1 实验原理

1.1 数据预处理

对房屋年龄，主人收入，房屋价格应用归一化处理，归一化公式为：

$$x = \frac{x - \min(x)}{\max(x) - \min(x)} \tag{1}$$

将各个数据映射到 [0,1] 区间，便于参数的训练和加速收敛。

1.2 感知机

作为回归问题，本实验中用到的一层、两层、三层感知机网络结构如图 1 所示
本实验将仔细对比各层数感知机的性能差异

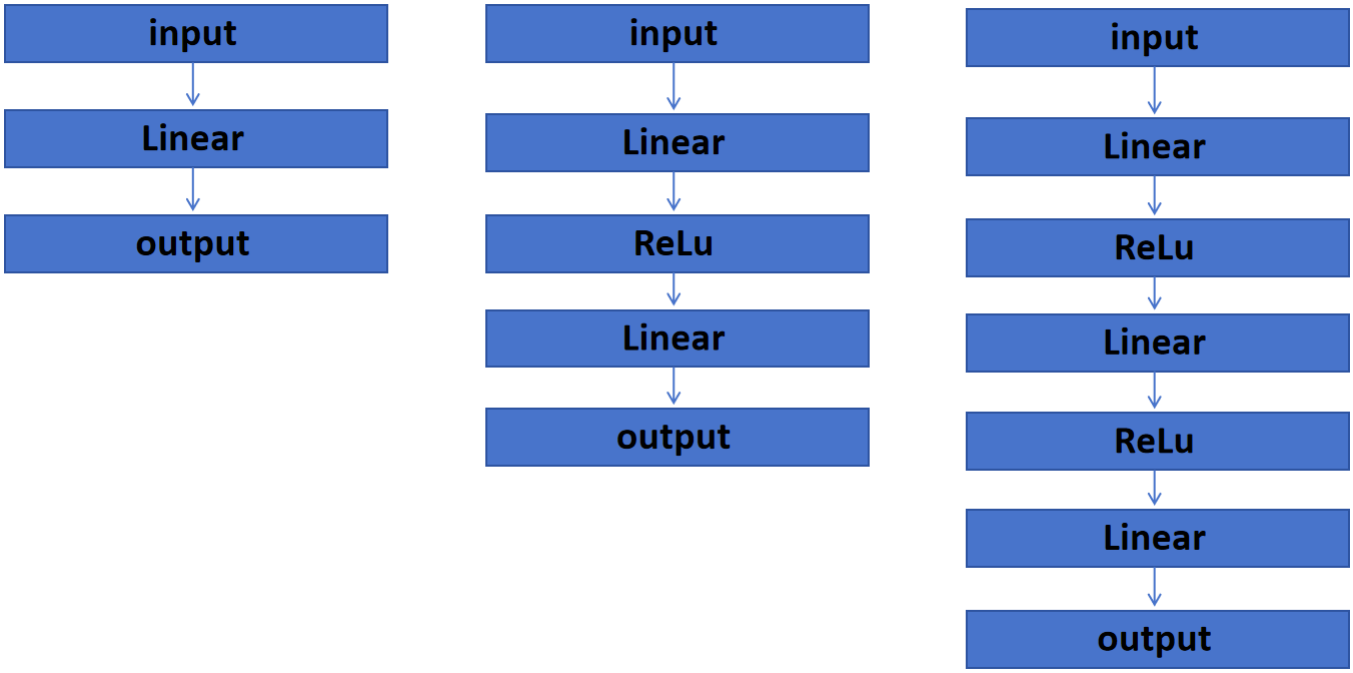


图 1: 感知机结构

1.3 激活函数

本实验使用 ReLU 作为激活函数，其表达式为：

$$ReLU(x) = \max(0, x) \tag{2}$$

此激活函数的优点是计算简单，且不存在梯度消失问题，适合用于本回归问题。
且本回归问题参数范围均为 [0,1]，不会存在“神经元死亡”的 Relu 经典缺陷。

1.4 参数初始化

本实验使用 He 初始化，其表达式为：

$$w = \sqrt{\frac{2}{n}} * \text{randn}(n, m) \tag{3}$$

这里 n 为输入层神经元个数，m 为输出层神经元个数。
He 初始化的目的是在使用 ReLU 激活函数时，保持各层输出的方差稳定，避免前向传播时信号因激活函数（ReLU 的非线性特性）导致方差衰减或放大，同时确保反向传播时梯度的稳定性

