实验二

多层感知机实现手写数字识别

2022年9月20日

1. SoftmaxCrossEntropyLoss

Forward函数：根据softmax计算损失

```python
input_size = len(logit)
shift = logit - np.max(logit, axis=1, keepdims=True)  # 减去最大值，避免上溢或下溢
self.softmax_score = np.exp(shift) / np.sum(np.exp(shift), axis=1, keepdims=True)  # 计算softmax
self.loss = np.sum(-gt * np.log(self.softmax_score)) / input_size  # 计算损失
self.acc = np.sum(np.argmax(gt, axis=1) == np.argmax(self.softmax_score, axis=1)) / input_size  # 计算正确率
self.labels = gt

return self.loss
```

Backward函数：返回梯度，

$$\boldsymbol{\delta}^{(L-1)} = \underbrace{\boldsymbol{y}^{(L)}}_{\text{Softmax输出}} - \boldsymbol{t}$$

```python
def backward(self):

    ############################################################################
    # TODO:
    # 计算并返回梯度(与logit具有同样的尺寸)
    ############################################################################

    return self.softmax_score - self.labels
```

2. EuclideanLossLayer

Forward函数：用欧式距离作为损失

```python
input_size = len(logit)
self.loss = np.sqrt(np.sum((logit - gt)**2)) / 2      # 计算欧式距离
self.acc = np.sum(np.argmax(gt, axis=1) == np.argmax(logit, axis=1)) / input_size
self.gt = gt
self.logit = logit

return self.loss
```

Backward函数：返回梯度，

$$\boldsymbol{\delta}^{(L)} = \boldsymbol{y}^{(L)} - \boldsymbol{t}$$

```python
def backward(self):

    ############################################################################
    # TODO:
    # 计算并返回梯度(与logit具有同样的尺寸)
    ############################################################################
    return self.logit - self.gt
```

3. FCLayer

Forward函数：计算wx+b

```
self.X = Input
return np.dot(Input, self.W) + self.b
```

Backward函数：计算gradW 和 gradB，并返回delta，

$$\frac{\partial E^{(n)}}{\partial \boldsymbol{W}^{(l)}} = \boldsymbol{\delta}^{(l)} (\boldsymbol{y}^{(l-1)})^\top, \quad \frac{\partial E^{(n)}}{\partial \boldsymbol{b}^{(l)}} = \boldsymbol{\delta}^{(l)}, \quad \boldsymbol{\delta}^{(l-1)} = (\boldsymbol{W}^{(l)})^\top \boldsymbol{\delta}^{(l)}$$

```
def backward(self, delta):
    # 输入的delta由下一层计算得到
    ############################################################################
    # TODO:
    # 根据delta计算梯度
    ############################################################################
    self.grad_W = np.dot(self.X.T, delta)
    self.grad_b = np.mean(delta, axis=0)
    return np.dot(delta, self.W.T)
```

4. FCLayer

Forward函数：计算ReLu

```
self.df = np.where(Input < 0, 0, 1)
return np.where(Input < 0, 0, Input)
```

Backward函数：返回delta

$$\boldsymbol{\delta}^{(l-1)} = \boldsymbol{\delta}^{(l)} \odot \boldsymbol{f}'(\boldsymbol{y}^{(l-1)}) \text{ 其中 } \quad f'(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{else.} \end{cases}$$

```
def backward(self, delta):

    ############################################################################
    # TODO:
    # 根据delta计算梯度
    ############################################################################
    return delta * self.df
```

5. SigmoidLayer

Forward函数：计算Sigmoid

```
sigmoid = lambda x: 1 / (1 + np.exp(-x))
y = sigmoid(Input)
self.df = y * (1 - y)
return y
```

Backward函数：返回delta

$$\boldsymbol{\delta}^{(l-1)} = \boldsymbol{\delta}^{(l)} \odot \boldsymbol{f}'(\boldsymbol{y}^{(l-1)}) \text{ 其中 } \quad f'(x) = f(x)(1 - f(x))$$

```
def backward(self, delta):

    ############################################################################
    # TODO:
    # 根据delta计算梯度
    ############################################################################
    return delta * self.df
```

