人工神经网络实验报告

作业 2 • CNN

周正平，计 54，2015011314，zhouzp15@mails.tsinghua.edu.cn

2017 年 10 月 23 日

Contents

1 实验内容

本次实验要求设计卷积神经网络（CNN）用于完成 MNIST 手写数字识别任务。以含有 2 个卷积层的网络为例，实验中所用网络结构如下：

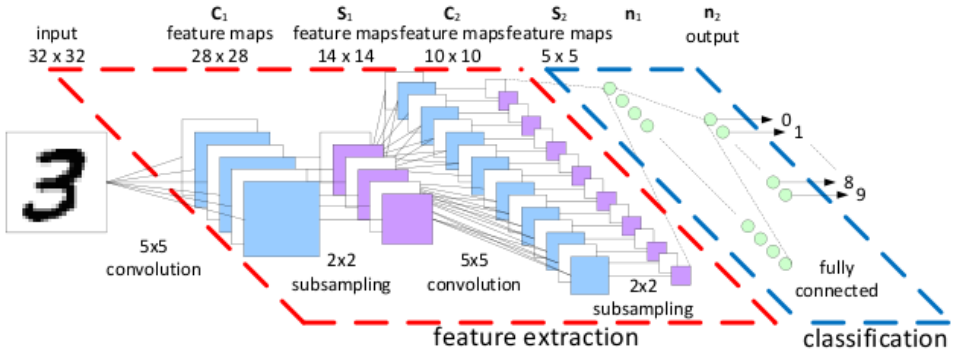

Figure 1: 网络结构总览

总体来说，本次作业需要在给出的框架基础上，实现以下几个模块：

1. **卷积层**：图中的 C_1 和 C_2，公式为 $output = \sum_{k_1=1}^{K_1} \sum_{k_2=1}^{K_2} input[m-k_1, n-k_2] W[k_1, k_2]$；
2. **池化层**：图中的 S_1 和 S_2，本次作业实现不重叠的平均值采样。
3. **Softmax 损失函数**：该模块已在 MLP 实验中实现，本次实验中对上溢、下溢 2 种情形进行了处理。

2 算法实现

算法中涉及的主要参数如下：

变量名	描述
n	每个 batch 中样例的数量
C_{in}/C_{out}	输入/输出图像的通道数
H_{in}/H_{out}	输入/输出图像的高度
W_{in}/W_{out}	输入/输出图像的宽度
k	卷积核的长/宽

2.1 im2col

在卷积层的算法实现中，需要用到一种重要的变换——im2col。该变换的性质及实现将在本节予以说明。

2.1.1 原理说明

参考 Caffe 作者贾扬清在知乎的回答 https://www.zhihu.com/question/28385679，卷积运算若暴力用 for 循环实现，则十分耗时，需进行向量化处理。通过将卷积操作向量化，笔者成功地避免了 for 循环，并获得了较高的运行效率。

由于卷积运算本质上，是输入图像中的各个图像块与卷积核逐元素相乘，故可以将输入图像 I 中所有这样的图像块都展开成列向量，再将这些列向量排列成一个矩阵 Feature Matrix。将 Feature Matrix 与卷积核 W 相乘，便得到了卷积的结果。

从输入 I 得到 Feature Matrix 的变换便是 im2col。在这一步变换中，在不考虑 pad 的情况下，一共有 $H_{out}W_{out}$ 个大小为 $k \times k$ 的图像块。将这些图像块各自表示为 k^2 的列再排列起来，便得到了 $H_{out}W_{out} \times C_{in}k^2$ 的 Feature Matrix。具体步骤如下所示：

