基于前馈神经网络的函数近似

学号：PB210000224　　姓名：陈鸿绪　　日期：3.29.2024

一． 实验原理：

网络结构：前馈神经网络由输入层、隐藏层和输出层组成。输入层负责接收外部数据，隐藏层对数据进行非线性变换，输出层则输出网络的最终结果。实验中我们将探究网络层数、宽度给训练拟合带来的影响。

激活函数：在隐藏层和输出层中，每个神经元都通过一个激活函数进行非线性变换。实验中将探究的激活函数包括 ReLU、Tanh、LeakyReLU。

损失函数：损失函数用于衡量网络输出与实际值之间的差异。常用的损失函数包括均方误差（MSE）、l1 损失、Huber 损失等。训练过程中，通过最小化损失函数来优化网络的权重和偏置。我们将探究不同损失函数带来的影响。实验中将探究的误差类型为 MSE、l1 损失、Huber 损失。

权重和偏置：网络中的每个连接都有一个权重，每个神经元都有一个偏置。这些权重和偏置在训练过程中通过反向传播算法进行更新，以最小化网络输出与实际值之间误差。

二． 实验环境：

表格 1. 版本与配置

Python	3.11.8
Pytorch(GPU)	2.2.1
Numpy:	1.26.4
Pandas	2.2.1
CPU	Intel(R) Core(TM) i9-14900HX
GPU	NVIDIA GeForce RTX 4070 Laptop GPU

三． 实验过程：

数据准备：本次实验中，我们使用随机均匀生成的数据集，对于每一个 N，生成横坐标在[0, 16]中均匀分布的函数上的点坐标，并存入 CSV 文件中。

构建网络以及超参数选择：由于任务是拟合单变量函数，故输入层、输出层必须固定 size 为 1。隐藏层深度、宽度、激活函数、学习率、batch size、epoch 等均为命令行运行程序时的附带参数。

初始化权重和偏置：随机初始化网络中的权重和偏置。

前向传播：将输入数据通过网络进行前向传播，计算网络输出。

计算损失：使用损失函数计算网络输出与实际值之间的差异。并画出模型在验证集上的 loss 变化图像。

反向传播：根据损失函数的梯度，使用反向传播算法更新网络的权重和偏置。

迭代训练：重复训练，直到网络性能达到要求或达到预设的迭代次数。将模型超参数和参数全部存入 checkpoint 文件中。最后选出最佳超参数进行测试集评估。

探究超参数影响：实验中探究一系列超参数对模型性能的影响（包括 batch size、损失函数类型、激活函数类型、模型宽度、模型深度、学习率）。

四 . 关键代码展示：

前馈神经网络代码：

```
class MLPs(nn.Module):

    def __init__(self, size_list: list, activate: str):
        super(MLPs, self).__init__()
        self.size_list = size_list
        self.activate = activate
        self.MLPs_layer = self.__make_layer()

    def __make_layer(self):
        layers = []
        for i in range(len(self.size_list))[:-1]:
            layers.append(nn.Linear(self.size_list[i], self.size_list[i + 1]))
            if i != len(self.size_list)-2:
                if self.activate == "ReLU":
                    layers.append(nn.ReLU())
                elif self.activate == "LeakyReLU":
                    layers.append(nn.LeakyReLU())
                elif self.activate == "Tanh":
                    layers.append(nn.Tanh())
        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.MLPs_layer(x)
        return x

```

模型训练代码：

```
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print(device)
    model = MLPs(size_list, activate)

```

```
    if path and load_model:
        model.load_state_dict(checkpoint['model_state_dict'])
    model.to(device)
    optimizer = opt.Adam(model.parameters(), lr=lr)
    #scheduler = StepLR(optimizer, step_size=1, gamma=0.99)

    criterion = nn.MSELoss()   # 定义损失函数 MSE
    criterion = criterion.to(device)

    for epoch in range(num_epochs):
        progress_bar(epoch, num_epochs, prefix='process: ', length=50)
        model.train()  # 设置模型为训练模式
        for i,(inputs,true_y) in enumerate(train_iter):
            inputs = inputs.to(device)
            true_y = true_y.to(device) # 将输入和标签移至设备
            optimizer.zero_grad()  # 清零梯度
            outputs = model(inputs)  # 前向传播
            loss = criterion(outputs, true_y)  # 计算损失
            loss.backward()  # 反向传播:
            optimizer.step()  # 更新模型参数
        #scheduler.step()
        model.eval()
```