6. Optimizer

与上一次实验的代码相似

```
layer.diff_W = self.weightDecay * layer.diff_W + self.learningRate * layer.grad_W
layer.diff_b = self.weightDecay * layer.diff_b + self.learningRate * layer.grad_b
# Weight update
layer.W += -layer.diff_W
layer.b += -layer.diff_b
```

3. 实验结果

3.1 欧式距离和sigmoid

1.1 使用欧式距离损失和Sigmoid激活函数训练多层感知机

训练带有一个隐含层且神经元个数为128的多层感知机，使用欧式距离损失和Sigmoid激活函数.

TODO

执行以下代码之前，请完成 **layers/fc_layer.py** 和 **layers/sigmoid_layer.py**.

In [7]:
```
from layers import FCLayer, SigmoidLayer

sigmoidMLP = Network()
# 使用FCLayer和SigmoidLayer构建多层感知机
# 128为隐含层的神经元数目
sigmoidMLP.add(FCLayer(784, 128))
sigmoidMLP.add(SigmoidLayer())
sigmoidMLP.add(FCLayer(128, 10))
```

In [8]:
```
sigmoidMLP, sigmoid_loss, sigmoid_acc = train(sigmoidMLP, criterion, sgd, data_train, max_epoch, batch_size, disp_f
```

```
Epoch [18]      Average training loss 1.9794    Average training accuracy 0.9349
Epoch [18]      Average validation loss 1.8081  Average validation accuracy 0.9530

Epoch [19][20]  Batch [0][550]  Training Loss 1.8303    Accuracy 0.9500
Epoch [19][20]  Batch [50][550]         Training Loss 1.9026    Accuracy 0.9439
Epoch [19][20]  Batch [100][550]        Training Loss 1.9287    Accuracy 0.9394
Epoch [19][20]  Batch [150][550]        Training Loss 1.9558    Accuracy 0.9370
Epoch [19][20]  Batch [200][550]        Training Loss 1.9502    Accuracy 0.9376
Epoch [19][20]  Batch [250][550]        Training Loss 1.9474    Accuracy 0.9376
Epoch [19][20]  Batch [300][550]        Training Loss 1.9513    Accuracy 0.9376
Epoch [19][20]  Batch [350][550]        Training Loss 1.9551    Accuracy 0.9371
Epoch [19][20]  Batch [400][550]        Training Loss 1.9587    Accuracy 0.9369
Epoch [19][20]  Batch [450][550]        Training Loss 1.9608    Accuracy 0.9366
Epoch [19][20]  Batch [500][550]        Training Loss 1.9661    Accuracy 0.9362

Epoch [19]      Average training loss 1.9641    Average training accuracy 0.9363
Epoch [19]      Average validation loss 1.7954  Average validation accuracy 0.9538
```

In [9]:
```
test(sigmoidMLP, criterion, data_test, batch_size, disp_freq)
```

```
Testing...
The test accuracy is 0.9388.
```

预测的正确率为0.9388

3.2 欧式距离和ReLU

预测的正确率为0.9590

1.2 使用欧式距离损失和ReLU激活函数训练多层感知机

训练带有一个隐含层且神经元个数为128的多层感知机，使用欧式距离损失和ReLU激活函数.

TODO

执行以下代码之前，请完成 **layers/relu_layer.py**.

In [10]:
```
from layers import ReLULayer

reluMLP = Network()
# 使用FCLayer和ReLULayer构建多层感知机
reluMLP.add(FCLayer(784, 128))
reluMLP.add(ReLULayer())
reluMLP.add(FCLayer(128, 10))
```

