专业课程实验报告

课程名称： 模式识别

开课学期： 2022 至 2023 学年 第 1 学期

专　　业： 智能科学与技术

年级班级： 20 级 3 班

学生姓名： 严中圣

学生学号： 222020335220177

实验教师： 杨颂华

人工智能学院

《模式识别》实验课报告书

实验编号：	2	**实验名称:**	基于 BP 神经网络的图像分类
姓　　名：	严中圣	**学　　号：**	222020335220177
日　　期：	2022 年 11 月 24 日	**教师打分：**	

1 实验目标

多层感知器 (Multi-Layer Perceptron,MLP) 也叫人工神经网络 (Artificial Neural Network,ANN), 除了输入输出层，它中间可以有多个隐层。最简单的 MLP 需要有一层隐层，即输入层、隐层和输出层才能称为一个简单的神经网络。通俗而言，神经网络是仿生物神经网络而来的一种技术，通过连接多个特征值，经过线性和非线性的组合，最终完成目标识别。通过本实验旨在掌握多层感知器神经网络的原理和实现算法，将其应用到目标识别任务。

2 实验环境

- PyCharm 2022.1.3 (Professional Edition)
- python 3.7.13, numpy 1.21.5, torchvision 0.11.3
- OS: Windows 11 22H2, CPU:12th Gen Intel(R) Core(TM) i7-12700H 2.30 GHz

3 实验原理

3.1 神经元模型

神经网络 (Neural Network) 是由具有适应性的简单单元组成的广泛并行互连的网络，它的组织能够模拟生物神经系统对真实世界物体所作出的交互反应。神经网络中最基本的成分是神经元模型，即上述定义中的”简单单元”在生物神经网络中每个神经元与其他神经元相连，当它”兴奋”时，就会向相连的神经元发送化学物质，从而改变这些神经元内的电位; 如果某神经元的电位超过了一个”阈值” (threshold) 那么它就会被激活, 即”兴奋”起来，向其他神经元发送化学物质。具体结构如图1所示。

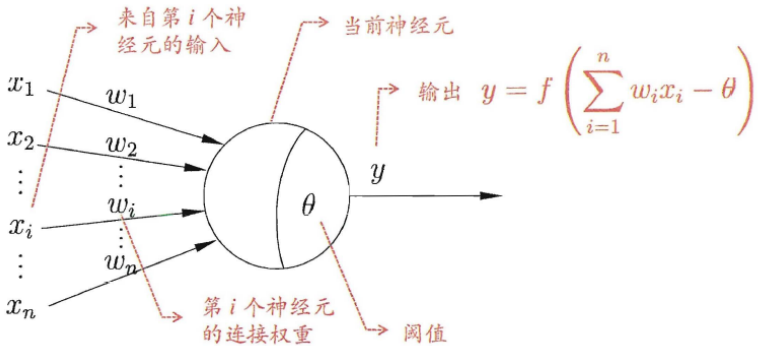

图 1: M-P 神经元模型

3.2 误差逆传播算法

如图2所示，网络结构设计为四层网络层，包括一个输入层，一个输出层和两个隐藏层。数据输入后，通过误差逆传播 (BP) 算法对其中的权值及偏置进行不断更新。各层输入输出表达式见式 (1)。