五． 超参数分析：

为了减少篇幅，**这里只展示 N 为 10000 的结果**，且默认隐藏层每一层的 size 都保持一致。

1. 网络深度：

我们在这一小节中固定以下参数的值:

```
1. num_epochs: 10000
2. batch size: 2048
3. learning rate: 0.003
4. Activation Function: ReLU
```

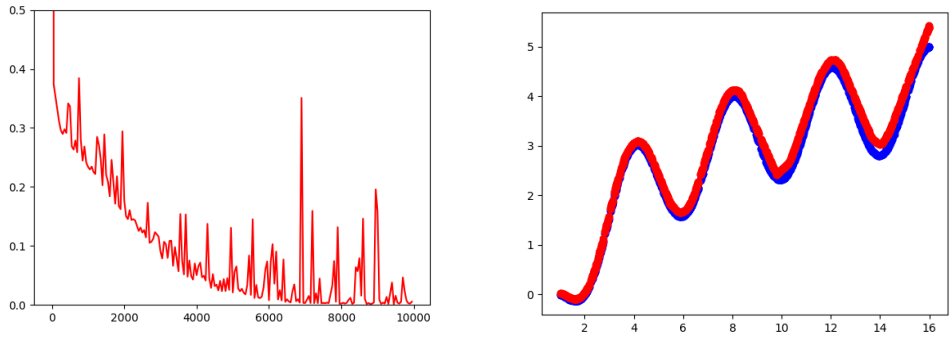

图 1. model size list: [1, 1024, 1]，running_loss: 0.02403175

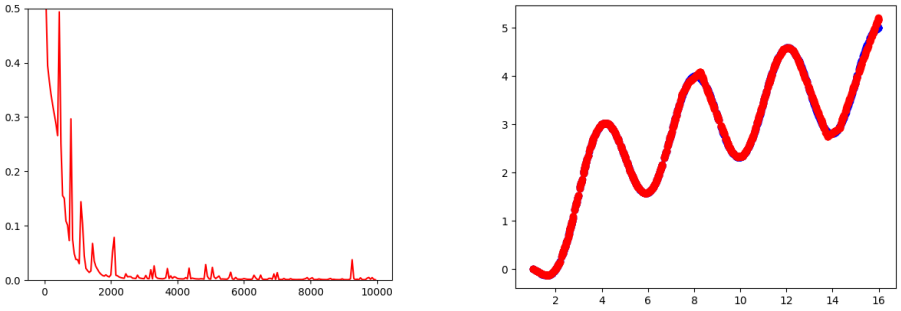

图 2. model size list: [1, 1024, 1024, 1], running_loss: 0.00098391

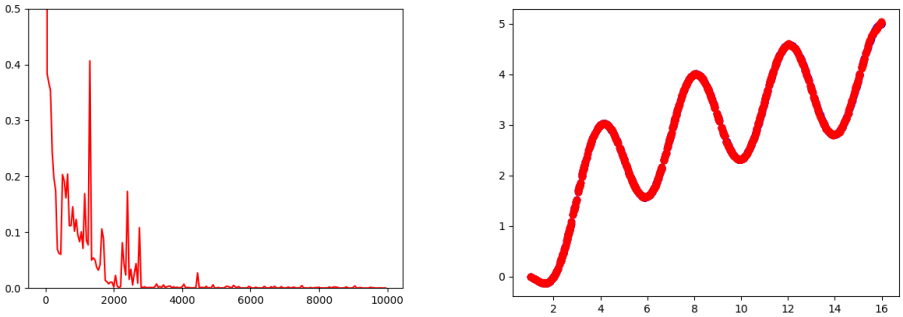

图 3. model size list: [1, 1024×4, 1], running_loss: 4.467e-05

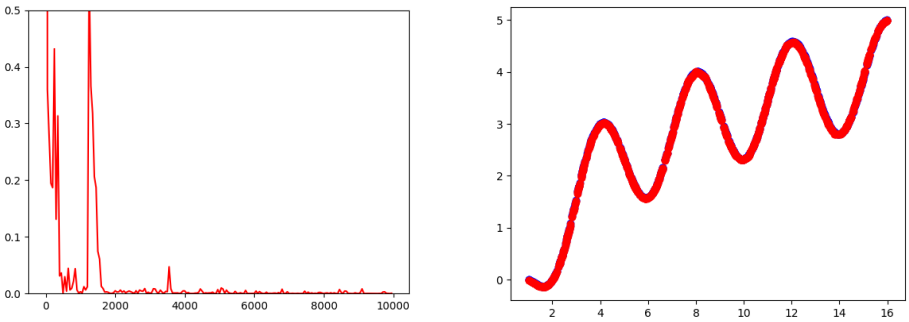

图 4. model size list: [1, 1024×6, 1], running_loss: 0.00021222

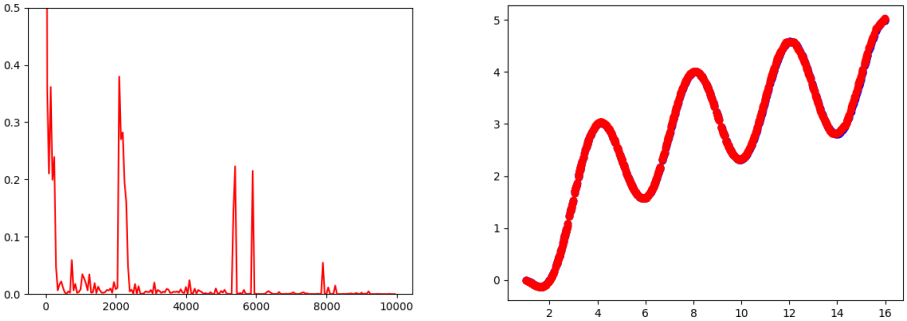

图 5. model size list: [1, 1024×8, 1]. running_loss: 0.00029199

2. 网络宽度：

我们在这一小节中固定以下参数的值：

```
1. num_epochs: 10000
2. batch size: 2048
3. learning rate: 0.003
4. Activation Function: ReLU
```