In [11]:
```
reluMLP, relu_loss, relu_acc = train(reluMLP, criterion, sgd, data_train, max_epoch, batch_size, disp_freq)
```

```
Epoch [18]       Average training loss 1.7017    Average training accuracy 0.9621
Epoch [18]       Average validation loss 1.6684  Average validation accuracy 0.9674

Epoch [19][20]   Batch [0][550]   Training Loss 1.6339    Accuracy 0.9800
Epoch [19][20]   Batch [50][550]          Training Loss 1.6365    Accuracy 0.9680
Epoch [19][20]   Batch [100][550]         Training Loss 1.6666    Accuracy 0.9645
Epoch [19][20]   Batch [150][550]         Training Loss 1.6826    Accuracy 0.9636
Epoch [19][20]   Batch [200][550]         Training Loss 1.6778    Accuracy 0.9646
Epoch [19][20]   Batch [250][550]         Training Loss 1.6734    Accuracy 0.9649
Epoch [19][20]   Batch [300][550]         Training Loss 1.6843    Accuracy 0.9638
Epoch [19][20]   Batch [350][550]         Training Loss 1.6854    Accuracy 0.9636
Epoch [19][20]   Batch [400][550]         Training Loss 1.6879    Accuracy 0.9635
Epoch [19][20]   Batch [450][550]         Training Loss 1.6879    Accuracy 0.9635
Epoch [19][20]   Batch [500][550]         Training Loss 1.6935    Accuracy 0.9627

Epoch [19]       Average training loss 1.6903    Average training accuracy 0.9629
Epoch [19]       Average validation loss 1.6607  Average validation accuracy 0.9678
```

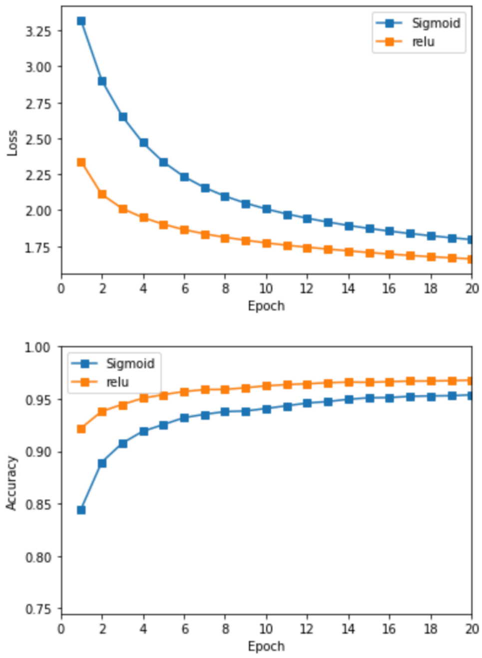
In [12]:
```
test(reluMLP, criterion, data_test, batch_size, disp_freq)
```

```
Testing...
The test accuracy is 0.9590.
```

可以看出用ReLu作为激活函数的收敛速度更快，并且准确率也更高。

绘制曲线

In [13]:
```
plot_loss_and_acc({'Sigmoid': [sigmoid_loss, sigmoid_acc],
                   'relu': [relu_loss, relu_acc]})
```



3.3 Softmax和Sigmoid

预测正确率为0.9507

2.1 使用Softmax交叉熵损失和Sigmoid激活函数训练多层感知机

训练带有一个隐含层且神经元个数为128的多层感知机，使用Softmax交叉熵损失和Sigmoid激活函数.

In [15]:
```
sigmoidMLP = Network()
# 使用FCLayer和SigmoidLayer构建多层感知机
# 128为隐含层的神经元数目
sigmoidMLP.add(FCLayer(784, 128))
sigmoidMLP.add(SigmoidLayer())
sigmoidMLP.add(FCLayer(128, 10))
```

