Ex3：LeNet5手写数字识别

- 姓名：陈睿颖
- 学号：2013544
- 专业：计算机科学与技术

1. 实验要求

在这个练习中，需要用Python实现LeNet5来完成对MNIST数据集中 0-9 10个手写数字的分类。代码只能使用python实现，**不能**使用PyTorch或TensorFlow框架。

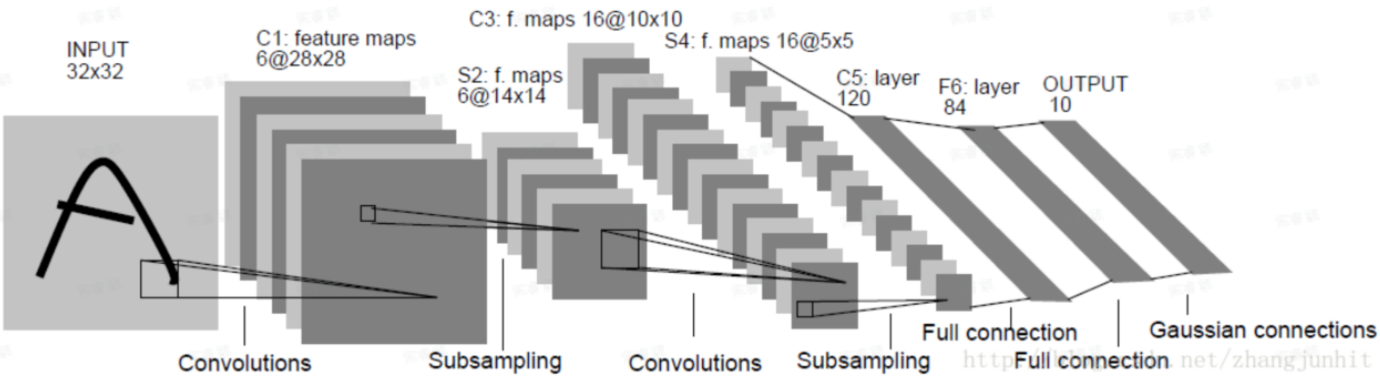
2. 实验环境

本实验环境如下：

运行环境	VS Code
Python版本	3.10.8

3. 实验原理

3.1 LeNet5网络



如图所示，LeNet 由特征提取模块和分类模块两部分组成，特征提取模块主要由卷积层和降采样层组成，分类模块主要由全连接层组成。与当时流行的模式识别算法不同，LeNet 为特征提取模块和分类模块均可训练的端到端模型。

3.2 结构各层

1. 输入层
   - 维度：$1 \times 32 \times 32$
2. 卷积层C1
   - 输入通道数：1
   - 输出通道数：6
   - 卷积核大小：5×5
   - 步长：1
   - 输出图像大小：$6 \times 28 \times 28$
   - 无填充
3. 池化层S2
   - 输入通道数：6

- 输出通道数：6
- 滤波器大小：2×2
- 输出图像大小：$6 \times 14 \times 14$
- 步长：2

4. 卷积层C3
    - 输入通道数：6
    - 输出通道数：16
    - 卷积核大小：5×5
    - 步长：1
    - 输出图像大小：$16 \times 10 \times 10$
    - 无填充
5. 池化层S4
    - 输入通道数：6
    - 输出通道数：6
    - 滤波器大小：2×2
    - 输出图像大小：$16 \times 5 \times 5$
    - 步长：2
6. 全连接层F5
    - 输入维度：400
    - 输出维度：120
7. 全连接层F6
    - 输入维度：120
    - 输出维度：84
8. 输出层
    - 输入维度：84
    - 输出维度：10
9. 损失函数

