

华中科技大学

2023

计算机视觉实验报告

上机实验一：基于前馈神经网络的回归任务设计

专 业： 计算机科学与技术

班 级： 计科 2008 班

学 号： U202015550

姓 名： 刘方兴

完成日期： 2023 年 3 月 26 日



目 录

1. 深度学习框架	1
1.1 编程环境.....	1
1.2 框架选择	1
1.3 代码结构设计	1
2. 实验过程设计	4
2.1 基本实验模型.....	4
2.2 相同层数下隐藏层设置不同的神经元个数.....	5
2.3 设置不同的隐藏层层数.....	7
2.4 设置不同的激活函数	8
3. 实验分析.....	10
3.1 实验结果分析.....	10
3.2 不同网络结构对比.....	10
4. 总结	11
4.1 实验总结	11
4.2 提交文件	11

1. 深度学习框架

1.1 编程环境

使用软件: jupyter notebook

辅助工具: wandb—对实验过程进行实时监控、实现数据可视化

1.2 框架选择

基本框架: tensorflow

1.3 代码结构设计

1) 工具包导入及环境设置部分

```
import tensorflow as tf
import numpy as np

n_example = 5000 #生成数据的个数
n_train = 4500
n_test = 500

n_dim = 2 # 输入数据的维数
#即显模式
tf.compat.v1.enable_eager_execution() # 动态图机制
#tf.compat.v1.disable_eager_execution()
```

```
# 设置学习率, 迭代次数, batch大小
learning_rate = 0.001
epoch = 50
batch = 16
```

```
import wandb #数据可视化
import time
wandb.init(project='ex1', name=time.strftime('%m%d%H%M%S'))
```

wandb: Currently logged in as: jyjy20011fx (jnjy). Use `wandb login --relogin` to force relogin

2) 实验数据准备部分

```
# 生成训练集和测试集 输入数据x,y
# 训练集输入数据 产生特征, 以均匀分布随机生成 数据点(x, y)
train_X = tf.random.uniform((n_train, n_dim), -10,10)

# 测试集输入数据
test_X = tf.random.uniform((n_test, n_dim), -10,10)

# 设置真实的 weight, bias
w_real = [1, 1, 1]
b_real = 0

#生成标准输出数据
labels_train = w_real[0] * train_X[:,0] * train_X[:,0] + w_real[1] * train_X[:,1] * train_X[:,1] + w_real[2] * train_X[:,2] * train_X[:,2]
labels_test = w_real[0] * test_X[:,0] * test_X[:,0] + w_real[1] * test_X[:,1] * test_X[:,1] + w_real[2] * test_X[:,2] * test_X[:,2]

# 为标准输出数据添加噪声得到训练与测试输出数据
train_z = labels_train + tf.random.normal(labels_train.shape, stddev=0.01)
test_z = labels_test + tf.random.normal(labels_test.shape, stddev=0.01)
```

3) 实验模型设计部分

华中科技大学课程设计报告

```
# 定义模型类
class Model():
    def __init__(self):
        # 初始化模型参数
        self.w = tf.Variable(tf.random.normal([n_dim, 10], mean=1.0, stddev=0.1))
        self.b = tf.Variable(tf.zeros(1,))

        self.w1 = tf.Variable(tf.random.normal([10, 1], mean=1.0, stddev=0.1))
        self.b1 = tf.Variable(tf.zeros(1,))

        self.w2 = tf.Variable(tf.random.normal([3, 1], mean=1.0, stddev=0.1))
        self.b2 = tf.Variable(tf.zeros(1,))
    # 变量通过 tf.Variable 类进行创建和跟踪, tf.Variable 表示张量, 对它执行运算可以改变其值, 利用特定运算可以读取和修改此张量的值.
    # 更高级的库 (如 tf.keras) 使用 tf.Variable 来存储模型参数.
    def __call__(self, X): # 该方法的功能类似于在类中重载 () 运算符, 使得类实例对象可以像调用普通函数那样, 以“对象名()”的形式使用.
        # 正向传递
        # 第一层
        H1 = tf.nn.relu(tf.matmul(X, self.w) + self.b)
        # 第二层
        H2 = tf.nn.relu(tf.matmul(H1, self.w1) + self.b1)
        # 第三层
        z = tf.nn.relu(tf.matmul(H2, self.w2) + self.b2)
        return z
```

