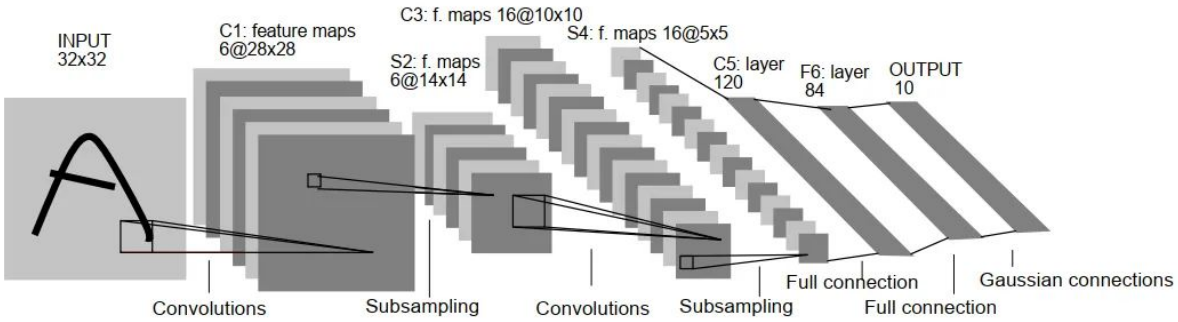
Ex3-LeNet5实现手写数字识别

LeNet5网络结构



各层参数

输入层

输入层是一维图像32×32。

C1卷积层

参数	value
input_channels	1
filter_size	5×5
stride	1
padding	0
output	6×28×28

S2降采样层

参数	value
pool_size	2×2
stride	2
padding	0
output	6×14×14

C3卷积层

参数	value
input_channels	6
filter_size	5×5

参数	value
stride	1
padding	0
output	16×10×10

S4降采样层

参数	value
pool_size	2×2
stride	2
padding	0
output	16×5×5

C5卷积层

该卷积层相当于 120 个神经元的全连接层，所以在实验中直接将16×5×5展开成维度为400的一维向量，用全连接层实现。

参数	value
input_channels	16
filter_size	5×5
stride	1
padding	0
output	120×1×1

F6全连接层

参数	维度
input	120
output	84

输出层

参数	维度
input	84
output	10

激活函数

实现中采用`Relu`作为激活函数，避免在卷积网络中出现梯度消失。

损失函数

使用交叉熵函数计算损失值。

实验环境

IDE	vscode
python版本	3.10.2
CPU	Core i7-10875H

代码细节实现

代码架构：

- Layers
  - conv.py：实现Conv层
  - maxpool.py：实现MaxPool层
  - fullyconnect.py：实现FullyConnect层
- 激活函数
  - ReLu
  - Softmax
- 损失函数 loss.py：softmax_loss
- 优化器：Adam
- 模型：LeNet5

各层Layers的实现

1. **卷积层 Conv.py**

$$
\begin{gathered}
input\_channels：输入通道数, \\
kernels：输出通道数, \\
filter\_size：卷积核大小, \\
stride：步长, \\
padding：填充, \\
W：权重, \\
b：偏置项
\end{gathered}
$$

Conv层定义：

```
class Conv():
    def __init__(self, input_channels, kernels, filter_size = 5):
        self.input = None
        self.input_channels = input_channels
        self.kernels = kernels
        self.filter_size = filter_size
        self.stride = 1
        self.padding = 0

        self.W = np.random.normal(scale=1e-2, size=(kernels, input_channels,
filter_size, filter_size))
        self.b = np.zeros(kernels)
```

前向传播计算公式：

$$Y = x \otimes W + b$$

实现代码：

```
def forward(self, x):
        self.input = x.copy()
        N, C, H, W = x.shape
        padded = np.pad(x, ((0, 0), (0, 0), (self.padding, self.padding),
(self.padding, self.padding)))

        output_h = 1 + (H + 2 * self.padding - self.filter_size) //
self.stride
        output_w = 1 + (W + 2 * self.padding - self.filter_size) //
self.stride

        Y = np.zeros((N, self.kernels, output_h, output_w))
        for h in range(output_h):
            for w in range(output_w):
                _x = h * self.stride
                _y = w * self.stride

                #y = x*w + b
                temp_x = padded[:, :, _x:_x + self.filter_size, _y:_y +
self.filter_size].reshape((N, 1, self.input_channels, self.filter_size,
self.filter_size))
                temp_w = self.W['m'].reshape((1, self.kernels,
self.input_channels, self.filter_size, self.filter_size))
                Y[:, :, h, w] = np.sum(temp_x * temp_w, axis=(2, 3, 4)) +
self.b['value']

        return Y
```