训练

In [16]:
```
sigmoidMLP, sigmoid_loss, sigmoid_acc = train(sigmoidMLP, criterion, sgd, data_train, max_epoch, batch_size, disp_f
```

```
Epoch [0][20]	 Batch [0][550]	 Training Loss 2.8934	 Accuracy 0.1400
Epoch [0][20]	 Batch [50][550]	 Training Loss 1.8810	 Accuracy 0.5108
Epoch [0][20]	 Batch [100][550]	 Training Loss 1.5511	 Accuracy 0.6375
Epoch [0][20]	 Batch [150][550]	 Training Loss 1.3478	 Accuracy 0.6901
Epoch [0][20]	 Batch [200][550]	 Training Loss 1.2056	 Accuracy 0.7251
Epoch [0][20]	 Batch [250][550]	 Training Loss 1.0961	 Accuracy 0.7507
Epoch [0][20]	 Batch [300][550]	 Training Loss 1.0136	 Accuracy 0.7690
Epoch [0][20]	 Batch [350][550]	 Training Loss 0.9538	 Accuracy 0.7817
Epoch [0][20]	 Batch [400][550]	 Training Loss 0.9001	 Accuracy 0.7925
Epoch [0][20]	 Batch [450][550]	 Training Loss 0.8564	 Accuracy 0.8010
Epoch [0][20]	 Batch [500][550]	 Training Loss 0.8196	 Accuracy 0.8083

Epoch [0]	 Average training loss 0.7877	 Average training accuracy 0.8147
Epoch [0]	 Average validation loss 0.3675	 Average validation accuracy 0.9114

Epoch [1][20]	 Batch [0][550]	 Training Loss 0.4342	 Accuracy 0.9000
Epoch [1][20]	 Batch [50][550]	 Training Loss 0.4168	 Accuracy 0.8906
Epoch [1][20]	 Batch [100][550]	 Training Loss 0.4200	 Accuracy 0.8896
Epoch [1][20]	 Batch [150][550]	 Training Loss 0.4267	 Accuracy 0.8861
```

测试

In [17]:
```
test(sigmoidMLP, criterion, data_test, batch_size, disp_freq)
```

```
Testing...
The test accuracy is 0.9507.
```

3.4 Softmax和ReLU

预测正确率为0.9760

2.2 使用Softmax交叉熵损失和ReLU激活函数训练多层感知机

训练带有一个隐含层且神经元个数为128的多层感知机，使用Softmax交叉熵损失和ReLU激活函数.

In [18]:
```
reluMLP = Network()
# 使用FCLayer和SigmoidLayer构建多层感知机
# 128为隐含层的神经元数目
reluMLP.add(FCLayer(784, 128))
reluMLP.add(ReLULayer())
reluMLP.add(FCLayer(128, 10))
```

In [19]:
```
reluMLP, relu_loss, relu_acc = train(reluMLP, criterion, sgd, data_train, max_epoch, batch_size, disp_freq)
```

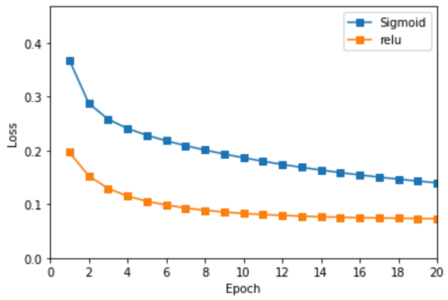
```
Epoch [18]	 Average training loss 0.0375	 Average training accuracy 0.9909
Epoch [18]	 Average validation loss 0.0734	 Average validation accuracy 0.9790

Epoch [19][20]	 Batch [0][550]	 Training Loss 0.0349	 Accuracy 0.9900
Epoch [19][20]	 Batch [50][550]	 Training Loss 0.0347	 Accuracy 0.9918
Epoch [19][20]	 Batch [100][550]	 Training Loss 0.0352	 Accuracy 0.9916
Epoch [19][20]	 Batch [150][550]	 Training Loss 0.0337	 Accuracy 0.9919
Epoch [19][20]	 Batch [200][550]	 Training Loss 0.0338	 Accuracy 0.9917
Epoch [19][20]	 Batch [250][550]	 Training Loss 0.0335	 Accuracy 0.9922
Epoch [19][20]	 Batch [300][550]	 Training Loss 0.0344	 Accuracy 0.9921
Epoch [19][20]	 Batch [350][550]	 Training Loss 0.0344	 Accuracy 0.9921
Epoch [19][20]	 Batch [400][550]	 Training Loss 0.0347	 Accuracy 0.9920
Epoch [19][20]	 Batch [450][550]	 Training Loss 0.0350	 Accuracy 0.9918
Epoch [19][20]	 Batch [500][550]	 Training Loss 0.0354	 Accuracy 0.9916

Epoch [19]	 Average training loss 0.0352	 Average training accuracy 0.9916
Epoch [19]	 Average validation loss 0.0731	 Average validation accuracy 0.9792
```