4) 损失函数 MSE 定义部分

```
# 定义损失函数MSE
# 通常情况下取回归模型的 Loss 函数为 MSE
def squared_loss(y, y_hat, n):
    y_observed = tf.reshape(y, y_hat.shape)
    return tf.matmul(tf.transpose(y_observed - y_hat),
                     y_observed - y_hat) / 2 / n
```

5) 数据迭代函数设计部分

```
# 设置数据迭代函数
def data_iter(features, labels, mini_batch):
    # Args:
    # - features: 特征矩阵 nxd 维
    # - labels: 样本, nx1 维
    # - mini_batch: 每次抽取的样本数
    features = np.array(features)
    labels = np.array(labels)
    indices = list(range(len(features)))
    random.shuffle(indices)
    for i in range(0, len(indices), mini_batch):
        j = np.array(indices[i:i+mini_batch, len(features)])
        yield features[j], labels[j]
```

6) 批量梯度下降部分

```
# 计算梯度, 并更新模型参数
def sgd(params, lr):
    # Args:
    # params: 模型参数
    # lr: 学习率 learning rate
    for param in params:
        param.assign_sub(lr * t.gradient(l, param))
```

7) 模型训练部分

华中科技大学课程设计报告

```
import random
# 开始训练
for i in range(epoch):
    for X, y in data_iter(train_X, train_z, batch):
        # 在内存中记录梯度过程
        with tf.GradientTape(persistent=True) as t:
            t.watch([model.w, model.b, model.wl, model.bl])
            # 计算本次小批量的 loss
            l = squared_loss(y, model(X), batch)
            # 计算梯度, 更新参数
            sgd([model.w, model.b], learning_rate)
        # 计算本次迭代的总误差

    train_loss = squared_loss(train_z, model(train_X), len(train_X))
    wandb.log({'epoch': i, 'train_loss': tf.reduce_mean(train_loss)})
    print('epoch %d, train_loss %f' % (i + 1, tf.reduce_mean(train_loss)))
    test_loss = squared_loss(test_z, model(test_X), len(test_X))
    wandb.log({'epoch': i, 'test_loss': tf.reduce_mean(test_loss)})
    print('epoch %d, test_loss %f' % (i + 1, tf.reduce_mean(test_loss)))

test_loss = squared_loss(test_z, model(test_X), len(test_X))
print('Final Test loss %f' % (tf.reduce_mean(test_loss)))
```

2. 实验过程设计

2.1 基本实验模型

```
# 定义模型类
class Model():
    def __init__(self):
        # 初始化模型参数
        self.w = tf.Variable(tf.random.normal([n_dim, 3], mean=1.0, stddev=0.1))
        self.b = tf.Variable(tf.zeros(1,))

        self.w1 = tf.Variable(tf.random.normal([3, 1], mean=1.0, stddev=0.1))
        self.b1 = tf.Variable(tf.zeros(1,))

        self.w2 = tf.Variable(tf.random.normal([3, 1], mean=1.0, stddev=0.1))
        self.b2 = tf.Variable(tf.zeros(1,))
        # 变量通过 tf.Variable 类进行创建和跟踪。tf.Variable 表示张量，对它执行运算可以改变其值。利用特定运算可以读取和修改此张量的值。
        # 更高级的库（如 tf.keras）使用 tf.Variable 来存储模型参数。
    def __call__(self, X): # 该方法的功能类似于在类中重载 () 运算符，使得类实例对象可以像调用普通函数那样，以“对象名()”的形式使用。
        # 正向传递
        # 第一层
        H1 = tf.nn.relu(tf.matmul(X, self.w) + self.b)
        # 第二层
        H2 = tf.nn.relu(tf.matmul(H1, self.w1) + self.b1)
        # 第三层
        z = tf.nn.sigmoid(tf.matmul(H2, self.w2) + self.b2)
        return z
```