1.5 损失函数

本实验使用均方误差作为损失函数，其表达式为：

$$loss = \frac{1}{2n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{4}$$

这里分母缩放 2 倍，便于求导

1.6 正向传播

以一层神经元为例，正向传播公式为：

$$\mathbf{z_1} = \mathbf{x}\mathbf{W_1} + \mathbf{b_1} \tag{5}$$

$$\mathbf{a_1} = ReLU(\mathbf{z_1}) \tag{6}$$

$$\mathbf{y_{pred}} = \mathbf{a_1}\mathbf{W_2} + \mathbf{b_2} \tag{7}$$

1.7 反向传播

以一层神经元为例，反向传播公式为：

$$\frac{\partial loss}{\partial \mathbf{y_{pred}}} = \frac{1}{n}(\mathbf{y_{pred}} - \mathbf{y}) \tag{8}$$

$$\frac{\partial \mathbf{y_{pred}}}{\partial \mathbf{W_2}} = \mathbf{a_1}^T \tag{9}$$

$$\frac{\partial \mathbf{y_{pred}}}{\partial \mathbf{b_2}} = \mathbf{1} \tag{10}$$

$$\frac{\partial \mathbf{y_{pred}}}{\partial \mathbf{a_1}} = \mathbf{W_2}^T \tag{11}$$

$$\frac{\partial \mathbf{a_1}}{\partial \mathbf{z_1}} = \mathbf{1}_{(\mathbf{z_1}>0)} \tag{12}$$

$$\frac{\partial \mathbf{z_1}}{\partial \mathbf{W_1}} = \mathbf{x}^T \tag{13}$$

$$\frac{\partial \mathbf{z_1}}{\partial \mathbf{b_1}} = \mathbf{1} \tag{14}$$

因此，各参数的梯度计算为：

$$\frac{\partial loss}{\partial \mathbf{W_2}} = \frac{\partial loss}{\partial \mathbf{y_{pred}}} \cdot \frac{\partial \mathbf{y_{pred}}}{\partial \mathbf{W_2}} = \frac{1}{n}(\mathbf{y_{pred}} - \mathbf{y}) \cdot \mathbf{a_1}^T \tag{15}$$

$$\frac{\partial loss}{\partial \mathbf{b_2}} = \frac{\partial loss}{\partial \mathbf{y_{pred}}} \cdot \frac{\partial \mathbf{y_{pred}}}{\partial \mathbf{b_2}} = \sum \frac{1}{n}(\mathbf{y_{pred}} - \mathbf{y}) \tag{16}$$

$$\frac{\partial loss}{\partial \mathbf{W_1}} = \frac{\partial loss}{\partial \mathbf{y_{pred}}} \cdot \frac{\partial \mathbf{y_{pred}}}{\partial \mathbf{a_1}} \cdot \frac{\partial \mathbf{a_1}}{\partial \mathbf{z_1}} \cdot \frac{\partial \mathbf{z_1}}{\partial \mathbf{W_1}} = \mathbf{x}^T \cdot \left(\frac{1}{n}(\mathbf{y_{pred}} - \mathbf{y}) \cdot \mathbf{W_2}^T \odot \mathbf{1}_{(\mathbf{z_1}>0)} \right) \tag{17}$$

$$\frac{\partial loss}{\partial \mathbf{b_1}} = \frac{\partial loss}{\partial \mathbf{y_{pred}}} \cdot \frac{\partial \mathbf{y_{pred}}}{\partial \mathbf{a_1}} \cdot \frac{\partial \mathbf{a_1}}{\partial \mathbf{z_1}} \cdot \frac{\partial \mathbf{z_1}}{\partial \mathbf{b_1}} = \sum \left(\frac{1}{n}(\mathbf{y_{pred}} - \mathbf{y}) \cdot \mathbf{W_2}^T \odot \mathbf{1}_{(\mathbf{z_1}>0)} \right) \tag{18}$$

