南開大學
Nankai University

计算机学院

机器学习实验报告

作业二　利用 Numpy 实现 Lenet5 识别手写数字

姓名：赵康明

学号：2110937

专业：计算机科学与技术

15th January 2024

1　作业要求

在这个练习中，需要使用 Python 实现 LeNet5 来完成对 MNIST 数据集中 0-9 共 10 个手写数字的分类。代码只能使用 Python 实现，其中数据读取可使用 PIL、opencv-python 等库，矩阵运算可使用 numpy 等计算库，网络前向传播和梯度反向传播需要手动实现，不能使用 PyTorch、TensorFlow、Jax 或 Caffe 等自动微分框架。

2　Lenet5 模型

2.1　Lenet5 介绍

LeNet-5（LeNet）是由 Yann LeCun 等人在 1998 年提出的卷积神经网络（CNN）架构，是深度学习领域中的经典模型之一。LeNet-5 主要用于手写数字识别，特别是美国邮政服务的邮政编码和数字识别任务。以下是 LeNet-5 的一些关键特点：

1. 网络结构：LeNet-5 由七层组成，包括卷积层、池化层和全连接层。其整体结构为：输入层（32x32 像素的手写数字图像）- 卷积层 - 池化层 - 卷积层 - 池化层 - 全连接层 - 全连接层 - 输出层（10 个类别，对应 0-9 的数字）。

2. 卷积和池化层：LeNet-5 使用卷积层来提取特征，这些特征通过 S 型激活函数进行非线性变换。卷积层之后是池化层，用于下采样和特征的空间压缩。

3. 激活函数：LeNet-5 中的激活函数采用 Sigmoid 函数，后来的深度学习模型通常使用 ReLU（Rectified Linear Unit）等激活函数。

4. 全连接层：在卷积和池化层之后是两个全连接层，用于将提取的特征映射到最终的输出类别。

5. Softmax 输出：输出层使用 Softmax 激活函数，生成每个类别的概率分布。

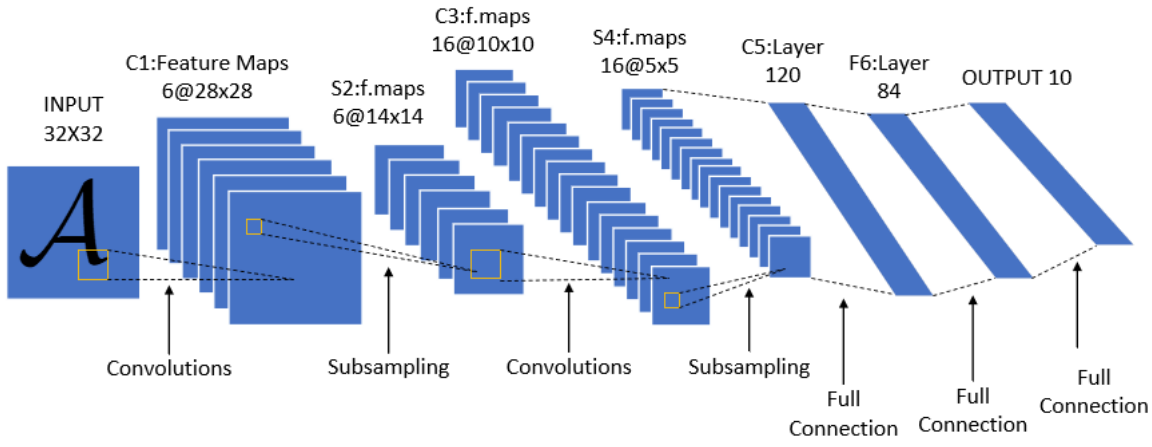

图 2.1: Lenet5 网络模型

接下来将逐层对该模型进行分析。

2.2 Input

Lenet5 的输入是一个 32x32 的灰度图像，通道数为 1。结合网络参考资料，手写数字体识别 minst 数据集的图片大小为 28x28，因此我们需要在读取数据后对其进行 padding，使其成为 32x32。