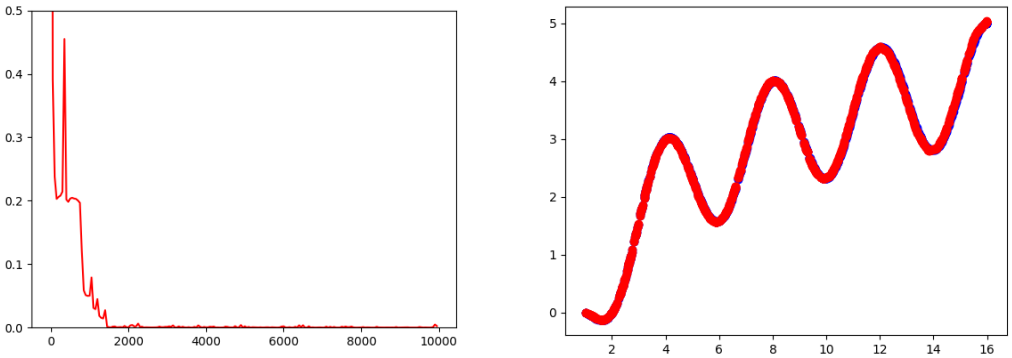

图 6. model size list: [1, 512×4, 1], running_loss: 0.00066505

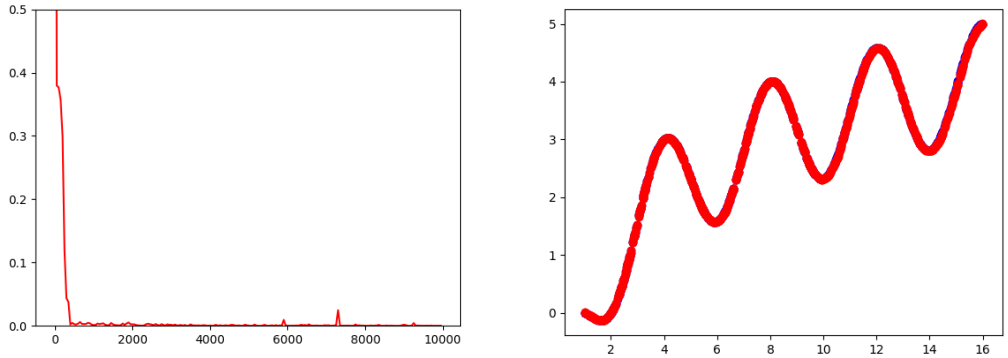

图 7. model size list: [1, 64×4, 1], running_loss: 0.00018237

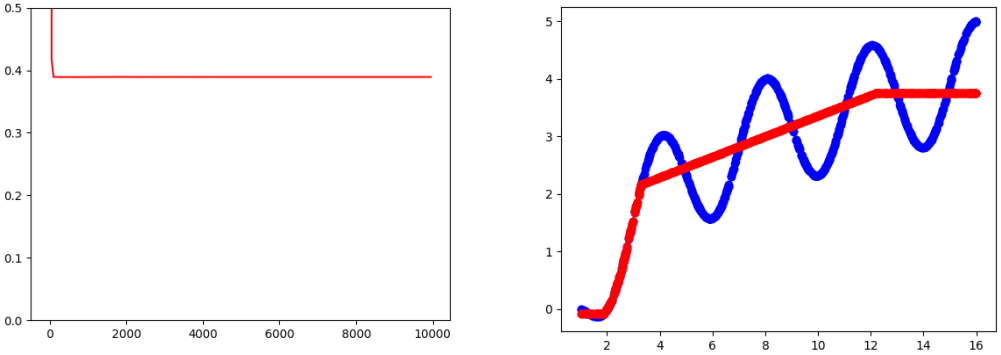

图 8. model size list: [1, 5×4, 1], running_loss: 0.41658613

3. 激活函数：

我们在这一小节中固定以下参数的值:

```
1. num_epochs: 10000
2. batch size: 2048
3. learning rate: 0.003
4. model size list: [1, 512, 512, 512, 1]
```

