南　开　大　学

网　络　空　间　安　全　学　院

机器学习第三次实验报告

LeNet5 实现

许健

学号：2013018

专业：信息安全

指导教师：谢晋

2023 年 1 月 7 日

目录

一、 实验内容 **1**

二、 实验环境 **1**

三、 LeNet5 网络结构 **1**

（一） Input layer . . . 1
（二） Conv1 . . . 1
（三） Subsampling2 . . . 2
（四） Conv3 . . . 2
（五） Subsampling4 . . . 3
（六） Conv5 . . . 3
（七） FC6 . . . 4
（八） Output7 . . . 4

四、 代码实现 **5**

（一） LeNet5 . . . 5
（二） Conv . . . 6
1. 前向传播 . . . 7
2. 反向传播 . . . 7
（三） MaxPool . . . 9
（四） FC . . . 10
（五） ReLU . . . 11
（六） Softmax . . . 11
（七） 损失函数 . . . 12
（八） 优化算法 . . . 13
（九） 训练代码 . . . 14

五、 实验结果 **15**

六、 实验结果分析 **15**

一、 实验内容

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。

二、 实验环境

硬件环境：CPU

软件环境：Python 3.11，Numpy，PyCharm 调试

三、 LeNet5 网络结构

LeNet-5 诞生于上世纪 90 年代, 是 CNN 的开山之作, 最早的卷积神经网络之一, 用于手写数字识别（图像分类任务），它的诞生极大地推动了深度学习领域的发展。LeNet 在多年的研究和迭代后，Yann LeCun 将完成的这项开拓性成果被命名为 LeNet5，并发表在论文《Gradient-Based Learning Applied to Document Recognition》上，如今的 AlexNet、ResNet 等都是在其基础上发展而来的，在当年是一种用于手写体字符识别的非常高效的卷积神经网络。

LeNet-5 的网络结构放在如今的深度学习时代不算难，是非常简单的网络结构了，所以非常适合作为对卷积神经网络的入门学习。网络结构图如下：

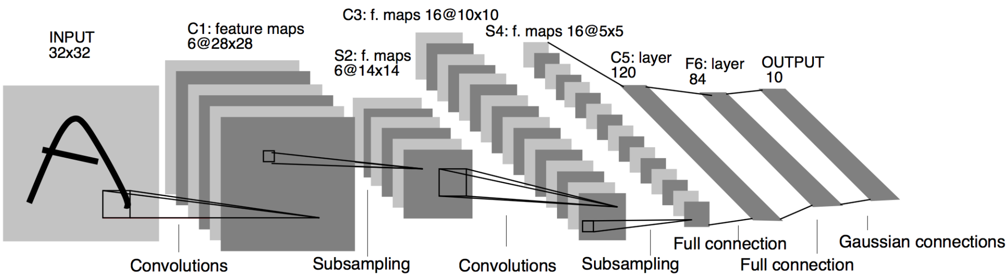

图 1: LeNet 网络结构

(一) Input layer

LeNet-5 的输入是一个 32×32 灰度图像，只有一层通道。

(二) Conv1

- 输入图片大小：32x32
- 卷积核大小：5x5，步长：1，不加 padding
- 卷积核个数：6
- 输出特征图大小：28x28
- 神经元数量：28x28x6
- 可训练参数为：(5x5+1)x6，“+1”是因为有偏置参数 bias
- 连接数：(5x5+1)x6x28x28=122304，“+1”是因为卷积后的激活函数也视为一次连接

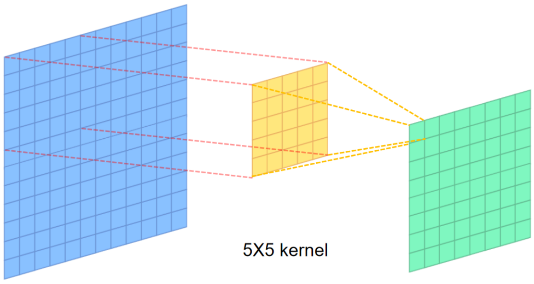

图 2: 卷积运算

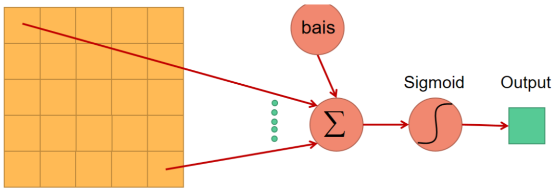

图 3: Sigmoid 激活函数

（三） Subsampling2

- 输入特征图大小：28x28
- 采样区域大小：2x2
- 采样方式：4 个输入相加，乘以一个可训练参数，再加上一个可训练偏置，结果通过 sigmoid 函数，步长为 2。
- 采样数量：6
- 输出特征图大小：14x14
- 神经元数量：14x14x6
- 连接数（和 Conv1 层连接）：(2x2+1)x6x14x14=5880，这里也是一样的“+1”是因为 sigmoid 算一次连接