2.3 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

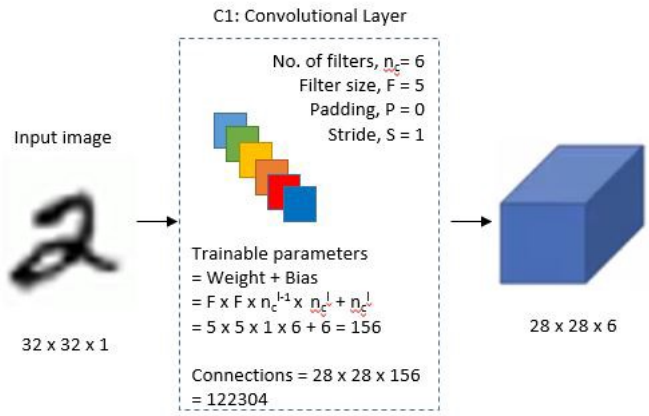

图 2.2: C1 层

2.4 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，结果通过激活函数 sigmoid 非线性输出，为避免梯度爆炸问题，本初
- 采样数量：6，因为有来自上一层的 6 个特征图
- 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 Relu 算一次连接

- Subsampling2 中每个特征图的大小是 Conv1 中特征图大小的 1/4。

池化层的数学原理如下：

平均池化层的前向传播

$$Y_{i,j,c} = \frac{1}{k^2}\sum_{p=0}^{k-1}\sum_{q=0}^{k-1} X_{s\cdot i+p,s\cdot j+q,c}$$

其中：

- $Y_{i,j,c}$ 是输出的第 i 行，第 j 列，第 c 通道的元素。
- $X_{s\cdot i+p,s\cdot j+q,c}$ 是输入的第 $s\cdot i+p$ 行，第 $s\cdot j+q$ 列，第 c 通道的元素。

平均池化层的反向传播

$$\frac{\partial E}{\partial X_{i,j,c}} = \frac{1}{k^2}\sum_{p=0}^{k-1}\sum_{q=0}^{k-1}\frac{\partial E}{\partial Y_{\lfloor i/s\rfloor,\lfloor j/s\rfloor,c}} \quad \text{，其中} s\cdot\lfloor i/s\rfloor = i, s\cdot\lfloor j/s\rfloor = j$$

其中：

- $\frac{\partial E}{\partial X_{i,j,c}}$ 是损失函数 E 对于输入的第 i 行，第 j 列，第 c 通道的梯度。
- $\frac{\partial E}{\partial Y_{\lfloor i/s\rfloor,\lfloor j/s\rfloor,c}}$ 是损失函数 E 对于输出的第 $\lfloor i/s\rfloor$ 行，第 $\lfloor j/s\rfloor$ 列，第 c 通道的梯度。

2.5 Conv3

接下来应当是 Lenet5 中的第三层。

- 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
- 输出特征图大小：10*10 于 Conv1 的区别：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。存在方式是：
  - 6 个特征图以 Subsampling2 中 3 个相邻的特征图子集作为输入, 卷积核大小为 5x5x3;
  - 6 个特征图以 Subsampling2 中 4 个相邻特征图子集为输入, 卷积核大小为 5x5x4;
  - 3 个特征图以不相邻的 4 个特征图子集作为输入, 卷积核大小为 5x5x4;
  - 1 个特征图将 Subsampling2 中所有特征图作为输入。

	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 2.3: C1 层

卷积核是 5×5 且具有 3 个通道，每个通道各不相同，这也是下面计算时 5*5 后面还要乘以 3,4,6 的原因。这是多通道卷积的计算方法。参数个数：(5*5*3+1)*6+(5*5*4+1)*6+(5*5*4+1)*3+(5*5*6+1)*1=1516。连接数：1516*10*10 = 151600。10*10 为输出特征层，每一个像素都由前面卷积得到，即总共经历 10*10 次卷积。

2.6 Subsampling4

S4 层与 S2 一样也是降采样层，具体参数介绍如下：

- 输入大小：(10*10)*16
- 采样区域 (池化核大小)：2*2
- 下采样数量：16
- 输出特征图大小：5*5
- 可训练参数量：(1+1)*16
- 计算量：(2*2+1)*5*5*16
- 神经元数量：5*5*16
- 连接数：(2*2+1)*5*5*16

