人工智能实验：基于numpy实现MNIST数字图像分类

1 实验内容

本实验基于 **numpy** 库搭建 **LeNet5** 神经网络来完成对 **MNIST** 数字手写体字符的自动识别。**MNIST** 数据集来自美国国家标准与技术研究所，共由以下四部分组成：

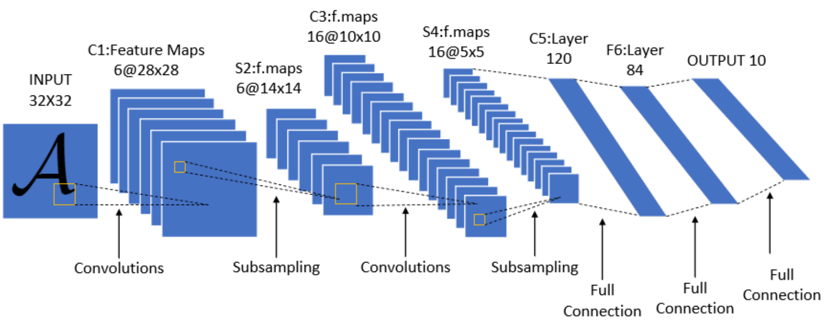
- Training set images: train-images-idx3-ubyte.gz (9.9 MB, 解压后 47 MB, 包含 60000 个样本)
- Training set labels: train-labels-idx1-ubyte.gz (29 KB, 解压后 60 KB, 包含 60000 个标签)
- Test set images: t10k-images-idx3-ubyte.gz (1.6 MB, 解压后 7.8 MB, 包含 10000 个样本)
- Test set labels: t10k-labels-idx1-ubyte.gz (5KB, 解压后 10 KB, 包含 10000 个标签)

本实验主要目的在于理解深度学习训练的基本框架，包括数据处理、模型构建、训练与测试等。

2 实验设计

2.1 LeNet5 总体架构

MNIST 字符识别选用 **LeNet5** 神经网络完成分类任务，其基本结构如下：



输入的二维图像，先经过两次卷积层到池化层，再经过全连接层，最后为输出层。

- 输入层 (INPUT) 是 32×32 像素的图像，其通道数为 1；
- C1 层是卷积层，使用 6 个 5×5 大小的卷积核，$\text{padding} = 0$，$\text{stride} = 1$ 进行卷积，得到 6 个 28×28 大小的特征图；
- S2 层是降采样层，使用 6 个 2×2 大小的卷积核进行池化，得到 6 个 14×14 的特征图，并通过激活函数非线性输出；
- S4 层也是降采样层，使用 16 个 2×2 大小的卷积核进行池化，得到 16 个 5×5 大小的特征图；
- F6 是全连接层，共有 84 个神经元，与 C5 层进行全连接，即每个神经元都与 C5 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 sigmoid 函数输出。
- 最后的 OUTPUT 层也是全连接层，本实验选用 Softmax 作为最后的多分类输出。

2.2 卷积层

实验中定义 `Conv2d` 类实现了一个简单的二维卷积层：

```
class Conv2d:
    def __init__(self, in_channels: int, out_channels: int, kernel_size: int,
                 stride: int = 1, padding: int = 0, dtype=None):
        self.x = None
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.kernel_size = kernel_size
```

```
        self.stride = stride
        self.padding = padding
        self.dtype = dtype
        self.weight = np.random.randn(out_channels, in_channels, kernel_size, kernel_size)
        self.bias = np.zeros(out_channels)
        self.w_grad = np.zeros_like(self.weight)
        self.b_grad = np.zeros_like(self.bias)
```

`in_channels`: 输入数据的通道数; `out_channels`: 卷积操作后输出的通道数; `kernel_size`: 卷积核的尺寸;

`stride`: 卷积操作的步幅; `padding`: 填充大小;

`self.weight`: 随机初始化的卷积核权重, 形状为 `(out_channels, in_channels, kernel_size, kernel_size)`;

`self.bias`: 初始化为零的偏置; `self.w_grad` 和 `self.b_grad`: 存储梯度的数组, 与权重和偏置形状相同。

前向传播过程实现如下:

```
def forward(self, x):
    self.x = x
    (N, C, H, W) = x.shape
    h_out = (H + 2 * self.padding - self.kernel_size) // self.stride + 1
    w_out = (W + 2 * self.padding - self.kernel_size) // self.stride + 1
    x_pad = np.pad(x, ((0, 0), (0, 0), (self.padding, self.padding), (self.padding,
self.padding)), 'constant')
    out = np.zeros((N, self.out_channels, h_out, w_out))

    for i in range(h_out):
        for j in range(w_out):
            x_window = x_pad[:, :, i * self.stride:i * self.stride + self.kernel_size,
                                 j * self.stride:j * self.stride + self.kernel_size]
            out[:, :, i, j] = np.sum(x_window[:, np.newaxis, :, :, :] * self.weight[np.newaxis,
:, :, :, :],axis=(2, 3, 4)) + self.bias

    return out
```