反向传播的计算公式：

$$dW = dy \otimes x$$
$$db = dy$$
$$dx = dy \otimes W$$

实现代码：

```
def backward(self, dy):
        N, C, H, W = self.input.shape
        padded = np.pad(self.input, ((0, 0), (0, 0), (self.padding,
self.padding), (self.padding, self.padding)))
        output_h = 1 + (H + 2 * self.padding - self.filter_size) //
self.stride
        output_w = 1 + (W + 2 * self.padding - self.filter_size) //
self.stride

        dW = np.zeros((self.kernels, self.input_channels, self.filter_size,
self.filter_size))
        db = np.zeros(self.kernels)
        dx = np.zeros_like(padded)

        for h in range(output_h):
            for w in range(output_w):
                _x = h * self.stride
                _y = w * self.stride

                #dw = dy * x
                temp_dy = dy[:, :, h, w].T.reshape((self.kernels, 1, 1, 1,
N))
                x_T = padded[:, :, h * self.stride:h * self.stride +
self.filter_size, w * self.stride:w * self.stride +
self.filter_size].transpose((1, 2, 3, 0))
                dW += np.sum(temp_grad * x_T, axis=4)

                #db
                db += np.sum(dy[:, :, h, w], axis=0)
                temp_dy = dy[:, :, h, w].reshape((N, 1, 1, 1, self.kernels))
                temp_w = self.W.transpose((1, 2, 3, 0)).reshape((1,
self.input_channels, self.filter_size, self.filter_size, self.kernels))

                #dx
                dx[:, :, _x:_x + self.filter_size, _y:_y + self.filter_size]
+= np.sum(temp_dy * temp_w, axis=4)

        dx = dx[:, :, self.padding:self.padding + H,
self.padding:self.padding + W]
        return dx, dW, db
```

2. **池化层 MaxPool.py**

$$pool\_size：池化采样大小$$
$$stride：步长$$

MaxPool层定义：

```
class MaxPool():
    def __init__(self, pool_size = [2,2] ,stride = 2):
        self.input = None
        self.pool_size = pool_size
        self.stride = stride
```

前向传播：实现最大池化的滤波器

```
    def forward(self, x):
        self.input = x.copy()

        pool_h = self.pool_size[0]
        pool_w = self.pool_size[1]
        N, C, H, W = x.shape
        output_h = 1 + (H - pool_h) // self.stride
        output_w = 1 + (W - pool_w) // self.stride
        Y = np.zeros((N, C, output_h, output_w))

        for h in range(output_h):
            for w in range(output_w):
                _x = h * self.stride
                _y = w * self.stride
                #滤波器选择最大元素
                Y[:, :, h, w] = np.max(x[:, :, _x:_x + pool_h, _y:_y +
pool_w], axis=(2, 3))
        return Y
```

反向传播：只有被滤波器选择到的元素所在位置才会反向传播梯度，其余都是0

```
    def backward(self, dy):
        pool_h = self.pool_size[0]
        pool_w = self.pool_size[1]
        N, C, H, W = self.input.shape
        output_h = 1 + (H - pool_h) // self.stride
        output_w = 1 + (W - pool_w) // self.stride
        dx = np.zeros_like(self.input)

        for h in range(output_h):
            for w in range(output_w):
                _x = h * self.stride
                _y = w * self.stride
                pool_ = np.zeros((N, C, pool_h * pool_w))
                #只有最大元素处为1，其余位0
                pool_[np.arange(N)[:, None], np.arange(C)[None, :],
np.argmax(self.input[:, :, _x:_x + pool_h, _y:_y + pool_w].reshape((N, C,
-1)), axis=2)] = 1
                pool_ = pool_.reshape((N,C,pool_h,pool_w))
                dx[:, :, _x:_x + pool_h, _y:_y + pool_w] = pool_ * dy[:, :,
h, w][:, :, None, None]

        return dx
```