2.7 Conv5

Conv5 也是卷积层，其具体参数介绍如下：

- 输入图片大小：(5*5)*16
- 卷积核大小：5*5
- 卷积核种类：120
- 输出特征图数量：120
- 输出特征图大小：(5-5+1)*(5-5+1)=1*1
- 可训练参数量：(16*5*5+1)*120

- 计算量：(16*5*5+1)*1*1*120
- 神经元数量：1*1*120
- 连接数：(16*5*5+1)*1*1*120

2.8 F6

F6 是全连接层，共有 84 个神经元，与 Conv5 层进行全连接，即每个神经元都与 Conv5 层的 120 个特征图相连。计算输入向量和权重向量之间的点积，再加上一个偏置，结果通过激活函数输出。

2.9 Output7

最后的 Output 层为全连接层，采用了 RBF 函数（径向欧式距离函数），计算输入向量和参数向量之间的欧式距离。但现今模型中常采用的是 Softmax 函数。

3 代码实现

3.1 卷积层

卷积层的初始化

```python
class conv:
    def __init__(self, filter_shape, stride=1, padding='SAME', bias=True,
        requires_grad=True):
        """
        初始化卷积层。

        :param filter_shape: 元组 (O, C, K,
            K)，表示输出通道数、输入通道数和卷积核大小。
        :param stride: 卷积操作的步幅。
        :param padding: 填充类型：{"SAME", "VALID"}。
        :param bias: 是否包含偏置。
        :param requires_grad: 是否在反向传播中计算梯度。
        """
        # 使用 Kaiming 初始化权重
        self.weight = parameter(np.random.randn(*filter_shape) * (2/reduce(lambda x,
            y: x*y, filter_shape[1:]))**0.5)
        self.stride = stride
        self.padding = padding
        self.requires_grad = requires_grad
        self.output_channel = filter_shape[0]
        self.input_channel = filter_shape[1]
        self.filter_size = filter_shape[2]

```

```
        # 如果启用偏置，则初始化偏置
        if bias:
            self.bias = parameter(np.random.randn(self.output_channel))
        else:
            self.bias = None
```

1. filter_shape: 包含四个值 (O, C, K, K)，分别表示了卷积层的输出通道数（O）、输入通道数（C）、卷积核的高度（K），以及卷积核的宽度（K）。
2. stride: 这是卷积操作的步幅，它决定了卷积核在输入特征图上滑动的步长，默认为 1，表示每次滑动一个像素。
3. padding: 填充类型，可以是”SAME” 或”VALID”。”SAME” 表示在输入特征图的边缘进行零填充，以保持输出特征图与输入特征图相同的尺寸；”VALID” 表示不进行填充，输出特征图的尺寸会缩小。

在构造函数内部，代码执行以下操作：使用 Kaiming 初始化方法来初始化权重（self.weight），Kaiming 初始化是一种用于神经网络权重初始化的方法，它有助于加速网络的训练和收敛。初始化的值是一个服从正态分布的随机数，乘以一个缩放因子，该缩放因子基于权重的维度进行调整。设置卷积层的步幅（self.stride）、填充类型（self.padding）、是否包含偏置项（self.bias）、是否计算梯度（self.requires_grad）等参数。如果启用了偏置（bias=True），则初始化偏置项（self.bias），偏置项的初始化值也是服从正态分布的随机数。

前向传播　卷积层的前向传播公式如下：

$$O_{n,o,h,w} = \sum_{c=1}^{C} \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} I_{n,c,S\cdot h+i,S\cdot w+j} \cdot W_{o,c,i,j} + b_o \tag{1}$$

其中，$O_{n,o,h,w}$：输出特征图中的元素，$I_{n,c,S\cdot h+i,S\cdot w+j}$：输入特征图中的元素，$W_{o,c,i,j}$：卷积核中的权重，$b_o$：偏置项