反向传播过程实现如下:

```
def backward(self, dy, lr):
    (N, O, H_out, W_out) = dy.shape
    (N, C, H, W) = self.x.shape
    x_pad = np.pad(self.x, ((0, 0), (0, 0), (self.padding, self.padding), (self.padding,
self.padding)), 'constant')
    dx_pad = np.zeros_like(x_pad)

    # Initialize gradients
    self.w_grad.fill(0)
    self.b_grad.fill(0)

    # Compute gradient for bias
    self.b_grad = np.sum(dy, axis=(0, 2, 3))

    # Compute gradients for weights and input
    for i in range(H_out):
        for j in range(W_out):
            x_window = x_pad[:, :, i * self.stride:i * self.stride + self.kernel_size,
                           j * self.stride:j * self.stride + self.kernel_size]
            dy_expanded = dy[:, :, i, j][:, :, np.newaxis, np.newaxis, np.newaxis]
            self.w_grad += np.sum(dy_expanded * x_window[:, np.newaxis, :, :, :], axis=0)
            dx_pad[:, :, i * self.stride:i * self.stride + self.kernel_size,
            j * self.stride:j * self.stride + self.kernel_size] += np.sum(
                dy_expanded * self.weight[np.newaxis, :, :, :, :], axis=1)
```

```python
    # Update weights and biases
    self.weight -= lr * self.w_grad
    self.bias -= lr * self.b_grad

    # Remove padding from dx_pad
    if self.padding > 0:
        dx = dx_pad[:, :, self.padding:-self.padding, self.padding:-self.padding]
    else:
        dx = dx_pad

    return dx
```

2.3 池化层

本实验采用的池化操作为平均池化，这种方式得到的特征信息对背景信息更加敏感，可以帮助更好完成数字分类。

平均池化的前向传播及反向传播过程实现如下：

```python
class AvgPool2d:

    # 初始化方法，设置卷积核大小
    def __init__(self, kernel_size: int):
        self.x = None
        self.kernel_size = kernel_size

    # 前向传播方法，计算平均池化
    def forward(self, x):
        self.x = x
        (N, C, H, W) = x.shape
        stride = self.kernel_size
        h_out = H // self.kernel_size
        w_out = W // self.kernel_size
        out = np.zeros((N, C, h_out, w_out))
        for i in range(h_out):
            for j in range(w_out):
                x_window = x[:, :, i * stride:i * stride + self.kernel_size,
                                 j * stride:j * stride + self.kernel_size]
                out[:, :, i, j] = np.mean(x_window, axis=(2, 3))
        return out

    # 反向传播方法，计算梯度
    def backward(self, dy):
        (N, C, H, W) = self.x.shape
        stride = self.kernel_size
        h_out, w_out = dy.shape[2], dy.shape[3]
        dx = np.zeros_like(self.x)
        for i in range(h_out):
            for j in range(w_out):
                dx[:, :, i * stride:i * stride + self.kernel_size,
                   j * stride:j * stride + self.kernel_size] += dy[:, :, i, j][:, :, np.newaxis,
np.newaxis] / (self.kernel_size * self.kernel_size)
        return dx
```

2.4 线性层

线性层将输入数据展平成一个一维向量，并对这些特征进行加权求和，从而组合特征。这种组合可以捕捉到高阶特征之间的线性关系，有助于提高模型的表达能力。

代码实现如下：

```python
class Linear:
    def __init__(self, in_features: int, out_features: int, bias: bool = True):
        # Initialize parameters and gradients for the linear layer
        self.x = None
        self.in_features = in_features
        self.out_features = out_features
        self.weight = np.random.randn(out_features, in_features)
        self.bias = np.zeros(out_features) if bias else None
        self.w_grad = np.zeros_like(self.weight)
        self.b_grad = np.zeros_like(self.bias) if bias else None

    def forward(self, x):
        # Compute the output for forward propagation
        self.x = x
        out = np.dot(x, self.weight.T)
        if self.bias is not None:
            out += self.bias
        return out

    def backward(self, dy, lr):
        # Compute gradients and update parameters for backward propagation
        self.w_grad = np.dot(dy.T, self.x)
        self.b_grad = np.sum(dy, axis=0)
        dx = np.dot(dy, self.weight)
        self.weight -= lr * self.w_grad
        if self.bias is not None:
            self.bias -= lr * self.b_grad
        return dx
```

2.5 损失函数

本实验选用交叉熵损失函数:

$$H(y_i, \hat{y}_i) = -y_i * \log \hat{y}_i$$

y 是样本 x 分类的真实概率，$\hat{y}$ 是模型预测概率。

其代码实现如下:

```python
class CrossEntropyLoss:
    def __init__(self):
        self.softmax = None
        self.label = None

    def forward(self, x, label):
        n = x.shape[0]
        x_exp = np.exp(x - np.max(x, axis=1, keepdims=True))
        softmax = x_exp / np.sum(x_exp, axis=1, keepdims=True)
        self.softmax = softmax
        self.label = label

        # 防止log(0)的情况
        softmax = np.clip(softmax, 1e-10, 1.0)
        loss = -np.sum(np.log(softmax[np.arange(n), label])) / n
        return loss

    def backward(self, x, label):
        n = x.shape[0]
        dx = self.softmax
        dx[np.arange(n), label] -= 1
        dx /= n
        return dx
```

2.6 激活函数

本实验均选用 Sigmoid 作为激活函数：

$$\sigma(x)=\frac{1}{1+e^{-x}}$$

由表达式可知，其输出值始终在 0 和 1 之间，使得它常被用于那些要求输出概率的模型中。

Sigmoid 激活函数的导数形式如下：

$$\sigma'(x)=\sigma(x)\cdot(1-\sigma(x))$$

其缺点也很明显：在输入值的绝对值非常大时，梯度会变得非常小，可能导致反向传播过程中的梯度消失问题。

前向传播过程：

```
def forward(self, x):
    self.x = x
    return 1 / (1 + np.exp(-x))
```

