西安交通大学

XI'AN JIAOTONG UNIVERSITY

机器学习

实验报告

2023-2024 学年　第二学期

学　　院：　数学与统计学院

班　　级：　数学强基 2102

学　　号：　22113210010

姓　　名：　周冠程

指导教师：　孟德宇老师

实验地点：　数学实验中心

1 实验内容

1.1 问题描述

考虑一个简单的 XOR 问题，包括两个输入特征和一个输出标签。你的任务是使用反向传播算法实现一个具有单隐藏层的神经网络来解决这个二分类问题。

1.2 实验步骤

1. 定义神经网络结构：包括输入层、隐藏层和输出层的节点数;
2. 初始化神经网络的参数：包括权重和偏置;
3. 实现前向传播算法：计算神经网络的输出;
4. 实现反向传播算法：计算梯度并更新参数;
5. 使用 XOR 数据集进行训练和测试。

1.3 实验要求

1. 输入特征：X = [[0, 0], [0, 1], [1, 0], [1, 1]];
2. 输出标签：Y = [0, 1, 1, 0] （对应 XOR 逻辑运算结果）;
3. 使用 Sigmoid 激活函数;
4. 使用 Mean Squared Error（均方误差）作为损失函数;
5. 设置合适的学习率和训练轮数。

1.4 提示

可以根据需要自行定义神经网络的结构和超参数。在实现时，可以分别编写函数来处理前向传播、反向传播和参数更新部分。

2 实验描述

2.1 算法介绍

BP 算法是一种常见的神经网络训练算法,也称为反向传播算法(Back-propagation)。它是一种基于梯度下降优化的算法，用于调整神经网络中的参数，以最小化网络输出与实际值之间的误差。BP 算法的基本思想是通过反向传播误差来更新神经网络中的权重，使得网络的输出更接近于期望的输出。

通常情况下，算法的核心步骤包括:

1. **前向传播 (Forward Propagation):** 将输入数据传递到网络中，通过网络的每一层进行计算，直到得到输出值。在这个过程中，网络会根据当前的权重和偏置值计算每个神经元的输出;

2. **计算误差 (Compute Error):** 将网络的输出与实际值进行比较，计算网络的误差。常用的误差函数包括均方误差（Mean Squared Error，MSE）和交叉熵误差（Cross Entropy Error）等;

3. **反向传播 (Backward Propagation):** 从输出层开始，将误差沿着网络反向传播回每一层，并计算每一层的误差贡献。这个过程中利用链式法则计算每个参数对误差的贡献，然后根据这些贡献调整网络中的权重和偏置值;

4. **更新参数 (Update Parameters):** 根据反向传播计算得到的梯度信息，利用梯度下降等优化算法来更新网络中的参数，使得网络的误差逐渐减小;

5. **重复训练 (Repeat Training):** 重复执行前向传播、误差计算、反向传播和参数更新的过程，直到网络的误差收敛到一个满意的水平，或达到预先设定的训练次数。

本次实验中采用经典的多层感知机（Multilayer Perceptron，MLP）结构作为模型进行试验，多层感知机由输入层、若干个隐藏层和输出层，其中隐藏层由上一层的结果进行线性运算再经过激活函数得到，线性运算通常被定义为线性全连接层（Linear Layer），激活函数通常是一个非线性函数用于向模型中添加非线性因素，提高模型的拟合能力。

2.2 理论推导

2.2.1 符号定义

在进行理论推导前，本部分将简要介绍一下推导中可能使用到的符号及定理。

[定义] 2.2.1 *(高维欧式空间向量)* $X = (X_1, X_2, \cdots, X_n)^T$ 表示一个 $\mathbb{R}^n$ 的 n 维向量，其中 X_i 表示 X 的第 i 个分量。

[定义] 2.2.2 *(矩阵表示)* $M = \begin{bmatrix} M_{1,1} & M_{1,2}\cdots & M_{1,n} \\ M_{2,1} & M_{2,2}\cdots & M_{2,n} \\ \vdots & \vdots\ddots & \vdots \\ M_{m,1} & M_{m,2}\cdots & M_{m,n} \end{bmatrix}$ 表示一个 $\mathbb{R}^{m*n}$ 的 $m*n$ 维矩阵，其中 M_{ij} 表示 M 的坐标为 *(i,j)* 的分量。