3. **全连接层FullyConnect**

$$
\begin{aligned}
input\_size&：输入维度，\\
output\_size&：输出维度，\\
W&：权重，\\
b&：偏置项
\end{aligned}
$$

FullyConnect层定义：

```
class FullyConnect():
    def __init__(self, input_size, output_size):
        self.input = None
        self.input_size = input_size
        self.output_size = output_size
        self.W = np.random.normal(scale=1e-2, size=(input_size,
output_size))
        self.b = np.zeros(output_size)
```

前向传播：

$$Y = x \times W + b$$

```
    def forward(self, x):
        self.input = x.copy()
        Y = np.dot(x, self.W) + self.b
        return Y
```

反向传播：

$$dx = dy \times W^T$$
$$dW = x^T \times dy$$
$$db = dy$$

```
    def backward(self, dy):
        dx = np.dot(dy, self.W.T)
        dW = np.dot(self.input.T, dy)
        db = np.sum(dy, axis=0)
        return dx, dW, db
```

ReLu激活函数

```
class ReLu():
    def __init__(self):
        self.input = None

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

    def backward(self, dy):
        dx = self.input > 0
        return dy * dx
```

损失函数

采用交叉熵损失函数，实现如下：

```
def softmax_loss(y_pred, y):
    ex = np.exp(y_pred)
    sumx = np.sum(ex, axis=1)
    size = y_pred.shape[0]
    loss = np.mean(np.log(sumx)-y_pred[range(size), list(y)])
    sumx = sumx.reshape(size, 1)
    dx = ex/sumx
    dx[range(size), list(y)] -= 1
    dx /= N
    return loss, dx
```

Adam优化算法

为了更好完成这次作业，放弃一般的随机梯度下降算法，选择目前效果更好的自适应学习率Adam算法。

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1^{st} moment vector)
$v_0 \leftarrow 0$ (Initialize 2^{nd} moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

为了方便实现，设计Adam类来实现一个adam对象对全部参数进行更新，具体实现如下：

```
import numpy as np

class Adam:
    def __init__(self, params, alpha=1e-3):
        #学习率
        self.alpha = alpha
        self.beta1 = 0.9
        self.beta2 = 0.999
        self.epsilon = 1e-8

        self.iter = 0
        #权重参数
        self.params = params
        #梯度
        self.params_grad = None
        self.m = []
        self.v = None
        for i in range(len(self.params)):
```

```
            self.m.append(np.zeros_like(self.params[i]))

    def setlr(self, alpha):
        self.alpha = alpha

    def set_grad(self, grad):
        self.params_grad = grad
        if self.v is None :
            self.v = []
            for i in range(len(self.params)):
                self.v.append(np.zeros_like(self.params_grad[i]))

    def grad(self):
        #update all param, using m, v and param_gard
        self.iter += 1
        for i in range(len(self.params_grad)):
            self.m[i] = (1 - self.beta1) * (self.params_grad[i]) + self.beta1 *
self.m[i]
            self.v[i] = (1 - self.beta2) * (self.params_grad[i] ** 2) +
self.beta2 * self.v[i]
            m_c = self.m[i] / (1.0 - self.beta1 ** self.iter)
            v_c = self.v[i] / (1.0 - self.beta2 ** self.iter)
            #adam to update param
            self.params[i] -= (self.alpha * m_c) / (np.sqrt(v_c) + self.epsilon)
```

LeNet5模型实现

借鉴目前改进的LeNet网络结构，在每个池化层后用ReLu激活函数，并在每个全连接层也使用ReLu作为激活函数，得到最后LeNet模型结构：是Conv层-->ReLu层-->MaxPool层-->Conv层-->ReLu层-->MaxPool层-->FullyConnect层-->ReLu层-->FullyConnect层-->ReLu层-->FullyConnect层->ReLu层-->FullyConnect层，具体实现如下：