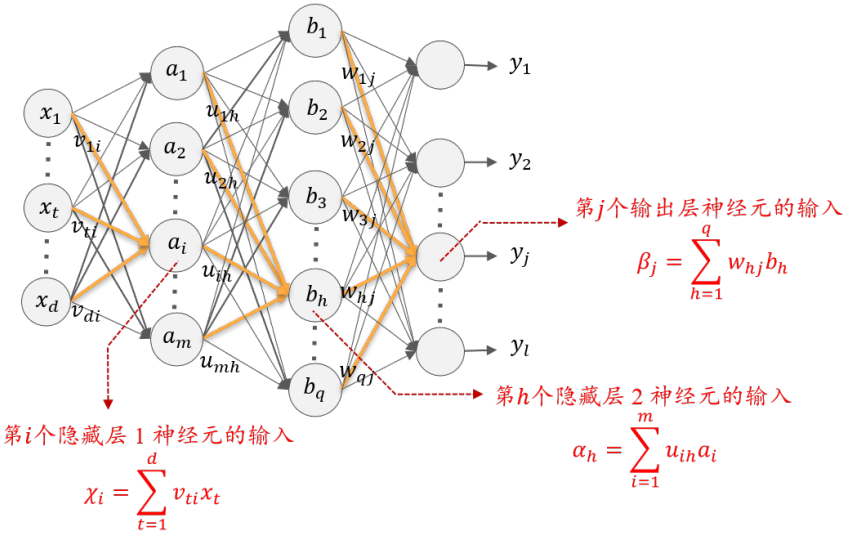

图 2: M-P 神经元模型

$$
\begin{aligned}
a_i &= f\left(\chi_i - e_i\right) \\
b_h &= f\left(\alpha_h - \gamma_h\right) \\
y_j &= f\left(\beta_j - \theta_j\right)
\end{aligned} \tag{1}
$$

对于我们所设计的如图2所示的网络结构，我们设置隐层神经元为 ReLU 函数，输出层采用 Sigmoid 函数

$$
\begin{aligned}
f_1 = \mathrm{ReLU}(x) &= \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases} \\
f_2 = \mathrm{Sigmoid}(x) &= \frac{1}{1+e^{-x}}
\end{aligned} \tag{2}
$$

此外设置损失函数为

$$
E_k = \frac{1}{2} \sum_{j=1}^{l} \left(y_j^k \hat{-} y_j^k\right)^2 \tag{3}
$$

BP 是一个迭代学习算法，在迭代的每一轮中采用广义的感知机学习规则对参数进行更新估计，则任意参数 v 的更新估计式为

$$
v \leftarrow v + \Delta v \tag{4}
$$

于是基于梯度下降策略，以目标的负梯度方向对参数进行调整，对式 (3) 的误差 E_k，给定学习率 η，有：

$$
E_k = \frac{1}{2} \sum_{j=1}^{l} \left(y_j^k \hat{-} y_j^k\right)^2 \tag{5}
$$

根据链式法则：

$$
\frac{\partial E_k}{\partial w_{hj}} = \frac{\partial E_k}{\partial \hat{y}_j^k} \frac{\partial \hat{y}_j^k}{\partial \beta_j} \frac{\partial \beta_j}{\partial w_{hj}} \tag{6}
$$

根据 β_j 的定义，显然有

$$\frac{\partial \beta_j}{\partial w_{hj}} = b_h \tag{7}$$

此外 Sigmoid 函数有一个很好的性质

$$f⊠(x) = f(x)(1-f(x)) \tag{8}$$

于是设

$$g_j = -\frac{\partial E_k}{\partial \hat{y}_j^k}\frac{\partial \hat{y}_j^k}{\partial \beta_j} = \hat{y}_j^k\left(1-\hat{y}_j^k\right)\left(y_j^k-\hat{y}_j^k\right) \tag{9}$$

所以 w_{hj} 的更新公式为

$$w_{hj} = \eta g_j b_h \tag{10}$$

同理可得：输出层偏置 θ_j 的更新公式为：

$$\theta_j = -g_j \tag{11}$$

故隐藏层 2 的权重及偏置更新公式为：

$$\begin{aligned}
\Delta u_{ih} &= -\eta\frac{\partial E_k}{\partial u_{ih}} \\
&= -\eta\sum_{j=1}^{l} -g_j w_{hj} b_h\left(1-b_h\right)a_i \\
&= \sum_{j=1}^{l}\eta g_j w_{hj} b_h\left(1-b_h\right)a_i \\
\Delta\gamma_h &= -\eta\frac{\partial E_k}{\partial \gamma_h} \\
&= -\sum_{j=1}^{l}\eta g_j w_{hj} b_h\left(1-b_h\right)
\end{aligned} \tag{12}$$