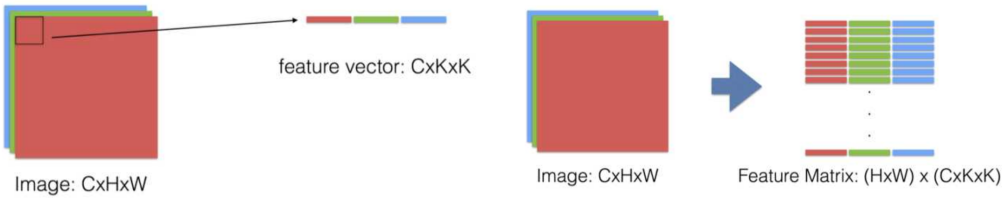

Figure 2: Feature Vector

Figure 3: Feature Matrix

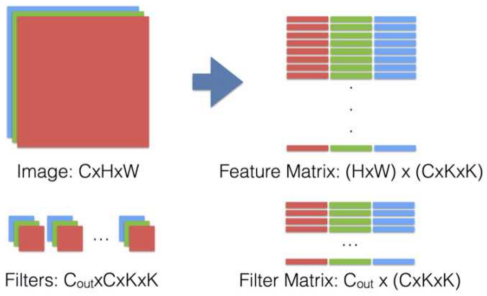

Figure 4: Filter Matrix & Feature Matrix

将 Feature Matrix 与卷积核展成的 $C_{out} \times C_{in}k^2$ 的 Filter Matrix 作一点乘，再进行 transpose、reshape 处理，便得到了卷积最终结果。

可以看出，im2col 变换本质上是一种坐标的对应关系，其逆变换 col2im 亦是如此。

2.1.2 算法实现

笔者实现了 'sliding' 和 'distinct' 两种模式的 im2col 与 col2im，分别对应于卷积核在移动的时候位置是否重叠（前者 stride 为 1，后者 stride 为 k）。相关算法的实现均位于文件 imutil.py 内：

函数名	参数	功能
im2col	input（输入向量） dims（各维度数值） mode（'sliding' 或 'distinct'）	将输入图像转化为图像块矩阵（Feature Matrix）
col2im	input（输入向量） dims（各维度数值） mode（'sliding' 或 'distinct'）	将图像块矩阵（Feature Matrix）转化回输入图像（注：该函数返回值相比原输入，对应元素位置放大了 N 倍，N 为该元素参与卷积运算的次数。元素越靠近边角，N 越小。可以证明这在计算 $grad\_input$ 时是正确的，此处略去。）
_im2col_2d_idx	dims（各维度数值）	计算 im2col 变换中的坐标对应关系

由于 im2col 及其逆变换 col2im 本质上是坐标变换，故算法的核心在于计算坐标的对应关系，由函数 _im2col_2d_idx 实现。其实现过程大致分为以下几步：

1. **计算 k^2 的块展平后在 input 中的坐标**

```
start_idx = np.arange(k)[:, None] * w_in + np.arange(k)
```

2. **计算块在 input 中移动时经过的偏移量**

需注意这里应对 'sliding'(overlapping) 和 'distinct'(non-overlapping) 两种情况进行分类讨论：

```
# 'sliding'
offset_idx = np.arange(h_out)[:, None] * w_in + np.arange(w_out)
```

```
# 'distinct'
offset_idx = np.arange(h_out * k, step=k)[:, None] * w_in + np.arange(w_out * k, step=k)
```

3. **计算在给定的偏移量下，大小为 k^2 的块在 input 中的坐标**

```
im2col_2d_idx = start_idx.ravel()[:, None] + offset_idx.ravel()
```

4. **将该坐标转化为行、列表示**

```
h_idx = im2col_2d_idx / w_in
w_idx = im2col_2d_idx % w_in
return h_idx, w_idx
```

在此基础上，im2col 函数只需将 input 中对应位置的元素取出重排，再做一点变形处理即可（以下给出核心代码，略去变形部分）：

```
input_col = input[:, :, h_idx, w_idx] # [n, c_in, k * k, h_out * w_out]
```

col2im 与此类似，只需将 Feature Matrix 中的图像块逐个加回原图对应位置即可。

```
input = np.zeros([n, c_in, h_in, w_in])
np.add.at(input, (slice(None), slice(None), h_idx, w_idx), input_col)
```

2.2 卷积层 Forward

2.2.1 函数描述

卷积层对输入图像进行特征提取，利用多个卷积核，分多个通道提取多种特征，输出多张特征图（feature map）。

位置	变量名	形状	描述
Input	input	$n \times C_{in} \times H_{in} \times W_{in}$	输入图像
	W	$C_{out} \times C_{in} \times k \times k$	卷积核（共享权值）
	b	C_{out}	共享偏置
	kernel_size	scalar	卷积核高宽
	pad	scalar	上下左右补零宽度
Output	output	$n \times C_{out} \times H_{out} \times W_{out}$	输出的特征图

