实验报告-实验二

摘要

在本次实验中，我们小组完成了三层 BP 网络的错误反传算法，实现了百分之 70 以上的准确率。同时，我们就优化器、批量归一化、丢弃法等 BP 神经网络常见优化方法进行了探究。

关键字： BP 神经网络　卷积神经网络　批量归一化　丢弃法

一、 导言

构造一个三层的 BP 神经网络和一个卷积神经网络，完成手写 0-9 数字的识别：

1. 设计网络的结构，比如层数，每层的神经元数，单个神经元的输入输出函数；
2. 根据数字识别的任务，设计网络的输入和输出；
3. 实现 BP 网络错误反传算法，完成神经网络的训练测试，最终识别率达到 70% 以上；
4. 数字识别训练集可以自己手工制作，也可以网上下载，要求具有可视化图形界面，能够输入输出。
5. 进一步的，用卷积神经网络实现以上任务，对比深度学习与浅层模型。

二、 实验过程-BP 神经网络

本次 BP 神经网络的实现借鉴了斯坦福大学 CS231n 课程作业的代码框架，核心代码都是由我们小组进行实现。

2.1 前向传播与反向传播

2.1.1 前向传播

仿射变换的前向传播如下，输入为大小 $(N, d_1, \cdots, d_k)$ 的向量，其中 N 为批量大小；仿射变换的矩阵大小为 (D, M)，其中 $D = \prod_{i=1}^{k} d_i$；偏置的矩阵大小为 $(M,)$；通过 numpy 库的 reshape 接口，经过简单的仿射变换即可将多维输入线性映射到 (N, M) 的大小。利用 cache 来存储用于后向传播的缓存。

```
def affine_forward(x, w, b):
    newx = np.reshape(x, (x.shape[0], -1))
    out = np.dot(newx, w) + b
    cache = (x, w, b)
    return out, cache
```

对于 Relu 函数的前向传播同理：

```
def relu_forward(x):
    out = np.maximum(x, 0)
    cache = x
    return out, cache
```

2.1.2 后向传播

根据链式法则分别计算对于 x、w、b 的偏导返回即可：

```
def affine_backward(dout, cache):
    batch_size = x.shape[0]
    newx = np.reshape(x, (x.shape[0], -1))
    dZ = dout
    dw = np.dot(newx.T, dZ)
    db = np.sum(dZ, axis=0)
    dx = np.dot(dZ, w.T).reshape(x.shape)
    return dx, dw, db
```

对于 Relu 函数的反向传播，只需要反向传递双前向输出大于 0 位置的导数即可：

```
def relu_backward(dout, cache):
    # 激活函数本身对x的偏导数dax
    dax = np.zeros(x.shape)
    # 根据z值计算daz
    dax[x > 0] = 1
    dx = np.multiply(dax, dout)
    return dx
```

2.2 优化器

本次实验中实现了 SGD（Stochastic Gradient Descent，随机梯度下降法）和 Adam（Adaptive Moment Estimation，自适应动量估计）优化器，并进行了相关对比实验：

2.2.1 随机梯度下降

按照梯度下降法，固定学习率进行更新：

```
def sgd(w, dw, config=None):
    if config is None:
        config = {}
    config.setdefault("learning_rate", 1e-2)
    w -= config["learning_rate"] * dw
    return w, config
```

2.2.2 Adam 优化器

自适应动量估计（Adaptive Moment Estimation，Adam）算法 [1] 可以看作是动量法和 RMSprop 算法 [2] 的结合，不但使用动量作为参数更新方向，而且可以自适应调整学习率。

Adam 算法一方面计算梯度平方 g_t^2 的指数加权平均（和 RMSprop 算法类似），另一方面计算梯度 g_t 的指数加权平均（和动量法类似）。

$$M_t = \beta_1 M_{t-1} + (1-\beta_1)\, g_t$$
$$G_t = \beta_2 G_{t-1} + (1-\beta_2)\, g_t \odot g_t$$

