实验一

袁雨

PB20151804

一、实验要求

使用 pytorch 或者 tensorflow 手写一个前馈神经网络，用于近似函数：

$$y = sin(x) + cos(x) + sin(x)cos(x), x \in [0, 2\pi)$$

并研究网络深度、学习率、网络宽度、激活函数对模型性能的影响。

二、实验步骤

1. **网络框架**：要求选择 pytorch 或 tensorflow 其中之一，依据官方网站的指引安装包。若你需要使用 GPU，可能还需安装 CUDA 驱动。本次实验仅利用 CPU 也可以完成，但仍强烈推荐大家安装 GPU 版本，以满足后续实验需求。
2. **数据生成**：本次实验的数据集仅需使用程序自动生成，即在 [0, 2π) 范围内随机 sample 样本作为 x 值，并计算 $y = sin(x) + cos(x) + sin(x)cos(x)$作为 y 值。要求生成三个**互不相交**的数据集分别作为**训练集、验证集、测试集**。训练只能在训练集上完成，实验调参只能在验证集上完成。
3. **模型搭建**：采用 pytorch 或 tensorflow 所封装的 module 编写模型，例如 torch.nn.Linear(),torch.nn.Relu() 等，无需手动完成底层 forward、backward 过程。
4. **模型训练**：将生成的训练集输入搭建好的模型进行前向的 loss 计算和反向的梯度传播，从而训练模型，同时也建议使用网络框架封装的 optimizer 完成参数更新过程。训练过程中记录模型在训练集和验证集上的损失，并绘图可视化。
5. **调参分析**：将训练好的模型在验证集上进行测试，以 **Mean Square Error(MSE)** 作为网络性能指标。然后，对网络深度、学习率、网络宽度、激活函数等模型超参数进行调整，再重新训练、测试，并分析对模型性能的影响。
6. **测试性能**：选择你认为最合适的（例如，在验证集上表现最好的）一组超参数，重新训练模型，并在测试集上测试（注意，这理应是你的实验中**唯一**一次在测试集上的测试），并记录测试的结果（MSE）。

三、实验过程

1. 实验环境

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

若当前实验环境支持 GPU，则将 device 设置为 "cuda"。否则，将 device 设置为 "cpu"。

2. 生成数据

```
class CustomDataset(Dataset):
    def __init__(self, low, high, size, device, dtype):
        self.X = torch.linspace(low, high, size, device=device,
dtype=dtype).unsqueeze(1)
        self.Y = torch.sin(self.X) + torch.cos(self.X) + torch.sin(self.X) *
torch.cos(self.X)

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        return self.X[idx], self.Y[idx]
```

定义`CustomDataset`类，需要传入的参数包括下界（low）、上界（high）、样本数量（size）、设备（device）和数据类型（dtype）。使用`torch.linspace`生成数据，在指定的区间$[low, high]$内生成一系列 size 个等间隔的数值，保证了数据不相交。

```
dataset = CustomDataset(0, 2 * np.pi, train_size + valid_size + test_size,
device, dtype)
train_dataset, val_dataset, test_dataset =
torch.utils.data.random_split(dataset, [train_size, valid_size, test_size])

train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
val_X, val_Y = val_dataset[:]
test_X, test_Y = test_dataset[:]
```

调用`CustomDataset`类，创建`dataset`，使用`torch.utils.data.random_split`函数将`dataset`随机分为：`train_dataset`、`val_dataset`和`test_dataset`，保证了训练集、验证集、测试集互不相交。

并将`train_dataset`传递给`DataLoader`，创建`train_loader`对象。这个数据加载器会按照`batch_size`指定的大小和`shuffle=True`指定的方式随机将数据打包成小批量以进行训练。

3. 定义前馈神经网络

```
class FeedforwardNet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size,
activation_func):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(input_size, hidden_size),
            activation_func,
            nn.Linear(hidden_size, hidden_size),
            activation_func,
            nn.Linear(hidden_size, hidden_size),
            activation_func,
            nn.Linear(hidden_size, output_size)
        )

    def forward(self, x):
        return self.net(x)
```