2.2.2 算法实现

整个 Forward 函数中，卷积实现大致分为如下几个步骤：

1. **im2col 变换**

   为 input 执行 im2col 变换，将其所有 $k \times k$ 的块取出，展开成 $H_{out} \times W_{out}$ 个长度为 k^2 的列向量，排列成一个 $H_{out}W_{out} \times C_{in}k^2$ 的矩阵（Feature Matrix）。W 也展为长度 k^2 的列向量，排列成一个 $C_{out} \times C_{in}k^2$ 的矩阵（Filter Matrix）。

   ```
   # Feature Matrix
   input_col = im2col(input_pad, dims, 'sliding')
   # Filter Matrix
   W_col = np.reshape(W, [c_out, c_in * k * k])
   ```

2. **矩阵相乘**

   经过 im2col 变换之后，卷积中间结果可表示为 $FeatureMatrix \cdot FilterMatrix^T$，得到大小为 $n \times H_{out}W_{out} \times C_{out}$ 的矩阵。

   ```
   output = np.matmul(input_col, W_col.T)
   ```

3. **变形转换**

   卷积中间结果经过进一步转置、变形后，即得卷积输出（$n \times C_{out} \times H_{out} \times W_{out}$）：

   ```
   output = output.transpose([0, 2, 1]).reshape([n, c_out, h_out, w_out])
   ```

2.3 卷积层 Backward

2.3.1 函数描述

本函数实现卷积层的反向传播算法。

位置	变量名	形状	描述
Input	input	$n \times C_{in} \times H_{in} \times W_{in}$	输入图像
	grad_output	$n \times C_{out} \times H_{out} \times W_{out}$	损失函数对输出的梯度
	W	$C_{out} \times C_{in} \times k \times k$	卷积核（共享权值）
	b	C_{out}	共享偏置
	kernel_size	scalar	卷积核高宽
	pad	scalar	上下左右补零宽度
Output	grad_input	$n \times C_{out} \times H_{in} \times W_{in}$	损失函数对输入的梯度
	grad_W	$n \times C_{out} \times H_{out} \times W_{out}$	损失函数对 W 的梯度
	grad_b	C_{out}	损失函数对 b 的梯度

2.3.2 算法实现

整个 Backward 函数大致分为以下几个步骤：

1. **变形逆变换**

   在 Forward 函数的最后，对卷积输出进行了变形处理（包括 transpose、reshape）。故在 Backward 中，需要首先对 grad_output 进行相应的逆变换：

```
grad_output = grad_output.reshape([n, c_out, h_out * w_out]).transpose([0, 2, 1])
```

2. **计算导数 grad_input**

   由于 Forward 中运用 im2col 变换将卷积转化为矩阵乘法，公式为 $output = input\_col \cdot W\_col^T$，故运用 MLP 的导数知识，可得 $grad\_input\_col = \frac{\partial E}{\partial input\_col} = grad\_output \cdot W\_col$。再对 grad_input_col 作一简单坐标变换 col2im，将图像块矩阵恢复为原图像即可。

```
grad_input_col = np.matmul(grad_output, W_col)
grad_input = col2im(grad_input_col, dims, 'sliding')
```

3. **计算导数 grad_W**

   同样地，利用 MLP 导数知识，可得 $grad\_W\_col = \frac{\partial E}{\partial W} = input\_col \cdot grad\_output$。再对 grad_W_col 作一个简单坐标变换，变形为 W 原形状即可。

```
grad_W_col = np.matmul(np.transpose(input_col, [0, 2, 1]), grad_output).sum(axis=0).T
grad_W = grad_W_col.reshape([c_out, c_in, k, k])
```

4. **计算导数 grad_b**

   grad_b 相对比较简单，只需对 grad_output 沿整个 batch、所有 channel 求和即可。

```
grad_b = np.sum(grad_output, axis=(0, 1))
```