其中，$\odot$ 表示元素级乘法，$\mathbf{1}_{(\mathbf{z_1}>0)}$ 是一个指示函数，当 $\mathbf{z_1} > 0$ 时为 1，否则为 0。

1.8 批量梯度下降

设置 batch_size, 在一个 epoch 中，随机打乱数据，并分批次训练，每批次计算梯度，并更新参数。此效果由于计算全局梯度。

1.9 学习率调度器

本实验使用学习率调度器，其表达式为：

$$lr = lr * \gamma \tag{19}$$

这里 γ 为衰减系数，本实验中根据神经网络层数取 0.999-0.9999，使得学习率在训练初期较大，后期较小，有助于收敛。

1.10 早停机制

本实验使用早停机制，当验证集损失连续 100 轮不再下降时，停止训练。

2 关键代码展示

2.1 数据预处理

拆分 70% 为训练集，15% 为验证集，15% 为测试集

```
df=pd.read_excel('MLP_data.xlsx')
x=df[['housing_age','homeowner_income']].values
y=df['house_price'].values
MAX_x=np.max(x,axis=0)
MIN_x=np.min(x,axis=0)
MAX_y=np.max(y)
MIN_y=np.min(y)
def transform_data(x,x_min,x_max):
    return (x-x_min)/(x_max-x_min)
def inverse_transform(x,x_min,x_max):
    return x*(x_max-x_min)+x_min
x=transform_data(x,MIN_x,MAX_x)
y=transform_data(y,MIN_y,MAX_y)
def preprocess_data(x,y):
    idx=list(range(len(x)))
    random.shuffle(idx)
    train_idx=idx[:int(0.7*len(idx))]
    val_idx=idx[int(0.7*len(idx)):int(0.85*len(idx))]
```

```
    test_idx=idx[int(0.85*len(idx)):]

    x_train=x[train_idx]
    y_train=y[train_idx]
    x_val=x[val_idx]
    y_val=y[val_idx]
    x_test=x[test_idx]
    y_test=y[test_idx]

    return x_train,y_train,x_val,y_val,x_test,y_test

x_train,y_train,x_val,y_val,x_test,y_test=preprocess_data(x,y)
```

2.2 感知机的构建

使用类来封装，每一层设置权重与偏置参数:

```
class MLP:
def __init__(self, input_dim, hidden_dim1, hidden_dim2, hidden_dim3
    , output_dim, lr=0.01):
    self.input_dim = input_dim
    self.hidden_dim1 = hidden_dim1
    self.hidden_dim2 = hidden_dim2
    self.hidden_dim3 = hidden_dim3
    self.output_dim = output_dim
    self.lr = lr

    # He 初始化
    self.w1 = np.random.randn(input_dim, hidden_dim1) * np.sqrt(2 /
        input_dim)
    self.b1 = np.zeros(hidden_dim1)
    self.w2 = np.random.randn(hidden_dim1, hidden_dim2) * np.sqrt(2
        / hidden_dim1)
    self.b2 = np.zeros(hidden_dim2)
```

```
    self.w3 = np.random.randn(hidden_dim2, hidden_dim3) * np.sqrt(2
        / hidden_dim2)
    self.b3 = np.zeros(hidden_dim3)
    self.w4 = np.random.randn(hidden_dim3, output_dim) * np.sqrt(2
       / hidden_dim3)
    self.b4 = np.zeros(output_dim)

    self.train_loss_list = []
    self.val_loss_list = []
```