定义`FeedforwardNet`类，创建前馈神经网络。在这个模型中，单个神经元的输出被用作下一层神经元的输入，这样整个网络就可以逐层运行，以生成最终的输出。

在`__init__`函数中，模型根据所提供的参数定义一组Sequential层，这些层叠加形成了一个前馈神经网络。

4. 训练

```
def train(net, data_loader, val_X, val_Y, lr, epochs, patience):
    net.to(device)

    optimizer = optim.Adam(net.parameters(), lr=lr)
    scheduler = StepLR(optimizer, step_size=len(data_loader), gamma=0.5)

    train_loss_history = []
    val_loss_history = []

    best_val_loss = float('inf')
    relative_train_loss = float('inf')
    best_model_params = net.state_dict()
    early_stop = 0
    for epoch in range(epochs):
        train_loss_epoch = 0
        for X_batch, Y_batch in data_loader:
            optimizer.zero_grad()
            output = net(X_batch)
            train_loss = f.mse_loss(output, Y_batch)
            train_loss.backward()
            optimizer.step()
            train_loss_epoch += train_loss.item()
        with torch.no_grad():
            output_val = net(val_X)
            val_loss = f.mse_loss(output_val, val_Y)

        train_loss_history.append(train_loss_epoch / len(data_loader))
        val_loss_history.append(val_loss.item())

        if epoch % 10 == 9:
            print(f"Epoch: {epoch + 1}, Train Loss: {train_loss_epoch /
len(data_loader):.8f}, Val Loss: {val_loss.item():.8f}")

        scheduler.step()

        if val_loss < best_val_loss:
            best_val_loss = val_loss
            relative_train_loss = train_loss_epoch / len(data_loader)
            best_model_params = net.state_dict()
            early_stop = 0
        else:
            early_stop += 1
            if early_stop == patience:
                print(f"Early stopping reached at epoch {epoch}.")
                print(f"best_val_loss:{best_val_loss:.8f}")
                print(f"relative_train_loss:{relative_train_loss:.8f}")
                break

    plt.plot(train_loss_history, label="Train Loss")
    plt.plot(val_loss_history, label="Val Loss")
```

```
    plt.title("loss")
    plt.xlabel("epoch")
    plt.ylabel("loss")
    plt.legend()
    plt.savefig("loss.jpg")
    plt.show()

    net.load_state_dict(best_model_params, strict=False)

    return net
```

定义`train`函数，用于训练神经网络模型，并返回训练好的模型。

训练过程记录训练集loss和验证集loss，并可视化。

训练循环中使用`Adam`优化器，其参数和学习率为`lr`，并使用`StepLR`学习率调度程序来降低学习率。损失函数的选择是均方误差（`MSE`）。

使用"early stop"，如果验证集损失比之前迭代的最佳损失更低，则当前模型被认为是当前最佳模型，记录它的参数及对应的训练集和验证集MSE。如果连续`patience`个周期没有发现更好的模型，则停止训练。

5. 测试

```
def test(net, test_X, test_Y):
    net.to(device)

    with torch.no_grad():
        output_test = net(test_X)
        test_loss = f.mse_loss(output_test, test_Y)

    print(f"Test Loss: {test_loss.item():.8f}")

    plt.scatter(test_X.cpu(), test_Y.cpu())
    plt.scatter(test_X.cpu(), output_test.cpu().detach())
    plt.title("test")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.legend(['Actual', 'Predicted'])
    plt.savefig("test.jpg")
    plt.show()
```

定义`test`函数，将神经网络模型应用于test_X上的输入数据，并计算其产生的输出与test_Y上的实际输出之间的MSE。输出损失值，并进行可视化，即作出模型预测结果和实际结果的散点图。