隐藏层 1 的权重及偏置更新公式为：

$$\begin{aligned}
\Delta v_{ti} &= -\eta\frac{\partial E_k}{\partial v_{ti}} \\
&= -\sum_{j=1}^{l}\sum_{h=1}^{q}\eta g_j w_{hj} b_h\left(1-b_h\right)u_{ih}a_i\left(1-a_i\right)x_t \\
\Delta e_i &= -\eta\frac{\partial E_k}{\partial e_i} \\
&= \sum_{j=1}^{l}\sum_{h=1}^{q}\eta g_j w_{hj} b_h\left(1-b_h\right)u_{ih}a_i\left(1-a_i\right)
\end{aligned} \tag{13}$$

4 实验步骤

4.1 数据集来源

本次实验采用来自 FashionMNIST 是一个替代 MNIST 手写数字集的图像数据集。它是由 Zalando（一家德国的时尚科技公司）旗下的研究部门提供，是经典 MNIST 数据集的简易替换，两者图像格式及大小都相同。Fashion MNIST 比常规 MNIST 手写数据将更具挑战性。Fashion MNIST 服饰数据集包含 70000 张灰度图像，其中包含 60,000 个示例的训练集和 10,000 个示例的测试集，每个示例都是一个 28x28 灰度图像，共具有十个类别。

4.2 数据加载

利用 torchvision 中所集成的 Fashion_MNIST 数据集加载数据，此时返回的数据为 PIL Image 格式，在后续处理中，直接利用 $numpy.array()$ 将其转化为数组处理即可。

```
def fashion_mnist_dataset(download=False):
    """
    dataset preparation
    :param download: whether to download to lacally or not
    :return: Fashion MNIST dataset (PIL Image)
    """
    train_dataset = dataset.FashionMNIST(root="fashion_mnist", train=True
    , download=download)
    test_dataset = dataset.FashionMNIST(root="fashion_mnist", train=False
    , download=download)
    return train_dataset, test_dataset
```

4.3 BP 网络训练

在网络的设计上，将层进行抽象作为独立设计单元，旨在更便于网络搭建。前馈传播即按照式 (1) 进行编写程序，反馈权重更新即根据式 (12)(13) 进行设计。训练中为了减小内存占用，采用分批次训练，同时每一轮训练都对训练集进行随机打乱以提高在测试集上的泛化能力。同时网络支持多种激活函数，且进行了封装方便调用。最终在 MNIST 数据集上得到了 87.12% 的准确率。

5 实验分析

本次 BP 实验过程将反向传播的过程进行了纯手写的搭建，对网络传播的过程有了更深刻和清晰的认识。同时在代码的编写上还仍有不足，没有封装地更为高效，且梯度的求解可以优化为自动求解，面向对象的设计思维仍有待提升。后续会进一步参考主流深度学习框架的设计思路，将网络设计地调用更方便，封装更完整，实现真正意义上的面向对象设计。

6 参考文献

[1] 周志华. 机器学习 [M]. 清华大学出版社, 2016.

[2] 李航. 统计学习方法 [M]. 清华大学出版社, 2019

[3] https://zhuanlan.zhihu.com/p/31886934

[4] Christopher M. Bishop: Pattern Recognition and Machine Learning, Chapter 4.3.4