其中 β_1 和 β_2 分别为两个移动平均的衰减率，通常取值为 $\beta_1 = 0.9, \beta_2 = 0.99$. M_t 可以看作是梯度的均值 (一阶矩), G_t 可以看作是梯度的未减去均值的方差 (二阶矩)。假设 $M_0 = 0, G_0 = 0$，那么在迭代初期 M_t 和 G_t 的值会比真实的均值和方差要小。特别是当 β_1 和 β_2 都接近于 1 时，偏差会很大。因此, 需要对偏差进行修正。

$$\hat{M}_t = \frac{M_t}{1-\beta_1^t}$$

$$\hat{G}_t = \frac{G_t}{1-\beta_2^t}$$

Adam 算法的参数更新差值为

$$\Delta\theta_t = -\frac{\alpha}{\sqrt{\hat{G}_t + \epsilon}}\hat{M}_t$$

其中学习率 α 通常设为 0.001，相关代码如下：

```
def adam(w, dw, config=None):
    config['t'] += 1
    config['m'] = config['beta1'] * config['m'] + (1 - config['beta1']) * dw
    config['v'] = config['beta2'] * config['v'] + (1 - config['beta2']) * (dw**2)
    mb = config['m'] / (1 - config['beta1']**config['t'])
    vb = config['v'] / (1 - config['beta2']**config['t'])
    next_w = w - config['learning_rate'] * mb / (np.sqrt(vb) + config['epsilon'])
    return next_w, config
```

2.3 批量归一化

批量归一化（Batch Normalization，BN）方法 [3] 是一种有效的逐层归一化方法，可以对神经网络中任意的中间层进行归一化操作。其核心思想时将每层的输出进行标准归一化后，再通过一个附加的缩放和平移变换改变取值区间

$$\hat{z}^{(l)} = \frac{z^{(l)} - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \odot \gamma + \beta$$
$$\triangleq \mathrm{BN}_{\gamma,\beta}\left(\boldsymbol{z}^{(l)}\right)$$

其中 γ 和 β 分别代表缩放和平移的参数向量。从最保守的角度考虑，可以通过标准归一化的逆变换来使得归一化后的变量可以被还原为原来的值，当 $\gamma = \sqrt{\sigma_{\mathcal{B}}^2}$，$\beta = \mu_{\mathcal{B}}$ 时，$\hat{z}^{(l)} = \boldsymbol{z}^{(l)}$ 被还原为原来的值。

2.3.1 前向传播

参考上述公式和论文 [3] 中的介绍即可进行前向传播的编写，使用滑动平均计算 $\mu_{\mathcal{B}}$ 和 $\sigma_{\mathcal{B}}^2$，关键代码如下：

```
    # Step 1 - shape of mu (D,)
    mu = 1 / float(N) * np.sum(x, axis=0)
    # Step 2 - shape of var (N,D)
    xmu = x - mu
    # Step 3 - shape of carre (N,D)
    carre = xmu**2
    # Step 4 - shape of var (D,)
    var = 1 / float(N) * np.sum(carre, axis=0)
    # Step 5 - Shape sqrtvar (D,)
    sqrtvar = np.sqrt(var + eps)
    # Step 6 - Shape invvar (D,)
    invvar = 1. / sqrtvar
    # Step 7 - Shape va2 (N,D)
    # (x - m)2/ sigma
    va2 = xmu * invvar
    # Step 8 - Shape va3 (N,D)
    va3 = gamma * va2
    # Step 9 - Shape out (N,D)
    out = va3 + beta
    # 滑动平均
    running_mean = momentum * running_mean + (1.0 - momentum) * mu
    running_var = momentum * running_var + (1.0 - momentum) * var
```

2.3.2 反向传播

使用链式法则求导，对前向传播进行逆向传播，可以编写反向传播关键代码如下：

```
    # Backprop Step 9
    dva3 = dout
    dbeta = np.sum(dout, axis=0)
    # Backprop step 8
```

```
    dva2 = gamma * dva3
    dgamma = np.sum(va2 * dva3, axis=0)
    # Backprop step 7
    dxmu = invvar * dva2
    dinvvar = np.sum(xmu * dva2, axis=0)
    # Backprop step 6
    dsqrtvar = -1. / (sqrtvar**2) * dinvvar
    # Backprop step 5
    dvar = 0.5 * (var + eps)**(-0.5) * dsqrtvar
    # Backprop step 4
    dcarre = 1 / float(N) * np.ones((carre.shape)) * dvar
    # Backprop step 3
    dxmu += 2 * xmu * dcarre
    # Backprop step 2
    dx = dxmu
    dmu = - np.sum(dxmu, axis=0)
    # Basckprop step 1
    dx += 1 / float(N) * np.ones((dxmu.shape)) * dmu
```