2.4 池化层 Forward

2.4.1 函数描述

本实验中实现的为平均值池化（Average pooling），即将输入划分为不相交的 $k \times k$ 块，对每一块取平均值后作为输出。池化操作不改变输入图像的通道数，仅在 H, W 两个维度上进行压缩。

位置	变量名	形状	描述
Input	input	$n \times C_{in} \times H_{in} \times W_{in}$	输入图像
	kernel_size	scalar	池化尺寸
	pad	scalar	上下左右补零宽度
Output	output	$n \times C_{in} \times H_{out} \times W_{out}$	池化压缩后的特征图

2.4.2 算法实现

尽管可以使用 'distinct' 模式的 im2col 来实现 AvgPooling，然而笔者最终采用了一种较为浅显直观，并且效率也较高的算法：直接将 $H \times W$ 两个维度划分成 $k \times k$ 大小的块，并对每一块取平均值：

```
input_patch = np.reshape(input_pad, [n, c_in, h_in / k, k, w_in / k, k])
output = input_patch.mean(axis=3).mean(axis=4)
```

笔者曾尝试使用前面实现的 'distinct' 版本 im2col 实现 AvgPooling，发现程序可以正确收敛，但一方面代码逻辑更复杂，另一方面效率上也无显著优势。故最终笔者采取了上述简明写法（im2col 版本的 AvgPooling 详见 README）。

2.5 池化层 Backward

2.5.1 函数描述

本函数用于实现平均值池化的反向传播。

位置	变量名	形状	描述
Input	input	$n \times C_{in} \times H_{in} \times W_{in}$	输入图像
	grad_output	$n \times C_{out} \times H_{out} \times W_{out}$	损失函数对输出的梯度
	kernel_size	scalar	池化尺寸
	pad	scalar	上下左右补零宽度
Output	grad_input	$n \times C_{out} \times H_{in} \times W_{in}$	损失函数对输入的梯度

2.5.2 算法实现

与 Forward 类似，笔者最终选择直接将 grad_output 延拓：因为在求平均值的时候，所有元素对应的系数均为 $1/k^2$，故根据链式法则，可以直接将 grad_output 中每个元素替代为 $k \times k$ 的矩阵，再将所有元素除以 k^2，便得到了 grad_input：

```
grad_input = grad_output.repeat(k, axis=2).repeat(k, axis=3) * (1.0 / k / k)
grad_input = grad_input[:, :, pad:-pad, pad:-pad] if pad > 0 else grad_input
```

(im2col 版本的 AvgPooling 详见 README)

2.6 Softmax 损失函数

2.6.1 函数描述

Softmax 损失函数将输出转化为一个归一化的概率分布，并将其与标准答案对比，计算交叉熵作为损失值。

位置	变量名	形状	描述
Input	y	$n \times 10$	预测输出
	t	$n \times 10$	目标输出
Output	cross_entropy	scalar	交叉熵 loss

2.6.2 算法实现

在实验中，笔者发现，当通道数目较多时，$softmax = \frac{e^{y_k}}{\sum e^y}$ 中的 exp 会发生上溢（overflow）或下溢（underflow）。这是因为当 y 全部为绝对值很大的负数时，分母 $\sum e^y \to 0$，发生 underflow；而当 y 中存在绝对值很大的正数时，$e^y \to \infty$，发生 overflow。

笔者为了解决这个问题，查阅了相关资料，发现可以对 y 作变换 $y[n,:] \leftarrow y[n,:] - max(y[n])$。可以证明这样计算得到的 softmax 与原式完全相等。此时，分母上至少有一个 exp 值为 1，故不会发生 underflow；而此时 y 中所有元素均 ≤ 0，故不会发生 overflow。笔者依此实现，发现此处的上溢与下溢的确得到了有效解决：

```
    input -= np.max(input, axis=1)[:, np.newaxis]
```

在进行该变换之后，Forward 及 Backward 实现方法如下：

1. **Forward**

   根据公式 $f(x) = -\sum_k t_k \log p_k,\ p_k = \frac{e^{y_k}}{\sum e^y}$，代码实现如下：

   ```
       self.softmax = np.transpose(np.transpose(np.exp(input)) / np.exp(input).sum(axis=1))
       cross_entropy = -np.mean(np.sum(target * np.log(self.softmax), axis=1))
   ```