7 附录 – BP 实验完整代码

```
import numpy as np
import matplotlib.pyplot as plt
import torchvision.datasets as dataset
from tqdm import tqdm
```

```
def fashion_mnist_dataset(download=False):
    """
    dataset preparation
    :param download: whether to download to lacally or not
    :return: Fashion MNIST dataset (PIL Image)
    """
    train_dataset = dataset.FashionMNIST(root="fashion_mnist", train=True
   , download=download)
    test_dataset = dataset.FashionMNIST(root="fashion_mnist", train=False
   , download=download)
    return train_dataset, test_dataset


class Layer:
    def __init__(self, n_input, n_output, activation=None, weights=None,
   bias=None):
        self.activation = activation
        self.weights = weights if weights is not None else np.random.
   randn(n_input, n_output) * np.sqrt(1 / n_output)
        self.bias = bias if bias is not None else np.random.rand(n_output
   ) * 0.1
        self.activation_output = None

    def forward(self, x_input):
        r = np.dot(x_input, self.weights) - self.bias
        self.activation_output = self.apply_activation(r)
        return self.activation_output

    def apply_activation(self, r):
        if self.activation is None:
            return r
        elif self.activation == 'relu':
            return np.maximum(r, 0)
        elif self.activation == 'sigmoid':
            x_ravel = r.ravel()
            length = len(x_ravel)
            y = []
            for index in range(length):
                if x_ravel[index] >= 0:
                    y.append(1.0 / (1 + np.exp(-x_ravel[index])))
                else:
                    y.append(np.exp(x_ravel[index]) / (np.exp(x_ravel[
   index]) + 1))
            return np.array(y).reshape(r.shape)

    def apply_activation_derivative(self, r):
```

```
        if self.activation is None:
            return np.ones_like(r)
        elif self.activation == 'relu':
            grad = np.array(r, copy=True)
            grad[r > 0] = 1.
            grad[r <= 0] = 0.
            return grad
        elif self.activation == 'sigmoid':
            return r * (1 - r)
        return r


class Network:
    def __init__(self, train_dataset=None, test_dataset=None, train_bs
   =16, test_bs=1, init_lr=0.0002):
        self.train_batch_size = train_bs
        self.test_batch_size = test_bs
        self.learning_rate = init_lr
        self.layers = []
        self.classNum = len(train_dataset.classes)
        self.labels = [i for i in range(self.classNum)]
        self.train_dataset = train_dataset
        self.test_dataset = test_dataset
        self.trainData = []
        self.trainLabel = []
        self.testData = []
        self.testLabel = []

        # for k in range(len(self.train_dataset)):
        for k in range(500):
            image, label = train_dataset[k]
            image = np.array(image).flatten() / 255
            self.trainData.append(image)
            self.trainLabel.append(label)

        # for k in range(len(self.test_dataset)):
        for k in range(100):
            image, label = test_dataset[k]
            image = np.array(image).flatten() / 255
            self.testData.append(image)
            self.testLabel.append(label)

    def add_layer(self, layer):
        self.layers.append(layer)

    def forward(self, x_input):
        for layer in self.layers:
```

```
            x_input = layer.forward(x_input)
        return x_input

    def backward(self, x_input, y, learning_rate):
        """
        back propagation
        :param x_input:
        :param y:
        :param learning_rate:
        :return:
        """
        output = self.forward(x_input)
        g = output * (1 - output) * (y - output)  # g.size=[n_output,1]
        for i in reversed(range(len(self.layers))):
            layer = self.layers[i]
            if layer == self.layers[-1]:  # output layer
                last_layer = self.layers[i - 1]
                # print(len(last_layer.activation_output))
                delta_weight = [[] for k in range(len(last_layer.
activation_output))]
                for h in range(len(last_layer.activation_output)):
                    for j in range(len(layer.activation_output)):
                        delta_weight[h].append(learning_rate * g[j] *
last_layer.activation_output[h])
                delta_weight = np.array(delta_weight)
                # delta_weight=learning_rate*g*last_layer.
activation_output
                layer.weights = layer.weights + delta_weight
                delta_bias = -learning_rate * g
                layer.bias = layer.bias + delta_bias
            else:
                next_layer = self.layers[i + 1]