[定义] 2.2.3 *(爱因斯坦和约定)* 略去求和式中的求和号。在此规则中两个相同指标就表示求和。

例如：$X \in \mathbb{R}^{mn}, Y \in \mathbb{R}^{mk}, Z \in \mathbb{R}^{kn}$

$$X_{ij} = \sum_{t=1}^{k} Y_{it} Z_{tj} \tag{1}$$

可以表示为：

$$X_{ij} = Y_{it} Z_{tj} \tag{2}$$

[定义] 2.2.4 *(高维向量值函数求导)* $f : \mathbb{R}^n \to \mathbb{R}^m, X \mapsto f(X)$ 足够光滑，定义其导数为一个 $m*n$ 维的矩阵函数，表示为：

$$\frac{df}{dX} = \begin{bmatrix} \frac{df_1}{dX_1} & \frac{df_1}{dX_2}\cdots & \frac{df_1}{dX_n} \\ \frac{df_2}{dX_1} & \frac{df_2}{dX_2}\cdots & \frac{df_2}{dX_n} \\ \vdots & \vdots\ddots & \vdots \\ \frac{df_m}{dX_1} & \frac{df_m}{dX_2}\cdots & \frac{df_m}{dX_n} \end{bmatrix} \tag{3}$$

[命题] 2.2.1 *(高维向量值函数导数链式法则)* 对于 $f:\mathbb{R}^n\to\mathbb{R}^m, X\mapsto Y$ 和 $g:\mathbb{R}^m\to\mathbb{R}^k, Y\mapsto Z$，有

$$\frac{d(g\circ f)_i}{dX_j}=\frac{dg_i}{dY_k}\frac{df_k}{dX_j} \tag{4}$$

或直接通过矩阵乘法格式表示为：

$$\frac{d(g\circ f)}{dX}=\frac{dg}{dY}\frac{df}{dX} \tag{5}$$

2.2.2 推导

本部分将通过一个算例的推导来引出之后模块化设计的推导，定义 MLP 中每层线性全连接层表示为：

$$X_i^{(k)}=W_{ik}^{(k)}X_k^{(k-1)}+b^{(k)} \tag{6}$$

其中，$X_i^{(k)}$ 表示第 k 层的向量（隐向量、输入向量或输出向量）,$W^{(k)}, b^{(k)}$ 表示第 k-1 层到第 k 层的权重和偏差。激活函数采用 Sigmoid 函数：

$$s(x)=\frac{1}{1+e^{-x}} \tag{7}$$

特别地，该函数具有性质：

$$\frac{ds(x)}{dx}=s(x)(1-s(x)) \tag{8}$$

损失函数采用均方差函数:

$$L=\frac{1}{2}(\hat{Y}-Y)^2 \tag{9}$$

定义该 MLP 网络共有 N 层，其中 $X^{(N)}=\hat{Y}$，首先尝试求取 $\frac{L}{X^{(k)}}$，有

$$\frac{dL}{dX_j^{(N)}}=(X_j^{(N)}-Y_j) \tag{10}$$

进一步地，根据链式法则（2.2.1）对于 k<N 有

$$\frac{dL}{dX^{(k)}}=\frac{dL}{dX^{(N)}}\frac{dX^{(N)}}{dX^{(N-1)}}\cdots\frac{dX^{(k+1)}}{dX^{(k)}} \tag{11}$$

若 $\frac{dL}{dX^{(k+1)}}$ 已知则

$$\frac{dL}{dX^{(k)}}=\frac{dL}{dX^{(k+1)}}\frac{dX^{(k+1)}}{dX^{(k)}} \tag{12}$$

这需要计算 $\frac{dX^{(k+1)}}{dX^{(k)}}$。

若 $X^{(k)}$ 到 $X^{(k+1)}$ 是线性运算，即：

$$X^{(k+1)} = W^{(k+1)}X^{(k)} \tag{13}$$

有

$$\frac{dX^{(k+1)}}{dX^{(k)}} = W^{(k+1)} \tag{14}$$

若 $X^{(k)}$ 到 $X^{(k+1)}$ 是 Sigmoid 运算, 即：

$$X_i^{(k+1)} = s(X_i^{(k)}) \tag{15}$$

有：

$$\frac{dX_i^{(k+1)}}{dX_j^{(k)}} = \delta_{ij}\frac{ds(X_j^{(k)})}{dX_j^{(k)}} = \delta_{ij}X_i^{(k+1)}(1 - X_i^{(k+1)}) \tag{16}$$

