机器学习实验报告

实验名称： 建立全连接神经网络

学生姓名： 唐梓烨

学生学号： 58122310

完成日期： 2024/4/23

目录

任务描述

通过两种方式实现全连接神经网络，并对图片分类任务行进行测试与实验。

1. 手动实现简单的全连接神经网络
2. 使用 Pytorch 库简洁实现全连接神经网络

数据集简介

Fashion-MNIST 图片分类数据集包含 10 个类别的时装图像，训练集有 60,000 张图片，测 试集中有 10,000 张图片。图片为灰度图片，高度（h）和宽度（w）均为 28 像素，通道数 （channel）为 1。

10 个类别分别为：t-shirt(T 恤), trouser（裤子）, pullover（套衫）, dress（连衣裙）, coat（外套）， sandal（凉鞋），shirt（衬衫），sneaker（运动鞋），bag （包），ankle boot（短靴）。使用训练集数据进行训练，测试集数据进行测试。

图 1 Fashion-MNIST 数据集示例

目标

1. 掌握多层前馈神经网络及 BP 算法的原理与构建

   bp 算法简要回顾

   1) 前向传播，输入为 x

$$z = w_1 x + b_1$$
$$z' = \mathrm{ReLU}(z)$$
$$\hat{y} = w_2 z' + b_2$$

   得到经过一层隐藏层的伸进网络的输出$\hat{y}$，激活函数为 ReLU 函数。

   2) 反向传播

$$\frac{\partial L}{\partial \hat{y}} = \hat{y} - y$$

其中 L 是损失函数。再进一步求 L 对w_2，w_1，b_2，b_1的偏导，找到对于它们的梯度。

3) 更新参数

$$b_i \leftarrow b_i - \eta \frac{\partial L}{\partial b_i}$$

$$w_i \leftarrow w_i - \eta \frac{\partial L}{\partial w_i}$$

2. 了解 PyTorch 库，掌握本实验涉及的相关部分
3. 进行参数分析实验，理解学习率等参数的影响

实验内容

任务一：使用 Python 编程手动实现单隐层全连接神经网络

1. 模型架构

输入层 28*28=784 个节点，输出层 10 个节点，隐藏层 256 个节点。

激活函数：ReLU 函数

损失函数：Cross entropy

性能指标：准确率

优化算法：小批量梯度下降算法

2. 代码实现

(1) 初始化模型参数

```
def __init__(self, input_size, hidden_size, output_size, learning_rate = 0.1):
        self.W1 = torch.randn(input_size, hidden_size, requires_grad=True)
        self.b1 = torch.randn(1, hidden_size, requires_grad=True)
        self.W2 = torch.randn(hidden_size, output_size, requires_grad=True)
        self.b2 = torch.randn(1, output_size, requires_grad=True)
        self.learning_rate = learning_rate
```

(2) 设置 ReLU 激活函数

```
def relu(X):
    return torch.max(X, torch.tensor(0.0))
```

(3) 前向计算

```
def forward(self, X):
        # 前向传播
        X = X.view(X.size(0), -1)
        self.z1 = torch.matmul(X, self.W1) + self.b1
        self.a1 = relu(self.z1)
        self.z2 = torch.matmul(self.a1, self.W2) + self.b2
        return self.z2
```

(4) 设置损失函数并且进行后向传播

```
def backward(self, X, y):
        # 后向传播
        loss = F.cross_entropy(self.z2, y)
        loss.backward()

        # 更新参数
        with torch.no_grad():
            self.W1 -= self.learning_rate * self.W1.grad
            self.b1 -= self.learning_rate * self.b1.grad
            self.W2 -= self.learning_rate * self.W2.grad
            self.b2 -= self.learning_rate * self.b2.grad

            # 梯度清零
            self.W1.grad.zero_()
            self.b1.grad.zero_()
            self.W2.grad.zero_()
            self.b2.grad.zero_()
```

(5) 训练和评估模型

```
def train_hands(train_loader, model, epochs,test_loader):
    for epoch in range(epochs):
        for images, labels in train_loader:
            outputs = model.forward(images)
            model.backward(images, labels)

        print(f'手动全连接神经网络 Epoch [{epoch+1}/{epochs}] 训练完成')
        # 评估模型
        correct = 0
        total = 0
        test_loss = 0
        with torch.no_grad():
            for images, labels in test_loader:
                outputs = model.forward(images)
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()
                test_loss += F.cross_entropy(outputs, labels,
reduction='sum').item()
        accuracy = 100 * correct / total
        average_loss = test_loss / total
        print(f'Accuracy: {accuracy:.2f}%')
        print(f'Test Loss: {average_loss:.4f}')
```