卷积层的前向传播

```
def forward(self, input):
        """
        执行卷积层的前向传播。

        :param input: 形状为 [N, C, H, W] 的特征图。
        :return: 卷积输出，形状为 [N, O, output_H, output_W]。
        """
        # 第一步：应用填充
        if self.padding == "VALID":
            self.x = input
        if self.padding == "SAME":
            p = self.filter_size // 2
```

```
            self.x = np.lib.pad(input, ((0, 0), (0, 0), (p, p), (p, p)), "constant")

        # 第二步：调整输入维度以适应步幅
        self.adjust_input_dimensions()

        # 第三步：实现卷积
        N, _, H, W = self.x.shape
        O, C, K, K = self.weight.data.shape
        weight_cols = self.weight.data.reshape(O, -1).T
        x_cols = self.img2col(self.x, self.filter_size, self.filter_size,
            self.stride)
        result = np.dot(x_cols, weight_cols) + self.bias.data
        output_H, output_W = (H-self.filter_size)//self.stride + 1,
            (W-self.filter_size)//self.stride + 1
        result = result.reshape((N, result.shape[0]//N, -1)).reshape((N, output_H,
            output_W, O))
        return result.transpose((0, 3, 1, 2))
```

img2col 函数　在实验过程中，通过 for 循环进行卷积操作效率太低，经过查阅网络资料发现常在卷积操作中使用 img2col 函数，将卷积的部分的局部特征图平展为二维矩阵，然后再进行卷积中的乘法操作，以提高效率。其具体操作步骤如下：

1. 计算输出特征图的高度 output_H 和宽度 output_W，这是根据输入特征图的大小、卷积核的尺寸和步幅计算得到的。
2. 初始化一个全零的二维矩阵 x_cols，其形状为 [N *out_size, C * filter_size_x * filter_size_y]，其中 out_size 是输出特征图的大小。
3. 使用两层嵌套的循环遍历输入特征图，从左上角到右下角，以步幅为单位，提取每个局部区域，并将其展平成一维向量。然后，将这些一维向量按顺序填充到 x_cols 矩阵的对应位置。
4. 最终，返回生成的二维矩阵 x_cols，它包含了输入特征图中的所有局部区域的展平表示。

有了这个平展后的向量后再进行卷积操作，卷积效率得到了极大的提高。

```
    def img2col(self, x, filter_size_x, filter_size_y, stride):
        """
        将输入特征图转换为二维矩阵。
        :param x: 输入特征图，形状为 [N, C, H, W]。
        :param filter_size_x: 卷积核的尺寸x。
        :param filter_size_y: 卷积核的尺寸y。
        :param stride: 卷积步长。
        :return: 二维矩阵，形状为 [(H-filter_size+1)/stride *
            (W-filter_size+1)/stride*N, C * filter_size_x * filter_size_y]。
        """
```

```
        N, C, H, W = x.shape
        output_H, output_W = (H-filter_size_x)//stride + 1,
            (W-filter_size_y)//stride + 1
        out_size = output_H * output_W
        x_cols = np.zeros((out_size*N, filter_size_x*filter_size_y*C))
        for i in range(0, H-filter_size_x+1, stride):
            i_start = i * output_W
            for j in range(0, W-filter_size_y+1, stride):
                temp = x[:,:, i:i+filter_size_x, j:j+filter_size_y].reshape(N,-1)
                x_cols[i_start+j::out_size, :] = temp
        return x_cols
```

反向传播

- 在 eta 的行和列之间插入零，以处理步长大于 1 的情况。这是为了使梯度的大小与输入特征图的大小相匹配，因为步长大于 1 会减小梯度的大小。
- 计算本层的权重和偏置的梯度。这包括计算卷积核权重的梯度和偏置的梯度。
- 使用学习率 lr 更新权重和偏置。权重的更新采用梯度下降法，其中梯度除以批量大小 N 以进行平均。如果卷积层包含偏置，也会更新偏置。
- 进行边缘填充。这一步是为了处理填充类型为”VALID” 或”SAME” 的情况，以确保梯度的大小与输入特征图的大小匹配。
- 计算传递到上一层的梯度。通过反向传播计算，以确定梯度如何传播到卷积层的输入，从而进一步更新前一层的权重和偏置。