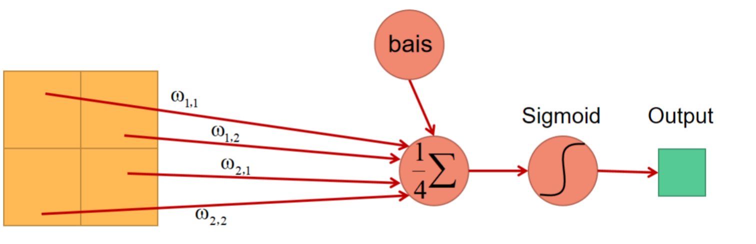

图 4: Pooling

（四） Conv3

- 输入：Subsampling2 中所有 6 个或者几个特征图组合

- 卷积核大小：5*5
- 卷积核数量：16
- 可训练参数：6x(3x5x5+1)+6x(4x5x5+1)+3x(4x5x5+1)+1x(6x5x5+1)
- 连接数：10x10x[6x(3x5x5+1)+6x(4x5x5+1)+3x(4x5x5+1)+1x(6x5x5+1)]=151600

Conv3 中的每个特征图是连接到 Subsampling2 中的所有 6 个或者几个特征图的，表示本层的特征图是上一层提取到的特征图的不同组合，如图5所示。在目前流行的卷积神经网络中，已较少使用这样的结构，这样设计的初衷有两个：

1. 有利于提取多种组合特征，希望能检测到不同的模式
2. 降低计算量

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

图 5: 特征图选取

（五） Subsampling4

- 输入：10x10
- 采样区域大小：2x2
- 采样方式：与 Subsampling2 保持一致。4 个输入相加，乘以一个可训练参数，再加上一个可训练偏置，结果通过 sigmoid 函数，步长为 2。
- 采样种类：16
- 输出特征图大小：5x5
- 神经元数量：5x5x16
- 连接数：(2x2+1)x5x5x16=2000

（六） Conv5

- 输入：5x5，即使用 Subsampling4 的全部 16 个通道作为输入
- 卷积核大小：5x5
- 卷积核数量：120
- 输出特征图大小：1x1
- 可训练参数：120x(16x5x5+1)

- 连接数：(5x5x16+1)x120=48120

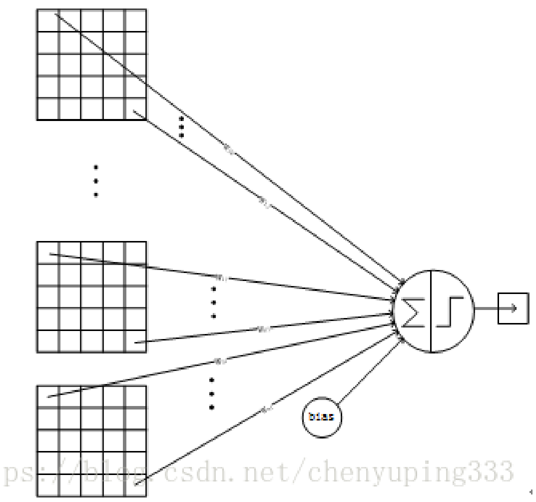

图 6: Conv5

（七） FC6

- 输入：120 维向量
- 输出：84 维向量
- 可训练参数：84x(120+1)=10164
- 计算方式：计算输入向量和权重向量之间的点积，再加上一个偏置，结果通过 sigmoid 函数输出

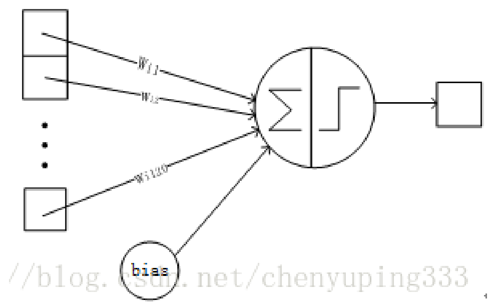

图 7: 全连接层

（八） Output7

- 输入：84 维向量
- 输出：10 维向量
- 可训练参数：84x10=840

- Output 层也是全连接层，共有 10 个节点，分别代表数字 0 到 9，且如果节点 i 的值为 0，则网络识别的结果是数字 i。采用的是径向基函数（RBF）的网络连接方式，目前已普遍用 Softmax 代替。