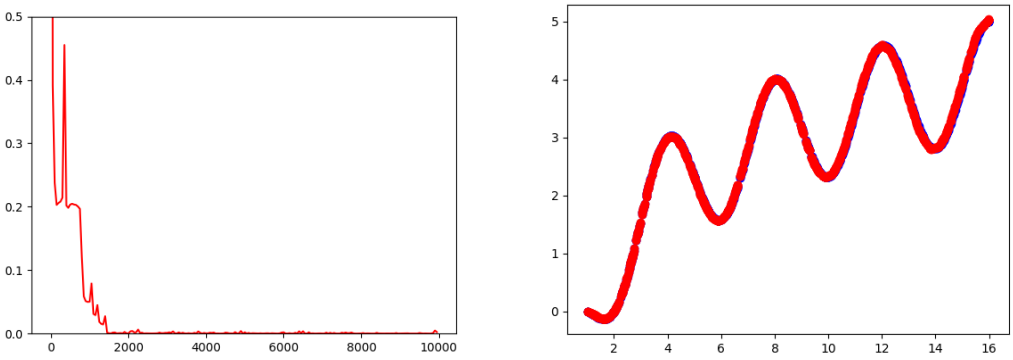

图 9. 激活函数: ReLU, running_loss: 0.000665055

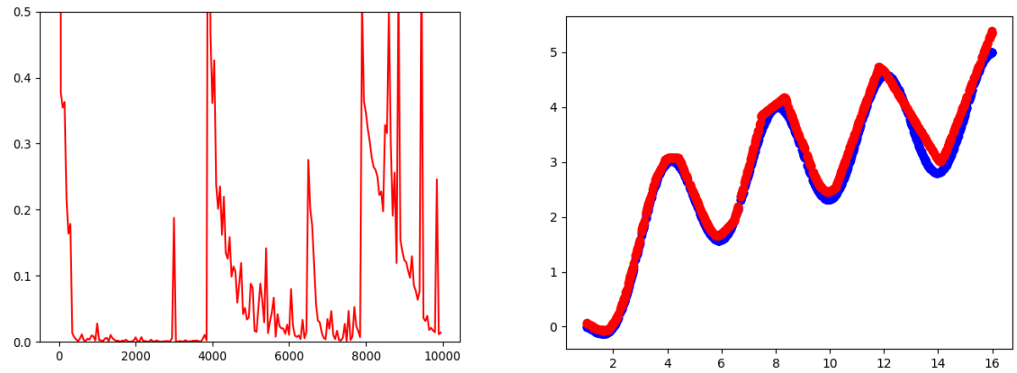

图 10. 激活函数: LeakyReLU, running_loss: 0.02475779

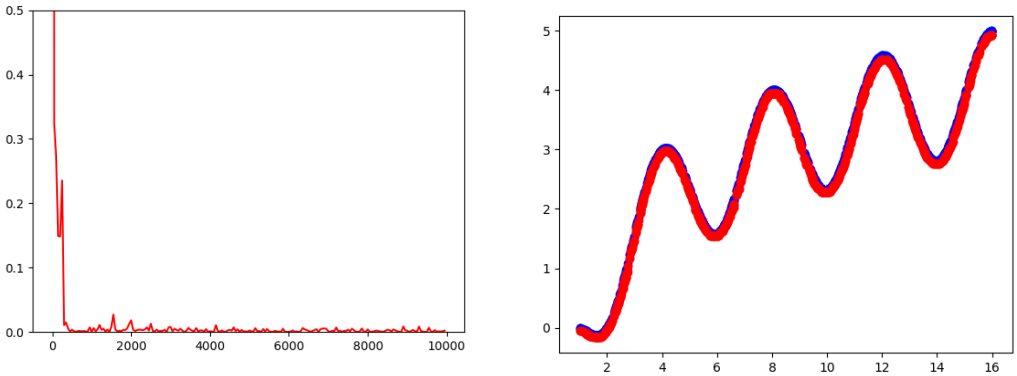

图 11. 激活函数:Tanh, running_loss: 0.00388890

4. 学习率：

我们在这一小节中固定以下参数的值：

```
1. num_epochs: 10000
2. batch size: 2048
3. model size list: [1, 512, 512, 512, 1]
4. Activation Function: ReLU
```