其中 δ_{ij} 为示性函数，定义为

$$\delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases} \tag{17}$$

基于以上分析，通过将式 (10)(14)(16) 代入式 (11) 或 (12) 即可计算出所有的 $\frac{dL}{dX^{(k)}}$，基于该计算结果可以进一步地计算出 $\frac{dL}{dW^{(k)_{ij}}}$ 和 $\frac{dL}{db_i^{(k)}}$，具体而言:

$$\begin{aligned} &\frac{dL}{dW_{ij}^{(k)}} \\ =&\frac{dL}{dX_s^{(k)}}\frac{dX_s^{(k)}}{dW_{ij}^{(k)}} \\ =&\frac{dL}{dX_s^{(k)}}\frac{d(W_{st}^{(k)}X_t^{(k-1)} + b_s^{(k)})}{dW_{ij}^{(k)}} \\ =&\frac{dL}{dX_s^{(k)}}\delta_{si}X_j^{(k-1)} \\ =&\frac{dL}{dX_i^{(k)}}X_j^{(k-1)} \end{aligned} \tag{18}$$

$$
\begin{aligned}
&\frac{dL}{db_i^{(k)}} \\
=&\frac{dL}{dX_s^{(k)}}\frac{dX_s^{(k)}}{db_i^{(k)}} \\
=&\frac{dL}{dX_s^{(k)}}\frac{d(W_{st}^{(k)}X_t^{(k-1)}+b_s^{(k)})}{db_i^{(k)}} \\
=&\frac{dL}{dX_s^{(k)}}\delta_{si} \\
=&\frac{dL}{dX_i^{(k)}}
\end{aligned}
\tag{19}
$$

基于此模型中所有参数 $W^{(k)}, b^{(k)}$ 关于损失函数的导数均可求得，最后再根据批牛顿梯度下降法优化参数即可，具体而言：

$$
\begin{aligned}
&\frac{dL}{dW_{ij}^{(k)}}^{*} = \mathbb{E}_{X\in Sample}\frac{dL}{dW_{ij}^{(k)}} \\
&\frac{dL}{db_i^{(k)}}^{*} = \mathbb{E}_{X\in Sample}\frac{dL}{db_i^{(k)}} \\
&W_{ij}^{(k)} \leftarrow W_{ij}^{(k)} - \epsilon * \frac{dL}{dW_{ij}^{(k)}}^{*} \\
&b_i^{(k)} \leftarrow b_i^{(k)} - \epsilon * \frac{dL}{db_i^{(k)}}^{*}
\end{aligned}
\tag{20}
$$

其中，Sample 表示某一批的采样数据分布。

2.3 模块化设计

通过上一节的分析，想要得到 $W^{(k)}, b^{(k)}$ 关于损失函数的导数，仅需要得到 $\frac{dL}{dX^{(k)}}$，而 $\frac{dL}{dX^{(k)}}$ 可以通过式 (12) 递推求得。基于这个思想可以模块设计线性算子和 Sigmoid 算子。

基于面向对象编程思想，构建算子对象，输入为 X，输出为 Y，成员函数包括

1. forward: 响应函数，输入 X 返回 Y；
2. train: 更改算子状态为训练模式，将存储 forward 中的 X 和 Y；
3. test: 更改算子状态为推理模式，将不存储 forward 中的 X 和 Y；

4. calc_grad: 在训练模式下，根据存储的信息计算梯度，输入 $\frac{dL}{dY}$，返回 $\frac{dL}{dX}$；

5. back_ward: 在训练模式下，根据计算得到的梯度进行参数更新；

具体而言，对于输入 $X \in \mathbb{R}^{nN}$ 到输出 $Y \in \mathbb{R}^{mN}$（N 表示该批数据数量）线性层：

$$
\begin{aligned}
&Y_{in} = W_{ij}X_{jn} + b_i \\
&\frac{dL}{dX}_{kin} = \frac{dL}{dY}_{kin} W_{ij} \\
&\frac{dL}{dW}_{ijn} = \frac{dL}{dY}_{kin} X_{jn} \\
&\frac{dL}{db} = \frac{dL}{dY} \\
&\frac{dL}{dW}^{*}_{ij} = \frac{1}{N}\frac{dL}{dW}_{ijn} \\
&\frac{dL}{db}^{*}_{i} = \frac{1}{N}\frac{dL}{db}_{in}
\end{aligned}
\tag{21}
$$