LeNet-5 是一种用于手写体字符识别的非常高效的卷积神经网络，整个网络共有 60840 个训练参数，340908 个连接。

四、 代码实现

代码实现的 LeNet 相较于 LeCun 在论文中提到的网络模型有些许改动，包括：

1. MNIST 图片维度 28*28，为了适应输入将 Conv1 input 从 32*32 resize 到 28*28
2. 激活函数使用 ReLu 代替 Sigmoid
3. 池化层采用最大池化
4. 为了统一实验中用到的卷积层（Conv1、Conv3、Conv5），不再对 Conv3 的输入特征图进行选择，而是将所有特征图都计算在内。

新的网络模型如图8所示，下面介绍 LeNet5 实现主要模块。

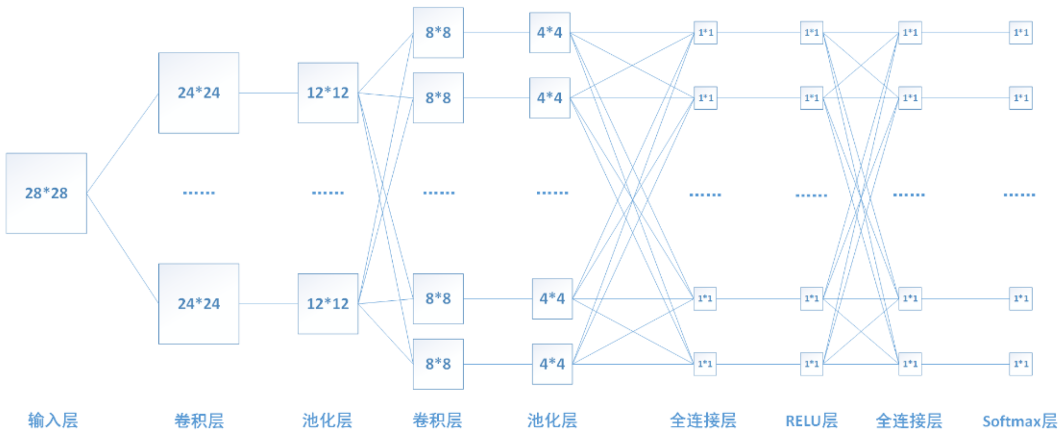

图 8: MNIST-LeNet 网络模型

(一) LeNet5

LeNet5 类是 LeNet-5 架构的实现，LeNet-5 架构是专为图像分类任务设计的卷积神经网络 (CNN)。LeNet5 类是 Net 类的子类，并在其构造函数中定义网络层。它还定义了前向传播，它通过将每一层依次应用于输入数据来执行前向传递网络，以及反向传播，它通过计算输出相对于每一层的输入。LeNet5 类还有一个实例变量 p2_shape，它存储了第二个池化层输出的形状。这在反向传播中用于在传递到第一个全连接 (FC) 层之前重塑梯度张量。

LeNet5 类实现

```
class LeNet5(Net):
    def __init__(self):
        self.conv1 = Conv(1, 6, 5)
        self.ReLU1 = ReLU()
```

```
        self.pool1 = MaxPool(2,2)
        self.conv2 = Conv(6, 16, 5)
        self.ReLU2 = ReLU()
        self.pool2 = MaxPool(2,2)
        self.FC1 = FC(16*4*4, 120)
        self.ReLU3 = ReLU()
        self.FC2 = FC(120, 84)
        self.ReLU4 = ReLU()
        self.FC3 = FC(84, 10)
        self.Softmax = Softmax()

        self.p2_shape = None

    def forward(self, X):
        h1 = self.conv1._forward(X)
        a1 = self.ReLU1._forward(h1)
        p1 = self.pool1._forward(a1)
        h2 = self.conv2._forward(p1)
        a2 = self.ReLU2._forward(h2)
        p2 = self.pool2._forward(a2)
        self.p2_shape = p2.shape
        fl = p2.reshape(X.shape[0],-1)
        h3 = self.FC1._forward(fl)
        a3 = self.ReLU3._forward(h3)
        h4 = self.FC2._forward(a3)
        a5 = self.ReLU4._forward(h4)
        h5 = self.FC3._forward(a5)
        a5 = self.Softmax._forward(h5)
        return a5

    def backward(self, dout):
        dout = self.FC3._backward(dout)
        dout = self.ReLU4._backward(dout)
        dout = self.FC2._backward(dout)
        dout = self.ReLU3._backward(dout)
        dout = self.FC1._backward(dout)
        dout = dout.reshape(self.p2_shape)
        dout = self.pool2._backward(dout)
        dout = self.ReLU2._backward(dout)
        dout = self.conv2._backward(dout)
        dout = self.pool1._backward(dout)
        dout = self.ReLU1._backward(dout)
        dout = self.conv1._backward(dout)
```