卷积层的反向传播

```
def backward(self, eta, lr):
    """
    反向传播，更新权重和计算传递到上一层的梯度。

    :param eta: 上一层返回的梯度 [N, O, output_H, output_W]。
    :param lr: 学习率。
    :return: 上一层的梯度。
    """
    # 在eta的行和列之间插入零，处理步长大于1的情况
    self.insert_zeros_in_eta(eta)
    # 计算本层的权重和偏置的梯度
    N, _, output_H, output_W = eta.shape
    self.calculate_gradients(eta, N, output_H, output_W)
    # 更新权重和偏置
    self.update_weights(lr, N)
```

```
        # 第四步：边缘填充
        self.pad_eta(eta)
        # 计算传递到上一层的梯度
        result = self.calculate_gradient_to_prev_layer(eta)
        return result
```

3.2　池化层

Lenet5 最初版本中的池化为平均池化，而在现代常用最大池化。而在手写数字体识别任务中，我认为使用最大池化效果更好。最大池化有助于保留图像中最显著的特征，这对于手写数字的识别是有益的，因为手写数字通常由笔画、边缘和角点等显著特征组成。最大池化能够强调局部最亮的区域，因此更容易捕捉到手写数字的笔画和边缘信息，这对于数字识别是非常关键的。于是我也分别对比了两种不同池化下对于模型训练收敛的速度和精确度进行比较判断，发现确实最大池化也更加合适。

$$O_{n,c,h,w} = \max_{i,j} I_{n,c,S\cdot h+i,S\cdot w+j} \tag{2}$$

其中，$O_{n,c,h,w}$：输出特征图中的元素，$I_{n,c,S\cdot h+i,S\cdot w+j}$：输入特征图中的局部区域，$\max_{i,j}$：表示在局部区域中找到最大值。

```
class Maxpooling:
    def __init__(self, kernel_size=(2, 2), stride=2, ):
        """
        :param kernel_size:池化核的大小(kx,ky)
        :param stride: 步长
        这里有个默认的前提条件就是: kernel_size=stride
        """
        self.ksize = kernel_size
        self.stride = stride

    def forward(self, input):
        """
        :param input:feature map形状[N,C,H,W]
        :return:maxpooling后的结果[N,C,H/ksize,W/ksize]
        """
        N, C, H, W = input.shape
        input_grid = input.reshape(N, C, H//self.stride, self.stride,
            W//self.stride, self.stride)
        out = np.max(input_grid,axis=(3,5))
        self.mask = out.repeat(self.ksize[0], axis=2).repeat(self.ksize[1], axis=3)
            != input
        return out

```

```
22      def backward(self, eta):
23          """
24          :param eta:上一层返回的梯度[N,O,H,W]
25          :return:
26          """
27          result = eta.repeat(self.ksize[0], axis=2).repeat(self.ksize[1], axis=3)
28          result[self.mask] = 0
29          return result
```

3.3　全连接层

前向传播

$$y = x \cdot W + b \tag{3}$$

其中 y：输出 x：输入特征 w：权重矩阵 b：偏置项

反向传播　计算传递到下一层的梯度 next_eta：

$$\frac{\partial L}{\partial W} = X^T \eta, \quad \frac{\partial L}{\partial b} = \sum_{i=1}^{N} \eta_i$$

$$W = W - \mathrm{lr} \cdot \frac{\partial L}{\partial W}, \quad b = b - \mathrm{lr} \cdot \frac{\partial L}{\partial b}$$

$$\text{next\_eta} = \eta W^T$$

其中：L 是损失函数，W 是权重矩阵，b 是偏置向量，X 是输入，η 是从上一层传入的梯度。

3.4　归一化处理

Batch Normalization 层是一种用于神经网络的标准化技术。它被广泛用于深度学习模型中，特别是在卷积神经网络（CNN）和全连接神经网络中。

在训练的过程中，为了防止梯度爆炸，决定加入 BN 层对数据进行归一化处理。BN 层的主要目的是加速神经网络的训练过程，减少训练时间，并提高模型的性能。它通过对每个批次的输入进行标准化，即通过调整输入的均值和方差，使其具有零均值和单位方差。这有助于缓解梯度消失问题，加速训练收敛，以及提高模型的鲁棒性。