反向传播过程：

```
def backward(self, dy):
    sig = self.forward(self.x)
    return dy * sig * (1 - sig)
```

2.7 numpy && tqmd 库

本实验主要用的库为 numpy 与 tqmd；tqmd 用于在 Python 循环中添加一个进度条显示，适用于处理长时间运行的任务；

numpy提供了对大型多维数组和矩阵的支持，此外还提供了大量的数学函数库来操作这些数组，大大增加了运行速度。

3 实验过程

该实验中分别调整学习率、训练轮次、每批次样本数量来探究训练效果。

3.1 设置不同的学习率（learning rate）

1. 设置学习率为 0.4，训练 30 轮，每轮训练 64 张图片；
2. 设置学习率为 0.1，训练 30 轮，每轮训练 64 张图片；
3. 设置学习率为 0.05，训练 30 轮，每轮训练 64 张图片；
4. 采用 Step Decay 更新学习率，训练 30 轮，每轮训练 64 张图片：

```
if epoch < 5:
    lr = 0.4
elif 5 <= epoch < 10:
    lr = 0.1
else:
    lr = 0.05
```

5. 采用 Cosine Decay 更新学习率，训练 30 轮，每轮训练 64 张图片：

```
if epoch < 5:
   lr = 0.1 + (lr - 0.1) * epoch / 5
elif 5 <= epoch < num_epochs:
   lr = 0.4 * math.cos(math.pi / 2 * (epoch - 5) / (num_epochs - 5))
```

3.2 设置不同的训练轮次（epoch)

采用 Step Decay 更新学习率，每轮训练 64 张图片，训练 60 轮，观察学习率提升情况。

```
if epoch < 5:
    lr = 0.5
elif 5 <= epoch < 10:
    lr = 0.1
elif 10 <= epoch < 30:
    lr = 0.05
else:
    lr = 0.01
```

3.3 设置不同的批量大小（batch_size)

采用 Cosine Decay 更新学习率，训练 20 轮，batch_size 分别设置为 4、32、64、128、256，观察学习率变化情况。

3.4 比较不同数字识别准确率

每次训练后，神经网络对不同字符的识别效果显然是不同的，为此需要检测每个字符的识别准确率。实验中选取采用 Cosine Decay 更新学习率，训练 30 轮，每轮训练 64 张图片的训练模型观察不同数字的准确率大小。

4 实验结果及分析

4.1 调整 learning rate

lr = 0.4 时，结果如下表所示：

epoach	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Accuracy(%)	83.18	88.48	92.12	93.08	94.16	93.78	94.39	96.7	95.44	96.7
epoach	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
Accuracy(%)	96.78	96.87	96.95	97.06	97.08	97.17	97.25	97.21	97.23	97.27
epoach	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**
Accuracy(%)	97.3	97.26	97.31	97.34	97.32	97.4	97.35	97.32	97.26	97.3

lr = 0.1 时，结果如下表所示：

epoach	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Accuracy(%)	86.15	90.03	91.94	93.02	93.92	94.5	94.99	95.39	95.66	95.87
epoach	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
Accuracy(%)	96.1	96.25	96.37	96.53	96.6	96.71	96.89	96.94	96.99	97.02
epoach	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**
Accuracy(%)	97.07	97.12	97.18	97.23	97.31	97.35	97.41	97.45	97.47	97.5

