机器学习实验报告

Exercise1-SoftmaxRegression

学　　院：网络安全学院

专　　业：信息安全

学　　号：2111252

姓　　名：李佳豪

班　　级：信安一班

1 实验目的

在这个练习中，需要训练一个分类器来完成对 MNIST 数据集中 0-9 10 个手写数字的分类。

2 实验环境

Python

3 实验原理

3.1 `Softmax Function`

Softmax 函数是逻辑回归在多类别分类中的推广。在二元逻辑回归中，我们使用 sigmoid 函数来预测两个类别的概率，而在多类别分类问题中，我们使用 softmax 函数来预测多个类别的概率。

在数学上，softmax 函数定义为：

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } i = 1, \ldots, K \tag{1}$$

K 是分类的类别数，

这里 z 是模型对一个实例输出的原始预测向量，通常称为 logits，它包含了每个类别的分数。Softmax 函数将这些 logits 转换为概率分布，即对于 K 个类别，每个类别 i 的预测概率 p_i 由其对应的 logit z_i 通过 softmax 函数计算得到。

而我们为什么不**直接使用这些 logits 数值（比如设置一个阈值，如选择具有最大的 logits 数值得到输出**呢，但我们希望模型输出的 y_i 可以视为属于类 i 的概率，然后选择具有最大输出值的类别 `argmax` y_i 作为预测结果，直接输出的 logits 数值仅仅是由一个线性层输出，他们不仅不满足概率和为 1 的特性，甚至可能出现负值。这都是不合理的。

而关注这个 softmax 函数，通过对每个 logit 使用指数函数，softmax 确保输出是正的，并且较大的 logits 会对应较大的概率，最终输出是一个概率分布，所有类别的概率之和为 1。并且这种指数的计算，使得 logits 之间相对差异的放大，即如果一个 logit 比其他的都要大，`Softmax` 会使其对应的概率远大于其他类别。

总之，我们需要一个训练的目标函数，来激励模型精准地估计概率，`Softmax` 函数能够将未规范化的预测变换为非负数并且总和为 1，同时让模型保持**可导**的性质

3.2 Cross-Entropy Loss

交叉熵在多分类问题中, cross-entropy loss 如下定义

$$L(\mathbf{y}, \mathbf{p}) = -\sum_{i=1}^{K} y_i \log(p_i) \tag{2}$$

y_i 是一个真实的标签结果 (one-hot 编码) 、p_i 是对于第 i 个样本的预测结果。

3.3 Gradient

对于线性层输出的 o 和 softmax 输出 p，对 o 求导结果如下，详细推导在下一小节。

$$\nabla_{\mathbf{z}} L(\mathbf{y}, \mathbf{p}) = \mathbf{p} - \mathbf{y} \tag{3}$$

3.4 梯度完整推导

1. 假设有一个样本 $\mathbf{x}$，其维度为 n，在一个具有 k 个分类通道的模型中，每个分类通道 i 的 logits o_i 的计算可以表示为样本 x 的每一个特征 x_j 与相应权重 w_{ji} 的乘积之和，再加上偏置项 b_i。具体如下：

$$\begin{aligned}
o_1 &= x_1 w_{11} + x_2 w_{21} + x_3 w_{31} + \cdots + x_n w_{n1} + b_1 \\
o_2 &= x_1 w_{12} + x_2 w_{22} + x_3 w_{32} + \cdots + x_n w_{n2} + b_2 \\
o_3 &= x_1 w_{13} + x_2 w_{23} + x_3 w_{33} + \cdots + x_n w_{n3} + b_3 \\
&\vdots \\
o_k &= x_1 w_{1k} + x_2 w_{2k} + x_3 w_{3k} + \cdots + x_n w_{nk} + b_k
\end{aligned}$$