(二) Conv

Conv 类是神经网络中卷积层的实现。它有几个实例变量：

1. Cin：输入通道数。
2. Cout：输出通道数。
3. F：卷积滤波器的大小。
4. S：卷积滤波器的步幅。
5. W：卷积层的权重，使用 Xavier 初始化进行初始化。
6. b：卷积层的偏置项，随机初始化。
7. cache：缓存，用于存储前向传播的输入，这是反向传播所需要的。
8. pad：应用于输入的填充。

Conv 类定义了前向传播和反向传播两个方法，前向传播将输入与权重进行卷积并添加偏置项以产生输出，反向传播计算输出相对于输入、权重和偏置项的梯度。每个 Layer 层都会定义这两个方法，这里以 Conv 为例具体讲解。

1. 前向传播

_forward：通过卷积层执行前向传递，详细过程包括：

1. 使用 np.pad 使用指定的填充 self.pad 填充输入张量 X。这样做是为了确保输出具有与输入相同的空间维度，这在某些情况下很有用。
2. 确定输入张量 X 的维度，并计算输出张量 Y 的维度。输出张量 Y 具有与输入张量 X 相同的样本数 N、输出通道数 self.Cout 以及使用输入空间维度 H 和 W 计算的空间维度 H_ 和 W_、滤波器大小 self.F，和步幅 self.S。
3. 输出张量 Y 被初始化为全零。
4. 嵌套循环遍历示例 n、输出通道 c 以及输出张量 Y 的空间维度 h 和 w。对于每次迭代，提取输入张量 X 的相应切片，并将其乘以权重 self .W 和偏置项 self.b 被添加以产生输出。然后将输出存储在输出张量 Y 中的相应位置。
5. 输入张量 X 存储在缓存实例变量中，用于反向传播。
6. 返回输出张量 Y。

2. 反向传播

_backward：通过卷积层执行反向传播，详细过程包括：

1. 从缓存实例变量中检索前向传递的输入张量 X。
2. 权重 dW 和偏置项 db 的梯度被初始化为全零。
3. 权重 dW 的梯度是使用嵌套循环计算的，该循环迭代输出通道 co、输入通道 ci 以及权重的空间维度 h 和 w。对于每次迭代，提取输入张量 X 的相应切片，并将其乘以输出 dout 的梯度以产生权重 dW 的梯度。
4. 偏置项 db 的梯度是使用在输出通道 co 上迭代的循环计算的。对于每次迭代，输出 dout 的梯度沿其他维度求和以产生偏置项 db 的梯度。

5. 使用 np.pad 填充输出 dout 的梯度，以补偿前向传递中应用于输入的填充。
6. 输入 dX 的梯度被初始化为全零。输入 dX 的梯度是使用嵌套循环计算的，该循环遍历示例 n

Conv5 类实现

```
class Conv():
    def __init__(self, Cin, Cout, F, stride=1, padding=0, bias=True):
        self.Cin = Cin
        self.Cout = Cout
        self.F = F
        self.S = stride
        self.W = {'val': np.random.normal(0.0,np.sqrt(2/Cin),(Cout,Cin,F,F)),
            'grad': 0}
        self.b = {'val': np.random.randn(Cout), 'grad': 0}
        self.cache = None
        self.pad = padding

    def _forward(self, X):
        X = np.pad(X, ((0,0),(0,0),(self.pad,self.pad),(self.pad,self.pad)),
            'constant')
        (N, Cin, H, W) = X.shape
        H_ = H - self.F + 1
        W_ = W - self.F + 1
        Y = np.zeros((N, self.Cout, H_, W_))

        for n in range(N):
            for c in range(self.Cout):
                for h in range(H_):
                    for w in range(W_):
                        Y[n, c, h, w] = np.sum(X[n, :, h:h+self.F, w:w+self.F
                            ] * self.W['val'][c, :, :, :]) + self.b['val'][c]

        self.cache = X
        return Y

    def _backward(self, dout):
        X = self.cache
        (N, Cin, H, W) = X.shape
        H_ = H - self.F + 1
        W_ = W - self.F + 1
        W_rot = np.rot90(np.rot90(self.W['val']))

        dX = np.zeros(X.shape)
        dW = np.zeros(self.W['val'].shape)
        db = np.zeros(self.b['val'].shape)

        # dW
```

```
            for co in range(self.Cout):
                for ci in range(Cin):
                    for h in range(self.F):
                        for w in range(self.F):
                            dW[co, ci, h, w] = np.sum(X[:, ci, h:h+H_, w:w+W_] *
                                dout[:, co, :, :])

        # db
        for co in range(self.Cout):
            db[co] = np.sum(dout[:, co, :, :])

        dout_pad = np.pad(dout, ((0,0),(0,0),(self.F,self.F),(self.F,self.F))
            , 'constant')

        # dX
        for n in range(N):
            for ci in range(Cin):
                for h in range(H):
                    for w in range(W):
                        dX[n, ci, h, w] = np.sum(W_rot[:, ci, :, :] * dout_pad[n
                            , :, h:h+self.F, w:w+self.F])

        return dX
```