针对本次多分类任务，我还在实验部分给最后输出层加了一个softmax函数，对比一下跟普通LeNet网络结构的差别。

```
class LeNet:
    def __init__(self):
        #c1
        self.c1 = Conv(1, 6, 5)
        self.relu1 = ReLu()
        #s2
        self.s2 = MaxPool((2, 2), 2)
        #c3
        self.c3 = Conv(6, 16, 5)
        self.relu2 = ReLu()
        #s4
        self.s4 = MaxPool((2, 2), 2)
        #c5卷积层相当于120个神经元的全连接层
        self.c5 = FullyConnect(400, 120)
        self.relu3 = ReLu()
        #fc6
        self.fc6 = FullyConnect(120, 84)
        self.relu4 = ReLu()
        #output
        self.output = FullyConnect(84, 10)
```

```
        #使用adam优化算法进行深度学习的权重更新
        self.adam = Adam(self.get_params())

    def forward(self, x):
        x = self.c1.forward(x)
        x = self.relu1.forward(x)
        x = self.s2.forward(x)
        x = self.c3.forward(x)
        x = self.relu2.forward(x)
        x = self.s4.forward(x)
        #展开为一维向量
        x = x.reshape(x.shape[0], -1)
        x = self.c5.forward(x)
        x = self.relu3.forward(x)
        x = self.fc6.forward(x)
        x = self.relu4.forward(x)
        x = self.output.forward(x)
        return x

    def backward(self, dy):
        #记录梯度
        params_grad = []
        dy, dw, db = self.output.backward(dy)
        params_grad.append(db)
        params_grad.append(dw)

        dy = self.relu4.backward(dy)
        dy, dw, db = self.fc6.backward(dy)
        params_grad.append(db)
        params_grad.append(dw)

        dy = self.relu3.backward(dy)
        dy, dw, db = self.c5.backward(dy)
        params_grad.append(db)
        params_grad.append(dw)

        dy = dy.reshape(dy.shape[0], 16, 5, 5)
        dy = self.s4.backward(dy)
        dy = self.relu2.backward(dy)
        dy, dw, db = self.c3.backward(dy)
        params_grad.append(db)
        params_grad.append(dw)

        dy = self.s2.backward(dy)
        dy = self.relu1.backward(dy)
        dy, dw, db = self.c1.backward(dy)
        params_grad.append(db)
        params_grad.append(dw)

        params_grad.reverse()
        self.adam.set_grad(params_grad)
        #使用adam优化器进行权重的更新
        self.adam.grad()
```

实验过程

如何进行训练

train函数的实现：

```
def train_model(data, model):
    model.setlr(lr)
    best_acc = 0.0
    best_model = None
    for e in range(epochs):
        for i in range(int(data['x_train'].shape[0]/batch_size):
            x = data["x_train"][i:i+batch_size]
            y = data["y_train"][i:i+batch_size]

            y_pred = model.forward(x)
            loss, dy= softmax_loss(y_pred, y)
            _loss.append(loss)
            model.backward(dy)

        x_test = data["x_test"]
        y_test = data["y_test"]
        y_pred = model.forward(x_test)
        y_pred = np.argmax(y_pred, axis=1)
        acc = np.mean(y_pred == y_test.reshape(1, y_test.shape[0]))
        #利用acc选择最优模型
        if acc > best_acc:
            best_acc = acc
            best_model = model.get_model_weight()
    return best_model
```