2. **Backward**

   根据公式 $f'(x) = p - t$，代码实现如下（需除以 batch_size）：

   ```
       return (self.softmax - target) / len(input)
   ```

3 实验结果

3.1 测试精度

对于有 2 个卷积层 +2 个池化层 +1 个线性层的网络，经调参优化后，在测试集上获得的最高精度为 98.58%。

网络配置如下：

Layer	**参数名称**	**取值**	**输出形状**
第一卷积层 conv1	in_channel	1	$N \times 4 \times 28 \times 28$
	out_channel	4	
	kernel_size	5	
	pad	2	
第一池化层 pool1	kernel_size	2	$N \times 4 \times 14 \times 14$
	pad	0	
第二卷积层 conv2	in_channel	4	$N \times 8 \times 14 \times 14$
	out_channel	4	
	kernel_size	5	
	pad	2	
第二池化层 pool2	kernel_size	2	$N \times 8 \times 7 \times 7$
	pad	0	
线性层 Linear	形状	$[8 \times 7 \times 7, 10]$	10
损失层 loss	损失函数	SoftmaxCrossEntropyLoss	scalar

训练超参数如下：

超参名称	**取值**
learning_rate	0.1（注：每隔 *5* 个 *epoch* 减半）
weight_decay	2e-4
momentum	1e-4
batch_size	50