2. 接着使用 softmax 转换为概率分布，对于每个分类通道 i，概率 p_i 通过以下方式计算：

$$p_i = \frac{e^{o_i}}{\sum_{j=1}^{k} e^{o_j}} \tag{4}$$

其中，e^{o_i} 是 o_i 的指数，分母是所有类别指数值的总和，确保所有概率值加起来等于 1。

3. 对于每个样本，交叉熵损失 L 可以通过以下公式计算：

$$L = -\sum_{i=1}^{K} y_i \log(p_i) \tag{5}$$

其中，y_i 是真实标签的 one-hot 编码，当类别 i 是正确的类别时为 1，否则为 0；p_i 是 softmax 函数得到的预测概率。

4. 最后推导计算梯度，

对于一个样本的真实标签向量 $\mathbf{y}$ 和通过 softmax 函数得到的预测概率 $\widehat{\mathbf{y}}$，交叉熵损失函数 $\ell(\mathbf{y}, \widehat{\mathbf{y}})$ 可以表示为：

$$\ell(\mathbf{y}, \widehat{\mathbf{y}}) = -\sum_{j=1}^{q} y_j \log\left(\frac{\exp(o_j)}{\sum_{k=1}^{q} \exp(o_k)}\right) \tag{6}$$

展开 softmax 函数中的 log 项得到：

$$\ell(\mathbf{y}, \widehat{\mathbf{y}}) = -\sum_{j=1}^{q} y_j \left(\log(\exp(o_j)) - \log\left(\sum_{k=1}^{q} \exp(o_k)\right)\right) \tag{7}$$

由于 y_j 是 0 或 1，且在每个求和中只有一个 y_j 是 1，进一步简化得到：

$$\ell(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{j=1}^{q} y_j o_j + \log\left(\sum_{k=1}^{q} \exp(o_k)\right) \tag{8}$$

梯度计算为：

$$\frac{\partial \ell(\mathbf{y}, \hat{\mathbf{y}})}{\partial o_j} = \text{softmax}(o_j) - y_j \tag{9}$$

因此, 对于逻辑回归的梯度，最后可以转换成为下面！

$$\frac{\partial \ell(\mathbf{y}, \hat{\mathbf{y}})}{\partial o_j} = \frac{\exp(o_j)}{\sum_{k=1}^{q} \exp(o_k)} - y_j = \text{softmax}(o_j) - y_j \tag{10}$$

接下来根据链式法则，可以看到交叉熵对线性层参数的梯度已经很好计算了，只需上式乘线性层的导数值–x_i

$$\frac{\partial J}{\partial w_i} = \frac{\partial J}{\partial o_i} \cdot \frac{\partial o_i}{\partial w_i} \tag{11}$$

4 使用 for 循环计算梯度

4.1 核心代码

- 关于 softmax 在代码中的实现，我们已经使用一个线性层对每一个样本得到了一个 10 维度的 logits 预测值,现在要对其每一个值进行指数运算,之后求和的操作,使用类似于 `np.exp(logits) / np.sum(np.exp(logits))` 计算，numpy 库对这种浮点数运算的精度和速度都有优化。
- 而在梯度计算中，下面的代码并没有考虑 batch 的概念，在一个 epoch 中，分别将每一个样本的梯度矩阵相加，最后除以样本数量，然后对 theta 进行一次更新。
- 最后把每一个 epoch(或者 iteration) 的 loss 和 acc 加入列表，便于最后图形化输出。
- 代码并不复杂，具体见下：

```
def softmax_regression_base(theta, x, y, iters, alpha):
    m, n = x.shape
    k = theta.shape[0]
    loss_ls = []
    acc_train = []
    acc_test = []
    from tqdm import trange
    for iteration in range(iters):
        cost = 0
        acc = 0
        grad = np.zeros_like(theta)

```

```
        for i in range(m):
            # 计算 softmax 概率
            logits = np.dot(theta, x[i])
            probabilities = np.exp(logits) / np.sum(np.exp(logits))

            # 计算损失
            cost -= np.log(probabilities[y[i]])

            # 计算梯度
            for j in range(k):
                indicator = 1 if y[i] == j else 0
                grad[j, :] += (probabilities[j] - indicator) * x[i]



        # 计算平均损失和梯度
        cost /= m
        grad /= m
        acc = get_acc(x,y,theta)
        loss_ls.append(cost.copy())
        acc_train.append(acc)
        # 更新 theta
        theta -= alpha * grad

        print(f"Epoch {iteration}: Loss:{cost} Acc:{acc}")
    plpot(loss_ls,acc_train)
    return theta
```

