pred,y):
    batch_size ,_ = y_pred.shape
    y_pred = np.exp(y_pred)          #进行指数运算
    y_sum = y_pred.sum(axis = 1)     #求和
    y_pred = y_pred/y_sum[:,None]    #softmax计算
    loss = -np.log(y_pred).T * y     #交叉熵损失函数计算
    loss = loss.sum()/batch_size     #得到（平均）损失值
    grad = y_pred - y.T              #计算得到梯度
    grad /= batch_size
    acc = (y_pred.argmax(axis=1) == y.argmax(axis=0)).mean()#计算正确率
    return loss,grad,acc  #返回损失值，梯度，训练集准确率
```

参数迭代优化

在参数迭代过程中，使用了Adam（Adaptive Moment Estimation）梯度下降优化算法，该算法结合了动量法和自适应学习率的思想，能够提高收敛速度和性能。Adam优化算法有以下特点：

1. **动量（Momentum）：** Adam使用动量的概念，通过维护一个动量变量$\hat{m}_t$，累积之前梯度的信息，帮助加速收敛。动量的引入有助于处理不同方向上的梯度变化，减缓在某一方向上的波动
2. **自适应学习率：** Adam维护一个适应性学习率α，会随着训练过程而变化，能够帮助模型更好地收敛
3. **偏差修正：** 由于在训练初期，$\hat{m}_t$和$\hat{v}_t$的估计值会偏向零，可能导致算法表现不佳。因此，Adam使用偏差修正，对$\hat{m}_t$和$\hat{v}_t$进行修正
4. **超参数：** Adam有两个主要超参数，即β_1和β_2，用于控制动量项和梯度平方项的指数加权移动平均。通常，取β_1取0.9，β_2取0.999,学习率α会根据超参数而进行变化，通常在训练过程中逐渐减小。

Adam的更新规则可以表示为：

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$

其中，θ是参数，α是学习率，$\hat{m}_t$和$\hat{v}_t$分别是动量和梯度平方项的偏差修正估计，ϵ是一个很小的数，避免分母为零。

而学习率α以及$\hat{m}_t$和$\hat{v}_t$的更新按照以下公式：

$$\alpha_t = \alpha \times \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t}$$
$$m_t = \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t$$
$$v_t = \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot g_t^2$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

其中：

- α_t 是迭代次数为 t 时的学习率
- α是初始学习率
- β_1和 β_2是控制一阶矩和二阶矩的衰减率的超参数（通常分别取0.9和0.999）

- t 是当前的时间步(即迭代次数)
- g_t是时间步 t 时的梯度

修正后的一阶矩和二阶矩会用于计算参数的更新，修正的目的是在算法初期，由于 m_t和 v_t 初始值较小，导致修正后的估计更准确

相比起随机梯度下降，Adam优化算法有许多好处，例如自适应学习率和动量使模型更具鲁棒性，适应性更好，不用手动调节学习率等等。具体的代码实现如下：

```
# in AdaGrad.py
def update(self):
    self.params = self.model.get_params()
    self.grad = self.model.get_grad()
    self.iter += 1
    #更新学习率
    alpha_t = self.alpha * np.sqrt(1.0 - self.beta2 ** self.iter) / (1.0 -
self.beta1 ** self.iter)
    #更新参数
    for i in range(len(self.params)):
        self.m[i] += (1 - self.beta1) * (self.grad[i] - self.m[i])
        self.v[i] += (1 - self.beta2) * (self.grad[i] ** 2 - self.v[i])
        self.params[i] -= alpha_t * self.m[i] / (np.sqrt(self.v[i]) + 1e-7)
```

网络组网与前向传播

在之前的所有实现好后，组网和前向传播就十分简单了，按照顺序一层层连接即可：

```
# in LeNet.py
class LeNet:
    def __init__(self):
        self.Conv1 = Conv(1,6,5,1,2)    # 卷积层1，输入通道为1，6个5*5的卷积
核,padding=2,得到32*32的输入,输出N*6*28*28
        self.Relu1 = Relu()
        self.Pool1 = MaxPool(2)     # 池化层1，2*2大小的最大池化,输出N*6*14*14
        self.Conv2 = Conv(6,16,5)   # 卷积层2，6输入通道，16个5*5的卷积核,输出
N*16*10*10
        self.Relu2 = Relu()
        self.Pool2 = MaxPool(2)     # 池化层2，2*2大小的最大池化，输出N*16*5*5
        self.Fc1 = Linear(16*5*5,120)   # 全连接层1，16*5*5的输入，120个神经元
        self.Relu3 = Relu()
        self.Fc2 = Linear(120,84)       # 全连接层2，120输入，84个神经元
        self.Relu4 = Relu()
        self.Output = Linear(84,10)     # 输出层，84输入，10个输出