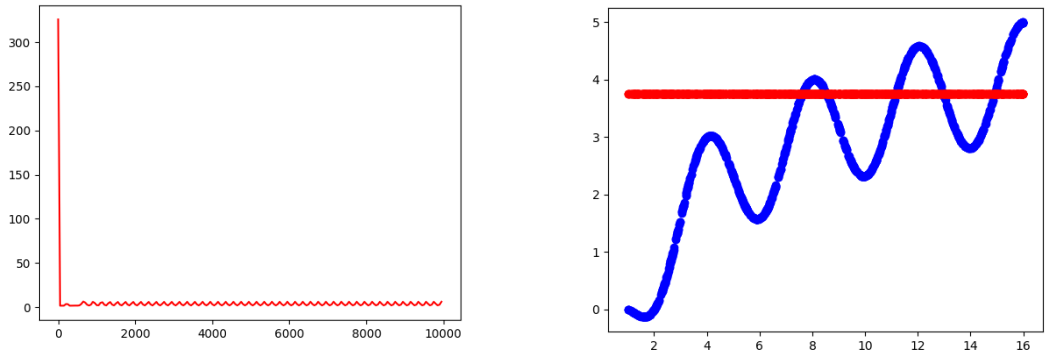

图 12. 学习率: 0.1, running_loss: 2.69729543

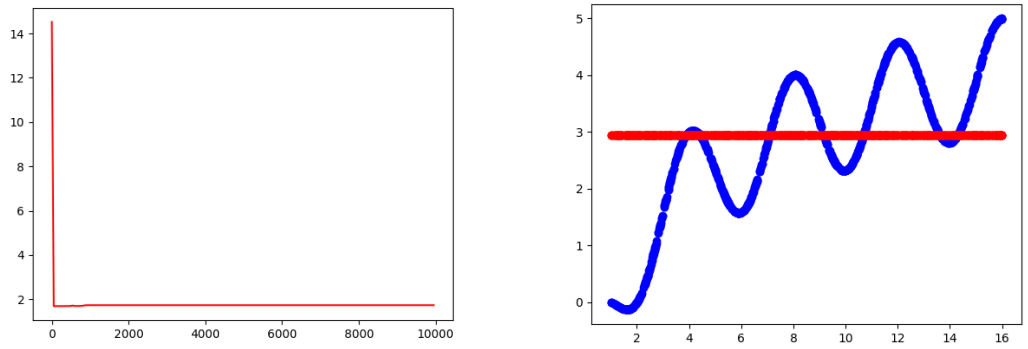

图 13. 学习率: 0.01, running_loss: 1.71281552

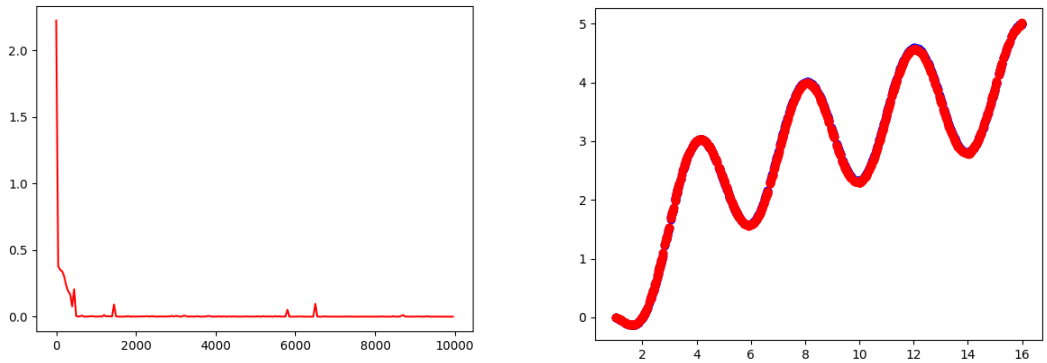

图 14. 学习率: 0.001, running_loss: 0.00023515

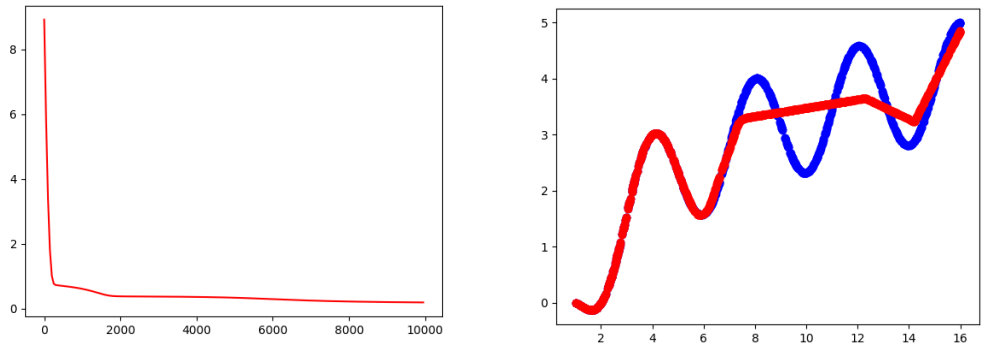

图 15. 学习率: 1e-6, running_loss: 0.20517459

5. 损失函数类型：

```
1. num_epochs: 10000
2. batch size: 2048
3. model size list: [1, 1024, 1024, 1024, 1]