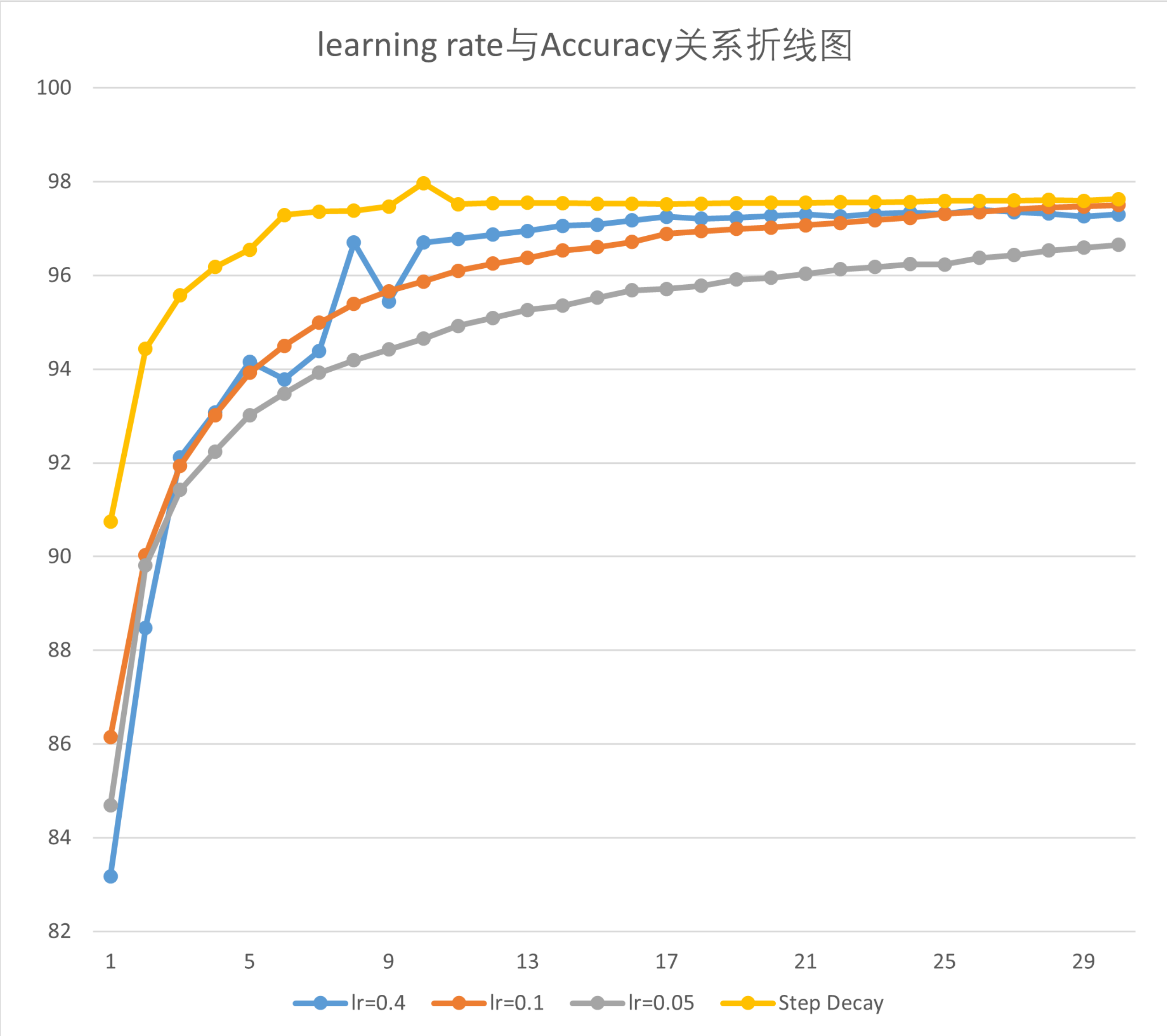
lr = 0.05 时，结果如下表所示：

epoach	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Accuracy(%)	84.69	89.81	91.42	92.24	93.02	93.48	93.92	94.19	94.42	94.65
epoach	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
Accuracy(%)	94.92	95.09	95.26	95.35	95.52	95.68	95.71	95.78	95.91	95.95
epoach	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**
Accuracy(%)	96.03	96.13	96.18	96.24	96.23	96.37	96.43	96.53	96.59	96.65

Step Decay 更新学习率：

epoach	1	2	3	4	5	6	7	8	9	10