2.4 Dropout 层

当训练一个深度神经网络时，我们可以随机丢弃一部分神经元（同时丢弃其对应的连接边）来避免过拟合, 这种方法称为丢弃法（Dropout Method ）[4]。每次选择丢弃的神经元是随机的.。最简单的方法是设置一个固定的概率 p，对每一个神经元都以概率 p 来判定要不要保留。对于一个神经层 $\boldsymbol{y} = f(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$，我们可以引入一个掩蔽函数 $\text{mask}(\cdot)$ 使得 $\boldsymbol{y} = f(\boldsymbol{W}\,\text{mask}(\boldsymbol{x}) + \boldsymbol{b})$. 掩蔽函数 $\text{mask}(\cdot)$ 的定义为

$$\text{mask}(\boldsymbol{x}) = \begin{cases} \boldsymbol{m} \odot \boldsymbol{x} & \text{当训练阶段时} \\ p\boldsymbol{x} & \text{当测试阶段时} \end{cases}$$

其中 $\boldsymbol{m} \in \{0,1\}^D$ 是丢弃掩码 (Dropout Mask) ，通过以概率为 p 的伯努利分布随机生成. 在训练时，激活神经元的平均数量为原来的 p 倍。而在测试时，所有的神经元都是可以激活的。这会造成训练和测试时网络的输出不一致。为了缓解这个问题，在测试时需要将神经层的输入 $\boldsymbol{x}$ 乘以 p，也相当于把不同的神经网络做了平均。保留率 p 可以通过验证集来选取一个最优的值。一般来讲，对于隐藏层的神经元，其保留率 $p = 0.5$ 时效果最好，这对大部分的网络和任务都比较有效。当 $p = 0.5$ 时，在训练时有一半的神经元被丢弃，只剩余一半的神经元是可以激活的，随机生成的网络结构最具多样性。对于输入层的神经元，其保留率通常设为更接近 1 的数，使得输入变化不会太大。对输入层神经元进行丢弃时，相当于给数据增加噪声，以此来提高网络的鲁棒性。

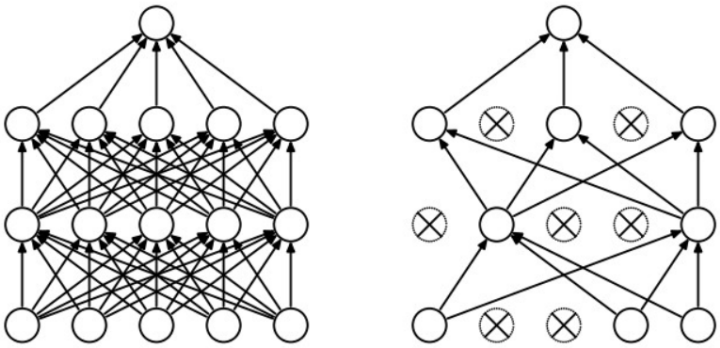

图 1　**dropout** 层

2.4.1 前向传播

根据上述定义便可以 Dropout 的前向传播代码进行编写，关键代码如下：

```
if mode == "train":
    mask = (np.random.rand(*x.shape) >= p) / (1-p)
    out = x * mask
elif mode == "test":
    out = x
cache = (dropout_param, mask)
out = out.astype(x.dtype, copy=False)
```

2.4.2 反向传播

后向传播时，训练阶段仅需要传播未 mask 的部分，关键代码如下：

```
if mode == "train":
    dx = dout * mask
elif mode == "test":
    dx = dout
```

三、 结果分析-BP 神经网络

采用三层神经网络，结构为 28 * 28 - 100 - 10 进行下述实验探究。

3.1 优化器的探究

采用早停法，设置在 MNIST 数据集迭代 3 次还没有验证集准确率的增加则停止，经过调试，对于 SGD 优化器，采用学习率 0.02 进行训练的效果较优；对于 Adam 优化