                if i + 1 == len(self.layers) - 1:  # hidder layer 2
                    last_layer = self.layers[i - 1]
                    delta_weight = [[] for k in range(len(last_layer.
activation_output))]
                    for k in range(len(last_layer.activation_output)):
                        for h in range(len(layer.activation_output)):
                            sum = 0
                            for j in range(len(next_layer.
activation_output)):
                                sum += next_layer.weights[h][j] * g[j] *
layer.activation_output[h] * (
                                        1 - layer.activation_output[h]) *
 last_layer.activation_output[i]
                            delta_weight[k].append(sum)
                    delta_weight = np.array(delta_weight)
```

```
                    layer.weights += delta_weight * learning_rate
                    delta_bias = []
                    for h in range(len(layer.activation_output)):
                        sum = 0
                        for j in range(len(next_layer.activation_output))
:
                            sum += next_layer.weights[h][j] * g[j] *
layer.activation_output[h] * (
                                    1 - layer.activation_output[h]) *
(-1)
                        delta_bias.append(sum)
                    delta_bias = np.array(delta_bias)
                    layer.bias += delta_bias
                else:  # hidden layer 1
                    output_layer = self.layers[-1]
                    delta_weight = [[] for i in range(len(x_input))]
                    for t in range(len(x_input)):
                        for k in range(len(layer.activation_output)):
                            sum = 0
                            for h in range(len(next_layer.
activation_output)):
                                for j in range(len(y)):
                                    sum += output_layer.weights[h][j] * g
[j] * next_layer.activation_output[h] * (
                                            1 - next_layer.
activation_output[h]) * (next_layer.weights[k][h]) * \
                                        layer.activation_output[k] *
(1 - layer.activation_output[k]) * x_input[t]
                            delta_weight[t].append(sum)
                    delta_weight = np.array(delta_weight) * learning_rate
                    layer.weights += delta_weight
                    delta_bias = []
                    for k in range(len(layer.activation_output)):
                        sum = 0
                        for h in range(len(next_layer.activation_output))
:
                            for j in range(len(y)):
                                sum += output_layer.weights[h][j] * g[j]
* next_layer.activation_output[h] * (
                                        1 - next_layer.activation_output[
h]) * (next_layer.weights[k][h]) * \
                                    layer.activation_output[k] * (1 -
layer.activation_output[k]) * (-1)
                        delta_bias.append(sum)
                    delta_bias = np.array(delta_bias) * learning_rate
                    layer.bias = layer.bias + delta_bias
```

```
    def train(self, learning_rate=0.0002, max_epochs=200):
        y_onehot = np.zeros((len(self.trainData), 10))  # one-hot
    encoding
        y_onehot[np.arange(len(self.trainData)), np.array(self.trainLabel
    ).flatten()] = 1

        for epoch in tqdm(range(max_epochs)):
            count = 0
            for i in tqdm(range(1, len(self.trainData) // self.
    train_batch_size + 1)):
                for k in range(count, count + self.train_batch_size):
                    self.backward(np.array(self.trainData[k]), y_onehot[k
    ], learning_rate)
                count += self.train_batch_size
                # print("batch {}".format(i))
                i += 1
            test_count = 0
            for i in range(len(self.testData)):
                y_label = np.argmax(self.forward(self.testData[i]))
                if y_label == self.testLabel[i]:
                    test_count += 1

            print('Epoch: #%s, Accuracy: %.2f' % (epoch + 1, test_count /
    len(self.testData)))

    def predict(self, x):
        y_predict = self.forward(x)
        y_predict = np.argmax(y_predict, axis=1)
        return y_predict


if __name__ == '__main__':
    train_dataset, test_dataset = fashion_mnist_dataset(download=False)

    network = Network(train_dataset=train_dataset, test_dataset=
    test_dataset, train_bs=2, test_bs=1, init_lr=0.0002)
    network.add_layer(Layer(28 * 28, 15, 'relu'))
    network.add_layer(Layer(15, 10, 'relu'))
    network.add_layer(Layer(10, 10, 'sigmoid'))
    network.train(max_epochs=10)
```