图 2.1 基本模型结构

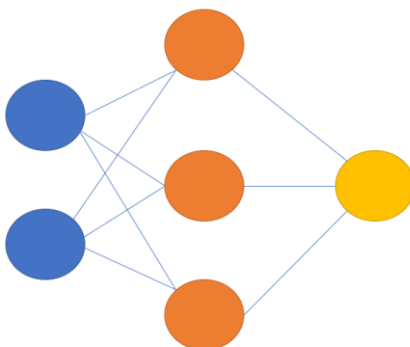


图 2.2 基本模型示意图

在模型中加入了一层隐藏层，在隐藏层中添加了三个神经元。

实验结果（loss 函数采用 MSE）：

```
epoch 39, train_loss 99.284737
epoch 39, test_loss 98.935883
epoch 40, train_loss 98.666824
epoch 40, test_loss 97.811905
Final Test loss 97.811905
```

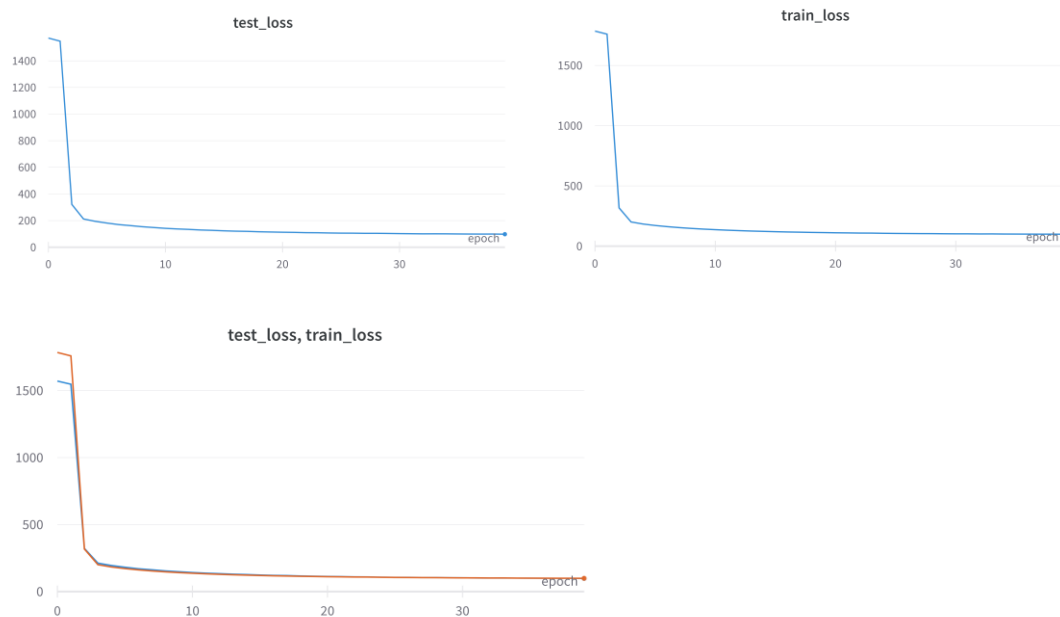


图 2.2 基本模型实验结果图

在运行 40 个 epoch 后，模型基本收敛，最终 loss 从接近 2000 下降到了 97.811905

2.2 相同层数下隐藏层设置不同的神经元个数