器，采用学习率 0.001 训练效果较优，实验结果表格如下：

表 **1**　实验结果表格

Optimizer	**SGD**	**Adam**
val acc	97.12	**97.57**
Epoch	36	**14**

采用 SGD 优化器的训练损失下降图、验证准确率及训练准确率图如下：

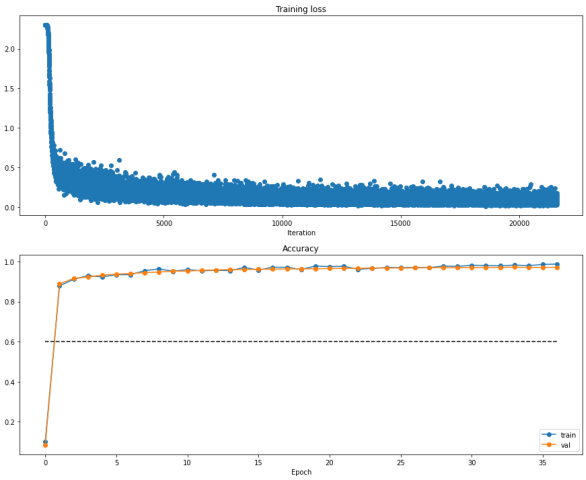

图 **2**　实验结果

采用 Adam 优化器的训练损失下降图、验证准确率及训练准确率图如下：

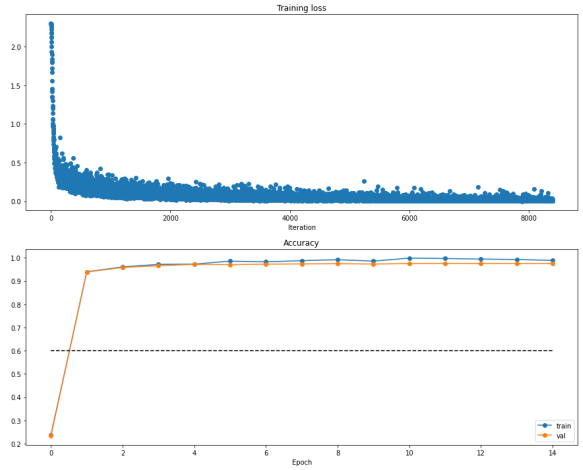

图 **3**　实验结果

从实验结果可得，使用 Adam 优化器相比 SGD 优化器对于 BP 网络的收敛速度和性能都有了很大的提升。

3.2 对比批量归一化和丢弃法

向 BP 网络中加入批量归一化和丢弃法，观察二者对于神经网络的影响，丢弃的概率选择 0.5，结果如下：

表 **2**　实验结果

Model	**BPNN**	**BPNN+BN**	**BPNN+DP**	**BPNN+BN+DP**
val acc	97.62	**97.91**	97.65	97.52
Epoch	14	14	**12**	23

结果显示，批量归一化和丢弃法单独使用时，都能够在准确率、迭代次数等方面使性能有所提升。然而，当二者同时使用时，BP 网络的性能在准确率、迭代次数等方面都有所下降。

四、 结论-BP 神经网络

在本次实验中，我们小组完成了三层 BP 网络的错误反传算法，实现了百分之 70 以上的准确率。同时，我们就优化器、批量归一化、丢弃法等 BP 神经网络常见优化方法进行了探究，发现如下结论：

- 从实验结果可得，使用 Adam 优化器相比 SGD 优化器对于 BP 网络的收敛速度和性能都有了很大的提升；
- 批量归一化和丢弃法单独使用时，都能够在准确率、迭代次数等方面使性能有所提升；
- 当批量归一化和丢弃法同时使用时，BP 网络的性能在准确率、迭代次数等方面都有所下降。

其中第三点出现的原因我们推测为批量归一化在某种程度上防止过拟合的功能 [3] 和丢弃法有所冲突，从而导致了 BP 网络的性能在准确率、迭代次数等方面性能的下降。

参考文献

[1] Kingma D, Ba J. Adam: A method for stochastic optimization[C]//Proceedings of International Conference on Learning Representations. 2015.

[2] Tieleman T, Hinton G. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude[Z]. 2012.

[3] Ioffe S, Szegedy C. Batch normalization: Accelerating deep network training by reducing internal covariate shift[C]//Proceedings of the 32nd International Conference on Machine Learning. 2015: 448-456.

[4] Srivastava N, Hinton G, Krizhevsky A, et al. Dropout: A simple way to prevent neural networks from overfitting[J]. The Journal of Machine Learning Research, 2014, 15(1):1929-1958.