四、实验分析

(一) 参数

- train_size：训练集大小
- valid_size：验证集大小
- test_size：测试集大小
- input_size：输入层特征数

- hidden_size：隐藏层神经元数
- output_size：输出层特征数
- epochs：训练轮数
- patience：当连续patience轮迭代都未发现更好的模型时，提前终止训练
- batch_size：每次迭代时，随机选择batch_size个样本，进行梯度更新
- activation_func：激活函数
- optimizer：用于更新模型参数的对象
- lr：学习率

取train_size = 6000，valid_size = 2000，test_size = 2000，input_size=1，output_size=1，epochs=3000，optimizer = optim.Adam(net.parameters(), lr=lr)。

(二）调参

- **patience**

要根据具体的模型做相应的调整。

实验发现当网络深度与宽度变大时，模型比较复杂，需要训练更多轮数来找到最优解，若patience设置得比较小，则在找到全局最优解之前训练就会停止，效果不好；而当学习率变大时，训练不稳定，若patience较小，则容易取到局部最优解之后的很多轮都找不到全局最优解，从而提前停止。

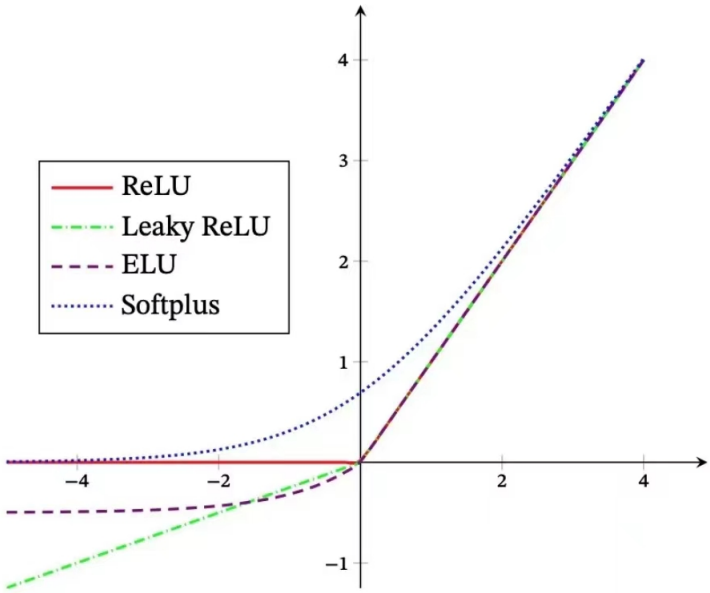
- **激活函数**

前馈神经网络通常使用的激活函数有Sigmoid、Tanh、ReLU及其变种。

Sigmoid和Tanh激活函数在输入较小的情况下可以将输入值压缩到一个较小的范围内，适合于二分类问题和回归问题，但在深度神经网络中容易出现梯度消失问题。

对于本次实验的近似函数问题，我们希望神经网络能够拟合出输入和输出之间的复杂非线性关系。故使用ReLU和其变种。ReLU和其变种激活函数在输入为正数时输出等于输入，可以避免梯度消失问题，适合于深度神经网络中的隐藏层。而且，ReLU函数具有计算速度快和易于优化的特点，可以加速神经网络的训练过程。此外，ReLU的变种函数如Leaky ReLU、ELU和softplus等可以进一步改进ReLU的性能，避免神经元死亡问题和提高拟合能力。

$$
\begin{aligned}
&\text{ReLU}: f(x) = \max(0, x) \\
&\text{Leaky ReLU}: f(x) = \begin{cases} x & x > 0 \\ \alpha x & x \leq 0 \end{cases} \\
&\text{ELU}: f(x) = \begin{cases} x & x > 0 \\ \alpha \left(e^x - 1\right) & x \leq 0 \end{cases} \\
&\text{Softplus}: f(x) = ln(1 + e^x)
\end{aligned}
$$



activation_func	训练集MSE	验证集MSE	epoch
ReLU	0.000396	0.000383	403
Leaky ReLU	0.000068	0.000072	580
ELU	0.000013	0.000022	821
Softplus	0.000002	0.000001	530