BN 的操作步骤如下：

1. 对于每个 mini-batch 中的输入数据，计算均值和方差。
2. 对输入数据进行标准化，将其转换为零均值和单位方差。
3. 通过缩放和平移操作，对标准化后的数据进行线性变换。
4. 将缩放和平移后的数据作为 BN 层的输出，供下一层使用。

考虑到在深层网络中，随着网络层数的增加，梯度的传播可能会导致梯度消失或梯度爆炸的问题。于是通过添加 BN 层通过标准化输入，有助于维持适度的梯度大小，从而改善梯度的传播。BN 层应当加在**激活函数之前**和**全连接层之后**。

3.5　随机丢弃

为了防止模型过拟合，我在实验过程中加入了 dropout 层，dropout 通过随机丢弃神经元，减少网络模型对于特定神经元的依赖，使得模型具有较强的鲁棒性。减少过拟合对于实验精确度的影响。

4　实验结果与分析

经过 10 轮训练后，准确率达到了 **95.98%**。

```
Epoch1: batch:170       Batch acc:0.5391        Batch loss:0.02173
...
Epoch10:        batch:930       Batch acc:0.9797        Batch loss:0.001117
[Epoch10]       Tarin accuracy:0.9589   Tarin loss:0.002127
[Epoch10]       Test Accuracy:0.9598
```

图 4.4: 模型结果

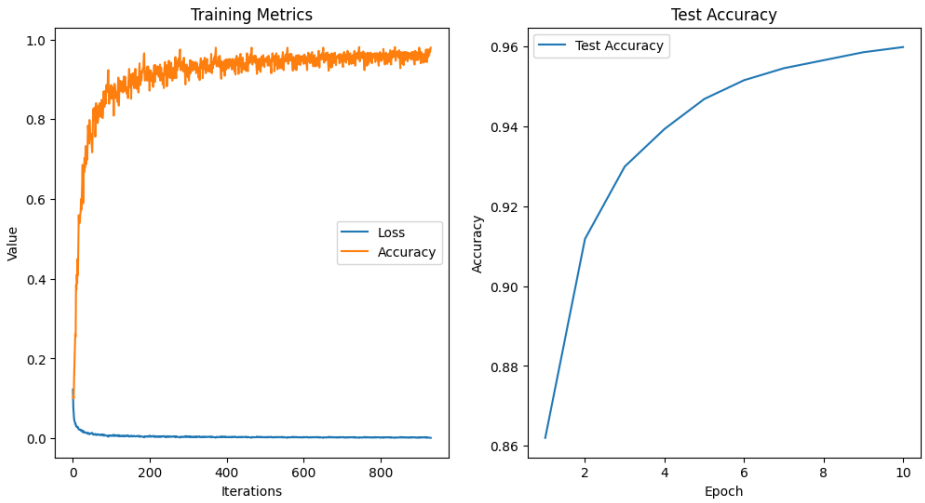

图 4.5: 实验结果

本次实验在学习了 Lenet5 的网络的基本架构之后，搭建了基本网络。同时通过调整参数来探讨对于模型训练的影响，最后得到了一个较为理想的准确率。之后在探索过程中学习到了 img2col 加快卷积速度的操作、归一化处理数据减小梯度爆炸和 dropout 层防止模型过拟合等深度学习中所常使用到的方法。同时还探讨了模型早期使用的平均池化和最大池化对于模型的影响，发现最大池化的效果要优于平均池化，在前面的池化层中也有所提及。

5 实验总结

本次实验让我第一次动手实现了一个小型的深度学习模型，通过不调用库函数的方式实现，增强了我对模型细节的理解，也提高了我的动手能力。在作业开始初期，由于对于这方面的知识不太了解，对于卷积的公式推导、反向传播、lenet5 模型等基本架构等，参考了许多网络上的文章或博客，摘录于下。

卷积神经网络 CNN 与 LeNet5 详解

卷积神经网络（CNN）开山之作——LeNet-5

Softmax 与交叉熵损失的实现及求导

Numpy 实现神经网络框架 (7)——Batch Normalization