    采用交叉熵损失函数。

4. 代码细节

4.1 整体结构

文件	功能
data_process.py	处理数据，将原始数据集中的的 28 × 28 原始图像转换为模型输入所需的 32 × 32 × 1
layers.py	定义了一个表示网络层结构的抽象基类 Layer，激活函数ReLU及卷积层、池化层、全连接层等
train.py	训练模型，包括加载数据，绘制实验结果曲线图及测试模型的准确度
model.py	定义LeNet5模型及softmax损失函数
optimizer.py	定义了基于Adam算法的进行权重的更新
main.py	主函数，执行以上定义的函数，进行手写数字的识别

4.2 主要代码

4.2.1 处理数据

将原数据集中的 28 × 28 原始图像转换为模型输入所需 的 32 × 32 × 1，并转换为与模型匹配的 float32 类型。从训练集中划分出大小为 1000 的验证 集，完成数据归一化，并返回包含训练集、验证集和测试集数据及标签的字典。

数据的加载：

```python
# call the load_mnist function to get the images and labels of training set and testing set
def load_data(mnist_dir, train_data_dir, train_label_dir, test_data_dir, test_label_dir):
    print('Loading MNIST data from files...')
    train_images = load_mnist(os.path.join(mnist_dir, train_data_dir), True)
    train_labels = load_mnist(os.path.join(mnist_dir, train_label_dir), False)
    test_images = load_mnist(os.path.join(mnist_dir, test_data_dir), True)
    test_labels = load_mnist(os.path.join(mnist_dir, test_label_dir), False)

    train_images = np.pad(train_images, ((0, 0), (2, 2), (2, 2)))
    test_images = np.pad(test_images, ((0, 0), (2, 2), (2, 2)))

    _, H, W = train_images.shape
    train_images = train_images.astype(np.float32).reshape(-1, 1, H, W)
    test_images = test_images.astype(np.float32).reshape(-1, 1, H, W)

    validation_images = train_images[-1000:]
    validation_labels = train_labels[-1000:]
```

```
    train_images = train_images[:-1000]
    train_labels = train_labels[:-1000]

    mean_image = np.mean(train_images, axis=0)
    train_images -= mean_image
    validation_images -= mean_image
    test_images -= mean_image

    return {
        "X_train": train_images,
        "y_train": train_labels,
        "X_val": validation_images,
        "y_val": validation_labels,
        "X_test": test_images,
        "y_test": test_labels,
    }
```

其中用到的 `load_mnist` 函数:

```
def load_mnist(file_dir, is_images='True'):
    # Read binary data
    bin_file = open(file_dir, 'rb')
    bin_data = bin_file.read()
    bin_file.close()
    # Analysis file header
    if is_images:
        # Read images
        fmt_header = '>iiii'
        magic, num_images, num_rows, num_cols = struct.unpack_from(fmt_header, bin_data, 0)
    else:
        # Read labels
        fmt_header = '>ii'
        magic, num_images = struct.unpack_from(fmt_header, bin_data, 0)
        num_rows, num_cols = 1, 1
    data_size = num_images * num_rows * num_cols
    mat_data = struct.unpack_from('>' + str(data_size) + 'B', bin_data, struct.calcsize(fmt_header))
    if is_images:
        mat_data = np.reshape(mat_data, [num_images, num_rows, num_cols])
    else:
        mat_data = np.reshape(mat_data, [num_images])
    print('Load images from %s, number: %d, data shape: %s' % (file_dir, num_images, str(mat_data.shape)))
    return mat_data
```