任务二：使用 PyTorch 库简洁实现全连接神经网络

1. 模型架构

输入层 28*28=784 个节点，输出层 10 个节点，隐藏层 256 个节点。

激活函数：ReLU 函数

损失函数：Cross entropy

性能指标：准确率

优化算法：小批量梯度下降算法

2. 代码实现

(1) 使用 PyTorch 库简洁实现前述的全连接神经网络

```
class FCNNapi(nn.Module):
    def __init__(self, num_inputs, num_hiddens, num_outputs,
initializer=None):
        super(FCNNapi, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(num_inputs, num_hiddens)
        self.fc2 = nn.Linear(num_hiddens, num_outputs)

        if initializer is None:
            initializer = self.init_weights
        self.apply(initializer)  # 使用自定义的初始化方法

    # 初始化网络权重和偏置的函数
    def init_weights(self, m):
        if isinstance(m, nn.Linear):  # 检查是否为线性层
            nn.init.normal_(m.weight, mean=0.0, std=0.01)  # 使用正态分布初始
化权重
            nn.init.constant_(m.bias, 0)  # 将偏置初始化为 0

    def forward(self, x):
        x = self.flatten(x)
        x = self.fc1(x)
        x = relu(x)
        x = self.fc2(x)
        return x
```

(2) 进行训练

```
def train_loop(dataloader, model, loss_fn, optimizer, device,
num_epochs,test_loader):
    for epoch in range(num_epochs):
        model.train()
        for batch, (X, y) in enumerate(dataloader):
            X, y = X.to(device), y.to(device)
            pred = model(X)
```

```
            loss = loss_fn(pred, y)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
        print(f'api 简洁实现全连接神经网络 Epoch [{epoch+1}/{num_epochs}] 训练完成')
      # 在每个 epoch 结束后调用评估函数
        test_loss, accuracy = evaluate_model(model, test_loader, device, loss_fn)
        print(f"Epoch {epoch+1}, Test Loss: {test_loss:.4f}, Accuracy: {accuracy:.2f}%")
```