4.2 结果

表 1: Model Performance Metrics

Learning Rate	iteration	Train Accuracy	Loss	Time (s)	Test Accuracy
0.15	100	87.2%	0.398	685	–
0.50	300	88.4%	0.389	2125	–
0.30	1000	89.88%	0.343	3204	90.83%

总的来说，训练时间较长，且训练效果也仅仅能使得准确率到达 90 左右，而且在迭代次数 1000 时，800 多的时候 loss 已经不再是稳定下降，出现抖动的现象。

注：截图只展示了部分，表格中的空则是这部分数据没有记录。

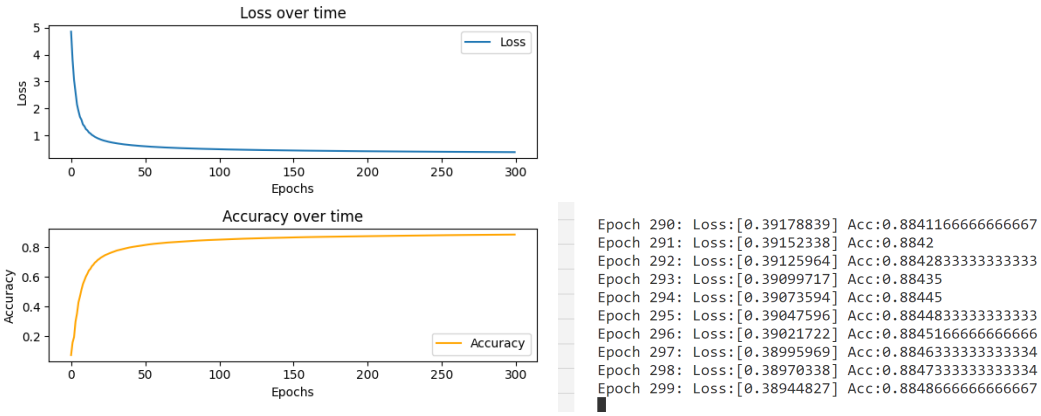


图 4.1: epoch 为 300: 图像 (左) 日志 (右)

图 4.2: epoch 为 1000: 图像 (左) 日志 (右)

5 考虑 batch，利用矩阵直接计算得到一个 batch 的梯度

5.1 使用矩阵计算梯度、loss 和正则

考虑到本次实验需要在 for 循环基础上的优化，可以利用矩阵直接对整个训练样本集直接导出一个 iteration 内的梯度和 loss；这部分梯度、loss 的代码大概如下：

- `loss` 函数，y_hat 是由线性层和 softmax 后得到的一个 n 10 的 logit 数组，n 是样本数量，y 是预测结果 [n*1]；我们巧妙的利用计算 `Cross-Entropy` 时，对于每一个样本的 10 维 y_hat 仅对应 y_i 的那个值参与到最后的 loss 值，其余均为 0，于是有了下面对一个的 y_hat 矩阵计算 loss 的代码。
- $get_gradient$,reg_rate 是正则项权重，为了防止训练后续过拟合，对于正则，这是计算 theta 的一种限制，防止它数值范围过大或者维度过高，当然这里维度我们是固定的，只需要控制其矩阵的数值。

```
1 def loss(y_hat,y):
```

```
    if len(y.shape) >1 :
        y = np.squeeze(y,axis=1)
    #y_hat [6000.10] y[60000]
    p = np.log(y_hat[list(range(len(y))),y]).mean()
    return -p
def get_gradient(x,y,y_hat,theta):
    reg_rate = 0.2
    l = loss(y_hat, y) + reg_rate * np.sum(theta*theta)
    y_hat[list(range(len(y))),y]-=1
    gradient = (y_hat.T) @ x - 2 * reg_rate * theta

    return gradient/len(x) ,l
```