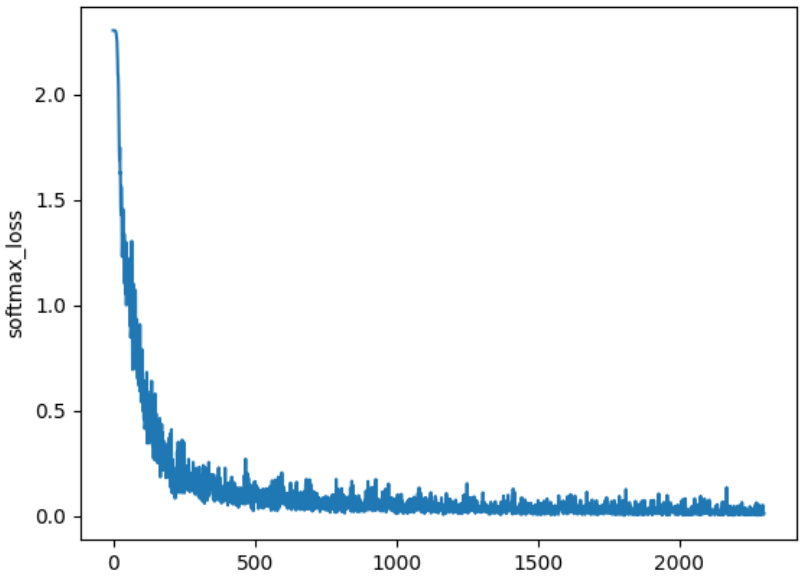
超参数

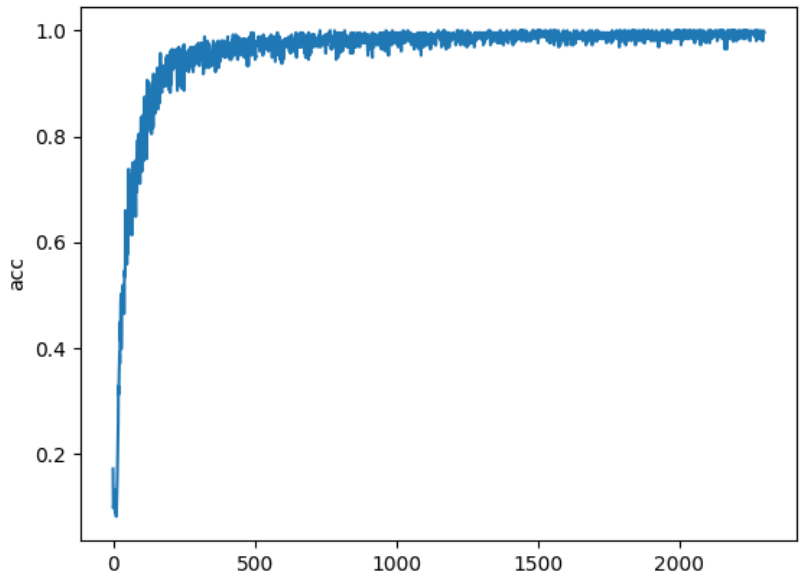
训练集:测试集	5:1
epoch	8
alpah	0.0015
batch_size	256
epsilon	1e-8

训练结果

最终在训练集中正确率为`99.09%`。进行五次测试，在测试集上的平均正确率为`98.93%`。说明模型效果比较不错。

绘出训练过程中损失值和正确率随迭代次数变化的曲线图：



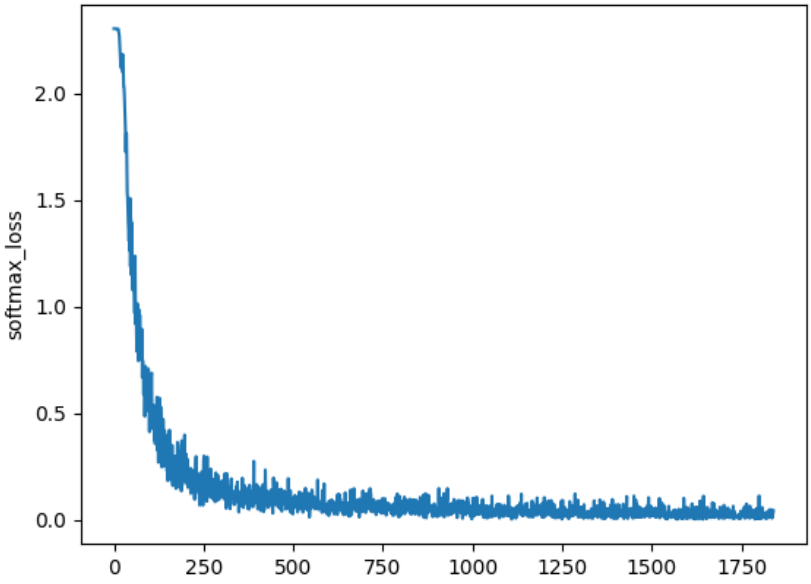


结果分析

由`softmax_loss`那张图可以发现，随着迭代次数增加，损失值也迅速下降，并在750次迭代学习后趋于0.10附近，已经能达到比较好的模型效果，模型收敛。

但是模型仍有波动，可以发现在900次迭代和1600次迭代左右损失值突然增大到0.25附近，我认为这可能是由于过拟合造成的波动，但是总体而言模型效果表现较好。

最后output层加一个softmax激活函数：最终测试集上准确率达到`99.11%`，下面是其损失值曲线，发现在迭代次数比较大时模型损失值的波动更小，且准确率得到略微提高，根本原因是softmax更适合这种多分类问题。


softmax_loss
2.0
1.5
1.0
0.5
0.0
0
250
500
750
1000
1250
1500
1750