4.2.2 模型设计

1. Layers

为了方便，将LeNet5网络结构中的各层单拎出来进行定义，方便后续构建LeNet5模型。

卷积层：

```python
class Conv(Layer):
    def __init__(self, in_channels, out_channels, filter_size, stride=1, padding=0):
        super().__init__()
        self.input = None
        self.output = None
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.W = {'value': np.random.normal(scale=1e-3, size=(out_channels, in_channels, filter_size, filter_size)),
                  'grad': np.zeros((out_channels, in_channels, filter_size, filter_size))}
        self.b = {'value': np.zeros(out_channels),
                  'grad': np.zeros(out_channels)}
        self.filter_size = filter_size
        self.stride = stride
        self.padding = padding

    def forward(self, X):
        self.input = X.copy()
        N, C, H, W = X.shape
        padding_x = np.pad(X, ((0, 0), (0, 0), (self.padding, self.padding), (self.padding, self.padding)))
        output_H = (H + 2 * self.padding - self.filter_size) // self.stride + 1
        output_W = (W + 2 * self.padding - self.filter_size) // self.stride + 1
        self.output = np.zeros((N, self.out_channels, output_H, output_W))
        for h in range(output_H):
            for w in range(output_W):
                tmp_x = padding_x[:, :, h * self.stride:h * self.stride + self.filter_size, w * self.stride:w * self.stride + self.filter_size].reshape((N, 1, self.in_channels, self.filter_size, self.filter_size))
                tmp_W = self.W['value'].reshape((1, self.out_channels, self.in_channels, self.filter_size, self.filter_size))
```

```
                self.output[:, :, h, w] = np.sum(tmp_x * tmp_W, axis=(2, 3, 4)) + self.b['value']
        return self.output

    def backward(self, back_grad):
        N, C, H, W = self.input.shape
        padding_x = np.pad(self.input, ((0, 0), (0, 0), (self.padding, self.padding), (self.padding, self.padding)))
        output_H = (H + 2 * self.padding - self.filter_size) // self.stride + 1
        output_W = (W + 2 * self.padding - self.filter_size) // self.stride + 1
        self.W['grad'] = np.zeros((self.out_channels, self.in_channels, self.filter_size, self.filter_size))
        self.b['grad'] = np.zeros(self.out_channels)
        grad = np.zeros_like(padding_x)
        for h in range(output_H):
            for w in range(output_W):
                tmp_back_grad = back_grad[:, :, h, w].reshape((N, 1, 1, 1, self.out_channels))
                tmp_W = self.W['value'].transpose((1, 2, 3, 0)).reshape((1, self.in_channels, self.filter_size, self.filter_size, self.out_channels))
                grad[:, :, h * self.stride:h * self.stride + self.filter_size, w * self.stride:w * self.stride + self.filter_size] += np.sum(tmp_back_grad * tmp_W, axis=4)
                tmp_back_grad = back_grad[:, :, h, w].T.reshape((self.out_channels, 1, 1, 1, N))
                tmp_x = padding_x[:, :, 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(back_grad[:, :, h, w], axis=0)
        grad = grad[:, :, self.padding:self.padding + H, self.padding:self.padding + W]
        return grad
```

池化层：

池化层采用了最大池化的方式，即通过滤波器筛选出最大的元素作为池化结果：

```
class MaxPooling(Layer):
    def __init__(self, pool_size=None):
        super().__init__()
        if pool_size is None:
            pool_size = [2, 2]
        self.input = None
        self.output = None
        self.pool_size = pool_size
```

```

    def forward(self, X):
        h_size = self.pool_size[0]
        w_size = self.pool_size[1]
        N, C, H, W = X.shape
        output_H = H // h_size
        output_W = W // w_size
        self.input = X.copy()
        self.output = np.zeros((N, C, output_H, output_W))
        for h in range(output_H):
            for w in range(output_W):
                self.output[:, :, h, w] = np.max(X[:, :, h*h_size:(h+1)*h_size, w*w_size:(w+1)*w_size], axis=(2, 3))
        return self.output

    def backward(self, back_grad):
        h_size = self.pool_size[0]
        w_size = self.pool_size[1]
        N, C, H, W = self.input.shape
        output_H = H // h_size
        output_W = W // w_size
        grad = np.zeros_like(self.input)
        for h in range(output_H):
            for w in range(output_W):
                tmp_x = self.input[:, :, h*h_size:(h+1)*h_size, w*w_size:(w+1)*w_size].reshape((N, C, -1))
                mask = np.zeros((N, C, h_size*w_size))
                mask[np.arange(N)[:, None], np.arange(C)[None, :], np.argmax(tmp_x, axis=2)] = 1
                grad[:, :, h*h_size:(h+1)*h_size, w*w_size:(w+1)*w_size] = mask.reshape((N, C, h_size, w_size)) * back_grad[:, :, h, w][:, :, None, None]
        return grad