5.2 引入 BatchSize 参数

- 同时，像上一节中一次性将所有样本当作一个 batch 进行一次梯度更新，过于浪费资源，不妨设置 batch 的迭代器，每一个小 batch 内样本进行一次梯度的更新，将提高模型的训练速度。

```
def batch_generator(x,y,batch_size):
    num = len(x)

    for st in range(0,num,batch_size):
        ed = min(st+batch_size,num)
        yield x[st:ed], y[st:ed]
```

5.3 训练代码

最后的训练代码如下了，也就是每一个 batch 进行一次梯度更新，若干 batch 后输出一次日志、计算一次训练集和测试集的准确度、以及加入列表便于后期绘图等等。

```
def softmax_regression(theta, x, y, iters, alpha,batch_size=32):
    # TODO: Do the softmax regression by computing the gradient and
    # the objective function value of every iteration and update the theta
    # softmax: np.exp(x[i])/np.exp(x).sum()
    # loss: min(-y_i * logy_hat)
    # gradient: softmax(xi)-yi
    #x[60000,784] y[60000,10] theta[10,784]
    if len(y.shape)>1:
        y=np.squeeze(y)
    batch_size = 600
    loss_ls = []
    acc_train = []
```

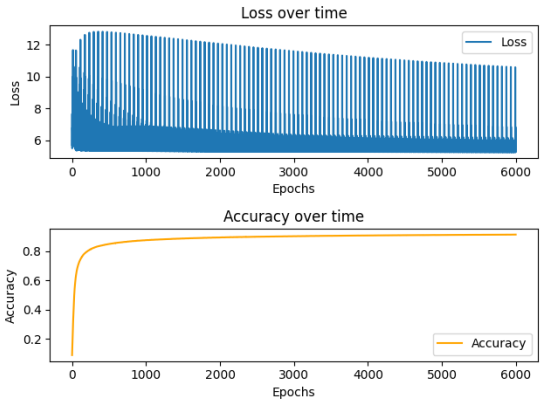
```
    acc_test = []

    print(f"Epoc:-1\n Acc:{get_acc(x, y, theta)}")
    from tqdm import trange
    for epoch in range(iters):
        loss_sum = 0
        genertor = batch_generator(x, y, batch_size)
        for i,(x_,y_) in enumerate(genertor):
            #print(f"epoch:{i}:")
            y_hat = softmax(x_ @ theta.T)
            gradient,l = get_gradient(x_,y_,y_hat,theta)
            loss_sum = loss_sum+l
            theta -= alpha * gradient

            if not (((i+1)*batch_size))%6000:
                loss_ls.append(l.copy())
                acc_train.append(get_acc(x, y, theta))
                #print(f"Loss:{l}")
        print(f"Epoch{epoch}\n Loss:{loss_sum/(60000)} Acc:{get_acc(x,y,theta)}")

    plpot(loss_ls,acc_train)
    return theta,loss_ls,acc_train,acc_test
```

5.4 结果

表 2: Matrix Model Performance Metrics

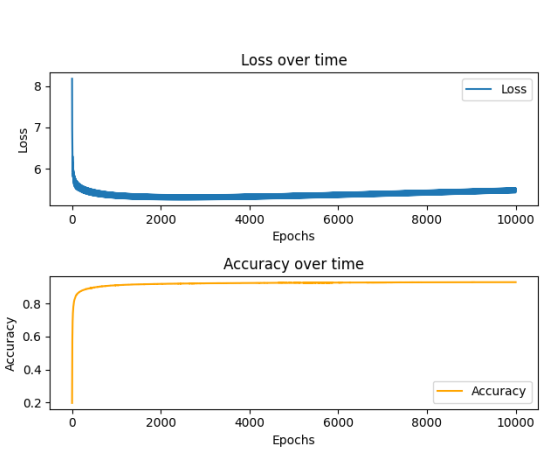
Learning Rate	Iteration	Batch Size	Reg. Penalty	Train Accuracy	Loss	Time (s)	Test Accuracy
0.001	100	64	0.2	85%	5.445	1345.2	83%
0.0005	100	64	0.2	91.201%	5.5624	1315.4	91.35%
0.0001	500	300	0.2	90%	–	1700 左右	–
0.0001	1000	600	0	92.946%	0.009148	2102.79	91.5%