（三） MaxPool

MaxPool 类是一个最大池化层，它通过对每个通道的局部邻域取最大值来执行输入张量的下采样。

MaxPool 类的 _forward 方法通过最大池化层执行前向传播，它使用与 X 相同数量的示例和通道初始化输出张量 Y，但根据池化参数 F 和步幅具有更小的空间维度。初始化一个与 X 具有相同形状的掩码张量 M 并用零填充它。迭代输出张量 Y 的示例 n、通道 cin 和空间维度 w_ 和 h_。对于每次迭代，它提取输入张量 X 的一个邻域，该邻域对应于当前空间维度 w_ 和 h_，并计算该邻域的最大值。在输出张量 Y 的当前位置存储最大值，在找到最大值的输入张量 X 的位置存储 1，在其他位置存储 0。将掩码张量 M 存储在缓存实例变量中，并返回输出张量 Y。

MaxPool 类的 _backward 方法执行通过最大池层的反向传播。它从缓存实例变量中检索掩码张量 M。初始化一个与输入张量 X 具有相同形状的梯度张量 dX 并用零填充它。迭代梯度张量 dX 的样本 n 和通道 c。对于每次迭代，它根据池化参数 F 和步长，沿着梯度张量 dX 的空间维度重复梯度张量 dout 的相应切片的值。它将梯度张量 dX 乘以掩码张量 M 并返回结果。

MaxPool 类实现

```
class MaxPool():
    def __init__(self, F, stride):
        self.F = F
        self.S = stride
        self.cache = None

    def _forward(self, X):
```

```
        (N,Cin,H,W) = X.shape
        F = self.F
        W_ = int(float(W)/F)
        H_ = int(float(H)/F)
        Y = np.zeros((N,Cin,W_,H_))
        M = np.zeros(X.shape)
        for n in range(N):
            for cin in range(Cin):
                for w_ in range(W_):
                    for h_ in range(H_):
                        Y[n,cin,w_,h_] = np.max(X[n,cin,F*w_:F*(w_+1),F*h_:F
                            *(h_+1)])
                        i,j = np.unravel_index(X[n,cin,F*w_:F*(w_+1),F*h_:F*(
                            h_+1)].argmax(), (F,F))
                        M[n,cin,F*w_+i,F*h_+j] = 1
        self.cache = M
        return Y

    def _backward(self, dout):
        M = self.cache
        (N,Cin,H,W) = M.shape
        dout = np.array(dout)
        dX = np.zeros(M.shape)
        for n in range(N):
            for c in range(Cin):
                dX[n,c,:,:] = dout[n,c,:,:].repeat(2, axis=0).repeat(2, axis
                    =1)

        return dX*M
```

(四) FC

FC 类表示神经网络中的全连接 (FC) 层。它有两个实例变量：W 和 b，分别是该层的权重矩阵和偏置向量。FC 层也有两个方法：_forward 和 _backward。_forward 方法通过 FC 层执行前向传递，计算给定输入张量 X 的层的输出。_backward 方法通过 FC 层执行反向传递，计算输出相对于输入的梯度。FC 层还有一个 _update_params 方法，它根据梯度和学习率 lr 更新权重矩阵和偏置向量。