在保持网络架构不变的前提下，尝试增加隐藏层神经元个数（3->10->30->50）

实验结果（loss 函数采用 MSE）：

隐藏层设置 10 个神经元：

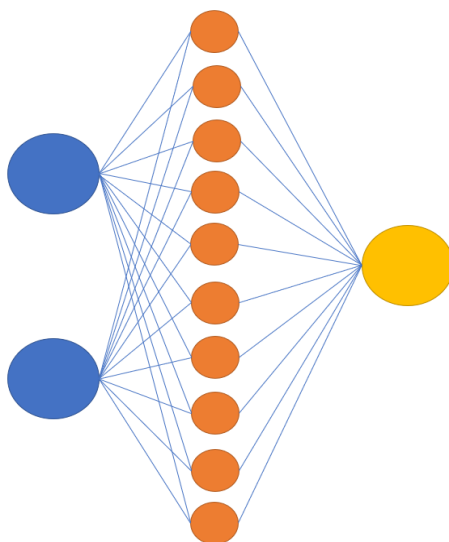
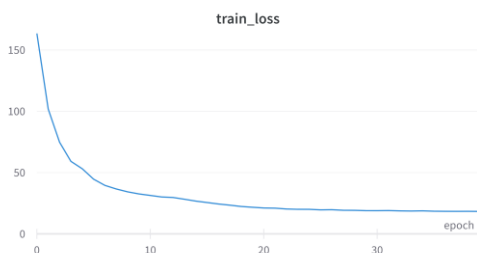
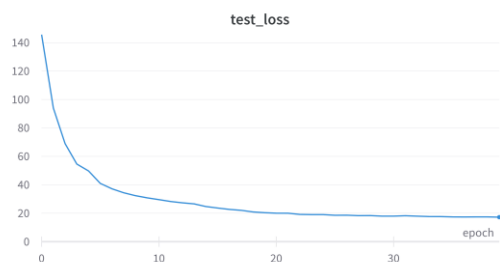
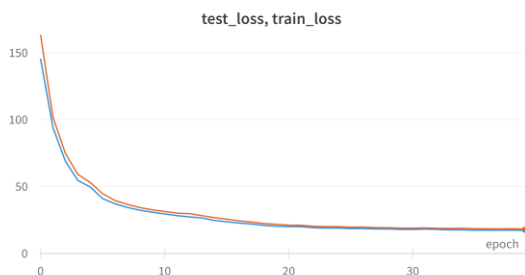


图 2.3 模型示意图

华中科技大学课程设计报告

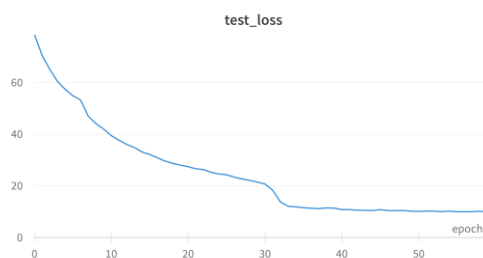
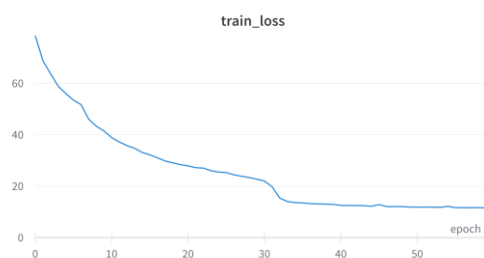


```
epoch 40, train_loss 18.318260  
epoch 40, test_loss 17.213566  
Final Test loss 17.213566
```