    def fit(self,x,batch_size):  #前向传播
        x = x.reshape((batch_size,1,28,28)) #N,C,H,W
        x = self.Pool1.forward(self.Relu1.forward(self.Conv1.forward(x)))
        x = self.Pool2.forward(self.Relu2.forward(self.Conv2.forward(x)))
        x = x.reshape(batch_size,-1) #进行全连接前，先展开
        x = self.Relu3.forward(self.Fc1.forward(x))
        x = self.Relu4.forward(self.Fc2.forward(x))
        x = self.Output.forward(x)
        return x
```

梯度反向传播与参数更新

反向传播时，通过softmax函数得到的grad用作输入梯度进行传播，在传播完毕后通过Adam优化器进行参数的优化：

```
def back_prop(self,grad):
    grad = self.Output.backprop(grad)
    grad = self.Relu4.backprop(grad)
    grad = self.Fc2.backprop(grad)
    grad = self.Relu3.backprop(grad)
    grad = self.Fc1.backprop(grad)
    grad = grad.reshape(grad.shape[0],16,5,5) #需要进行reshape
    grad = self.Pool2.backprop(grad)
    grad = self.Relu2.backprop(grad)
    grad = self.Conv2.backprop(grad)
    grad = self.Pool1.backprop(grad)
    grad = self.Relu1.backprop(grad)
    grad = self.Conv1.backprop(grad)

def update(self):
    self.optimizer.update() #optimizer变量是一个AdamGrad类的实例
```

batchsize取样

本次实验设置batchsize为256，每次随机取样，防止模型只学习到一定排序下的数字特征：

```
x,y = data_convert(train_images, train_labels,60000,10)
def shuffle_batch(batch_size):
    index = np.random.randint(0,len(x),batch_size) #随机取batchsize个整数
    return x[index],y.T[index].T
```

实验结果

设置batchsize = 256 ， Adam优化器初始学习率1e-3， 对60000个样本进行10次迭代训练，使用tqdm包输出训练进度，得到的结果如下（在LeNet.ipynb中）：

```
batch_size = 256
model = LeNet()
for e in range(1):
    bar = tqdm(range(0, int(x.shape[0]/batch_size)), ncols=100)
    for i in bar:
        X_train,y_train = shuffle_batch(batch_size)
        y_pred = model.fit(X_train,batch_size)

        loss, grad, acc = softmax(y_pred, y_train)
        Loss.append(loss)
        Acc.append(acc)
        model.back_prop(grad)
        model.update()
        bar.set_postfix(loss=loss, acc=acc)
```