以上为实现两种方法的主要代码，下面对模型参数进行分析。

实验结果

任务一：手动实现单隐层全连接神经网络

(1) 训练过程中，训练集与验证集误差随 epoch 变化的曲线图

在学习率为 0.1，batch_size 为 100 的条件下，训练集与验证集误差随 epoch 变化的曲线如图

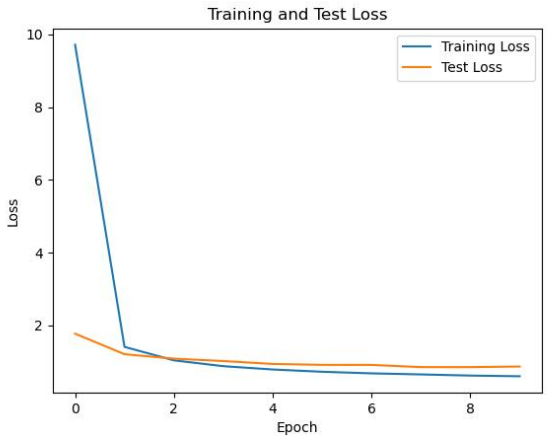


图 2 任务一中 Training&Test Loss 随 epoch 变化曲线

(2) 性能评估结果

在学习率为 0.1，batch_size 为 100 的条件下，验证集准确度随 epoch 变化的曲线如图

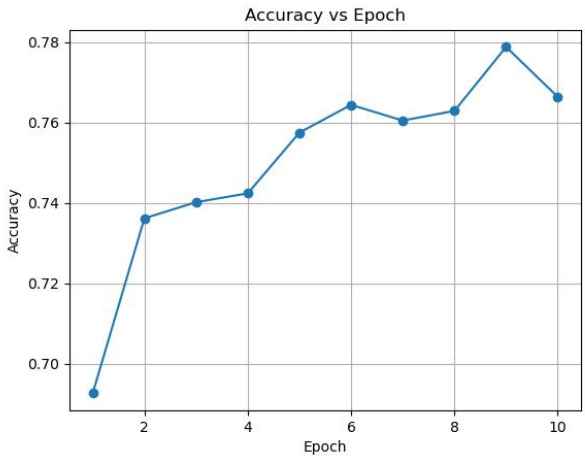


图 3 任务一中 Test Accuracy 随 epoch 变化曲线

(3) 参数分析实验：

i. 在所有其他参数保持不变的情况下，更改超参数 num_hiddens 的值，并查看此超参数的变化对结果有何影响。

num_hiddens = [32,64,128,256,512,1024]时，在 lr=0.1，batch_size=100 条件下，epoch=10 的验证集精度如图。

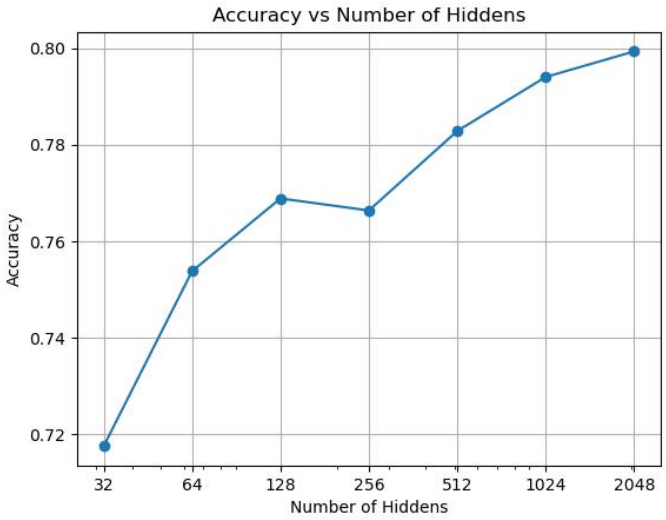


图 4 任务一中 Test Accuracy 随 Hidden nodes 变化曲线

可以看出在一定范围内 num_hiddens 越大，最终的模型（epoch=10）验证集精度越好，在测试的这几个 num_hiddens 中，效果最好的参数值是 2048。

隐藏节点数量实际对模型的影响：

1.表示能力：隐藏层结点数的增加可以增加神经网络的表示能力。更多的隐藏结点可以提供更多的自由度，使得网络能够学习更复杂的函数。这可能是在一定范围内验证集精度随隐层节点数增加而增加的原因。

2. 模型复杂度：同理隐藏层结点数的增加增加了神经网络的模型复杂度，这说明当隐层节点增大时会增大过拟合风险，导致模型在训练集上损失减小但在验证集上损失增大。

3. 训练时间：隐藏层结点数的增加会使得计算代价变大，训练时间延长。在实验时可以很明显的感受到随着隐藏层节点增大，训练时间逐渐变长。

4. 收敛速度：在改变超参数 num_hiddens 进行实验时，还观察到，num_hiddens 越大，第一代（epoch=1）模型的训练集损失和验证集损失都越大，并且达到收敛时的迭代次数需求越多。