激活函数曾没有参数，只需要进行反传梯度即可，类似上方表示，在此不多加赘述，详细结果可以参考代码。

对于整体模型，$\frac{dL}{dX^{(k)}}$ 的计算需要反向递推计算，在全部求得后，再输入学习率进行反传。

2.4 正则化

为了防止模型过拟合训练数据，提高模型的泛化能力。在此引入 L2 正则化损失，表达为：

$$
\lambda \parallel W \parallel_2^2 \tag{22}
$$

其中，$\parallel W \parallel_2$ 为 W 的 L2 范数 [?]，此时仅需要在原损失导数的基础上增加一项 $2\lambda W$ 即可，λ 是一个超参数，需要手动地进行调整。

3 程序框图

首先展示模块化算子的伪代码

Algorithm 1: Forward

Data: $X \in \mathbb{R}^{n*N}$

Result: $Y \in \mathbb{R}^{m*N}$

Caculate Y;

if *Train mode* **then**

    Store X and Y;

return Y

Algorithm 2: Calculate Gradient

Data: $\frac{dL}{dY} \in \mathbb{R}^{1*m*N}$

Result: $\frac{dL}{dX} \in \mathbb{R}^{1*n*N}$

Calculate $\frac{dL}{dX}$ by X and Y;

Calculate $\frac{dL}{dW}$ by $\frac{dL}{dX}$;

Store $\frac{dL}{dW}$;

return $\frac{dL}{dX}$

Algorithm 3: Backward

Data: ϵ:Learning rate; λ:Normalize parameter

Calculate ${\frac{dL}{dW}}^*$ by $\frac{dL}{dW}$;

$W \leftarrow W - \epsilon {\frac{dL}{dW}}^* - 2\lambda W$ **return** $\frac{dL}{dX}$

整体网络有多层构成，存储于 layers 的线性表数据结构中，其算法整体框架为：

Algorithm 4: Forward of Whole Model

Data: $X \in \mathbb{R}^{n*N}$

Result: $Y \in \mathbb{R}^{m*N}$

$Y \leftarrow X$;

for *layer in layers* **do**

    $Y \leftarrow layer.forward(Y)$

return Y

Algorithm 5: Train through 1 batch data

Data: $X \in \mathbb{R}^{n*N}$;$Y \in \mathbb{R}^{m*N}$;ϵ:Learning Rate;λ:Normalize parameter

Result: Loss without Normalization Term

$\hat{Y} \leftarrow X$;

for *layer in layers* **do**

  $\hat{Y} \leftarrow layer.forward(\hat{Y})$

Calculate Loss;

Calculate $\frac{dL}{d\hat{Y}}$;

for *layer in* ***REVERSE*** *layers* **do**

  $\frac{dL}{d\hat{Y}} \leftarrow layer.calc_grad(\frac{dL}{d\hat{Y}})$;

  $layer.backward(\epsilon, \lambda)$;