3.2 acc/loss 曲线

绘制训练曲线如下（相邻采样点间隔 25 个 iteration）：

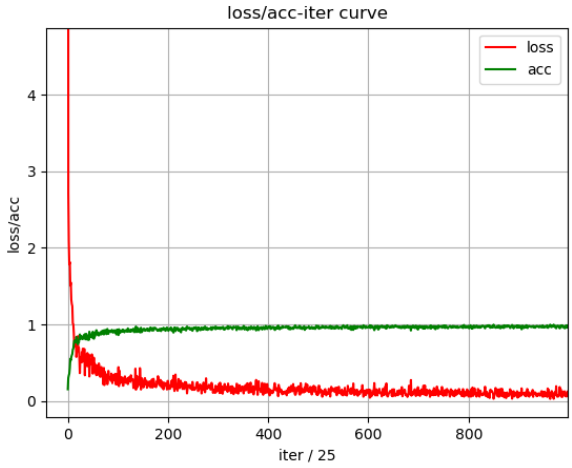

Figure 5: loss/acc-iter 训练曲线

前 30 个 epoch 的测试曲线如下（相邻采样点间隔 1 个 epoch）：

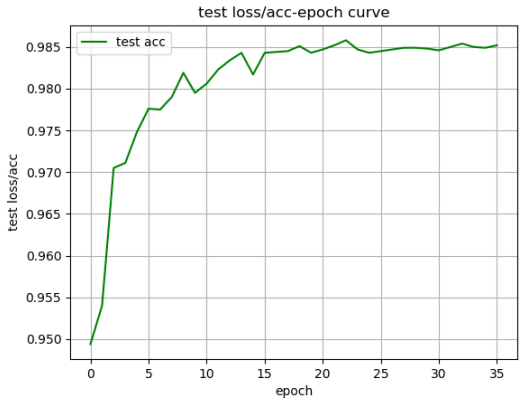

Figure 6: acc-epoch 测试曲线

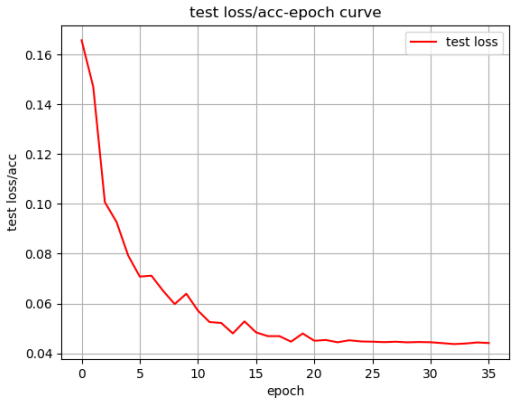

Figure 7: loss-epoch 测试曲线

4 CNN vs MLP

以下实验中，将涉及以下几种不同结构的网络，以下列出其在实验中所使用的默认超参数：

1. **双卷积层 CNN**：网络结构、训练设置与上节所述最优值相同；
2. **单隐层 MLP**：各超参数设置如下（同作业 1 中最优值）：

超参数	取值
隐层节点数	392
激活函数	Relu
损失函数	SoftmaxCrossEntropyLoss
learning_rate	0.1
weight_decay	0.0001
momentum	0.0001
batch_size	100
max_epoch	100

3. **双隐层 MLP**：各超参数设置如下（同作业 1 中最优值）：

超参数	取值
隐层 1 节点数	392
隐层 2 节点数	196
激活函数 1	Relu
激活函数 2	Relu
损失函数	EuclideanLoss
learning_rate	0.1
weight_decay	0.0002
momentum	0.0001
batch_size	100
max_epoch	100

4.1 训练表现

以下从训练时间、收敛性、精度 3 个方面分析 CNN 与 MLP 的差异：

网络类型	训练时间（30 个 epoch）	收敛性	精度
CNN	$\geq 2h$	收敛快，稳定	98.58%
MLP（单隐层）	2min	收敛较快	98.32%
MLP（双隐层）	3.5min	收敛较快	98.75%

训练曲线如下（相邻采样点间隔 50 个 iteration）：

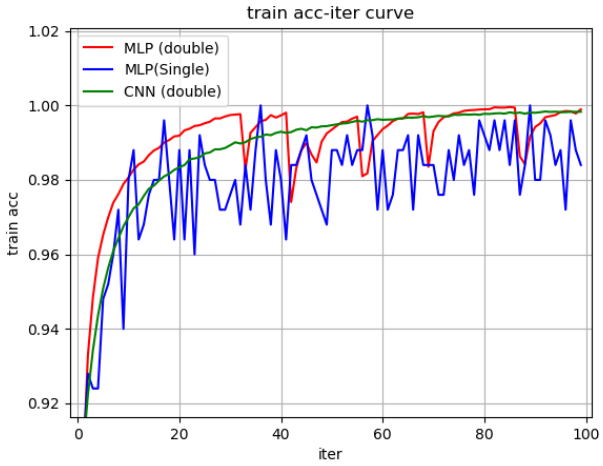

Figure 8: CNN vs MLP

绘制测试精度曲线如下（相邻采样点间隔 1 个 epoch，由于 CNN 网络训练实在过于缓慢，故笔者只对前 30 个 epoch 加以对比分析）：

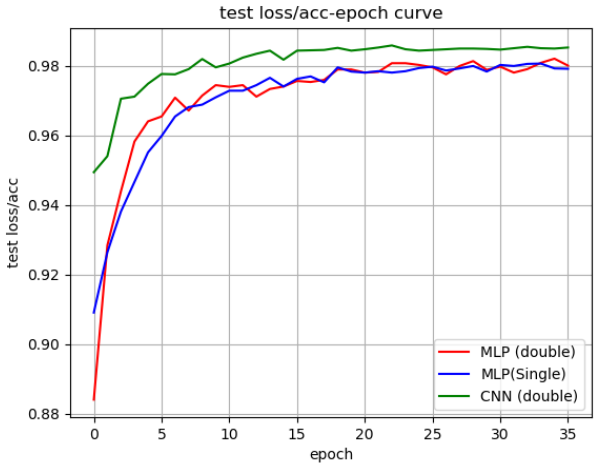

Figure 9: CNN vs MLP

从图中可以看出，尽管 CNN 相比 MLP 而言，训练极为缓慢，但随 epoch 收敛的速度在测试集上显著地快于 MLP。同时，就训练曲线而言，CNN 的收敛也更加稳定。从精度上看，CNN 与 MLP 相差不大，但这也可能是笔者训练时间不足，导致 CNN 未能完全收敛所致。尽管笔者在代码中避免了 for 循环，并尽可能使用 numpy 库函数，然而跑出 30 个 epoch 的数据还是花费了数个小时。