实验可知Softplus函数效果较好，可能是因为它比ReLu更平滑，连续可导，使得神经网络可以更好得更新模型。

- **lr**

因为学习率一开始要保持大些保证收敛速度，在收敛到最优点附近时要小些以避免来回振荡。故使用`stepLR`进行学习率衰减。`StepLR`在训练过程中，每经过`step_size`个epoch，将学习率乘以一个给定的因子`gamma`，从而逐渐降低学习率。

初始lr	gamma	训练集MSE	验证集MSE	epoch	patience
0.001	0.5	0.0000464	0.0000476	1764	100
0.003	0.5	0.0000090	0.0000087	1748	100
0.01	0.5	0.0000021	0.0000020	1953	100
0.01	0.1	0.0000058	0.0000056	697	100
0.03	0.5	0.0000028	0.0000027	1999	100
0.03	0.1	0.0000059	0.0000058	816	100
0.1	0.5	0.0000030	0.0000028	2250	100
0.1	0.1	0.0000026	0.0000026	690	100
0.3	0.1	0.0000067	0.0000065	2711	600

当学习率较小时，收敛较慢，适合较大的gamma。当学习率较大时，容易发生震荡，适合较小的gamma。实验可知 lr = 0.01，gamma=0.5 较合适。

- **网络深度**

在一般情况下，神经网络的深度越深，其拟合能力就越强，但也会相应地增加模型的复杂度和计算成本。此外，若深度过深，可能会导致过拟合。

网络深度	训练集MSE	验证集MSE	epoch	patience
2	0.00011500	0.00011800	697	100
3	0.00000210	0.00000200	1953	100
4	0.00000009	0.00000009	1743	100
5	0.00000026	0.00000024	1742	100
6	0.00000018	0.00000014	656	50

由实验可知，4层网络已经足够。

- **网络宽度**

在一般情况下，神经网络的宽度越大，其拟合能力就越强，但也会相应地增加模型的复杂度和计算成本。如果数据集相对较小，并且特征数量较少，则神经网络的宽度窄一些可以更好地捕捉数据的特征。

hidden_size	训练集MSE	验证集MSE	epoch	patience
8	0.00000190	0.00000178	1616	100
16	0.00000009	0.00000009	1743	100
32	0.00000136	0.00000038	655	200
64	0.00000020	0.00000020	2135	100

实验可知，hidden_size=16已经足够。

- **batch_size**

batch_size不影响梯度期望，但会影响梯度方差。batch_size越大，随机梯度的方差越小，引入的噪声也越小，训练也越稳定，因此可以设置较大的学习率。batch_size较小时，需要设置较小的学习率，否则模型会不收敛。

batch_size	训练集MSE	验证集MSE	epoch	lr
32	0.00000014	0.00000011	1792	0.01
64	0.00000009	0.00000009	1743	0.01
64	0.00000013	0.00000012	1552	0.03
128	0.00001209	0.00001224	988	0.01
128	0.00000175	0.00000178	925	0.03
256	0.00004294	0.00004034	502	0.03