return *Loss*

4 实验代码

该项目所有代码已上传github 仓库，在此也将展示所有代码，首先展示算子类代码：

Code Listing 1: 线性算子代码

```
class Linear:
    def __init__(self, in_ch, out_ch, bias=False):
        self.in_ch = in_ch
        self.out_ch = out_ch
        self.W = np.random.random((out_ch, in_ch))
        self.dLdW = None
        self.bias = bias
        if self.bias:
            self.b = np.zeros((out_ch, 1))
            self.dLdb = None
        self.y = None
        self.x = None
        self.dydx = None
        self.train_flag = False

    def train(self):
        self.train_flag = True
```

```
    def test(self):
        self.train_flag = False

    def forward(self, x):
        """
        Forward Pass
        :param x: in_ch * Batch_size
        :return: out_ch * Batch_size
        """
        y = np.einsum('ij,jn->in', self.W, x)
        if self.bias:
            y = y + self.b
        if self.train_flag:
            self.y = y
            self.x = x
        return y

    def calc_grad(self, dLdy):
        """
        Calculate the gradient
        :param dLdy: 1 * out_ch * Batch_size
        :return: dLdx: 1 * in_ch * Batch_size
        """
        dLdx = np.einsum('kin,ij->kjn', dLdy, self.W)
        self.dLdW = np.einsum('kin,jn->ijn', dLdy, self.x)

        if self.bias:
            self.dLdb = dLdy.transpose((1, 0, 2))
        return dLdx

    def back_ward(self, lr, alpha=.001):
        self.W = self.W - lr * np. sum(self.dLdW, axis=-1) - 2*lr*alpha*self.W
        if self.bias:
            self.b = self.b - lr * np. sum(self.dLdb, axis=-1)

    def show(self):
        print('Linear\n␣in:{}␣out:{}␣bias:{}\n'.
            format(self.in_ch, self.out_ch, self.bias))
```

Code Listing 2: Sigmoid 算子代码

```
class Sigmoid:
    def __init__(self):
        self.train_flag = False
        self.x = None
```

```
        self.y = None

    def train(self):
        self.train_flag = True

    def test(self):
        self.train_flag = False

    def forward(self, x):
        # print('run')
        y = 1/(1+np.exp(-x))
        if self.train_flag:
            self.x = x
            self.y = y
        return 1/(1+np.exp(-x))

    def calc_grad(self, dLdy):
        """
        Calculate the gradient
        :param dLdy: 1 * c * Batch_size
        :return: dLdx: 1 * c * Batch_size
        """
        dLdx = np.einsum('kin,in->kin', dLdy, self.y*(1-self.y))
        return dLdx

    def back_ward(self, lr, alpha=1.):
        pass

    def show(self):
        print('Sigmoid\n')
```

整体网络代码见下：

Code Listing 3: 整体网络代码

```
class net:
    def __init__(self, in_ch, out_ch, layers, bias=False, alpha=.001, reg=False):
        self.in_ch = in_ch
        self.out_ch = out_ch
        self.layers = []
        self.alpha = alpha
        if  len(layers)==1:
            c = layers[0]
            self.layers.append(
                Linear(in_ch, c, bias)
```

```
            )
            self.layers.append(
                Sigmoid()
            )
            self.layers.append(
                Linear(c, out_ch, bias)
            )
            # self.layers.append(
            #     Sigmoid()
            # )
        else:
            self.layers.append(
                Linear(in_ch, layers[0], bias)
            )
            self.layers.append(
                Sigmoid()
            )
            for i in  range( len(layers)-1):
                self.layers.append(
                    Linear(layers[i], layers[i+1], bias)
                )
                self.layers.append(
                    Sigmoid()
                )
            self.layers.append(
                Linear(layers[-1], out_ch, bias)
            )

    def train(self):
        for layer in self.layers:
            layer.train()

    def test(self):
        for layer in self.layers:
            layer.test()

    def forward(self, x):
        """
        forward pass
        :param x: in_ch * N
        :return: out_ch * N
        """
        y = x
        for layer in self.layers:
```

```
            # print(y.shape)
            print(y)
            y = layer.forward(y)
        return y

    def train_loop(self, x, y, lr):
        """
        train model through x
        :param x: in_ch * N
        :param y: 1 * N
        :return: Loss
        """
        _y = x
        for layer in self.layers:
            _y = layer.forward(_y)
        L = np. sum(.5*(_y-y)**2)
        dLdy = np.expand_dims((_y-y), axis=0)
        for i in  range( len(self.layers)):
            layer = self.layers[ len(self.layers)-i-1]
            dLdy = layer.calc_grad(dLdy)
            layer.back_ward(lr, self.alpha)
        return np.mean(L)

    def show_model(self):
        for layer in self.layers:
            layer.show()
```