Accuracy(%)	90.74	94.43	95.57	96.18	96.55	97.29	97.36	97.38	97.47	97.47
epoach	11	12	13	14	15	16	17	18	19	20
Accuracy(%)	97.52	97.54	97.55	97.54	97.53	97.53	97.52	97.53	97.54	97.74
epoach	21	22	23	24	25	26	27	28	29	30
Accuracy(%)	97.55	97.56	97.56	97.57	97.59	97.59	97.60	97.61	97.59	97.63

将结果用折线图表示如下:



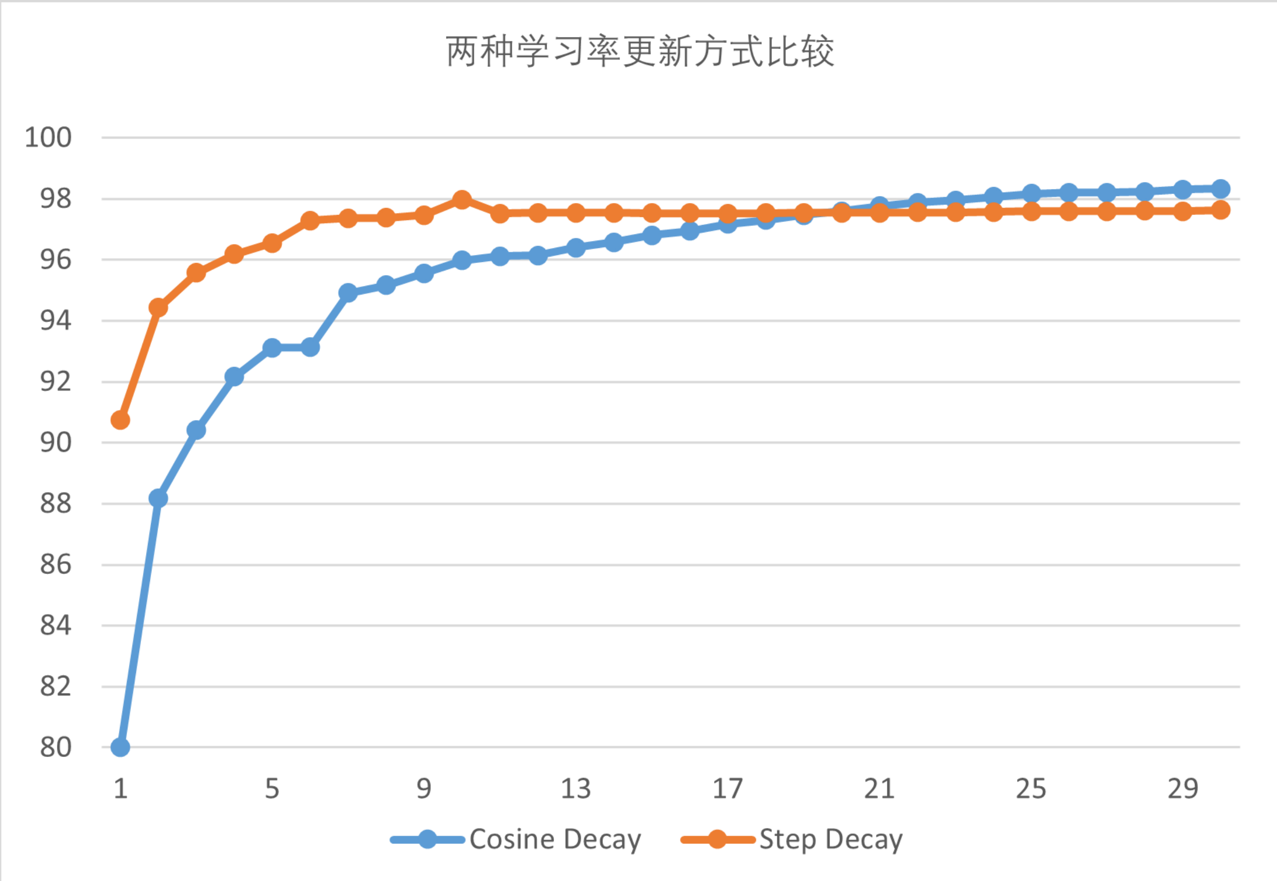
当学习率较大时，一开始准确率提升速度较快，但随着训练轮次的上升，准确率并不会一直增加，反而出现振荡情况，最终难以继续提升；当学习率较小时，虽然准确率会随着学习率的上升一直提升，但是准确率提升速度较慢，且有可能陷入局部最优。在学习率固定情况下，lr = 0.1 应该是训练效果最优的。