```

全连接层：

```
class FullyConnected(Layer):
    def __init__(self, input_size, output_size):
        super().__init__()
        self.input = None
        self.output = None
        self.input_size = input_size
        self.output_size = output_size
        self.W = {'value': np.random.normal(scale=1e-3, size=(input_size, output_size)),
```

```
                  'grad': np.zeros((input_size, output_size))}
        self.b = {'value': np.zeros(output_size),
                  'grad': np.zeros(output_size)}

    def forward(self, X):
        self.input = X.copy()
        self.output = np.dot(X, self.W['value']) + self.b['value']
        return self.output

    def backward(self, back_grad):
        grad = np.dot(back_grad, self.W['value'].T)
        self.W['grad'] = np.dot(self.input.T, back_grad)
        self.b['grad'] = np.sum(back_grad, axis=0)
        return grad
```

2. 激活函数

此外，还定义了本次实验使用的激活函数ReLu。由于sigmod函数包含指数运算，会降低训练速度；另一方面，sigmoid 函数在输入绝对值较大时存在严重的梯度消失问题。

$$\mathrm{ReLu}(x)=\begin{cases} x, x \geq 0 \\ 0, x < 0 \end{cases}$$

上面的公式为前向计算时使用的公式，反向传播使用的公式为：

$$\mathrm{grad}(x)=\begin{cases} \mathrm{grad}_{back} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

代码实现如下：

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

    def forward(self, X):
        self.input = X.copy()
        self.output = np.maximum(0, X)
        return self.output

    def backward(self, back_grad):
        grad = self.input > 0
        return back_grad * grad
```

3. LeNet5模型

在实验原理部分，已经介绍了LeNet5网络的基本结构，再加上前面定义的Layers，基于这些可以根据定义写出LeNet5模型代码实现部分：

```python
class LeNet5:
    def __init__(self):
        self.conv1 = layers.Conv(1, 6, 5)
        self.relu1 = layers.ReLu()
        self.pool1 = layers.MaxPooling((2, 2))
        self.conv2 = layers.Conv(6, 16, 5)
        self.relu2 = layers.ReLu()
        self.pool2 = layers.MaxPooling((2, 2))
        self.fc1 = layers.FullyConnected(16 * 5 * 5, 120)
        self.relu3 = layers.ReLu()
        self.fc2 = layers.FullyConnected(120, 84)
        self.relu4 = layers.ReLu()
        self.fc3 = layers.FullyConnected(84, 10)

    def get_params(self):
        return [self.conv1.W, self.conv1.b, self.conv2.W, self.conv2.b, self.fc1.W, self.fc1.b, self.fc2.W, self.fc2.b, self.fc3.W, self.fc3.b]

    def set_params(self, params):
        self.conv1.weights = params[0]
        self.conv1.biases = params[1]
        self.conv2.weights = params[2]
        self.conv2.biases = params[3]
        self.fc1.W = params[4]
        self.fc1.b = params[5]
        self.fc2.W = params[6]
        self.fc2.b = params[7]
        self.fc3.W = params[8]
        self.fc3.b = params[9]

    def forward(self, X):
        X = self.conv1.forward(X)
        X = self.relu1.forward(X)
        X = self.pool1.forward(X)
        X = self.conv2.forward(X)
        X = self.relu2.forward(X)
        X = self.pool2.forward(X)
        X = X.reshape(X.shape[0], -1)
        X = self.fc1.forward(X)
        X = self.relu3.forward(X)
        X = self.fc2.forward(X)
        X = self.relu4.forward(X)
        X = self.fc3.forward(X)
```