训练代码见下：

```
import math

from data import *
from net import *

def xor_train(
    iter_num=1000,
    lr_0=1e-3,
    decay_rate=.8,
    decay_step=100,
    alpha=1.,
):
    model = net(2, 1, [10, 10], alpha=alpha, bias=True)
    model.show_model()
    model.train()
    test_x, test_y = generate_xor_data(10, 0, 2)
```

```
for  iter in  range(iter_num):
    lr = lr_0 * (decay_rate**( iter/decay_step))
    train_x, train_y = generate_xor_data(4, 0, 2)
    # train_x = np.array([[0,0],[1,0],[0,1],[1,1]]).transpose()
    # train_y = np.array([[0],[1],[1],[0]]).transpose()
    Loss = model.train_loop(train_x, train_y, lr)
    # if iter%10000==0:
    if  iter % decay_step==0:
        print('Loss{} : {} {}'. format( iter, Loss, lr))

model.test()
test_x = np.array([[0, 0], [1, 0], [0, 1], [1, 1]]).transpose()
_y = model.forward(test_x)
print(_y.transpose())
```

5 实验结果与分析

本次实验使用简易的 01 异或运算拟合，网络设置见下表1：

Layer Name	Parameter	Dimension of Output
Linear	$W^{(1)}$(10*2)	10
Sigmoid	None	10
Linear	$W^{(2)}$(10*10)	10
Sigmoid	None	10
Linear	$W^{(3)}$(1*10)	1

表 1: 网络结构

超参数设置如下表2：

超参数	超参数数值
训练轮数	100000
批大小	4
初始学习率	0.1
学习衰减率	0.98
衰减周期	10000
正则化参数	1e-5

表 2: 超参数设置

训练过程中的学习率采用动态分配策略，此处采用指数衰减，表达为:

$$lr_t = lr_0 * decay\_rate^{\frac{t}{decay\_iter}} \tag{23}$$

损失值变化情况见图1，可以发现最初损失下降很快，随后下降速度变慢。

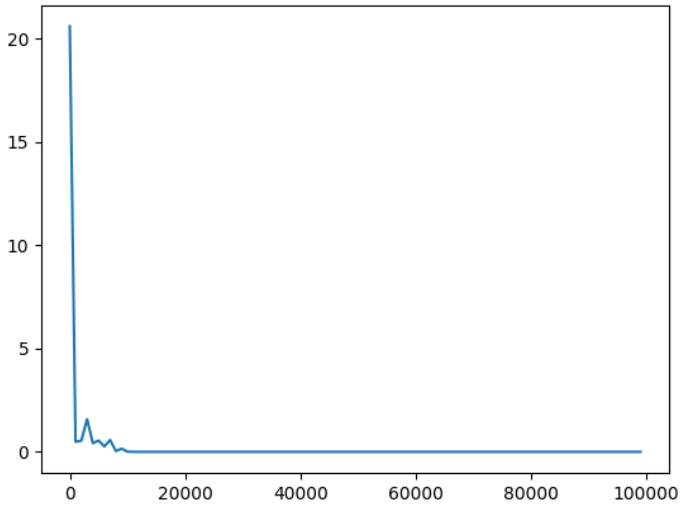

图 1: Loss 随轮数变化情况，横坐标表示轮数，纵坐标表示损失值

(1,1)、(1,0)、(0,1)、(0,0) 的预测结果为 5.18684857e-04、9.99314661e-01、9.99306007e-01、1.26207792e-03，基本正确，符合预期。

为了测试超参数影响，接下来进行消融实验，消融实验结果见下表3。通过消融实验发现，训练结果对于初始学习率较为敏感，不能过大或者过

小；随着批尺寸的提高，效果呈现更优的趋势，符合预期，但是这也通常意味着更高的计算量；值得注意的是正则化参数越小结果越优，这是因为该问题中训练集和测试集并未区分，所以不存在过拟合的问题，此时正则化阻碍了模型的拟合。但是对于一般的问题，正则化项将可以有效提高泛化能力。

初始学习率	**批尺寸**	**正则化参数**	**Loss**
0.1	4	1e-5	1.5417e-6
0.01	4	1e-5	0.3044
1	4	1e-5	0.6469
0.1	1	1e-5	1.6030e-6
0.1	16	1e-5	**1.2576e-6**
0.1	4	1e-4	1.3022e-4
0.1	4	1e-3	0.5002

表 3: 消融实验结果

为了证明正则化的效果，我们将算法迁移到更为困难的图像分类问题中。之后的实验将在 Boston 房价预测数据集 [?] 上进行，该数据集共有 507 个房价预测数据，由 13 个特征预测房价。该实验中，数据进行了标准化预处理，网络结构超参数见表4和表5。