2.3 激活函数与损失函数

```
def relu(self, x):
    return np.maximum(0, x)
def loss(self, y_pred, y_true):
    # 均方误差
    y_pred=y_pred.reshape(-1)
    return np.mean((y_pred - y_true) ** 2)
```

2.4 正向传播

```
def forward(self, x):
    # 缓存输入，用于反向传播
    self.x = x
    # 第一层
    self.z1 = x.dot(self.w1) + self.b1
    self.a1 = self.relu(self.z1)
    # 第二层
    self.z2 = self.a1.dot(self.w2) + self.b2
    self.a2 = self.relu(self.z2)
    # 第三层
    self.z3 = self.a2.dot(self.w3) + self.b3
    self.a3 = self.relu(self.z3)
```

```
    # 输出层（线性）
    self.z4 = self.a3.dot(self.w4) + self.b4
    return self.z4
```

2.5 反向传播

```
def backward(self, y_true):
    # 假设 y_true 形状为 (batch_size,) 或 (batch_size,1)
    y=y_true.reshape(-1, 1)
    batch_size=y.shape[0]

    # 输出层梯度
    grad_z4=2 * (self.z4 - y) / batch_size
    grad_w4=self.a3.T.dot(grad_z4)
    grad_b4=np.sum(grad_z4, axis=0)

    # 第三隐层
    grad_a3=grad_z4.dot(self.w4.T)
    grad_z3=grad_a3 * (self.z3 > 0)
    grad_w3=self.a2.T.dot(grad_z3)
    grad_b3=np.sum(grad_z3, axis=0)

    # 第二隐层
    grad_a2=grad_z3.dot(self.w3.T)
    grad_z2=grad_a2 * (self.z2 > 0)
    grad_w2=self.a1.T.dot(grad_z2)
    grad_b2=np.sum(grad_z2, axis=0)

    # 第一隐层
    grad_a1=grad_z2.dot(self.w2.T)
    grad_z1=grad_a1 * (self.z1 > 0)
    grad_w1=self.x.T.dot(grad_z1)
    grad_b1=np.sum(grad_z1, axis=0)
```

```
    # 参数更新
    self.w4-=self.lr*grad_w4
    self.b4-=self.lr*grad_b4
    self.w3-=self.lr*grad_w3
    self.b3-=self.lr*grad_b3
    self.w2-=self.lr*grad_w2
    self.b2-=self.lr*grad_b2
    self.w1-=self.lr*grad_w1
    self.b1-=self.lr*grad_b1
```

2.6 训练、验证、测试过程

使用批量梯度下降，并使用学习率调度器与早停机制, 同时记录下训练集与验证集的损失

```
def train(self,x_train,y_train,x_val,y_val,epochs,lr,batch_size=32,
   discount_rate=0.999,constrant=0.0001):
    self.lr=lr
    min_lr=1e-7
    val_best_loss=float('inf')
    n_samples=x_train.shape[0]
    stop_count=0
    for epoch in range(epochs):
    idx=np.random.permutation(n_samples)
    x_train=x_train[idx]
    y_train=y_train[idx]
    train_loss=0

    self.lr=max(self.lr*discount_rate,min_lr)

    for i in range(0,n_samples,batch_size):
        x_batch=x_train[i:i+batch_size]
        y_batch=y_train[i:i+batch_size]
        pred=self.forward(x_batch)
```