相较于固定的学习率， Step Decay 是一种更优化的学习率调度策略，它有助于在训练的早期阶段快速收敛，并在后期阶段通过降低学习率来细化参数，从而更快地收敛。

本实验还考虑了 Cosine Decay 更新学习率的方法:

epoach	1	2	3	4	5	6	7	8	9	10
Accuracy(%)	80.01	88.18	90.41	92.16	93.12	93.13	94.91	95.17	95.56	95.98
epoach	11	12	13	14	15	16	17	18	19	20
Accuracy(%)	96.12	96.14	96.4	96.58	96.8	96.95	97.18	97.3	97.47	97.6
epoach	21	22	23	24	25	26	27	28	29	30
Accuracy(%)	97.76	97.88	97.95	98.06	98.17	98.2	98.21	98.23	98.3	98.33

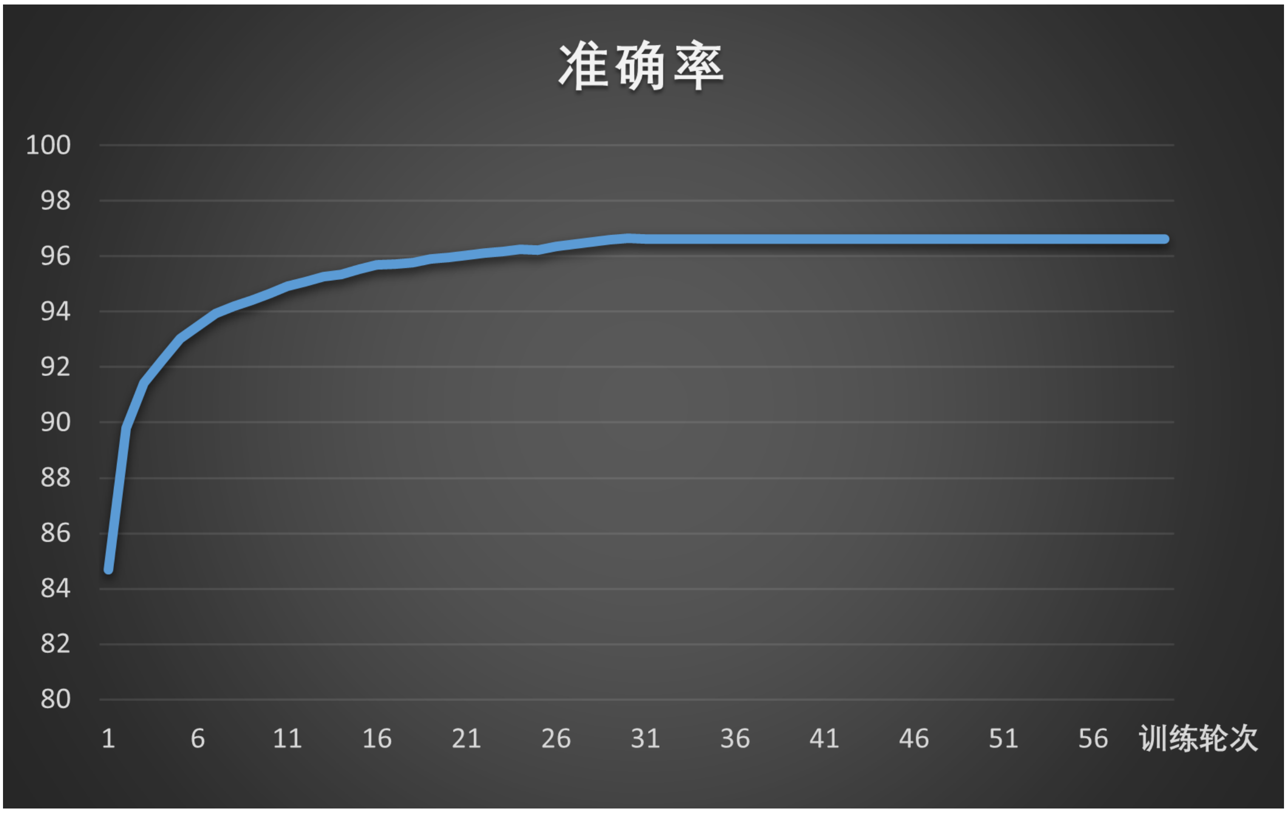
其与 Step Decay 的比较：



相对于 Step Decay，Cosine Decay 对于学习率的调整更加灵活，在本次训练中是更加优化的选择。

4.2 调整 epoch

采用 Step Decay 更新学习率，并将训练轮次由 30 轮提升至 60 轮：



训练到三十轮左右后，准确率无法进一步提升（保持在 96.63%），这导致了训练时间与资源的浪费。综合分析，这应该是学习率在 30 轮后设置的太小导致的，训练过程陷入局部最优解，无法有效优化。这也是 Step Decay 在本次实验中表现出的缺陷之一。

4.3 调整 batch_size

batch_size = 4 时，结果如下表所示：

epoach	1	2	3	4	5	6	7	8	9	10
Accuracy(%)	94.68	96.02	96.3	96.65	97.24	96.43	97.44	97.81	97.73	97.88
epoach	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
Accuracy(%)	97.91	98.49	98.42	98.38	98.2	98.37	98.34	98.52	98.6	98.64
epoach	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**
Accuracy(%)	98.77	98.78	98.8	98.83	98.84	98.86	98.9	98.95	98.96	98.99

batch_size = 32 时，结果如下表所示：

epoach	1	2	3	4	5	6	7	8	9	10
Accuracy(%)	87.92	90.66	92.38	93.85	94.4	94.53	94.9	95.98	96.86	96.64
epoach	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
Accuracy(%)	97.31	97.34	97.63	97.88	98.01	98.05	98.04	98.16	98.28	98.34
epoach	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**
Accuracy(%)	98.34	98.39	98.44	98.45	98.49	98.52	98.6	98.62	98.66	98.69

batch_size = 64 时，结果如下表所示：

epoach	1	2	3	4	5	6	7	8	9	10
Accuracy(%)	83.91	88.18	90.41	92.16	93.12	93.13	94.91	95.17	95.56	95.98
epoach	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
Accuracy(%)	96.12	96.14	96.4	96.58	96.8	96.95	97.18	97.3	97.47	97.6
epoach	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**
Accuracy(%)	97.76	97.88	97.95	98.06	98.17	98.2	98.21	98.23	98.3	98.33