```
100%|| 234/234 [04:45<00:00,  1.22s/it, acc=0.957, loss=0.153]
100%|| 234/234 [04:37<00:00,  1.19s/it, acc=0.961, loss=0.133]
100%|| 234/234 [04:32<00:00,  1.17s/it, acc=0.977, loss=0.0763]
100%|| 234/234 [04:36<00:00,  1.18s/it, acc=0.992, loss=0.0335]
100%|| 234/234 [04:34<00:00,  1.17s/it, acc=0.984, loss=0.0325]
100%|| 234/234 [04:59<00:00,  1.28s/it, acc=1, loss=0.00766]
100%|| 234/234 [04:53<00:00,  1.25s/it, acc=0.992, loss=0.0541]
100%|| 234/234 [04:33<00:00,  1.17s/it, acc=0.992, loss=0.0294]
100%|| 234/234 [04:29<00:00,  1.15s/it, acc=0.98, loss=0.061]
100%|| 234/234 [04:31<00:00,  1.16s/it, acc=0.996, loss=0.00886]
```

可见，在训练集上最终能够达到99.6%的准确率，损失值为0.00886，相比于第一次作业损失值0.31，提升十分巨大，可见LeNet5多层网络带来的显著效应。

实际上，通过训练的结果可以看出，仅仅对60000个样本进行一轮迭代，就可以达到95.7%的准确率了，在第四轮迭代完成后，在训练集上的准确率已经可以达到99.2%，之后的迭代也只是在这个数值周围变化，但是考虑到Adam优化器的自适应学习度，以及损失值不断下降的效果来看，额外的训练次数虽然在准确率上提升没有那么多，但是提高了模型的泛化能力。

最终在测试集上的表现也十分优秀：

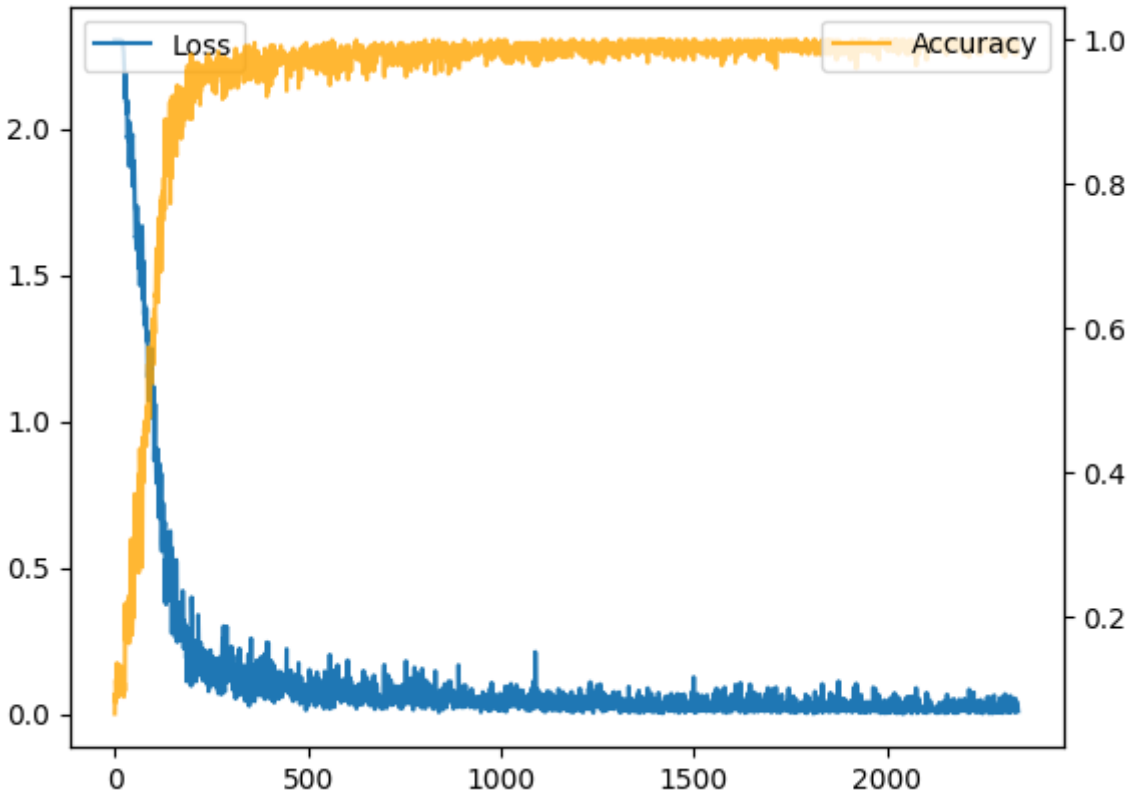
```
accuracy  = cal_accuracy(model,x_val, y_val)
print("accuracy: {:.2%}".format(accuracy))
print("Finished test. ")
```

✓ 0.0s

```
accuracy: 98.85%
Finished test.
```

最终的测试集准确率可达98.85%，相比起第一次作业单个全连接的迭代1500轮后才达到的最高准确率93.75%，有了显著的提升，基本上可以正确无误地识别数字了，而且这只是对训练集迭代10次的结果，如果适当地调整迭代次数，或许还能有进一步的提升。

将训练过程准确率和损失值进行记录后，得到如下结果：



可见在最开始的200轮迭代，就已经将准确率提升到了90%以上，准确率的提升速度以及损失值的下降速度十分高，可见Adam优化器快速收敛的优点。在500轮之后，准确率和损失值实际上变化已经不大了，但是从图中可以看出，随着迭代次数的增加，准确率和损失值的波动在逐步减小，最终已经稳定地收敛。

参考资料

[卷积神经网络经典回顾之LeNet-5]

[卷积神经网络(CNN)反向传播算法推导]

[The Architecture and Implementation of LeNet-5]