4. Activation Function: ReLU
```

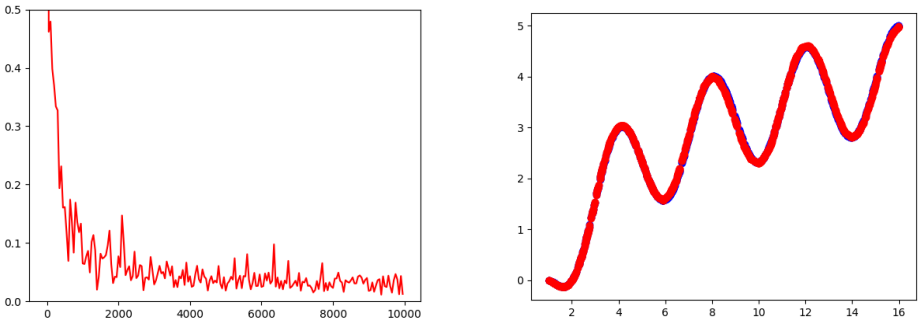

图 16. 损失函数：L1 损失, running_loss: 0.02023888

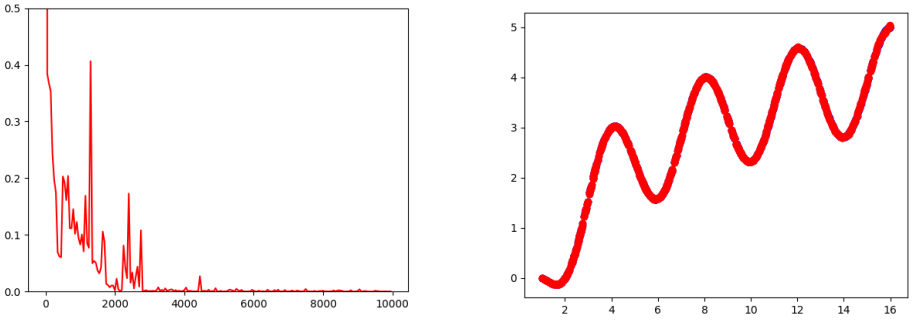

图 17. 损失函数：MSE 损失, running_loss: 4.467e-05

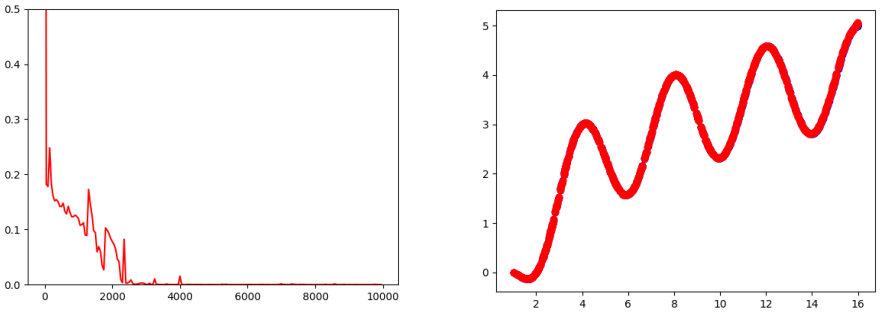

图 18. 损失函数：Huber 损失, running_loss: 4.898e-5

6. Batch Size：

我们在这一小节中固定以下参数的值:

```
1. num_epochs: 10000
2. learning rate: 0.003
3. model size list: [1, 512, 512, 512, 1]
4. Activation Function: ReLU
```