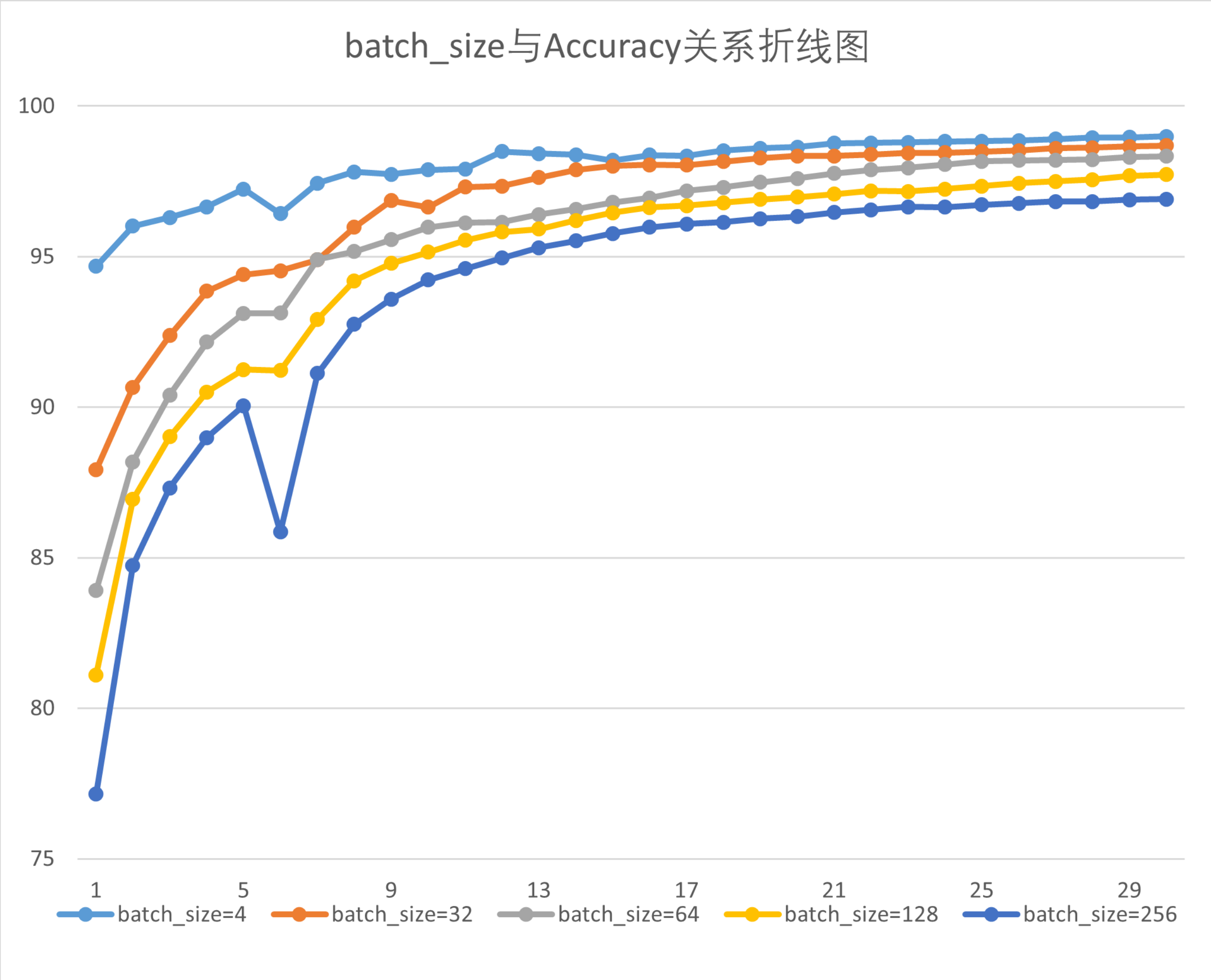
batch_size = 128 时，结果如下表所示：

epoach	1	2	3	4	5	6	7	8	9	10
Accuracy(%)	81.11	86.94	89.02	90.5	91.25	91.22	92.92	94.2	94.78	95.15
epoach	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
Accuracy(%)	95.54	95.82	95.92	96.2	96.46	96.63	96.69	96.79	96.9	96.98
epoach	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**
Accuracy(%)	97.08	97.18	97.16	97.24	97.34	97.44	97.5	97.56	97.69	97.73

batch_size = 256 时，结果如下表所示：

epoach	1	2	3	4	5	6	7	8	9	10
Accuracy(%)	77.15	84.75	87.32	88.99	90.05	85.86	91.12	92.75	93.59	94.23
epoach	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
Accuracy(%)	94.6	94.95	95.3	95.52	95.77	95.98	96.08	96.14	96.26	96.33
epoach	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**
Accuracy(%)	96.47	96.56	96.65	96.64	96.72	96.77	96.83	96.83	96.89	96.91

将结果用折线图表示：



根据以上结果可知，训练中采用小批量时，参数更新频率更高，可以加速收敛，使模型在更少的轮次内内达到较好的性能；不过实验中也遇到一个明显的问题，当 batch_size 较小时每一轮次训练时间显著增加，收敛过程需要更多的时间。

查阅资料后发现，大批量训练和小批量训练还有以下几点区别：

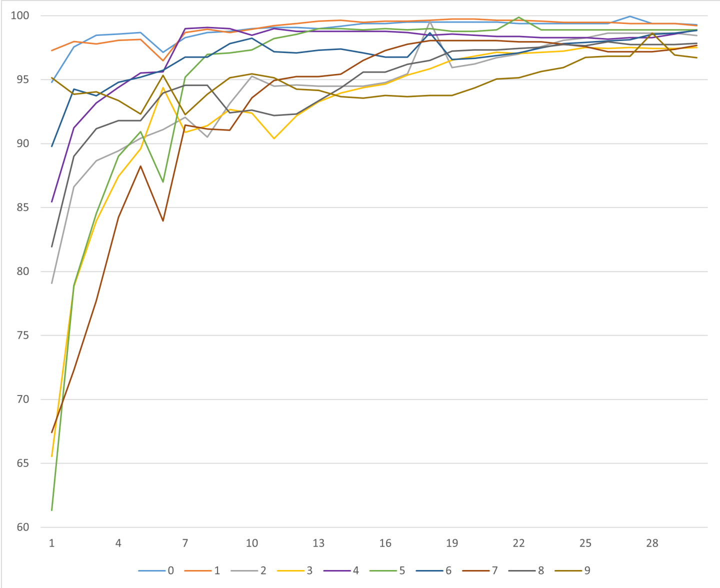
大批量提供更准确的梯度估计，减少梯度的方差，使训练过程更加稳定，而小批量梯度估计的不稳定性较高；