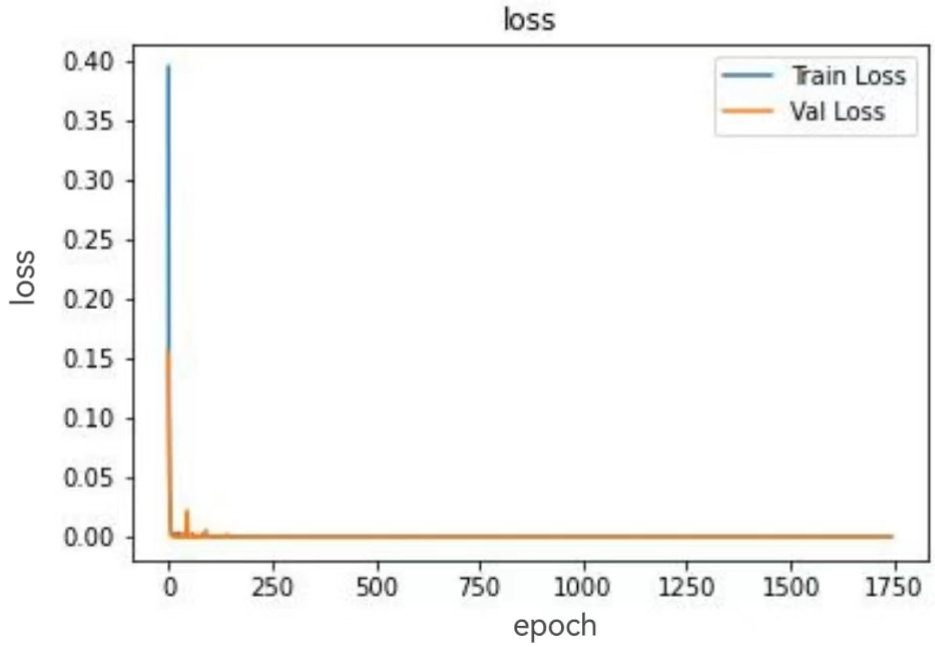
实验可知，batch_size = 64，lr = 0.01较合适。

五、实验结果

- 最合适的一组超参数
  - activation_func：Softplus
  - lr：0.01
  - gamma：0.5
  - patience：100
  - 网络深度 ：4
  - hidden_size：16
  - batch_size：64

- 模型在训练集和验证集上的损失及可视化图
  - loss

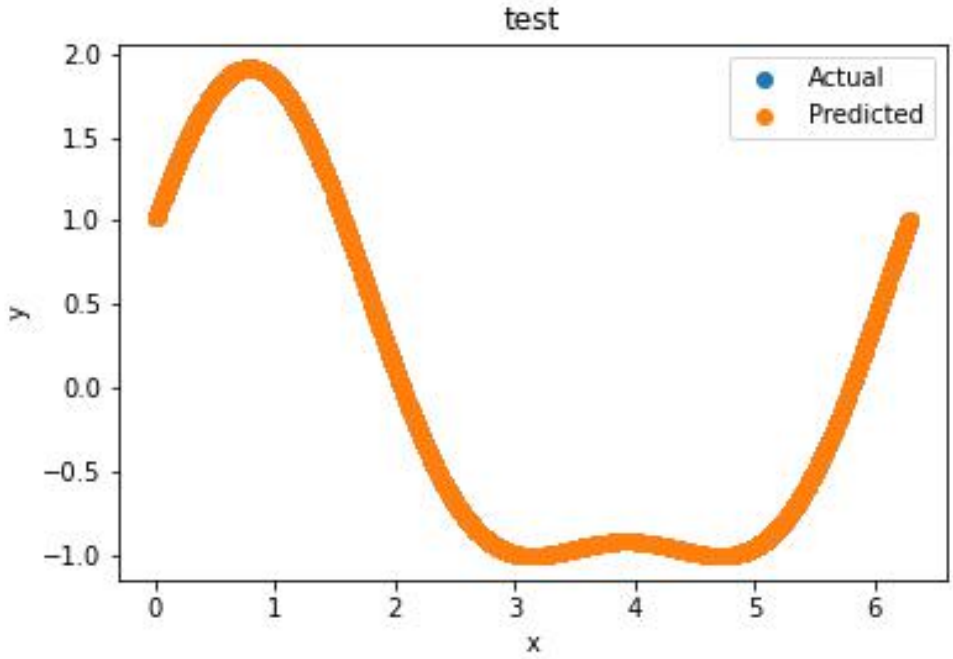
    Early stopping reached at epoch 1743.
    val_loss:0.00000009
    train_loss:0.00000009
  - 可视化



可见train_loss和val_loss在迭代前期有些波动，后期缓慢下降。且二者数值接近，没有发生过拟合或者欠拟合。

- 测试结果
  - 测试集MSE

    test_loss: 0.00000010
  - 测试集预测数据与实际数据的对比图



可见预测结果与实际数据几乎重合，预测效果较好。