笔者认为，本次作业实现的 CNN 之所以训练速度如此缓慢，一来对比成熟的深度学习框架，例如 TensorFlow，通过 session 连接了高效的 C++ 后端，从而克服了 python 的效率瓶颈（笔者实测，运行结构相同的 CNN，TensorFlow 的运行效率至少是笔者程序的 100 倍左右），在多个层次上进行了更加深刻的优化；二来是因为无 GPU 加速支持，无法利用卷积运算的硬件优化来提高速度。

4.2 参数数目

CNN 相比 MLP 而言的一个显著优势便是独立参数的数目更少。

以下对比上述 3 种网络的参数数目：

网络类型	权值参数 W	偏置参数 b	总参数数目
CNN	$W_1 \to C_{out} \times C_{in} \times k \times k = 4 \times 1 \times 5 \times 5 = 100$ $W_2 \to C_{out} \times C_{in} \times k \times k = 8 \times 4 \times 5 \times 5 = 800$ $W_{Linear} \to 8 \cdot 7 \cdot 7 \times 10 = 3920$	$b_1 \to C_{out} = 4$ $b_2 \to C_{out} = 8$ $b_{Linear} \to 10$	4842
MLP（单隐层）	$W_1 \to 784 \times 392 = 307328$ $W_2 \to 392 \times 10 = 3920$	$b_1 \to 392$ $b_2 \to 10$	311650
MLP（双隐层）	$W_1 \to 784 \times 392 = 307328$ $W_2 \to 392 \times 196 = 76832$ $W_3 \to 196 \times 10 = 1960$	$b_1 \to 392$ $b_2 \to 196$ $b_3 \to 10$	386718

可以看出，在 3 种网络中，CNN 的参数最少，原因是其采取了共享权值及偏置量。相比全连接层而言，卷积层只关注相邻像素连接形成的特征，而忽略边远处像素的影响，从而节约了这部分参数。

5 可视化

笔者使用 pickle 对模型进行了导出，并使用如下 4 张图片进行模型的可视化（可视化代码实现见 visualize.py），并采用 2 个卷积层均为 4 个输出通道的网络进行实验：