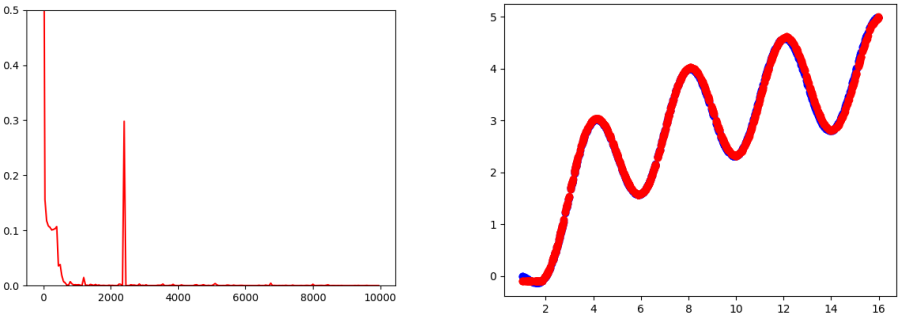

图 19. batch size：2048, running_loss: 0.00068986

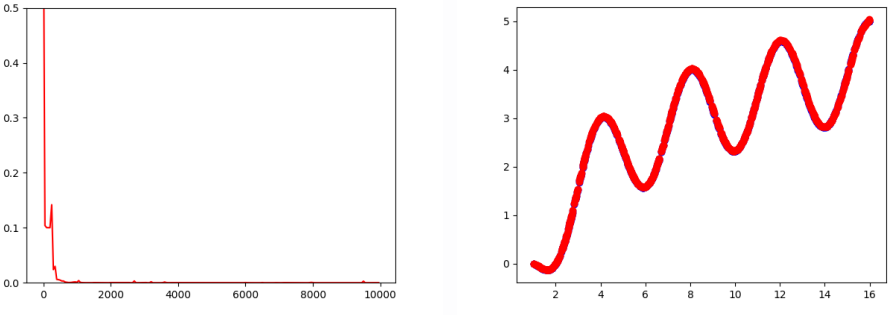

图 20. batch size：1024, running_loss: 8.9595e-05

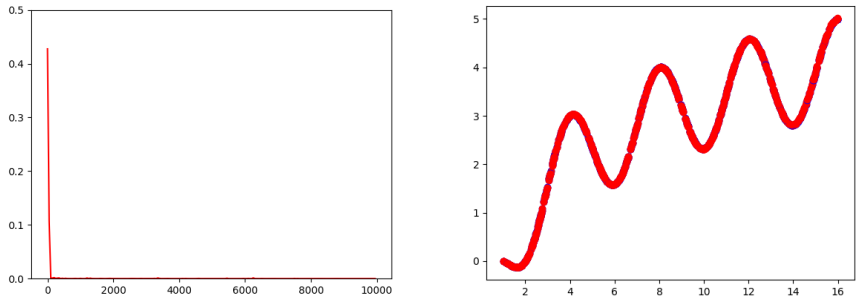

图 21. batch size：512, running_loss: 1.5380e-05

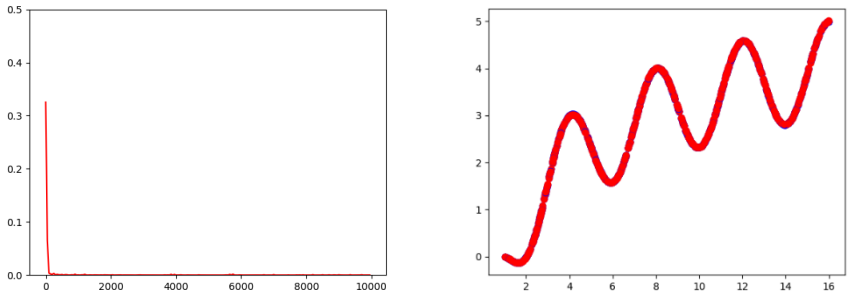

图 22. batch size：256, running_loss: 9.8899e-06

7. 分析：

下面我们将通过分析以上列出的一系列性能图，阐述超参数对于模型性能的影响。

网络深度：在图 1 到图 5 中展示了固定网络宽度为 1024 的条件下，网络深度对于模型收敛效果的影响。在网络只有一层的时候，在验证集上的 loss 量级大致在 1e-2 级别，两层为 1e-4 级别，四层达到最优，量级为 1e-5。然而 4 层后网络层数的增加并没有带来验证集上损失量级的减少。所以我们可以观察到，并不是网络层数越深越好，因为网络层数越深，理论上我们需要更多数据或者更多的 epoch 让模型得到更好的训练拟合。需要通过实践、并结合具体情况选择适合的层数。

网络宽度：保持模型深度为 4，改变模型的宽度为 512、64、5，我们可以发现训练之后的模型在验证集上的 loss 在模型宽度为 5 的时候最大。结合验证集上的拟合图像图 8，可以发现模型处于欠拟合状态。这其实是由于在宽度为 5 的时候，模型的参数量过少，导致模型可以拟合的函数复杂度不高，理论上我们可以在模型宽度较小的时候，通过增加模型深度来提高泛化性能（实验中保存的 N=200 和 N=2000 checkpoint 文件可以表现这点）.

激活函数：在实验中尝试了三种不同的激活函数，分别为 ReLU, Tanh, LeakyReLU。图 9 到图 11 中展示了其余超参数相同，但是在不同激活函数情形下模型在验证集上的性能。其中 ReLU 激活函数效果最佳、Tanh 其次、最后是 LeakyReLU。同时实验过程中也可以注意到 ReLU 的计算效率也是最好的。

学习率：图 12 到图 15 记录了学习率逐渐从 1e-1 降低到 1e-6 过程中，模型在验证集上的表现。1e-1 到 1e-2 我们可以注意到，验证集上的 loss 一直得不到有效降低，模型也得不到很好的拟合（甚至一直为一条直线），这是因为在学习率比较大的时候，模型一方面无法去有效逐步逼近最佳参数，另一方面网络可能会出现“Dead ReLU”现象，即某一层的输出会几乎全部为负值，经过 ReLU 会被截断成 0，一旦该层出现这样的情况会使得模型的向后传播算法无法有效执行，模型无法更新迭代。所以我们观察到在较小的学习率 1e-3 开始时，模型在验证集上得到良好拟合。如果学习率过小，如图 12 所示为 1e-6 的学习率，模型每次循环无法得到充分更新，所以在给定最大 epoch 下，模型处于欠拟合的时候结束了训练。