大批量训练可以更好地利用硬件的并行计算能力，而小批量训练每次计算样本较少，并行计算能力不能完全发挥；

大批量训练需要更多的内存来存储数据和梯度，有硬件资源的限制，小批量占用更少的内存，适合资源有限的情况。

4.4 不同字符识别效果

如下图：图例 0 – 9 分别代表了每个识别数字，纵坐标为准确率 Accuracy%

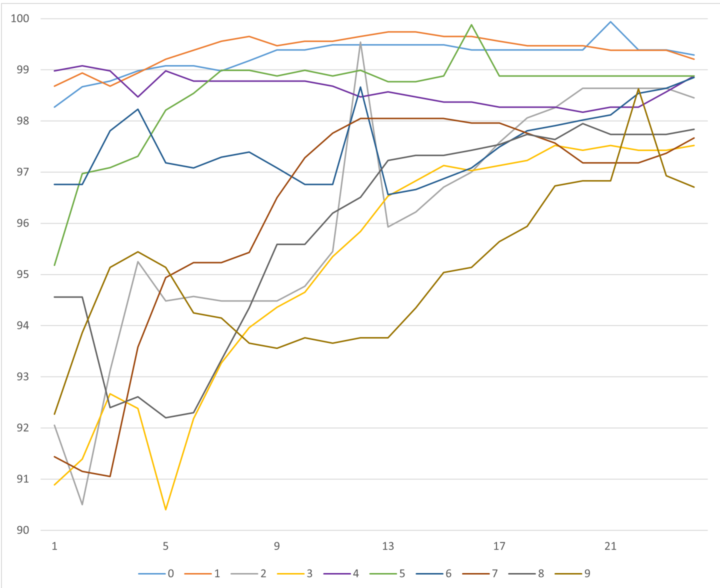


从图中可知：

3、5、7 的识别准确率在训练前几轮显著低于其他字符；

0、1 从第一轮训练结束后就始终保持着较高的识别准确率。

为了更好的分析结果，选取 7 轮训练后的结果，将纵坐标范围设置为 90% – 100% 进行观察：



从图中不难发现，不同字符的识别效果均有振荡情况，0、1 的识别效果最好，9 的识别效果最不好。

查阅资料可知，不同字符的识别准确率差异可能是由多个因素引起的，如：

字符形状的复杂性：形状复杂的字符（例如 8 和 5 ）可能比形状简单的字符（例如 1 和 0 ）更难识别；

字符的相似度：一些字符之间的形状非常相似（例如 1 和 7，3 和 8），因此容易被误识别。

同时，训练模型和训练参数也会影响不同字符训练准确率差异，比如在 batch_size 为128 的条件下，根据实验结果，最终识别效果最差的数字是 7。因此，分析不同数字识别准确率的差异较为复杂，但合理调整参数最终都能达到理想训练效果。

5 模型总结

总的来说，该模型完成了一个完整的手写体字符分类流程，如果合理设置参数，在几轮训练后能达到 97% 以上的准确率，同时训练时长较短（每轮5分钟左右）。当然，该模型也存在以下几点不足：

LeNet-5与现代深度学习架构相比，可能在性能上有所不足；

整个网络仅使用了 Sigmoid 激活函数，这可能导致梯度消失或爆炸问题；

虽然对数据进行了归一化处理，但没有进行其他数据增强操作，这可能限制了模型对新数据的泛化能力。

以下是一种种我觉得可能有用的改进方法：**在两次卷积操作后添加 Batch Normalization 层**

Batch Normalization 通过将每一层的输入标准化为均值为 0、方差为 1，归一化后的数据更接近于标准正态分布，能使梯度下降更快地收敛；同时，其助于减轻训练过程中的梯度消失和梯度爆炸问题，尤其是在深层网络中。这使得网络训练更加稳定；不仅如此，使用 BatchNorm 后，对参数初始化的要求降低，因为每一层的输入都会进行归一化处理，使得不好的初始化不会对训练过程产生过大影响。

代码实现如下：

```
class BatchNorm:
    def __init__(self, num_features, eps=1e-5, momentum=0.1):
        self.num_features = num_features
        self.eps = eps
        self.momentum = momentum

        # Parameters
        self.gamma = np.ones((1, num_features, 1, 1))
        self.beta = np.zeros((1, num_features, 1, 1))

        # Running statistics
        self.running_mean = np.zeros((1, num_features, 1, 1))
        self.running_var = np.ones((1, num_features, 1, 1))

        # Cache for backward pass
        self.cache = None

    def forward(self, x, training=True):
        if training:
            batch_mean = np.mean(x, axis=(0, 2, 3), keepdims=True)
            batch_var = np.var(x, axis=(0, 2, 3), keepdims=True)

            # Normalize
            x_normalized = (x - batch_mean) / np.sqrt(batch_var + self.eps)

            # Scale and shift
            out = self.gamma * x_normalized + self.beta

            # Update running statistics
            self.running_mean = self.momentum * batch_mean + (1 - self.momentum) *
self.running_mean
            self.running_var = self.momentum * batch_var + (1 - self.momentum) *
self.running_var

            # Save cache for backward pass
            self.cache = (x, x_normalized, batch_mean, batch_var)
        else:
            # Use running statistics for inference
            x_normalized = (x - self.running_mean) / np.sqrt(self.running_var + self.eps)
            out = self.gamma * x_normalized + self.beta
```

```
        return out

    def backward(self, dy, lr=None):
        x, x_normalized, mean, var = self.cache
        (N, C, H, W) = x.shape

        # Gradients w.r.t. parameters
        dgamma = np.sum(dy * x_normalized, axis=(0, 2, 3), keepdims=True)
        dbeta = np.sum(dy, axis=(0, 2, 3), keepdims=True)

        # Gradients w.r.t. normalized x
        dx_normalized = dy * self.gamma

        # Gradients w.r.t. input
        dvar = np.sum(dx_normalized * (x - mean) * -0.5 * (var + self.eps) ** -1.5, axis=(0, 2,
3), keepdims=True)
        dmean = np.sum(dx_normalized * -1 / np.sqrt(var + self.eps), axis=(0, 2, 3),
keepdims=True) + dvar * np.sum(-2 * (x - mean), axis=(0, 2, 3), keepdims=True) / N / H / W
        dx = dx_normalized / np.sqrt(var + self.eps) + dvar * 2 * (x - mean) / N / H / W + dmean
/ N / H / W

        # Update parameters
        self.gamma -= lr * dgamma
        self.beta -= lr * dbeta

        return dx
```

可能是因为 LeNet5 对 MNIST 手写体字符识别本来就有很好的效果，本次实验添加 BatchNorm 后训练优化效果不是特别明显，希望在日后的学习过程中能够进一步探索。