Layer Name	**Parameter**	**Dimension of Output**
Linear	$W^{(1)}$(10*13)	10
Sigmoid	None	10
Linear	$W^{(2)}$(10*10)	10
Sigmoid	None	10
Linear	$W^{(3)}$(1*10)	1

表 4: 波士顿房价预测网络结构

超参数	超参数数值
训练轮数	20000
批大小	32
初始学习率	0.1
学习衰减率	0.98
衰减周期	10000
训练集：测试集	4：1
正则化项	L2

表 5: 超参数设置

实验中，改变正则化参数，并进行实验，结果见表6，结果可以发现，合理地设置正则化参数可以有效提高算法在测试集上的泛化能力。但是其对于参数较为敏感，设置过小的正则化参数会导致模型与未添加正则化参数的模型间无显著性差异，而设置过大的正则化参数会导致算法其泛化能力和拟合能力都低于原模型。综上所述，正则化参数确实可以有效地提高模型在测试集上的泛化能力，但是设置极度依赖经验，具有偶然性和不确定性。

正则化参数	Test Loss
0	4.976e-3
1e-3	1.370e-2
1e-6	4.904e-3
1e-8	2.879e-3
1e-10(best)	**1.968e-3**
1e-12	4.507e-3

表 6: 波士顿房价预测消融实验结果

6 遇到的问题及其解决措施

6.1 无法解决复杂异或问题和图像问题

对于实验内容，我最初考虑的是拟合 [0,100] 范围内的异或问题，但是在实际操作过程发现训练过程难以收敛，最终仅对于简单的 [0,1] 异或

问题进行求解。

该现象可能是模型的参数较少，而广义的异或问题具有极强的非线性，仅仅通过单层或者双层的 MLP 是难以实现拟合的，需要更多层的网络或者更高效的优化策略做支撑。更复杂更多层的网络往往意味着训练的困难，梯度传递可能出现梯度搞炸或者梯度消失，需要引入新的结构（批正则化、残差结构 [2] 等）来提高模型的可训练性。

当然，模型的拟合能力可能是足够的，模型无法收敛也可能是优化算法的超参数设置不够合理导致的。

在验证正则化项效果时，我最初希望在 MNIST[3] 下进行，但是实际操作时模型同样难以收敛，我认为这可能是模型的拟合能力不足导致的，MNIST 手写数据集的数据是 28*28 的灰度图片，在 MLP 框架下需要将其转变为 784 维的向量，而由于实验时间所限，我设置的隐层维度为 10 层和 16 层，甚至低于原有维度，在传播过程极有可能丢失有效信息。

进一步地，将图像转化为一个一维向量本身破坏了数据的结构信息，而结构信息在图像任务中是十分重要的，网络可能需要成百上千倍的数据量才有可能从原有受破坏的数据中学习到空间信息，这对于数据是一种极大的浪费。基于此，图像问题应该采用保留结构信息的算子来进行，比如空间卷积算子。

6.2 超参数选取

超参数的选取极其依赖经验，基于此，我在实验中的超参数设置上花费了大量的时间，其中最有效的方案是直接借鉴其他人经常选用的超参数进行。

同时，在模块初始化参数的设置时也出现了一些问题，最初的初始化参数过于奇异导致模型难以训练，最终对于初始化参数的上界进行改进后成功。

6.3 复杂网络计算较慢

由于本实验仅不用需使用流行的深度学习库函数，所以我仅使用了常用的数学库 numpy 等，而全连接的线性层可以通过 GPU 的并行计算来进行加速，虽然目前业界已经有了一系列实现 numpy、scipy 等基础库使用 cuda 进行加速的方案，但是这需要大量的时间去配置环境和熟悉操

作，该实验提供的时间是不足够我完成这部分的学习的，而如果将此视为一个长期进行的研究的话，我更倾向于使用现在已经极为成熟的深度学习方案，比如 pytorch 或者 tensorflow 等。最终我选择降低模型的复杂度来进行实验。

参考文献

[1] Harrison, D. and Rubinfeld, D.L. 'Hedonic prices and the demand for clean air', J. Environ. Economics Management, vol.5, 81-102, 1978.

[2] He, K., Zhang, X., Ren, S., Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

[3] LeCun, Y., Bottou, L., Bengio, Y., Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278-2324.