任务二：使用 PyTorch 库简洁实现全连接神经网络

(1) 训练过程中，训练集与验证集误差随 epoch 变化的曲线图

在学习率为 0.01，batch_size 为 100 的条件下，训练集与验证集误差随 epoch 变化的曲线如图

图 5 任务二中 Training&Test Loss 随 epoch 变化曲线

(2) 性能评估结果

在学习率为 0.01，batch_size 为 100 的条件下，验证集准确度随 epoch 变化的曲线如图。

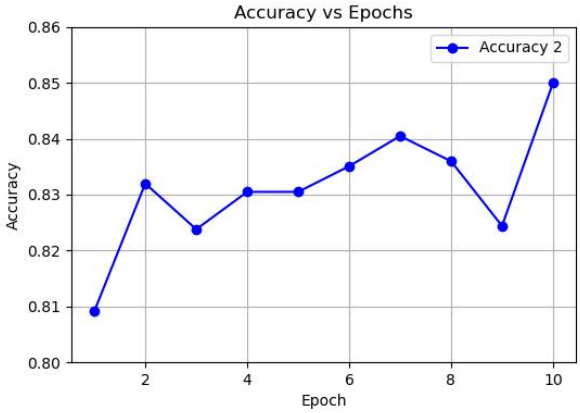


图 6 任务二中 Test Accuracy 随 epoch 变化曲线

(3) 参数分析实验：

i. 在所有其他参数保持不变的情况下，更改超参数 learning_rate 的值，并查看此超参数的变化对结果有何影响。

在 batch_size=100,epochs=10 的条件下，Accuracy 随 learning_rate 的变化情况如下图。

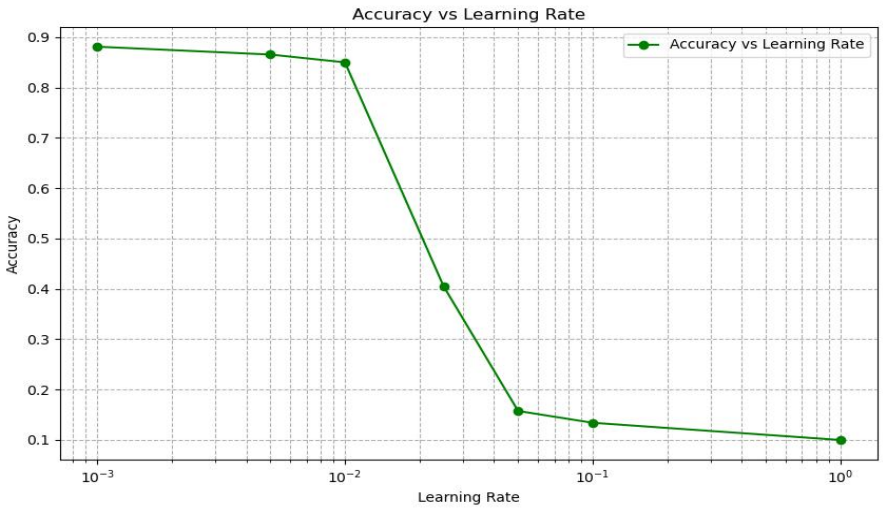
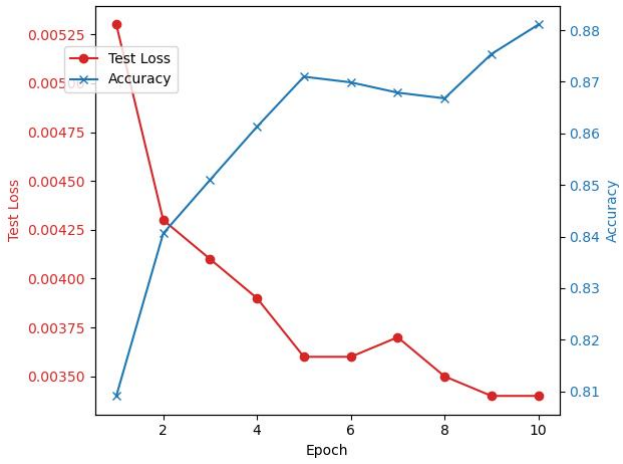


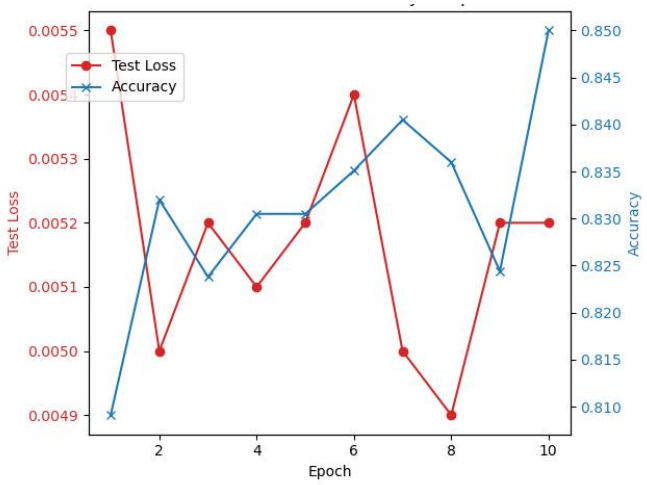
图 7 任务二中 Test Accuracy 随 Learning rate 变化曲线