损失函数：图 16 到图 18 记录了只在不同损失函数类型下，模型在验证集上的表现。我们使用了 L1、MSE 以及 Huber 损失函数，但是根据验证集上的性能显示 MSE 的表现效果最佳，Huber 损失函数效果略低于 MSE，L1 损失在验证集得到的 loss 量级比前者大了 3 个数量级，这是因为原函数是一个完全光滑的曲线，然而这三个损失函数中，只有 L1 损失拟合出的曲线没有良好的光滑性质，所以最后验证集上表现的效果 L1 最差。

Batch Size：图 19 到图 21 记录了不同 batch size 大小下（2024、1024、512 和 256），模型在验证集上的表现。实验过程中可以明显感知到 batch size 越大，训练模型速度越快（前提是在显存没有占满的情况）。然而还可以发现在 batch size 越小的时候，模型在验证集上的 loss 越小，在 batch size 为 256 的时候 loss 甚至达到了 1e-6 的量级，这是因为较小的 batch size 通常有助于模型更好地泛化到新数据，因为它们在训练过程中接触到了更多的数据组合。这种随机性有助于减少过拟合，并可能使模型在未见过的数据上表现更好。

五．最佳参数选择和测试集表现：

最后在经过一系列比较后我们导入如下最佳参数为：

```
1. num_epochs: 10000
2. batch size: 256
3. learning rate: 0.003
4. model size list: [1, 512, 512, 512, 512, 1]
5. Activation Function: ReLU
```

最后在测试集上的 loss 为：3.3246335533476667e-06。这也就表明对于该超参数，前馈神经网络已经很好拟合了该函数。

```
num_epochs: 10000
batch size: 256
learning rate: 0.003
model size list: [1, 512, 512, 512, 512, 1]
Activation Function: ReLU
cuda
Retrain the model from checkpoint
process:  |                                                  -| 100.0%
test running_loss: 3.3246335533476667e-06
```