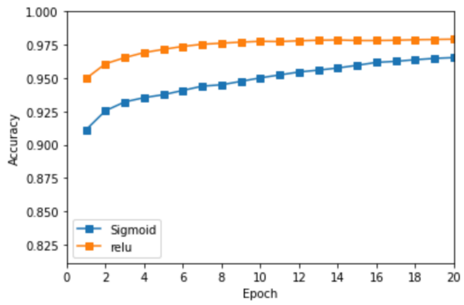
In [20]:
```
test(reluMLP, criterion, data_test, batch_size, disp_freq)
```

```
Testing...
The test accuracy is 0.9760.
```

可以看出，ReLU作为激活函数的收敛速度更快，并且准确率更高

```
In [21]: plot_loss_and_acc({'Sigmoid': [sigmoid_loss, sigmoid_acc],
                            'relu': [relu_loss, relu_acc]})
```





从上面四个模型中，可以看出用Softmax作为损失函数和用ReLU作为激活函数的效果最好。

3.5 两个隐含层

在输入层和输出层中间添加了128结点和64结点的隐含层，预测正确率为0.9776

具有两层隐含层的多层感知机

接下来，根据案例要求，还需要完成**构造具有两个隐含层的多层感知机，自行选取合适的激活函数和损失函数，与只有一个隐含层的结果相比较**.

注意: 请在下方插入新的代码块，不要直接修改上面的代码.

```
In [22]: twoLayerMLP = Network()
         twoLayerMLP.add(FCLayer(784, 128))
         twoLayerMLP.add(ReLULayer())
         twoLayerMLP.add(FCLayer(128, 64))
         twoLayerMLP.add(ReLULayer())
         twoLayerMLP.add(FCLayer(64, 10))
```

```
In [23]: twoLayerMLP, tl_loss, tl_acc = train(twoLayerMLP, criterion, sgd, data_train, max_epoch, batch_size, disp_freq)

         Epoch [18]        Average training loss 0.0097    Average training accuracy 0.9989
         Epoch [18]        Average validation loss 0.0847  Average validation accuracy 0.9792

         Epoch [19][20]    Batch [0][550]   Training Loss 0.0046    Accuracy 1.0000
         Epoch [19][20]    Batch [50][550]          Training Loss 0.0091    Accuracy 0.9992
         Epoch [19][20]    Batch [100][550]         Training Loss 0.0100    Accuracy 0.9991
         Epoch [19][20]    Batch [150][550]         Training Loss 0.0090    Accuracy 0.9993
         Epoch [19][20]    Batch [200][550]         Training Loss 0.0088    Accuracy 0.9993
         Epoch [19][20]    Batch [250][550]         Training Loss 0.0084    Accuracy 0.9994
         Epoch [19][20]    Batch [300][550]         Training Loss 0.0084    Accuracy 0.9993
         Epoch [19][20]    Batch [350][550]         Training Loss 0.0084    Accuracy 0.9993
         Epoch [19][20]    Batch [400][550]         Training Loss 0.0087    Accuracy 0.9992
         Epoch [19][20]    Batch [450][550]         Training Loss 0.0085    Accuracy 0.9993
         Epoch [19][20]    Batch [500][550]         Training Loss 0.0086    Accuracy 0.9992

         Epoch [19]        Average training loss 0.0085    Average training accuracy 0.9992
         Epoch [19]        Average validation loss 0.0853  Average validation accuracy 0.9790
```

```
In [24]: test(twoLayerMLP, criterion, data_test, batch_size, disp_freq)

         Testing...
         The test accuracy is 0.9776.
```

对比可以看出，两个隐含层相较于一个隐含层前面收敛的速度更快，后面loss基本不变了，最终两者的预测正确率也相差不大。

```
In [25]: plot_loss_and_acc({'2-layer': [tl_loss, tl_acc],
                            'relu': [relu_loss, relu_acc],
                            'Sigmoid': [sigmoid_loss, sigmoid_acc]})
```