FC 类实现

```
class FC():
    def __init__(self, D_in, D_out):
        self.cache = None
        self.W = {'val': np.random.normal(0.0, np.sqrt(2/D_in), (D_in,D_out))
            , 'grad': 0}
        self.b = {'val': np.random.randn(D_out), 'grad': 0}

    def _forward(self, X):
        out = np.dot(X, self.W['val']) + self.b['val']
```

```python
        self.cache = X
        return out

    def _backward(self, dout):
        X = self.cache
        dX = np.dot(dout, self.W['val'].T).reshape(X.shape)
        self.W['grad'] = np.dot(X.reshape(X.shape[0], np.prod(X.shape[1:])).T
            , dout)
        self.b['grad'] = np.sum(dout, axis=0)
        return dX

    def _update_params(self, lr=0.01):
        # Update the parameters, lr = 0.01
        self.W['val'] -= lr*self.W['grad']
        self.b['val'] -= lr*self.b['grad']
```

（五） ReLU

ReLU 类表示神经网络中的整流线性单元 (ReLU) 激活层。_forward 通过 ReLU 层执行前向传递，将 ReLU 激活函数按元素应用于输入张量 X。_backward 通过 ReLU 层执行反向传递，计算梯度相对于输入的输出。ReLU 层还有一个缓存实例变量，它存储了 _forward 方法的输入，以供 _backward 方法使用。

ReLU 类实现

```python
class ReLU():
    def __init__(self):
        self.cache = None

    def _forward(self, X):
        out = np.maximum(0, X)
        self.cache = X
        return out

    def _backward(self, dout):
        X = self.cache
        dX = np.array(dout, copy=True)
        dX[X <= 0] = 0
        return dX
```

（六） Softmax

Softmax 类表示神经网络中的 Softmax 损失函数。在前向传播中，输入 X 使用 softmax 函数转换为概率分布 Z。在反向传播中，损失相对于输入 X 的梯度是使用链式法则计算的。损失相对于 X 的梯度由 dout 给出，X 相对于 Z、Y 和中间变量的梯度使用代码中提供的表达式计算。然后通过将 dout 相对于 Z、Y 和中间变量的梯度相乘来计算损失相对于 X 的最终梯度。

Softmax 类实现

```python
class Softmax():
    def __init__(self):
        self.cache = None

    def _forward(self, X):
        maxes = np.amax(X, axis=1)
        maxes = maxes.reshape(maxes.shape[0], 1)
        Y = np.exp(X - maxes)
        Z = Y / np.sum(Y, axis=1).reshape(Y.shape[0], 1)
        self.cache = (X, Y, Z)
        return Z

    def _backward(self, dout):
        X, Y, Z = self.cache
        dZ = np.zeros(X.shape)
        dY = np.zeros(X.shape)
        dX = np.zeros(X.shape)
        N = X.shape[0]
        for n in range(N):
            i = np.argmax(Z[n])
            dZ[n,:] = np.diag(Z[n]) - np.outer(Z[n],Z[n])
            M = np.zeros((N,N))
            M[:,i] = 1
            dY[n,:] = np.eye(N) - M
        dX = np.dot(dout,dZ)
        dX = np.dot(dX,dY)
        return dX
```

（七） 损失函数

NLLLoss 代表“负对数似然损失”，这是一种用于分类任务的损失函数，该函数返回 Y_pred 中所有样本的平均负对数似然损失。它接受两个参数：

1. Y_pred：形状为 (N, C) 的二维 NumPy 数组，其中 N 是样本数，C 是类别数。Y_pred 存储每个样本的预测类别概率。
2. Y_true：与 Y_pred 形状相同的 NumPy 数组，其中如果第 i 个样本属于第 j 类，则 Y_true[i, j] 为 1，否则为 0。

SoftmaxLoss 是表示 softmax 损失函数的类，它是 NLL 损失函数对多类分类的推广。__init__ 函数不执行任何操作。SoftmaxLoss 类的 get 方法采用与 NLLLoss 相同的参数，并返回 softmax 损失和损失相对于预测类概率的梯度。具体来说，它使用 NLLLoss 计算平均 NLL 损失，然后计算损失相对于预测类概率的梯度，方法是将梯度设置为等于预测类概率和真实类概率之间的差值，并按样本数缩放。

Softmax 损失函数

```python
def NLLLoss(Y_pred, Y_true):
    # Negative log likelihood loss
```

```python
    loss = 0.0
    N = Y_pred.shape[0]
    M = np.sum(Y_pred*Y_true, axis=1)
    for e in M:
        if e == 0:
            loss += 500
        else:
            loss += -np.log(e)
    return loss/N

class SoftmaxLoss():
    def __init__(self):
        pass

    def get(self, Y_pred, Y_true):
        N = Y_pred.shape[0]
        loss = NLLLoss(Y_pred, Y_true)
        Y_serial = np.argmax(Y_true, axis=1)
        dout = Y_pred.copy()
        dout[np.arange(N), Y_serial] -= 1
        return loss, dout
```