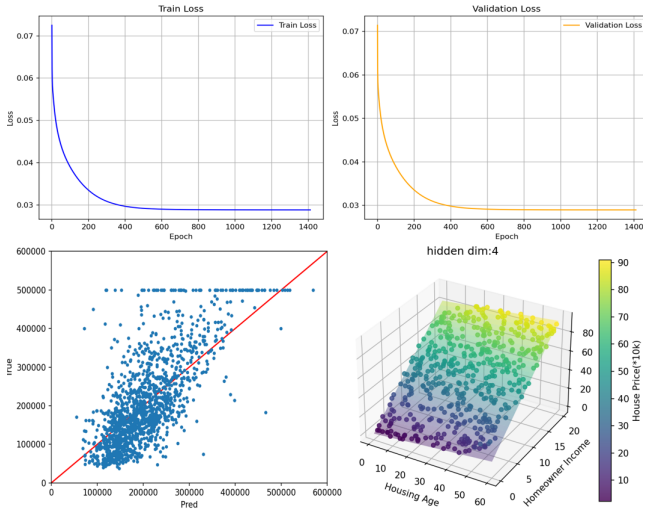
```
            self.loss(pred,y_batch)
            self.backward(y_batch)
        train_pred=self.forward(x_train)
        train_loss=self.loss(train_pred,y_train)
        self.train_loss_list.append(train_loss)

        #validation
        val_pred=self.forward(x_val)
        val_loss=self.loss(val_pred,y_val)
        self.val_loss_list.append(val_loss)
        if val_loss<val_best_loss:
            val_best_loss=val_loss
            self.save_model('best_model')
        if epoch%10==0:
            print(f'Epoch {epoch}, Train Loss: {train_loss}, Val best
                Loss: {val_best_loss}, LR: {self.lr:.2e}')
        if val_loss>val_best_loss:
            stop_count+=1
            if stop_count>=100:
                break
            else:
                stop_count=0
def test(self,x_test,y_test):
pred=self.forward(x_test)
test_loss=self.loss(pred,y_test)
return test_loss,pred
```

3 实验结果

3.1 一层感知机

3.1.1 隐藏层维度:4

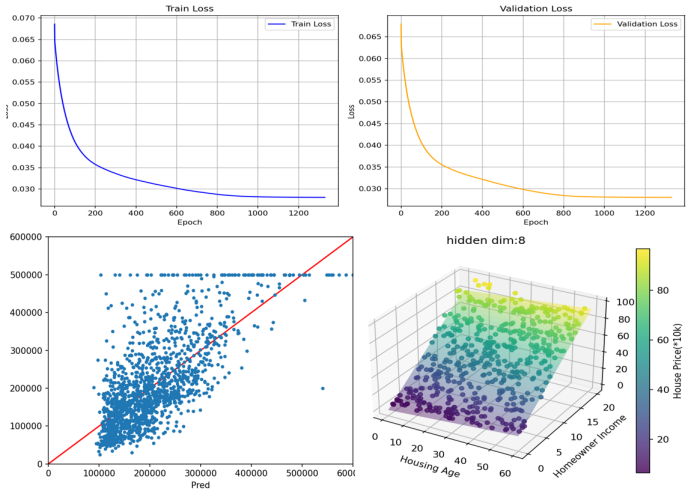
在训练了 1500 轮后收敛，测试集 MSE 为 0.02750544515060434，结果如图所示



其中左下角为测试集中预测值与真实值的散点图，红线为 y=x，点越接近红线，模型效果越好。

3.1.2 隐藏层维度:8

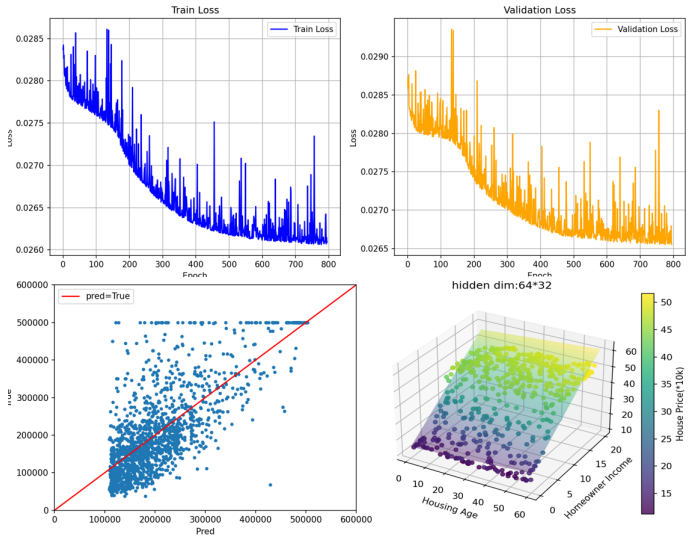
训练了 1320 轮后收敛，测试集 MSE 为 Test Loss: 0.028540942878366043，结果如图所示