注: 截图较多，只展示了每个表格项的部分，表格中的空则是这部分数据没有记录。




```
Loss:5.568356996118604 Acc:0.91085
Epoch93
Loss:5.567461518950619 Acc:0.91095
Epoch94
Loss:5.566581414020293 Acc:0.9112333333333333
Epoch95
Loss:5.56571639050299 Acc:0.9113166666666667
Epoch96
Loss:5.564866165852053 Acc:0.9114166666666667
Epoch97
Loss:5.5640304654687185 Acc:0.9116
Epoch98
Loss:5.563209022388096 Acc:0.9118833333333334
Epoch99
Loss:5.562401576982373 Acc:0.9120166666666667
Finished training.
Sum of Time:1315.400357723236 s
Test Acc 0.9135
Finished test.
```

图 5.3: Epoch 100 lr 0.0005 曲线 (左) 日志 (右)




```
Epoch988
Loss:0.0091433311849352 21 Acc:0.9294
Epoch989
Loss:0.009143796573394808 Acc:0.9294
Epoch990
Loss:0.009144261984119585 Acc:0.9294
Epoch991
Loss:0.009144727416814567 Acc:0.9294166666666667
Epoch992
Loss:0.00914519287118604 Acc:0.9294166666666667
Epoch993
Loss:0.009145658346941546 Acc:0.9294166666666667
Epoch994
Loss:0.009146123843789884 Acc:0.9294166666666667
Epoch995
Loss:0.009146589361441105 Acc:0.9294166666666667
Epoch996
Loss:0.009147054899606485 Acc:0.9294166666666667
Epoch997
Loss:0.009147520457998557 Acc:0.92945
Epoch998
Loss:0.009147986036331074 Acc:0.9294666666666667
Epoch999
Loss:0.009148451634319016 Acc:0.9294666666666667
Finished training.
Sum of Time:2102.793912410736 s
```

图 5.4: Epoch 1000 曲线 (左) 日志 (右)

图 5.5

6 结果分析、对比

6.1 Batch 带来优势

以第二个模型最后迭代次数 1000 的尝试同 for 模型的结果对比可以知道

在它 epoch 到达 30 次左右时，已经具备了上一节中 **for 模型的水平**，ACC 已经到达 0.89，而上一个模型却需要迭代 1000 次；**时间上说**，上一个模型耗时 3000 余秒的训练成果，在我们加入 batch 后，时间仅仅需要 30/1000 * 2000 大概 1 分钟。

```
Epoch29
 Loss:0.009153802216679508 Acc:0.8858666666666667
Epoch30
 Loss:0.00914780987049881 Acc:0.8868666666666667
Epoch31
 Loss:0.009142029839097789 Acc:0.88765
Epoch32
 Loss:0.009136447878910945 Acc:0.88835
Epoch33
 Loss:0.009131051205513519 Acc:0.8888166666666667
Epoch34
 Loss:0.009125828299904142 Acc:0.8893666666666666
Epoch35
 Loss:0.009120768745572556 Acc:0.8900666666666667
Epoch36
 Loss:0.009115863090733558 Acc:0.8908
Epoch37
 Loss:0.009111102731257295 Acc:0.8914166666666666
Epoch38
```

图 6.6: Epoch30 左右

6.2 模型上限

下面以 iteration1000 的第二个模型为例子，展示不同训练轮次（epoch）下达到的精确度（Accuracy）

表 3: Model Accuracy and Loss at Various Epochs

Epoch	Epoch Increase	Accuracy	Loss
9	-	0.85	0.50
56	47	0.90	0.45
92	36	0.91	0.40
216	124	0.92	0.35
431	215	0.925	0.30
997	566	0.92945	0.25