（八） 优化算法

SGD 类是随机梯度下降 (SGD) 优化算法的实现。__init__ 函数接受三个参数：

1. params：字典列表，每个字典代表一个模型参数，有键'val' 和'grad'，分别存储参数的当前值和梯度
2. lr：SGD 算法的学习率（一个超参数）
3. reg：正则化强度（另一个超参数）

step 函数通过使用梯度、学习率和正则化强度更新 self.parameters 中的参数值来执行单个优化步骤。具体来说，它通过减去学习率乘以参数的梯度 self.lr*param['grad'] 加上正则化项 self.reg*param['val'] 来更新每个参数 param。

SGD 实现

```python
class SGD():
    def __init__(self, params, lr=0.01, reg=0):
        self.parameters = params
        self.lr = lr
        self.reg = reg

    def step(self):
        for param in self.parameters:
            param['val'] -= (self.lr*param['grad'] + self.reg*param['val'])
```

（九） 训练代码

代码运行首先需要加载数据，然后对数据预处理。之后开始迭代训练，每次训练获取当前批次数据，对标签进行独热编码。首先需要进行前向传播计算损失和反向传播矩阵，然后进行反向传播并更新迭代器。训练完成后将参数保存，打印 loss 变化过程，并在训练数据集和测试数据集上测试模型精确度。

训练 LeNet 网络

```
//Load
X_train, Y_train, X_test, Y_test = mnist.load()
X_train, X_test = X_train/np.float32(255), X_test/np.float32(255)
X_train -= np.mean(X_train)
X_test -= np.mean(X_test)
X_train = X_train.reshape(X_train.shape[0],1,28,28)
X_test = X_test.reshape(X_test.shape[0],1,28,28)

//Train
for i in range(ITER):
    X_batch, Y_batch = get_batch(X_train, Y_train, batch_size)
    Y_batch = MakeOneHot(Y_batch, D_out)
    Y_pred = model.forward(X_batch)
    loss, dout = criterion.get(Y_pred, Y_batch)
    model.backward(dout)
    optim.step()

    if i % 100 == 0:
        print("%s%% iter: %s, loss: %s" % (100*i/ITER,i, loss))
        losses.append(loss)

//Draw
util.draw_losses(losses)

# Test
# TRAIN SET ACC
Y_pred = model.forward(X_train)
result = np.argmax(Y_pred, axis=1) - Y_train
result = list(result)
print("TRAIN--> Correct: " + str(result.count(0)) + " out of " + str(X_train.
    shape[0]) + ", acc=" + str(result.count(0)/X_train.shape[0]))
# TEST SET ACC
Y_pred = model.forward(X_test)
result = np.argmax(Y_pred, axis=1) - Y_test
result = list(result)
print("TEST--> Correct: " + str(result.count(0)) + " out of " + str(X_test.
    shape[0]) + ", acc=" + str(result.count(0)/X_test.shape[0]))
```

五、 实验结果

我们将使用随机梯度下降（SGD）训练我们的网络，实验中主要的超参数包括：批量大小 batch_size、迭代轮数 ITER、学习率 lr。

初始时设置批量大小为 64，迭代 20000 轮，学习率为 0.0001，实验结果如图9所示。在测试数据集上精度达到 98.61%，表现不错。训练过程中的 loss 变化如图10所示，横轴的单位为 250 次迭代。

```
96.25% iter: 19250, loss: 0.0021850379796853957
97.5% iter: 19500, loss: 0.0011547345012980481
98.75% iter: 19750, loss: 0.019489697853986625
TEST--> Correct: 9861 out of 10000, acc=0.9861
```