```
        return X

    def backward(self, grad):
        grad = self.fc3.backward(grad)
        grad = self.relu4.backward(grad)
        grad = self.fc2.backward(grad)
        grad = self.relu3.backward(grad)
        grad = self.fc1.backward(grad)
        grad = grad.reshape(grad.shape[0], 16, 5, 5)
        grad = self.pool2.backward(grad)
        grad = self.relu2.backward(grad)
        grad = self.conv2.backward(grad)
        grad = self.pool1.backward(grad)
        grad = self.relu1.backward(grad)
        grad = self.conv1.backward(grad)
```

4. 损失函数

使用交叉熵损失函数：

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

4.2.3 Adam算法优化器

实验开始时，采用的是普通的随机梯度下降法。但运行时间过长，大约经过1200-1600次迭代才能达到收敛。故查阅资料，了解到可以使用Adam算法替代普通的随机梯度下降法，故采用该种方法；

论文"ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION"（参考Diederik P. Kingma, Jimmy Ba：《Adam: A Method for Stochastic Optimization》）提出了Adam 优化算法（adaptive moment estimation)，用于解决机器学习中的大数据量，高特征纬度的优化问题。他集合了两个流行算法“Adagrad”（用于处理稀疏的梯度）和“RMSPro”（处理非稳态数据）。并且Adam算法仅需要少量的内存。论文中给出的Adam算法伪代码如下：

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

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 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd 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)

具体实现如下：

```
class Adam:
    def __init__(self, params, lr=1e-3, beta1=0.9, beta2=0.999):
        self.lr = lr
        self.beta1 = beta1
        self.beta2 = beta2
        self.iter = 0
        self.m = None
        self.v = None
        self.params_grad = params

    def step(self):
        if self.m is None:
            self.m, self.v = [], []
            for param in self.params_grad:
                self.m.append(np.zeros_like(param['value']))
                self.v.append(np.zeros_like(param['grad']))

        self.iter += 1
        lr_t = self.lr * np.sqrt(1.0 - self.beta2 ** self.iter) / (1.0 - self.beta1 ** self.iter)