3.2 两层感知机

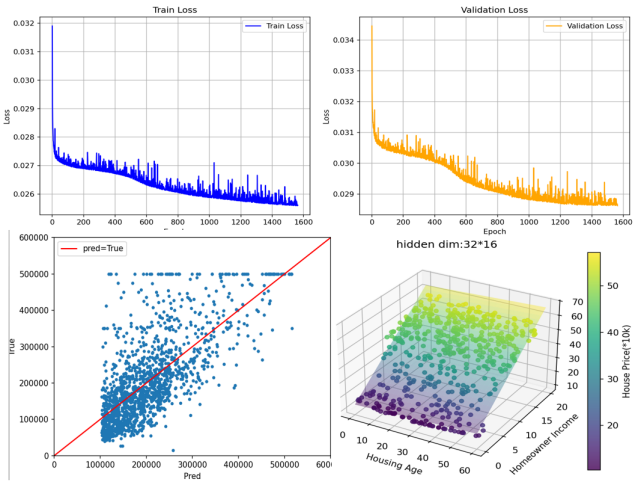
3.2.1 隐藏层维度:64*32

训练了 890 轮后收敛，测试集 MSE 为 Test Loss: 0.028540942878366043，结果如图所示



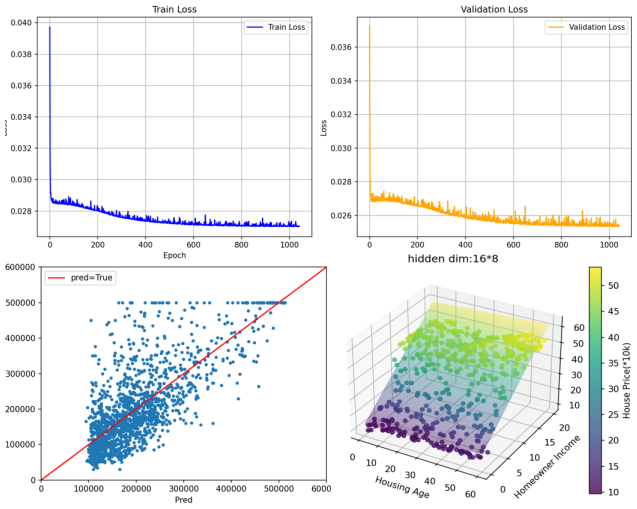
3.2.2 隐藏层维度:32*16

训练了 1660 轮后收敛，测试集 MSE 为 Test Loss:0.028708756026348765，结果如图所示



3.2.3 隐藏层维度:16*8

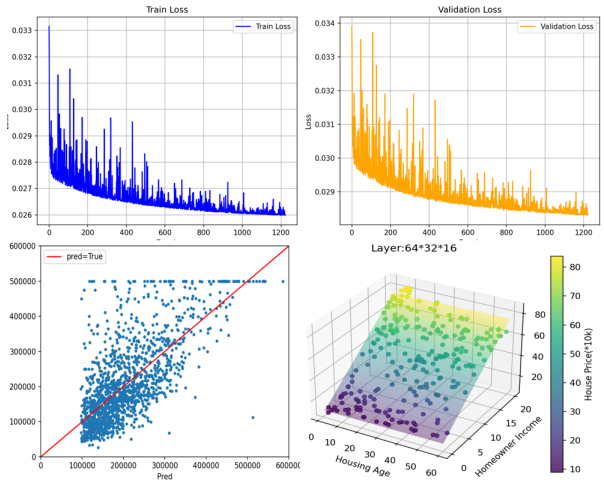
训练了 1050 轮后收敛，测试集 MSE 为 Test Loss:0.026050139810580172，结果如图所示