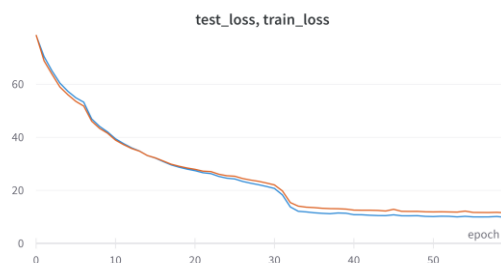


在运行 40 个 epoch 后，模型基本收敛，最终 loss 下降到了 17.213566

隐藏层设置 30 个神经元：

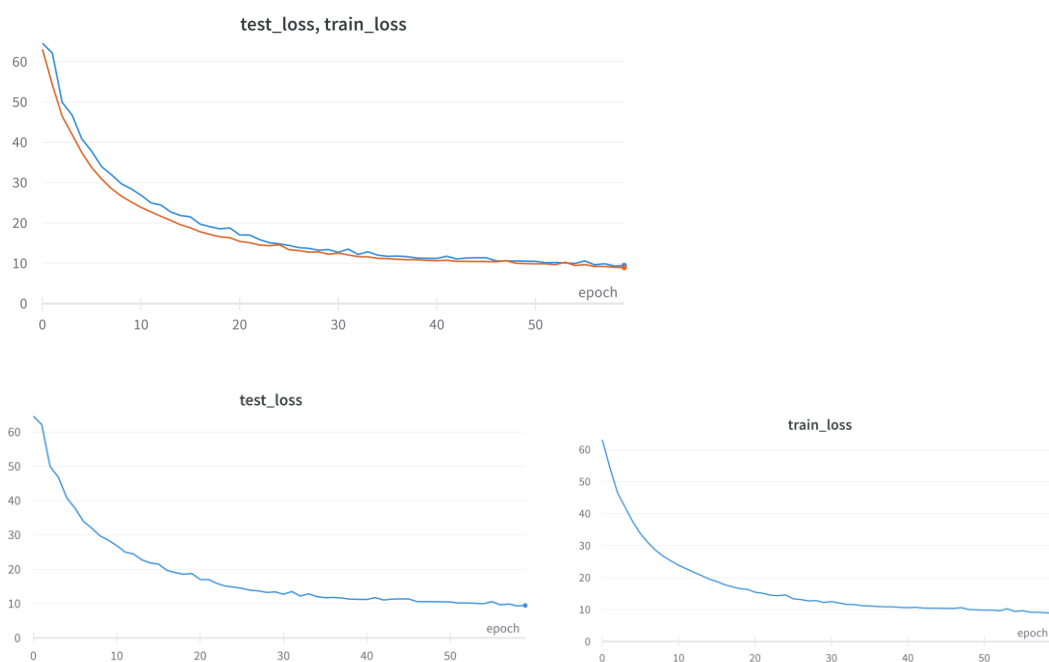


```
epoch 39, train_loss 13.053830  
epoch 39, test_loss 11.453703  
epoch 40, train_loss 12.926464  
epoch 40, test_loss 11.362703
```



在运行 40 个 epoch 后，模型基本收敛，最终 loss 下降到了 11.362703

隐藏层设置 50 个神经元：



在运行 40 个 epoch 后，模型基本收敛，最终 loss 下降到了 10.619

2.3 设置不同的隐藏层层数

在基本模型的基础上，尝试增加隐藏层个数

实验结果 (loss 函数采用 MSE)：

1) 模型结构：2-3-2-1

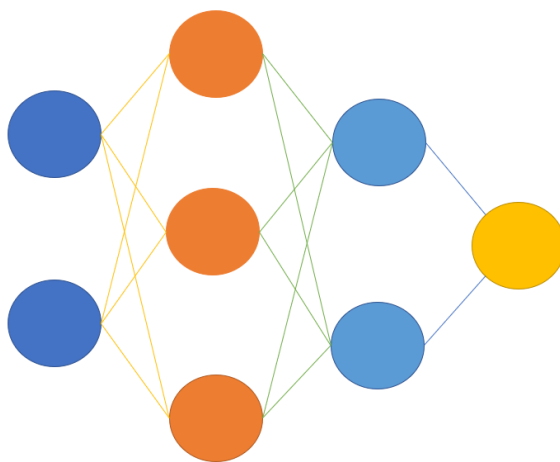
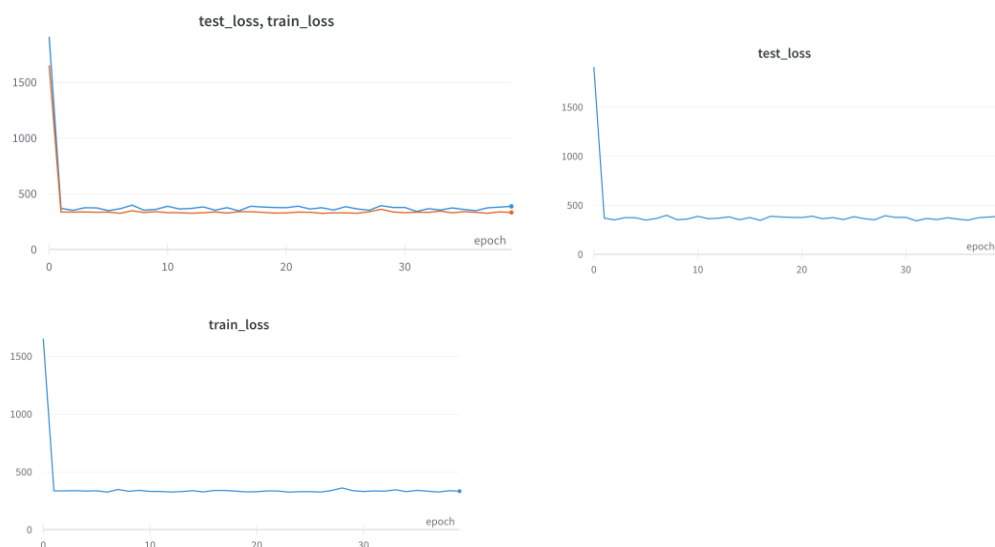