图 9: 模型训练精度

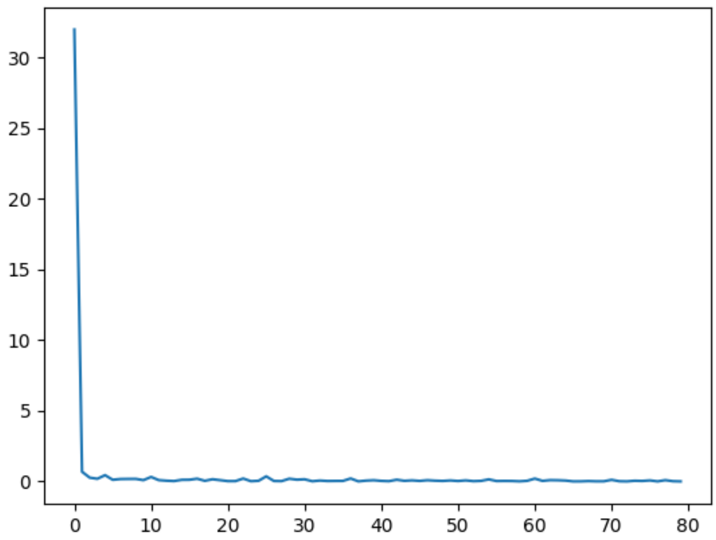

图 10: SGD 优化器训练 loss

六、 实验结果分析

测试训练数据集的精度时报错：*numpy.core._exceptions._ArrayMemoryError*。这是因为需要处理的数据量太大，电脑内存不足，GPU 性能不够，存在内存溢出现象，所以实验中没有测试训练数据集的精度。但是从 LeNet 模型的承载能力来看，MNIST 数据集并不会出现过拟合的问题，也可以添加正则化项，不过本次实验增加正则化项对于模型精确度的提升没有帮助。

LeNet 模型基本可以在迭代 10000 以内收敛，迭代 20000 次训练网络大约需要半个小时。我尝试调整批量大小与学习率之间的关系，增大批量大小的同时适当提高学习率，一个比较合适的超参数是批量大小 64，SGD 算法学习率 0.0001，模型收敛到一个较好的地方。从方差的角度上看，更大的 batch 意味着一个 mini-batch 中样本的方差更小，也同时意味着一个 mini-batch 带来的梯度方差也更小，梯度更加可信，噪声给模型带来的影响也会相应减少，在可信的梯度下，

我们可以使用更大的 learning rate 来更新参数，提高收敛速度是可行的。适当的增大 learning rate 还可以有效避免模型走到一个比较差的 local minima，大 lr 可有效逃离并收敛到更好的地方。

为了加快模型的收敛速度，我在 SGD 的基础上增加动量机制，即 SGD with momentum。在每次迭代中，将上一次的梯度乘上一个超参数 γ 加到当前的梯度上，从而让优化器在下降时更快，在上升时更慢。这可以帮助模型更快地收敛，同时还可以避免震荡。

```
def __init__(self, params, lr=0.0001, momentum=0.99, reg=0):
    self.l = len(params)
    self.parameters = params
    self.velocities = []
    for param in self.parameters:
        self.velocities.append(np.zeros(param['val'].shape))
    self.lr = lr
    self.rho = momentum
    self.reg = reg

def step(self):
    for i in range(self.l):
        #update velocities[i]
        #update parameters[i]['val']
```

SGD 优化算法的另一个缺点是容易在最优值（梯度较小值）附近波动，loss 有轻微的震荡。如果换用其他的优化器，比如 Adam，在模型训练期间自动调整学习率，能够很好地处理噪声。从图11中可以看出，损失值下降速度更快，且下降到一定范围后基本维持不变。

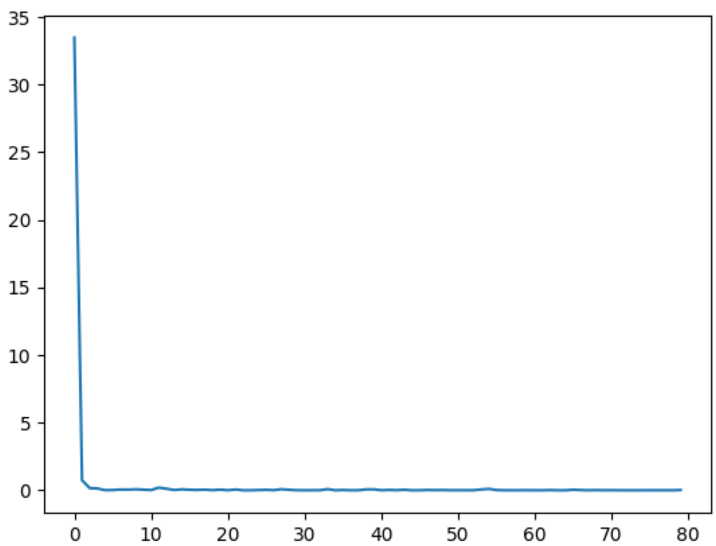

图 11: Adam 优化器训练 loss