        for i in range(len(self.params_grad)):
```

```python
            self.m[i] += (1 - self.beta1) * (self.params_grad[i]['grad'] - self.m[i])
            self.v[i] += (1 - self.beta2) * (self.params_grad[i]['grad'] ** 2 - self.v[i])
            self.params_grad[i]['value'] -= lr_t * self.m[i] / (np.sqrt(self.v[i]) + 1e-7)
```

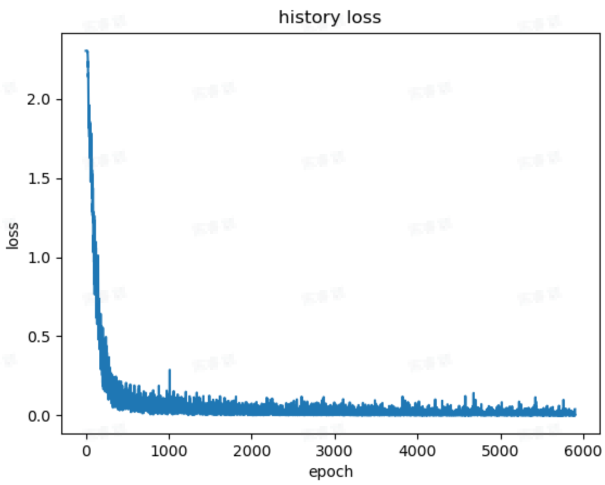
5. 实验验证

5.1 超参数设置

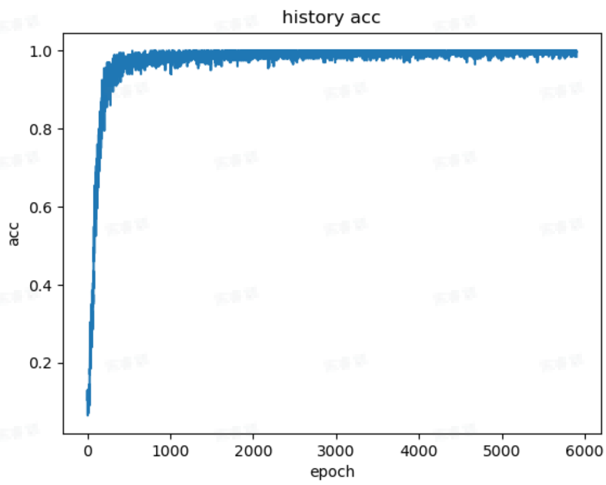
```python
batch_size = 200
epochs = 10
learning_rate = 1e-3
```

5.2 实验结果

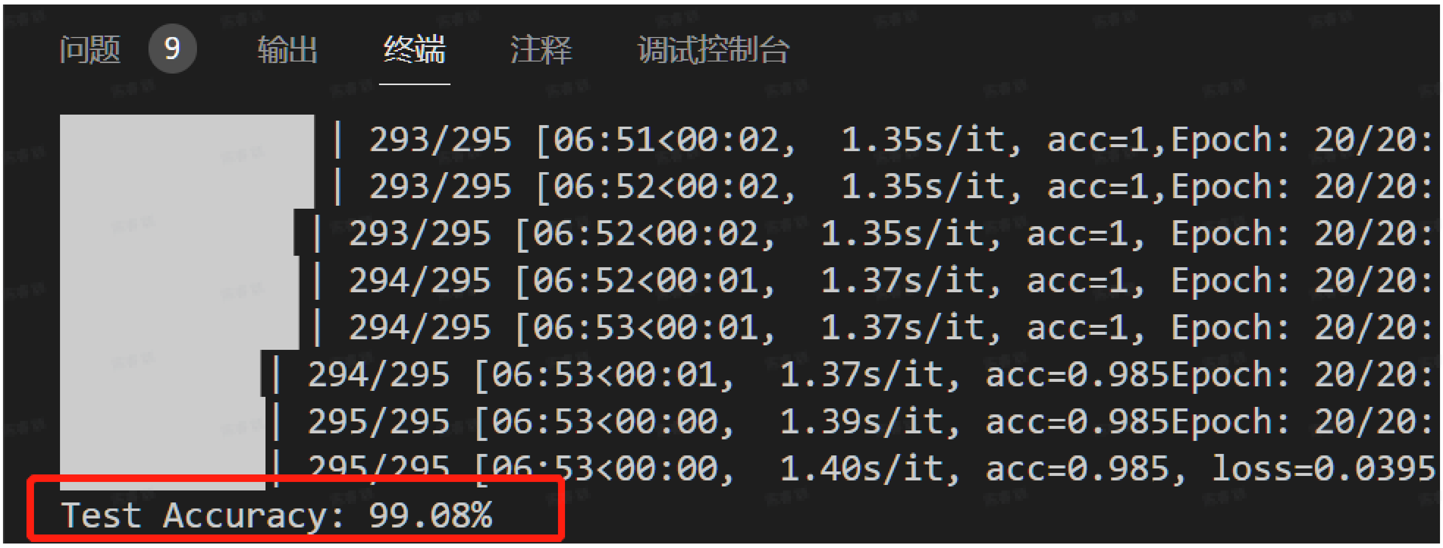
历史的损失值：



历史准确率：



上图实验结果可以看出，在epoch=250左右，损失值下降非常快。在epoch>500后，损失值基本保持稳定。准确率在epoch>500后达到了一个较高的稳定值，大约在98~99%。



如图，运行最终结果的准确率为99.08%。