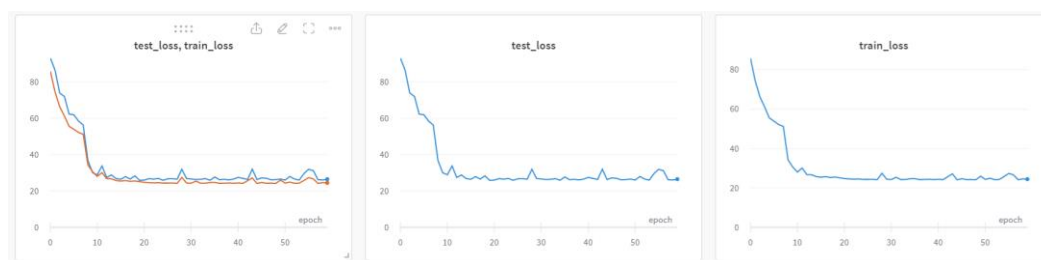
图 2.4 模型示意图

华中科技大学课程设计报告



在运行 40 个 epoch 后，模型基本收敛，最终 loss 下降到了 300+

2) 模型结构：2-10-3-1



```
epoch 39, train_loss 13.053830
epoch 39, test_loss 11.453703
epoch 40, train_loss 12.926464
epoch 40, test_loss 11.362703
```