3.3 三层感知机

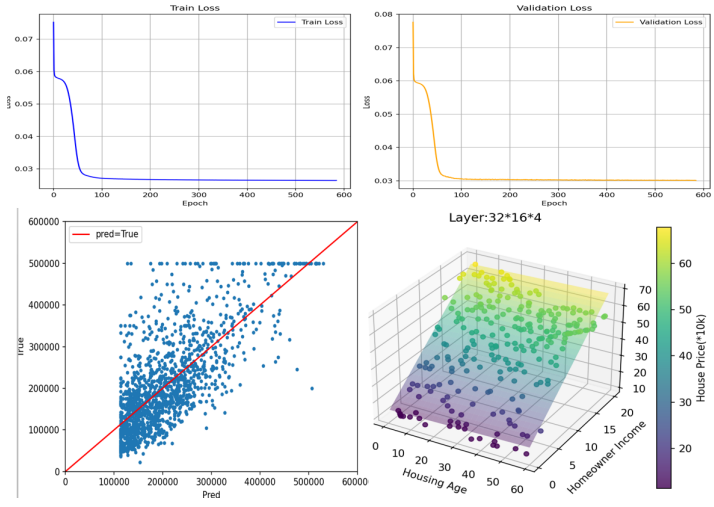
3.3.1 隐藏层维度:64*32*16

训练了 1220 轮后收敛，测试集 MSE 为 Test Loss:0.029437753958786003，结果如图所示



3.3.2 隐藏层维度:32*16*4

训练了 680 轮后收敛，测试集 MSE 为 0.025684265444430112，结果如图所示



3.4 结果综述

表 1: 不同网络结构的实验结果对比

层数	隐藏层维度	收敛轮次	**Test Loss**	学习率	学习衰减率
1	4	1500	0.02750	0.05	0.9999
1	8	1320	0.02854	0.05	0.9999
2	64*32	890	0.02854	1e-2	0.9999
2	32*16	1660	0.02871	1e-2	0.9999
2	16*8	1050	0.02605	1e-2	0.9999
3	64*32*16	1220	0.02944	5e-3	0.999
3	32*16*4	680	0.02568	5e-3	0.999

从各结果的图中看出，随着网络层数的增加，模型的性能表现出不同的特点：
复杂的网络结构并不一定会得到好的结果，损失会产生震荡现象，也会有过拟合的风险。
综合 pred-true 图，单层的感知机效果最优。

4 创新点 & 优化

- 使用 He 初始化，使得各层输出的方差稳定，避免前向传播时信号因激活函数（ReLU 的非线性特性）导致方差衰减或放大，同时确保反向传播时梯度的稳定性
- 使用学习率调度器，使得学习率在训练初期较大，后期较小，有助于收敛
- 使用早停机制，当验证集损失连续 100 轮不再下降时，停止训练，避免过拟合
- 实现了模型保存与加载功能，保存训练过程中性能最佳的模型参数
- 综合对比了不同网络结构，比较了不同隐藏层维度对模型性能的影响
- 使用批量梯度下降（mini-batch）而非全批量或随机梯度下降，平衡了计算效率和收敛稳定性
- 在每个训练周期开始时对训练数据进行随机打乱，增强模型的泛化能力
- 实现了数据归一化处理，加速模型收敛并提高稳定性
- 使用可视化技术展示训练过程和预测结果，包括损失曲线和三维散点图，便于直观理解模型性能

- 设计了约束参数（constrant）来控制验证损失改善的阈值，使早停机制更加灵活