- 在训练的早期阶段，模型的精确度迅速提高。例如，在仅仅 9 轮训练后，模型的精确度已经达到了 0.85。随着训练轮次的增加，模型继续提高其精确度，但提升速度逐渐放缓。这可以从 56 轮时的 0.90 精确度和 216 轮时的 0.92 精确度看出。
- 在更多的训练轮次之后，模型精确度的提升变得更加微小。例如，从 431 轮的 0.925 到 997 轮的 0.92945，精确度的提升不再那么显著。这表明模型可能接近其性能极限，进一步增加训练轮次带来的精确度提升将非常有限。

6.3 正则项

在使用第二个模型一开始我就加入了正则项，因为看到网上教程都说不加会容易数值上溢，而且容易过拟合，但开始的几次测试，感觉加入正则后，模型精度也就 90 左右，测试集能高一点点，于是考虑删除了正则，重新进行 1000 次的迭代，最后训练集精度到达了 0.929，但是测试集精确度只有 0.91，和有正则时候接近，说明这个模型确实有过拟合的趋势。

不过在使用正则训练时，我更大的问题在于正则计算的是 theta 的欧几里得距离，它由于数值远大于交叉熵得到的 loss，导致训练 loss 经常 nan。如下面展示。最后我解决也是将其系数调低，希望后期能更好去解决这个问题。

```
Epoch5
 Loss:19.824228588316597 Acc:0.8742666666666666
Epoch6
 Loss:25.035474585431437 Acc:0.8760333333333333
Epoch7
 Loss:31.65427230539659 Acc:0.8766166666666667
Epoch8
 Loss:40.05983730795534 Acc:0.8776833333333334
Epoch9
 Loss:50.73445548575674 Acc:0.8781666666666667
Epoch10
 Loss:64.29124150485494 Acc:0.8785833333333334
g:\[illegible]\code\ex1\softmax_regression.py:7: RuntimeWarning: overflow encountered in exp
  x_exp = np.exp(x)
g:\[illegible]\code\ex1\softmax_regression.py:9: RuntimeWarning: invalid value encountered in true_divide
  return x_exp / (np.sum(x_exp,axis=1,keepdims=True))
g:\[illegible]\code\ex1\softmax_regression.py:15: RuntimeWarning: divide by zero encountered in log
  p = np.log(y_hat[list(range(len(y))),y]).mean()
Epoch11
 Loss:nan Acc:0.09871666666666666
Epoch12
 Loss:nan Acc:0.09871666666666666
Epoch13
 Loss:nan Acc:0.09871666666666666
Epoch14
 Loss:nan Acc:0.09871666666666666
Epoch15
 Loss:nan Acc:0.09871666666666666
Epoch16
```

图 6.7: 正则不恰当导致的 Nan

7 使用 torch

由于无法很好解决正则化和过拟合的问题，希望能使用 torch 内置的优化器，能给我更高的准确率，这里训练代码见仓库链接，核心代码如下

```python
def train(model, train_iter, test_iter, lr, epochs,device):
    def init_weights(m):
        if type(m) == nn.Linear or type(m) == nn.Conv2d:
            nn.init.xavier_uniform_(m.weight)

    model.apply(init_weights)
    optimizer = torch.optim.SGD(model.parameters(), lr=lr)
    loss = nn.CrossEntropyLoss()
    drawer = Drawer(xlabel='epoch', xlim=[0, epochs],
                            legend=['train loss', 'train acc', 'test acc'])
    timer = d2l.Timer()
    model.to(device)
    print(f'--------- {model.__class__.__name__} is training on {device}
        ----------')
    num_batches = len(train_iter)
    for epoch in range(epochs):
```