在运行 40 个 epoch 后，模型基本收敛，最终 loss 下降到了 11.362703

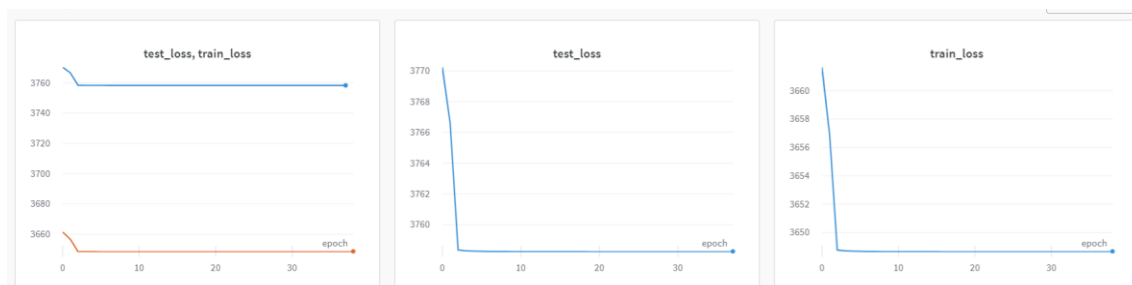
2.4 设置不同的激活函数

在相同的网络架构（2-3-2-1）下，分别尝试 relu 和 sigmoid 两种激活函数：

实验结果（loss 函数采用 MSE）：

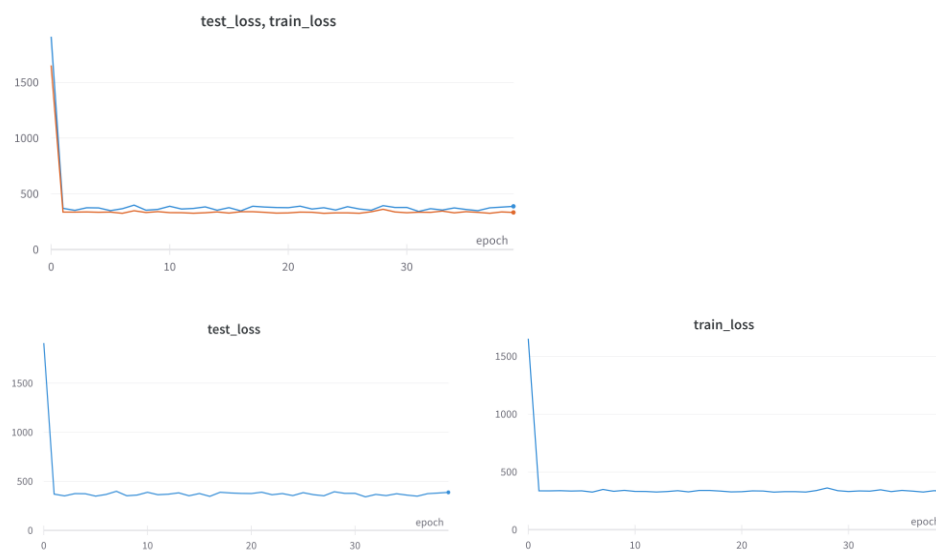
华中科技大学课程设计报告

1) sigmoid:



在运行 40 个 epoch 后，模型基本收敛，但最终 loss 仍旧非常大

2) relu:



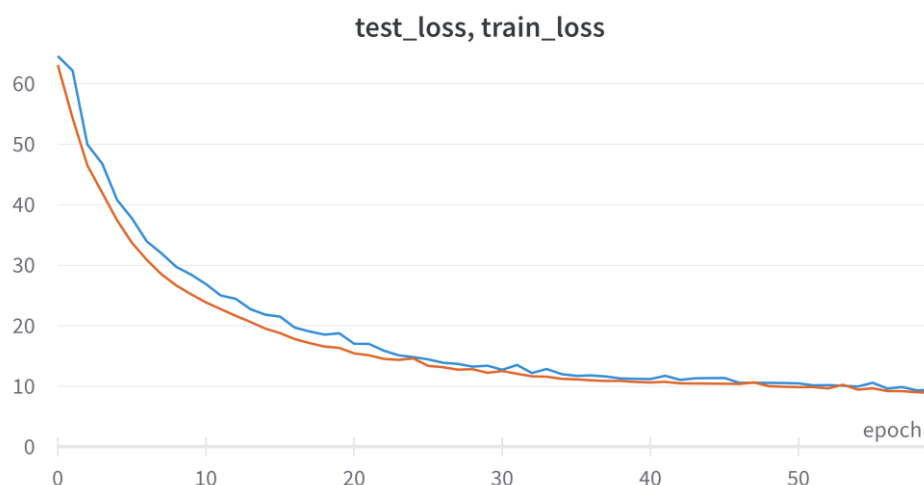
在运行 40 个 epoch 后，模型基本收敛，最终 loss 下降到了 300+

3. 实验分析

3.1 实验结果分析

任务要求：设计一个前馈神经网络，对一组数据实现回归任务。

实验结果：在本实验的架构基础上，若采用 2-50-1 的网络架构，进行 60 个 epoch 的训练，可以将 loss(MSE) 降低到 10 左右。



根据该评价指标，可以发现本次设计的网络模型较好的实现了要求的二元非线性回归任务。

3.2 不同网络结构对比

1) 相同层数下隐藏层设置不同的神经元个数

根据 2.2 的结果，就该二元回归问题而言。可以发现在一定范围之内，增加单层网络神经元的个数可以有效提高模型的预测能力。

2) 设置不同的隐藏层层数

根据 2.3 的结果，就该二元回归问题而言。尝试添加隐藏层的个数并不会为模型的预测能力带来很大的提升。其原因可能是该问题相对较为简单，不需要很深的网络结构就能很好的近似表示所给的方程。

3) 设置不同的激活函数

根据 2.4 的结果，就该二元回归问题而言。可以很容易的发现 relu 函数的表现明显优于 sigmoid 函数。





4. 总结

4.1 实验总结

本次实验设计了一个基本的网络模型，较好的实现了要求的二元非线性回归任务。

在实验的过程中也对网络的整体架构有了更加清晰的认识。

4.2 提交文件

 ex1_result	2023/3/27 17:03	文件夹	
 CV_ex1	2023/3/27 16:43	Jupyter 源文件	15 KB
 刘方兴-U202015550-实验报告 1	2023/3/27 17:00	Microsoft Word ...	4,738 KB
 刘方兴-U202015550-实验报告 1	2023/3/27 17:00	Foxit PDF Editor ...	846 KB

包含实验报告、源代码、实验结果文件（.PNG、.CSV）