可以看出随着 lr 的降低，Accuracy 逐渐减小。在 0.001-0.01 范围内 Accuracy 能维持在 80%以上的较高水平，当 lr=0.025 时，Accuracy 骤降，lr 大于 0.1 时，Accuracy 接近 10%，已达到最低水平。lr 的最优选择为 0.001。在具体 lr=0.001，0.01，0.1 下的 Test Loss&Accuracy 将在下面展示。

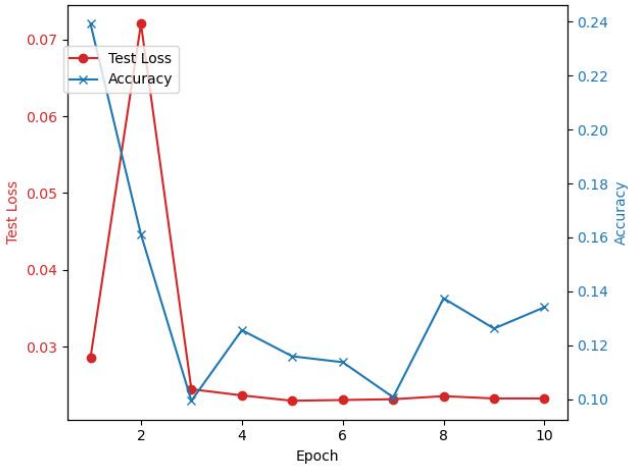
lr=0.001 时，Test Loss&Accuracy 关于 epoch 的变化曲线如图。



lr=0.01 时，Test Loss&Accuracy 关于 epoch 的变化曲线如图。



lr=0.1 时，Test Loss&Accuracy 关于 epoch 的变化曲线如图。



lr=1 时，学习率过大出现震荡导致模型完全无法收敛，在 epoch=10 时，Test Loss 甚至溢出无法显示。

```
Epoch 10, Test Loss: nan, Accuracy: 10.00%
```

学习率对模型的实际影响：

1. 收敛速度：学习率直接影响模型的收敛速度。较大的学习率会使模型更快地收敛到局部最优解，但可能会导致参数在最优解周围震荡，如上面的 lr=1 的情况；较小的学习率则会导致收敛速度较慢，但更有可能达到更好的全局最优解。

2. 网络性能：学习率直接影响模型在训练数据和测试数据上的性能。过大的学习率可能会导致过拟合。过小的学习率可能会导致模型无法充分学习数据的模式和特征，表现为欠拟合。

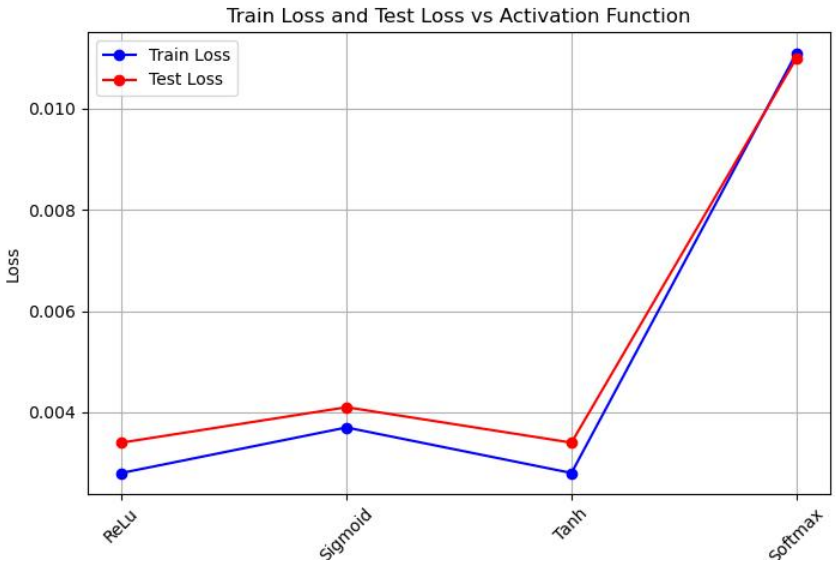
3. 是否能找到局部最优解：学习率还会影响模型收敛到的局部最优解的质量。较大的学习率可能会导致模型陷入局部最优解，而无法找到全局最优解；较小的学习率通常更有可能找到全局最优解，但需要更长的训练时间。