标签	模型预测标签	是否正确	图片
0	0	是	
3	3	是	
6	6	是	
9	9	是	

5.1 第一卷积层

5.1.1 输出

第一卷积层在 Relu 后的输出（共 4 个输出通道）可视化如下：

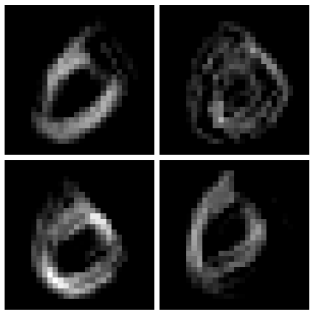

Figure 10: 数字 0

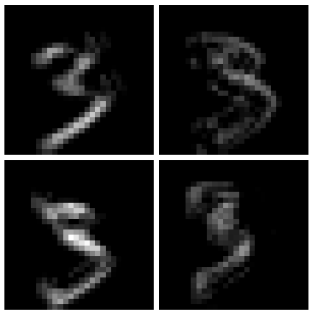

Figure 11: 数字 3

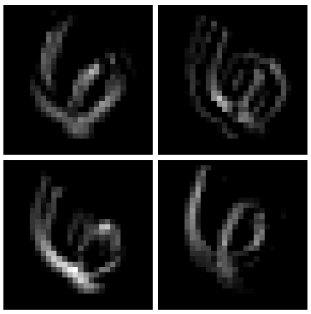

Figure 12: 数字 6

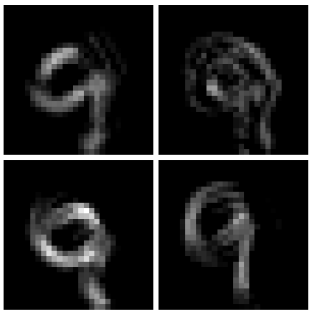

Figure 13: 数字 9

5.1.2 卷积核

第一卷积层的卷积核（共 1 个输入通道、4 个输出通道）可视化如下：

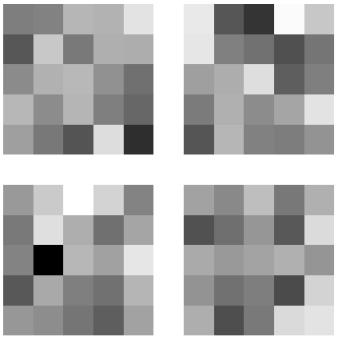

Figure 14: conv1 卷积核

5.2 第二卷积层

5.2.1 输出

第二卷积层在 Relu 后的输出（共 4 个输出通道）可视化如下：

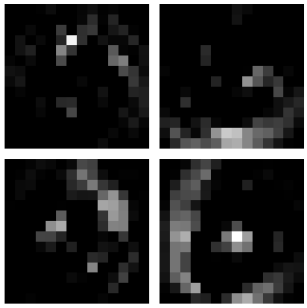

Figure 15: 数字 0

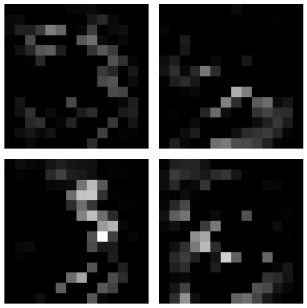

Figure 16: 数字 3

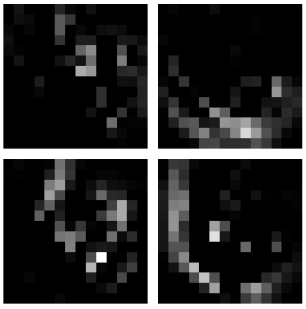

Figure 17: 数字 6

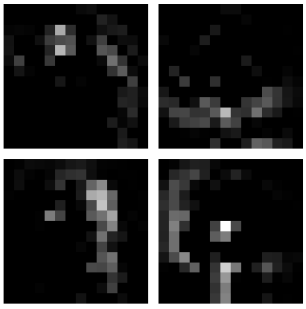

Figure 18: 数字 9

5.2.2 卷积核

第二卷积层的卷积核（共 4 个输入通道、4 个输出通道）可视化如下：

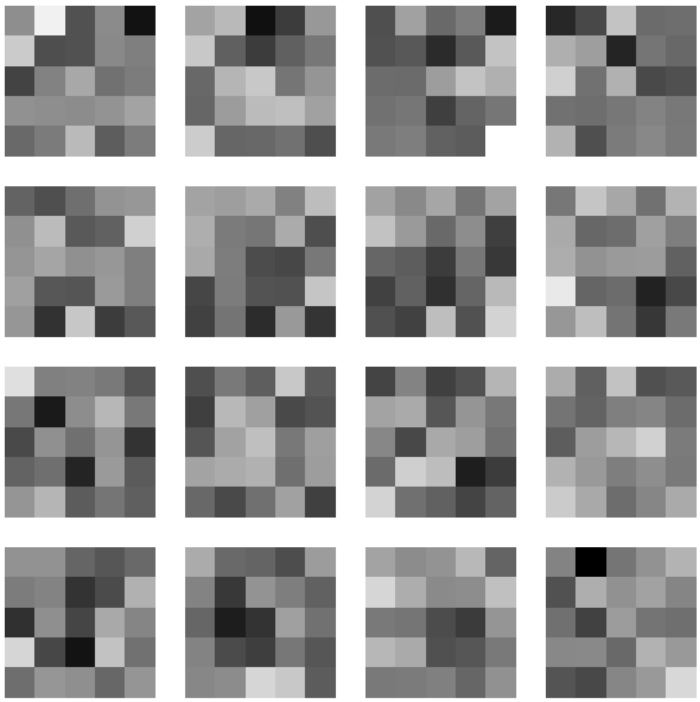

Figure 19: conv2 卷积核

References

[1] https://www.zhihu.com/question/28385679

[2] http://nbviewer.jupyter.org/github/BVLC/caffe/blob/master/examples/00-classification.ipynb

[3] https://wiseodd.github.io/techblog/2016/07/16/convnet-conv-layer/