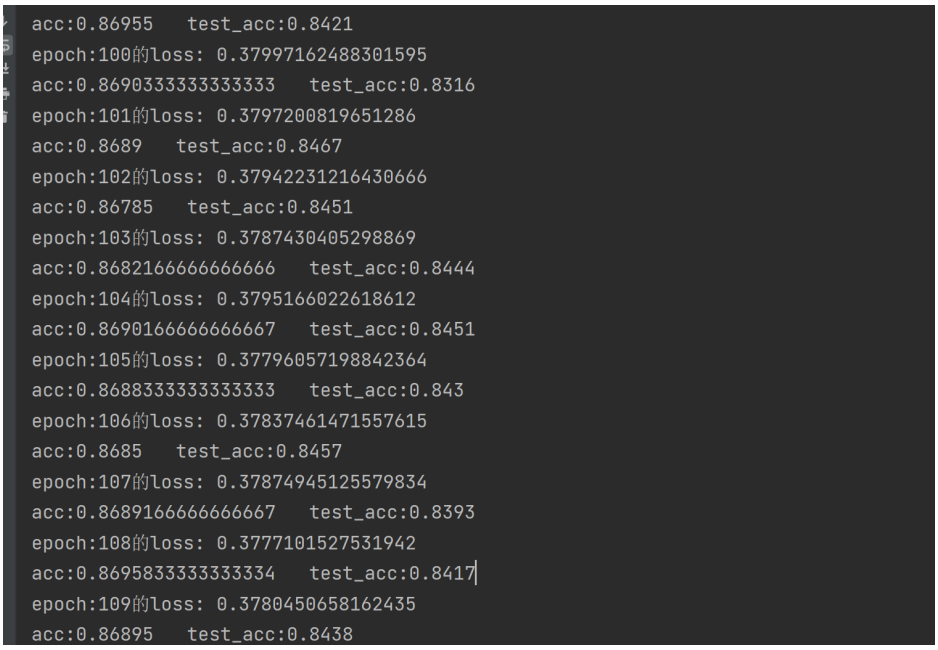


图 7.8

```
        metric = d2l.Accumulator(3)
        model.train()
        for i, (X, y) in enumerate(train_iter):
            optimizer.zero_grad()
            timer.start()
            X, y = X.to(device), y.to(device)
            y_hat = model(X)
            l = loss(y_hat, y)
            l.backward()
            optimizer.step()
            with torch.no_grad():
                metric.add(l * X.shape[0], accuracy(y_hat, y), X.shape[0])

net = nn.Sequential(nn.Flatten(), nn.Linear(784, 10))
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size=256)
device = d2l.try_gpu()
lr, epoch = 0.05, 400
train(model,train_iter,test_iter,lr,epoch,device)
```

但这个网络结果并不好，发现 88 左右时就极限，略微增加网络复杂度，再次尝试

```
    net = nn.Sequential(
```

```
        nn.Flatten(),
        nn.Linear(784, 256),
        nn.ReLU(),
        nn.Linear(256, 256),
        nn.ReLU(),
        nn.Linear(256, 10)
    )
```

最后结果如下，测试集能稳定在 0.89 左右，但是训练集已经到达 0.958，再训练下去应该就过拟合了。这里时间原因并没有进一步研究；

图 7.9: 程序开始 epoch 100 时结束

8 实验结论及心得体会

- 本次实验核心还是在于 softmax 和 Cross-Entropy 结合后导数的计算，因为其化简后的形式较为简单，简化了编程的难度。
- 对于 softmax，只是多分类问题上的入门，也只涉及到一层网络，实验更重要是让我们理解网络底层的运算逻辑，从网络参数的设置，到目标函数的计算，最后计算梯度，逐渐将网络的梯度更新的全过程。
- 最后是对正则项的感触，本是说不加入正则会导致指数上溢什么的，但是我实践过程中反而导致中间正则惩罚项由于数值过大导致占据了 loss 的主要部分，训练时候 loss 直接 Nan。最后也是将正则系数调小才勉强可以训练，但从最后 1000 次不加正则的尝试看，似乎加入正则的结果也并没有很好。