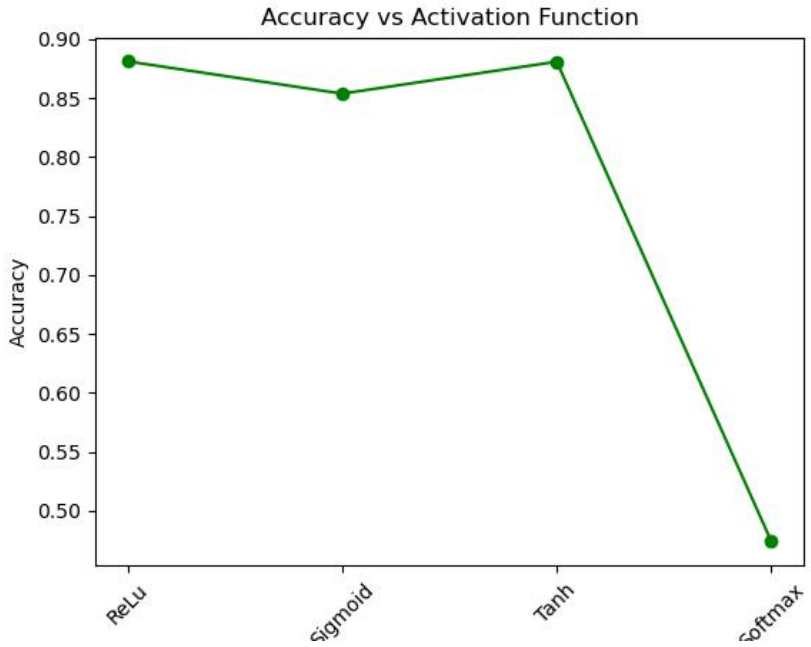
可以采用学习率衰减的方法来平衡收敛速度和找到最优解。学习率衰减是一种常用的技术，它可以使学习率随着训练的进行逐渐减小，以提高模型的稳定性和泛化能力。学习率衰减可以根据训练的进度自动调整学习率，从而在训练过程中保持较好的性能。

在本实验中，由于采用了批处理方法，实际更新参数的次数是 epochs×60000/batch_size，即使学习率较小，也有充足的次数调整至收敛，因此 lr 较小时，训练效果更好。

ii. 尝试不同的激活函数，哪个效果最好？

在 lr=0.001，batch_size=100, epochs=10 的条件下，尝试激活函数为 ReLU，sigmoid，Softmax，Tanh，看看谁的效果更好。





可以看出，ReLU 和 Tanh 函数效果差不多，Sigmoid 函数次之，Softmax 函数效果最差。

激活函数对模型训练的影响：

1. 梯度消失和梯度爆炸问题：在深层神经网络中，使用某些激活函数可能会导致梯度消失或梯度爆炸问题。例如，Sigmoid 函数在输入值较大或较小时梯度接近于 0，容易造成梯度消失问题；而 ReLU 函数在输入值为负时梯度为 0，容易造成梯度爆炸问题。

2. 稀疏激活：一些激活函数（如 ReLU）具有稀疏激活的特性，即在神经网络的某些神经元上，激活值为 0。这有助于减少模型的参数量和计算量，提高模型的效率和泛化能力。

3. 非线性变换：激活函数能够引入非线性变换，使神经网络可以学习复杂的非线性关系。这对于处理实际数据中的复杂模式非常重要。

4. 收敛速度：不同的激活函数对于模型的收敛速度可能会有所影响。一些激活函数（如

ReLU）具有更快的收敛速度，而另一些激活函数（如 Sigmoid 和 Tanh）可能会导致训练速度较慢。

总结

本次实验实现了使用 argparse 库定义 config.py 存放模型超参数，这样便于写脚本来执行超参数的不同取值进行训练和分析，找到最优超参数。实现了手动搭建全连接神经网络和 bp 算法，加深了我对 bp 算法的理解。实现了调用 pytorch 库的 nn.module api 快速实现多层神经网络的搭建，其提供的线性层类，一键式初始化参数，各种激活函数，损失函数都十分的方便快捷，并且 pytorch 自动记录梯度对于梯度下降算法十分的方便。

在实验初期，我发现在 batch_size 很小（=4）时，会出现 Training Loss 在 2.处震荡不减小，这可能是因为批数据量太小，模型无法从每个 batch 中获得足够的信息来准确地更新参数。